2018

Motion Planning For Micro Aerial Vehicles

Sikang Liu
University of Pennsylvania, lskwdlskwd@gmail.com

Motion Planning For Micro Aerial Vehicles

Abstract
A Micro Aerial Vehicle (MAV) is capable of agile motion in 3D making it an ideal platform for developments of planning and control algorithms. For fully autonomous MAV systems, it is essential to plan motions that are both dynamically feasible and collision-free in cluttered environments. Recent work demonstrates precise control of MAVs using time-parameterized trajectories that satisfy feasibility and safety requirements. However, planning such trajectories is non-trivial, especially when considering constraints, such as optimality and completeness. For navigating in unknown environments, the capability for fast re-planning is also critical. Considering all of these requirements, motion planning for MAVs is a challenging problem. In this thesis, we examine trajectory planning algorithms for MAVs and present methodologies that solve a wide range of planning problems. We first introduce path planning and geometric control methods, which produce spatial paths that are inadequate for high speed flight, but can be used to guide trajectory optimization. We then describe optimization-based trajectory planning and demonstrate this method for solving navigation problems in complex 3D environments. When the initial state is not fixed, an optimization-based method is prone to generate sub-optimal trajectories. To address this challenge, we propose a search-based approach using motion primitives to plan resolution complete and sub-optimal trajectories. This algorithm can also be used to solve planning problems with constraints such as motion uncertainty, limited field-of-view and moving obstacles. The proposed methods can run in real time and are applicable for real-world autonomous navigation, even with limited on-board computational resources. This thesis includes a carefully analysis of the strengths and weaknesses of our planning paradigm and algorithms, and demonstration of their performance through simulation and experiments.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Mechanical Engineering & Applied Mechanics

First Advisor
Vijay Kumar

Keywords
Collision Avoidance, MAV, Motion Planning, Optimal Control, Quadrotor

Subject Categories
Engineering | Robotics

MOTION PLANNING FOR MICRO AERIAL VEHICLES

Sikang Liu

A DISSERTATION

in

Mechanical Engineering and Applied Mechanics

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Dedicated to my parents

Acknowledgments

Looking back on the long journey at Penn, I've seen, experienced and learned way more than what I could expect. I feel grateful to my advisor professor Vijay Kumar, who has given me such a great opportunity to work in his lab. Through these many years of working and studying under his supervision, I've grown up to be a responsible and honest man. I would also want to thank my thesis committee: Ani Hsieh, CJ Taylor and Nikolay Atanasov, for spending time and efforts on my graduation. CJ has also been an excellent leader in the FLA team, in which we worked together through many challenges. Nikolay has been a wonderful collaborator in many of my search-based planning works, his advice is always wise and helpful.

I also want to thank my lab mates in the MRSL. Since the first time I joined in 2011, Kartik Mohta has been there and helped me doing experiments. I've learned a lot from him and also many others in the lab. I will remember the days of hard working in the lab, it is a precious memory in my life.

I want to thank my parents Liping Zhou and Haiyang Liu, they are patient and supportive through my whole life. I will not successfully graduate without them. I'd also like to thank my girlfriend Xiaodi Wang for being company with me during the hard time when I was working on my thesis.

In the last, I'd like to give special thanks to Kate Tolstaya and Laura Jarin-Lipschitz who spent time reading my thesis and helped me with editorial comments.

ABSTRACT

MOTION PLANNING FOR MICRO AERIAL VEHICLES

Sikang Liu

Vijay Kumar

A Micro Aerial Vehicle (MAV) is capable of agile motion in 3D making it an ideal platform for developments of planning and control algorithms. For fully autonomous MAV systems, it is essential to plan motions that are both *dynamically feasible* and *collision-free* in cluttered environments. Recent work demonstrates precise control of MAVs using time-parameterized trajectories that satisfy feasibility and safety requirements. However, planning such trajectories is non-trivial, especially when considering constraints, such as *optimality* and *completeness*. For navigating in unknown environments, the capability for fast re-planning is also critical. Considering all of these requirements, motion planning for MAVs is a challenging problem. In this thesis, we examine trajectory planning algorithms for MAVs and present methodologies that solve a wide range of planning problems. We first introduce path planning and geometric control methods, which produce spatial paths that are inadequate for high speed flight, but can be used to guide trajectory optimization. We then describe optimization-based trajectory planning and demonstrate this method for solving navigation problems in complex 3D environments. When the initial state is not fixed, an optimization-based method is prone to generate sub-optimal trajectories. To address this challenge, we propose a search-based approach using motion primitives to plan resolution complete and sub-optimal trajectories. This algorithm can also be used to solve planning problems with constraints such as motion uncertainty, limited *field-of-view* and moving obstacles. The proposed methods can run in real time and are applicable for real-world autonomous navigation, even with limited on-board computational resources. This thesis includes a carefully analysis of the strengths and weaknesses of our planning paradigm and algorithms, and demonstration of their performance through simulation and experiments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Micro Aerial Vehicles (MAVs) are small multi-rotor helicopters that are able to vertically take-off and land (VTOL) and freely fly in complex 3D environments. Compared to fixed-wing aircrafts and traditional helicopters, multi-rotor MAVs have higher system robustness and are easier to control due to their simple mechanical structure. Additionally, an MAV's powerful rotors and relatively light body weight provide a high *thrust-to-weight* ratio such that the MAV is able to achieve high acceleration and move agilely. Through recent break-throughs in technology, computers and sensors have become smaller and lighter such that an MAV is now able to carry sufficient computational and sensing resources to complete complicated tasks. All of these factors make the multi-rotor MAV an ideal platform for both research and practical applications. Existing commercial drones are well-known for video streaming and photo shooting. However, there is more potential for drones than just digital entertainment. For example, autonomous MAVs are extremely useful in applications such as exploration, search and rescue, target tracking, cargo delivery, and many others [8, 10, 52, 64, 66, 92]. Thus, exploration and advances in the field of MAVs could impact our community in many positive ways in the future.

As a fundamental component of an autonomous MAV system, motion planning has attracted much more attention in the past decade. A key part of motion planning is navigating a robot from one place to another. Considering system dynamics and collision avoidance, it is challenging to plan motion of an MAV. In this dissertation, we focus on this problem and discuss the challenges and solutions for many practical planning problems.

1.1 Related Work

In general, motion planning methods can be categorized into two classes according to the optimality of the planning results: (1) randomized approaches such as PRM [23, 38] and RRT, [36, 41] and (2) deterministic approaches such as graph search algorithms [39, 42, 48]. Randomized approaches aim to find feasible solutions while deterministic methods solve for optimal solutions. In many applications, including robotic manipulation and non-holonomic systems, it is almost impossible to find the optimal result in high dimensional spaces within a reasonable time. For those scenarios where calculating the optimal solution is computationally intractable, the randomized approach is more popular. However, the deterministic approach is always preferable if computation time allows, since it can grant optimality of minimum distance, control effort, traveling time, and also prevents unexpected motions.

The goal of motion planning is to find control sequences for the robot to follow that are *dynamically feasible* and *collision-free*. Traditional path planning techniques are not sufficient to satisfy the requirement of feasibility, since the path generated relies only on spatial information, and cannot actually be performed due to the dynamics of the robot. Consequently, an additional path following layer [31, 70, 78, 90] is needed to extract control commands from the geometric path. There are many drawbacks of this two-step process, including the lack of optimality and decreased safety and agility. Optimal control can instead provide much better trajectories. A trajectory is a time parameterized function of robot states that contains both spatial and temporal information. Trajectories recover the full states of a dynamic system such that they are adequate to guarantee the system's feasibility and safety. Since the system dynamics and time-varying states are taken into account, trajectory generation is more complicated than path planning. Traditionally, we try to solve for trajectories with certain optimality criteria, *e.g.,* using minimum time [7, 101] or minimum efforts [61, 68]. In these works, trajectory generation is formulated as an optimization problem. In has been shown in [61, 68, 82], a minimum jerk/snap trajectory can be solved for from an unconstrained quadratic optimization problem (QP). To account for dynamical and collision constraints, additional inequality constraints are necessary. To plan for collision-free trajectories, [61, 82] intermediate waypoints are used, which can not guarantee the safety. The mixed integer programming (MIP) approach proposed in [13, 15, 62] guarantee trajectories' safety, but their integer constraints are hard to create, and solving a MIP is usually time-consuming. Other approaches try to overcome the problem of computational time with short trajectories [56, 71, 96, 105], but these local planners are not suitable to solve planning problems in complicated environments due to their lack of optimality and completeness. As a contrast, a search-based method solves similar optimization problems based on motion primitives [16, 24, 47]. Due to the 'curse of dimensionality', the search-based method is considered to be computationally inefficient and intractable for dynamic systems such as MAVs. As a compromise, planning with kinodynamics primitives have been used for MAVs in [59, 75, 76]. A properly tuned primitive library and discretized state space are critical to obtain successful plans. The search process is extremely slow when higher order dynamics are considered, such that the existing related works are limited only to kinematic systems. In this thesis, we will present more detailed discussions of this and other related work in individual chapters.

Throughout this thesis, we analyze the performance of a planner with respect to theoretical and practical challenges by addressing the following five properties:

1. *Feasibility*: whether the planning results are executable or not for MAVs?
2. *Safety*: whether the planning results are collision-free or not?
3. *Optimality*: whether the planning result is optimal or sub-optimal?
4. *Completeness*: if there exists a solution, whether the planner is able to find it?
5. *Run time*: how much time it takes to find the planning result?

For example, we use the following table to show the evaluation of the described *state-of-the-art* techniques:

Method	***Feasibility***	***Safety***	***Optimality***	***Completeness***	***Run time***
Path Planning	Not feasible	Collision free	Globally optimal	Resolution complete	Very fast
Unconstrained QP [61, 82]	May violate constraints	Not guaranteed	Sub-optimal	Not complete	Very fast
MIP [15, 62]	Dynamically feasible	Collision free	Sub-optimal	Complete	Very slow
Short Trajectories [56, 71]	Dynamically feasible	Collision free	Locally optimal	Not complete	Very fast

Table 1.1: Evaluation of *state-of-the-art* techniques using the five properties. Red blocks indicate the drawbacks of the corresponding algorithm.

In general cases, the planner that conditionally satisfies the proposed requirements is acceptable for solving certain planning problems. For example, path planning quickly finds paths that are collision-free. Even through the path is not dynamically feasible, we can still use it to guide a mobile robot to reach goals with a certain control strategy, at relatively low speeds. For fast and agile flight of MAVs however, dynamically feasible trajectories are required. Therefore, the path planner is not suitable for navigating MAVs at high speed.

We are interested in designing an ultimate planner that satisfies all five requirements and generates trajectories that are *dynamically feasible, collision-free, globally optimal and complete* in *real-time*. In this thesis, we will show how difficult it is to propose such an ultimate planner and propose our approaches that are near-ultimate, which are suitable to solve real-world motion planning problems for MAVs.

1.2 Motivation and Outline

In this thesis, we mainly consider the planning problems in three scenarios: (1) a known static map, (2) an unknown static map and (3) a dynamic map. Planning in a known static map is the prerequisite for the other two scenarios. For planning in unknown environments, we use the receding horizon planning framework that is similar to [54, 85, 105] to re-plan new trajectories that avoid newly detected obstacles. In dynamic environments, we need to consider the time as a state variable in order to predict collisions of future trajectories.

We consider multi-robot planning as a special case of planning in dynamic environments, in which each robot is treated as an obstacle that follows a trajectory. This work is largely inspired by the DARPA Fast Light-weight Autonomy (FLA) project [1] in which a fully autonomous MAV is developed to navigate in unknown obstacle-cluttered environments. [65, 67, 91] introduces more details about the FLA project. In this thesis, we are specifically interested in designing planners for the MAV to explore unknown environments and re-plan new trajectories.

The thesis follows the flow of development of motion planning algorithms for solving proposed planning problems in three scenarios. In Chapter 2, we briefly introduce background information about the MAV's dynamics and control, path planning and path following algorithms that are widely used to navigate mobile robots. In Chapter 3, we explain the optimization-based trajectory generation and its applications in both 2D and 3D obstacle-cluttered environments. In Chapter 4, we propose our search-based planner that uses motion primitives for solving deterministic shortest trajectory problems. Chapter 5 is a complimentary chapter that shows extensions of the proposed search-based planner for real world planning problems where sensors have limited range and field-of-view (FOV), the robot has motion uncertainty, and the attitude constraints need to be taken into account to fly through narrow gaps. In Chapter 6, we show another novel extension of the search-based planner for solving the planning problem in dynamically changing environments. Our planner is able to plan collision-free trajectories for dynamically moving obstacles, such that we are able to further apply this technique for a team of robots. We demonstrate and analyze the performance of our planner for navigating both centralized and decentralized multi-robot systems. Chapter 7 is a conclusion chapter and proposes future work. A list of Open-sourced repositories and relevant publications can be found in Appendix A.

Chapter 2

Preliminary

In this chapter, we introduce *state-of-the-art* techniques for geometric control and path planning. Section 2.1 introduces the motion and control model for a simple MAV. Section 2.2 describes path planning based on Dijkstra algorithm. Section 2.3 illustrates the path following and receding horizon control framework for navigation in unknown environments.

2.1 Quadrotor Model and Geometric Control

The MAV platform, in particular, the quadrotor, is a dynamical system with six *degree-of-freedom* (DOF), including both position and orientation in $\mathbb{R}^3$. In this thesis, we use the notation of $SE(3)$ to address the 3D rigid transformation, which combines both translation in $\mathbb{R}^3$ and rotation in $SO(3)$. The $SO(3)$ and $SE(3)$ groups and their associated Lie algebra are commonly used in computer vision and robotics to represent spatial transformation.

2.1.1 Model

Following [61], we define the state of an MAV as the position and velocity of its center of mass and orientation of Euler angles in the world frame:

$$\mathbf{x} := [\ x,\ y,\ z,\ \phi,\ \theta,\ \psi,\ \dot{x},\ \dot{y},\ \dot{z},\ p,\ q,\ r\ ]^{\mathsf{T}}. \tag{2.1}$$

The rotation matrix from the body frame to the world frame $\mathbf{R}_{\mathrm{B}}^{\mathrm{W}}$ can be derived from roll ϕ, pitch θ and yaw ψ angles. The control input to the quadrotor system is denoted as $\mathbf{u} = [\, u_1,\ u_2,\ u_3,\ u_4 \,]^{\mathsf{T}}$ which contains four components corresponding to thrust and torques in the body frame. To be specific, given L is the arm length, the control input $\mathbf{u}$ for the system in Figure 2.1 can be expressed as:

$$\begin{aligned} u_1 &= F_1 + F_2 + F_3 + F_4, \\ u_2 &= F_2 L - F_4 L, \\ u_3 &= -F_1 L + F_3 L, \\ u_4 &= M_1 - M_2 + M_3 - M_4. \end{aligned} \tag{2.2}$$

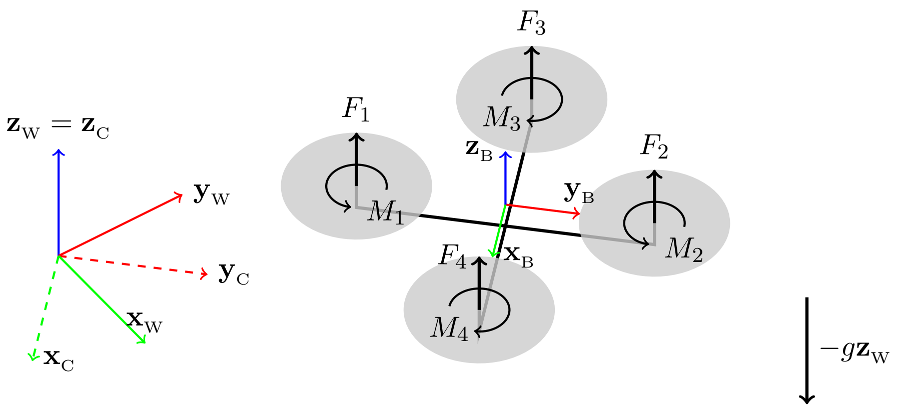


Figure 2.1: A simple quadrotor model.

The presented model is the simplest quadrotor model that is symmetric and rigid. In the following sections, we assume that the general MAV platform has a similar configuration. Please note that in Figure 2.2, the generated force from the motors is perpendicular to the robot body's x-y plane. Denote $\mathbf{a}$ as the quadrotor's acceleration in the world frame and m as the quadrotor's mass. The mass-normalized equation of the dynamics is

$$\mathbf{a} = -g\mathbf{z}_\mathrm{W} + \frac{u_1}{m}\mathbf{z}_\mathrm{B}. \tag{2.3}$$

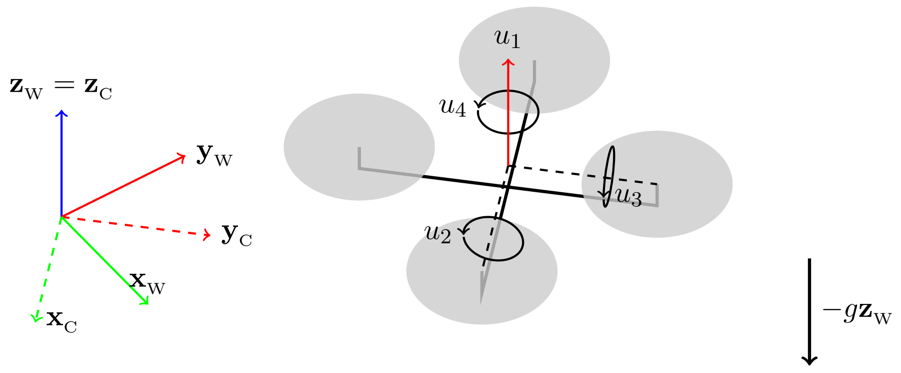


Figure 2.2: Control input $\mathbf{u} = [\ u_1,\ u_2,\ u_3,\ u_4\ ]^\mathsf{T}$ of a simple quadrotor model.

2.1.2 Differential Flatness

It has been shown in [61] that the quadrotor dynamics are *differentially flat* with the following flat outputs:

$$\sigma = [\ x,\ y,\ z,\ \psi\ ]^\mathsf{T}. \tag{2.4}$$

This means that the system state $\mathbf{x}$ and control input $\mathbf{u}$ can be represented in terms of σ and its derivatives. The translation part of the system state can be easily derived from the first three elements of σ. To see that the rotation $\mathbf{R}_B^W$ is a function of σ, we consider the previous dynamics equation (2.3). The body z-axis $\mathbf{z}_B$ is parallel to the thrust vector, or *i.e.,* acceleration plus gravity, such that

$$\mathbf{z}_B = \frac{\mathbf{a} + g\mathbf{z}_W}{\|\mathbf{a} + g\mathbf{z}_W\|}. \tag{2.5}$$

The acceleration $\mathbf{a}$ is the second derivative of the first three elements in σ as $\mathbf{a} = [\ \ddot{x},\ \ddot{y},\ \ddot{z}\ ]^\mathsf{T}$. Thus, the body z-axis $\mathbf{z}_B$ is a function of σ.

The body's x-axis and y-axis can be derived using an intermediate virtual vector described in [44, 61]. The intermediate vector is a function of the robot's yaw ψ, the fourth element of σ, as

$$\mathbf{x}_C = [\ \cos\psi,\ \sin\psi,\ 0\ ]^\mathsf{T}. \tag{2.6}$$

Thus, the body's axes, $\mathbf{x}_{\mathrm{B}}$ and $\mathbf{y}_{\mathrm{B}}$, can be determined as

$$\mathbf{x}_{\mathrm{B}} = \mathbf{y}_{\mathrm{B}} \times \mathbf{z}_{\mathrm{B}}, \ \mathbf{y}_{\mathrm{B}} = \frac{\mathbf{z}_{\mathrm{B}} \times \mathbf{x}_{\mathrm{C}}}{\|\mathbf{z}_{\mathrm{B}} \times \mathbf{x}_{\mathrm{C}}\|}. \tag{2.7}$$

Since the rotation matrix is constructed as $\mathbf{R}_{\mathrm{B}}^{\mathrm{W}} = [\ \mathbf{x}_{\mathrm{B}} \mid \mathbf{y}_{\mathrm{B}} \mid \mathbf{z}_{\mathrm{B}}\ ]$, we conclude that the rotation part of the system state is also a function of σ. Thus, as the position and orientation are both functions of flat outputs σ, their derivatives are also functions of σ and its derivatives. In fact, the system's full state $\mathbf{x}$ is non-linear and is usually hard to directly compute. This property of the *differential flatness* is very important in motion planning and control as it implies that $\mathbf{x}$ can be represented by the flat outputs σ.

2.1.3 Geometric Control

A controller maps the robot's current and desired states to the robot's control input. The robot's current state $\mathbf{x}(t)$ at time t can be found using state estimation. The corresponding desired state $\mathbf{x}_d(t)$ can be extracted from a specified trajectory at time t [44, 60]. Here we denote $\mathbf{x}_d$ as the flat outputs $[\ x_d,\ y_d,\ z_d,\ \psi_d\ ]^{\mathsf{T}}$. According to Section 2.1.2, the true system state in form of (2.1) can be expressed by the flat outputs. Denote the translation part of the robot's state as $\mathbf{r} = [\ x,\ y,\ z\ ]^{\mathsf{T}}$, then the desired force $\mathbf{f}_d$ can be computed as

$$\mathbf{f}_d = k_p(\mathbf{r}_d - \mathbf{r}) + k_d(\dot{\mathbf{r}}_d - \dot{\mathbf{r}}) + m\ddot{\mathbf{r}}_d + mg\mathbf{z}_{\mathrm{W}}, \tag{2.8}$$

where $k_p,\ k_d$ are constant control gains.

According to (2.3) and (2.5), the desired thrust u_1 and desired body z-axis $\mathbf{z}_{\mathrm{B,d}}$ can be expressed as

$$u_1 = \mathbf{f}_d \cdot \mathbf{z}_{\mathrm{B}}, \ \mathbf{z}_{\mathrm{B,d}} = \frac{\mathbf{f}_d}{\|\mathbf{f}_d\|}. \tag{2.9}$$

Therefore, the rest of the desired rotation $\mathbf{R}_{\mathrm{B,d}}^{\mathrm{W}}$ can be derived from (2.7) with $\mathbf{x}_{\mathrm{C,d}} = [\ \cos\psi_d,\quad \sin\psi_d,\ 0\ ]^{\mathsf{T}}$:

$$\mathbf{x}_{\mathrm{B,d}} = \mathbf{y}_{\mathrm{B,d}} \times \mathbf{z}_{\mathrm{B,d}}, \ \mathbf{y}_{\mathrm{B,d}} = \frac{\mathbf{z}_{\mathrm{B,d}} \times \mathbf{x}_{\mathrm{C,d}}}{\|\mathbf{z}_{\mathrm{B,d}} \times \mathbf{x}_{\mathrm{C,d}}\|}. \tag{2.10}$$

The desired torques can be calculated using the tracking error for orientation and angular velocity [44]. Since it is a solved problem and not the research focus of our work, we omit the derivation of the low level attitude controller. The important insight from the controller design is that using the flat outputs, we are able to control the whole quadrotor system using geometric control with asymptotic stability.

2.2 Path Planning

In the well-established problem of path planning, the goal is to find a sequence of collision-free waypoints connecting start and goal in a given configuration space. In this section, we review the basic search-based algorithms for path planning using Dijkstra and A* search algorithm. In addition, we introduce the *Jump Point Search* [27] and propose its 3D variation in Section 2.2.3, which is an efficient path planner that we used in many of our projects. The paths from these planners are normally close to obstacles, which are potentially dangerous for the robot to track due to uncertainty in its motion. To address this problem, we propose a simple but effective technique based on *artificial potential fields* (APFs) to find a safer path given a nominal shortest path in Section 2.2.4. The raw path from the planner must then be smoothed into piece-wise linear segments. This simplification is explained in the last section of this chapter.

2.2.1 Problem Formulation

A path in $\mathbb{R}^m$ is defined as a sequence of waypoints $P := \langle\, \mathbf{p}_0 \rightarrow \mathbf{p}_1 \rightarrow \ldots \rightarrow \mathbf{p}_N \,\rangle$ with the waypoint $\mathbf{p}_i \in \mathbb{R}^n$. In fact, path planning is an optimization problem with a cost function $J(\cdot)$ and constraints $F(\cdot)$, which can be formulated as:

$$\begin{aligned} \arg\min_{P} \;\; & J(P) \\ s.t \;\; & F(P). \end{aligned} \tag{2.11}$$

The objective of a typical path planning problem is to find the shortest path, whose cost is equal to the total distance. In addition, such problems have collision constraints on the start and goal states. Hence, given a configuration space $\mathcal{X} \in \mathbb{R}^m$ which contains the collision-free subspace $\mathcal{X}^{free}$, the constraints $F(P)$ for finding a path from the start $\mathbf{p}_s$ to goal $\mathbf{p}_g$ can be written as

$$\mathbf{p}_s,\ \mathbf{p}_g \in P,\ P \in \mathcal{X}^{free}. \tag{2.12}$$

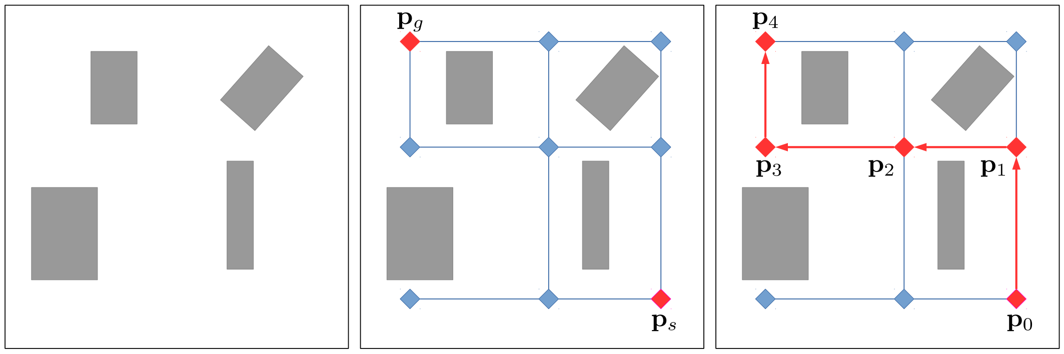
There are many graph search algorithms that solve a deterministic shortest path problem from one vertex to another. In robotic navigation, the planning problem can be converted into a graph search problem, in which the objective is to find a path connecting the start and goal in the configuration space. If the configuration space is represented as a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges connecting vertices in $\mathcal{V}$, the deterministic shortest path problem can be expressed as

$$\begin{aligned} \arg\min_{P} \sum_{i=0}^{N-1} \|\mathbf{p}_{i+1} - \mathbf{p}_i\|_p \\ s.t\ \mathbf{p}_0 = \mathbf{p}_s,\ \mathbf{p}_N = \mathbf{p}_g, \\ \mathbf{p}_i \in \mathcal{V},\ i = 0, \dots N, \\ e(\mathbf{p}_i, \mathbf{p}_{i+1}) \in \mathcal{E},\ i = 0 \dots N-1. \end{aligned} \tag{2.13}$$

where $\|\cdot\|_p$ indicates the *p-norm*. When planning in the Euclidean space, L-2 norm is often used to determine the Euclidean distance. The function $e(\cdot,\cdot)$ indicates the directional edge from one state to the other, the last constraint in (2.13) ensures that any waypoint in P connects to its neighbors. To find a collision-free path, we only consider the graph in the collision-free space such that $\mathcal{G}(\mathcal{V}, \mathcal{E}) \in \mathcal{X}^{free}$ (Figure 2.3).

2.2.2 Graph Search Algorithms

We use graph search algorithms based on Dijkstra algorithm to solve the problem in (2.13). Dijkstra algorithm is a dynamic programming (DP) approach that deterministically finds the optimal solution. A* is the informed Dijkstra, which biases the search using a *heuristic function*. The heuristic function is a preliminary estimation of the *cost-to-goal* value. The use of the heuristic can potentially reduce the number of vertices to be expanded during the A* search as compared to the uninformed Dijkstra algorithm search. It is a critical evaluation of the output's optimality and computational efficiency. To enable these improvements, the



(a) Configuration space. (b) Graph in free space. (c) Path from start to goal.

Figure 2.3: An example of the configuration space and the corresponding graph. The graph consists of vertices and edges in free space. The path connects the start $\mathbf{p}_s$ and the goal $\mathbf{p}_g$.

heuristic must have the following two properties:

Admissible The heuristic function never overestimates the minimal *cost-to-goal*.

Consistent(monotonic) Let $c(\cdot, \cdot)$ be the edge cost between two vertices, $h(\cdot)$ denotes the heuristic value, h satisfies

$$h(u_i) \leq c(u_i, u_j) + h(u_j), \text{ and } h(u_g) = 0, \tag{2.14}$$

where u_g is the goal.

The *admissible* heuristic guarantees the optimality of the search result. A *consistent* heuristic is also *admissible*. In general, the closer a heuristic is towards the true *cost-to-goal* value, the faster the algorithm finds the optimal result since it tends to expand fewer vertices.

Notation

We use following notations in the pseudo-code. For each vertex in the graph, namely $u \in \mathcal{V}$, we define following attributes:

$g(u)$	total cost to reach u from the start state
$h(u)$	heuristic value of u
$\text{Succ}(u)$	one-step successors
$\text{Pred}(u)$	one-step predecessors

A priority queue $\mathcal{Q}$ is used to store the open set of $\mathcal{V}$. The following functions are used to manage $\mathcal{Q}$:

Top()	return the state with the smallest priority
Pop()	delete the state with the smallest priority
Remove(u)	remove the state u from $\mathcal{Q}$
Insert(u, f)	insert the state u with priority f

Pseudo-code

The pseudo-code for Dijkstra and A* algorithms is illustrated in Algorithm 1. In the *plan* procedure, we first initialize the state space $\mathcal{V}$ by setting the *start-to-state* value for all the vertices to infinity. The start state u_s is assigned with an initial g-value (we select 0, but it takes arbitrary finite value) and is pushed onto the priority queue, which is initially empty. The loop in lines 16 to 32 shows the key steps of the algorithm. The list of predecessors for every vertex that has been expanded is maintained, so that the optimal path can be recovered from the final state by recursively selecting the best predecessors. To avoid internal loops, the edge cost $c(u, s)$ must be positive.

ϵ is the weight of heuristic and can be adjusted for different purposes. For example, Dijkstra algorithm is a special case of A* with $\epsilon = 0$. In some cases, set $\epsilon > 1$ is able to achieve fast computation, even though the heuristic value becomes *inadmissible* [48, 108].

Graph Representation

As shown in Figure 2.3, the graph is a representation of the configuration space. A configuration space, or *C-space*, is the space of all configurations of the robot. For a robot with certain geometry in $\mathbb{R}^m$, it is represented as a single dot in the corresponding *C-space*. The actual world where the robot is moving is called the *workspace*, in which the collision

Algorithm 1 Dijkstra's and A* algorithms. Given the start and goal vertices u_s, u_g, the algorithm finds the optimal path for problem (2.13). The heuristic weight ϵ is equal to 0 for Dijkstra and 1 for A*.

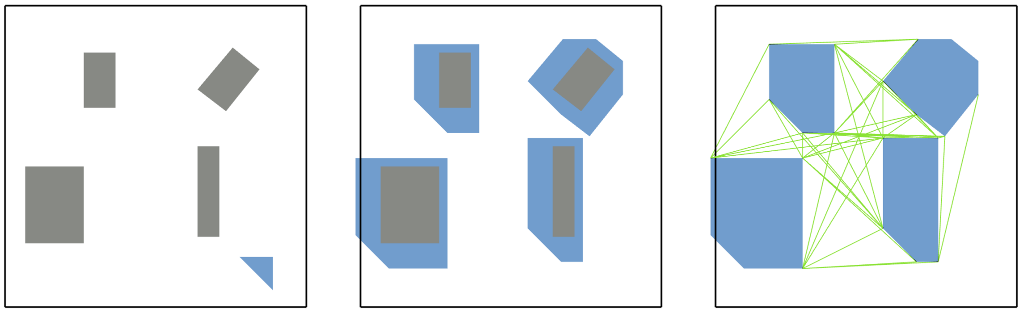
```
 1: function RecoverPath($u$)
 2:     $P \leftarrow \emptyset$;
 3:     while $\text{Pred}(u) \neq \emptyset$ do
 4:         $p \leftarrow \arg\min_{p' \in \text{Pred}(u)} (g(p') + c(p', u))$;
 5:         $P \leftarrow \langle p,\ P \rangle$;
 6:         $u \leftarrow p$;
 7:     end while
 8:     return $P$;
 9: end function
```

```
10: procedure Plan($u_s, u_g, \epsilon$)
11:     for all $u \in \mathcal{V}$ do
12:         $g(u) \leftarrow \infty$;
13:     end for
14:     $g(u_s) \leftarrow 0,\ \mathcal{Q}.\text{Insert}(u_s, 0)$;
15:
16:     while $\mathcal{Q} \neq \emptyset$ do
17:         $u \leftarrow \mathcal{Q}.\text{Top}()$;
18:         $\mathcal{Q}.\text{Pop}()$;
19:         for all $s \in \text{Succ}(u)$ do
20:             if $u \notin \text{Pred}(s)$ then
21:                 $\text{Pred}(s) \leftarrow \{u\} \cup \text{Pred}(s)$;
22:             end if
23:             $g_{\text{tmp}} \leftarrow g(u) + c(u, s)$;
24:             if $g_{\text{tmp}} < g(s)$ then
25:                 $g(s) \leftarrow g_{\text{tmp}}$;
26:                 $\mathcal{Q}.\text{Insert}(s, g(s) + \epsilon \cdot h(s))$;
27:             end if
28:         end for
29:         if $u = u_g$ then
30:             return RecoverPath($u$);
31:         end if
32:     end while
33:     return Failure;
34: end procedure
```

checking between robot's geometry and obstacles can be non-intuitive. Therefore, we plan in *C-space* to acquire the optimal and safe path in the workspace. It is straightforward to convert a workspace in $\mathbb{R}^m$ into the corresponding *C-space* in $\mathbb{R}^m$ through Minkowski dilation which is the sum of two sets of position vectors:

$$A \oplus B = \{a + b \mid a \in A,\ b \in B\}, \quad A, B \subset \mathbb{R}^m. \tag{2.15}$$

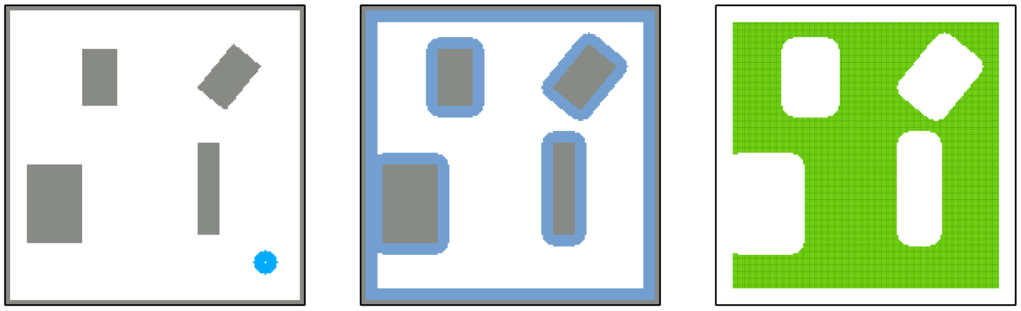
An example in $\mathbb{R}^2$ is shown in Figure 2.4, in which the robot and obstacles are represented as polygons. In Figure 2.4b, the robot is collision-free anywhere in the white space within the bounding box. The visibility graph [57] is one technique where a graph connects start and goal is constructed in a polygonal map and a shortest path can be found using Algorithm 1.



(a) Workspace. (b) Configuration space. (c) Visibility graph.

Figure 2.4: An example of the workspace and the configuration space. As shown in (a), the geometry of the robot is a triangle in 2D. In (b), we dilate the obstacle using Minkowski addition to get the corresponding configurationn space. The visibility graph in (c) can be easily computed based on polygonal map in (b).

Another example is shown in Figure 2.5, in which the robot is treated as a disk and the map is represented as an occupancy grid. An occupancy grid can be constructed from real sensor data, such as scans from laser range finder or depth point clouds from stereo or RGB-D cameras. Compared to the polygonal map in Figure 2.4a, the map in Figure 2.5a is easier to acquire for the real robot. In addition, the grid implicitly indicates a graph in which each cell is connected to its neighboring cells such that we do not need another mechanism like visibility graph. Also, it is straightforward to convert the occupancy grid into a voxel grid in $\mathbb{R}^3$. Thus, the occupancy grid map is often used in robotics for path planning in both 2D and 3D environments.



(a) Workspace. (b) Configuration space. (c) Free cells.

Figure 2.5: An example of the workspace and the configuration space. As shown in (a), the geometry of the robot is a disk. In (b), we dilate the obstacle using Minkowski addition to get the corresponding configurationn space. The free space is consist of cells colored in green in (c).

The map and the graph are two independent representations of the configuration space. Thus, in order to to apply Algorithm 1, a graph in Figure 2.4c or 2.5c is the prerequisite.

Deterministic Shortest Path

Once the graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is given, we are able to apply Algorithm 1 to find the shortest path. To plan using the visibility graph in Figure 2.4c, we first connect the start and goal to the graph. The planned path in the *C-space* is also safe in the original workspace in Figure 2.6

In the occupancy grid map, we can find the corresponding cells for start and goal coordinates and further search the shortest path between those two cells using Algorithm 1. Figure 2.7b indicates that the output path guarantees the safety in work space, thus planning in the *C-space* is equivalent to planning in the original work space.

So far, we have introduced the A* graph search algorithm and described its application to find the shortest path in a graph. In the following paragraphs, we discuss several performance-enhancing variations.

2.2.3 Jump Point Search

If the graph consists of a uniform-cost grid (*e.g.*, occupancy grid), Dijkstra and A* search can be optimized through a technique called *Jump Point Search* (JPS) [27]. Using JPS results

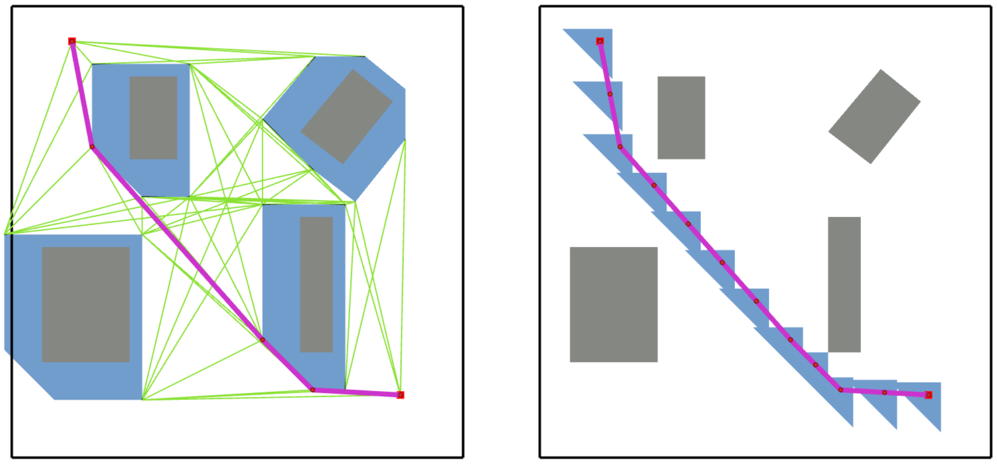

(a) Shortest path. (b) Robot follows path.

Figure 2.6: By connecting start and goal states in the graph, we find the shortest path (highlighted in magenta) through Algorithm 1 for a triangle-shaped robot. (b) shows that the robot's planned path is collision-free in the original workspace.

in a large reduction in computational time for the problem of finding the shortest path. This is achieved through pruning the neighbors of the vertex being searched. Therefore, the total number of expanded vertices during JPS is much smaller than during the standard A* search. In Figure 2.8, the number of expanded cells is 244 for JPS and 6658 for A* while the computational time is 7 ms for JPS and 98 ms for A*. The pruning process in JPS significantly reduces the number of cells that must be expanded, but it can be extremely complicated to check in $\mathbb{R}^m$ where $m > 2$. Thus, JPS has only been used widely in $\mathbb{R}^2$ and it is hard to adapt for graphs in higher dimensional spaces.

In order to use JPS with 3D voxel maps, we extend the 2D algorithm with different pruning rules for 3D voxel grids as presented in Figure 2.9 and 2.10. As defined in [27], the *natural neighbors* refer to the set of vertices that remain after pruning. For those neighbors which cannot be pruned due to obstacles, we call them *forced neighbors*.

The details of the recursive pruning and jump processes can be found in [27]. The proposed pruning in Figure 2.10 is a compromise between checking all the situations and maintaining the simplicity of the algorithm: we add more neighbors than required (three *forced neighbors* case) but it is easier to check (*i.e.,* more efficient). JPS provides the

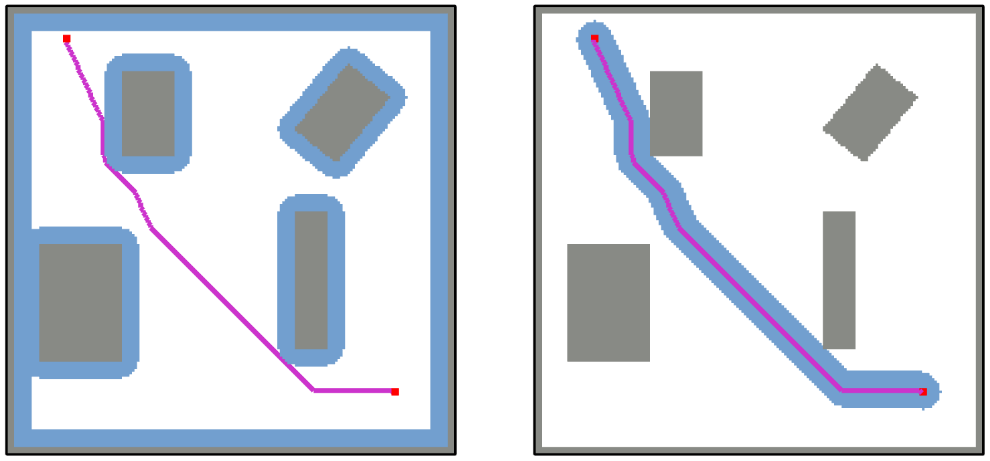

(a) Shortest path. (b) Robot follows path.

Figure 2.7: By connecting start and goal states in the graph, we find the shortest path (highlighted in magenta) through Algorithm 1 for a disk-shaped robot. (b) shows that the robot's planned path is collision-free in the original workspace.

same completeness and optimality guarantees as A* [27], with the only limitation being the assumption of uniform-cost grid, which holds for our case. Our 3D JPS significantly speeds up the run time of planning (100 times or more in obstacle-sparse maps).

2.2.4 Path Perturbation

The path resulting from A* or JPS is normally close to obstacles in order to be "shortest" (*e.g.,* Figure 2.8). In the real world, when the robot gets close to obstacles, the control error due to disturbances and state estimation drifts could easily result in a crash into nearby obstacles. Hence, the resulting shortest paths are hard to follow due to their proximity to obstacles. A common solution for real world navigation is to plan in the *C-space* with over-inflated obstacles. Over-inflating can easily block small gaps such as doors, windows and narrow corridors. Thus, this solution is not complete, especially in an obstacle-cluttered indoor environment.

We propose a *path perturbation* technique to sacrifice path length for safety, resulting in increased clearance of obstacles, while avoiding unnecessary detours [106]. We model the collision cost using an artificial potential field (APF). The traditional APF method

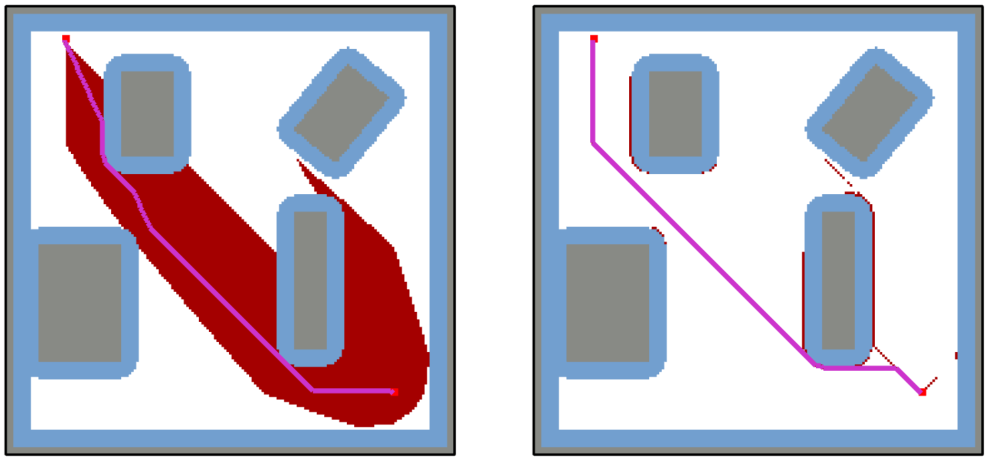

(a) A*. (b) JPS.

Figure 2.8: The search for the shortest path using A* (left) and JPS (right). The output paths have the same lengths. Cells in red show the expanded vertices in the close-set.

in [2, 19, 25, 46, 104] only works for simple convex obstacles and is prone to get stuck in local minima, which makes it hard to use in complex environments. In our proposed method, we perturb a nominal shortest path into an optimal path using a graph search algorithm that considers collision costs and regional constraints. A similar approach demonstrated in [80] uses an iterative method to optimize the optimal safe path. In contrast, our method is more efficient due to its formulation. Our pipeline consists of three steps: first, find the shortest path from the start to the goal using the A* algorithm; second, inflate the path to get a tunnel for perturbation; third, generate the APF and search for the optimal path inside the tunnel.

Figure 2.11 illustrates the effect and importance of this perturbation process in navigation, in which the map is partially unknown. To penalize the path approaching obstacles too closely, we generate a local APF around nearby obstacles (indicated by the colored dots). There are three different paths from the robot's current position to the goal inside the room: the blue path is the shortest path derived from A*; the green path is the path that avoids both the potential field and obstacles; the magenta path is the optimal path planned using the proposed method. The magenta path goes through the middle of the door without any

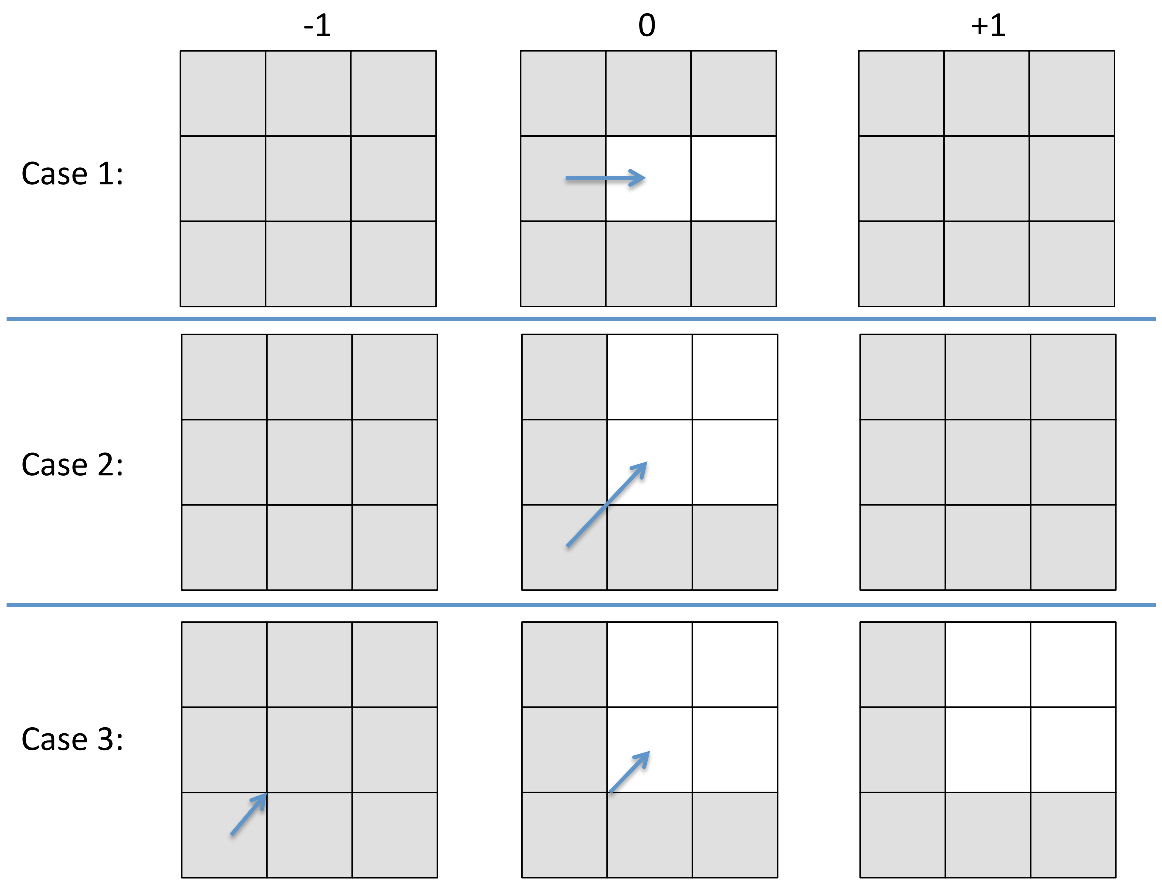


Figure 2.9: Neighbor Pruning. We draw a $3 \times 3 \times 3$ voxel grid as three 3×3 2D layers – bottom (-1), middle (0), top (+1). The center node indicated by the blue arrow is currently being expanded. The *natural neighbors* of the current node are marked in white. The pruned neighbors are marked in grey. The blue arrow also shows the direction of travel from its parent which includes three cases: (1) straight, (2) 2D diagonal, and (3) 3D diagonal.

detours and is better than other two paths considering both travel distance and collision cost.

The first step in the proposed pipeline is to find the shortest path, which has already been described in previous sections. In the following paragraphs, we introduce the last two steps.

Tunnel Around Path

A tunnel in configuration space around the given path bounds the perturbation. Let $D(r)$ be the disk with radius r, P be the given path, the tunnel $T(D, P)$ is the Minkowski sum

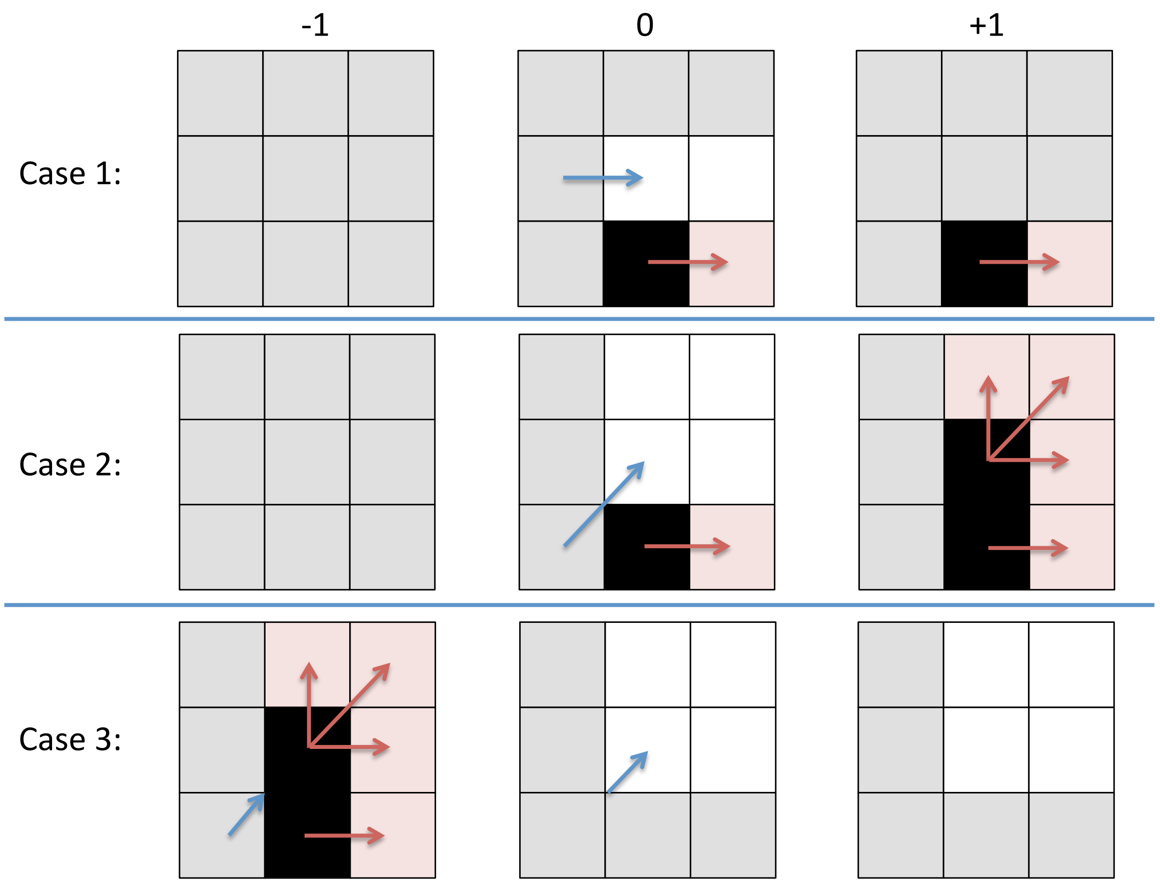


Figure 2.10: Forced Neighbors. When the current node is adjacent to an obstacle (black), the highlighted *forced neighbors* (light red) cannot be pruned. The red arrow indicates the pair of an obstacle and its corresponding *forced neighbor*: if the tail voxel is occupied, its head voxel is a *forced neighbor*. For example, in Case 1, if the voxel $(0, 1, 0)$ is occupied, $(1, 1, 0)$ is a *forced neighbor*. In Case 2, the occupied voxel $(0, 0, 1)$ results in three *forced neighbors* and similarly in Case 3. We omit drawing the symmetric situations with respect to the blue arrow.

of P and D:

$$T(D, P) = P \oplus D. \tag{2.16}$$

Figure 2.12 shows three examples of tunnels generated using different $D(r)$ from the same path. In general, the tunnel is non-convex. To implement the tunnel constraint in the pseudo-code, we simply augment each cell state with a flag to indicate whether it is inside the tunnel or not.

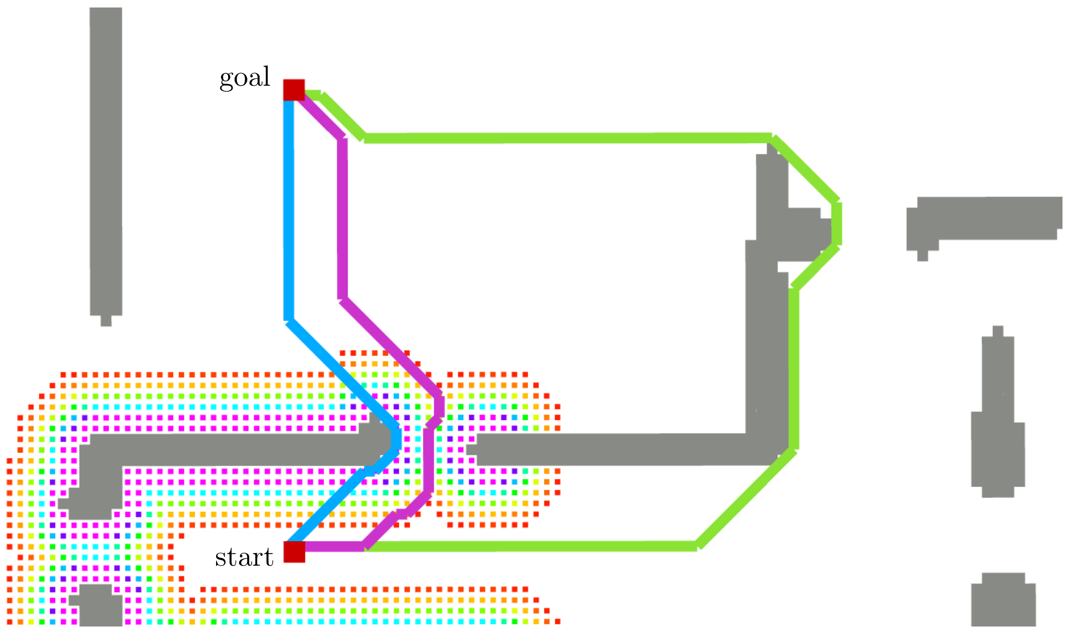


Figure 2.11: When planning a path from the robot's current position to a goal inside the room, we examine three possible paths: the blue one is the shortest path derived from A*; the green one is the safest path that does not intersect with potential field (colored dots) or obstacles; the magenta one is the optimal path planned using our method.

Artificial Potential Fields

In our implementation on a grid map, the APF is discretized and truncated. Denote the truncated potential value of cell u as $U(u)$, it is defined as:

$$U(u) = \begin{cases} 0, & d(u) \geq d_{\text{thr}} \\ F(d(u)), & d_{\text{thr}} > d(u) \geq 0 \end{cases} \tag{2.17}$$

where $d(u)$ is the distance of cell u to the closest obstacle. The potential function $F(d)$ should be a scalar, positive, and monotonically decreasing function of distance d with domain $[0,\ d_{\text{thr}})$. One choice that we use in Figure 2.11 is an exponential function of order k:

$$F(d) = F_{\max}(1 - d/d_{\text{thr}})^k \tag{2.18}$$

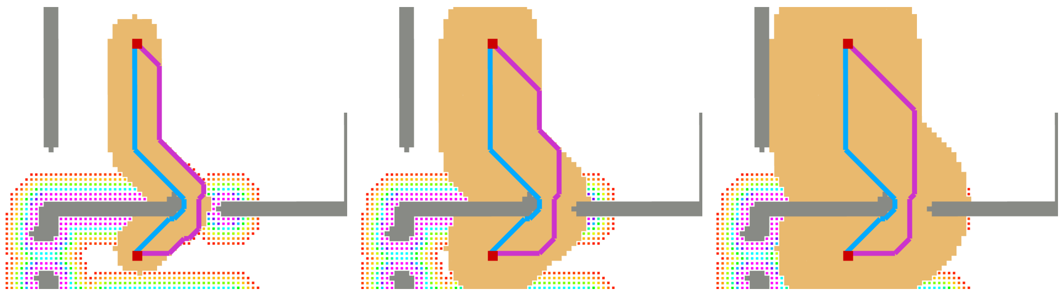

(a) $r = 0.5m$. (b) $r = 1.0m$. (c) $r = 1.5m$.

Figure 2.12: Our proposed method with different size tunnels, indicated in yellow. Left to right, the tunnel is created using the nominal shortest path (blue) with an increasing radius r. The perturbed path (magenta) is away from obstacles and is much safer to follow than the nominal path.

with $F_{\max},\ k > 0$. The parameter d_{thr} determines the maximum distance that is affected by the APF. To improve algorithm efficiency, we only generate the APF for the local map around the robot's current location in Figure 2.11.

Perturbation

To perturb the path, we need to change the optimization problem in (2.11) to account for a different cost function and a new set of constraints as compared to (2.13). Denote the u_s, u_g as start and goal vertices, w_U as the weight of potential value, $T(D, P_{\text{prior}})$ as the tunnel around the given path P_{prior}, the path perturbation is to solve the optimal P^* for the following problem:

$$\begin{aligned} \arg\min_{P} \quad & \oint_P 1 + w_U \cdot U(s)\, ds \\ s.t \quad & u_s,\, u_g \in P, \\ & P \subset T(D, P_{\text{prior}}) \cap \mathcal{X}^{free}. \end{aligned} \tag{2.19}$$

The line integration in (2.19) is equivalent to the sum of the path length and the weighted value of the potential along the path. Therefore, the modified edge cost is simply the sum of edge length and weighted values of the potentials at the two vertices:

$$c(u, s) = d(u, s) + \frac{1}{2} w_U (U(u) + U(s)). \tag{2.20}$$

The tunnel constraint requires that the path should be located inside the tunnel and is collision-free. In the corresponding pseudo-code of Algorithm 1, we only need to modify the edge cost $c(u, s)$ in line 23 and forbid the successors of the vertex u from going outside of the tunnel. The heuristic function of Euclidean distance is still admissible for $w_U \cdot U(u) \geq 0$, such that we are able to guarantee the optimality of the planned path. Given the tunnel and APF, we can search for the optimal path using the modified Algorithm 1 to observe the results shown in Figure 2.12.

Please note the following two propositions:

1. The perturbed path converges to the local minimum within the tunnel.

2. The computational time increases as tunnel size increases.

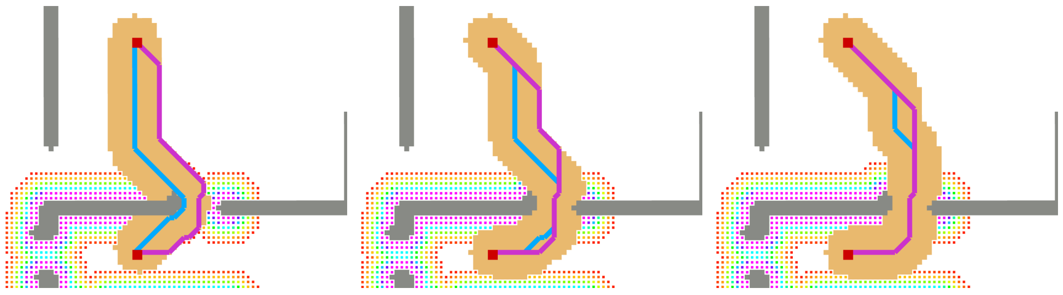
We briefly describe these two propositions below. By setting the size of tunnel to infinity, we can get the globally optimal path, but there is a trade-off between the path's sub-optimality and the algorithm's run time. In addition, as illustrated in Figure 2.11, it is not desirable to use the globally optimal path to navigate the robot in partially unknown environments.

For our application, we naively choose the tunnel radius r to be an arbitrary constant. An outer loop for iterative path optimization (2.19) can be used to ensure that the result converges to a locally optimal path. As shown in Figure 2.13, after 3 iterations, the optimal path converges to Figure 2.13c. We also extend this approach for anytime re-planning [48], which requires extremely fast computation.

2.2.5 Path Simplification

The raw path that comes from grid search consists of many consecutive cells. *Path simplification* removes redundant cells along the raw path and shortens the total path distance as much as possible. The simplified path contains long linear segments, which are easy to use for further control or trajectory generation.

The pseudo-code is illustrated in Algorithm 2. The shortening process skips the intermediate cells along the path such that the output path has as few cells as possible. The condition in line 7 guarantees the new path P' has a lower cost as compared to the cost of



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3.

Figure 2.13: Iterative perturbation of the robot's path. The blue path is the output from previous iteration. The magenta path is the optimized path from the current iteration. The path cost converges in (c) after 3 iterations.

the unsimplified path. The edge cost $c(u, v)$ is infinity if the line segment $u \to v$ is blocked by obstacles, and otherwise, it will stay the same as its cost during the graph search process. Thus, the new path P' will also be collision-free. Figure 2.14 compares the raw and shortened paths. Clearly, the simplified path is smoother than the raw path. For simplicity, the path refers to the simplified path in this thesis.

Algorithm 2 Path shortening. Given an initial path P, the function returns a shortened path P' by skipping intermediate waypoints on P. $P[i]$ refers to the i-th waypoint of P. $|P|$ means the number of P's waypoints. $c(\cdot,\ \cdot)$ is the edge cost.

```
1:  function Shortening(P)
2:      $p_{\text{ref}} \leftarrow P[1]$;
3:      $P' \leftarrow \langle p_{\text{ref}} \rangle$;
4:      $i \leftarrow 2$;
5:      for $i < |P|$ do
6:          $p_1 \leftarrow P[i],\ p_2 \leftarrow P[i+1]$;
7:          if $c(p_{\text{ref}},\ p_2) \geq c(p_{\text{ref}},\ p_1) + c(p_1,\ p_2)$ then
8:              $p_{\text{ref}} \leftarrow p_1$;
9:              $P' \leftarrow \langle P',\ p_{\text{ref}} \rangle$;
10:         end if
11:         $i \leftarrow i + 1$;
12:     end for
13:     $P' \leftarrow \langle P',\ P[|P|] \rangle$;
14:     return $P'$;
15: end function
```

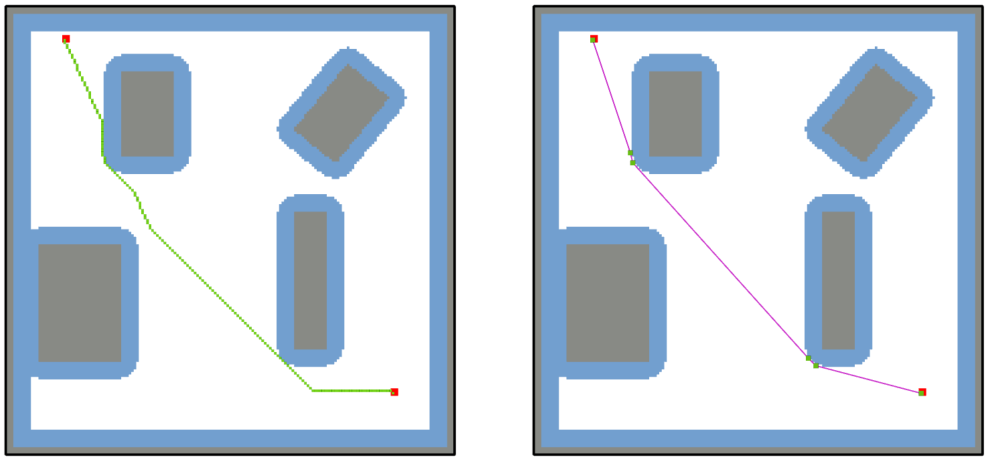

(a) Raw path. (b) Shortened path.

Figure 2.14: Path simplification. The raw path contains many consecutive cells (green dots), many of which can be removed without affecting the path's safety.

2.3 Navigating MAVs in Unknown Environments

In this section, we describe a simple but effective navigation pipeline combining the proposed planner and controller. A typical framework is plotted in Figure 2.15: the 'System' block stands for the robot, equipped with sensors that stream measurements m for mapping and y_m for localization; the 'State Estimation' estimates the robot's current state $\mathbf{x}$ at certain frequency using the sensor measurements y_m; the 'Mapper' fuses the robot's state and sensor measurements to generate the map M for planning; the 'Planner' finds a path using the up-to-date map M for a given start state u_s and a goal state u_g; the 'Controller' uses the robot's current state $\mathbf{x}$ and desired path P to generate a control command u. Thus, we have a closed-loop control paradigm for safely navigating a robot to a given destination.

We assume that the state estimation problem is solved using a vision-based approach [91] or laser-based scan matching [86], and a high-level state machine determines the goal coordinate. We mainly focus on the blocks inside the bounding box in Figure 2.15. In a non-trivial planning task, the environment is usually unknown and the path needs to be updated frequently when new map information is available.

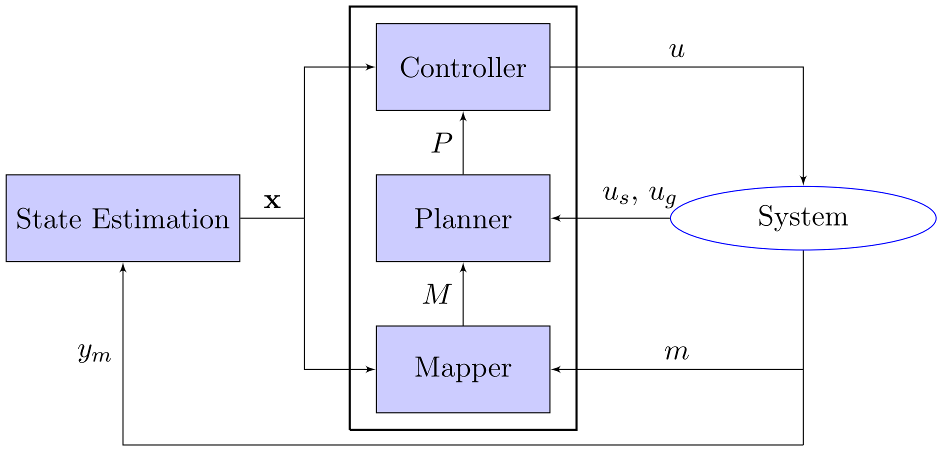


Figure 2.15: Navigation system diagram. In this section, we only consider the blocks inside the bounding box.

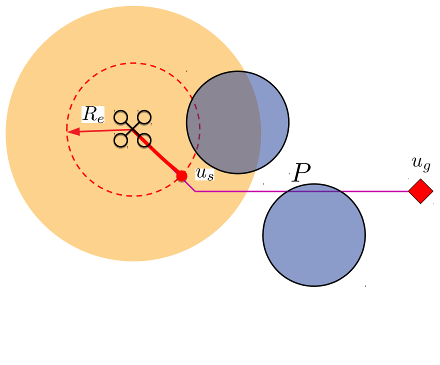
2.3.1 Mapper

There are many map representations commonly used in robotics, such as polygonal map, point cloud, occupancy grid map and octomap [33]. We choose to use the occupancy grid map because it can be updated efficiently. Each cell in the occupancy grid map has a state indicated by one of three possible values: $\{free, occupied, unknown\}$. Denote M as the occupancy grid map, M has three parts as M^{free}, $M^{occupied}$, $M^{unknown}$ corresponding to the sets of cells which are currently free, occupied and unknown.

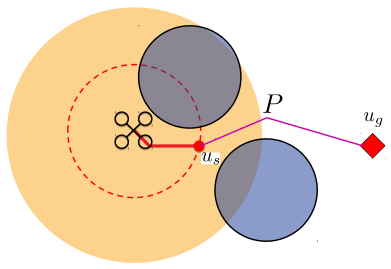
In our application, we start with a map that is partially unknown. In order to find a feasible path from start to goal, we treat unknown cells as free. This is a greedy assumption, since the unknown space can contain obstacles. Thus, in order to ensure safety, we need to keep updating the map and the desired path as new information becomes available. The frequency of map updates is determined by the sensor frequency, which is normally between 5 and 20 Hz

2.3.2 Planner

To begin with, we first review the concept of *Receding Horizon Control* (RHP) [85], which is a re-planning framework that we use to handle the navigation task in unknown environments (Figure 2.16). For path planning, we define the planning horizon in travel distance instead of



(a) Planning epoch 0.



(b) Planning epoch 1.

Figure 2.16: Re-planning using receding horizon control. The yellow disk indicates the robot's sensing rangeR_s and the execution horizon R_e (red dashed circle) must be smaller than this sensing range. The red line indicates the portion of the planned path (magenta) that will be executed. The robot detects the bottom right obstacle during the latter planning epoch, and the new path P avoids the potential collision.

time, as is typically done in *Model Predictive Control* (MPC). Since we are able to efficiently compute the path from start to goal, we assume a planning horizon R_h to be unlimited as $R_h \to \infty$. The execution horizon R_e depends on the computational time t and sensing range R_s. To guarantee the safety, we must make sure that R_e is smaller than the sensing range R_s. Denote $t_{\max}$ as the maximum time for the execution of path planning, $v_{\max}$ as the nominal maximum velocity. R_e should satisfy following inequality constraint:

$$v_{\max} \cdot t_{\max} < R_e < R_s. \tag{2.21}$$

Since our path planning algorithm is fast and complete, we do not need a long range planner and short range planner as described in [105]. Following the paradigm as illustrated in Figure 2.16, we are able to incrementally move the robot towards the goal.

Deterministic vs Randomized Path Planning

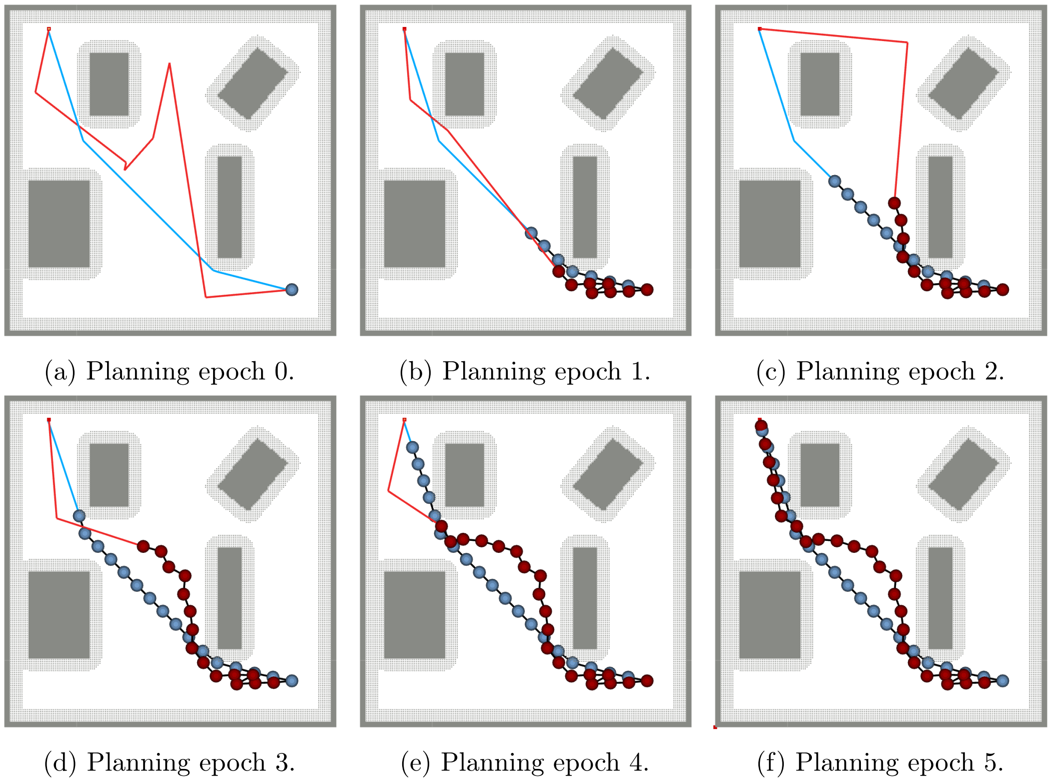
The comparison between search-based and sampling-based methods has a long history. Sampling-based approaches like the *rapidly-exploring random tree* approach (RRT) and its variations [37, 41, 43] find paths in a confined environment within a short computation duration. Sampling-based methods can solve certain problems that are hard or impossible to solve using graph search. For example, for robot manipulation and locomotion, the state space is extremely large thus it is unrealistic to solve the planning problem in a deterministic way. However, planning in the configuration space $\mathbb{R}^2$ or $\mathbb{R}^3$ is tractable as shown in Section 2.2.

On the other hand, randomized approaches are not suitable for re-planning due to their unpredictable behaviors: paths from two consecutive re-planning epochs using a sample-based method can be drastically different. For example, two robots are moving from the same starting location to the same destination in Figure 2.17 using A* and RRT*, assuming the map is static and known to simplify the problem. The robot using A* deterministically follows the optimal path after every re-planning epoch. The other robot using RRT* (here we use OMPL's implementation [89]) frequently changes its heading and follows detours. As a result, the robot using RRT* reaches the goal much more slowly than the robot using A* search. We draw the conclusion that deterministic planning approaches provide higher quality paths than randomized approaches do. Thus, if computation time allows, we always prefer to use the deterministic approach for planning rather than the randomized approach.

2.3.3 Controller

The basic control paradigm is introduced in Section 2.1 for MAVs. However, the geometric controller takes input as the desired flat output $\mathbf{x}_d$, not the path P. Thus, we need to extract the desired controller input from the planned path to fill this gap. There are many methods for path following [4, 11, 31, 70, 78, 90]. However, these methods can hardly guarantee dynamically feasibility and safety. For navigation at low speed $(1 < m/s)$, path following is an acceptable strategy since actuators are not prone to be saturated. But when it comes to flight at high speeds and agile maneuvers, path following is not adequate for precise control. Thus, the control paradigm in Figure 2.15 based on path following works for low speed navigation.

In this section, we describe a simple reference governor method in which we use the 1st order governor system to guide the MAV system. The state of governor robot is defined in form of the desired flat outputs as $\mathbf{x}_d(t) = [p_d^\mathsf{T}(t), \psi_d(t)]^\mathsf{T}$, where p_d is the desired position



(a) Planning epoch 0. (b) Planning epoch 1. (c) Planning epoch 2.

(d) Planning epoch 3. (e) Planning epoch 4. (f) Planning epoch 5.

Figure 2.17: A comparison of re-planning using search-based and sample-based methods. The blue path is planned by A* while the red path is planned by RRT*. Each robot's motion is plotted with red circles (A*) and blue circles (RRT*). Both robot re-plans after traveling 1 m along its previously planned path.

in $\mathbb{R}^m$ and ψ_d is the desired yaw. We update its state as

$$\begin{aligned} p_d(t+dt) &= p_d(t) + v_d(t)dt, \\ \psi_d(t+dt) &= \psi_d(t) + w_d(t)dt. \end{aligned} \tag{2.22}$$

$v_d(t),\ w_d(t)$ are the linear and angular governor velocities which are determined by the current governor state.

Determine v_d

The linear governor velocity v_d is determined through a collision-free sphere B. Denote the current robot position as p_c, the collision-free sphere is defined as

$$B(p_c, r) = \{\ p \mid \|p - p_c\| < r,\ p \in M^{free}\ \} \tag{2.23}$$

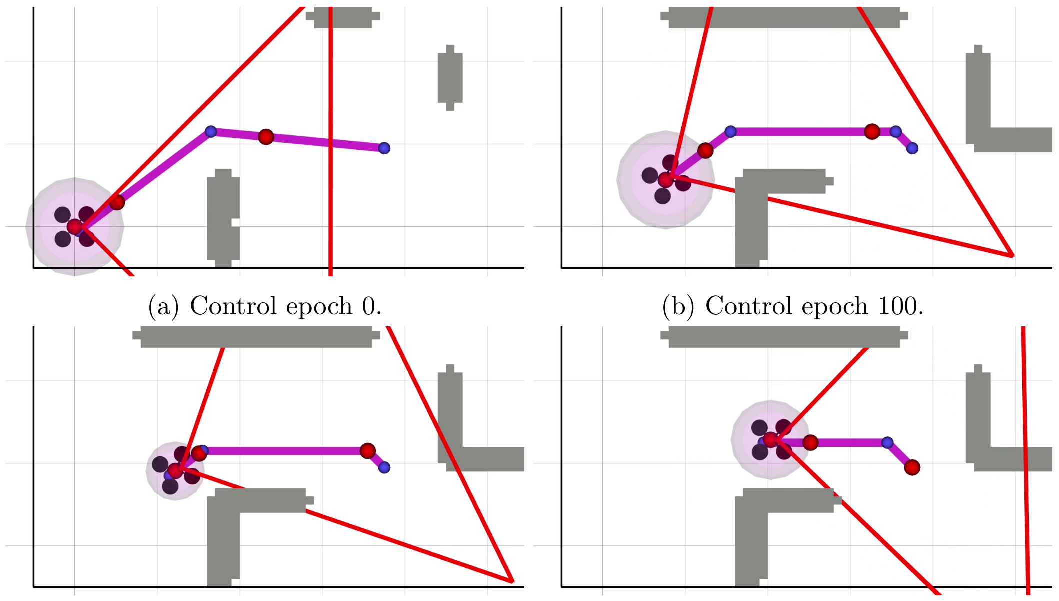

(a) Control epoch 0. (b) Control epoch 100. (c) Control epoch 200. (d) Control epoch 300.

Figure 2.18: Path following with the reference gorvernor method. The magenta sphere indicates the B_{max} at each control planning epoch. The red dot on B_{max} indicates the corresponding p_b.

The maximum collision-free sphere B_{max} is the largest collision-free sphere that satisfies that constraint $r < \bar{r}_{max}$ where $\bar{r}_{max}$ is an user-defined parameter. Denote $p_b(t)$ as the intersection point of B_{max} with the desired path P at current time stamp, the governor velocity $v_d(t)$ is defined as

$$v_d(t) = \frac{p_b(t) - p_c(t)}{\bar{r}_{max}} v_{max}, \ p_b(t) = B_{max} \cap P. \tag{2.24}$$

Where v_{max} is the maximum speed.

The definition of the intersection point p_b in (2.24) is not comprehensive. In particular, B_{max} can potentially intersect with P at multiple places. In order to prevent the robot going backward, we always choose p_b to be the point that is on the line segment of P that has the largest index.

Determine w_d

In the general case, the sensor for mapping has limited *field-of-view* (FOV). For example, the RGB-D sensor used in Figure 2.18 has 90° horizontal FOV (indicated by the red triangle).

Thus, when the robot follows the path, we align its yaw with respect to the direction of its desired linear velocity v_d which is defined in (2.24). We adopt the bang-bang control with a small dead-zone $[-\psi_{thr}, \psi_{thr}]$ to set the angular velocity:

$$w_d(t) = \begin{cases} -w_{max}, & \psi(t) > \psi_d(t) + \psi_{thr}, \\ 0, & |\psi(t) - \psi_d(t)| \leq \psi_{thr}, \\ w_{max}, & \psi(t) < \psi_d(t) - \psi_{thr}. \end{cases} \tag{2.25}$$

Where w_{max} is the maximum angular velocity for yawing.

The proposed control paradigm is able to navigate the robot with small values of v_{max}, w_{max}, but there is no guarantee for the high control authority since the control inputs are discontinuous and unbounded in acceleration. For precise and aggressive control of the MAV system, we need smoother control inputs which are available from time-parameterized trajectories. In the latter chapters, we propose trajectory planning methods that are able to properly solve the navigation problem as described in this chapter.

Chapter 3

Optimization-based Trajectory Planning

A trajectory can be expressed as a time-parameterized function that indicates a robot's desired states over time. This formulation has been widely used in combination with high frequency controllers for accurate and dynamic flight maneuvers [7, 29, 63, 68, 82]. Following the approach of these works, we formulate the robot's trajectory as a piece-wise polynomial function with respect to the flat outputs defined in Section 2.1.2. In this chapter, we introduce the *state-of-the-art* trajectory generation methods based on convex optimization.

3.1 Problem Formulation

Denote a trajectory $\Phi(t)$ as a piece-wise function which has N segments as:

$$\Phi(t) = \begin{cases} \Phi_0(t - t_0), & t_0 \le t < t_1 \\ \Phi_1(t - t_1), & t_1 \le t < t_2 \\ \vdots & \\ \Phi_{N-1}(t - t_{N-1}), & t_{N-1} \le t \le t_N. \end{cases} \tag{3.1}$$

Define the knot between segments Φ_i and Φ_{i+1} as u_i. A typical trajectory is drawn in Figure 3.1. Each of the segment Φ_i starts from the global time t_i to t_{i+1} and continuous with its consecutive segments at the corresponding knot. The end states u_0, u_N for the whole trajectory can be arbitrary constants.

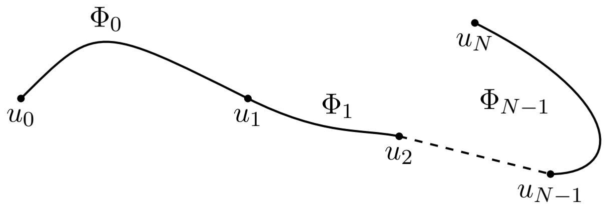


Figure 3.1: A piece-wise polynomial trajectory that has N segments.

As described in [63] and many other related works, the differential flatness of quadrotor systems allows us to construct control inputs from 1D time-parameterized polynomial trajectories specified independently in axis of each flat output of $\sigma = [\ x,\ y,\ z,\ \psi\ ]^\mathsf{T}$. Thus, a trajectory Φ has four dimensions which are respectively denoted as Φ_x, Φ_y, Φ_z and Φ_ψ. For a quadrotor system, the yaw is normally irrelevant to either collision checking or dynamic constraints. Hence, we ignore Φ_ψ in the optimization formulation and assume ψ is constant.

The general formula of an n-th order polynomial is denoted as:

$$p(t) := \sum_{n=0}^{n} a_n t^n = a_n t^n + \ldots + a_1 t + a_0, \tag{3.2}$$

where a_i, $i = 0, \ldots, n$ is constant. The polynomial state in one axis can be written in terms of $p(t)$ and its derivatives as

$$\begin{aligned} v(t) &= \dot{p}(t) = n a_n t^{n-1} + \ldots + a_1, \\ a(t) &= \ddot{p}(t) = n(n-1) a_n t^{n-2} + \ldots + 2a_2, \\ j(t) &= \dddot{p}(t) = n(n-1)(n-2) a_n t^{n-3} + \ldots + 6a_3. \end{aligned} \tag{3.3}$$

The full state of system's dynamics in $\mathbb{R}^3$ can be easily expressed by individual polynomials in three axes. For example, the trajectory segment Φ_i can be written as $\Phi_i(t) = [\ p_x^i(t),\ p_y^i(t),\ p_z^i(t)\ ]^\mathsf{T}$ where p_j^i, $j = x, y, z$ indicates the polynomial (3.2) of x, y, z-axis for the i-th segment.

3.1.1 Minimum Effort Trajectories

A minimum effort trajectory is the optimal solution of the following optimization problem:

$$\begin{aligned} \arg\min_{\Phi} \; & J(\Phi) \\ s.t \; & F(\Phi) \end{aligned} \tag{3.4}$$

where the cost function $J(\Phi)$ is a function relates to control efforts of the system. It can be expressed explicitly as:

$$J(\Phi) = \sum_{i=0}^{N-1} \int_{0}^{\Delta t_i} ||\Phi_i^{(q)}(t)||^2 dt \tag{3.5}$$

where the derivative order $q = 1, 2, 3, 4$ respectively applies for min-velocity, min-acceleration, min-jerk and min-snap trajectory. It should be noticed that the time for each segment $\Delta t_i = t_{i+1} - t_i$ is given as a prior in (3.5). It has been shown in [68] that the highest order of polynomial (3.2) n has an fixed relation with the derivative q:

$$n = 2q - 1. \tag{3.6}$$

Given the highest polynomial order n and the time for each segment Δt_i, the problem (3.4) is solvable in polynomial time for a variety of constraint functions corresponding to different applications. We will introduce two applications in this chapter to find collision-free trajectories using two types of constraints: waypoint constraints and *Safe Flight Corridor* constraints.

3.2 Waypoint Constraints

Waypoint constraints are specified by predefined waypoints u_i in Figure 3.1, in which the waypoint refers to either the robot's partial state or full state, including velocity, acceleration, etc. The trajectory is constrained to reach the each state u_i at a corresponding time t_i. For partially defined u_i, we need to guarantee the continuity of the unconstrained derivatives. As shown in [82], this problem can be formulated as an unconstrained Quadratic Program

(QP) which has a closed-form solution. In this section, we briefly review the technique to formulate and solve this unconstrained QP.

3.2.1 Cost Function in Matrix Form

For a single polynomial segment $p(t)$ from one state to another starting at time $t = 0$ with duration Δt, the cost function (3.5) *w.r.t.* q-th derivative can be written in the matrix form as:

$$J^q(p) = \int_0^{\Delta t} \|p^{(q)}(t)\|^2 dt = \mathbf{a}^\mathsf{T} Q^q(\Delta t)\mathbf{a} \tag{3.7}$$

where $\mathbf{a} = [a_n, \dots, a_0]^\mathsf{T}$ is the vector of coefficients and Q is the is the cost matrix constructed by the weighted sum of the Hessian matrix of polynomial derivatives. As shown in [82], the element in i-th row and j-th column of the Hessian matrix Q_{ij} can be calculated as:

$$Q^q_{ij} = \begin{cases} \left(\prod_{m=0}^{q-1}(i-m)(j-m)\right)\dfrac{\Delta t^{i+j-2q+1}}{i+j-2q+1}, & i \geq q \text{ and } j \geq q, \\ 0, & i < q \text{ or } j < q. \end{cases} \tag{3.8}$$

Thus, we compute the matrices for minimum-velocity, minimum-acceleration and minimum-jerk trajectories, respectively, as:

$$Q^1(\Delta t) = \begin{bmatrix} 0 & 0 \\ 0 & \Delta t \end{bmatrix}, \tag{3.9}$$

$$Q^2(\Delta t) = \begin{bmatrix} \mathbf{0}_{2\times 2} & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \begin{matrix} 4\Delta t & 6\Delta t^2 \\ 6\Delta t^2 & 12\Delta t^3 \end{matrix} \end{bmatrix}, \tag{3.10}$$

$$Q^3(\Delta t) = \begin{bmatrix} \mathbf{0}_{3\times 3} & \mathbf{0}_{3\times 3} \\ \mathbf{0}_{3\times 3} & \begin{matrix} 36\Delta t & 72\Delta t^2 & 120\Delta t^3 \\ 72\Delta t^2 & 192\Delta t^3 & 360\Delta t^4 \\ 120\Delta t^3 & 360\Delta t^4 & 720\Delta t^5 \end{matrix} \end{bmatrix}. \tag{3.11}$$

The Hessian cost matrix for trajectory in (3.1) Q^q can be constructed as:

$$Q^q = \begin{bmatrix} Q_0^q(\Delta t_0) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & Q_1^q(\Delta t_1) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & Q_{N-1}^q(\Delta t_{N-1}) \end{bmatrix}. \tag{3.12}$$

3.2.2 Equality Constraints

As mentioned earlier, the constraints on the the endpoints of a trajectory segment $p(t)$ are specified by the given waypoints u_i which can be expressed as linear equality constraints of the coefficients:

$$A(0) \cdot \mathbf{a} = b(0),\ A(\Delta t) \cdot \mathbf{a} = b(\Delta t). \tag{3.13}$$

Similar to the cost matrix Q^q defined in (3.8), A is a constant matrix *w.r.t.* time t as:

$$A_{ij}(t) = \begin{cases} t^{j-i} \prod_{m=0}^{i-1} (i-m), & i \geq j \\ 0, & i < j \end{cases} \tag{3.14}$$

Thus, we get the explicit matrices for minimum-velocity, minimum-acceleration and minimum-jerk trajectories, respectively, as:

$$A^1(0) = \begin{bmatrix} 1 & 0 \end{bmatrix}, \qquad A^1(\Delta t) = \begin{bmatrix} 1 & \Delta t \end{bmatrix}, \tag{3.15}$$

$$A^2(0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \qquad A^2(\Delta t) = \begin{bmatrix} 1 & \Delta t & \Delta t^2 & \Delta t^3 \\ 0 & 1 & 2\Delta t & 3\Delta t^2 \end{bmatrix}, \tag{3.16}$$

$$A^3(0) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}, \quad A^3(\Delta t) = \begin{bmatrix} 1 & \Delta t & \Delta t^2 & \Delta t^3 & \Delta t^4 & \Delta t^5 \\ 0 & 1 & 2\Delta t & 3\Delta t^2 & 4\Delta t^3 & 5\Delta t^4 \\ 0 & 0 & 2 & 6\Delta t & 12\Delta t^2 & 20\Delta t^4 \end{bmatrix}. \tag{3.17}$$

The big A matrix for trajectory in (3.1) can be concatenated as:

$$A = \begin{bmatrix} A_0(0) & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & A_0(\Delta t_0) & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & A_{N-1}(0) & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & A_{N-1}(\Delta t_{N-1}) \end{bmatrix}. \tag{3.18}$$

The matrix $b(t)$ on the right side of (3.13) is defined by the fixed value of waypoints and free derivatives. The free derivatives determine the total cost of the trajectory $J(\Phi)$. In the next section, we will compute the optimal values of the free derivatives as part of the optimal solution.

3.2.3 Reformulation as Unconstrained Quadratic Programming

According to previous two sections, we can formulate the optimization problem (3.4) as:

$$\begin{aligned} \arg\min_{\mathbf{a}} \ & \mathbf{a}^\mathsf{T} Q \mathbf{a} \\ s.t \ & A\mathbf{a} = b \end{aligned} \tag{3.19}$$

where $\mathbf{a} = [\ \mathbf{a}_0^\mathsf{T},\ \mathbf{a}_0^\mathsf{T},\ \ldots,\ \mathbf{a}_{N-1}^\mathsf{T}\ ]^\mathsf{T}$ is the vector of coefficients of all the trajectory segments and A, Q are calculated in (3.18) and (3.12).

The A matrix can be used to convert the endpoint derivatives b to polynomial coefficients as:

$$\mathbf{a} = A^{-1} b. \tag{3.20}$$

b is not fully defined because of the free derivatives. We can use a permutation matrix M to re-formulate b as two parts: fixed derivatives D_F and free derivatives D_P:

$$b = M \begin{bmatrix} D_F \\ D_P \end{bmatrix}. \tag{3.21}$$

Thus, the cost function in (3.19) is equivalent to

$$J = \begin{bmatrix} D_F \\ D_P \end{bmatrix}^{\mathsf{T}} \underbrace{M^T A^{-T} Q A^{-1} M}_{R} \begin{bmatrix} D_F \\ D_P \end{bmatrix} \tag{3.22}$$

where $R = M^T A^{-T} Q A^{-1} M$ can be expressed using four components:

$$R = \begin{bmatrix} R_{FF} & R_{FP} \\ R_{PF} & R_{PP} \end{bmatrix}. \tag{3.23}$$

Thus we get

$$J = D_F^{\mathsf{T}} R_{FF} D_F + D_F^{\mathsf{T}} R_{FP} D_P + D_P^{\mathsf{T}} R_{PF} D_F + D_P^{\mathsf{T}} R_{PP} D_P. \tag{3.24}$$

It is not hard to prove that $R_{FP}^T = R_{PF}$. Let $\frac{\partial J}{\partial D_P^*} = 0$, we can calculate the optimal free derivatives as

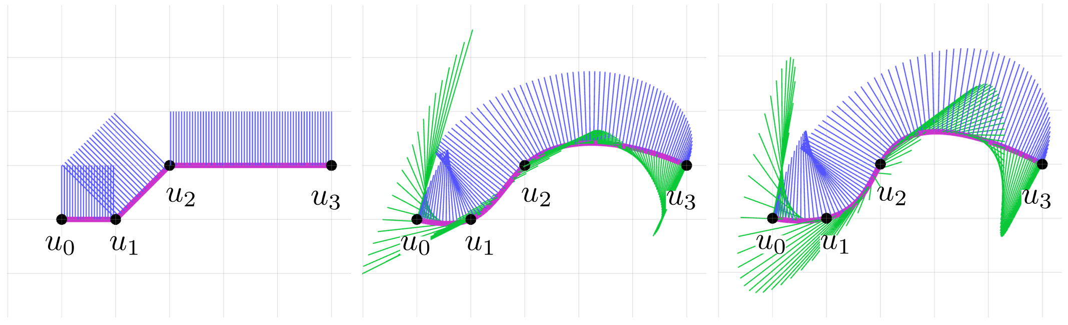
$$D_P^* = -R_{PP}^{-1} R_{FP}^{\mathsf{T}} D_F. \tag{3.25}$$

Substitute (3.21) into (3.20), the optimal polynomial coefficients are

$$\mathbf{a}^* = A^{-1} M \begin{bmatrix} D_F \\ D_P^* \end{bmatrix} = A^{-1} M \begin{bmatrix} I \\ -R_{PP}^{-1} R_{FP}^T \end{bmatrix} D_F. \tag{3.26}$$

3.2.4 Output Trajectories

The derivation for the minimum effort trajectories assumes that (3.6) holds. Thus the resulting trajectory will be continuous up to the order of $2(q-1)$. Figure 3.2 compares min-velocity, min-acceleration, and min-jerk trajectories given the same time allocation and waypoint constraints. For intermediate waypoints u_1, u_2, we fix only their positions. The endpoints u_0, u_3 are static. The time allocation is the same for all three trajectories, $\Delta t_0 = 1$, $\Delta t_1 = 1$, $\Delta t_2 = 3$. The min-velocity trajectory is only continuous in position, while the min-acceleration trajectory has continuous acceleration. The min-jerk trajectory has smooth acceleration and jerk in addition to continuous snap.



(a) Min-velocity. (b) Min-acceleration. (c) Min-jerk.

Figure 3.2: Output trajectories with the same time allocation and waypoints. We plot the velocity (blue), acceleration (green) at its corresponding position and rotate 90° for visualization. For the min-velocity trajectory, we omit the acceleration since the polynomial is first-order.

Though the computational time of the closed-form solution of the unconstrained QP (3.19) is negligible, the collision and dynamic constraints still need to be encoded. The dynamic constraints can be formulated as inequality constraints for the flat outputs: let $v_{\max}$, $a_{\max}$, $j_{\max}$ denote the max velocity, acceleration and jerk of polynomial p

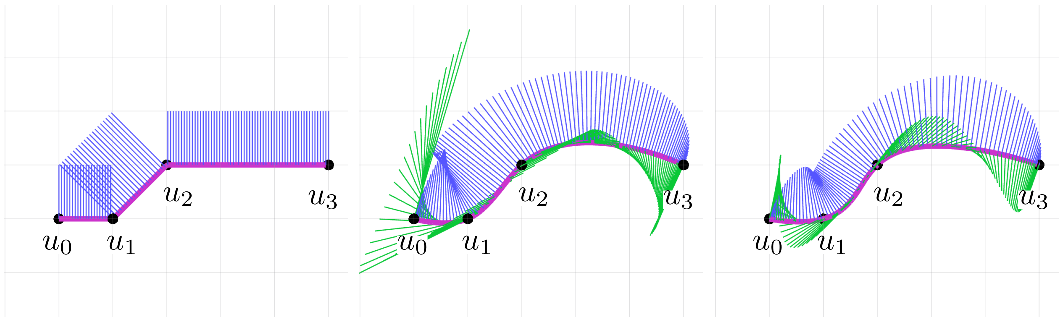
$$v_{\max} = \max\{\dot{p}_x, \dot{p}_y, \dot{p}_z\},\ a_{\max} = \max\{\ddot{p}_x, \ddot{p}_y, \ddot{p}_z\},\ j_{\max} = \max\{\dddot{p}_x, \dddot{p}_y, \dddot{p}_z\} \tag{3.27}$$

and denote the maximum bound of velocity, acceleration and jerk as scalars $\bar{v}_{\max}$, $\bar{a}_{\max}$ and $\bar{j}_{\max}$, the dynamic constraints are simply

$$v_{\max} \leq \bar{v}_{\max},\ a_{\max} \leq \bar{a}_{\max},\ j_{\max} \leq \bar{j}_{\max}. \tag{3.28}$$

It has been shown in [68] that if the Δt for a segment is large enough, the derivatives of its flat outputs including velocity, acceleration, and etc.will be bounded. Thus, we can scale the time profile to approximate the dynamic constraints. With Δt as the original time for the polynomial trajectory, we estimate the new polynomial time $\Delta t'$ through scaling.

$$\Delta t' = \max\{1, (\frac{v_{\max}}{\bar{v}_{\max}}), (\frac{a_{\max}}{\bar{a}_{\max}})^{\frac{1}{2}}, (\frac{j_{\max}}{\bar{j}_{\max}})^{\frac{1}{3}}\}\Delta t. \tag{3.29}$$



(a) Min-velocity. (b) Min-acceleration. (c) Min-jerk.

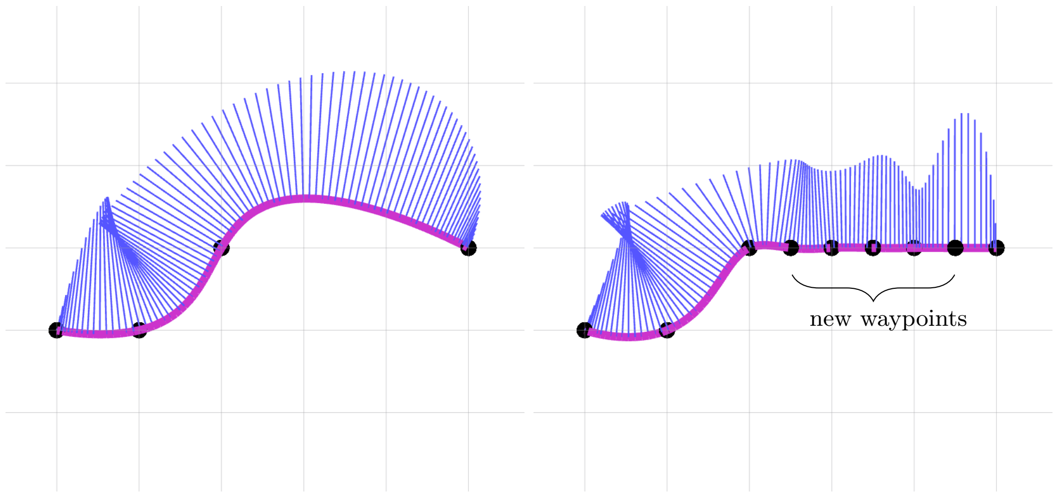
Figure 3.3: Scaled trajectories from Figure 3.2.

For example, in Figure 3.3 we scale the Δt for an individual trajectory segment *w.r.t.* the original trajectories in Figure 3.2 using the velocity bound $\bar{v}_{\max} = 1$. In Figure 3.3a, the time allocation does not change compared to Figure 3.2a since the $v_{\max} = 1$ for all the segments. In Figure 3.3b, the time for each segment is increased from $\Delta t_0 = 1$, $\Delta t_1 = 1$, $\Delta t_2 = 3$ (Figure 3.2b) to $\Delta t'_0 = 1.36$, $\Delta t'_1 = 1.24$, $\Delta t'_2 = 4.0$ and therefore the maximum velocity decreases from $v_{\max,0} = 1.36$, $v_{\max,1} = 1.24$, $v_{\max,2} = 1.33$ (Figure 3.2b) to $v'_{\max,0} = 1.0$, $v'_{\max,1} = 0.94$, $v'_{\max,2} = 1.0$. Similarly, in Figure 3.3c, the time for each segment is increased from $\Delta t_0 = 1$, $\Delta t_1 = 1$, $\Delta t_2 = 3$ (Figure 3.2c) to $\Delta t'_0 = 1.60$, $\Delta t'_1 = 1.60$, $\Delta t'_2 = 3.54$ therefore the maximum velocity decreases from $v_{\max,0} = 1.60$, $v_{\max,1} = 1.60$, $v_{\max,\,2} = 1.18$ (Figure 3.2c) to $v'_{\max,0} = 0.94$, $v'_{\max,1} = 0.94$, $v'_{\max,2} = 1.32$. Thus, the scaling of time allocation can reduce the maximum dynamics of the optimized trajectory, but it does not ensure the dynamic constraints (3.28) can be strictly satisfied.

In conclusion, the unconstrained QP is able to quickly optimize minimum effort trajectories *w.r.t.* the given waypoints and time allocation constraints. However, it fails to ensure that the generated trajectory is collision-free and obeys the dynamics constraints (Table 1.1). In the next section, we introduce our method to handle these two constraints without dramatically increasing the run time.

3.3 Safe Flight Corridor Constraints

The method proposed in Section 3.2 only works for unconstrained problems with equality constraints. To generate collision-free trajectories in cluttered environments, one can add collision-free intermediate waypoints accordingly as described in [63, 82]. However, this method is not able to guarantee safety of the final trajectory and can easily result in undesirable velocity profiling (Figure 3.4). Thus, this method is not appropriate for navigation in cluttered, obstacle-dense environments.



(a) Original trajectory. (b) New trajectory.

Figure 3.4: To avoid collision, an easy solution is to add collision-free intermediate waypoint constraints and re-optimize the original trajectory. As shown in (b), the new trajectory obstains an oscillating velocity profile (blue line strips) which is not desirable.

Generating a collision-free trajectory has been attempted with Mixed Integer methods in [12, 15, 62]. However, solving a MILP/MIQP problem is generally slow, so other approaches have been developed to remove the integer variables and solve the QP instead, which is much faster [9, 54, 81]. [9] requires an OctoMap [33] representation and produces sequences of axes-aligned cubes in free space to generate trajectories. This formulation of convex free space is not generic and is efficient only when obstacles are rectangular parallelepipeds. In this section, we propose a method that uses a linear piece-wise path produced by a fast graph search algorithm to guide the convex decomposition of the map to find a

Safe Flight Corridor (SFC). The SFC is a collection of convex connected polyhedra, which model the free space in the map and can be treated as linear inequality constraints in the QP for trajectory optimization. Inspired by [14], we developed a novel convex decomposition method to construct the SFC using ellipsoids. The total time for trajectory generation using this pipeline is sufficiently small such that we use it with a Receding Horizon Control framework as described in Section 2.3 to build our navigation system.

The overall architecture of our algorithm is shown in Figure 3.5. Since the path planning is solved in Section 2.2, we mainly discuss the *SFC Construction* and *Trajectory Optimization* in this section.

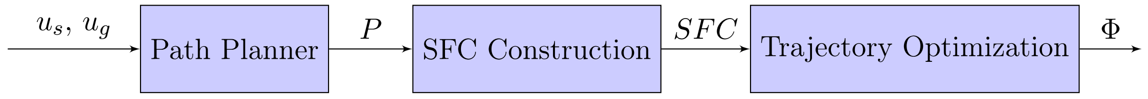


Figure 3.5: Block diagram of our algorithm that generates a trajectory from start u_s to goal u_g. We first find a valid path in a grid map towards the goal, based on which we construct the *Safe Flight Corridor* (SFC) through convex decomposition. The trajectory Φ inside the SFC is achieved from solving a optimization problem.

3.3.1 Safe Flight Corridor Construction

The set of points that constitute the obstacles (the occupied voxels in the map representation of the environment) are represented as O. A piece-wise linear path P from start to goal in the free space is denoted as $P = \langle \mathbf{p}_0 \rightarrow \mathbf{p}_1 \rightarrow \ldots \rightarrow \mathbf{p}_N \rangle$, where $\mathbf{p}_i$ are points in the free space and $\mathbf{p}_i \rightarrow \mathbf{p}_{i+1}$ are directed line segments in the free space. We generate a convex polyhedron around each line segment in P to construct a valid SFC. The i-th line segment is represented as $L_i = \langle \mathbf{p}_i \rightarrow \mathbf{p}_{i+1} \rangle$. Denote the generated convex polyhedron from each L_i as C_i. The space covered by these convex polyhedra forms the *Safe Flight Corridor*. We denote the collection of these convex polyhedra as $\text{SFC}(P) = \{C_i \mid i = 0, 1, \ldots, N-1\}$. Figure 3.6 shows an example of a path P and the corresponding $\text{SFC}(P)$.

One criterion for the construction of the SFC is that two consecutive polyhedra, C_i and C_{i+1}, need to intersect in a non-empty set containing $\mathbf{p}_{i+1}$. This ensures continuity in the SFC.

To generate the convex polyhedron C_i from L_i, we describe two procedures: (1) "Find

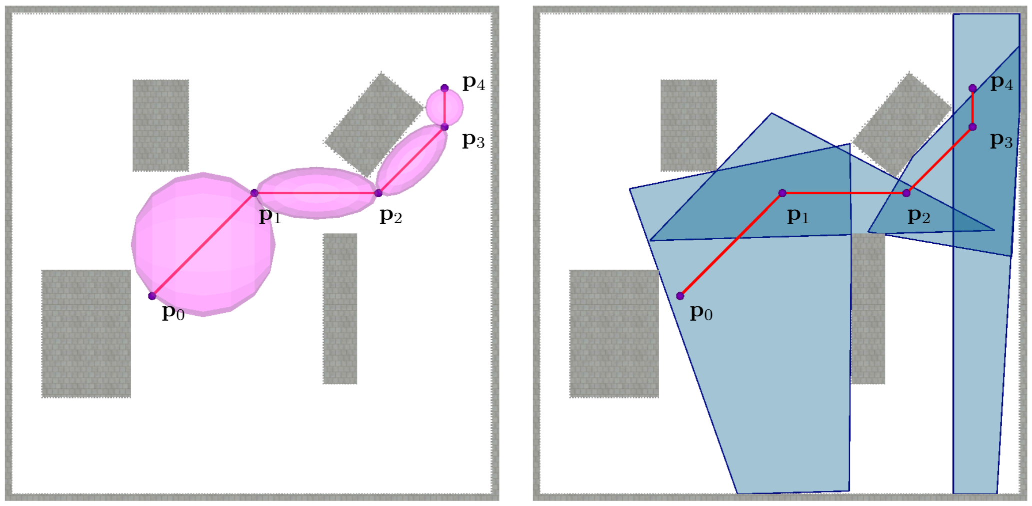


Figure 3.6: Generate a *Safe flight corridor* (blue region) from a given path $P = \langle \mathbf{p}_0 \rightarrow \ldots \rightarrow \mathbf{p}_4 \rangle$. Left: find the collision-free ellipsoid for each line segment. Right: inflate each individual ellipsoid to find a convex polyhedron.

Ellipsoid", that first fits an ellipsoid around L_i, and, (2) "Find Polyhedron", that constructs the polyhedron C_i from tangent planes to a sequence of dilated ellipsoids. In order to reduce the computation time, we add a bounding box to confine the space around L_i in which we consider obstacles. In addition, we propose a shrinking process to guarantee that a non-point robot is collision-free. In the following subsections we introduce the details on these procedures. For simplicity, we remove the subscripts "i" and simply use L, C to denote the corresponding line segment and polyhedron.

Step 1 – Find Ellipsoid

In this step we find an ellipsoid which includes the line segment L and does not contain any obstacle points from O. An ellipsoid is described as

$$\xi(\mathbf{E}, \mathbf{d}) = \{\mathbf{p} = \mathbf{E}\bar{\mathbf{p}} + \mathbf{d} \mid \|\bar{\mathbf{p}}\| \leq 1\}. \tag{3.30}$$

For an ellipsoid in $\mathbb{R}^n$, $\mathbf{E}$ is a $n \times n$ symmetric positive definite matrix that represents a deformation of a sphere ($\|\bar{\mathbf{p}}\| \leq 1$). Considering 3D case, $\mathbf{E}$ can be decomposed as $\mathbf{E} = \mathbf{R}^T\mathbf{S}\mathbf{R}$ where $\mathbf{R}$ is the rotation matrix aligning the ellipsoid axes with map axes and $\mathbf{S} = \text{diag}(a, b, c)$ is the diagonal scale matrix whose diagonal elements stand for the corresponding lengths of ellipsoid semi-axes. $\mathbf{d}$ indicates the center of the ellipsoid. Without loss of generality, we assume $a \geq b,\ a \geq c$. Our goal is to find $\mathbf{E}, \mathbf{d}$ given the line segment L and obstacles O.

This ellipsoid is computed in two steps: first, we shrink an initial sphere to derive the maximal spheroid (an ellipsoid with two axes of equal length); second, we "stretch" this spheroid along the third axis to obtain the final ellipsoid. In the first step, the initial ellipsoid is a sphere centered at the mid point of L and with diameter equals to the length of L. Assume the length of ellipsoid's $\tilde{x}$-axis is fixed and aligned with L, we reduce the length of other two axes until the spheroid contains no obstacles. This is done by searching for the closest obstacle in O from the center of ξ. Figure 3.7 shows the shrinking process from a 2D perspective.

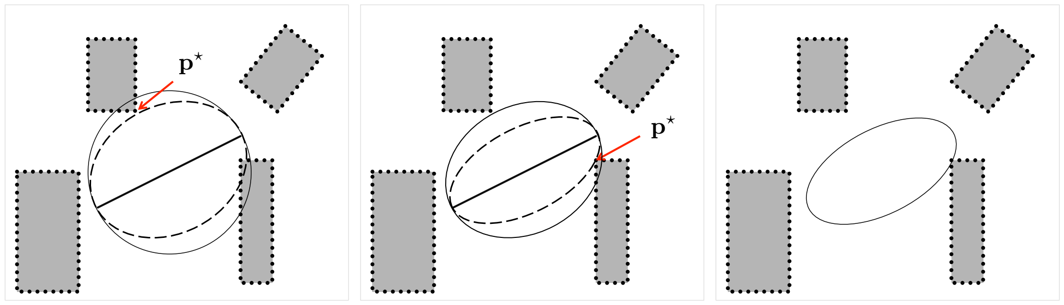


Figure 3.7: Shrink ellipsoid ξ. The bold line segment is L, gray region indicates obstacle while the white region is free space. Left: start with a sphere, we find the closest point $\mathbf{p}^\star$ to the center of L and adjust the length of short axes such that the dashed ellipsoid touches this $\mathbf{p}^\star$. Middle: repeat the same procedure, find a new closest point $\mathbf{p}^\star$ and the new ellipsoid. Right: no obstacle is inside the ellipsoid, current ellipsoid is the max spheroid. Several iterations are required to ensure the final spheroid excludes all the obstacles.

The maximal spheroid touches an obstacle at $\mathbf{p}^\star$ which, along with the line segment L, defines the plane of $\tilde{x}$-$\tilde{y}$ axes of the spheroid. Following that, we stretch the length of the $\tilde{z}$-axis of the spheroid to make it equal to a to form a new initial ellipsoid. The actual value of c can be determined through finding another closest point using the similar process as shown in Figure 3.7.

Step 2 – Find Polyhedron

Denote the ellipsoid found in the previous step as ξ^0, which touches an obstacle point at $\mathbf{p}_0^c = \mathbf{p}^*$. The tangent plane to the ellipsoid at this point creates a half space $H_0 = \{\mathbf{p} \mid \mathbf{a}_0^T\mathbf{p} < b_0\}$, containing the ellipsoid. After computing H_0, we remove all the obstacles in O that lie outside H_0 (call this the set of *remaining* obstacles, O_{remain}), and "dilate" the ellipsoid (keeping its aspect ratio constant) until it is in contact with another obstacle point, $\mathbf{p}_1^c$, at which point the new ellipsoid is called ξ^1 and the new tangent hyper-plane creates a new half-space H_1. This process is continued to obtain a sequence of half-spaces, $H_0, H_1, \cdots, H_m$. The intersection of these $m+1$ half-spaces gives the convex polyhedron, $C = \bigcap_{j=0}^m H_j = \{\mathbf{p} \mid \mathbf{A}^T\mathbf{p} < \mathbf{b}\}$, where $\mathbf{a}_j$ and b_j are the j-th column of matrix $\mathbf{A}$ and element of vector b respectively.

Figure 3.8 shows an example of ellipsoid dilation. In each dilate iteration, the ellipsoid ξ^j touches an obstacle at a point $\mathbf{p}_j^c$. Algorithm 3 shows the pseudo-code. The hyper-plane defining the j-th half-space, H_j, is the tangent to ξ^j at p_j^c, and is computed as

$$\begin{aligned} \mathbf{a}_j &= \left.\frac{d\xi_r}{d\mathbf{p}}\right|_{\mathbf{p}=\mathbf{p}_j^c} = 2\mathbf{E}^{-1}\mathbf{E}^{-T}(\mathbf{p}_j^c - \mathbf{d}), \\ b_j &= \mathbf{a}_j^T\mathbf{p}_j^c. \end{aligned} \tag{3.31}$$

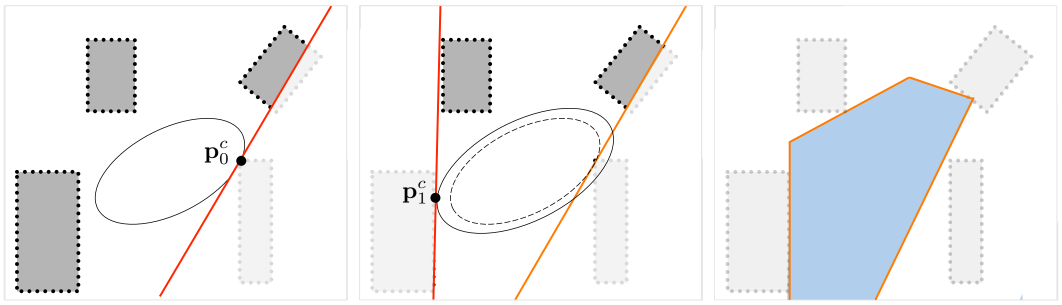


Figure 3.8: Dilate ellipsoid ξ^0 to find halfspaces. Left: find the first intersection point $\mathbf{p}_0^c$ for ξ^0 and hyperplane (red line), the obstacle points outside corresponding halfspace H_0 are removed (shadowed). Middle: find the next intersection point $\mathbf{p}_1^c$ (dashed ellipsoid shows the original ellipsoid ξ^0 and the solid ellipoid shows the new ellipsoid ξ^1), keep removing obstacle points from the map that are outside the new halfspace. Right: keep dilating until no obstacle remains in the current map, the convex space C (blue region) is defined by the intersection of the halfplanes.

Algorithm 3 Given $\xi^0(\mathbf{E}, \mathbf{d})$, find the $C(\mathbf{A}, \mathbf{b})$. The set of obstacle points is denoted as O.

```
function FindPolyhedron($\xi^0$, $O$)
    $O_{remain} \leftarrow O$;
    $j \leftarrow 0$;
    while $O_{remain} \neq \emptyset$ do
        $\mathbf{p}_j^c \leftarrow \text{ClosestPoint}(\xi^0, O_{remain})$;
        $\xi^j \leftarrow \text{DilateEllipsoid}(\xi^0, \mathbf{p}_j^c)$;
        $\mathbf{a}_j \leftarrow 2\mathbf{E}^{-1}\mathbf{E}^{-T}(\mathbf{p}_j^c - \mathbf{d})$;
        $b_j \leftarrow \mathbf{a}_j^T \mathbf{p}_j^c$;
        $O_{remain} \leftarrow \text{RemovePoints}(\mathbf{a}_j, b_j, O_{remain})$;
        $j = j + 1$;
    end while
    $C : \mathbf{A}^T \leftarrow \begin{bmatrix} \mathbf{a}_0^T \\ \mathbf{a}_1^T \\ \vdots \end{bmatrix}, \mathbf{b} \leftarrow \begin{bmatrix} b_0 \\ b_1 \\ \vdots \end{bmatrix}$;
    return $C(\mathbf{A}, \mathbf{b})$;
end function
```

So far, we are able to generate the polyhedron C for L, given the obstacles O. We apply this method on each individual line segment of the path P to get the *Safe Flight Corridor* as $\text{SFC}(P) = \{C_i \mid i = 0, 1, \ldots, n-1\}$ (Figure 3.6). Since the original ellipsoid is inside the corresponding polyhedron, we have a guarantee that the line segment L is also inside the polyhedron. Thus the whole path P is guaranteed to be inside $\text{SFC}(P)$.

Bounding Box

The algorithm, as presented, needs to search through all the points in O at least twice to check for the intersection with the inflated ellipsoid when constructing the polyhedron C for each line segment L. This is an expensive process. We decrease the number of points to be checked for by adding a bounding box around L, and thus only searching for the obstacle points inside it. This process saves a large amount of computation time and also prevents the trajectory from going too far away from the original path. The bounding box for L is composed of 6 rectangles such that the axis of the bounding box is aligned with L and the minimum distance from each face to L is r_s. If the maximum speed and acceleration of the MAV is $v_{\max}, a_{\max}$, the condition imposed on the safety radius is $r_s \geq \frac{v_{\max}^2}{2a_{\max}}$. Figure 3.9

shows the typical result from applying the bounding box. The generated SFC contains similar halfspaces as shown in Figure 3.6.

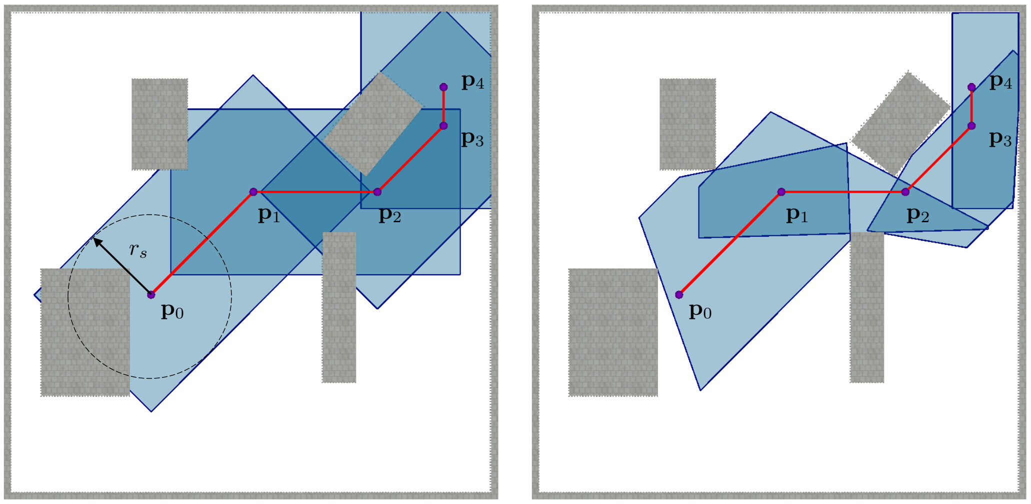


Figure 3.9: Left: apply bounding box on each line segment with safety radius r_s. Right: inflate individual line segment to find the convex polyhedron. We only process the obstacles inside corresponding bounding box comparing to Figure 3.6.

Shrink

We model the robot as a sphere with radius r_r and expand occupied voxels in the original map M to generate the configuration space M_e such that we are able to treat the robot as a single point for planning. When constructing the SFC for path P planned in M_e, using M_e could generate narrow ellipsoids and polyhera (Figure 3.10a – 3.10b). In order to avoid such kind of bad SFC, we use the original map M to generate the SFC and shrink the SFC by the robot radius r_r in order to guarantee safety. The shrinking process is applied by pushing every support hyper-plane along its normal by r_r. This process ensures the safety of the shrunken SFC as we increase the distance between obstacles and each hyper-plane by r_r, but may also exclude some portion of the path (Figure 3.10d) which may cause discontinuity of the *Safe Flight Corridor*. To guarantee the continuity, we have to make sure the line segment L is inside the shrunken polyhedron C'. For this, we modify Algorithm 3: for any half-space $H_j \in C$ (C is the raw polyhedron), we check the minimum distance $d(L, H_j)$ from L to the hyper-plane of H_j. If $d(L, H_j) < r_r$, we adjust the normal of the hyper-plane such that $d(L, H'_j) = r_r$ (H'_j is the adjusted half-space). The hyper-plane of the new half-space H'_j also passes through the intersection point of H_j with the dilated ellipsoid (Figure 3.10e).

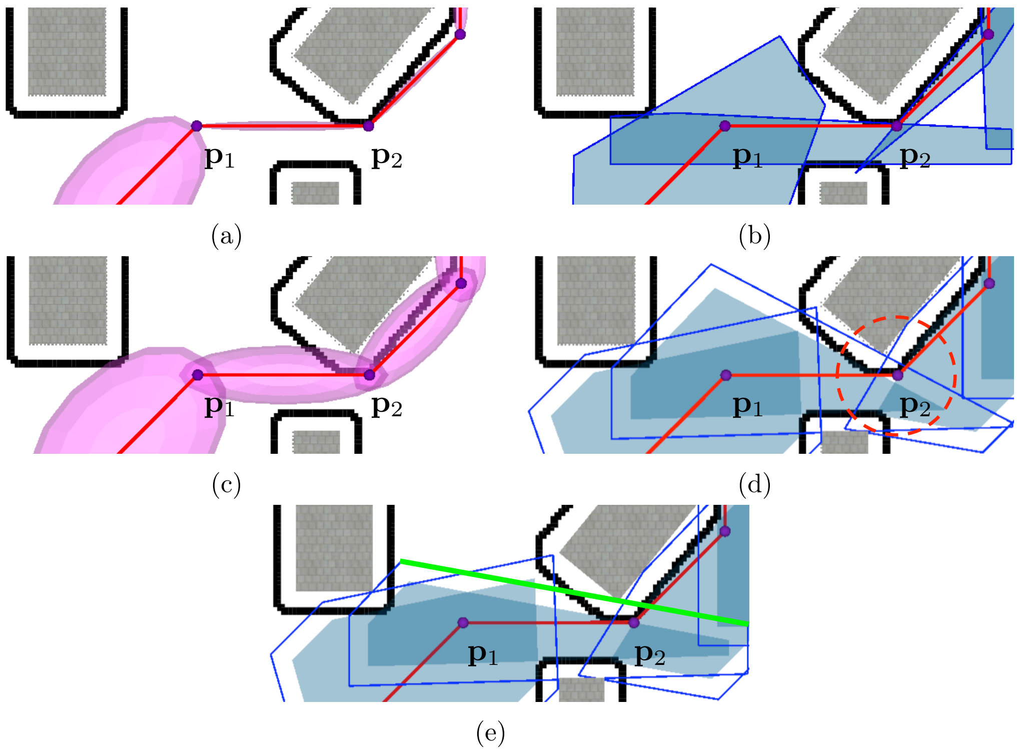


Figure 3.10: Constructing the SFC through the shrinking process. For clarity, we draw the contour of the expanded map M_e using black bold lines. The contours of the SFC are indicated by blue boundaries while the shrunken SFCs are drawn as blue regions inside. The SFC in (a) and (b) is derived using M_e without shrinking. Several ellipsoids and corresponding polyhedra are quite narrow. In (c) and (d), the SFC is generated using the original map M such that the corridor is "wider" compared to (a) and (b). Since this SFC also penetrates obstalces in the expanded map, we shrink it by the robot radius r_r to derive the "safe" SFC. However, this shrinking process may cause discontinuities in the SFC, for example $\mathbf{p}_2$ (circled) is outside of the shrunken polyhedron generated from line segment $\mathbf{p}_1 \to \mathbf{p}_2$ in (d). In (e), the green hyperplane is adjusted such that $\mathbf{p}_2$ is still inside the shrunken polyhedron.

3.3.2 Trajectory Convex Optimization

Given the SFC, we could modify the original optimization (3.4) with collision constraints to generate minimum effort trajectories. A trajectory is defined as the piece-wise polynomial in (3.1). The cost function is the same as (3.5):

$$J^q(\Phi) = \sum_{i=0}^{N-1} \int_0^{\Delta t_i} \|\Phi_i^{(q)}(t)\|^2 dt \tag{3.32}$$

where $q = 1 \ldots 4$ correspond to min-velocity, min-acceleration, min-jerk and min-snap trajectories. For the intermediate waypoints, we enforce the continuity constraints similar to (3.19) as:

$$\Phi_i^{(k)}(\Delta t_i) = \Phi_{i+1}^{(k)}(0), \quad k = 0 \ldots q \text{ and } i = 1 \ldots N-2. \tag{3.33}$$

The endpoints of the trajectory is specified as input:

$$\Phi_0^{(k)}(0) = u_s^{(k)},\ \Phi_{N-1}^{(k)}(\Delta t_{N-1}) = u_g^{(k)}, \quad k = 0 \ldots q. \tag{3.34}$$

Different from Section 3.2, we do not need to encode the specific position constraint for the intermediate waypoint – it is free to be inside the overlapped space between the two connected polyhedra.

Assume the SFC contains N convex polyhedra, the whole trajectory is composed of N polynomials and the i-th polynomial is inside the i-th polyhedron C_i. Thus, the explicit expression of the optimization for minimum effort trajectories can be formed as a convex optimization as

$$\begin{aligned}
\arg\min_{\Phi}\ J^q &= \sum_{i=0}^{n-1} \int_0^{\Delta t_i} ||\Phi_i^{(q)}(t)||^2 dt \\
s.t\ \ \Phi_i^{(k)}(\Delta t_i) &= \Phi_{i+1}^{(k)}(0), \quad k = 0 \ldots q \text{ and } i = 1 \ldots N-2, \\
\Phi_0^{(k)}(0) &= u_s^{(k)},\ \Phi_{N-1}^{(k)}(\Delta t_{N-1}) = u_g^{(k)}, \quad k = 0 \ldots q, \\
\mathbf{A}_i^T \Phi_i(t) &< \mathbf{b}_i, \quad i = 0 \ldots N-1.
\end{aligned} \tag{3.35}$$

Here the matrices $\mathbf{A}_i$, $\mathbf{b}_i$ correspond to the i-th polyhedron C_i. Figure 3.11 shows an example of the optimized trajectory by solving (3.35) that is confined by the certain SFC.

Similar to Section 3.2.4, we approximate the dynamic constraints using *Time Allocation*. This is mainly for saving computational time by reducing the number of inequality constraints in the proposed convex optimization. *Time Allocation* significantly affects the resulting trajectories. As every SFC contains a path P, the naive *Time Allocation* method is to map this P into time domain using trapezoid velocity profile (Figure 3.12). We assume

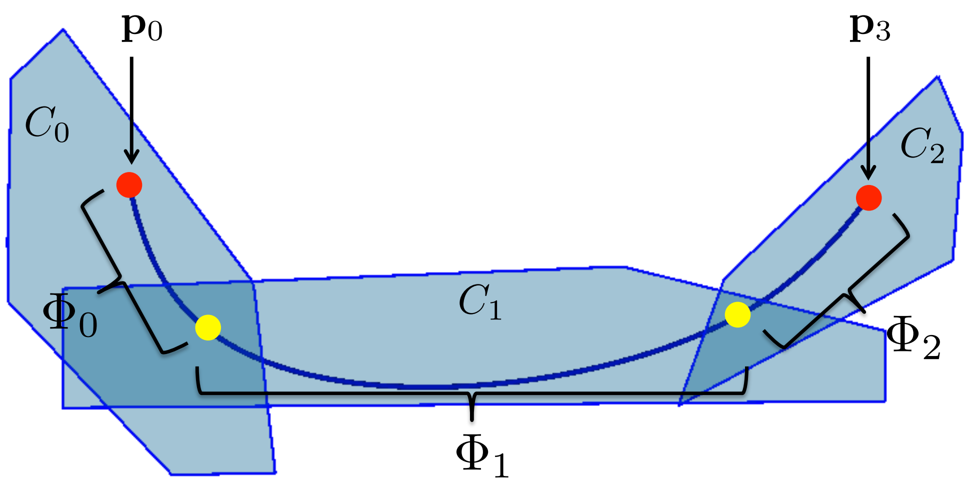


Figure 3.11: Example trajectory has three polyhedra C_i and each segment Φ_i is confined to be inside its corresponding polyhedron. The red start and end points are confined to be at those locations and the yellow knot points are only constrained to be continuous and are allowed to vary within the intersection of adjacent pairs of polyhedra.

the robot is a particle that moves in 1D as a second order system. The time Δt for each segment is a function of the path segment length and the robot speed. In a more delicate model, when the initial state of the real robot is non-static, the initial and end speeds for the particle are not necessarily to be zero.

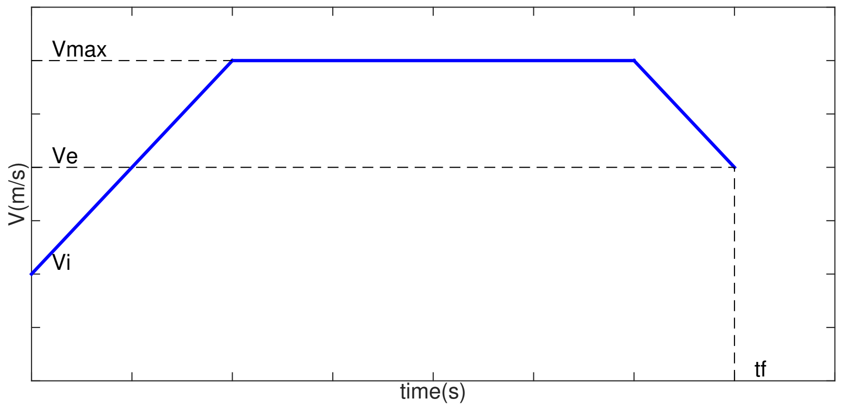


Figure 3.12: *Time Allocation* for a given path, denote robot initial speed as v_i, max speed as $v_{\max}$ and end speed as v_e.

Solving (3.35) with the naive *Time Allocation* will result in trajectories with large velocity, acceleration or jerk that exceed the maximum thresholds of the MAV. We modify Δt_i

according to (3.29) to adjust the *Time Allocation* and re-optimize (3.35) to solve the final trajectory using $\Delta t_i'$. These two-step approach will generate the trajectory that does not violate the dynamic constraints during most of the traversing time.

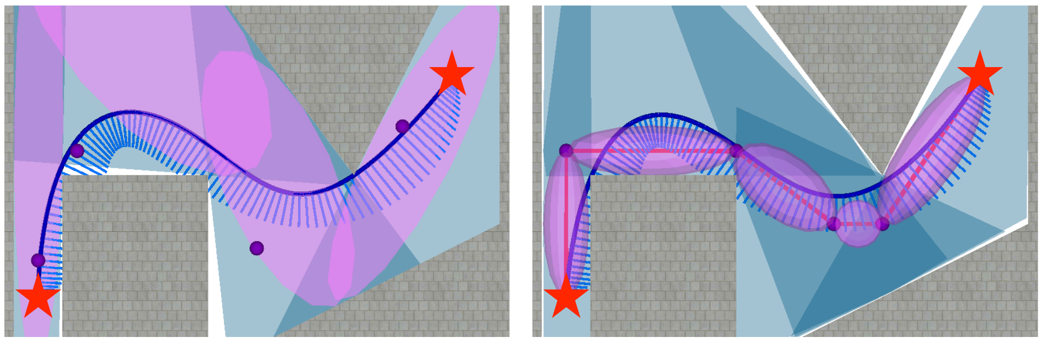
It is important to note that we use a sample-based method to confine each polynomial when solving (3.35). We omit the details which can be found in [61]. Also, we always assume a stopping policy for a static end state with zero velocity, acceleration and jerk to ensure the flight safety.

3.3.3 Evaluation of Output Trajectories

In this part, we evaluate the proposed algorithm.

Comparison with IRIS

The existing algorithms for generating the collision-free convex region [14, 15] requires a proper selection of seeds and a geometric representation of obstacles which is hard to get from real sensor data. In their process (called IRIS), solving the maximum ellipsoid through convex optimization takes a long time. For the map shown in Figure 3.13, the IRIS algorithm takes around 110 ms while our algorithm only requires 4.8 ms. In fact, the selection of seeds for growing ellipsoids in IRIS is non-trivial, which also makes it harder to run IRIS for decomposition in real-time applications.

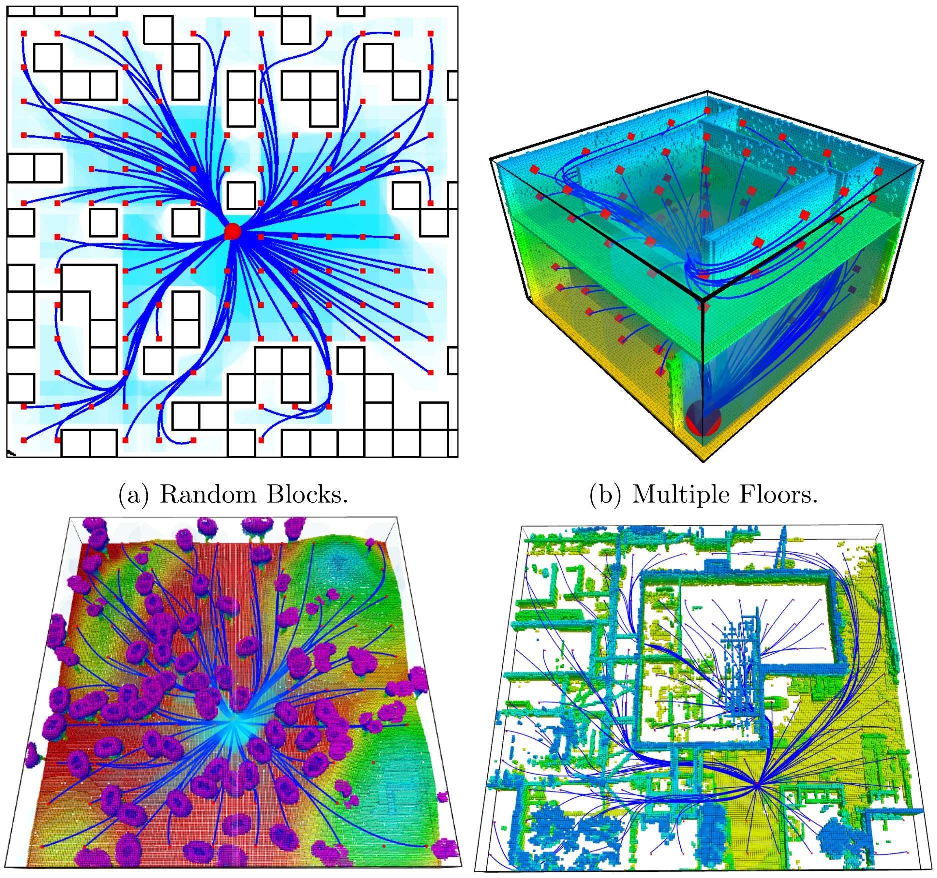


(a) IRIS. (b) SFC.

Figure 3.13: Comparing our convex decomposition approach with IRIS. Red stars point out the start and goal. The generated trajectories are very similar, even though using two different *Safe Flight Corridors*. The light blue short lines that are perpendicular to the trajectory show the speeds at corresponding positions.

Run Time Analysis

We use four different maps to test the run time of our algorithm by generating hundreds of trajectories through them. The four maps are named as 'Random Blocks', 'Multiple Floors', 'The Forest' and 'Outdoor Buildings'. We sample goals at a certain density in each map and manually select a start. Figure 3.14 shows these maps and generated results. These maps are selected because they are typical for different environments encountered in the real world (namely 2.5D, fully 3D, randomly scattered complex obstacles, and real-world data).



(a) Random Blocks.

(b) Multiple Floors.

(c) The Forest.

(d) Outdoor Buildings.

Figure 3.14: Generate trajectories from a start (big red ball) to sampled goals (small red balls) in different maps. The blue curves are generated trajectories, cyan region is the overlapped SFC.

To evaluate the computational expense of our algorithm, we split the whole trajectory generation into three parts: path planning, convex decomposition and trajectory optimization. Table 3.1 indicates the time cost for each component when generating trajectories as shown in Figure 3.14 on an i7-4800MQ processor. For path planning, we compared two different methods: A* and JPS to show the impact on run time by using JPS. As can be seen from the results, we are able to generate trajectories in under a few hundred milliseconds, which is sufficient fast for re-planning at 3 Hz.

Map	Size	# of Cells	# of Trajs	Time (s)	Path Planning A*	Path Planning JPS	Convex Decomp	Traj Opt	Replan (JPS)
Random Blocks	$40 \times 40 \times 1$	1.4×10^6	130	Avg	0.57	0.034	0.0021	0.028	0.065
				Std	1.26	0.034	0.0028	0.022	0.051
				Max	9.98	0.19	0.020	0.099	0.27
Multiple Floors	$10 \times 10 \times 6$	5.9×10^5	147	Avg	6.12	0.039	0.0064	0.082	0.13
				Std	15.77	0.046	0.0038	0.041	0.081
				Max	84.56	0.22	0.021	0.23	0.45
The Forest	$50 \times 50 \times 6$	1.8×10^6	89	Avg	0.65	0.033	0.0039	0.055	0.094
				Std	1.57	0.044	0.0024	0.031	0.068
				Max	7.78	0.20	0.010	0.12	0.30
Outdoor Buildings	$100 \times 110 \times 7$	6.2×10^5	127	Avg	0.54	0.028	0.0066	0.099	0.14
				Std	1.46	0.045	0.0053	0.064	0.10
				Max	10.96	0.27	0.027	0.24	0.47

Table 3.1: Trajectory Generation Run Time Analysis (sec).

Completeness

Here, we discuss the algorithmic completeness within the local map: whether a trajectory will be found if one exists up to the resolution of the map. Since construction of SFC starts with line segments, it will at least produce a set of convex regions that includes those line segments. In this case, the feasible set of the optimization always contains the solution where the trajectory Φ is polynomial with static starting and ending states. For other cases where the initial non-static dynamics cause the failure of the trajectory optimization, the vehicle will either follow the existing collision-free trajectory or execute a stopping policy. Eventually, the vehicle will stop in a hover mode and from that static state we can always generate a trajectory if there exists a path to the final goal. In sum, our navigation pipeline is complete since the path planning algorithm we use is complete and we can always stop and re-plan. However, the trajectory generation method is not globally complete since it is constrained by the SFC.

As a conclusion, our algorithm solves the optimal trajectory inside the given SFC *w.r.t.* the given time allocation, thus the generated trajectory is not globally optimal. To guarantee feasibility, we re-optimize the trajectory using a adjusted time allocation. Practically, this method almost ensures that the resulting trajectory obeys the dynamic constraints if we set the maximum velocity, acceleration and jerk to be conservative in (3.29).

Method	***Feasibility***	***Safety***	***Optimality***	***Completeness***	***Run time***
SFC	Dynamically feasible	Collision free	Sub-optimal	Complete	Fast

Table 3.2: Evaluation of the proposed SFC method. Red blocks indicate the drawbacks of the corresponding algorithm.

3.3.4 Experimental Results

In this section, we demonstrate capability of the proposed algorithm for navigation in complex and real word environments.

Flight Speed

Here, we analyze the speed of the autonomous flight through non-dimensional parameters. We describe an MAV model by the maximum acceleration $\bar{a}_{\max}$ (constrained by the vehicle thrust to weight ratio) and the maximum velocity $\bar{v}_{\max}$ (bounded by air drag). These two parameters reflect how fast an quadrotor can travel. For different platforms, we usually have different $\bar{v}_{\max}$, $\bar{a}_{\max}$ values due to their differing hardware configurations. The planning

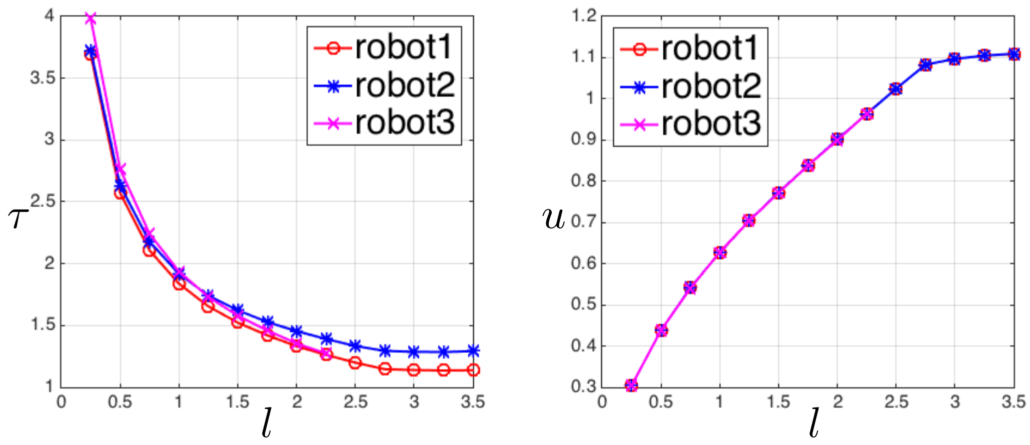


Figure 3.15: Non-dimensional analysis for 3 different quadrotors while keeping t_e fixed at 0.1 and changing l: for robot 1, $\bar{v}_{\max} = 20m/s, \bar{a}_{\max} = 10m/s^2$; for robot 2, $\bar{v}_{\max} = 10m/s, \bar{a}_{\max} = 5m/s^2$; for robot 3, $\bar{v}_{\max} = 5m/s, \bar{a}_{\max} = 5m/s^2$. The total time for reaching the goal τ and the maximum speed u goes to 1.1 with increasing l (due to sample-based method we use for trajectory optimization, the maximum speed will exceed the actual bound by a small amount), which means the longer *planning horizon* leads to a faster flight.

horizon d_r (limited by the sensing range) and execution horizon T_e (limited by the on-board computation power) are two independent variables that affect the flight speed of the vehicle in *Receding Horizon Control* (Section 2.3). They can be non-dimensionalized through normalization as:

$$l = \frac{2\bar{a}_{\max}}{\bar{v}_{\max}^2} d_r, \quad \tau_e = \frac{\bar{a}_{\max}}{\bar{v}_{\max}} T_e \tag{3.36}$$

The flight speed can be evaluated using two parameters: total time for reaching a goal T and the max speed $v_{\max}$. Suppose the total distance is d_{goal}, we are able to evaluate the nominal flight time and maximum speed using the notation as:

$$\tau = \frac{\bar{v}_{\max}}{d_{\text{goal}}} T, \quad u = \frac{v_{\max}}{\bar{v}_{\max}} \tag{3.37}$$

We plot the test results from using three different robots in simulation using these non-dimensional parameters (Figure 3.15). We can conclude that fast flight can be achieved through setting a large planning horizon. However, in the actual experiments, the planning horizon is limited by the sensing range and will not increase the flight speed after a certain threshold. The execution horizon is also limited by the on-board computation, for example in Table 3.1 the max time cost for re-plan takes up to 0.47 s which places a lower bound on T_e.

To test high speed obstacle avoidance, we simulate environments by randomly scattering N convex obstacles inside a region. A typical environment is shown in Figure 3.16. With a simulated Velodyne Puck VLP-16 of 40 m sensing range, the robot is able to achieve a max speed of 19.2 m/s in this forest and reach the goal 200 m away in 14.3 s.

FLA Test

In the FLA project, we apply the proposed navigation pipeline on the quadrotor platform shown in Figure 3.17. We use a stereo version of the MSCKF algorthm [91] with Unscented Kalman Filter [103] for state estimation and a Velodyne VLP-16 to build a local map. All the computation is performed on an on-board Intel NUC computer (dual-core i7). Figure 3.19 shows several experiments in the outdoor scenario where the robot has zero prior knowledge

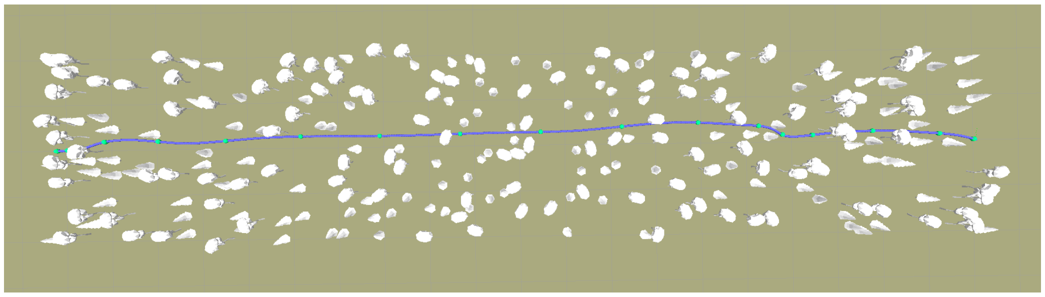


Figure 3.16: 400 trees are randomly placed onto a $200 \times 40m$ square. The RHP planning horizon is 50 m, execution horizon is 1 s. Blue curves show the robot trajectory from one end to the other. Green dot show the start position of each re-plan.

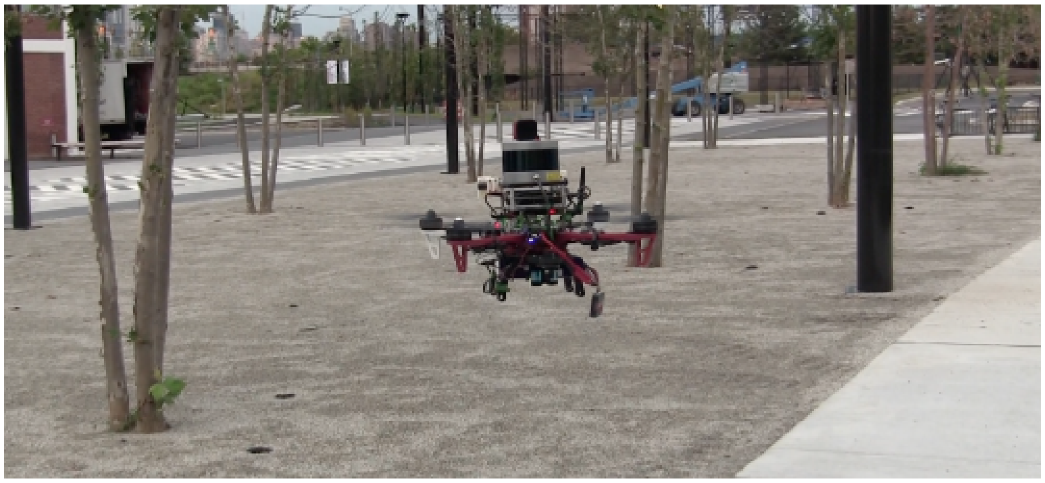


Figure 3.17: Quadrotor platform used in the FLA project.

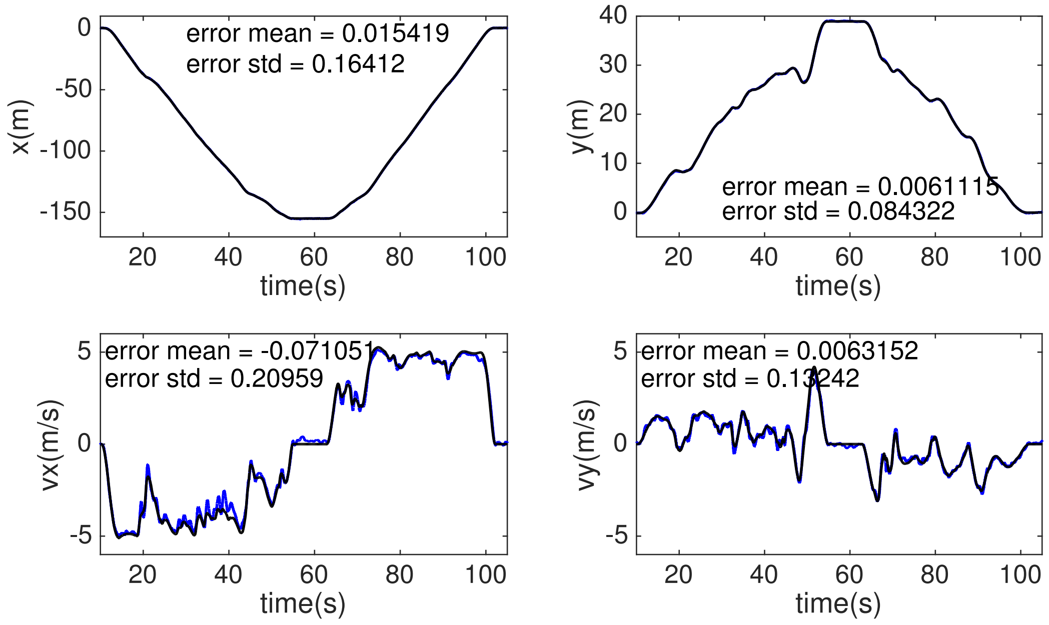
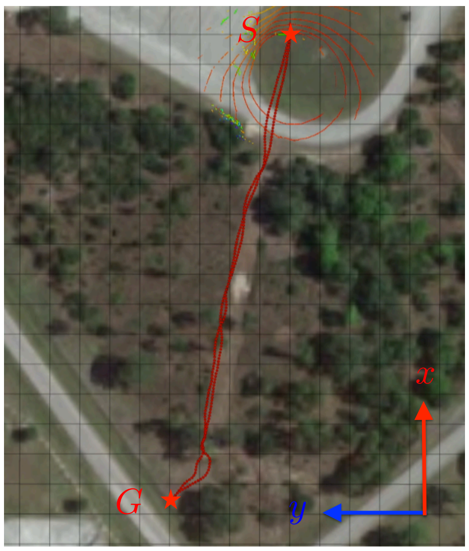


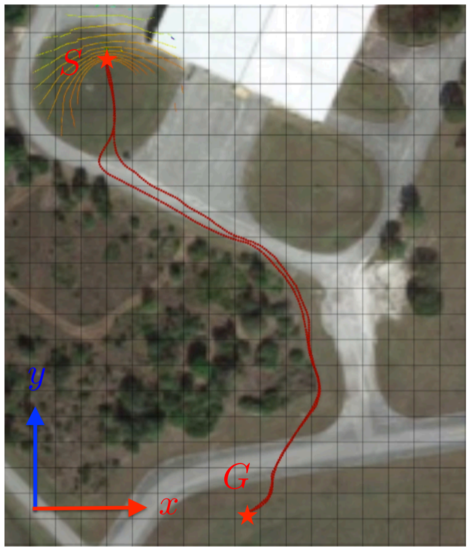
Figure 3.18: State estimation *vs* desired command for test 1. The actual robot state is marked in blue, while the desired command is shown in black.

about the environment. Given a goal with respect to initial robot position, our system can successfully reach the goal and come back without hitting any obstacle. The vehicle travels at speeds up to 5 m/s for the runs shown in Figure 3.19. In test 1, the robot successfully avoids trees and bushes with complicated 3D geometries. In test 2, since the forest is dense the robot decides to fly around it instead of flying through it. In test 3, the robot avoids trees, forests and buildings, the total distance traveled by the robot is around 1 km. Our trajectories are smooth and constrained by thresholds on velocity, acceleration and jerk which helps to decrease the error in vision-based state estimation: the general drift in position after coming back to the start position is less than 1 %.

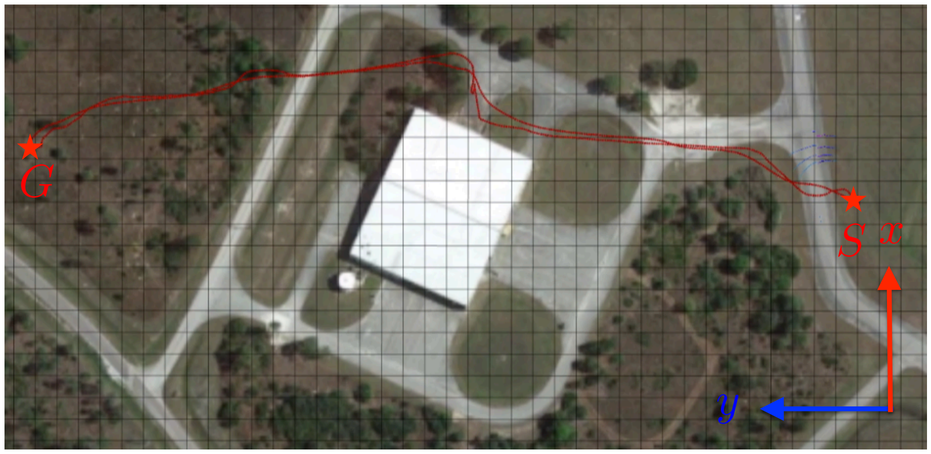
As we set the maximum acceleration to be relatively small ($3\,\mathrm{m/s^2}$), the robot is able to closely track the generated trajectories. Figure 3.18 shows the performance of the controller during test 1 and we can see that the errors are smaller than 0.2 m in position.



(a) Test 1: Goal $(-155, 39)$.



(b) Test 2: Goal $(46, -184)$.



(c) Test 3: Goal $(23, 384)$.

Figure 3.19: Outdoor experiments. The grid cell size is $10m \times 10m$. The maximum speed is set to be $5\,\mathrm{m/s}$ while we also limit the maximum acceleration as $3\,\mathrm{m/s^2}$. The 2D axes shows the direction of $x - y$ axes, the origin is located at the start point marked as a red star denoted by S.

Chapter 4

Search-based Trajectory Planning

As introduced in Chapter 3, smooth trajectories obtained by minimizing jerk or snap have been widely used to control differentially flat dynamical systems such as quadrotors. These trajectories are represented via time-parameterized polynomials, which convert the trajectory generation problem into one of finding polynomial coefficients that satisfy certain constraints. Recent work exploring time-optimal trajectory generation includes [7, 35]. If additionally, obstacle avoidance is added as a consideration, the trajectory generation problem becomes more challenging. Recent work in [9, 54, 82] demonstrate practical applications of quadratic programming to derive collision-free trajectories in real-time, including the approach proposed in Section 3.3. These methods separate the trajectory generation problem in two parts: (i) planning a collision-free geometric path and (ii) optimizing it locally to obtain a dynamically-feasible time-parameterized trajectory. In this way, one can solve for a locally optimal trajectory with respect to a given *time allocation*. However, the prior geometric path restricts the generated trajectory to be inside a given homology class which may not contain a globally optimal (or even feasible) trajectory (Figure 4.1).

In this chapter, we propose an approach for global trajectory optimization that obtains collision-free, dynamically-feasible, minimum-time, smooth trajectories in real time. Instead of using a geometric path as a prior, our approach explores the space of trajectories using a set of short-duration motion primitives generated by solving an optimal control problem. We

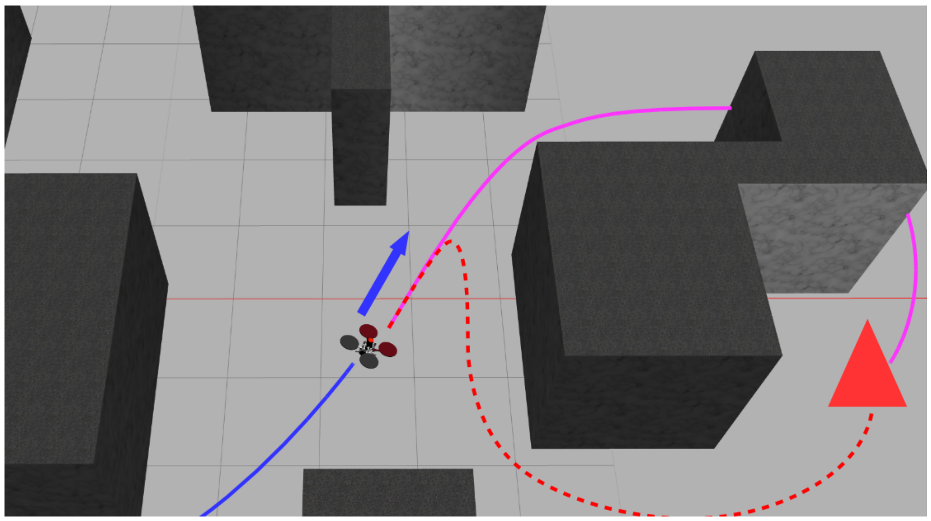


Figure 4.1: Taking the quadrotor dynamics into account is important for obtaining a smooth trajectory (magenta) while flying at non-zero velocity towards a goal (red triangle). In contrast, existing methods generate a trajectory (red dashed curve) from a shortest path that ignores the system dynamics. Instead of relying on a prior shortest path, the approach proposed in this paper plans globally-optimal trajectories based on time and control efforts.

prove that the primitives induce a finite lattice discretization on the state space, which can in turn be explored using a graph-search algorithm. It is well-known that graph search in high-dimensional state spaces is not computationally efficient because there are many states to be explored. However, with the help of a tight lower bound (heuristic) on the optimal cost we can inform and significantly accelerate the search. The main contribution of this paper is the design of a heuristic function based on the explicit solution of a Linear Quadratic Minimum Time problem. In contrast with previous works based on motion primitives like [47, 59, 76], our approach does not require a large pre-computed look-up table to find connections between different graph nodes. To reduce the run time, we propose to plan a trajectory in a lower dimension state space and refine a final trajectory that is executable by quadrotors through an unconstrained quadratic program. We also show that our method generates better trajectories compared to the traditional path-based trajectory generation approaches. We demonstrate that our approach can be used for online re-planning during fast quadrotor navigation in various cluttered environments.

4.1 Problem Formulation

Let $x(t) \in \mathcal{X} \subset \mathbb{R}^{3\times q}$ be a dynamical system state, consisting of position and its $(q-1)$ derivatives (velocity, acceleration, jerk, etc.) in 3D. Let $\mathcal{X}^{free} \subset \mathcal{X}$ denote free region of the state space that, in addition to capturing the obstacle-free positions $\mathcal{P}^{free}$, also specifies constraints $\mathcal{D}^{free}$ on the system's dynamics, *i.e.,* maximum velocity $\bar{v}_{max}$, acceleration $\bar{a}_{max}$, jerk $\bar{j}_{max}$ and higher order derivatives in each axis. Note that $\mathcal{P}^{free}$ is bounded by the size of the map that we are planning in. Thus,

$$\mathcal{X}^{free} = \mathcal{P}^{free} \times \mathcal{D}^{free}, \tag{4.1}$$

$$\mathcal{D}^{free} = [-\bar{v}_{max},\, \bar{v}_{max}]^3 \times [-\bar{a}_{max},\, \bar{a}_{max}]^3 \times [-\bar{j}_{max},\, \bar{j}_{max}]^3. \tag{4.2}$$

And we denote the obstacle region as $\mathcal{X}^{obs} = \mathcal{X} \setminus \mathcal{X}^{free}$.

As described in [63] and many other related works, the differential flatness of quadrotor systems allow us to construct control inputs from 1D time-parameterized polynomial trajectories specified independently in each of the three position axes. Thus, we consider polynomial state trajectories $x(t) := [p_D(t)^\mathsf{T}, \dot{p}_D(t)^\mathsf{T}, \ldots, p_D^{(q-1)}(t)^\mathsf{T}]^\mathsf{T}$, where

$$p_D(t) = \sum_{i=0}^{n} d_i \frac{t^i}{i!} = d_n \frac{t^n}{n!} + \ldots + d_1 t + d_0 \in \mathbb{R}^3 \tag{4.3}$$

and $D = [d_0 \ldots d_n] \in \mathbb{R}^{3\times(n+1)}$. To simplify the notation, we denote the system's velocity by $v(t) = \dot{p}_D^\mathsf{T}(t)$, acceleration by $a(t) = \ddot{p}_D^\mathsf{T}(t)$, jerk by $j(t) = \dddot{p}_D^\mathsf{T}(t)$, etc., and drop the subscript D where convenient. Polynomial trajectories of the form (4.3) can be generated by considering a linear time-invariant dynamical system $p_D^{(q)}(t) = u(t)$, where the control input is $u(t) \in \mathcal{U} = [-u_{max}, u_{max}]^3 \subset \mathbb{R}^3$. In state space form, we obtain a system as

$$\dot{x} = Ax + Bu$$
$$A = \begin{bmatrix} \mathbf{0} & \mathbf{I}_3 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_3 & \cdots & \mathbf{0} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \mathbf{0} & \cdots & \cdots & \mathbf{0} & \mathbf{I}_3 \\ \mathbf{0} & \cdots & \cdots & \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad B = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{I}_3 \end{bmatrix} \tag{4.4}$$

We are interested in planning state trajectories that are *collision-free*, respect the constraints on the dynamics, and are *minimum-time* and *smooth*. We define the *smoothness* or *effort* of a trajectory as the square L^2-norm of the control input $u(t)$:

$$J(D) = \int_0^T \|u(t)\|^2 \, dt = \int_0^T \left\| p_D^{(q)}(t) \right\|^2 dt \tag{4.5}$$

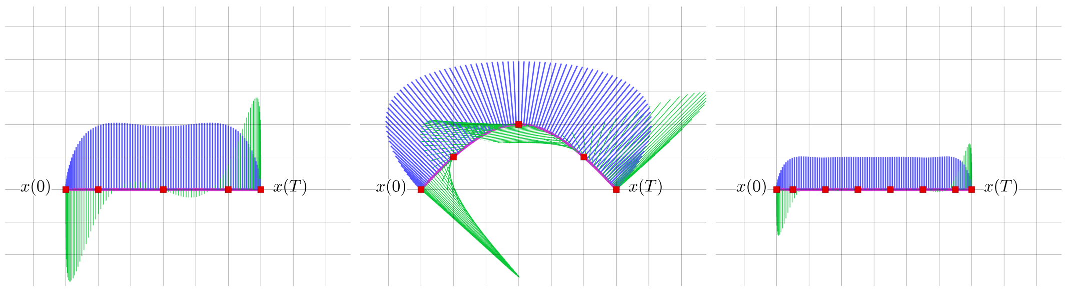
and consider the following problem.

Problem 1. *Given an initial state $x_0 \in \mathcal{X}^{free}$ and a goal region $\mathcal{X}^{goal} \subset \mathcal{X}^{free}$, find a polynomial trajectory parametrization $D \in \mathbb{R}^{3\times(n+1)}$ and a time $T \geq 0$ such that:*

$$\begin{aligned} &\arg\min_{D,T} \; J(D) + \rho T \\ s.t. \;\; &\dot{x}(t) = Ax(t) + Bu(t), \quad \forall\, t \in [0,\; T] \\ &x(0) = x_0, \quad x(T) \in \mathcal{X}^{goal} \\ &x(t) \in \mathcal{X}^{free}, \quad u(t) \in \mathcal{U}, \quad \forall\, t \in [0,\; T] \end{aligned} \tag{4.6}$$

where the parameter $\rho \geq 0$ determines the relative importance of the trajectory duration T versus its smoothness J.

We denote the optimal cost from an initial state x_0 to a goal region $\mathcal{X}^{goal}$ by $C^*\big(x_0, \mathcal{X}^{goal}\big)$. The reason for choosing such an objective function is illustrated in Figure 4.2. This problem is a Linear Quadratic Minimum-Time problem [102] with state constraints, $x(t) \in \mathcal{X}^{free}$,



(a) $T = 4, J = 19$. (b) $T = 4, J = 48$. (c) $T = 7, J = 5$.

Figure 4.2: Three trajectories start from $x(0)$ to $x(T)$. Blue and green rays indicate the magnitude of velocity and acceleration along trajectories respectively. If the effort J is disregarded, *i.e.,* $\rho \to \infty$ in (4.6), trajectories (a) and (b) have equivalent cost of $T = 4$. If the time T is not considered, *i.e.,* $\rho = 0$, trajectory (c) become optimal. Since we are interested in low-effort trajectories, ρ should not be infinite (so that (a) is preferable to (b)) but it should still be large enough to prioritize fast trajectories. Thus, in this comparison, (a) is preferable to both (b) and (c).

and input constraints, $u(t) \in \mathcal{U}$. As the derivation in Section 4.3.1 shows, if we drop the constraints $x(t) \in \mathcal{X}^{free}, u(t) \in \mathcal{U}$, the optimal solution can be obtained via Pontryagin's minimum principle [45, 102] and the optimal choice of polynomial degree is $n = 2q - 1$. The main challenge is the introduction of the constraints $x(t) \in \mathcal{X}^{free}, u(t) \in \mathcal{U}$. In this chapter, we show that these safety constraints can be handled by converting the problem to a deterministic shortest path problem [5, Ch.2] with a $3 \times q$ dimensional state space $\mathcal{X}$ and a 3 dimensional control space $\mathcal{U}$. Since the control space $\mathcal{U}$ is always 3 dimensional, a search-based planning algorithm such as A^* [48] that discretizes $\mathcal{U}$ using *motion primitives* is efficient and resolution-complete (*i.e.,* , it can compute the optimal trajectory in the discretized space in finite-time, unlike sampling-based planners such as RRT [36, 41]).

4.2 Motion Primitives and Induced Graph

4.2.1 Motion Primitives

First, we discuss the construction of motion primitives for the system in (4.4) that will allow us to convert Problem 1 from an optimal control problem to a graph-search problem. Instead of using the control set $\mathcal{U}$, we consider a lattice discretization [75] $\mathcal{U}_M = \{u_1, \ldots, u_M\} \subset \mathcal{U}$, where each control $u_m \in \mathbb{R}^3$ vector will define a motion of short duration for the system. One way to obtain the discretization $\mathcal{U}_M$ is to choose a number of samples $z \in \mathbb{Z}^+$ along each axis

$[0,\ u_{max}]$, which defines a discretization step $du = \frac{u_{max}}{z}$ and results in $M = (2z+1)^3$ motion primitives. Given an initial state $x_0 = [p_0^\intercal,\ v_0^\intercal,\ a_0^\intercal, \dots]^\intercal$, we generate a motion primitive of duration $\tau > 0$ that applies a constant control input $u(t) \equiv u_m \in \mathcal{U}_M$ for $t \in [0, \tau]$ so that:

$$u(t) = p_D^{(q)}(t) = \sum_{i=0}^{n-q} d_{q+i} \frac{t^i}{i!} \equiv u_m. \tag{4.7}$$

The control input being constant implies that all coefficients that involve time need to be identically zero, *i.e.,*

$$d_{(q+1):n} = \mathbf{0} \implies u_m = d_n. \tag{4.8}$$

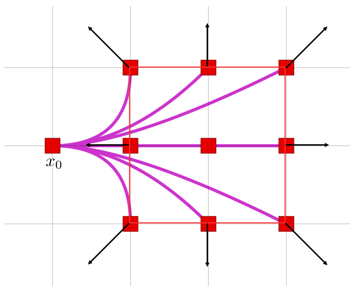
Integrating the control expression $u(t) = u_m$ with an initial condition x_0 results in

$$p_D(t) = u_m \frac{t^q}{q!} + \dots + a_0 \frac{t^2}{2} + v_0 t + p_0. \tag{4.9}$$

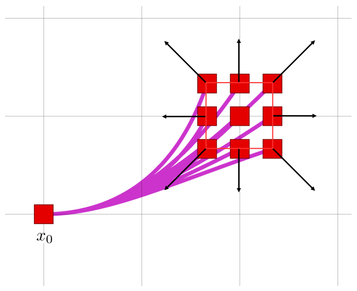
Or, equivalently, the resulting trajectory of the linear time-invariant system in (4.4) is:

$$x(t) = \underbrace{e^{At}}_{F(t)} x_0 + \underbrace{\left[\int_0^t e^{A(t-\sigma)} B d\sigma \right]}_{G(t)} u_m. \tag{4.10}$$

An example of the resulting system trajectories is given in Figure 4.3. Since both the



(a) Discretized Acceleration.



(b) Discretized Jerk.

Figure 4.3: Example of 9 planar motion primitives from initial state x_0 for an acceleration-controlled ($n = 2$) system (left) and a jerk-controlled ($n = 3$) system (right). The black arrow indicates correpsonding control input. The red boundary shows the feasible region for the end states (red squares), which is induced by the control limit u_{max}. The initial velocity and acceleration are $v_0 = [1, 0, 0]^\intercal$ and $a_0 = [0, 1, 0]^\intercal$ (only for the right figure).

duration τ and the control input u_m are fixed, the cost of the motion primitive according to (4.6) is

$$\left(\|u_m\|^2 + \rho\right)\tau. \tag{4.11}$$

4.2.2 Graph Construction

Starting at an initial state x_0, we can apply all primitives in $\mathcal{U}_M$ to obtain the M possible states after the duration τ (Figure 4.3). Repeating this process iteratively, we can build a graph, denoted as $\mathcal{G}(\mathcal{S}, \mathcal{E})$, where $\mathcal{S}$ is the discrete set of reachable system states and $\mathcal{E}$ is the set of edges that connect states in the graph, each defined by a motion primitive $e := (u_m, \tau)$. Let s_0 be the state corresponding to x_0.

Algorithm 4 shows the pseudo-code for state propagation and can be used to explore the free state space $\mathcal{X}^{free}$ and build the connected graph: in line 4, the primitive is calculated using the fully defined state s and a control input u_m given the constant time τ; line 5 checks the feasibility of the primitive, this step will be further discussed in Section 4.3.2; in line 6, we evaluate the end state of a valid primitive and add it to the set of successors of the current node; in the meanwhile, we estimate the edge cost from the corresponding primitive. After checking through all the primitives in the finite control input set, we add the nodes in successor set $\mathcal{R}(s)$ to the graph, and we continue expanding until we reach the goal region.

Proposition 1. *The motion primitive $u_{ij} \in \mathcal{U}_M$ which connects two consecutive states $s_i, s_j \in \mathcal{S}$ with $s_j = F(\tau)s_i + G(\tau)u_{ij}$ is optimal according to the cost function in* (4.6).

Proof of Proposition 1. Since the trajectory connecting s_i and s_j is collision-free by construction of the graph $\mathcal{G}$ (see Algorithm 4), the optimal control from s_i to s_j according to the cost function in (4.6) has the form prescribed by Proposition 3. In detail

$$\delta_\tau = s_j - F(\tau)s_i = G(\tau)u_{ij}, \tag{4.12}$$

Algorithm 4 Given $s \in \mathcal{S}$ and a motion primitive set $\mathcal{U}_M$ with duration τ, find the states $\mathcal{R}(s)$ that are reachable from s in one step and their associated costs $\mathcal{C}(s)$.

```
1: function GetSuccessors($s$,$\mathcal{U}_M$, $\tau$)
2:     $\mathcal{R}(s) \leftarrow \emptyset$;
3:     $\mathcal{C}(s) \leftarrow \emptyset$;
4:     for all $u_m \in \mathcal{U}_M$ do
5:         $e_m(t) \leftarrow F(t)s + G(t)u_m,\ t \in [0, \tau]$;
6:         if $e_m(t) \subset \mathcal{X}^{free}$ then
7:             $s_m \leftarrow e_m(\tau)$;
8:             $\mathcal{R}(s) \leftarrow \mathcal{R}(s) \cup \{s_m\}$;
9:             $\mathcal{C}(s) \leftarrow \mathcal{C}(s) \cup \{(\|u_m\|^2 + \rho)\tau\}$;
10:        end if
11:    end for
12:    return $\mathcal{R}(s), \mathcal{C}(s)$;
13: end function
```

and the optimal control is:

$$\begin{aligned} u^*(t) &= B^\mathsf{T} e^{A^\mathsf{T}(\tau - t)} W_\tau \delta_\tau \\ &= B^\mathsf{T} \underbrace{e^{A^\mathsf{T}(\tau-t)} \left(\int_0^\tau e^{As} B B^\mathsf{T} e^{A^\mathsf{T} s} ds \right)^{-1} \int_0^\tau e^{As} ds\, B}_{C} u_{ij} \end{aligned} \tag{4.13}$$

Since only the bottom 3×3 block of B is non-zero and the matrix C has its bottom-right 3×3 block equal to $I_{3\times3}$, we get:

$$B^\mathsf{T} e^{A^\mathsf{T}(\tau-t)} \left(\int_0^\tau e^{As} B B^\mathsf{T} e^{A^\mathsf{T} s} ds \right)^{-1} \int_0^\tau e^{As} ds B = I_{3\times3} \tag{4.14}$$

which implies that $u^*(t) \equiv u_{ij}$. □

4.2.3 Induced Space Discretization

The motion primitives generated using (4.9) will build a state space that is discretized. Before solving Problem 1, we look into some propertied of the induced state space.

Proposition 2. *The motion primitives defined as* (4.9) *induce a discretization on the state space* $\mathcal{X}$.

Proof of Proposition 2. Since all the flat outputs are independent from each other, we can equivalently consider 1D case with out loss of generality. Given an initial state x_0 and a sequence of k inputs, $u_1, \dots, u_k$, are applied each for time τ. The final state up to $(q-1)$-th derivative after applying the k inputs is given by,

$$\begin{aligned} x(k\cdot\tau) &= F^k(\tau)x_0 + \sum_{i=0}^{k-1} F^i(\tau)G(\tau)u_{k-i}, \\ F^k(\tau) &= \begin{bmatrix} 1 & k\tau & \cdots & \frac{(k\tau)^{q-1}}{(q-1)!} \\ 0 & 1 & \cdots & \frac{(k\tau)^{q-2}}{(q-2)!} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}, \\ F^i(\tau)G(\tau) &= \begin{bmatrix} [(i+1)^q - i^q]\frac{\tau^q}{q!} \\ \left[(i+1)^{q-1} - i^{q-1}\right]\frac{\tau^{q-1}}{(q-1)!} \\ \vdots \\ \tau \end{bmatrix}. \end{aligned} \tag{4.15}$$

Our discretized inputs are of the form $u_i = \kappa_i \cdot du$ where $\kappa_i \in \mathbb{Z}$ leading to $x(k\tau)$ being of the form

$$x(k\cdot\tau) = F^k(\tau)x_0 + \begin{bmatrix} \frac{\tau^q}{q!}\sum_{i=0}^{k-1}[(i+1)^q - i^q]\kappa_{k-i} \\ \frac{\tau^{q-1}}{(q-1)!}\sum_{i=0}^{k-1}\left[(i+1)^{q-1} - i^{q-1}\right]\kappa_{k-i} \\ \vdots \\ \tau\sum_{i=0}^{k-1}\kappa_{k-i} \end{bmatrix} du \tag{4.16}$$

Thus we can see that each term in the expression for $x(k\tau)$ is a variable integer times a constant which means that our state space is discretized due to discretization of the inputs. $\square$

Consider an second-order system in 1D for which $q = 2$. Apply the control sequence

$u_1, \ldots, u_k$, from (4.15) we get the system's end state:

$$x(k \cdot \tau) = \begin{bmatrix} 1 & k\tau \\ 0 & 1 \end{bmatrix} x_0 + \begin{bmatrix} \frac{1}{2}\tau^2 \\ \tau \end{bmatrix} u_k + \begin{bmatrix} \frac{3}{2}\tau^2 \\ \tau \end{bmatrix} u_{k-1} + \ldots + \begin{bmatrix} \frac{2k+1}{2}\tau^2 \\ \tau \end{bmatrix} u_1. \tag{4.17}$$

Let the initial state be $x_0 = [\ p_0,\ v_0\ ]^\mathsf{T}$, consider two arbitrary states $x(k_1\tau)$, $x(k_2\tau)$ which are derived from the control sequence $u_1^1, \ldots, u_{k_1}^1$ and $u_1^2, \ldots, u_{k_2}^2$ from x_0:

$$\begin{aligned} x(k_1\tau) &= \begin{bmatrix} p_0 + k_1\tau v_0 \\ v_0 \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\tau^2 \\ \tau \end{bmatrix} u_{k_1}^1 + \begin{bmatrix} \frac{3}{2}\tau^2 \\ \tau \end{bmatrix} u_{k_1-1}^1 + \ldots + \begin{bmatrix} \frac{2k_1+1}{2}\tau^2 \\ \tau \end{bmatrix} u_1^1, \\ x(k_2\tau) &= \begin{bmatrix} p_0 + k_2\tau v_0 \\ v_0 \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\tau^2 \\ \tau \end{bmatrix} u_{k_2}^2 + \begin{bmatrix} \frac{3}{2}\tau^2 \\ \tau \end{bmatrix} u_{k_2-1}^2 + \ldots + \begin{bmatrix} \frac{2k_2+1}{2}\tau^2 \\ \tau \end{bmatrix} u_1^2. \end{aligned} \tag{4.18}$$

The difference between these two states is:

$$dx = x(k_1\tau) - x(k_2\tau). \tag{4.19}$$

Let du be the discretization step in control, it is obvious that the velocity is discretized with resolution $\tau \cdot du$ from its analytic formulation:

$$\begin{aligned} dx(1) &= \tau\big[(u_1^1 + \ldots + u_{k_1}^1) - (u_1^2 + \ldots + u_{k_2}^2)\big] \\ &= \tau du \cdot \underbrace{\big[(\kappa_1^1 + \ldots + \kappa_{k_1}^1) - (\kappa_1^2 + \ldots + \kappa_{k_2}^2)\big]}_{\kappa_v} \end{aligned} \tag{4.20}$$

where $\kappa_v \in \mathbb{Z}$ can be any integer according to the arbitrary select of control inputs.

The position discretization is more complicated, we look into the analytic expression for difference in position at first:

$$\begin{aligned}dx(0) &= (k_1 - k_2)\tau v_0 + (\frac{1}{2}\tau^2 u^1_{k_1} + \ldots + \frac{2k_1+1}{2}\tau^2 u^1_1) - (\frac{1}{2}\tau^2 u^1_{k_2} + \ldots + \frac{2k_2+1}{2}\tau^2 u^2_1) \\ &= (k_1 - k_2)\tau v_0 + \frac{\tau^2}{2} du \cdot \underbrace{\left[(\kappa^1_{k_1} + \ldots + (2k_1+1)\kappa^1_1) - (\kappa^1_{k_2} + \ldots + (2k_2+1)\kappa^2_1)\right]}_{\kappa_p}\end{aligned} \tag{4.21}$$

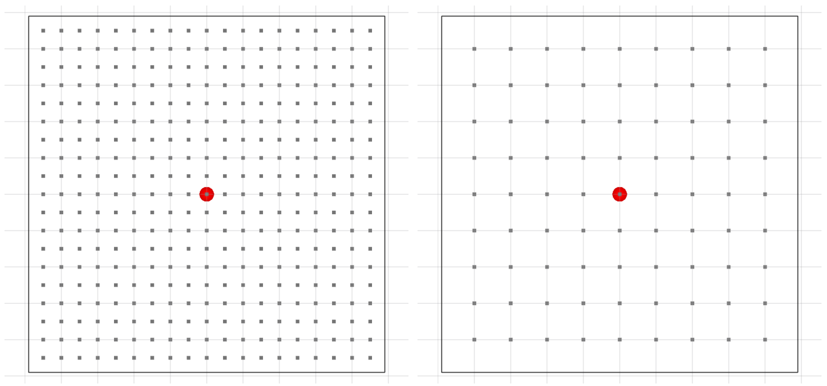
where $\kappa_p \in \mathbb{Z}$ can be also any integer similar to κ_v except that its value is related to κ_v. Substitute (4.20) into (4.21) we get

$$dx(0) = (k_1 - k_2)\tau v_0 + \frac{\tau}{2} dx(1) + \tau^2 du \underbrace{\left[(\kappa_{k_1-1} + \ldots + k_1\kappa^1_1) - (\kappa_{k_2-1} + \ldots + k_2\kappa^2_1)\right]}_{\kappa'_p}. \tag{4.22}$$

where $\kappa'_p \in \mathbb{Z}$ is independent from κ_v. Thus, we can see that the position discretization is formed from three objects:

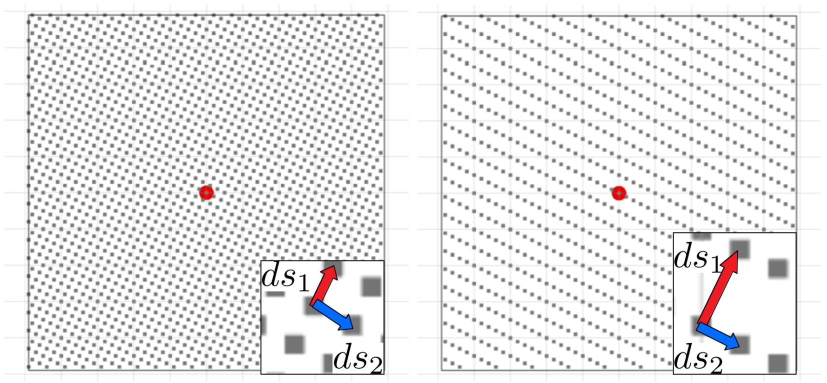
1. initial velocity term $(k_1 - k_2)\tau v_0$ which refers to resolution of τv_0;
2. velocity discretization term $\frac{\tau}{2} dx(1)$ which refers to resolution of $\frac{1}{2}\tau^2 du$;
3. term relates to κ'_p which refers to resolution of $\tau^2 du$.

Each of these three terms will create an uniformed grid with associate resolution, but the overall position grid is not simply the overlapped version of these three grids: the minimum distance between any two states in the grid is not the minimum value of $\{\tau v_0,\ \frac{1}{2}\tau^2 du,\ \tau^2 du\}$, but it is the minimum value of the permutation and combination of those values. We show two 2D examples with the same time and control discretizations as $\tau = 1,\ du = 1$ but different initial velocity in Figure 4.4 and 4.5. When the initial velocity is zero ($v_0 = [0,\ 0]^\mathsf{T}$), the position discretization only relates to τ and du according to (4.22). The minimum distance of the overall grid in Figure 4.4a is $\frac{1}{2}\tau^2 du = 0.5$. We are also interested in the position discretization considering the velocity constraints. Denote dv as the difference between state's velocity and the initial velocity v_0, we only plot the states that have the same velocity as v_0 in Figure 4.4b (zero velocity). In this case, it is equivalent to set $dx(1) = 0$ in (4.22) as, and the resolution of the grid is simply decided by the third term as $\tau^2 du = 1$.



(a) Overall Discretization. (b) Discretization with $dv = \mathbf{0}$.

Figure 4.4: An example of position discretization in 2D of a second-order system. The red dot indicates the initial position p_0. Grey dots are the positions of the induced grid. The grid is generated from $x_0 = [\ 0,\ 0,\ 0,\ 0\ ]^\mathsf{T}$.



(a) Overall Discretization. (b) Discretization with $dv = \mathbf{0}$.

Figure 4.5: An example of position discretization in 2D of a second-order system generated from $x_0 = [\ 0,\ 0,\ 0.4,\ 0.3\ ]^\mathsf{T}$. The bottom right rectangle shows the zoomed-in grid and the direction of ds_1, ds_2.

In Figure 4.5, the initial state has non-zero velocity as $v_0 = [0.4,\ 0.3]^\mathsf{T}$. Figure 4.5a shows the overall grid, of which the difference ds between two closest points is $ds_1 = [\ 0.1,\ 0.2\ ]^\mathsf{T}$ and $ds_2 = [0.2,\ \ -0.1]^\mathsf{T}$. ds can be described by $\tau v_0 = [\ 0.4,\ 0.3\ ]^\mathsf{T}$, $\frac{1}{2}\tau^2 du = 0.5$ and $\tau^2 du = 1$ as:

$$ds_1 = -v_0 + \frac{1}{2}\tau^2[du,\ du]^\mathsf{T},\ ds_2 = 3v_0 - \tau^2[du,\ du]^\mathsf{T}. \tag{4.23}$$

Figure 4.5b only plots the states with velocity equal to $v_0 = [\ 0.4,\ 0.3\ ]^\mathsf{T}$. Thus the $dx(1)$ term in (4.22) is zero, and the minimum difference in position depends on $\tau v_0 = [\ 0.4,\ 0.3\ ]^\mathsf{T}$ and $\tau^2 du = 1$. The discretization in Figure 4.5b are $ds_1 = [\ 0.2,\ 0.4\ ]^\mathsf{T}$ and $ds_2 = [\ 0.2,\ -0.1\ ]^\mathsf{T}$ which can be derived from:

$$ds_1 = -2v_0 + \tau^2[\ du,\ du\ ]^\mathsf{T},\ ds_2 = 3v_0 - \tau^2[\ du,\ du\ ]^\mathsf{T}. \tag{4.24}$$

From the above two examples, we can conclude that the induced state lattices are discretized in position, velocity and higher order derivatives. The general grid in position is not uniformed (*e.g.,* Figure 4.5) except the special case where the initial state is static. We are able to estimate the discretization using (4.22). We can estimate the induced grid for higher order system with $q > 2$ using the same approach, which is omitted in this paper. In addition, since the space $\mathcal{X}$ and dynamics are bounded and the induced state is discretized, the induced graph is finite.

4.3 Deterministic Shortest Trajectory

Given the set of control inputs $\mathcal{U}_M$ and the induced space discretization discussed in the previous section, we can re-formulate Problem 1 as a graph-search problem. This can be done by introducing additional constraints that stipulate that the control input $u(t)$ in (4.6) is piece-wise constant over intervals of duration τ. More precisely, we introduce an additional variable $N \in \mathbb{Z}^+$, such that $T = N\tau$, and $u_k \in \mathcal{U}_M$ for $k = 0, \ldots, N-1$ and a constraint in (4.6):

$$u(t) = \sum_{k=0}^{N-1} u_k \mathbb{1}_{\{t\in[k\tau,(k+1)\tau)\}}$$

that forces the control trajectory to be a composition of the motion primitives in $\mathcal{U}_M$. This leads to the following deterministic shortest path problem [5, Ch.2].

Problem 2. *Given an initial state $x_0 \in \mathcal{X}^{free}$, a goal region $\mathcal{X}^{goal} \subset \mathcal{X}^{free}$, and a finite set of motion primitives $\mathcal{U}_M$ with duration $\tau > 0$, choose a sequence of control inputs $u_{0:N-1}$ of* *length N such that:*

$$\begin{aligned}
&\min_{N, u_{0:N-1}} \left(\sum_{k=0}^{N-1} \|u_k\|^2 + \rho N \right) \tau \\
s.t.\ & x_k(\tilde{t}) = F(\tilde{t}) s_k + H(\tilde{t}) u_k \subset \mathcal{X}^{free}, \quad \tilde{t} \in [0, \tau] \\
& x_k(\tilde{t}) \subset \mathcal{X}^{free} \quad \forall k \in \{0, \dots, N-1\},\ \tilde{t} \in [0, \tau] \\
& s_{k+1} = x_k(\tau), \quad \forall k \in \{0, \dots, N-1\} \\
& s_0 = x_0, \quad s_N \in \mathcal{X}^{goal} \\
& u_k \in \mathcal{U}_M, \quad \forall k \in \{0, \dots, N-1\}
\end{aligned} \tag{4.25}$$

The optimal cost of Problem 2 is an upper bound to the optimal cost of Problem 1 because Problem 2 is just a constrained version of Problem 1. However, this re-formulation to discrete control and state-spaces enables an efficient solution. Such problems can be solved via search-based [28, 48] or sampling-based [3, 36, 41] motion planning algorithms. Since only the former guarantees finite-time (sub-)optimality, we use an A^* method and focus on the design of an accurate, consistent heuristic and efficient, guaranteed collision checking methods in following subsections.

4.3.1 Heuristic Function Design

Devising an efficient graph search for solving Problem 2 requires an approximation of the optimal cost function, i.e., a heuristic function, that is admissible[1], informative (i.e., provides a tight approximation of the optimal cost), and consistent[2] (i.e., can be inflated in order to obtain solutions with bounded suboptimality very efficiently [48]). Since by construction, the optimal cost of Problem 2 is bounded below by the optimal cost of Problem 1, we can obtain a good heuristic function by solving a relaxed version of Problem 1. Our idea is to replace constraints in (4.6) that are difficult to satisfy, namely, $x(t) \in \mathcal{X}^{free}$ and $u(t) \in \mathcal{U}$, with a constraint on the time T. In this section, we show that such a relaxation of Problem 1

[1]A heuristic function h is *admissible* if it underestimates the optimal cost-to-go from x_0, i.e., $0 \leq h(x_0) \leq C^*\big(x_0, \mathcal{X}^{goal}\big), \forall x_0 \in \mathcal{X}$.

[2]A heuristic function h is *consistent* if it satisfies the triangle inequality, i.e., $h(x_0) \leq C^*(x_0, \{x_1\}) + h(x_1), \forall x_0, x_1 \in \mathcal{X}$.

can be solved optimally and efficiently.

Minimum Time Heuristic

Intuitively, the constraints on maximum velocity, acceleration, jerk, etc. due to $\mathcal{X}^{obs}$ and $\mathcal{U}$ induce a lower bound $\bar{T}$ on the minimum achievable time in (4.6). For example, since the system's maximum velocity is bounded by v_{max} along each axis, the minimum time for reaching the closest state x_f in the goal region $\mathcal{X}^{goal}$ is bounded below by $\bar{T}_v := \frac{\|p_f - p_0\|_\infty}{v_{max}}$. Similarly, since the system's maximum acceleration is bounded by a_{max}, the state $x_f := [p_f^\intercal, v_f^\intercal]^\intercal$ cannot be reached faster than:

$$\begin{aligned}
&\min_{\bar{T}_a, a(t)} \;\; \bar{T}_a \\
\text{s.t.} \quad & \|a(t)\| \leq a_{max}, \quad \forall\, t \in [0, T] \\
& p(0) = p_0,\; v(0) = v_0 \\
& p(\bar{T}_a) = p_f,\; v(\bar{T}_a) = v_f
\end{aligned}$$

The above is a minimum-time (Brachistochrone) optimal control problem with input constraints, which may be difficult to solve directly in 3D [18] but can be solved in closed-form along individual axes [45, Ch.5] to obtain lower bounds $\bar{T}_a^x$, $\bar{T}_a^y$, $\bar{T}_a^z$. This procedure can be continued for the constraint on jerk j_{max} and those on higher-order derivatives but the problems become more complicated to solve and the computed times are less likely to provide better bounds the previous ones. Hence, we can define a lower bound on the minimum achievable time via $\bar{T} := \max\{\bar{T}_v, \bar{T}_a^x, \bar{T}_a^y, \bar{T}_a^z, \bar{T}_j, \ldots\}$ but for simplicity we use the easily computable but less tight bound $\bar{T} = \bar{T}_v$.

Hence, to find a heuristic function, we relax Problem 1 by replacing the state and input constraints, $x(t) \in \mathcal{X}^{free}$ and $u(t) \in \mathcal{U}$, with the lower bound $T \geq \bar{T}_v$:

$$\begin{aligned} &\min_{D,T} \; J(D) + \rho T \\ \text{s.t. } &\dot{x}(t) = Ax(t) + Bu(t), \quad \forall t \in [0,T] \\ &x(0) = x_0, \quad x(T) \in \mathcal{X}^{goal} \\ &T \geq \bar{T} \end{aligned} \tag{4.26}$$

Since $J(D) \geq 0$, a straight-forward way to obtain a lower-bound on the optimal cost is:

$$C^*\left(x_0, \mathcal{X}^{goal}\right) = J(D^*) + \rho T^* \geq \rho \bar{T}_v$$

Hence, given nodes $s_0, s_f \in \mathcal{S}$ in the discretized space, the following is an admissible heuristic function:

$$h_1(s_0) = \rho \bar{T}_v = \frac{\rho \|p_f - p_0\|_\infty}{v_{max}} \tag{4.27}$$

for Problem 2. It is easy to see that it is also *consistent* due to the triangle inequality for distances.

Linear Quadratic Minimum Time

While the minimum-time heuristic is very easy to compute and takes velocity constraints into account, it is not a very tight lower bound on the optimal cost in (4.25) because it disregards the control effort. The reason is that instead of solving (4.26), we simply found a lower bound in the previous subsection. An important observation is that after removing the constraints $x(t) \in \mathcal{X}^{free}$ and $u(t) \in \mathcal{U}$, the relaxed problem (4.26) is in fact the classical Linear Quadratic Minimum-Time Problem [102]. The optimal solution to (4.26) can be obtained from [102, Thm.2.1] with a minor modification introducing the additional constraint on time $T \geq \bar{T}$.

Proposition 3. *Let $x_f \in \mathcal{X}^{goal}$ be a fixed final state and define $\delta_T := x_f - e^{AT}x_0$ and the controllability Gramian $W_T := \int_0^T e^{At}BB^\mathsf{T}e^{A^\mathsf{T}t}dt$. Then, the optimal time T in (4.26) is* *either the lower bound $\bar{T}$ or the solution of following equation:*

$$-\frac{d}{dT}\left\{\delta_T^\top W_T^{-1}\delta_T\right\} = 2x_f^\top A^\top W_T^{-1}\delta_T + \delta_T^\top W_T^{-1} B B^\top W_T^{-1}\delta_T = \rho \tag{4.28}$$

The optimal control is:

$$u^*(t) := B^\top e^{A^\top (T-t)} W_T^{-1}\delta_T \tag{4.29}$$

While the optimal cost is:

$$h_2(x_0) = \delta_T^\top W_T^{-1}\delta_T + \rho T \tag{4.30}$$

The polynomial coefficients $D \in \mathbb{R}^{3\times(2n)}$ in (4.3) are:

$$d_{0:(n-1)} = x_0, \qquad d_{n:(2n-1)} = \delta_T^\top W_T^{-\top} e^{AT} H^\top$$

where $H \in \mathbb{R}^{(3n)\times(3n)}$ with $H_{ij} = \begin{cases} (-1)^j, & i = j \\ 0, & i \neq j \end{cases}$.

Thus, the optimal cost $h_2(x_0)$ obtained in Proposition 3 is a better heuristic for Problem 2 than h_1 because h_2 takes the control efforts into account. It is also admissible by construction because the optimal cost of Problem 2 is lower bounded by the optimal cost of Problem 1, which in turn is lower bounded by $h_2(x_0)$. Below, we give examples of the results in Proposition 3 for several practical cases with a fixed value of T.

Velocity Control Let $n = 1$ so that $\mathcal{X} \subset \mathbb{R}^3$ is position space and $\mathcal{U}$ is velocity space. Then, the optimal solution to (4.26) according to Proposition 3 is:

$$d_1 = \frac{1}{T}(x_f - x_0),$$
$$x^*(t) = d_1 t + x_0, \; u^*(t) = d_1,$$
$$C^* = \frac{1}{T}\|x_f - x_0\|^2 + \rho T.$$

Acceleration Control Let $n = 2$ so that $\mathcal{X} \subset \mathbb{R}^6$ is position-velocity space and $\mathcal{U}$ is acceleration space. Then, the optimal solution to (4.26) according to Proposition 3 is:

$$\begin{aligned}
\begin{pmatrix} d_2 \\ d_3 \end{pmatrix} &= \begin{bmatrix} \frac{6}{T^2} & -\frac{2}{T} \\ -\frac{12}{T^3} & \frac{6}{T^2} \end{bmatrix} \begin{bmatrix} p_f - p_0 - v_0 T \\ v_f - v_0 \end{bmatrix}, \\
x^*(t) &= \begin{bmatrix} \frac{d_3}{6} t^3 + \frac{d_2}{2} t^2 + v_0 t + x_0 \\ \frac{d_3}{2} t^2 + d_2 t + v_0 \end{bmatrix}, \; u^*(t) = d_3 t + d_2, \\
C^* &= \frac{12\|p_f - p_0\|^2}{T^3} - \frac{12(v_0 + v_f) \cdot (p_f - p_0)}{T^2} + \\
&\quad \frac{4(\|v_0\|^2 + v_0 \cdot v_1 + \|v_1\|^2)}{T} + \rho T.
\end{aligned}$$

4.3.2 Collision Checking

For a calculated edge $e(t) = [p(t)^\mathsf{T}, v(t)^\mathsf{T}, a(t)^\mathsf{T}, \ldots]^\mathsf{T}$ in Algorithm 4, we need to check if $e(t) \subset \mathcal{X}^{free}$ for $t \in [0, \tau]$. We check collisions in the geometric space $\mathcal{P}^{free} \subset \mathbb{R}^3$ separately from enforcing the dynamic constraints $\mathcal{D}^{free} \subset \mathbb{R}^{3(n-1)}$. The edge $e(t)$ is valid only if its geometric shape $p(t) \subset \mathcal{P}^{free}$ and derivatives $(v(t), a(t), ...) \subset \mathcal{D}^{free}$, *i.e.*, ,

$$(v, a, ...) \subset \mathcal{D}^{free} \Leftrightarrow \begin{array}{ll} \|v\|_\infty \leq v_{max}, & \forall t \in [0, \tau] \\ \|a\|_\infty \leq a_{max}, & \forall t \in [0, \tau] \\ \vdots & \end{array} \tag{4.31}$$

Since the derivatives $v, a, ...$ are polynomials, we calculate their extrema within the time period $[0, \tau]$ to compare with maximum bounds on velocity, acceleration, etc. For $n \leq 3$, the order of these polynomials is less than 5, which means we can easily solve for the extrema in closed-form.

The more challenging part is checking collisions in $\mathcal{P}^{free}$. In this work, we model $\mathcal{P}$ as an *Occupancy Grid Map*. Other representations such as a *Polyhedral Map* are also possible but these are usually hard to obtain from real-world sensor data [14, 54] and out of the scope of the discussion in this paper. Let $P := \{p(t_i) \mid t_i \in [0, \tau], i = 0, \ldots, I\}$ be a set of positions that the system traverses along the trajectory $p(t)$. To ensure a collision-free trajectory, we just need to show that $p(t_i) \in \mathcal{P}^{free}$ for all $i \in \{0, \dots, I\}$. Given a polynomial $p(t),\ t \in [0, \tau]$, the positions $p(t_i)$ are sampled by defining:

$$t_i := \frac{i}{I}\tau \quad \text{such that} \quad \frac{\tau}{I} v_{max} \geq R. \tag{4.32}$$

Here R is the occupancy grid resolution and is needed to guarantee safety. The condition ensures that the maximum distance between two consecutive samples will not exceed the map resolution. It is an approximation, since it can miss cells that are traversed by $p(t)$ with a portion of the curve within the cell shorter than R, but it prevents the trajectory from penetrating or hitting obstacles.

4.4 Trajectory Smoothing

In the proposed planning approach, the dimension of the state space increases with increasing requirements on the continuity of the final trajectory. More precisely, if C^2 continuity is required for the final trajectory, jerk should be used as a control input and the state space of the associated second order system would be $\mathbb{R}^9$ (position, velocity acceleration). Generally, planning in higher dimensional spaces (*e.g.,* , snap input) requires more time and memory to explore and store lattices/states. In this section, we discuss two approaches to reduce the computational cost to find a C^n continuous trajectory from a C^m continuous trajectory where $m < n$. The first approach is called "trajectory refinement" which is more or less the same as the traditional method in [61, 82] and Section 3.2 that generates the minimum snap trajectory from a geometric path. The second is a hierarchical approach to planning a feasible trajectory in high dimensional space by utilizing guidance from a trajectory planned in lower dimensional space. We show that the overall computation time of this hierarchical planning is shorter than the total time it takes to plan a optimal trajectory directly. Due to the fact that the final trajectory is calculated from a trajectory in lower dimensional space, we call this process as *trajectory smoothing*.

4.4.1 Trajectories Planned in Different Control Spaces

Before introducing the refinement, we look into the property of trajectories for different systems. Denote the trajectories planned using velocity, acceleration or jerk inputs as $\Phi^q, q = 1, 2, 3$ respectively. Given the same start and goal, dynamics constraints and discretization, examples of the optimal trajectories in each case are plotted in Figure 4.6, where the control effort $J^q, q = 1, 2, 3$ of the whole trajectory is measured as

$$J^q = \int_0^T \|\mathbf{x}^{(q)}\|^2 dt. \tag{4.33}$$

Denote the execution and computation time of the trajectory as T^q and t^q, $q = 1, 2, 3$ accordingly. From the planning results in 4.6, two conclusions can be drawn with increasing q:

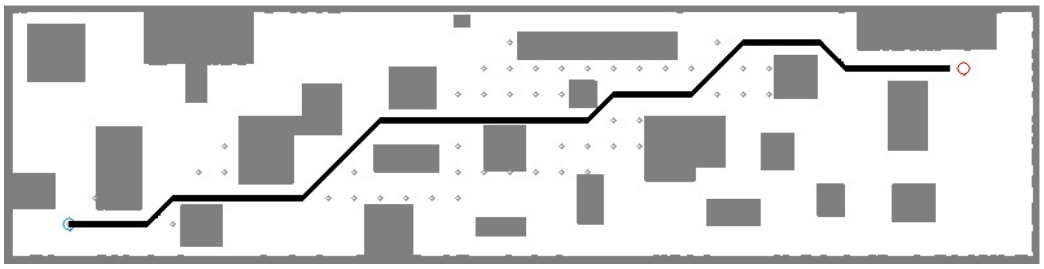
1. The execution time increases, *i.e.*, $T^1 < T^2 < T^3$;
2. The computation time increases, *i.e.*, $t^1 < t^2 < t^3$.

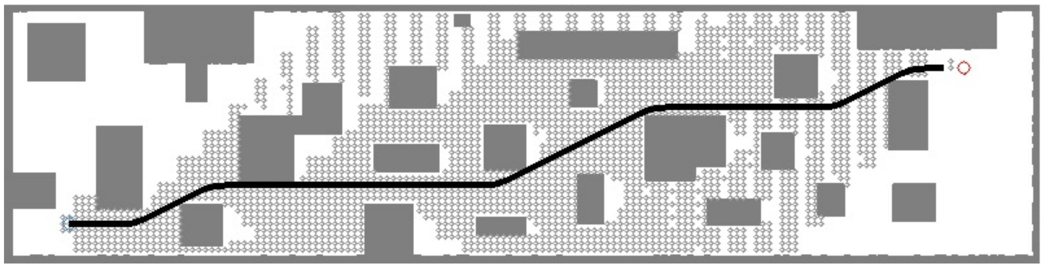
Note that the computation time increases dramatically as q increases.

4.4.2 Trajectory Refinement

As described in Section 3.2.4, a trapezoid velocity profile is commonly used to describe the robot following a path, in which the robot is assumed to move as a particle that exactly tracks the path with defined velocity function. The velocity is treated as a linear piecewise continuous function while the acceleration is a step function. This model gives the so-called *time allocation* for a large group of trajectory optimization approaches described in [54, 61, 82] and [9]. However, this approximation is naive and the resulting trajectory significantly deforms from the given path that a robot aims to follow since the modeled particle is not obeying the expected dynamics.

In this chapter, we proposed the complete solution for planning a trajectory that is valid in control space. The resulting trajectory gives not only the collision-free path, but also the


(a) $\Phi^1 : T^1 = 32s, J^1 = 42, t^1 = 2ms$


(b) $\Phi^2 : T^2 = 33s, J^2 = 2.25, t^2 = 60ms$

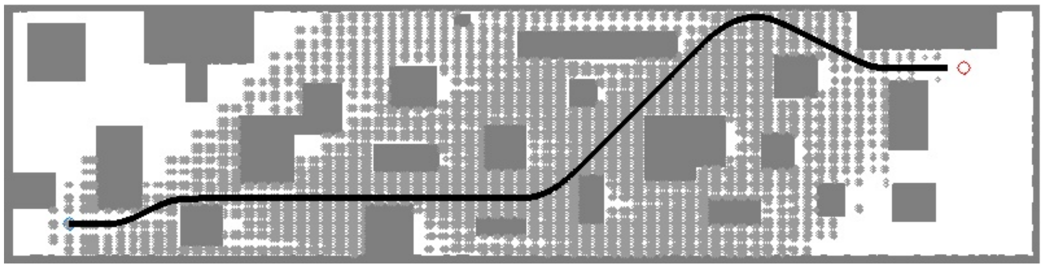

(c) $\Phi^3 : T^3 = 34s, J^3 = 3.75, t^3 = 1646ms$

Figure 4.6: Optimal trajectories planned using piecewise constant (a) velocity, (b) acceleration, (c) jerk from a start (blue dot) to a goal (red dot) state. Grey dots indicates explored states.

time for reaching individual waypoints. Generally, planning in higher dimension is slower than in lower dimension as the number of states grows. We propose to plan a trajectory in lower dimension and use the result as the *time allocation* for a refined trajectory in higher dimension to reduce the time for calculating. The refined trajectory $x^*(t)$ is derived from solving following unconstrained QP:

$$\begin{aligned} \min_{D} \sum_{k=0}^{N-1} \int_0^{\tau_k} \left\| p_{Dk}^{(n)}(t) \right\|^2 dt \\ \text{s.t. } x_0(0) = s_0, \quad x_{N-1}(\tau_{N-1}) = s_g \\ x_{k+1}(0) = x_k(\tau_k),\ k \in \{0, \dots, N-2\} \\ p_{Dk}(\tau_k) = p_k, \qquad k \in \{0, \dots, N-1\} \end{aligned} \tag{4.34}$$

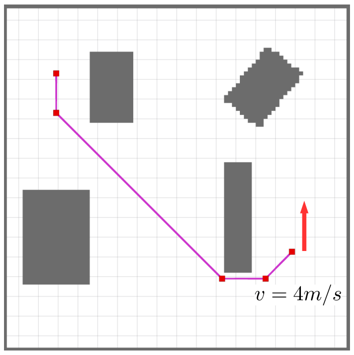
where the initial, end state s_0, s_g and the intermediate waypoints $p_k, k \in \{0, \dots, N-1\}$ are given. The time for each trajectory segment τ_k is also given from the prior trajectory. The solution for (4.34) is proposed in [61]. We ignore the mathematical details in this section and only show the trajectory refinement results in Figure 4.7.

Even though the deformation of the refined trajectory from the prior trajectory is small as shown in above figure, to guarantee the safety, we still need to check if the new trajectory hits any obstacle. One solution is to follow the planning in adaptive dimensionality as proposed in [24]. The other simpler algorithm is to rectify the final trajectory by adding intermediate waypoints on the original collision-free trajectory for the time when the trajectory contacts obstacles.

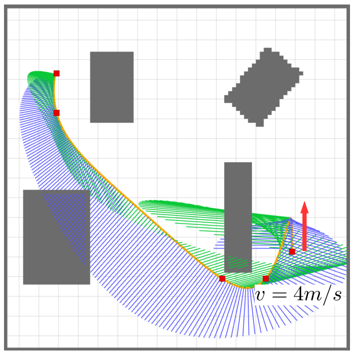
4.4.3 Using Trajectories as Heuristics

The fact that searching a optimal trajectory in the lower dimensional space is much faster than in a higher dimensional space leads to the approach described in this subsection to speed up the planning speed for the actual MAV system.

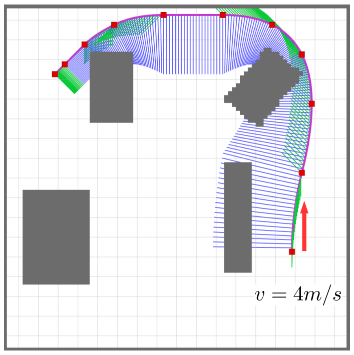
Denote the prior trajectory in lower dimensional space as Φ^p, we are searching for a trajectory in higher dimensional space Φ^q $(q > p)$. Assume the duration of each primitive



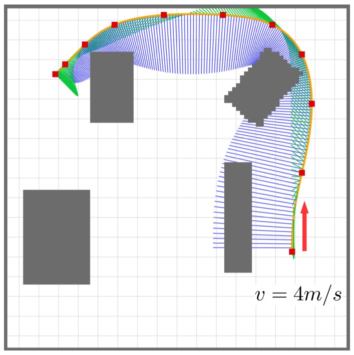
(a) $T = 8.5$.



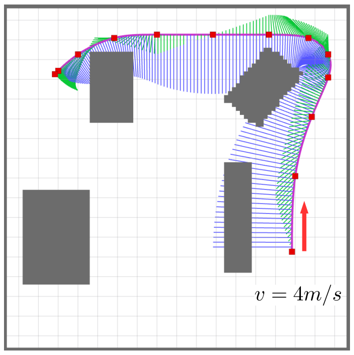
(b) $T = 8.5, J = 296.6$.



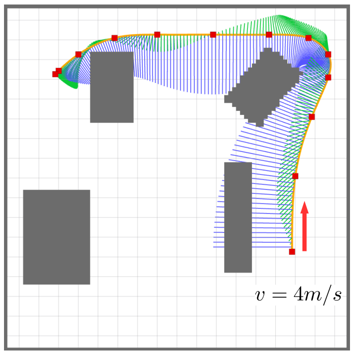
(c) $T = 10, J = 14.0$.



(d) $T = 10, J = 21.3$.



(e) $T = 12, J = 11.3$.



(f) $T = 12, J = 13.6$.

Figure 4.7: Trajectories planned from start s to goal g with initial velocity ($4m/s$). The blue/green lines show the speed/acceleration along trajectories respectively and the red points are the intermediate waypoints. (a) shows the shortest path. The time is allocated using the trapezoid velocity profile for generating min-jerk trajectory in (b). The resulting trajectory has a large cost for efforts J. (c), (e) show the trajectory planned using acceleration-controlled and jerk-controlled system. In (d) and (f), we show the corresponding refined trajectories.

in Φ^q is τ, each lattice $\mathbf{s}_n^q$ in the graph is associated with a time T_n which is the minimum time it takes from the start to the current lattice. T_n is an integer multiplication of τ. Instead of calculating the heuristic $H(\mathbf{s}_n^q)$ from current state $\mathbf{s}_n^q$ to the goal $\mathbf{s}_g$ directly as described in Section 4.3.1, we propose to use the intermediate goal $\mathbf{s}_n^p = \Phi^p(T_n)$ evaluated from trajectory Φ^p at T_n such that the heuristic value is calculated as below:

$$H(\mathbf{s}_n^q, \Phi^p) = H_1(\mathbf{s}_n^q, \mathbf{s}_n^p) + H_2(\mathbf{s}_n^p, \mathbf{s}_g). \tag{4.35}$$

The first term $H_1(\cdot)$ on the RHS of (4.35) is proposed in the following subsection where $\mathbf{s}_n^q$ is fully defined but $\mathbf{s}_n^p$ has undefined states.

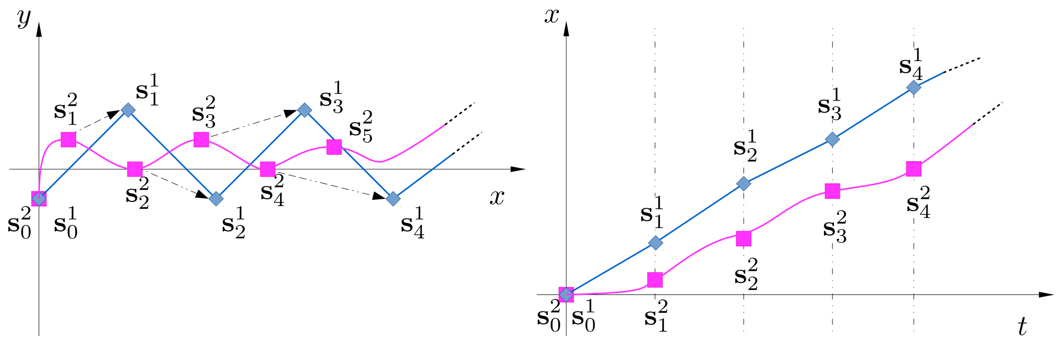
The second term $H_2(\cdot)$ is given directly as the cost from $\mathbf{s}_n^p$ to the goal by following Φ^p, to be more specific:

$$H_2(\mathbf{s}_n^p, \mathbf{s}_g) = J^q(\mathbf{s}_n^p, \mathbf{s}_g) + \rho(T^p - T_n) \tag{4.36}$$

where T^p is execution time of Φ^p and $J^q(\mathbf{s}_n^p, \mathbf{s}_g)$ is the control effort from $\mathbf{s}_n^p$ to $\mathbf{s}_g$ along Φ^p as expressed in (4.33). This formulation is consistent with the cost function defined before in (4.30). As the prior trajectory is in the lower dimensional space, J^q for Φ^p is always zero (e.g. for planning a optimal trajectory Φ^2 that uses acceleration input, the corresponding control efforts of Φ^1 is zero as there is no acceleration along Φ^1). Thus $H_2(\cdot)$ turns out to be only the execution time between $\mathbf{s}_n^p$ and goal:

$$H_2(\mathbf{s}_n^p, \mathbf{s}_g) = \rho(T^p - T_n). \tag{4.37}$$

4.8 shows an example of applying (4.35) to search a trajectory Φ^2 using acceleration with a prior trajectory Φ^1 planned using velocity. The new trajectory Φ^2 tends to stick with the prior trajectory Φ^1 due to the effect of $H_1(\cdot)$. $H_2(\cdot)$ will push the searching moving forward towards the goal. In fact, the heuristic function defined in (4.35) is not admissible since it may not necessarily be the under-estimation of the actual cost-to-goal. However, we are able to search for trajectories in higher dimensional space in a much faster speed by searching the neighboring regions of the given trajectory instead of exploring the whole state space with the same priority.



y
x
$\mathbf{s}_0^2$ $\mathbf{s}_0^1$
$\mathbf{s}_1^2$ $\mathbf{s}_1^1$
$\mathbf{s}_2^2$ $\mathbf{s}_2^1$
$\mathbf{s}_3^2$ $\mathbf{s}_3^1$
$\mathbf{s}_4^2$ $\mathbf{s}_4^1$
$\mathbf{s}_5^2$
x
t

(a) $x - y$ plot (b) $t - x$ plot

Figure 4.8: Search Φ^2 (magenta) using Φ^1 (blue) as the heurisric. Left figure plots the trajectories in $x - y$ plane, the black arrows indicate the H_1. Right figure shows the corresponding x position with respect to time t along each trajectory, for states with the same subscript, they are at the same time T_n.

The results of applying (4.35) for the same planning tasks in Figure 4.6 are shown in Figure 4.9, in which Φ^1 is used to plan for both trajectory Φ^2 and Φ^3. Comparing Figure 4.9 to 4.6, the total cost of control effort and execution time, namely $J^q + \rho T^q$, of the new trajectories Φ^q in Figure 4.9 are greater than the optimal trajectories in 4.6, but the computation time t^q and the number of expanded nodes are much less.

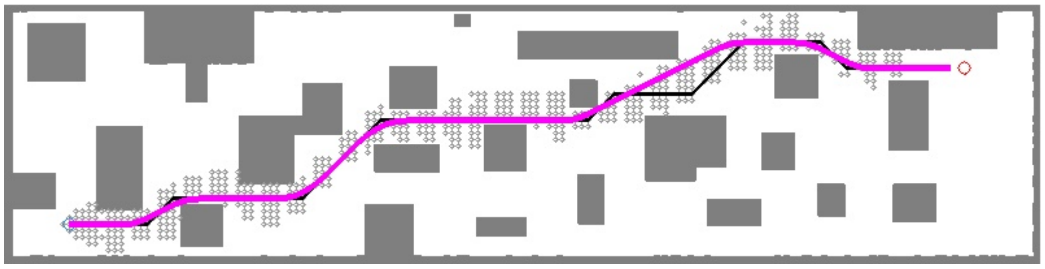
Linear Quadratic Minimum Time for Jerk Control

The heuristic function $H(\mathbf{s}, \mathbf{s}_g)$ for graph search is an under-estimation of actual cost from the state $\mathbf{s}$ to the goal $\mathbf{s}_g$ by relaxing the dynamics and obstacles constraints. We try to find a state-to-state optimal trajectory of Problem 3, whose cost serves as the heuristic H. The explicit solution for the optimal cost for velocity, acceleration control has been shown in previous section, here we show the explicit solution for jerk control.

Problem 3. *Given a current state* $\mathbf{s}$, *the goal state* $\mathbf{s}_g$, *find the optimal trajectory according to the cost function*

$$\min_{\mathbf{j},T} \int_0^T \|\mathbf{j}\|^2 dt + \rho T \tag{4.38}$$

Assume the initial state is given as $\mathbf{s} = [\mathbf{p}_0^\mathsf{T}, \mathbf{v}_0^\mathsf{T}, \mathbf{a}_0^\mathsf{T}]^\mathsf{T}$, the formulation of position of the


(a) $\Phi^2 : T^2 = 35s, J^2 = 3.0, t^2 = 11ms$

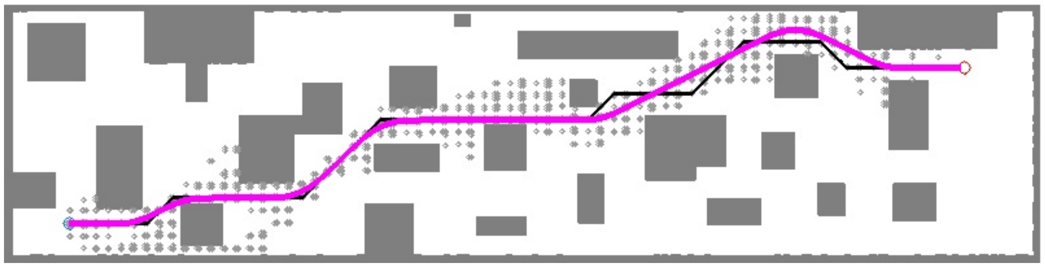

(b) $\Phi^3 : T^3 = 36s, J^3 = 4.25, t^3 = 98ms$

Figure 4.9: Trajectories (magenta) planned using Φ^1 (black) as the heuristic. The computation time t^q and the number of expanded nodes are much less than the searching results in 4.6.

optimal trajectory for (4.38) is given from the Pontryagin's minimum principle [68] as

$$\mathbf{p} = \frac{\mathbf{d}_5}{120}t^5 + \frac{\mathbf{d}_4}{24}t^4 + \frac{\mathbf{d}_3}{6}t^3 + \frac{\mathbf{a}_0}{2}t^2 + \mathbf{v}_0 t + \mathbf{p}_0 \tag{4.39}$$

The coefficients $\{\mathbf{d}_5, \mathbf{d}_4, \mathbf{d}_3\}$ are defined in [68] by $\mathbf{s}, \mathbf{s}_g$ and T. As a result, the total cost of (4.38) can be written as a function of time T as

$$\begin{aligned} \mathcal{C}(T) &= \int_0^T (\frac{\mathbf{d}_5}{2}t^2 + \mathbf{d}_4 t + \mathbf{d}_3)^2 dt + \rho T \\ &= \frac{\mathbf{d}_5^2}{20}T^5 + \frac{\mathbf{d}_4^\intercal \mathbf{d}_5}{4}T^4 + (\frac{\mathbf{d}_4^\intercal \mathbf{d}_4}{3} + \frac{\mathbf{d}_3^\intercal \mathbf{d}_5}{3})T^3 \\ &\quad + \mathbf{d}_3^\intercal \mathbf{d}_4 T^2 + \mathbf{d}_3^2 T + \rho T \end{aligned} \tag{4.40}$$

The minimum of $\mathcal{C}(T)$ can be derived by taking the derivative with respect to T and finding the root T^* of

$$\frac{d\mathcal{C}}{dT} = c_0 + \ldots + c_6 T^{-6} = 0, \ T \in [0, \infty) \tag{4.41}$$

Therefore, $H(\mathbf{s}, \mathbf{s}_g) = \mathcal{C}(T^*)$. The coefficients in (4.41) are derived as follows:

(1) Fully Defined $\mathbf{s}_g = [\mathbf{p}_1^\mathsf{T}, \mathbf{v}_1^\mathsf{T}, \mathbf{a}_1^\mathsf{T}]^\mathsf{T}$

$$\begin{aligned}
c_0 &= \rho,\ c_1 = 0,\ c_2 = -9\mathbf{a}_0^2 + 6\mathbf{a}_0^\mathsf{T}\mathbf{a}_1 - 9\mathbf{a}_1^2, \\
c_3 &= -144\mathbf{a}_0^\mathsf{T}\mathbf{v}_0 - 96\mathbf{a}_0^\mathsf{T}\mathbf{v}_1 + 96\mathbf{a}_1^\mathsf{T}\mathbf{v}_0 + 144\mathbf{a}_1^\mathsf{T}\mathbf{v}_1, \\
c_4 &= 360(\mathbf{a}_0 - \mathbf{a}_1)^\mathsf{T}(\mathbf{p}_0 - \mathbf{p}_1) - 576\mathbf{v}_0^2 - 1008\mathbf{v}_0^\mathsf{T}\mathbf{v}_1 - 576\mathbf{v}_1^2, \\
c_5 &= 2880(\mathbf{v}_0 + \mathbf{v}_1)^\mathsf{T}(\mathbf{p}_0 - \mathbf{p}_1), \\
c_6 &= -3600(\mathbf{p}_0 - \mathbf{p}_1)^2.
\end{aligned} \tag{4.42}$$

(2) Partially Defined $\mathbf{s}_g = [\mathbf{p}_1^\mathsf{T}, \mathbf{v}_1^\mathsf{T}]^\mathsf{T}$

$$\begin{aligned}
c_0 &= \rho,\ c_1 = 0,\ c_2 = -8\mathbf{a}_0^2, \\
c_3 &= -112\mathbf{a}_0^\mathsf{T}\mathbf{v}_0 - 48\mathbf{a}_0^\mathsf{T}\mathbf{v}_1, \\
c_4 &= 240\mathbf{a}_0^\mathsf{T}(\mathbf{p}_0 - \mathbf{p}_1) - 384\mathbf{v}_0^2 - 432\mathbf{v}_0^\mathsf{T}\mathbf{v}_1 - 144\mathbf{v}_1^2, \\
c_5 &= (1600\mathbf{v}_0 + 960\mathbf{v}_1)^\mathsf{T}(\mathbf{p}_0 - \mathbf{p}_1), \\
c_6 &= -1600(\mathbf{p}_0 - \mathbf{p}_1)^2.
\end{aligned} \tag{4.43}$$

(3) Partially Defined $\mathbf{s}_g = \mathbf{p}_1$

$$\begin{aligned}
c_0 &= \rho,\ c_1 = 0,\ c_2 = -5\mathbf{a}_0^2, \\
c_3 &= -40\mathbf{a}_0^\mathsf{T}\mathbf{v}_0, \\
c_4 &= 60\mathbf{a}_0^\mathsf{T}(\mathbf{p}_0 - \mathbf{p}_1) - 60\mathbf{v}_0^2, \\
c_5 &= 160\mathbf{v}_0^\mathsf{T}(\mathbf{p}_0 - \mathbf{p}_1), \\
c_6 &= -100(\mathbf{p}_0 - \mathbf{p}_1)^2.
\end{aligned} \tag{4.44}$$

4.5 Analysis and Experimental Results

4.5.1 Heuristic Function

We proposed two different heuristic functions for planning the trajectory with graph search algorithms in Section 4.3.1: the first one estimates the minimum time using the max speed constraint as shown in (4.27), denoted as h_1; the other one estimates the minimum cost function using the dynamic constraints as shown in (4.30), denoted as h_2. The former heuristic h_1 is fast to compute, but it fails to take in to account of the system's dynamics; the latter heuristic h_2 requires to solve for the real roots of a polynomial, but it reveals the lower bound of the cost regarding system's dynamics and thus it is a tighter underestimation of the actual cost. Here we compare the performance of the algorithm with respect to the two heuristics h_1, h_2. As a reference, by setting the heuristic function to zero changes the algorithm into Dijkstra search. Figure 4.10 visualizes the expanded nodes while searching towards the goal from a state with initial velocity $3m/s$ in positive vertical direction.

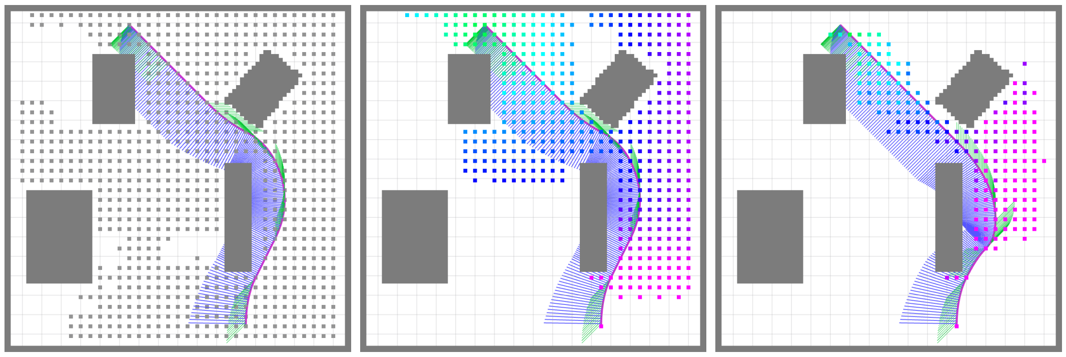

(a) Dijkstra. (b) A^* with h_1. (c) A^* with h_2.

Figure 4.10: Generated trajectories using different heuristic functions. The expanded nodes (small dots) are colored by the corresponding cost value of the heuristic function. Grey nodes have zero heuristic cost, high cost nodes are colored red while low cost nodes are colored green.

Table 4.1: Comparison of Heuristic Functions

	Time(s)	# of Expanded Nodes
Dijkstra	0.16	2707
A^* with h_1	0.064	1282
A^* with h_2	0.016	376

The run time analysis for the two heuristic functions is shown in Table 4.1. We can see that the *Minimum Cost Heuristic* h_2 makes the searching faster as it expands less nodes without loss of optimality. However, when it comes to the system with higher dimension, calculating h_2 becomes harder as one can not analytically find the roots of a polynomial with order greater than 4. As claimed in Section 4.3.1, when the maximum velocity is low, h_1 is efficient enough for any dynamic system.

4.5.2 Run Time Analysis

To evaluate the computational efficiency of the algorithm, we record the run time of generating hundreds of trajectories (Figure 4.11) using either acceleration-controlled or jerk-controlled system in both 2D and 3D environments. Table 4.2 shows the time it takes for each system. We can see that planning in 3D takes more time than in 2D; also, planning in jerk space is much slower (10 times) than in acceleration space.

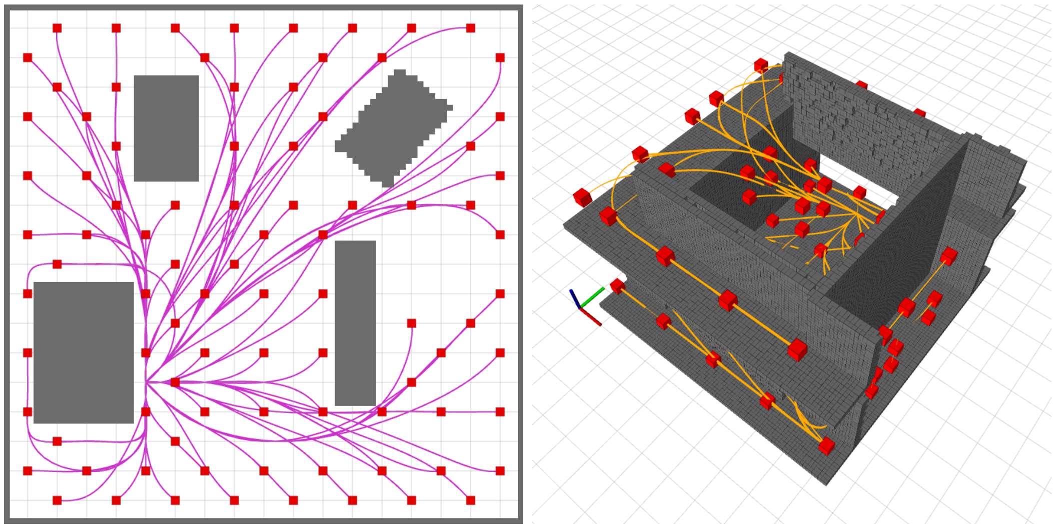

(a) 2D Planning. (b) 3D Planning.

Figure 4.11: Trajectories generated to sampled goals (small red balls). For the 2D case, we use 9 primitives while for the 3D case, the number is 27.

4.5.3 Re-planning and Comparisons

In this section, we show results of our navigation system that builds on the Receding Horizon Control (RHC) framework as described in Section 2.3 with the proposed trajectory gener-

Table 4.2: Trajectory Generation Run Time

Map	Time(s)	Accel-controlled	Jerk-controlled
2D	Avg	0.016	0.147
	Std	0.015	0.282
	Max	0.086	2.13
3D	Avg	0.094	2.98
	Std	0.155	3.78
	Max	0.515	9.50

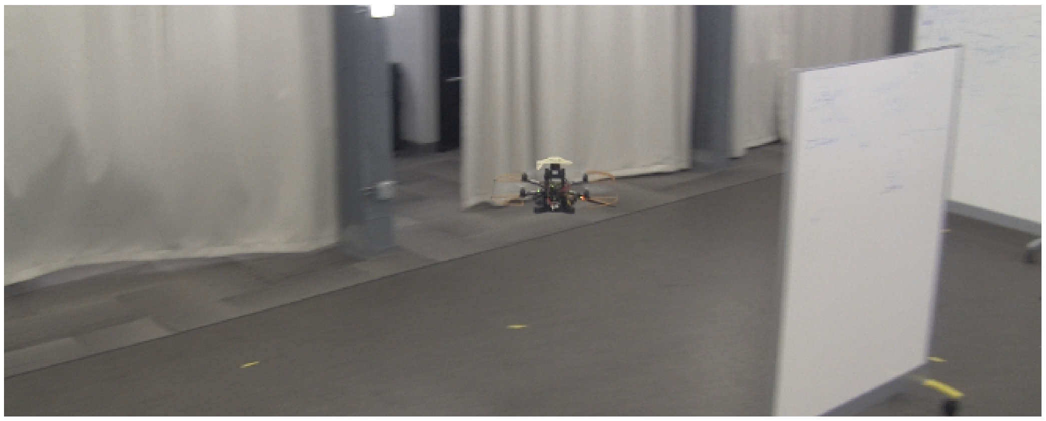
ation method. As a comparison, we also set up the system that utilizes the prior planned path as the guide for trajectory generation. To demonstrate the fully autonomous collision avoidance on a quadrotor, we use the AscTec Pelican platform with a Hokuyo laser range-finder. We run state estimation and obstacle detection (mapping) based on Hokuyo lidar on an onboard Intel NUC-i7 computer. Figure 4.12 shows the performance of using these two approaches to avoid an obstacle by re-planning at the circle position where the desired speed is non-zero. The traditional path-based approach in Figure 4.12b leads to a sharp turn while our approach generates a smoother trajectory shown in Figure 4.12c.

Figure 4.13 shows the results in simulation where we set up a longer obstacle-cluttered corridor for testing. The re-planning is triggered constantly at 3 Hz and the maximum speed is set to be 3 m/s. Our method generates a better overall trajectory compared to the traditional method as it avoids sharp turns when avoiding obstacles.

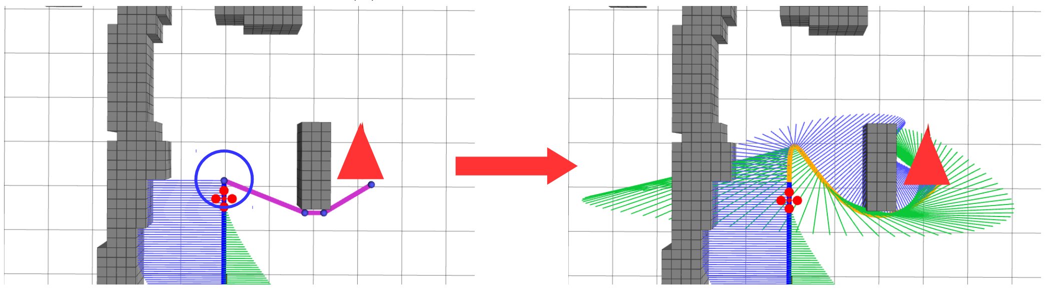
As a conclusion, the proposed search-based motion planning method is able to generate trajectories that are *dynamically feasible, collision-free, resolution optimal* and *complete* in real time. As shown in Table 4.3, we will further use this near-ultimate method to solve various practical planning problems in the latter chapters.

Method	***Feasibility***	***Safety***	***Optimality***	***Completeness***	***Run time***
SMP	Dynamically feasible	Collision free	Resolution optimal	Globally complete	Fast

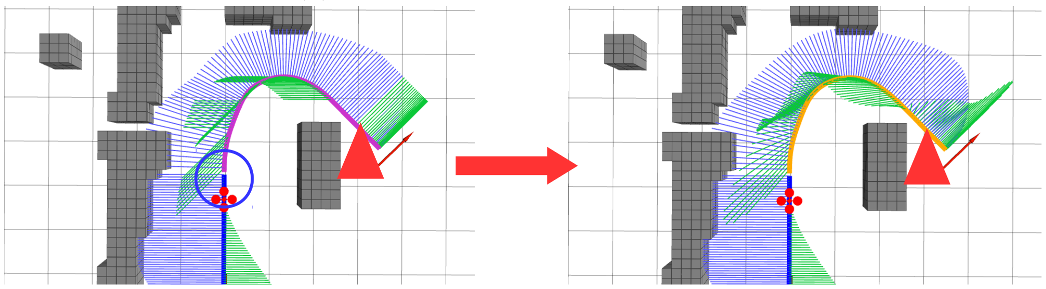
Table 4.3: Evaluation of search-based motion planning (SMP) method. Red blocks indicate the drawbacks of the corresponding algorithm.



(a) Experimental environment

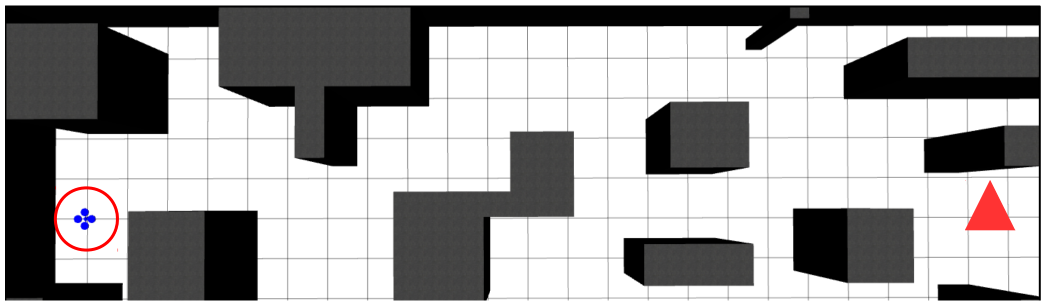


(b) Re-plan with path-based approach.



(c) Re-plan with our method.

Figure 4.12: Pelican experiments using different trajectory generation pipelines. The robot is initially following a trajectory (blue curve) and needs to re-plan at the end of this prior trajectory (circled) to go to the goal (red triangle). The state from which the robot re-plans is non-static and the speed is $2m/s$ in positive vertical direction. (b) shows the result of using traditional path-based trajectory generation method, the shortest path (purple line segments in the left figure) leads to the final trajectory (yellow curve in the right figure); (c) shows the result of using our trajectory generation method, the shortest trajectory (purple curve in the left figure) leads to the smoother final trajectory (yellow curve in the right figure).


(a) Simulation Environment.

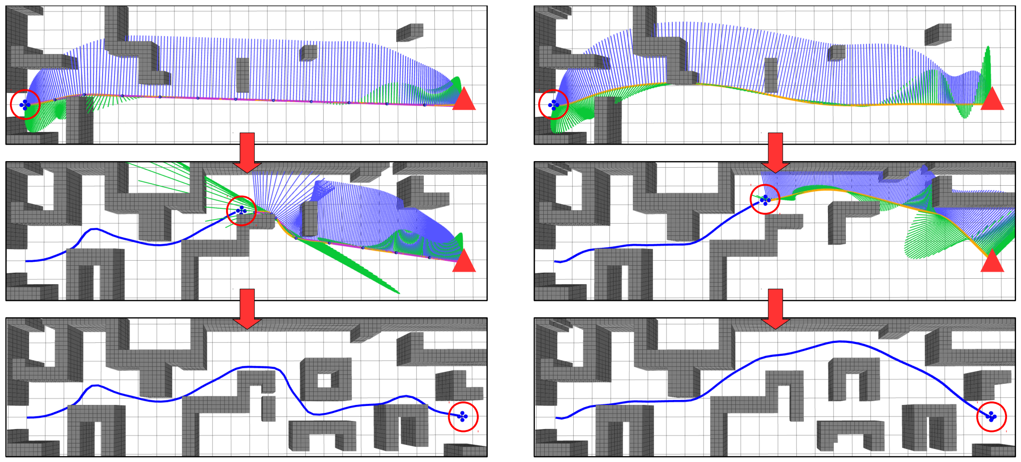

(b) Path-based approach. (c) Our method.

Figure 4.13: Re-planning with RHC in simulation using different trajectory generation pipelines. The robot starts from the left (circled) and the goal is at the right side of the map (red triangle). Blue curves show the traversed trajectory. (b) shows the re-planning processes using traditional path-based trajectory generation method. (c) shows the re-planning processes using proposed method in this paper. We can see that the overall trajectories in (c) is smoother than in (b).

Chapter 5

Extensions of Search-based Trajectory Planning

The search-based motion planning framework based on Chapter 4 can be extended to many applications. In this chapter, we explore its potential to solve three practical planning problems in real-world navigation tasks:

1. *Planning with Motion Uncertainty*: The robot is not able to perfectly track the nominal trajectory in the presence of disturbances. We consider how to plan a safer trajectory that is less likely to crash the robot in an obstacle-cluttered environment.

2. *Planning with Limited FOV*: Vision-based state estimation and the limited FOV of sensors to detect obstacles require the robot to travel with constraints on the yaw angle. We propose a way to find the desired yaw profile along the trajectory that obeys this constraint.

3. *Planning in SE(3)*: An MAV can be treated as ellipsoid instead of sphere such that we are able to fully exploit its agility to plan trajectories in narrow environments.

We show that these problems are variations of Problem 1 in Chapter 4, in which we solve an optimal trajectory that is defined as (3.1). As a quick reminder, an optimal trajectory (not guaranteed to be unique) that respects the dynamical and collision constraints, and is *minimum-time* and *smooth* can be obtained from:

Problem 4. *Given an initial state $x_0 \in \mathcal{X}^{free}$ and a goal region $\mathcal{X}^{goal} \subset \mathcal{X}^{free}$, find a polynomial trajectory $\Phi(t)$ such that:*

$$\begin{aligned} &\arg\min_{\Phi} \; J_q(\Phi) + \rho_T T \\ s.t. \;\; &\dot{x}(t) = Ax(t) + Bu(t), \\ &x(0) = x_0, \quad x(T) \in \mathcal{X}^{goal}, \\ &x(t) \in \mathcal{X}^{free}, \quad u(t) \in \mathcal{U}. \end{aligned} \tag{5.1}$$

where the parameter $\rho_T \geq 0$ determines the relative importance of the trajectory duration T versus its smoothness J_q.

In the latter sections, we modify the above problem for individual case, all of which is able to be solved using the graph search technique with motion primitives as described in Chapter 4.

5.1 Planning with Motion Uncertainty

Existing work in trajectory planning assumes the availability of high control authority allowing robots to perfectly track the generated trajectories. However, this assumption is impractical in the real world since unpredictable environmental factors such as wind, air drag, wall effects can easily disturb the robot from the nominal trajectory. Thus, even though a nominal trajectory is in free space, it can easily lead to a collision when the robot gets close to obstacles. To reduce this risk, a trajectory that stays away from obstacles is desired. Traditionally, this is worked around by inflating the obstacle by a radius that is much larger than the actual robot size. However, this over-inflation strategy is not a complete solution for motion planning in obstacle-cluttered environments since it is prone to block small gaps such as doors, windows, and narrow corridors.

The reachable set (funnel) [94] is used to model motion uncertainty for a robot following a time-varying trajectory. Assuming bounded and time-invariant disturbances leads to bounded funnels. It is straightforward to show that the funnel of a linear system, as in (4.4), controlled by a PD-controller [44] is bounded by a certain radius with respect to the control gains. However, the planning strategy in [94] treats the motion uncertainty as a hard constraint for collision checking which is an over conservative strategy that discards all the trajectories close to obstacles. Besides, it is computationally expensive to search using funnels.

Alternatively, Artificial Potential Fields (APFs) are used to plan paths that are away from obstacles efficiently [2, 46, 104]. APFs have been used to model collision costs in trajectory generation through line integrals [22, 71, 77, 96] in which the safe trajectory is refined from an initial nominal trajectory through gradient descent. However, this gradient-based approach strongly relies on the initial guess of time allocation and the sampling of end derivatives for fast convergence and it ignores the dynamical constraints during the re-optimization. Moreover, the result is easily trapped in undesired local minima. Thus, it is not an appropriate method to solve the safe planning problem in complex environments.

In this section, we propose a novel approach that models the motion uncertainty as a soft constraint and plans for trajectories that are as safe as possible with respect to the collision cost through the line integral of the APF. The resulting trajectory is constrained to be within a tunnel from the initial trajectory, such that it is suitable for planning in unknown environments. The proposed approach does not require the Jacobian and Hessian of the cost functions and hence is computationally efficient.

5.1.1 Problem Formulation

We call the trajectory derived from solving Problem 4 that ignores the collision cost as the nominal trajectory Φ_0. We treat trajectory planning with motion uncertainty as a problem of finding a locally optimal trajectory around the nominal Φ_0 that takes into account the collision cost. It can be formulated as a variation of Problem 4 where we add a collision cost J_c in the objective function and a search region (tunnel) $\mathcal{T}(\Phi_0)$ around Φ_0 in the constraints:

Problem 5. *Given an initial state $x_0 \in \mathcal{X}^{free}$, a goal region $\mathcal{X}^{goal} \subset \mathcal{X}^{free}$ and a search* *region $\mathcal{T}(\Phi_0)$ around the nominal trajectory Φ_0, find a polynomial trajectory Φ such that:*

$$\begin{aligned} &\arg\min_{\Phi}\ J_q + \rho_T T + \rho_c J_c \\ s.t.\ \ &\dot{x}(t) = Ax(t) + Bu(t) \\ &x(0) = x_0,\ x(T) \in \mathcal{X}^{goal} \\ &x(t) \in \mathcal{T}(\Phi_0) \cap \mathcal{X}^{free}, \quad u(t) \in \mathcal{U} \end{aligned} \tag{5.2}$$

where the weights $\rho_T,\ \rho_c \geq 0$ determines the relative importance of the trajectory duration T and collision cost J_c versus its smoothness J_q.

In this section, we show that Problem 5 can be converted into a search problem and solved using motion primitives.

Collision Cost J_c

We define the collision cost in Problem 5 as the line integral:

$$J_c(\Phi) = \int_{\Phi} U(s)\, ds. \tag{5.3}$$

where $U(s)$ is the potential value of position $s \in \mathbb{R}^m$ that is defined as:

$$U(s) = \begin{cases} 0, & d(s) \geq d_{\text{thr}} \\ F(d(s)), & d_{\text{thr}} > d(s) \geq 0 \end{cases} \tag{5.4}$$

where $d(s)$ is the distance of position s from the the closest obstacle. In addition, for positions that are away from obstacles more than a distance d_{thr}, we consider their collision cost to be negligible. Thus, the potential function $U(s)$ should be a non-negative and monotonically decreasing function in domain $[0, d_{\text{thr}})$ and equal to zero when $d \geq d_{\text{thr}}$. One choice for $F(\cdot)$ is an polynomial function with order $k > 0$:

$$F(d) = F_{\max}\left(1 - \frac{d}{d_{\text{thr}}}\right)^k. \tag{5.5}$$

The analytic expression of the line integral in (5.3) is hard to compute, instead we sample the trajectory at I points with uniform time step dt for approximation:

$$\int_{\Phi} U(s)ds \approx \sum_{i=0}^{I-1} U(p_i)\|v_i\|dt \tag{5.6}$$

where $dt = \frac{T}{I-1}$ and p_i, v_i are corresponding position and velocity at time $i \cdot dt$. This approximation can be easily calculated when the obstacle and potential field are represented as a grid as shown in Figure 5.1.

The proposed collision cost evaluates the "danger level" of a trajectory with respect to its relative position to obstacles. J_c can be more general than spatial penalty. For example, we can consider the velocity penalty by using the gradient of APF to slow down the trajectory when it is moving toward obstacles:

$$J_c(\Phi) = \oint_{\Phi} U(s) + \rho_v \dot{U}(s) \cdot \dot{s}\, ds. \tag{5.7}$$

Tunnel Constraint $\mathcal{T}$

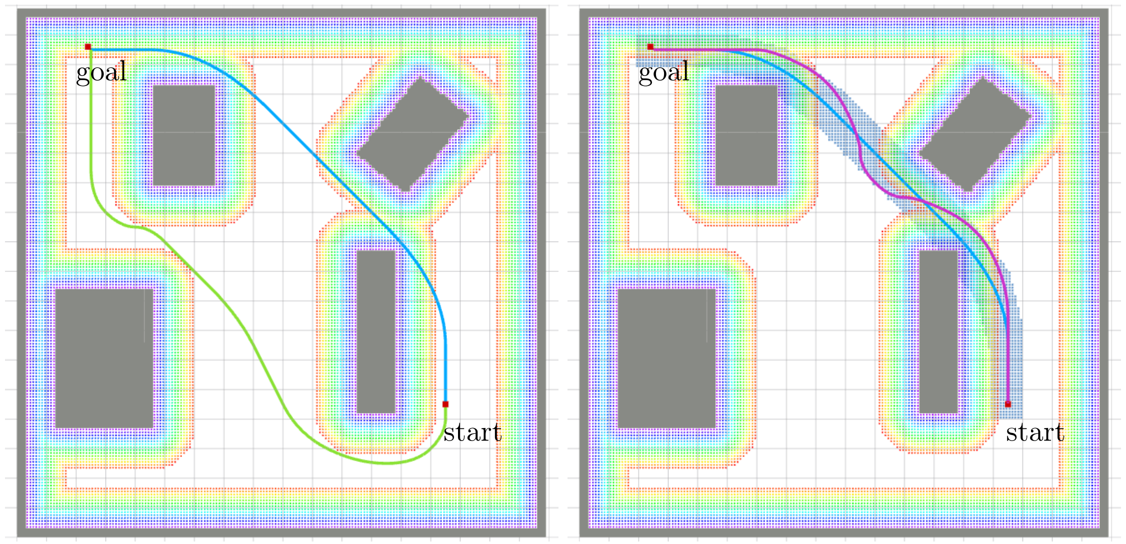
A tunnel is a configuration space around the nominal trajectory Φ_0 that is used to bound the perturbation. Let $D(r)$ be the disk with radius r, the tunnel $\mathcal{T}(\Phi_0, r)$ is the Minkowski sum of D and Φ as:

$$\mathcal{T}(\Phi_0, r) = \Phi_0 \oplus D(r). \tag{5.8}$$

Note that $\mathcal{T}$ could overlap with obstacles. Thus, we enforce the valid state to be inside the intersection of $\mathcal{T}$ and free space $\mathcal{X}^{free}$ to guarantee safety. When $r \to \infty$, Problem 5 is equivalent to computing a globally optimal trajectory.

5.1.2 Solution

Given the set of motion primitives $\mathcal{U}_M$ and the induced space discretization, we can reformulate Problem 5 as a graph-search problem similar to Chapter 4 which is solvable through dynamic programming algorithms such as Dijkstra and A*. For each primitive Φ_n, we



(a) Global plans. (b) Local plans.

Figure 5.1: Planning in an occupancy grid map. Rainbow dots indicate the truncated *Artificial Potential Field* (APF) generated from (5.4). In the left figure, the blue trajectory is the shortest trajectory that ignores collision cost; the green trajectory is the shortest trajectory that treats the APF as obstacles. In the right figure, the magenta trajectory is the planned trajectory using the proposed method that takes into account the collision cost. It is locally optimal within the tunnel (blue region) around the nominal shortest trajectory from (a).

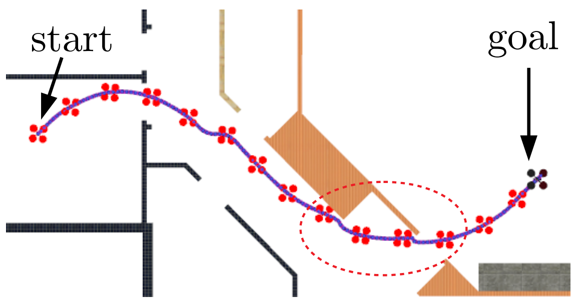
sample I_n points to calculate its collision cost according to (5.6). In the grid map, the I_n should be dense enough to cover all the cells that Φ_n traverses. One choice of automatically selecting I_n is

$$I_n = \frac{\bar{v}_{max} \cdot \Delta t_n}{r_M}, \;\; \bar{v}_{max} = \max\{v_x, v_y, v_z\}. \tag{5.9}$$

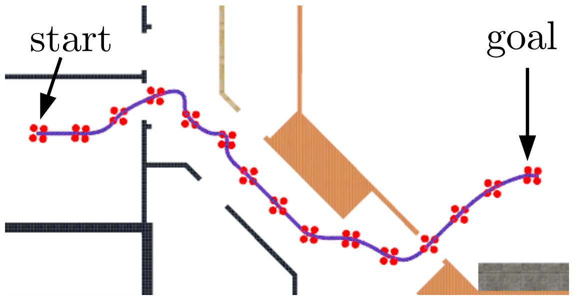
where r_M is the grid resolution.

5.1.3 Experimental Results

In Figure 5.2, a quadrotor tries to reach the goal position using the proposed planner in an office environment. The environment is shown as a 2D colored schematic, but the robot initially has no information about the environment. Therefore, it needs to constantly re-plan at certain frequency to avoid new obstacles that appear in the updated map. Figure 5.2a shows the results using traditional method in Chapter 4 that doesn't consider collision costs, in which the quadrotor occasionally touches the wall inside the circled region. Figure 5.2b



(a) Traditional method.



(b) Our method with APF.

Figure 5.2: MAV with limited sensing navigating in an office environment. The left and right figures show the results from using two different planners: (a) original search-based method that does not consider collision cost; (b) the proposed method that plans for optimal trajectories with respect to collision cost. In (a), the robot touches the wall multiple times in the circled region. The trajectory in (b) is much safer.

shows the results from using the proposed method, in which the robot stays away from walls and safely goes in and out of rooms through the middle of open doors. The re-planning time using our method is fast in this 2D scenario, the run time of which is below 10 ms for a 2^{nd} order dynamic model.

5.2 Planning with Limited FOV

Due to the fact that the yaw of an MAV system does not affect the system dynamics, this flat output is frequently ignored in existing planning works. Except when using omni-directional sensors, a fully autonomous MAV system is usually directional. In order to guarantee safety while navigating in an unknown environment, an MAV prefers to move in the direction that can be seen by a range sensor such as RGB-D or *time-of-flight* (TOF) camera which has limited FOV. Thus, its yaw $\psi(t)$ should constatnly change as the robot moves to different locations. Specifically, the desired yaw ψ is related to the velocity direction: $\xi = \arctan v_y/v_x$. This constraint is non-linear and couples the flat outputs x and y, thus it is hard to model it in the optimization framework as proposed in [14, 82]. In this section, we develop a search-based method that resolves this constraint properly by splitting it into two parts: a soft constraint that minimizes the difference between ψ and ξ and a hard constraint that enforces the moving direction ξ to be inside the FOV of the range sensor.

5.2.1 Problem Formulation

We define an additional cost term representing a soft FOV constraint as the integral of the square of angular difference between velocity direction and desired yaw:

$$J_\psi(\Phi) = \int_0^T [\psi(t) - \xi(t)]^2 dt, \tag{5.10}$$

while the hard constraint can be formulated by the absolute angular difference and the sensor's horizontal FOV θ:

$$|\psi(t) - \xi(t)| \leq \frac{\theta}{2}, \tag{5.11}$$

We modify Problem 4 to add these constraints as:

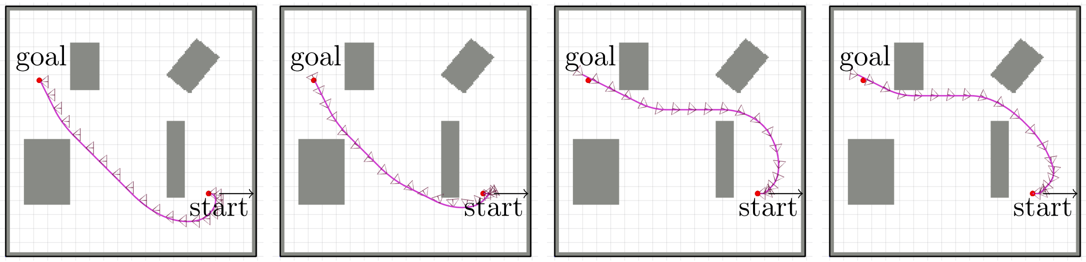
Problem 6. *Given an initial state $x_0 \in \mathcal{X}^{free}$, a goal region $\mathcal{X}^{goal} \subset \mathcal{X}^{free}$ and a sensor FOV θ, find a polynomial trajectory Φ such that:*

$$\begin{aligned} \arg\min_{\Phi} \;\; & J_q + \rho_T T + \rho_\psi J_\psi \\ s.t. \;\; & \dot{x}(t) = Ax(t) + Bu(t) \\ & x(0) = x_0, \; x(T) \in \mathcal{X}^{goal} \\ & x(t) \in \mathcal{X}^{free}, \quad u(t) \in \mathcal{U} \\ & |\psi(t) - \xi(t)| \leq \frac{\theta}{2} \end{aligned} \tag{5.12}$$

where the weights $\rho_T, \;\rho_\psi \geq 0$ determine the relative importance of the trajectory duration T, the yaw cost J_ψ, and its smoothness J_q.

5.2.2 Solution

Since both of the additional constraints contain ξ which is an arctan function, it is difficult to get their analytic expressions. We use a sampling method similar to the one in (5.6) to approximate the FOV constraint. The control $u_\psi \in u$ for yaw can be applied in a different control space compared to the other flat outputs. To be specific, we set u_ψ as the angular velocity assuming the robot does not need to aggressively change the heading.



(a) $\rho_\psi = 0,\ \theta = 2\pi$. (b) $\rho_\psi = 1,\ \theta = 2\pi$. (c) $\rho_\psi = 0,\ \theta = \pi/2$. (d) $\rho_\psi = 1,\ \theta = \pi/2$.

Figure 5.3: Planning from a start that faces towards right to a goal with a non-zero initial velocity (black arrow), with yaw constraint. We draw the desired yaw as a small triangle at the corresponding position. As we adjust the parameters ρ_ψ and θ, the desired yaw along the planned trajectory follows different profiles.

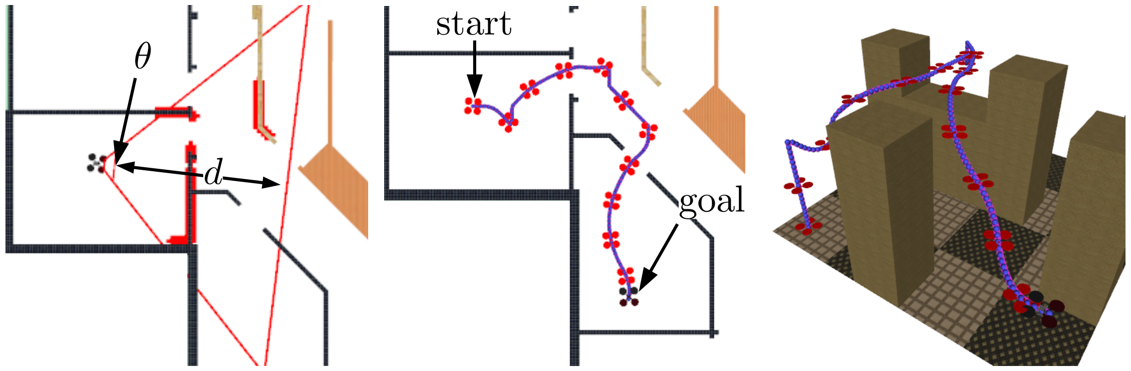
Figure 5.3 shows the planning results from solving Problem 6 with different parameters ρ_ψ, θ: in (a), we ignore the FOV constraint; in (b), we ignore the hard FOV constraint by setting $\theta = 2\pi$; in (c), we ignore the soft FOV constraint on J_ψ; and in (d), we consider both soft and hard constraints. Obviously, trajectories in (a) and (b) are not safe to follow since the robot is not always moving in the direction that the obstacle is visible within the sensor's FOV. The trajectory in Figure 5.3d is desirable as its yaw is always following the velocity direction. Besides, even though the shapes of all the trajectories in Figure 5.3 look the same, the trajectories in (c) and (d) have longer duration since the robot needs to rotate to align the yaw along the trajectory at the beginning.

5.2.3 Experimental Results

The yaw constraints can be used with the APF constraint described in Section 5.1 by adding J_c in the cost function of Problem 6 as

$$J_q + \rho_T T + \rho_\psi J_\psi + \rho_c J_c. \tag{5.13}$$

The solution to this modified problem satisfies the requirements of directional movement and safety. Similar to Section 5.1.3, we use this planner to generate and re-plan trajectories from start to goal in both 2D and 3D environments (Figure 5.4). The environment is initially unknown, and the robot uses its onboard depth sensor with a horizontal FOV θ to detect



(a) Sensor model. (b) 2D Navigation. (c) 3D Navigation.

Figure 5.4: Navigation of a quadrotor equipped with an RGB-D camera in an office environment. The red triangle in the left image indicates the sensor's FOV θ and sensing range d. The red cells in (a) stand for the points detected by the sensor. The trajectory in (b) and (c) shows the quadrotor approaching the goal with changing yaw using the proposed method.

obstacles. To be able to plan trajectories reaching the goal, the unexplored space is treated as the free space. This greedy assumption introduces the risk that the trajectory could potentially crash the robot into hidden obstacles that are outside of the sensor's FOV. Our planner is able to generate yaw movements along the trajectory such that the robot is always moving into the region within the sensor's FOV. Therefore, the robot is able to avoid hitting hidden obstacles and reach the goal safely.

5.3 Planning in SE(3)

Existing planning approaches usually model the MAV as a sphere or prism, which allows obtaining a simple configuration space (C-space) by inflating the obtacles with the robot size (Section 2.2.1). As a result, the robot can be treated as a single point in C-space and the collision-checking even for trajectories that take dynamics into account is simplified. Even though this spherical model assumption is widely used in motion planing, it is very conservative since it invalidates many trajectories whose feasbility depends on the robot attitude (Figure 5.5). Several prior works have demonstrated aggressive maneuvers for quadrotors that pass through narrow gaps [17, 30, 55] but, instead of solving the planning problem, those works focus on trajectory generation with given attitude constraints. Those constraints are often hand-picked beforehand or obtained using gap detection algorithms

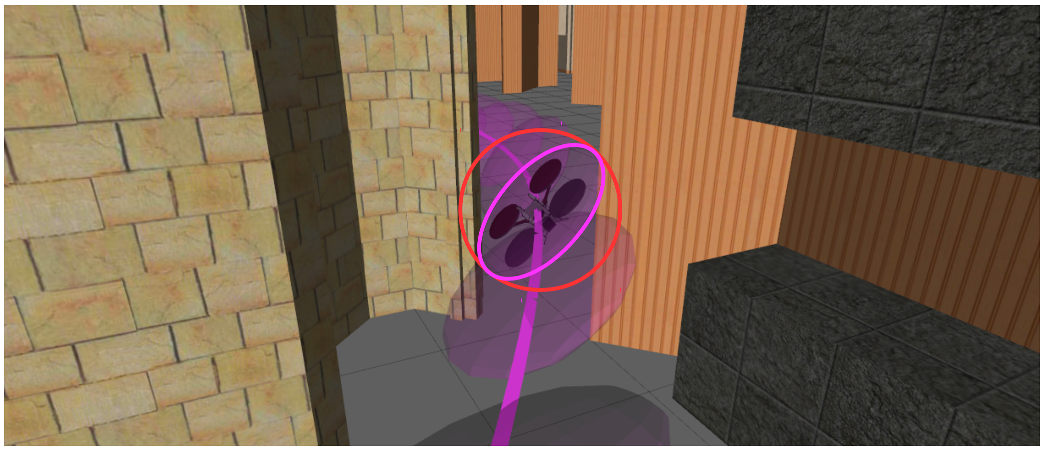


Figure 5.5: By taking the shape and dynamics of a quadrotor into account, our planner is able to generate a trajectory that allows the quadrotor to pass through a door, narrower than robot's diameter. In contrast, existing methods that conservatively model the quadrotor as a sphere (red circle) would not be able to find a feasible path in this environment.

which only works for specific cases.

We are interested in designing a planner that considers the robot's actual shape and dynamics in order to obtain aggressive trajectories in cluttered environments (Figure 5.5). Since quadrotors are under-actuated systems, they cannot translate and rotate independently. In this section, we extend our search-based motion planning by explicitly computing the robot attitude along the motion primitives and using it to enforce collision constraints.

5.3.1 System Dynamics in Planning

Before introducing the planning approach, we inspect the relation between polynomial trajectories and system dynamics. The position $\mathbf{x} = [\ x,\ y,\ z\ ]^\mathsf{T}$ in $\mathbb{R}^3$ of the quadrotor can be defined as a differentially flat output as described in [61]. The associated velocity $\mathbf{v}$, acceleration $\mathbf{a}$ and jerk $\mathbf{j}$ can be obtained by taking derivatives with respect to time as $\dot{\mathbf{x}}, \ddot{\mathbf{x}}, \dddot{\mathbf{x}}$ respectively. The desired trajectory for the geometric SE(3) controller as described in [44] can be written as $\Phi(t) = [\ \mathbf{x}_d^\mathsf{T},\ \mathbf{v}_d^\mathsf{T},\ \mathbf{a}_d^\mathsf{T},\ \mathbf{j}_d^\mathsf{T}\ ]^\mathsf{T}$. According to [29], we assume the force and angular velocity are our control inputs to the quadrotor. From Section 2.1.3, ignoring feedback control errors, the desired mass-normalized force in the inertial frame can be obtained as

$$\mathbf{f}_d = \mathbf{a}_d + g\mathbf{z}_{\mathrm{W}}. \tag{5.14}$$

where g is the gravitational acceleration and $\mathbf{z}_{\mathrm{W}} = [0, 0, 1]^{\mathsf{T}}$ is the z-axis of the inertial world frame. Similar to [44], given a specific yaw ψ, the desired orientation in SO(3) can be written as $\mathbf{R}_d = [\ \mathbf{r}_1 \mid \mathbf{r}_2 \mid \mathbf{r}_3\ ]$ where

$$\mathbf{r}_3 = \mathbf{f}_d / \|\mathbf{f}_d\|, \quad \mathbf{r}_1 = \frac{\mathbf{r}_{2c} \times \mathbf{r}_3}{\|\mathbf{r}_{2c} \times \mathbf{r}_3\|}, \quad \mathbf{r}_2 = \mathbf{r}_3 \times \mathbf{r}_1 \tag{5.15}$$

and

$$\mathbf{r}_{2c} = [\ -\sin\psi,\ \cos\psi,\ 0\ ]^{\mathsf{T}}. \tag{5.16}$$

which is assumed to be not parallel to $\mathbf{r}_3$. The associated angular velocity in the inertial frame, $\dot{\mathbf{R}}_d = [\ \dot{\mathbf{r}}_1 \mid \dot{\mathbf{r}}_2 \mid \dot{\mathbf{r}}_3\ ]$, can be calculated as

$$\begin{aligned} \dot{\mathbf{r}}_3 &= \mathbf{r}_3 \times \frac{\dot{\mathbf{f}}_d}{\|\mathbf{f}_d\|} \times \mathbf{r}_3, \\ \dot{\mathbf{r}}_1 &= \mathbf{r}_1 \times \frac{\dot{\mathbf{r}}_{2c} \times \mathbf{r}_3 + \mathbf{r}_{2c} \times \dot{\mathbf{r}}_3}{\|\mathbf{r}_{2c} \times \mathbf{r}_3\|} \times \mathbf{r}_1, \\ \dot{\mathbf{r}}_2 &= \dot{\mathbf{r}}_3 \times \mathbf{r}_1 + \mathbf{r}_3 \times \dot{\mathbf{r}}_1 \end{aligned} \tag{5.17}$$

where

$$\dot{\mathbf{r}}_{2c} = [\ -\cos\psi,\ -\sin\psi,\ 0\ ]^{\mathsf{T}} \dot{\psi}, \quad \dot{\mathbf{f}}_d = \mathbf{j}_d^{\mathsf{T}}. \tag{5.18}$$

Therefore, the desired angular velocity $\mathbf{w}_d$ in body frame is obtained as:

$$[\mathbf{w}_d]_\times = \mathbf{R}_d^{\mathsf{T}} \dot{\mathbf{R}}_d. \tag{5.19}$$

Once the desired force $\mathbf{f}_d$, orientation $\mathbf{R}_d$ and angular velocity $\mathbf{w}_d$ are defined, it is straightforward to compute the desired control inputs for the quadrotor system. Notice that: 1) orientation is algebraically related to the desired acceleration and gravity and 2) angular velocity is algebraically related to the desired jerk.

5.3.2 Collision Free Primitives

As indicated before, traditional collision checking though inflating obstacles is over-conservative and not suitable for planning agile trajectories in cluttered environments since it fails to take the actual robot shape and attitude into account. In this section, we model the quadrotor as an ellipsoid ξ in $\mathbb{R}^3$ with radius r and height h and the obstacle map as a point cloud $\mathcal{O} \subset \mathbb{R}^3$ (Figure 5.6). Given a quadrotor state $\mathbf{s}$, its body configuration ξ at $\mathbf{s}$ can be obtained as

$$\xi(\mathbf{s}) := \{ \ \mathbf{p} = \mathbf{E}\tilde{\mathbf{p}} + \mathbf{d} \mid \|\tilde{\mathbf{p}}\| \leq 1 \ \} \tag{5.20}$$

where

$$\mathbf{d} = \mathbf{x}(t), \ \mathbf{E} = \mathbf{R} \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & h \end{bmatrix} \mathbf{R}^\intercal \tag{5.21}$$

and the orientation $\mathbf{R}$ can be calculated from $\ddot{\mathbf{x}}(t)$ and gravity as shown in (5.15).

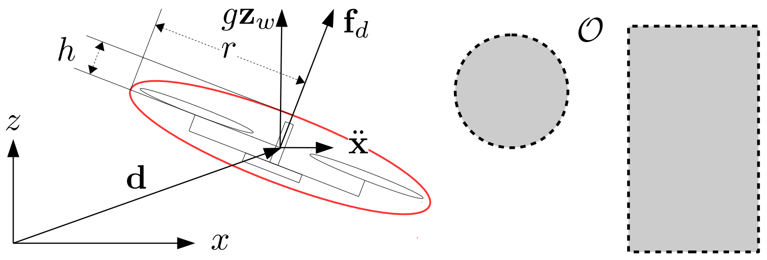


Figure 5.6: A quadrotor can be modeled as an ellipsoid with radius r and height h. Its position and attitude can be estimated from the desired trajectory. A point cloud $\mathcal{O}$ is used to represent obstacles.

Checking whether the quadrotor hits obstacles while following a trajectory is equivalent to checking if there is any obstacle inside the ellipsoid along the trajectory. In other words, we need to verify that the intersection between ξ and the point cloud $\mathcal{O}$ is empty:

$$\mathcal{O} \cap \xi = \{ \ \mathbf{o} \mid \|\mathbf{E}^{-1}(\mathbf{o} - \mathbf{d})\| \leq 1, \ \forall \mathbf{o} \in \mathcal{O} \ \} = \varnothing \tag{5.22}$$

Instead of checking through every point in $\mathcal{O}$, it is more efficient to use *KD-tree* [84] to crop a subset $\mathcal{O}_{r,\mathbf{d}}$ of $\mathcal{O}$ at first and then check the intersection between ξ and obstacles inside $\mathcal{O}_{r,\mathbf{d}}$. The subset $\mathcal{O}_{r,\mathbf{d}}$ is created by looking for neighbor points around $\mathbf{d}$ within radius r, assuming $r \geq h$.

Since the contour of an ellipsoid following a primitive is not convex, it is difficult to get an analytic approximation for the union of the contour of the ellipsoids along the primitive. Instead, as shown in Figure 5.7, we sample I states in time along a primitive F_n and consider the primitive F_n collision-free if

$$\mathcal{O} \cap \xi(\mathbf{s}_{i,n}) = \varnothing, \; \forall i = \{0, 1, \dots, I-1\} \tag{5.23}$$

where $\mathbf{s}_{i,n}$ is the i-th sampled state on F_n.

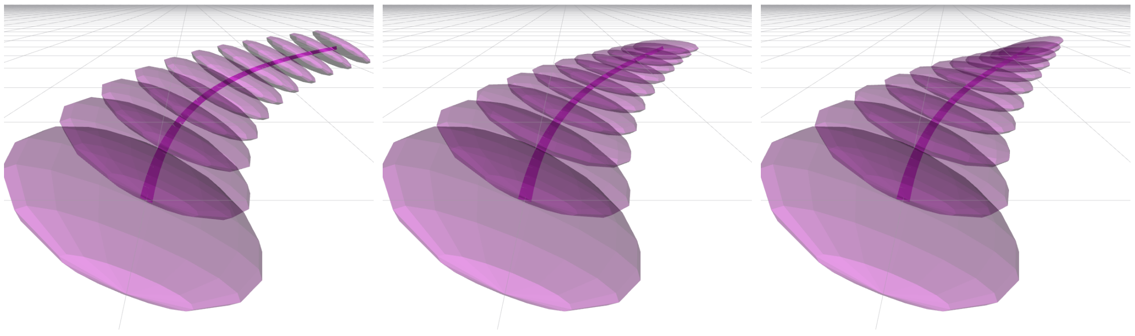

(a) Min-acceleration primitive. (b) Min-jerk primitive. (c) Min-snap primitive.

Figure 5.7: Sampe ellipsoids along different primitives.

In sum, the explicit formulation of the feasibility constraints $F_n(t) \subset \mathcal{X}^{free}$ in Problem 1 is written as:

$$F_n(t) \preceq [\; \mathbf{v}_{max}^{\mathsf{T}}, \; \mathbf{a}_{max}^{\mathsf{T}}, \; \mathbf{j}_{max}^{\mathsf{T}} \;]^{\mathsf{T}}, \tag{5.24}$$
$$\mathcal{O} \cap \xi(\mathbf{s}_{i,n}) = \varnothing, \; \forall i = \{0, 1, \dots, I\}.$$

5.3.3 Evaluation

We apply the proposed algorithms over several typical maps to evaluate planner's capability.

2D Planning 2D planning is efficient and useful in 2.5D environments where the obstacles are vertical to the floor. We start by showing 2D planning tasks of flying though gaps with different widths. In Figure 5.8, we show how planned trajectories using jerk as a control input vary as the gap in a wall is shrinking (left wall moves closer to the right wall from (a) to (f)). Accordingly, the angle of the desired roll at the gap ϕ_{gap} increases. Assume the robot has radius $r = 0.35$m, height $h = 0.1$m, and the maximum acceleration in each axis is $a_{max} = g$. Denoting the roll along trajectory as ϕ, according to (5.14) and (5.15), we have

$$-\arctan\frac{a_{max}}{g} \leq \phi \leq \arctan\frac{a_{max}}{g} \tag{5.25}$$

since the desired acceleration in z-axis is zero. In other words, the smallest gap that the robot can pass through using 2D planning is approximately equal to $2r\cos\theta$ (which is approximately equal to 0.525m).

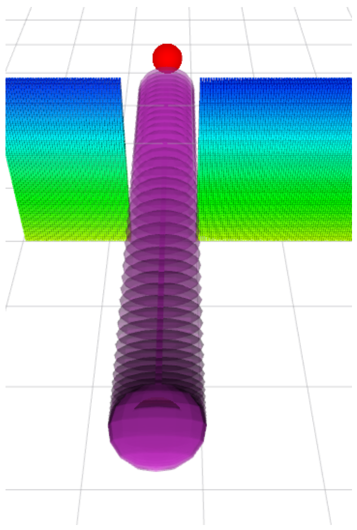
3D Planning By adding control in the z-axis, we are able to plan in 3D space and relax the constraint in (5.25) as follows:

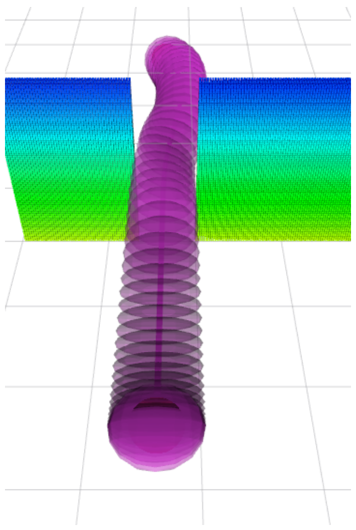
$$-\arctan\frac{a_{max}}{g - a_{max}} \leq \phi \leq \arctan\frac{a_{max}}{g - a_{max}}. \tag{5.26}$$

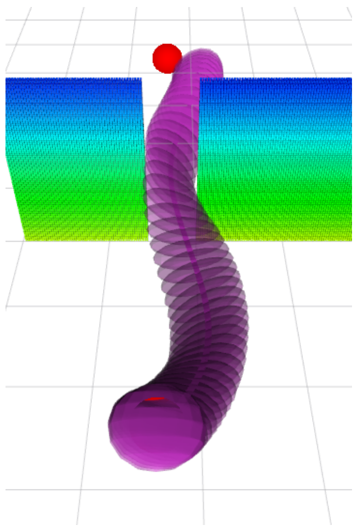
When $a_{max} \geq g$, $\phi \in (-\frac{\pi}{2}, \frac{\pi}{2}]$ can be arbitrary. Letting $a_{max} = g$, we are able to reduce the gap width even more as shown in the following Figure 5.8.

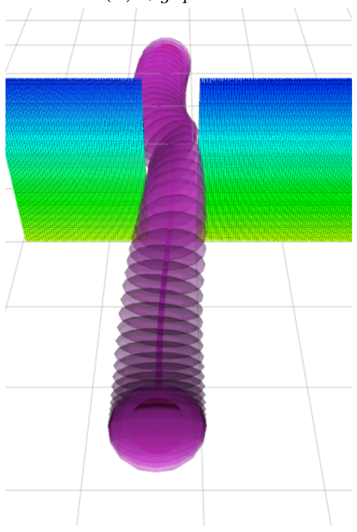
Another example of 3D planning using a window with a rectangular hole in the middle is considered. By modifying the window's inclination ϕ_{win}, we are able to verify the planner's capability to generate trajectories as shown in Figure 5.9.

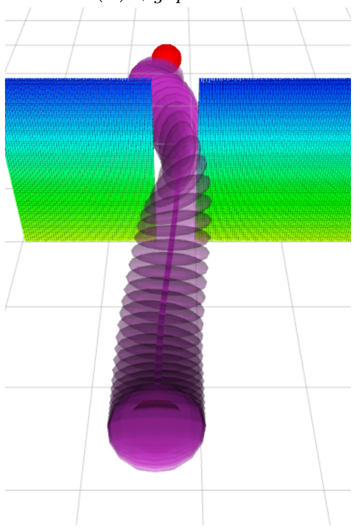
Parameters There are a few parameters that significantly affect the planning performance including computation time, resolution completeness, continuity and dynamics constraints. In this section, we analyze these relationships and provide a rough guidance on how to set the parameters in our planner. In the above examples of 2D and 3D planning, we used the following settings (here the control input is defined as jerk such that $u_{max} = j_{max}$):


(a) $\phi_{gap} = 0°$


(b) $\phi_{gap} = 27°$


(c) $\phi_{gap} = 45°$


(d) $\phi_{gap} = 46°$


(e) $\phi_{gap} = 73°$

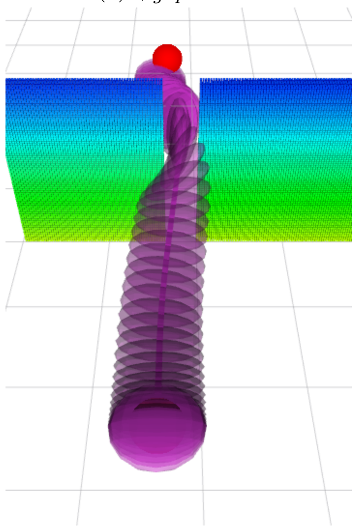

(f) $\phi_{gap} = 90°$

Figure 5.8: Trajectories through gaps with different widths: $0.75, 0.65, 0.55$m from (a) to (c) (2D planning) and $0.55, 0.45, 0.35$m from (d) to (f) (3D planning). ϕ_{gap} indicates the maximum roll at the gap. Red dots show the start and goal.

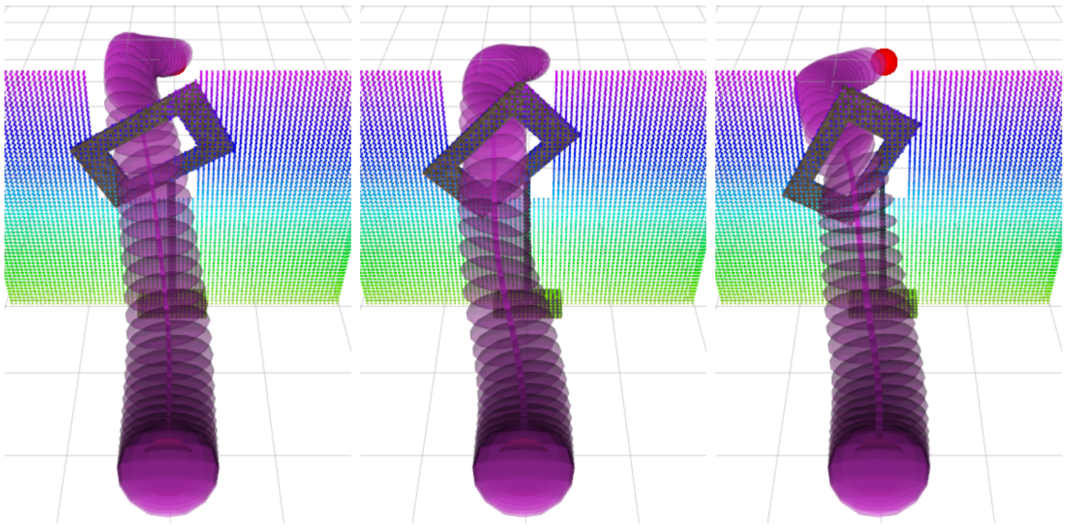
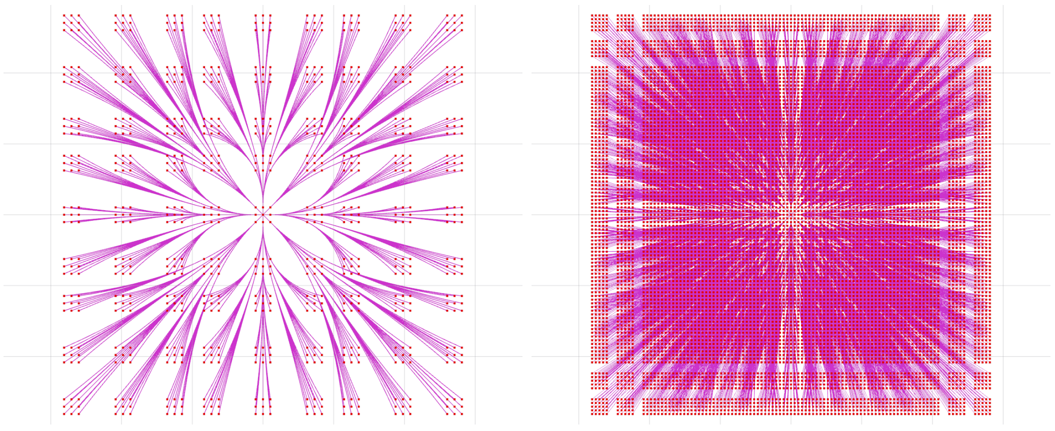

(a) $\phi_{win} = 30°$ (b) $\phi_{win} = 45°$ (c) $\phi_{win} = 60°$

Figure 5.9: Trajectories generated through a rectangular hole of size 0.4×0.8m oriented at different angles. A robot with radius $r = 0.35$m needs to fly through the hole with certain non-zero roll and pitch angles. The colored dots represent walls in the map that invalidate trajectories that go around the window.

ρ	τ	v_{max}	a_{max}	u_{max}	du
10000	0.2s	7m/s	10m/s^2	50m/s^3	12.5m/s^3

A larger ρ results in faster trajectories. The scale of ρ should be comparable to the scale of the associated control effort. Here we use $\rho \approx 4u_{max}^2$. The motion primitive duration τ decided the density of the lattices and computation time, for moderate flight speed ($<$ 10 m/s), we find $\tau = 0.2$s to be a reasonable choice. A small τ makes the graph dense and requires more explorations to reach the goal, while a large τ may easily result in searching failure since the graph may be too sparse to cover the feasible region. The discretization in the control space $\mathcal{U}_M$ also affects the density of the graph as shown in Figure 5.10. Its effect is similar to τ – finer discretization in $\mathcal{U}_M$ leads to a slower but more complete search and smoother trajectories and vice versa. The maximum velocity and acceleration are limited by the system's dynamics including thrust-to-weight ratio, max angular speed and air drag etc, but in many cases, we also want to limit the agility due to the space, state estimation and control limitations.



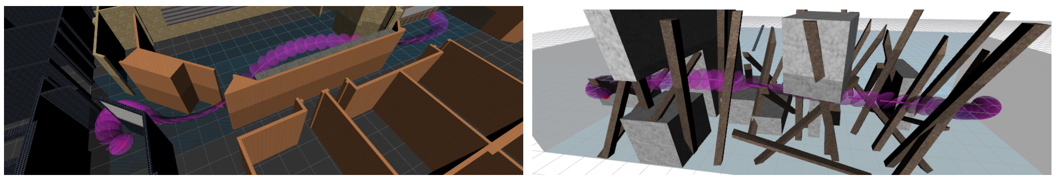
(a) $\tau = 0.5$, $|\mathcal{U}_M| = 9$. (b) $\tau = 0.5$, $|\mathcal{U}_M| = 25$.

Figure 5.10: Graph $\mathcal{G}(\mathcal{S}, \mathcal{E})$ generated by applying BFS for a finite planning horizon over a set of motion primitives $\mathcal{U}_M$ with 9 elements (a) and 25 elements (b). Red dots represent states in $\mathcal{S}$ and magenta splines represent edges in $\mathcal{E}$.

5.3.4 Experimental Results

Simulation Results

The proposed planner is used to generate trajectories in complicated environments as shown in Figure 5.11. A geometric model of the environment is converted into a point cloud and used to construct an obstacle *KD-tree* with 5cm resolution.



(a) Office environment. (b) Unstructured environment.

Figure 5.11: Generated trajectories in two different environments. The robot radius is $r = 0.5$m, making its diameter much larger than the door width in (a). If the obstacles in these environments are inflated by r, no feasible paths exist.

In general, finding the optimal trajectories in complicated environments like Figure 5.11 is slow (Table 5.1 gives the computation time of trajectory planning on a moderate fast computer with an Intel i7 processor with clock rate of 3.4GHz). As proposed in Section 4.4.2, we plan trajectories Φ^2 using acceleration control at first, based on which we plan the trajectory Φ^3_* using jerk control. As shown in Table 5.1, the computation time for hierarchical planning is much less than that for planning in the original 9 dimensional space with jerk input. We can also see in Figure 5.12 that the refinement process tends to explore fewer states. As expected, the refined trajectory Φ^3_* has a higher cost compared to the optimal trajectory Φ^3. While use trajectories planned in the lower dimensional space as heuristics for searching in the higher dimensional space will make the trajectories sub-optimal, the algorithm guarantees safety and completeness.

	Office			Unstructured 3D		
	t(s)	$J(\times 10^3)$	T(s)	t(s)	$J(\times 10^3)$	T(s)
Φ^3	89.42	8.9	4.6	129.58	5.6	3.0
Φ^2	9.34	0	4.4	21.64	0	3.6
Φ^3_*	2.03	11.1	5.0	24.02	15.1	4.8

Table 5.1: Evaluation: t refers to the computation time, J is the total control (jerk) effort and T is the total trajectory execution time.

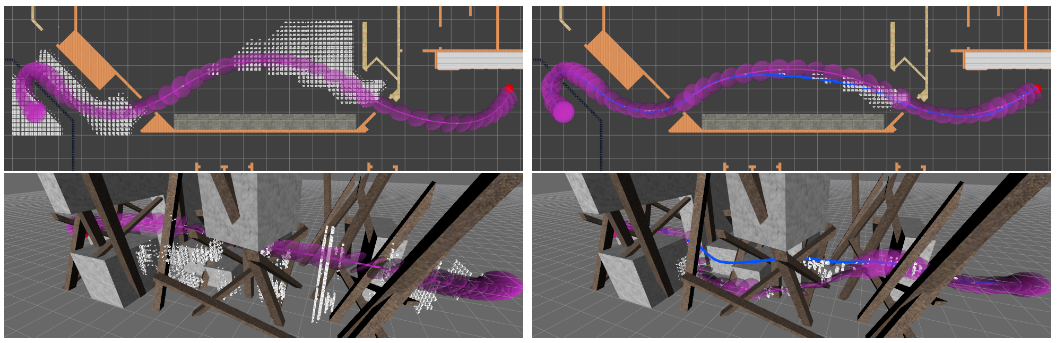


Figure 5.12: Comparison between the optimal method (left) and refinement (right). The prior trajectory Φ^2 is plotted in blue, while the white dots indicate explored states. It is clear that the refinement explores fewer irrelevant regions but the generated trajectory is suboptimal.

5.3.5 Real World Experiments

The experiments is aiming to demonstrate the feasibility of planned trajectories on a real robot. We use AscTec Hummingbird as our quadrotor platform, we also use VICON motion capture system to localize the quadrotor and the obstacle map is obtained by depth sensor in advance to generate trajectories. The robot is able to avoid hitting obstacles by following the control commands extracted from the planned trajectory, received via wireless. Figure 5.13 shows a flight when the quadrotor needs to roll aggressively in order to pass through the gap between white boards.

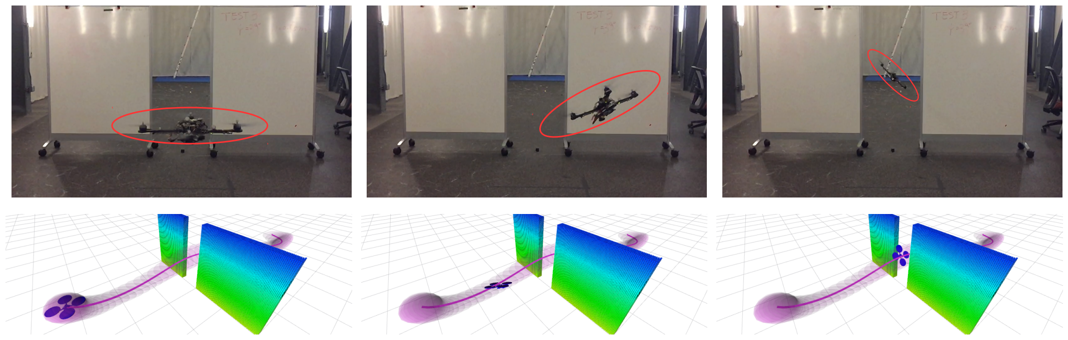
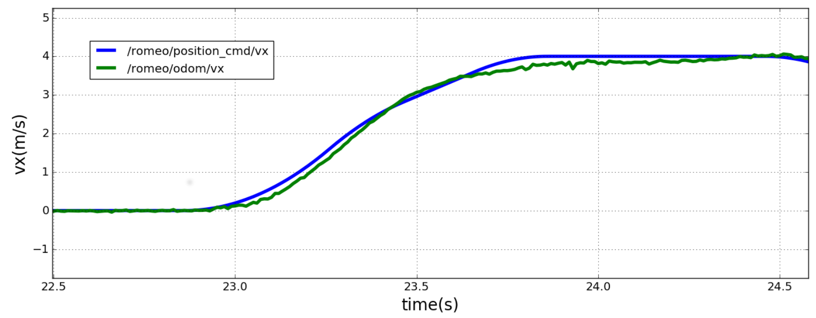


Figure 5.13: Quadrotor tracks the planned trajectory to fly through a narrow gap. Top figures are the snapshots of the video, bottom figures are corresponding visualizations in ROS. Maximum roll angle at the gap is 40° as drawn in the top right figure.

The control errors in velocity and roll are plotted in Figure 5.14. The commanded roll includes the feedback attitude errors such that it is not as smooth as the desired roll from the planned trajectory. The existed lag in the attitude is due to the fact that the actual robot is not able to achieve specified angular velocity instantly, however for a moderate angular speed, this assumption still holds valid. A more accurate model for the quadrotor should use snap as the control input instead of the jerk. The trajectory planned using the snap as the control input is straightforward to solve following the same pipeline as proposed in this thesis, which has also been implemented in our open-sourced planner.



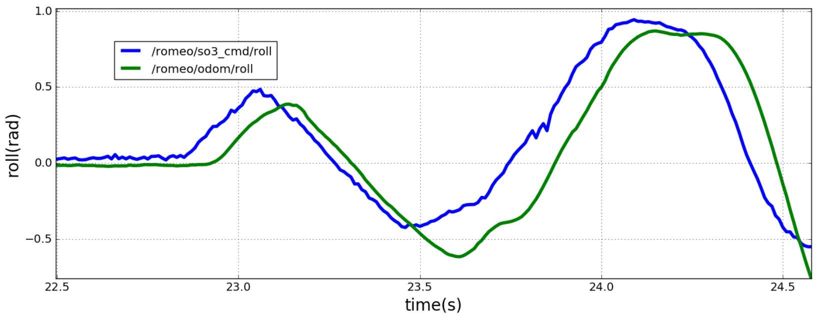


Figure 5.14: Plots of control errors, the blue curve is the command value while the green curve shows the actual robot state. The top figure shows $v_x - t$, and the bottom figure shows $\phi - t$.

Chapter 6

Search-based Trajectory Planning in Dynamic Environments

All planning problems addressed so far involved a static map. Ensuring optimality and completeness in environments with mobile obstacles is much harder [79]. Existing planning methods based on fast re-planning including [21, 34, 39, 87] or safe interval [69, 74] are neither complete nor optimal. Reactive collision avoidance using the concept of *velocity obstacle* (VO) [20, 83, 97, 98, 107] discards trajectories' global optimality and completeness to gain the guarantee of flight safety and real-time computation. However, these VO based frameworks assume a simple straight line path with constant velocity and cannot be used to follow a dynamically feasible trajectory for systems with second or higher order dynamics.

In this chapter, we directly solve the planning problem in a dynamic environment using our search-based framework which is resolution optimal and complete. To ensure the flight safety, the robot needs to re-plan since the information of surrounding moving obstacles is being constantly updated. We will model a moving obstacle as the *linear velocity polyhedron* (LVP) in $\mathbb{R}^m$ whose position and velocity are observable as explained later in Figure 6.1. In fact, a linear model for a moving obstacle is only an approximation of its motion in the general case. To increase the success of future re-plans, we inflate the LVP with respect to time. In the meanwhile, to avoid wasting time searching over the same region repeatedly, we use an incremental trajectory planning approach based on Lifelong Planning A* (LPA*) [40]. The proposed planner can further be developed for planning for multi-robot systems, in which the inter-robot collision avoidance is guaranteed.

6.1 Planning with Moving Obstacles

As shown in Figure 6.1, a single translating obstacle can be represented by a *linear velocity polyhedron* c in $\mathbb{R}^m$ with velocity v_c (no rotation). We first show that the collision between a polynomial trajectory Φ and c can be checked by solving for roots of a polynomial. We then describe our model of motion uncertainty using the concept of the LVP, and show its use in re-planning.

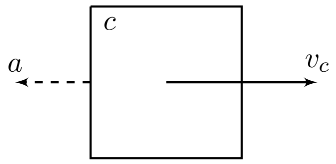


Figure 6.1: A 2D example of a *linear velocity polyhedron* (LVP) which is a convex polyhedron with velocity v_c. a indicates the outward normal of the attached half-space.

6.1.1 Collision Checking against LVPs

Denote a half-space in $\mathbb{R}^m$ as $h = \{\ p \mid a^\mathsf{T} p \leq b,\ p \in \mathbb{R}^m\ \}$. The intersection of M half-spaces gives a convex polyhedron, $c = \bigcap_{j=0}^{M-1} h_j = \{\ p \mid \mathbf{A}^\mathsf{T} p \leq \mathbf{b},\ p \in \mathbb{R}^m\ \}$, where a_j corresponding to h_j is the j-th column of matrix $\mathbf{A}$ and b_j is the j-th element of vector $\mathbf{b}$. The LVP is described as a polyhedron c with a velocity v_c, thus b_j is time-varying if v_c is non-zero. Denote $h_j^0 = \{\ p \mid a_{j,0}^\mathsf{T} \cdot p \leq b_{j,0}\ \}$ as the initial half-space $h_j^0 = h_j(t = 0)$, we can mathematically define the LVP by:

$$a_j(t) = a_{j,0},\ b_j(t) = b_{j,0} + a_{j,0}^\mathsf{T} \cdot v_c t. \tag{6.1}$$

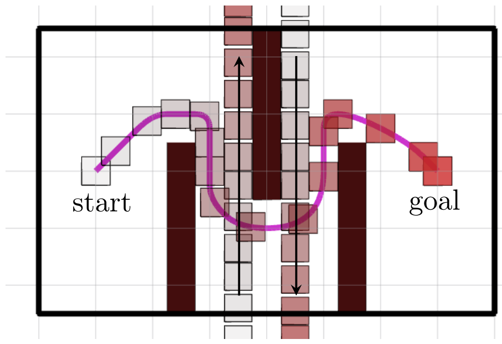
If a polynomial trajectory Φ defined by (3.1) collides with a polyhedron c, we must have one of its trajectory segments Φ_i intersect c in the time interval $[\ t_i,\ t_{i+1}\ ]$. It can be verified by finding roots of the polynomial function of time $a_j^\mathsf{T} \Phi_i(t) = b_j(t)$: if there exists a root t_c

located in the interval $[\ 0,\ t_{i+1} - t_i\ ]$ and the intersecting point $\Phi_i(t_c)$ (*i.e.,* $\Phi(t_i + t_c)$) is on the boundary of c, we claim that Φ collides with c.

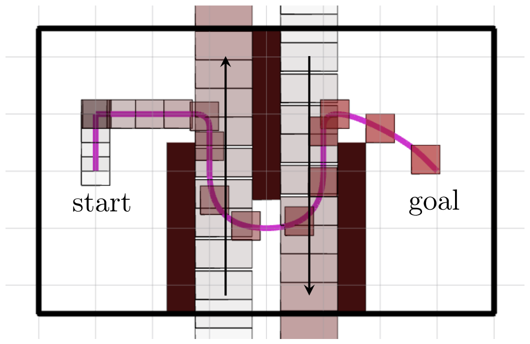
Proposition 4. *A trajectory segment $\Phi_i(t)$ intersects a polyhedron c that is composed of half-spaces described by $\mathbf{A}$ and $\mathbf{b}$ if and only if*

$$\exists\ t_c \in [\ 0,\ \Delta t_i\ ] \quad s.t. \quad \mathbf{A}^{\mathsf{T}} \Phi_i(t_c) \leq \mathbf{b} \tag{6.2}$$

Since we assume that v_c is constant for each planning interval, $b_j(t)$ is a time-parameterized polynomial function. Therefore, we are still able to solve roots from the polynomial function in (6.2). Figure 6.2 shows the planning results in two configurations with LVPs. For better visualization, the animation of robot following planned trajectories in corresponding dynamic environments is shown in the video at https://www.youtube.com/watch?v=2u08T3j3iwg.



(a) Configuration 1.



(b) Configuration 2.

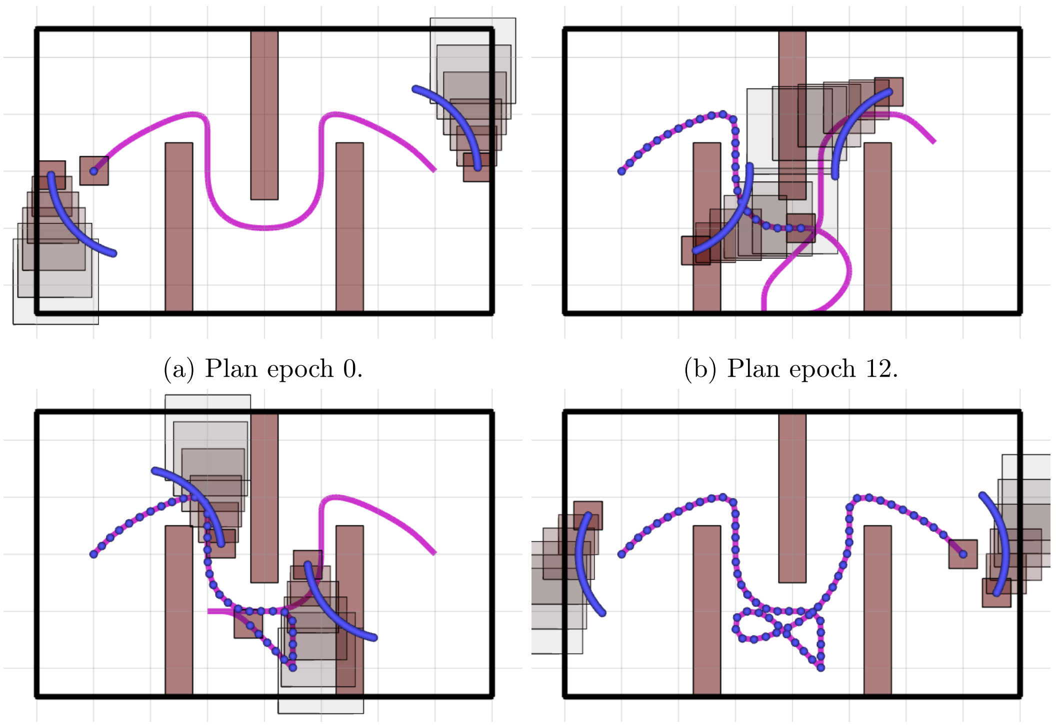
Figure 6.2: Planning with linearly moving obstacles. We use different transparencies to represent positions of moving obstacles and robot at different time stamps. The moving obstacles in configuration 2 is wider than configuration 1, thus the planned trajectory in configuration 2 lets the robot wait for the first obstacle passing through the tunnel instead of entering the tunnel in parallel.

6.1.2 Uncertainty of Linear Polyhedra

Since the movement of a moving obstacle is unpredictable, our LVP model in (6.1) is only a prediction for the purposes of re-planning. To address this problem, we use a simple but effective strategy similar to [100] that grows the obstacle's geometry: shift all the half-spaces in the direction of the outward normal with certain speed $v_e > 0$. As a result, (6.1) is modified as:

$$a_j(t) = a_{j,0}, \; b_j(t) = b_{j,0} + (a_{j,0}^{\mathsf{T}} \cdot v_c + \|a_{j,0}\| v_e)t. \tag{6.3}$$

Substituting (6.3) into (6.2), we can still get a polynomial function to check for collision. An example of growing obstacles is illustrated in Figure 6.3 where the robot constantly re-plans at 1 Hz. The robot is able to avoid the non-linearly moving obstacles with the proposed linear model in (6.3) with a properly selected v_e.


(a) Plan epoch 0.

(b) Plan epoch 12.

(c) Plan epoch 15.

(d) Plan epoch 20.

Figure 6.3: Re-planning in a map with 2 moving obstacles. The blue splines show the future trajectories of moving obstacles which are unobservable. The stacked transparent rectangles indicate the evolution of the moving obstacles as predicted by (6.3), which are used in re-planning.

6.1.3 Problem Formulation

For a general planning problem in the environment that has both static and moving obstacles, we separate the collision checking in two workspaces: a static workspace $\mathcal{X}_s$ and a dynamic workspace $\mathcal{X}_d(t)$. $\mathcal{X}_s$ can be represented by a standard map which contains a collision-free subset $\mathcal{X}_s^{free}$ and an occupied subset $\mathcal{X}_s^{obs} = \mathcal{X}_s \setminus \mathcal{X}_s^{free}$. $\mathcal{X}_d(t)$ is a time-varying set whose occupied subset consists of K moving obstacles $\mathcal{X}_d^{obs}(t) = \bigcup_{k=0}^{K-1} c_k(t)$. Denote the free subset as $\mathcal{X}_d^{free}(t) = \mathcal{X}_d(t) \setminus \mathcal{X}_d^{obs}(t)$, the original collision constraint $x(t) \in \mathcal{X}^{free}$ in Problem 4 is re-written as:

$$x(t) \in \mathcal{X}_s^{free}, \text{ and } x(t) \in \mathcal{X}_d^{free}(t). \tag{6.4}$$

Since the collision checking is a function of time, the lattice state in Problem 4 should be augmented by the corresponding time stamp. In such case, a maximum planning horizon T_{max} is the criterion to determine if the search should be terminated or not. Otherwise, if the goal is occupied permanently by moving obstacles, the planner will keep expanding the same state at different time stamps.

6.2 Incremental Trajectory Planning

A planned trajectory needs to be updated when new information of moving obstacles is updated in order to guarantee safety and optimality. Re-planning from scratch every time is not efficient since we may waste time searching places that were already explored in previous planning epochs. To leverage incremental search techniques for dynamic systems, we replace the A* with Lifelong Planning A* (LPA*) [40]. For searching with motion primitives, an additional *graph pruning* process is necessary to maintain the correctness and optimality of the planning results. By combining LPA* and *graph pruning*, we can efficiently solve the re-planning problem in a dynamic environment.

D* [88], D* Lite [39] and LPA* [40] are the most popular incremental graph search algorithms used in practice. Even though they are more popular than LPA*, the D* and D* Lite algorithms are not suitable for incremental graph search in the state lattice graph described in the previous section. This is due to the input discretization leading to a state lattice where it is almost impossible to find a trajectory that exactly reaches the goal state. Thus, the trajectory planned from the goal state to start (as done in D* and D* Lite) will not be able to exactly reach the start state. We cannot have discontinuities in the trajectory for it to be dynamically feasible for the quadrotor. Considering this, we decided to use LPA* for the incremental graph search. Similar to A*, LPA* always determines a shortest path from a given start state to a goal state, given the graph and edge costs.

When using graph search for trajectory planning, it is usually not easy to construct the graph. Most of the computation time during planning is spent in finding successors for the nodes in the graph and checking for collisions along the edges. Thus, if we are able to keep the graph from a previous plan and update its edge costs as the map changes, we should be able to save a significant amount of computation time in the new planning query. LPA* is designed to solve this exact problem. In LPA*, once the graph has local inconsistency due to a map update, the algorithm will re-expand only the affected states until all the locally inconsistent states become consistent. Section 6.2.2 provides an overview of the LPA* algorithm for those who are not familiar with it. In the following subsections, we discuss the two places in our algorithm that are different from the original LPA*.

6.2.1 Notation

Before describing the algorithms, we introduce the notation used in the following pseudocode. A graph is composed of state lattices and directional edges [75]. Denote the set of state lattices as $\mathcal{U}$, a state lattice $u \in \mathcal{U}$ is defined as the combination of position, velocity and acceleration in $\mathbb{R}^3$ as $u := \{\mathbf{p}, \mathbf{v}, \mathbf{a}\}$ (Section 4.2.1).

Two states are equal if their coordinates of position, velocity and acceleration are equal. In the pseudocode, we use the symbol "$\approx$" to indicate that the two states u_1, u_2 are close to each other.

We use the following attributes for state $u \in \mathcal{U}$:

$\mathrm{g}(u)$	total cost to reach u from the start state
$\mathrm{rhs}(u)$	one-step look-ahead value based on $\mathrm{g}(u)$
$\mathrm{Succ}(u)$	one-step successors
$\mathrm{Pred}(u)$	one-step predecessors

Let $e(u, v)$ be the directional edge connects the state u to its successor v, denote its cost as $\mathrm{c}(u, v)$ which should be positive. We set $\mathrm{c}(u, v)$ to infinity if the corresponding edge collides with any obstacle, otherwise it should be a finite cost as the weighted sum of duration and control effort.

Similar to the original LPA*, a priority queue $\mathcal{Q}$ is used to store states in the open set of $\mathcal{U}$. The following functions are used to manage $\mathcal{Q}$:

Top()	returns a state with the smallest priority
TopKey()	returns the smallest priority value
Pop()	deletes the state with the smallest priority
Remove(u)	removes the state u from $\mathcal{Q}$
Insert(u, f)	inserts the state u with priority f

6.2.2 LPA* with motion primitives

The pseudocode of LPA* with motion primitive is presented in Algorithm 5. The *Main* procedure is the same as the original LPA*, our graph is initialized as empty in procedure *InitializeGraph* and is incrementally constructed in function *ComputeTrajectory*. In function *ComputeTrajectory*, since the arbitrary goal state g is almost impossible to be exactly connected in the state lattice graph, we use an intermediate goal state u_g to indicate the termination of the search and recover a trajectory. We initialize u_g at first with arbitrarily coordinates but infinite g and rhs values. The loop starts from line 3 and won't stop until the condition at line 18 is satisfied. This condition set u_g as the current state u if u is close to the goal g. We process *UpdateState* procedure for the current state u before updating its successors. The successors of u are expanded at line 9 through the function *GetSuccessors*, in which we propagate u using motion primitives as described in Section 4.2.1. We only keep the successor s if the edge $e(u, s)$ obeys the dynamic constraints. If $e(u, s)$ hits obstacles, we append s in Succ(u) but with an infinite edge cost. In addition, since the predecessor of s is not given, we need to append u into Pred(s) at line 14.

The functions used in Algorithm 5 are detailed as followings:

These procedures are almost the same as the original LPA*. The function *CalculateKey* returns the priority value for sorting the priority queue $\mathcal{Q}$. For each state u, we calculate its heuristic according to Section 4.3.1 once and store it to avoid additional computation in the future. The function *ComputeTrajectory* returns the trajectory Φ which is simply recovered from tracing back along the shortest trajectory from u_g in the function *RecoverTrajectory*.

Algorithm 5 LPA* with motion primitives. Given the start u_s and goal g, it finds the optimal trajectory Φ.

```
1: function ComputeTrajectory()
2:     InitializeState($u_g$);
3:     while $\mathcal{Q}$.TopKey() < CalculateKey($u_g$) or $g(u_g) \neq \mathrm{rhs}(u_g)$ do
4:         $u \leftarrow \mathcal{Q}$.Pop();
5:         if $g(u) > \mathrm{rhs}(u)$ then $g(u) \leftarrow \mathrm{rhs}(u)$;
6:         else $g(u) \leftarrow \infty$, UpdateState($u$);
7:         end if
8:         Succ($u$) $\leftarrow$ GetSuccessors($u$);
9:         for all $s \in$ Succ($u$) do
10:            if $s \notin \mathcal{U}$ then InitializeState($s$);
11:            end if
12:            if $u \notin$ Pred($s$) then
13:                Pred($s$) $\leftarrow \{u\} \cup$ Pred($s$);
14:            end if
15:            UpdateState($s$);
16:        end for
17:        if $u \approx g$ then
18:            $u_g \leftarrow u$;
19:        end if
20:    end while
21:    return RecoverTrajectory($u_g$);
22: end function

23: procedure Main()
24:    InitializeGraph();
25:    while $u_s \not\approx g$ do
26:        $\Phi \leftarrow$ ComputeTrajectory();
27:        Wait for changes in edge costs or goal;
28:        for all $e(u, v)$ that has changed cost do
29:            UpdateState($v$);
30:        end for
31:    end while
32: end procedure
```

```
1: procedure InitializeGraph()
2:     $\mathcal{Q}$, $\mathcal{U} \leftarrow \emptyset$;
3:     InitializeState($u_s$);
4:     rhs($u_s$) $\leftarrow 0$, $\mathcal{Q}$.Insert($u_s$, CalculateKey($u_s$));
5: end procedure
```

```
procedure $InitializeState(u)$
    if $u \notin \mathcal{U}$ then
        $\mathcal{U} \leftarrow \{u\} \cup \mathcal{U}$;
    end if
    $\mathrm{g}(u) \leftarrow \infty,\ \mathrm{rhs}(u) \leftarrow \infty$;
    $\mathrm{Succ}(u) \leftarrow \emptyset,\ \mathrm{Pred}(u) \leftarrow \emptyset$;
end procedure
```

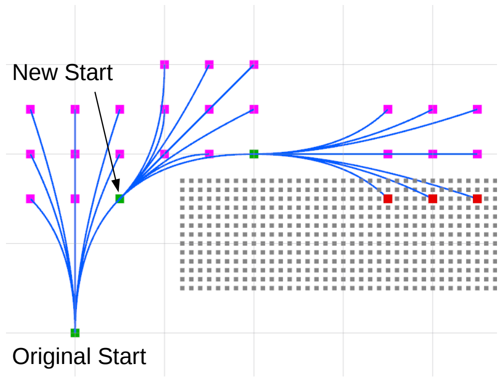
```
function $CalculateKey(u)$
    return $\min(\mathrm{g}(u), \mathrm{rhs}(u)) + CalculateHeuristic(u, g)$;
end function
```

```
procedure $UpdateState(u)$
    if $\mathrm{rhs}(u) \neq 0$ then
        $p \leftarrow \arg\min_{p' \in \mathrm{Pred}(u)}(\mathrm{g}(p') + \mathrm{c}(p', u))$;
        $\mathrm{rhs}(u) \leftarrow \mathrm{g}(p) + \mathrm{c}(p, u)$;
    end if
    if $u \in \mathcal{Q}$ then
        $\mathcal{Q}.\mathrm{Remove}(u)$;
    end if
    if $\mathrm{g}(u) \neq \mathrm{rhs}(u)$ then
        $\mathcal{Q}.\mathrm{Insert}(u, CalculateKey(u))$;
    end if
end procedure
```

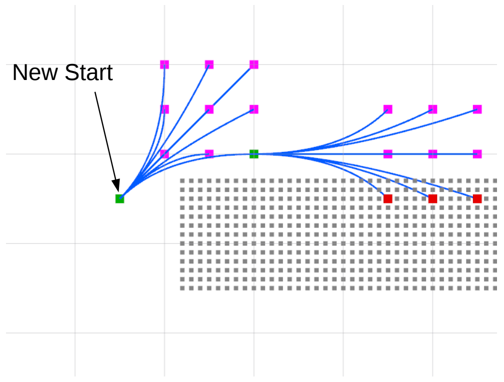
```
function $RecoverTrajectory(u)$
    $\Phi \leftarrow \emptyset$;
    while $\mathrm{Pred}(u) \neq \emptyset$ do
        $p \leftarrow \arg\min_{p' \in \mathrm{Pred}(u)}(\mathrm{g}(p') + \mathrm{c}(p', u))$;
        $\Phi \leftarrow \{e(p, u),\ \Phi\}$;
        $u \leftarrow p$;
    end while
    return $\Phi$;
end function
```

6.2.3 Graph Pruning

When the robot starts following the planned trajectory, the start state for the new planning moves to the successor of the previous start state along the planned trajectory. Now, since the start state for the graph has been changed, the start-to-state cost (namely, the g and rhs values, see Section 6.2.1) of all the nodes needs to be updated. Since our graph is directional (edges are irreversible), a large portion of the existing graph that originated from the old start state becomes unreachable. In addition, the states which remain in the new graph can be affected by the existing graph. Hence, it is important to prune an existing graph and update the start-to-state cost of the remaining states according to the new start state. In this step, we do not need to explore new states, check for collision against map or calculate any heuristic values, thus it is very fast.



(a) Original Graph



(b) Pruned Graph

Figure 6.4: An example of graph pruning. Green, magenta and red dots indicate the states that are closed, in the open set and created.

In an existing graph, a node u can be in one of three modes: 1) in the open set, 2) has been closed or 3) created, but neither opened nor closed. The third status indicates that the node is blocked by obstacles while being created. We don't insert these nodes in the open set in order to save memory and computation time. We show an example of the result from pruning a existing graph in following Figure 6.4, in which we indicate the status of each node using different colors.

The graph pruning process works by going through all the states in the graph that have been closed. By reopening these states, we update their and their successors' g and rhs values. The pseudocode of graph pruning is presented in Algorithm 6.

We run the procedure *ResetGraph* at first to reset all the attributes except successors of all the states in the existing graph. Succ(u) is kept to re-expand the relevant parts later. At line 6, we initialize g and rhs value of the new start state u'_s as zero. To build the new graph, we create a temporary set of states $\mathcal{U}_{tmp}$ and a priority queue $\mathcal{Q}_{tmp}$. $\mathcal{U}_{tmp}$ stores

Algorithm 6 *Graph Pruning.* Prune the existing graph and update start-to-state costs according to the new start state u'_s.

```
1:  procedure ResetGraph()
2:      for all $u \in \mathcal{U}$ do
3:          $\mathrm{g}(u) \leftarrow \infty, \mathrm{rhs}(u) \leftarrow \infty$;
4:          $\mathrm{Pred}(u) \leftarrow \emptyset$;
5:      end for
6:      $\mathrm{g}(u'_s) \leftarrow 0,\ \mathrm{rhs}(u'_s) \leftarrow 0$;
7:  end procedure
8:  procedure PruneGraph()
9:      ResetGraph();
10:     $\mathcal{U}_{tmp} \leftarrow \{u'_s\}$;
11:     $\mathcal{Q}_{tmp} \leftarrow \emptyset,\ \mathcal{Q}_{tmp}.\mathrm{Insert}(u'_s, \mathrm{rhs}(u'_s))$;
12:     while $\mathcal{Q}_{tmp} \neq \emptyset$ do
13:         $u \leftarrow \mathcal{Q}_{tmp}.\mathrm{Pop}()$;
14:         for all $s \in \mathrm{Succ}(u)$ do
15:             if $s \notin \mathcal{U}_{tmp}$ then $\mathcal{U}_{tmp} \leftarrow \{s\} \cup \mathcal{U}_{tmp}$;
16:             end if
17:             $\mathrm{Pred}(s) \leftarrow \{u\} \cup \mathrm{Pred}(s)$;
18:             if $\mathrm{rhs}(u) + \mathrm{c}(u, s) < \mathrm{rhs}(s)$ then
19:                 $\mathrm{rhs}(s) \leftarrow \mathrm{rhs}(u) + \mathrm{c}(u, s)$;
20:                 if $\mathrm{closed}(s)$ then
21:                     $\mathrm{g}(s) \leftarrow \mathrm{rhs}(s)$;
22:                     $\mathcal{Q}_{tmp}.\mathrm{Insert}(s, \mathrm{rhs}(s))$;
23:                 end if
24:             end if
25:         end for
26:     end while
27:     $\mathcal{Q} \leftarrow \emptyset$;
28:     for all $u \in \mathcal{U}_{tmp}$ do
29:         if $\mathrm{opened}(u)$ and not $\mathrm{closed}(u)$ then
30:             $\mathcal{Q}.\mathrm{Insert}(u, \mathit{CalculateKey}(u))$;
31:         end if
32:     end for
33:     $\mathcal{U} \leftarrow \mathcal{U}_{tmp}$;
34: end procedure
```

the states of the new graph, $\mathcal{Q}_{tmp}$ is used to sort the states to expand. As described in previous paragraph, a state $u \in \mathcal{U}$ can have one of three status. We record the status of u in the existing graph by two labels: $\mathrm{opened}(u)$ and $\mathrm{closed}(u)$ and we have the following relationship between status and label:

Status	opened(u)	closed(u)
in open set	True	False
closed	True	True
neither	False	False

The construction of $\mathcal{U}_{tmp}$ from line 12 to 26 is a variation of the Dijkstra algorithm where we only expand the state that has been closed in the existing graph. From line 27 to 32, after expanding all the closed states on the branch of u'_s, we recreate the priority queue for the new graph. We use the same function *CalculateKey* in Algorithm 5 to get the priority value. Replace the existing $\mathcal{U}$ with the new $\mathcal{U}_{tmp}$ at line 33, we have built the new graph that is valid for future plan.

6.2.4 Map Association

In our application, the state lattice graph for trajectory planning is independent from the map that is used for collision checking. Hence, we need to associate the state lattice graph with the map in order to correctly update edge costs.

Occupancy Grid Map

The occupancy grid map is commonly used in robot navigation, since it is easily constructed from the sensor data. Each cell in an occupancy grid map has a binary state of either being free or occupied. Along each edge in the graph, we sample cells that the edge goes through and for each cell in the map we create a list to store the associated edges. As a result, whenever a cell changes its state, we are able to retrieve the list of affected edges and update the graph accordingly. In Figure 6.5 we show an example of this process. The figure illustrates two facts: (1) a cell may be associated with multiple edges and (2) we only update the end states of edges when the map changes.

There are two types of edge cost changes: either increasing from the previous cost to ∞ due to new obstacles blocking the edge or decreasing from ∞ to some finite cost due to a new cleared cell. It is straightforward to increase cost by checking if the edge has been blocked by newly occupied cells but for decreasing the cost, we need to be careful and examine all the cells along each edge in order to make sure it is collision-free.

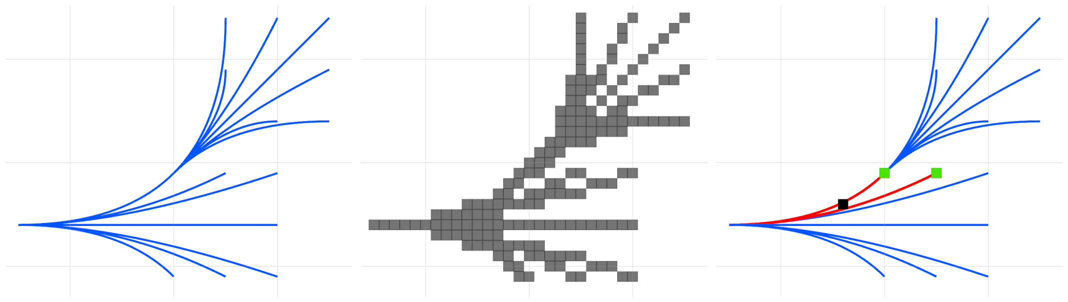


Figure 6.5: Associating cells in the occupancy grid map to the edges of the graph. Left: edges in the graph. Note that they are directional from left to right; Middle: sampled cells that the edges covers (grey cells); Right: if a cell is occupied (black cell in the middle), we change the costs of the related edges (red edges) and update the corresponding end states (green dots).

Polygonal Map

As illustrated in Section 6.1, we can use polyhedron to represent the obstacle that is either static or dynamic. Thus, in a polygonal map where obstacles are represented as polyhedra, we can check collision against given primitive using (6.2). This collision checking is verified in closed-form. However, keep track of the edge cost changes in the polygonal map is not easy. In order to guarantee the correctness, we use a naive method that iterates through all the edges to update their costs whenever the map is updated.

6.2.5 Incremental Trajectory Re-planning

In the framework of Receding Horizon Control, the robot only executes a small portion of each planned trajectory between two planning iterations. In our re-planning framework, this small portion corresponds to the first edge in the planned trajectory. Generally, the sensor rate is much higher than re-planning rate. Hence in-between two planning iterations, there might be multiple map changes. We modify the *Main* procedure in Algorithm 5 to formulate the replan with incremental trajectory planning as shown in Algorithm 7.

At line 6, we set the new start state for the next planning iteration as the successor of the current start state along the planned trajectory Φ where τ is the traversing time of the edge. First, we prune the graph according to this new start state and while the robot is approaching the new start state, we keep on updating the edge costs based on the pruned

Algorithm 7 *Incremental Trajectory Planning.*

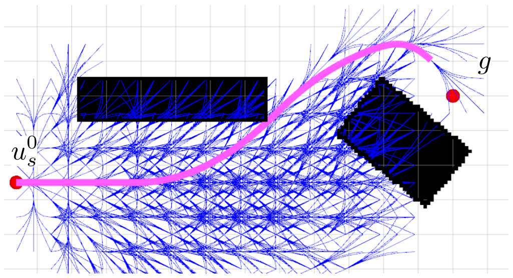
```
1:  procedure Main()
2:      InitializeGraph();
3:      $u_g \leftarrow g$, InitializeState($u_g$);
4:      while $u_s \not\approx g$ do
5:          $u_g, \Phi \leftarrow$ ComputeTrajectory($u_g$);
6:          $u_s \leftarrow \Phi(\tau)$;
7:          process Graph Pruning with the new $u_s$;
8:          while robot approaching $u_s$ do
9:              for all $e(u, v)$ that has changed cost do
10:                 UpdateState($v$);
11:             end for
12:         end while
13:     end while
14: end procedure
```

graph.

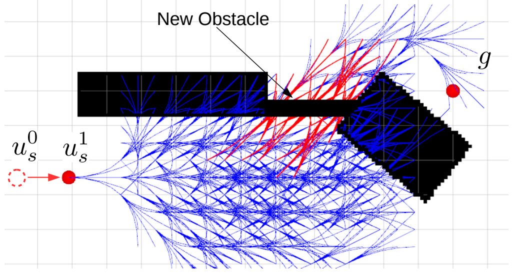
In Figure 6.6, we show an example of running the whole incremental trajectory planning framework. The robot starts from the left side of the map (left red ball) and plans to the goal on the right side (right red ball). We show the planning results from three planning iterations, each iteration (except iteration 0) uses the graph from previous iteration. The left column shows the results after running *ComputeTrajectory* in Algorithm 7, the right column shows the graph that has been pruned and updated according to the new start and map changes. For example, at iteration 0, the robot follows the planned trajectory till the new start state u_s^1 (in (b)). As the robot is moving, we add a new obstacle that blocks the original trajectory, thus the pruned graph that starts from the new start state, u_s^1, has local inconsistency. The affected edges are indicated by the red splines. The inconsistent graph in (b) is used for planning in the next iteration as shown in (c). We can repeat this process until the robot reaches the goal.

6.2.6 Evaluations

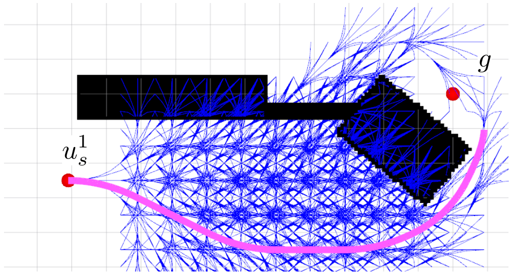
In this section, we discuss the efficiency of incremental trajectory planning and present results of running ITP for navigating a quadrotor in the simulated environment.



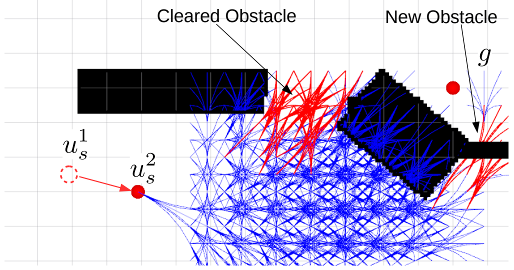
(a) Planning iteration 0



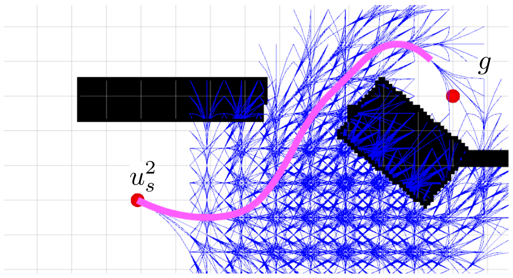
(b) Prune and update graph



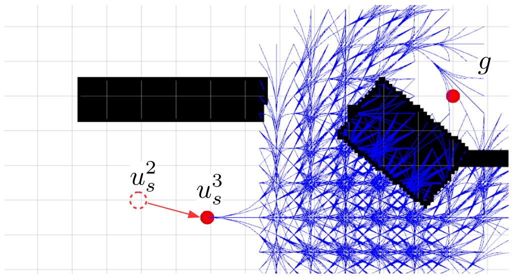
(c) Planning iteration 1



(d) Prune and update graph



(e) Planning iteration 2

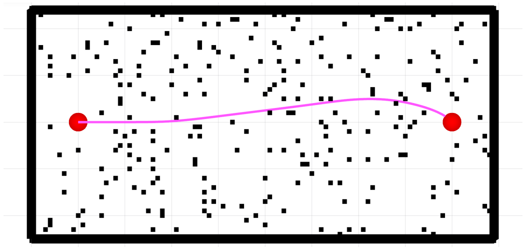


(f) Prune and update graph

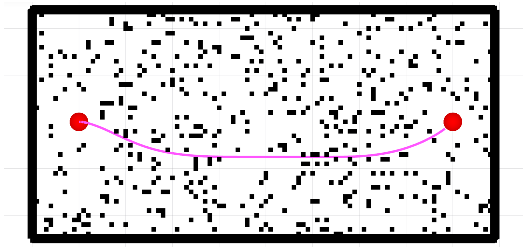
Figure 6.6: Example of 3 iterations in incremental trajectory planning. Planned trajectory is indicated by magenta spline at every planning iteration. The graph consists of all the edges in blue. Between two planning iterations, the map is updated according to the new sensor measurements. For example, after iteration 0, there is a new obstacle that blocks the planned trajectory; at iteration 2, this obstacle goes away and a new obstacle appears on the right side of the map. We use red splines to highlight the affected edges due to map updates.

Efficiency of Incremental Graph Search

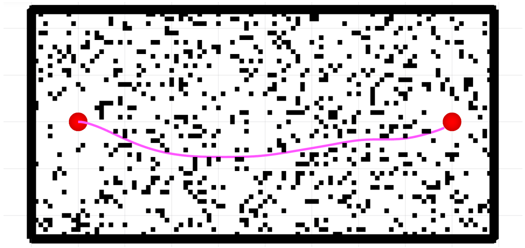
We evaluate the efficiency of the proposed incremental graph search based on LPA* by comparing its planning time and number of expansions to running A* from scratch in an incrementally changing map. The expansion in A* and LPA* refers to the step where we open a state and update the attributes of itself and its successors, this is equivalent to the part from line 19 to 28 in Algorithm 5. The number of expansions relates directly to the planning time, but since the implementation of A* and LPA* are a little different, it may



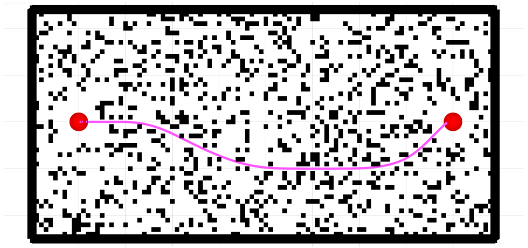
(a) $\rho = 0.05$



(b) $\rho = 0.10$



(c) $\rho = 0.15$



(d) $\rho = 0.20$

Figure 6.7: Snapshots in a test sequence of incremental graph search. From (a) to (d), the ostacles are randomly added into the previous map. Left and right red dots indicate the start and goal state, magenta spline is the planned trajectory.

take more time to process the same number of expansions for LPA* compared to A*. Thus, in the evaluation plots, we show both the planning time and the number of expansions. We denote the planning time of A* and LPA* as t(A*) and t(LPA*), the number of expansions as n(A*) and n(LPA*), and plot the ratio instead of their actual values for the purpose of clarity.

We run a sequence of planning from a start state to the goal state using both A* and LPA* with the same parameters. The map is initialized as empty, and obstacle density ρ is increased from 0 to 0.2 by a constant rate $d\rho$ by randomly adding new occupied cells in each iteration. The obstacle density ρ is defined as:

$$\rho = \frac{\text{\# of occupied cells}}{\text{\# of cells in the map}}$$

Figure 6.7 shows snapshots of maps and planned trajectories from start to end in a sequential test. We run 50 independent sequences and collect the data for plotting. In Figure 6.8, the rate of adding obstacles $d\rho$ from row 1 to 3 is 0.01, 0.02 and 0.03. In each plot figure, the median of the ratio of planning time versus the obstacle density is shown as the green line segment, while the x-axis shows the increasing obstacle density (in percentage %). We can see that the LPA* significantly reduces the amount of time and number of expansions when $d\rho = 0.01$. When $d\rho > 0.01$, the improvements are small. The initial plans of A* and LPA* take almost the same amount of time since they both plan from scratch in the first iteration.

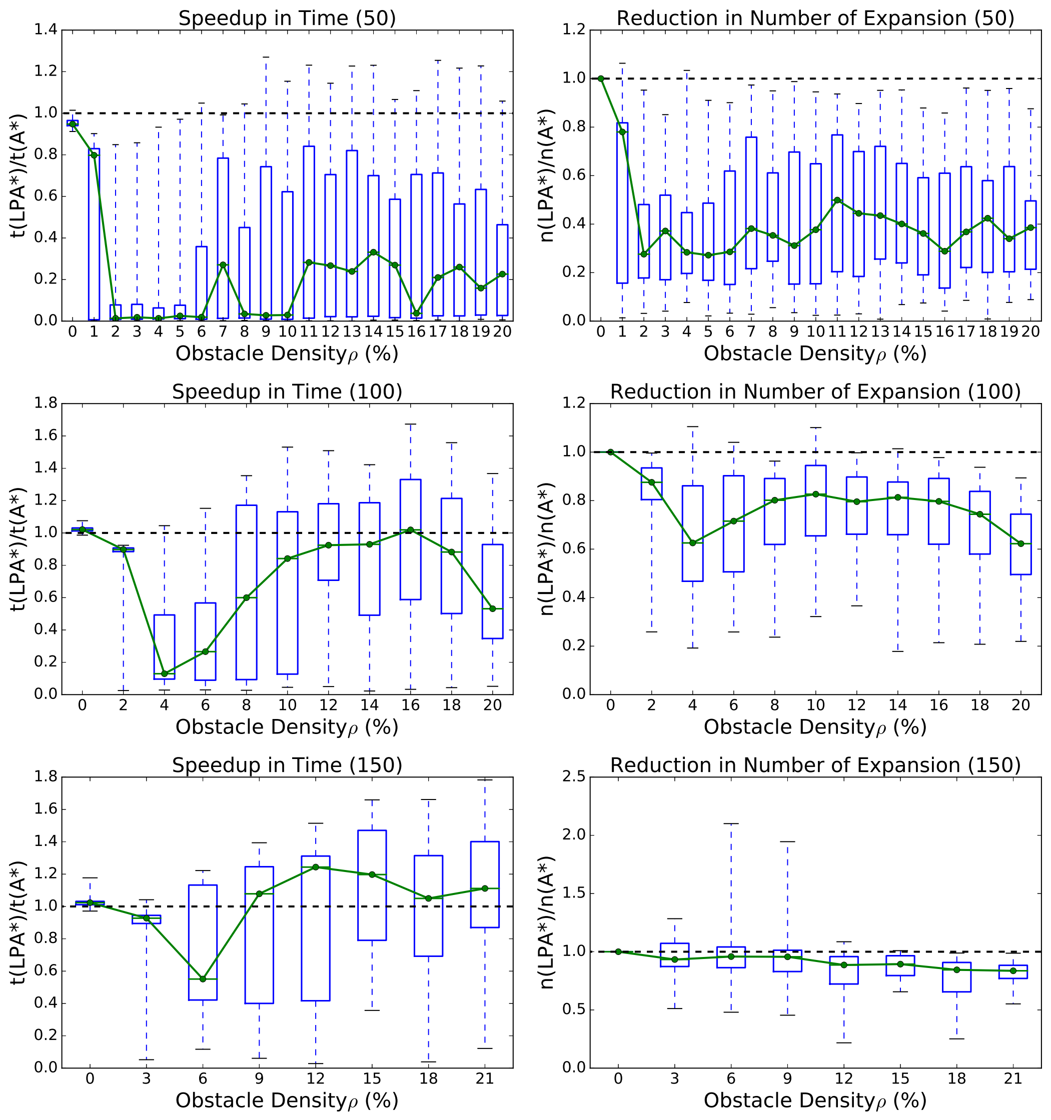


Figure 6.8: Comparing LPA* with A* in 50 independent sequential test with increasing obstacle density. Left columns shows the plot of ratio of planning time versus the obstacle density; right column shows the plot of ratio of number of expanded states versus the obstacle density. From row 1 to 3, we set the incremental obstacle density as 0.01, 0.02 and 0.03.

Similarly, Figure 6.9 shows the evaluation when removing obstacles. The map is initial-

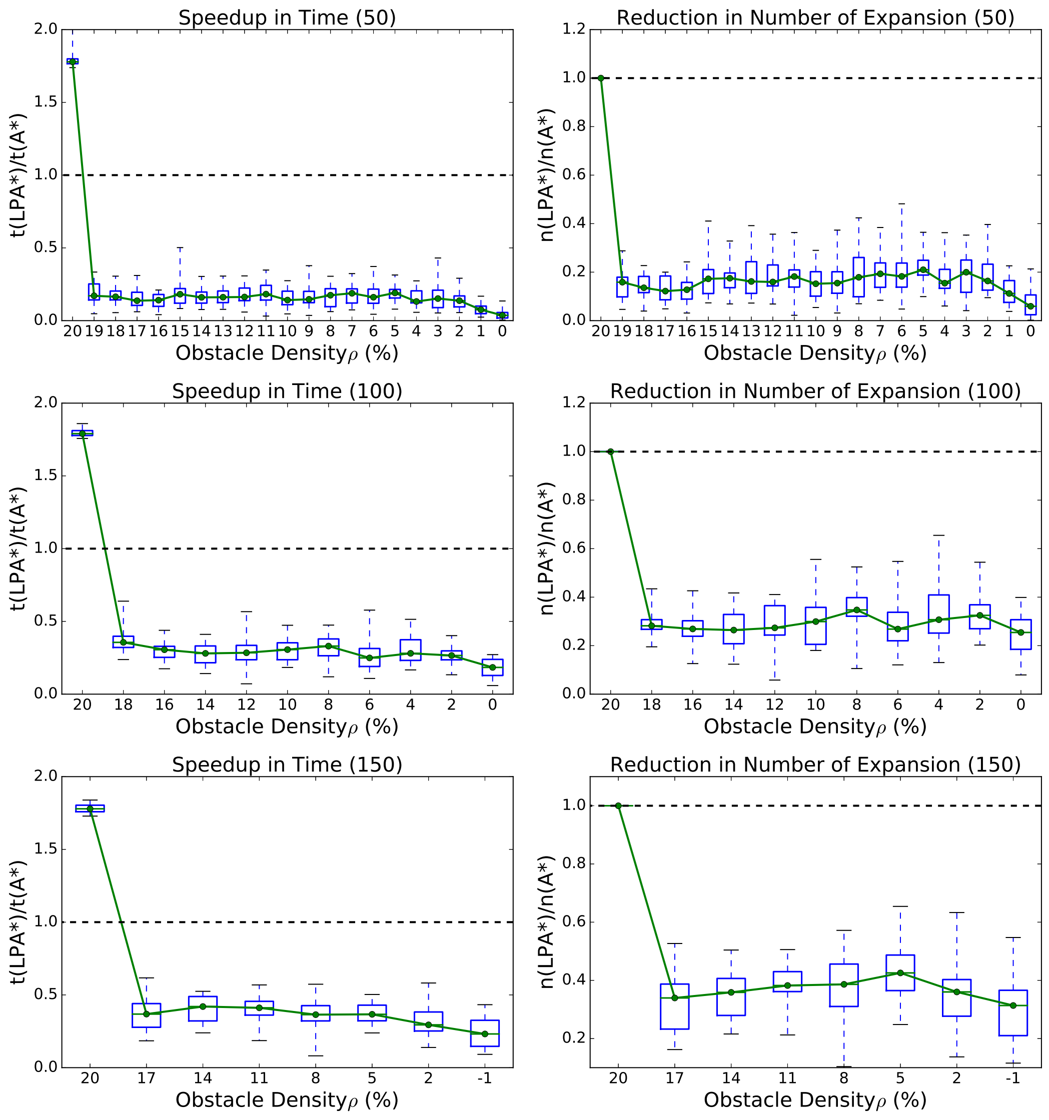


Figure 6.9: Comparing LPA* with A* in 50 independent sequential test with decreasing obstacle density. Left columns shows the plot of ratio of planning time versus the obstacle density; right column shows the plot of ratio of number of expanded states versus the obstacle density. From row 1 to 3, we set the incremental obstacle density as 0.01, 0.02 and 0.03.

ized with randomly sampled obstacles of density $\rho = 0.2$ and we randomly remove occupied cells to get a new map for planning in the next iteration. We also run 50 independent sequences and collect the data for plotting. We change the rate of removing obstacles $d\rho$ from row 1 to 3 as 0.01, 0.02 and 0.03. In this case, LPA* significantly reduces the amount of time and number of expansions for any $d\rho$. The initial plans of A* and LPA* have the same amount of number of expansions but the planning times are different, this is because

LPA* takes more time to process each expansion and the overhead for maintaining the data structures is significant when the number of states in the graph is large.

Even though the random map changes in this test is not realistic, it helps to understand the limitation and performance of the proposed algorithm in extremely challenging planning tasks. In conclusion, LPA* is more efficient than planning from scratch specially for small incremental changes in the map.

3D Planning

The proposed planner can be easily extended to 3D by adding control inputs along the third axis (z-axis) when applying motion primitives. We show an example in Figure 6.10 for re-planning in a 3D voxel map. Here we illustrate the planning results from two iterations, between which the robot detects a new obstacle. We run both LPA* and A* for the same planning queries. The two images on the right side plot the expanded states from running LPA* and normal A* planning from scratch in the planning iteration 1. LPA* expands much fewer states and is more than 10 times faster than A*.

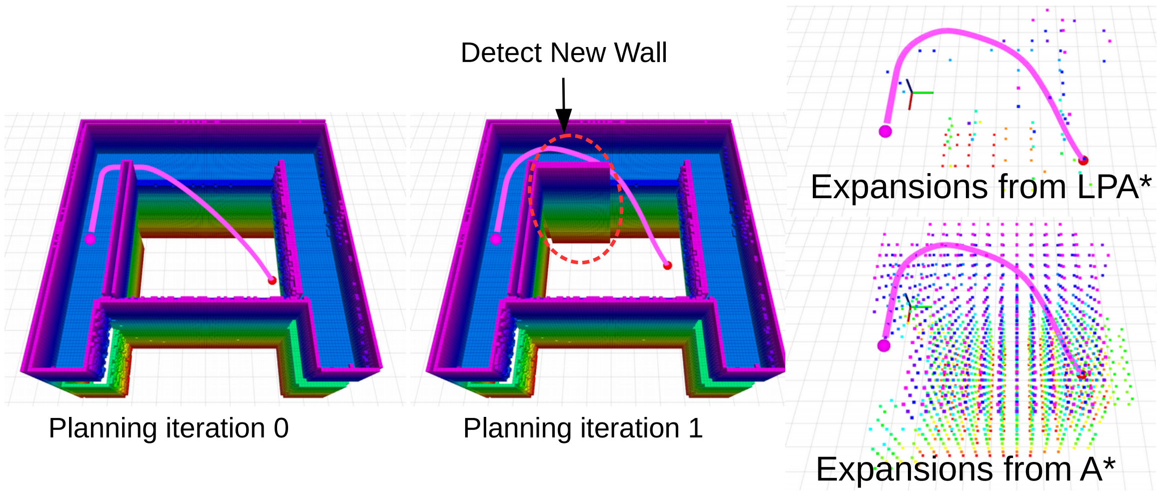


Figure 6.10: Incremental trajectory planning in 3D. We run both LPA* and normal A* for these two planning iterations. For the iteration 1, LPA* takes 43ms and expands 567 states to find the new optimal trajectory while A* takes 527ms and expands 5826 states to find the same trajectory. The expanded states from these two algorithms are plotted in images on the right side.

Navigation Test

In the physical world, the map is incrementally updated based on the sensor inputs and obstacles are not randomly generated or removed like in Figure 6.7. We set up a simulated environment in Gazebo to evaluate the performance of the proposed algorithm in the practical navigation task. The simulated quadrotor is sensing the environment using a laser rangefinder and keeps re-planning at a constant frequency. We run two planners independently in the same test: A* that plans from scratch at every planning iteration and incremental trajectory planning that reuses the previous graph based on LPA*. We use the same parameters for setting up both planners and the planning horizon is set to 7.5 s.

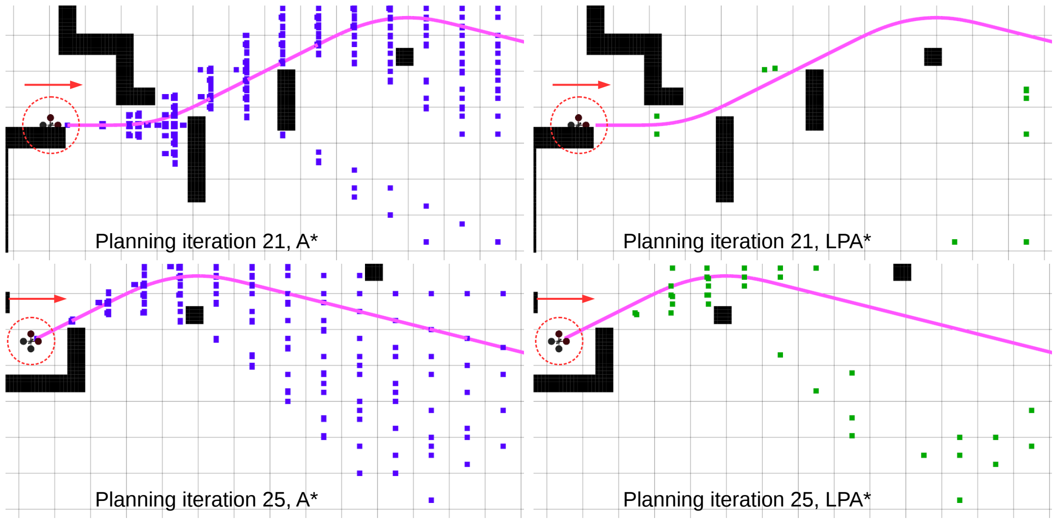

Planning iteration 21, A*
Planning iteration 21, LPA*
Planning iteration 25, A*
Planning iteration 25, LPA*

Figure 6.11: Simulation example. Left and right columns show the results from planning from scratch and from incremental planning respectively. The blue and green dots indicate the expanded states, which shows that the incremental planner expands much fewer states than A*. Note that the planned trajectories from both are the same.

Figure 6.11 shows several snapshots while the robot is flying towards the goal at planning iteration 21 and 25. Obviously, two planners find the same trajectory but A* explores more states. We show the planning time and number of expansions of two planners in Figure 6.12, from which we can clearly see that LPA* is more efficient than A* since it takes less planning time and requires fewer number of expansions over the whole mission.

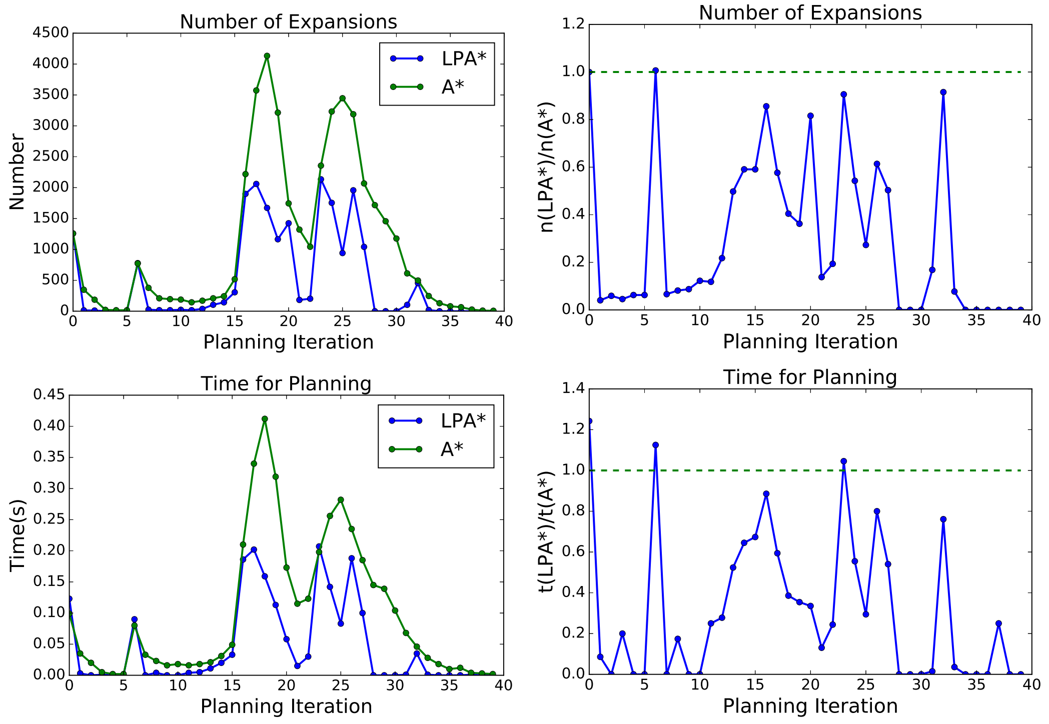


Figure 6.12: Comparison of efficiency between LPA* and A* in the navigation task. Left column shows the absolute values of corresponding number of expansions and planning time, right column shows the corresponding ratios between LPA* and A*.

6.3 Multi-robot Planning

We consider a scenario where a team of homogeneous robots operates in a environment at the same time. We assume that some mission control algorithm such as exploration assigns a target to each of the robots. Thus, it is a decoupled problem in which individual robot plans its own trajectory. Different from existing works such as [32, 58, 93, 95, 99], we mainly focus on finding the optimal trajectory for robot without colliding with other robots. We show that the proposed framework can be used to plan trajectories for each robot by treating other robots as moving obstacles. Thus, we are able to perform either sequential or decentralized planning for multiple robots in the same workspace.

6.3.1 Collision Checking between Robots

In Section 6.1, we modeled the obstacle as a linearly moving polyhedron in $\mathbb{R}^m$, which can be generalized for non-linear moving obstacles that follow piece-wise polynomial trajectories.

Denote $c_i(t)$ as the i-th robot configuration which is a *non-linearly moving polyhedron* (NMP) in $\mathbb{R}^m$, it is represented as the robot geometry $c_{i,0} = \{p \mid \mathbf{A}_i^\top p \leq \mathbf{b}_i\}$ that centered at robot's center of mass following a trajectory ${}^i\Phi(t)$. Thus, it can be represented as:

$$c_i(t) = \{p \mid \mathbf{A}_i^\top (p - {}^i\Phi(t)) \leq \mathbf{b}_i\}. \tag{6.5}$$

For robot i and j, they are not colliding with each other if and only if

$${}^i\Phi(t) \cap [c_{i,0} \oplus c_j(t)] = \emptyset, \tag{6.6}$$

where "$\oplus$" denotes the Minkowski addition. Constraint (6.6) can be verified by solving for roots of a polynomial equation similar to (6.2). For a team of robots, we can verify whether the i-th robot's trajectory is collision-free by checking (6.6) against all the other robots.

6.3.2 Sequential Planning

For a team of Z robots, we can sequentially plan trajectory for robots from 0 to $Z - 1$ by assigning priorities to the robots. When planning for i-th robot, we only consider collision checking with robots that have higher priority than i. Equivalently, we need to verify the following equation for the i-th robot:

$${}^i\Phi(t) \cap \bigcup_{j=0}^{i-1} [c_{i,0} \oplus c_j(t)] = \emptyset. \tag{6.7}$$

Sequential planning is able to guarantee inter-robot collision-free and find the optimal trajectory for each robot with respect to the priority. The planning results for two navigation tasks are shown in Figure 6.13. Its computational complexity is polynomial, thus we are able to quickly plan the trajectories for the whole team.

6.3.3 Decentralized Planning

In the decentralized case, each robot re-plans at their own clock rate and there is no priority. It is practical to assume that the robot is able to share information about its current tra-

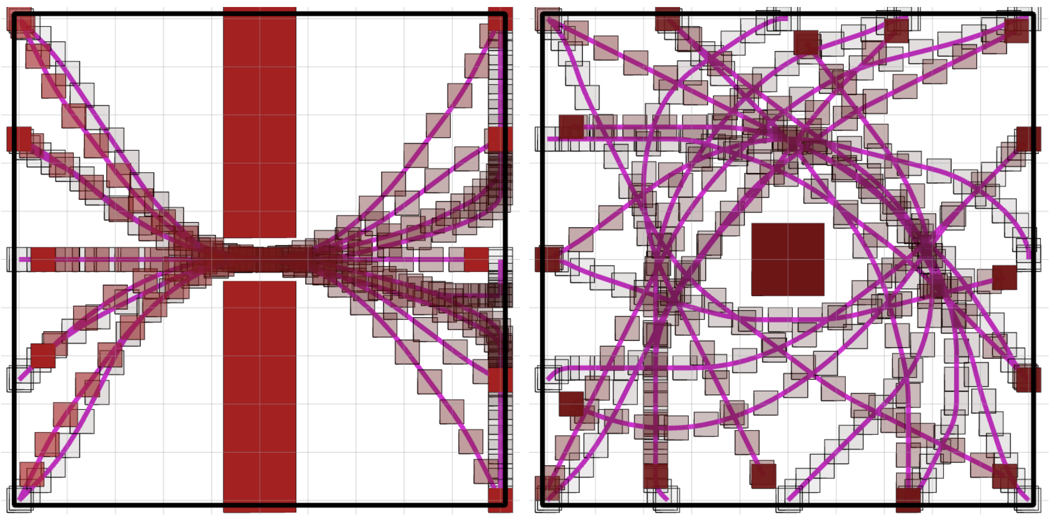

(a) Tunnel configuration. (b) Star configuration.

Figure 6.13: Example planning tasks for a multi-robot system. Blue rectangles represent the obstacles and robots' geometries. Magenta trajectories are the planning results from the sequential planning.

jectory with other robots. For accurate collision checking, we also assume there is a global time frame and a local time frame for each robot representing its trajectory start time. Use τ and t to represent the time in global time frame and local time frame respectively. The conversion between this two frames is simply $t = \tau - \tau^s$ where τ^s is the start time in global time frame of the trajectory. Thus, we formulate the collision checking for Z robots in the decentralized manner as:

$$ {}^{i}\Phi(\tau - \tau_i^s) \cap \bigcup_{j=0}^{Z-1} [c_{i,0} \oplus c_j(\tau - \tau_j^s)] = \emptyset, \tag{6.8} $$

or in local time frame as:

$$ {}^{i}\Phi(t) \cap \bigcup_{j=0}^{Z-1} [c_{i,0} \oplus c_j(t + \tau_i^s - \tau_j^s)] = \emptyset. \tag{6.9} $$

Here $\tau_i^s - \tau_j^s$ is a constant for the given trajectory pair ${}^{i}\Phi$ and ${}^{j}\Phi$. Since we plan for robot i with the presence of robot j, $\tau_i^s - \tau_j^s \geq 0$ should always be true.

Denote ${}^{j}T$ as the duration of trajectory ${}^{j}\Phi$. Ideally, we use the whole trajectory ${}^{j}\Phi(t)$

from $t = 0$ to ${}^jT$ for collision checking when plan for robot i. However, we know that robot j is also constantly re-planning, such that the future trajectory of j can be meaningless to be considered in collision checking. We are able to improve the efficiency of collision checking in (6.9) by setting a cutoff time T_c. Namely, we ignore the part of trajectory ${}^j\Phi(t)$ of other robot j for the domain $t > {}^jT_c$. Consequently, as T_c gets smaller, the computational time for inter-robot collision checking is also smaller. The smallest T_c for a complete solution is determined by the system's dynamic constraints. For example, for a second order system that is constrained by maximum velocity $\bar{v}_{max}$ and acceleration $\bar{a}_{max}$, the smallest value of T_c is the minimum time it takes to stop the robot from the maximum velocity. Thus, we set the value of T_c for trajectory Φ_j as

$$ {}^jT_c = \min\{\bar{v}_{max}/\bar{a}_{max}, {}^jT\}. \tag{6.10}$$

To make the algorithm complete, we need to ignore robot j for $t > {}^jT_c$ when planning for robot i instead of treating it as a static obstacle. Figure 6.14 shows the results of two planning tasks using the decentralized planning with T_c.

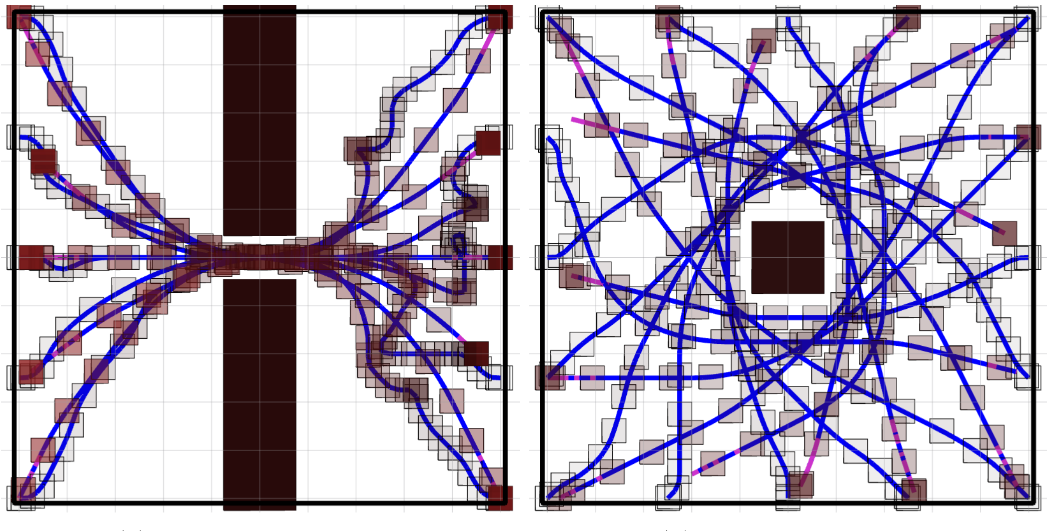

(a) Tunnel configuration. (b) Star configuration.

Figure 6.14: Example planning tasks for a multi-robot system. Blue rectangles represent the obstacles and robots' geometries. Blue trajectories are the traversed trajectories of the team guided by the proposed decentralized planning.

Chapter 7

Concluding Remarks

7.1 Contributions

In this thesis, we consider motion planning problems for micro aerial vehicles (MAVs) in real world applications. We systematically analyze the planning and control problems of MAVs and propose our solutions: (1) optimization-based approach based on *safe flight corridors* (SFC) that is adequate for aggressive flight in complex 3D environments (Chapter 3) and (2) search-based motion planning (SMP) with motion primitives that is resolution complete and optimal considering non-static initial states (Chapter 4). As a conclusion, we compare our proposed methodologies with several *state-of-the-art* techniques as shown in Table 7.1:

Method	***Feasibility***	***Safety***	***Optimality***	***Completeness***	***Run time***
Path Planning	Not feasible	Collision free	Globally optimal	Resolution complete	Very fast
Unconstrained QP [61, 82]	May violate constraints	Not guaranteed	Sub-optimal	Not complete	Very fast
MIP [15, 62]	Dynamically feasible	Collision free	Sub-optimal	Complete	Very slow
Short Trajectories [56, 71]	Dynamically feasible	Collision free	Locally optimal	Not complete	Very fast
SFC	Dynamically feasible	Collision free	Sub-optimal	Complete	Fast
SMP	Dynamically feasible	Collision free	Resolution optimal	Globally complete	Fast

Table 7.1: Evaluation of *state-of-the-art* and proposed (SFC and SMP) techniques using the five properties. Red blocks indicate the drawbacks of the corresponding algorithm.

Table 7.1 concludes that our approaches outperform the widely-used planning methods for navigating MAVs in unknown environments. In Chapter 5, we show that the SMP can be extended to solve more general planning problems with different constraints such as motion uncertainty, limited sensor's *field-of-view* (FOV) and attitude constraints. To plan in dynamic environments with both moving obstacles and multiple robots, we show that SMP is also applicable to fast find complete solutions in Chapter 6.

7.2 Future Work

The flexibility of the proposed search-based motion planning (SMP) method in this thesis indicates several interesting and promising research directions. Our framework provides a novel way to address these existing problems, which could also be a powerful tool for navigating general mobile robots.

Target Tracking Target tracking is desired in many practical applications. Traditional methods to plan trajectories that follow moving targets are based on path planning and constrained optimization. Instead, we can apply SMP to generate such trajectories by modifying the cost function and constraints accordingly.

Anytime Motion Planning Anytime planning is quite useful in real world applications where the run time is limited and the optimality of generated trajectories is not critical. It has been shown in Section 6.2 that the back-end of SMP is interchangeable between different graph search algorithms. We successfully applied LPA* that solves the re-plan problem, which indicates the possibility of the usage of other graph search algorithms including ARA*, RRT* and etc. To obtain anytime motion planning, the only work required is to implement these different algorithms.

Information-based Motion Planning To avoid potential dangers caused by motion uncertainty, we use the artificial potential field to perturb the nominal trajectory that is close to obstacles in Section 5.1. When the uncertainty of state estimation or obstacle detection is taken into account, motion planning is more challenging. Existing works including POMDP-based and probabilistic planning are extremely slow to find an optimal solution. The SMP can be potentially developed to properly model these uncertainties and quickly solve optimal trajectories.

Centralized Multi-robot Planning The planning problem of multi-robot system has recently attracted much attention. We proposed the solution for multi-robot planning in both sequential and decentralized manners in Section 6.3. For centralized motion planning, the global cost such as total traveling distance and time and completeness of different configurations are important to address. The SMP has the potential to efficiently solve such a centralized optimal planning problem.

Appendix A

Open-sourced Resources

A.1 Publications

The work presented in this thesis has been progressively developed during past few years. Our previous publications on the related subject are listed as following:

1. Chapter 3 – Planning dynamically feasible trajectories for quadrotors using safe flight corridors in 3D complex environments [53];
2. Chapter 4 – Search-based motion planning for quadrotors using linear quadratic minimum time control [49];
3. Chapter 5 – Search-Based Motion Planning for Aggressive Flight in SE (3) [50].
4. Chapter 5 – Towards Search-Based Motion Planning for Micro Aerial Vehicles [51].

A.2 Code

We open-sourced almost all the planning code in C++ on Github. In this section, we list some of them and briefly introduce their usage.

1. MRSL Quadrotor Simulator: `https://github.com/KumarRobotics/mrsl_quadrotor`
2. JPS3D: `https://github.com/KumarRobotics/jps3d`

3. DecompUtil: https://github.com/sikang/DecompUtil

4. DecompROS: https://github.com/sikang/DecompROS

5. Motion Primitive Library: https://github.com/sikang/motion_primitive_library

6. MPL ROS: https://github.com/sikang/mpl_ros

A.2.1 MRSL Quadrotor Simulator

This package is the simulation for testing planning with quadrotors in Gazebo environment [72]. MRSL stands for Multi-Robot System Lab at University of Pennsylvania. This package depends on `quadrotor_control` repository which can be found on KumarRobotics: https://github.com/KumarRobotics/quadrotor_control, which is currently unpublished. But it is not hard to build up the geometric controller from scratch.

To set up an simulation environment, we need two things: (1) a raw 3D model of the world environment in the format of `.dae` (`Collada`) and (2) a configuration file in the format of `.world`. The `.dae` file can be exported from models in SketchUp [26] or Blender [6]. We recommend to use SketchUp to draw the 3D models with good-looking textures. The `.world` file is a script in `.sdf` format [73]. There are several examples of `.world` file in the folder `mrsl_quadrotor_descriptions/world`. Figure A.1 shows three built-in environment models.

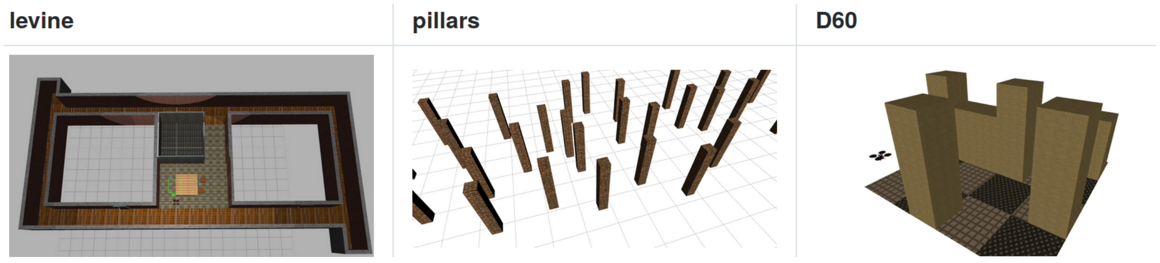


Figure A.1: Three built-in environments in MRSL Quadrotor Simulator.

To set up a simulated quadrotor, we can use the existing quadrotor model and sensors in `mrsl_quadrotor_descriptions/urdf`. We can mount different sensors such as RGB-D and laser rangefinder on different platforms. Figure A.2 shows three models. The robots subscribe and response to `quadrotor_msgs` which is part of `quadrotor_control` packages.

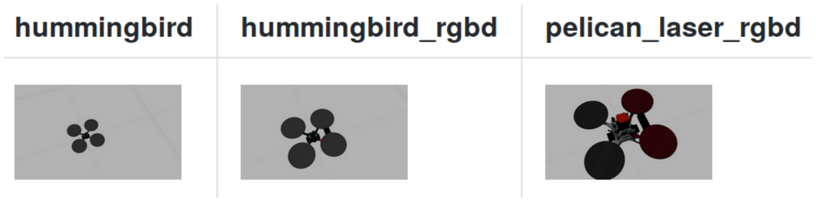


Figure A.2: Three built-in quadrotor models in MRSL Quadrotor Simulator.

A.2.2 JPS3D

In this package, we implement Jump Point Search (JPS) in both 2D and 3D maps. The test nodes are presented in `test` folder. A 2D example of the comparison between A* and JPS is shown in Figure A.3.

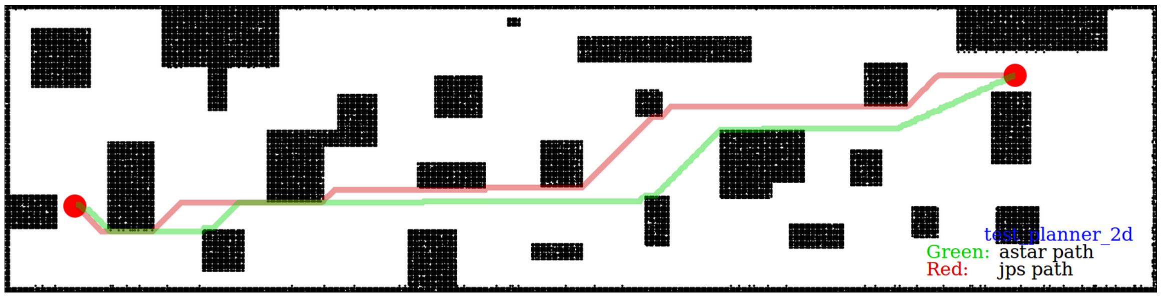


Figure A.3: A 2D planning example of using A* and JPS.

A.2.3 DecompUtil and DecompROS

The `DecompUtil` is the back-end of convex decomposition as proposed in Section 3.3. The `DecompROS` is the associated ROS package. In `DecompUtil`, we provide four different tools to do regional inflation in a point cloud (Figure A.4). These tools work for both 2D and 3D environments.

A.2.4 Motion Primitive Library and MPL ROS

The `Motion Primitive Library` (MPL) is the implementation of search-based motion planning. The `mpl_ros` is the ROS wrapper of MPL.

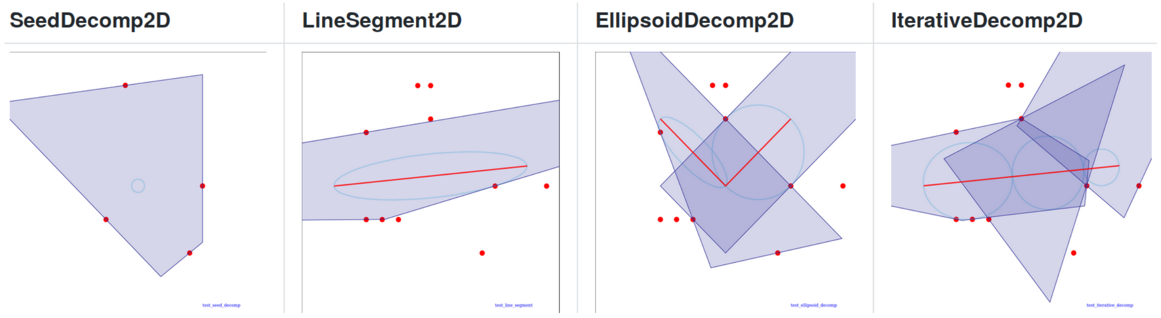


Figure A.4: A 2D example of convex decomposition.

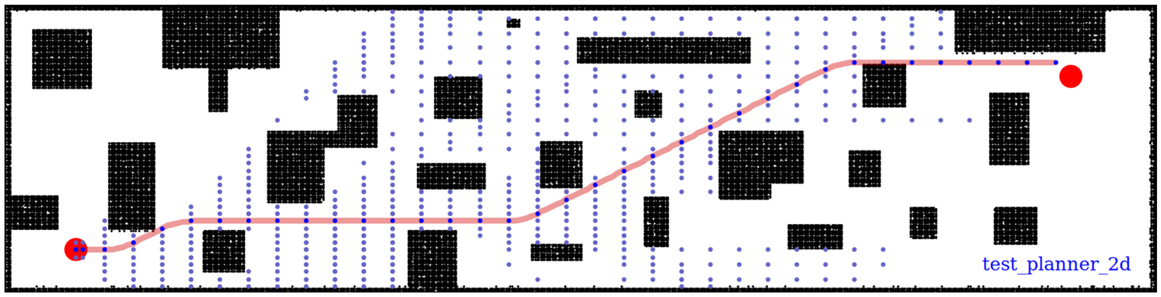


Figure A.5: Trajectory planned using MPL in a given map.

Bibliography

[1] "The DARPA Fast Lightweight Autonomy project," https://www.darpa.mil/program/fast-lightweight-autonomy/.

[2] H. Adeli, M. Tabrizi, A. Mazloomian, E. Hajipour, and M. Jahed, "Path planning for mobile robots using iterative artificial potential field method," *International Journal of Computer Science Issues (IJCSI)*, vol. 8, no. 4, p. 28, 2011.

[3] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *IEEE Int. Conf. on Robotics and Automation (ICRA)*. IEEE, 2013, pp. 2421–2428.

[4] O. Arslan and D. E. Koditschek, "Smooth extensions of feedback motion planners via reference governors," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 4414–4421.

[5] D. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[6] Blender, "Gazebo." [Online]. Available: https://www.blender.org

[7] Y. Bouktir, M. Haddad, and T. Chettibi, "Trajectory planning for a quadrotor helicopter," in *16th Mediterranean Conference on Control and Automation*, 2008.

[8] B. Charrow, S. Liu, V. Kumar, and N. Michael, "Information-theoretic mapping using cauchy-schwarz quadratic mutual information," in *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2015.

[9] J. Chen, T. Liu, and S. Shen, "Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 1476–1483.

[10] S. Chung, A. A. Paranjape, P. Dames, S. Shen, and V. Kumar, "A survey on aerial swarm robotics," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 837–855, Aug 2018.

[11] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, Tech. Rep., 1992.

[12] K. F. Culligan, "Online trajectory planning for uavs using mixed integer linear programming," Ph.D. dissertation, Massachusetts Institute of Technology, 2006.

[13] R. Deits and R. Tedrake, “Footstep planning on uneven terrain with mixed-integer convex optimization,” in *IEEE-RAS International Conference on Humanoid Robots*, 2014.

[14] ——, “Computing large convex regions of obstacle-free space through semidefinite programming,” in *Algorithmic Foundations of Robotics XI*. Springer, 2015, pp. 109–124.

[15] ——, “Efficient mixed-integer planning for uavs in cluttered environments,” in *Proceedings of the 2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[16] B. Donald, P. Xavier, J. Canny, and J. Reif, “Kinodynamic motion planning,” *Journal of the ACM (JACM)*, vol. 40, no. 5, pp. 1048–1066, 1993.

[17] D. Falanga, E. Mueggler, M. Faessler, and D. Scaramuzza, “Aggressive quadrotor flight through narrow gaps with onboard sensing and computing using active vision,” in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 5774–5781.

[18] D. Feng and B. Krogh, “Acceleration-constrained time-optimal control in n dimensions,” *IEEE Transactions on Automatic Control*, vol. 31, no. 10, pp. 955–958, 1986.

[19] D. Ferguson and A. Stentz, “Using interpolation to improve path planning: The field d* algorithm,” *Journal of Field Robotics*, vol. 23, no. 2, pp. 79–101, 2006.

[20] P. Fiorini and Z. Shiller, “Motion planning in dynamic environments using velocity obstacles,” *The International Journal of Robotics Research*, vol. 17, no. 7, pp. 760–772, 1998.

[21] K. Fujimura and H. Samet, “A hierarchical strategy for path planning among moving obstacles (mobile robot),” *IEEE transactions on robotics and Automation*, vol. 5, no. 1, pp. 61–69, 1989.

[22] F. Gao, Y. Lin, and S. Shen, “Gradient-based online safe trajectory generation for quadrotor flight in complex environments,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept 2017, pp. 3681–3688.

[23] R. Geraerts and M. H. Overmars, “A comparative study of probabilistic roadmap planners,” in *Algorithmic Foundations of Robotics V*. Springer, 2004, pp. 43–57.

[24] K. Gochev, B. Cohen, J. Butzke, A. Safonova, and M. Likhachev, “Path planning with adaptive dimensionality,” in *Fourth annual symposium on combinatorial search*, 2011.

[25] M. A. Goodrich, “Potential fields tutorial.”

[26] Google, “Sketchup.” [Online]. Available: https://www.sketchup.com

[27] D. D. Harabor, A. Grastien *et al.*, “Online graph pruning for pathfinding on grid maps,” in *AAAI*, 2011.

[28] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[29] M. Hehn and R. D’Andrea, “Quadrocopter trajectory generation and control,” *IFAC Proceedings Volumes*, vol. 44, no. 1, 2011.

[30] T. Hirata and M. Kumon, “Optimal path planning method with attitude constraints for quadrotor helicopters,” in *Int. Conf. on Advanced Mechatronic Systems*, 2014, pp. 377–381.

[31] G. Hoffmann, S. Waslander, and C. Tomlin, “Quadrotor helicopter trajectory tracking control,” in *AIAA guidance, navigation and control conference and exhibit*, p. 7410.

[32] W. Hönig, J. A. Preiss, T. S. Kumar, G. S. Sukhatme, and N. Ayanian, “Trajectory planning for quadrotor swarms,” *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 856–869, 2018.

[33] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, “Octomap: An efficient probabilistic 3d mapping framework based on octrees,” *Autonomous Robots*, 2013.

[34] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, “Randomized kinodynamic motion planning with moving obstacles,” *The International Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, 2002.

[35] J. Jamieson and J. Biggs, “Near minimum-time trajectories for quadrotor uavs in complex environments,” in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2016, pp. 1550–1555.

[36] S. Karaman and E. Frazzoli, “Sampling-based algorithms for optimal motion planning,” *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[37] ——, “Sampling-based algorithms for optimal motion planning,” *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[38] L. Kavraki, P. Svestka, and M. H. Overmars, *Probabilistic roadmaps for path planning in high-dimensional configuration spaces.* Unknown Publisher, 1994, vol. 1994.

[39] S. Koenig and M. Likhachev, “D* lite,” in *Eighteenth national conference on Artificial intelligence.* American Association for Artificial Intelligence, 2002, pp. 476–483.

[40] S. Koenig, M. Likhachev, and D. Furcy, “Lifelong planning aâĹŮ,” *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

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

[42] S. M. LaValle, *Planning Algorithms.* New York, NY, USA: Cambridge University Press, 2006.

[43] S. M. Lavalle, J. J. Kuffner, and Jr., "Rapidly-exploring random trees: Progress and prospects," in *Algorithmic and Computational Robotics: New Directions*, 2000, pp. 293–308.

[44] T. Lee, M. Leoky, and N. H. McClamroch, "Geometric tracking control of a quadrotor uav on se (3)," in *49th IEEE conference on decision and control (CDC)*. IEEE, 2010, pp. 5420–5425.

[45] F. Lewis and V. Syrmos, *Optimal control*. John Wiley & Sons, 1995.

[46] G. Li, Y. Tamura, A. Yamashita, and H. Asama, "Effective improved artificial potential field-based regression search method for autonomous mobile robot path planning," *International Journal of Mechatronics and Automation*, vol. 3, no. 3, pp. 141–170, 2013.

[47] M. Likhachev and D. Ferguson, "Planning long dynamically feasible maneuvers for autonomous vehicles," *The International Journal of Robotics Research*, vol. 28, no. 8, pp. 933–945, 2009.

[48] M. Likhachev, G. Gordon, and S. Thrun, "ARA* : Anytime A* with provable bounds on sub-optimality," in *Advances in Neural Information Processing Systems*, 2004, pp. 767–774.

[49] S. Liu, N. Atanasov, K. Mohta, and V. Kumar, "Search-based motion planning for quadrotors using linear quadratic minimum time control," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*. IEEE, 2017, pp. 2872–2879.

[50] S. Liu, K. Mohta, N. Atanasov, and V. Kumar, "Search-based motion planning for aggressive flight in se (3)," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2439–2446, 2018.

[51] ——, "Towards search-based motion planning for micro aerial vehicles," *arXiv preprint arXiv:1810.03071*, 2018.

[52] S. Liu, K. Mohta, S. Shen, and V. Kumar, "Towards collaborative mapping and exploration using multiple micro aerial robots," in *Experimental Robotics*. Springer, 2016, pp. 865–878.

[53] 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*, vol. 2, no. 3, pp. 1688–1695, 2017.

[54] S. Liu, M. Watterson, S. Tang, and V. Kumar, "High speed navigation for quadrotors with limited onboard sensing," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1484–1491.

[55] 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*, vol. 2, no. 2, pp. 404–411, April 2017.

[56] B. T. Lopez and J. P. How, "Aggressive collision avoidance with limited field-of-view sensing," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept 2017, pp. 1358–1365.

[57] T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Communications of the ACM*, vol. 22, no. 10, pp. 560–570, 1979.

[58] R. Luna and K. E. Bekris, "Efficient and complete centralized multi-robot path planning," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on.* IEEE, 2011, pp. 3268–3275.

[59] B. MacAllister, J. Butzke, A. Kushleyev, H. Pandey, and M. Likhachev, "Path planning for non-circular micro aerial vehicles in constrained environments," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on.* IEEE, 2013, pp. 3933–3940.

[60] T. Madani and A. Benallegue, "Backstepping control for a quadrotor helicopter," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on.* IEEE, 2006, pp. 3255–3260.

[61] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *Proceedings of the 2011 IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[62] D. Mellinger, A. Kushleyev, and V. Kumar, "Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams," in *Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA)*, 2012.

[63] D. W. Mellinger, "Trajectory generation and control for quadrotors," Ph.D. dissertation, University of Pennsylvania, 2012.

[64] N. Michael, S. Shen, K. Mohta, V. Kumar, K. Nagatani, Y. Okada, S. Kiribayashi, K. Otake, K. Yoshida, K. Ohno *et al.*, "Collaborative mapping of an earthquake damaged building via ground and aerial robots," in *Field and Service Robotics.* Springer, 2014, pp. 33–47.

[65] K. Mohta, K. Sun, S. Liu, M. Watterson, B. Pfrommer, J. Svacha, Y. Mulgaonkar, C. J. Taylor, and V. Kumar, "Experiments in fast, autonomous, gps-denied quadrotor flight," 2018.

[66] K. Mohta, M. Turpin, A. Kushleyev, D. Mellinger, N. Michael, and V. Kumar, "Quadcloud: a rapid response force with quadrotor teams," in *Experimental Robotics.* Springer, 2016, pp. 577–590.

[67] K. Mohta, M. Watterson, Y. Mulgaonkar, S. Liu, C. Qu, A. Makineni, K. Saulnier, K. Sun, A. Zhu, J. Delmerico *et al.*, "Fast, autonomous flight in gps-denied and cluttered environments," *Journal of Field Robotics*, vol. 35, no. 1, pp. 101–120, 2018.

[68] M. Mueller, M. Hehn, and R. D'Andrea, "A computationally efficient motion primitive for quadrocopter trajectory generation," *IEEE Trans. on Robotics (T-RO)*, vol. 31, no. 6, pp. 1294–1310, 2015.

[69] V. Narayanan, M. Phillips, and M. Likhachev, "Anytime safe interval path planning for dynamic environments," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on.* IEEE, 2012, pp. 4708–4715.

[70] D. R. Nelson, D. B. Barber, T. W. McLain, and R. W. Beard, "Vector field path following for miniature air vehicles," *IEEE Transactions on Robotics*, vol. 23, no. 3, pp. 519–529, 2007.

[71] H. Oleynikova, M. Burri, Z. Taylor, J. Nieto, R. Siegwart, and E. Galceran, "Continuous-time trajectory optimization for online uav replanning," in *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on.* IEEE, 2016, pp. 5332–5339.

[72] OSRF, "Gazebo." [Online]. Available: http://gazebosim.org

[73] ——, "Sdf." [Online]. Available: http://sdformat.org

[74] M. Phillips and M. Likhachev, "Sipp: Safe interval path planning for dynamic environments," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 5628–5635.

[75] M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *Journal of Field Robotics*, vol. 26, no. 3, pp. 308–333, 2009.

[76] M. Pivtoraiko, D. Mellinger, and V. Kumar, "Incremental micro-uav motion replanning for exploring unknown environments," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on.* IEEE, 2013, pp. 2452–2458.

[77] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on.* IEEE, 2009, pp. 489–494.

[78] A. Ratnoo, P. Sujit, and M. Kothari, "Adaptive optimal path following for high wind flights."

[79] J. Reif and M. Sharir, "Motion planning in the presence of moving obstacles," *Journal of the ACM (JACM)*, vol. 41, no. 4, pp. 764–790, 1994.

[80] A. Richardson and E. Olson, "Iterative path optimization for practical robot planning," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on.* IEEE, 2011, pp. 3881–3886.

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

[82] ——, "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments," in *Robotics Research.* Springer, 2016, pp. 649–666.

[83] M. Rufli, J. Alonso-Mora, and R. Siegwart, "Reciprocal collision avoidance with motion continuity constraints," *IEEE Transactions on Robotics*, vol. 29, no. 4, pp. 899–912, 2013.

[84] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[85] T. Schouwenaars, É. Féron, and J. How, "Safe receding horizon path planning for autonomous vehicles," in *Proceedings of the Annual Allerton Conference On Communication Control and Computing. Vol. 40. No. 1. The University*, 2002.

[86] S. Shen, N. Michael, and V. Kumar, "Autonomous multi-floor indoor navigation with a computationally constrained mav," in *Safety, Security, and Rescue Robotics (SSRR), 2016 IEEE International Symposium on.* IEEE, 2016, pp. 23–27.

[87] Z. Shiller, F. Large, and S. Sekhavat, "Motion planning in dynamic environments: Obstacles moving along arbitrary trajectories," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 4. IEEE, 2001, pp. 3716–3721.

[88] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on.* IEEE, 1994, pp. 3310–3317.

[89] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[90] P. Sujit, S. Saripalli, and J. B. Sousa, "An evaluation of uav path following algorithms," in *Control Conference (ECC), 2013 European.* IEEE, 2013, pp. 3332–3337.

[91] K. Sun, K. Mohta, B. Pfrommer, M. Watterson, S. Liu, Y. Mulgaonkar, C. J. Taylor, and V. Kumar, "Robust stereo visual inertial odometry for fast autonomous flight," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 965–972, 2018.

[92] A. Tagliabue, M. Kamel, S. Verling, R. Siegwart, and J. Nieto, "Collaborative transportation using mavs via passive force control," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on.* IEEE, 2017, pp. 5766–5773.

[93] S. Tang, J. Thomas, and V. Kumar, "Hold or take optimal plan (hoop): A quadratic programming approach to multi-robot trajectory generation," *The International Journal of Robotics Research*, p. 0278364917741532.

[94] R. Tedrake, I. R. Manchester, M. Tobenkin, and J. W. Roberts, "Lqr-trees: Feedback motion planning via sums-of-squares verification," *The International Journal of Robotics Research*, vol. 29, no. 8, pp. 1038–1052, 2010.

[95] M. Turpin, N. Michael, and V. Kumar, "CAPT: Concurrent assignment and planning of trajectories for multiple robots," *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 98–112, 2014.

[96] V. Usenko, L. von Stumberg, A. Pangercic, and D. Cremers, "Real-time trajectory replanning for mavs using uniform b-splines and 3d circular buffer," *arXiv preprint arXiv:1703.01416*, 2017.

[97] J. Van Den Berg, S. J. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," in *Robotics research.* Springer, 2011, pp. 3–19.

[98] J. Van den Berg, M. Lin, and D. Manocha, "Reciprocal velocity obstacles for real-time multi-agent navigation," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on.* IEEE, 2008, pp. 1928–1935.

[99] J. van Den Berg, J. Snoeyink, M. C. Lin, and D. Manocha, "Centralized path planning for multiple robots: Optimal decoupling into sequential plans."

[100] J. P. van den Berg and M. H. Overmars, "Planning the shortest safe path amidst unpredictably moving obstacles."

[101] W. Van Loock, G. Pipeleers, and J. Swevers, "Time-optimal quadrotor flight," in *Control Conference (ECC), 2013 European.* IEEE, 2013, pp. 1788–1792.

[102] E. Verriest and F. Lewis, "On the linear quadratic minimum-time problem," *IEEE Transactions on Automatic Control*, vol. 36, no. 7, pp. 859–863, 1991.

[103] E. A. Wan and R. Van Der Merwe, "The unscented kalman filter for nonlinear estimation," in *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000.* Ieee, 2000, pp. 153–158.

[104] C. W. Warren, "Global path planning using artificial potential fields," in *Proceedings, 1989 International Conference on Robotics and Automation*, May 1989, pp. 316–321 vol.1.

[105] M. Watterson and V. Kumar, "Safe receding horizon control for aggressive mav flight with limited range sensing," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

[106] R. Wein, J. Van Den Berg, and D. Halperin, "Planning high-quality paths and corridors amidst obstacles," *The International Journal of Robotics Research*, vol. 27, no. 11-12, pp. 1213–1231, 2008.

[107] D. Wilkie, J. Van Den Berg, and D. Manocha, "Generalized velocity obstacles," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on.* IEEE, 2009, pp. 5573–5578.

[108] C. M. Wilt and W. Ruml, "When does weighted A* fail?" 2012.