POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Planning and trajectory generation methods for aggressive maneuvering of UAVs in the presence of obstacles

Tesi di Laurea Magistrale in
Aeronautical Engineering - Ingegneria Aeronautica

Author: **Marta Manzoni**

Student ID: 996282
Advisor: Prof. Davide Invernizzi
Co-advisors: Eng. Roberto Rubinacci
Academic Year: 2023-24

Abstract

Unmanned Aerial Vehicles (UAVs) represent a leap in engineering innovation, embodying autonomous flight capabilities without the need for human pilots on board. The achievement of full autonomy in UAVs is significantly dependent on effective motion planning. Specifically, it is essential to plan collision-free trajectories that allow the vehicle to transition from an initial to a final configuration. However, finding a solution executable by the actual system requires an additional level of complexity: the planned motion must be dynamically feasible. This involves meeting rigorous criteria that include vehicle dynamics, input constraints, and state constraints. In this thesis, the optimal motion planning problem for fast and aggressive maneuvering of UAVs in known cluttered environments is addressed by implementing two methodologies. First, a search-based approach is introduced, specifically designed for quadrotors, which uses motion primitives generated by discretization of the control input. Subsequently, a second method is proposed that addresses the motion planning problem for arbitrary system dynamics. This approach introduces a library of motion primitives created by discretization of the state space. This method reduces the computational load during online planning by shifting the computationally intensive part of computing the motion primitive to the offline phase. Both methods yield resolution-complete, resolution-optimal, collision-free, and dynamically feasible trajectories. Their versatility extends to dynamic and unknown environments, demonstrating a crucial capability for rapid re-planning. In particular, these methods are designed for real-time execution and are applicable to real-world autonomous navigation. The thesis meticulously analyzes the strengths and weaknesses of the proposed methods. Moreover, their performance is demonstrated through numerical examples, simulations, and real-world experiments.

Keywords: motion planning, autonomous vehicles, motion primitives.

Abstract in lingua italiana

Gli Aeromobili a Pilotaggio Remoto (APR) rappresentano un significativo avanzamento nell'innovazione ingegneristica, esibendo capacità di volo autonomo senza la necessità di piloti umani a bordo. Il raggiungimento della piena autonomia nei APR dipende significativamente da una pianificazione del movimento efficace. In particolare, è essenziale pianificare traiettorie prive di collisioni che consentano al veicolo di passare da una configurazione iniziale a una finale. Tuttavia, trovare una soluzione eseguibile dal sistema effettivo richiede un ulteriore livello di complessità: il movimento pianificato deve essere dinamicamente fattibile. Ciò implica il soddisfacimento di rigorosi criteri che includono dinamica del veicolo, vincoli di input e vincoli di stato. In questa tesi, il problema di pianificazione del movimento ottimale per manovre veloci e aggressive dei APR in ambienti conosciuti e ricchi di ostacoli è affrontato mediante l'implementazione di due metodologie. In primo luogo, viene introdotto un approccio basato sulla ricerca, progettato specificamente per quadrotori, che utilizza primitive di movimento generate mediante discretizzazione dell'input di controllo. Successivamente, viene proposto un secondo metodo che si occupa del problema di pianificazione del movimento per sistemi con dinamicha arbitraria. Questo approccio introduce una libreria di primitive di movimento creata mediante discretizzazione dello spazio di stato. Questo metodo riduce il carico computazionale durante la pianificazione online spostando la parte computazionalmente intensiva della generazione della primitiva di movimento alla fase offline. Entrambi i metodi producono traiettorie complete in risoluzione, ottimali in risoluzione, prive di collisioni e dinamicamente fattibili. La loro versatilità si estende a ambienti sconosciuti e con ostacoli dinamici, dimostrando una capacità per la rapida ri-pianificazione. In particolare, questi metodi sono progettati per l'esecuzione in tempo reale e sono applicabili alla navigazione autonoma del mondo reale. La tesi analizza meticolosamente i punti di forza e le debolezze dei metodi proposti. Inoltre, ne vengono dimostrate le prestazioni attraverso esempi numerici, simulazioni ed esperimenti reali.

Parole chiave: pianificazione del movimento, veicoli autonomi, primitive di movimento.

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Preliminary **7**
1.1 Path planning 7
1.2 Graph search algorithms 9
1.2.1 A* algorithm 9
1.3 System dynamics 16
1.3.1 Quadrotor dynamics 17

2 Kinodynamic motion planning **21**
2.1 Problem formulation 22
2.1.1 Quadrotor problem formulation 24
2.2 Kinodynamic planning with motion primitives 25
2.3 Search-based kinodynamic planning 26
2.3.1 Global planner and local planner 27
2.3.2 Heuristic function design 28
2.3.3 Collision checking 29

3 Search-based kinodynamic planning with online motion primitives generation **31**
3.1 Motion primitives generation 32
3.2 Graph construction 35
3.3 Kinodynamic motion planning 36
3.3.1 A* algorithm 38

3.4 Numerical examples . . . 40
3.4.1 Acceleration-controlled system . . . 40
3.4.2 Jerk-controlled system . . . 43
3.5 Conclusion . . . 47

4 Search-based kinodynamic planning with motion primitives library 49
4.1 Motion primitives generation . . . 50
4.1.1 Invariance properties . . . 51
4.1.2 Motion primitives library . . . 52
4.1.3 Example . . . 53
4.2 Search space design . . . 57
4.3 Kinodynamic motion planning . . . 58
4.3.1 A* algorithm . . . 58
4.4 Numerical examples . . . 61
4.4.1 Acceleration-controlled system . . . 61
4.4.2 Jerk-controlled system . . . 66
4.5 Conclusion . . . 71

5 Experimental results 73
5.1 Problem setup . . . 73
5.1.1 Simulations setup . . . 75
5.1.2 Real-world experiments setup . . . 76
5.2 Experimental test 1 . . . 76
5.2.1 Simulation 1 results . . . 76
5.2.2 Real-world experiment 1 results . . . 81
5.3 Experimental test 2 . . . 84
5.3.1 Simulation 2 results . . . 84
5.3.2 Experimental test 2 results . . . 88

6 Conclusions and prospective works 91
6.1 Conclusions . . . 91
6.2 Methods comparison . . . 92
6.3 Impact and applications . . . 93
6.4 Prospective works . . . 93

Bibliography 95

List of Figures **97**

List of Tables **101**

Introduction

Unmanned Aerial Vehicles (UAVs) showcase innovative engineering, enabling autonomous flight without the need for a human pilot on board. These autonomous platforms encompass a complex integration of aerospace, electronics, control systems, and software engineering. The versatility of UAVs is evident in their autonomous navigation and task execution, expanding their applications across a broad spectrum from civilian to military domains. These include surveillance, reconnaissance, package delivery, and exploration [3]. However, as their roles become more dynamic and demanding, the need for UAVs to navigate complex environments with agility and precision becomes increasingly essential. Consequently, the challenge of planning a path within an obstacle-cluttered environment has attracted significant attention within the communities of artificial intelligence and robotics in the last years. Most efforts in this domain have focused on addressing kinematic motion problems. However, kinematic motion planning, while effective in producing collision-free paths, may not guarantee their feasibility when executed by the actual systems. This holds particularly true for scenarios involving agile autonomous vehicles, where supplementary limitations on the vehicle's movement that arise from its dynamics or nonholonomic constraints must be considered. This is the central focus of a contemporary direction in motion planning research commonly known as kinodynamic motion planning. Kinodynamic planning combines finding a path that avoids collisions with considering the system's dynamics, ensuring that the resulting trajectory is both safe and executable by the vehicle.

This thesis tackles the challenge of optimal motion planning by taking into account system dynamics, state constraints, input constraints, and collision avoidance. It presents two distinct approaches: the first one employs a forward propagation method specifically designed for quadrotor dynamics, while the second addresses the kinodynamic motion planning problem for arbitrary system dynamics.

State of the art

Motion planning involves navigating from an initial position to a desired state while adhering to specified constraints. Active research in this domain has produced a multitude of trajectory-generation algorithms. Frazzoli et al. [4], [5] provided some of the pioneering work on real-time kinodynamic motion planning. The proposed technique involves mapping the vehicle's dynamics onto a finite-dimensional space, restricting the vehicle's potential states to two categories: either a trim state or a maneuver state. A first group of algorithms follows a two-step approach, separating geometric and temporal planning. Initially, a geometric trajectory is generated by linking a series of waypoints by straight-flight trim conditions. In this phase, time information is not taken into account. The path is then smoothed and parameterized in time by selecting the optimal sequence of trims and maneuvers from within a precomputed library of motion primitives. This results in a smoother trajectory while ensuring compliance with the vehicle dynamics [1]. Moreover, the real-time computation is facilitated by using closed-form expressions. All nonlinearities due to the vehicle model are confined within the stored library of motion primitives. A second group of algorithms leverage the differential flatness inherent in the dynamics of the quadrotor to derive constraints on the trajectory [12], [13], [8], [9]. Subsequently, an optimization problem is solved over a trajectory class, such as minimum snap [11]. These trajectories are expressed through time-parameterized polynomials, transforming the trajectory generation problem into the task of determining polynomial coefficients that meet specific constraints. However, this method is only applicable to systems with straightforward dynamics, such as quadrotors. Improvements are achieved by a third group of algorithms that address the motion planning for arbitrary system dynamics by introducing a database of precomputed motion primitives [15]. Here, the more computationally demanding aspect of motion planning is shifted to the offline phase of database construction.

Motion primitives, generate a discretization of the state space which allows a graph representation $G(V,E)$, where V is the set of reachable system states and E is the set of edges that connect states in the graph, each associated with a cost function. A key feature of motion primitives is the inherent translational, which play a key role in the process of constructing complete flight paths by arranging sequences of alternating trim trajectories and maneuvers. Given the graph, the problem consists of determining the optimal collision-free sequence of motion primitives that achieve the goal while minimizing a cost function. This problem can be solved using search-based [8], [9] or sampling-based [4], [5], [14], [15] algorithms, which differ mainly in the optimality of the solution and computational time.

Sampling-based methods avoid the explicit construction of the configuration space by randomly or systematically sampling the space to build a roadmap or a graph representation of the environment. Therefore, these algorithms are fast and particularly useful for complex and high-dimensional spaces. Furthermore, they have been demonstrated to be *asymptotically optimal*, meaning that as the cardinality of the tree grows toward infinity, the probability of discovering an optimal solution, if it exists, approaches one. They are not complete, but exhibit the property of *probabilistic completeness*. This means that as the number of samples approaches infinity, sampling-based methods return a solution with a probability converging to one if such a solution exists. Moreover, the utilization of a dynamic tree, whose size depends only on the number of samples and not on the number of motion primitives, contributes to the inherent speed of these algorithms. The most widely used types of sampling-based methods include Rapidly Exploring Random Trees (RRT) [14], [15], Probabilistic Roadmaps (PRM) and their variations. RRT presents strong theoretical properties, including probabilistic completeness, but its randomized approach could represent an obstacle when fast online replanning is required. In particular, paths generated by a sampling-based method during consecutive replanning periods may exhibit significant variations, making them less suitable for applications where path consistency is crucial [9].

Search-based methods, on the other hand, employ traditional search algorithms to navigate a predefined representation of the configuration space. The A* algorithm is a well-known example of a search-based method [8], [9]. These methods fall into the category of *exact* methods and are *complete*. This means that they will return a solution within a finite time if one exists, or they will correctly indicate its nonexistence. The primary distinction between search-based and sampling-based methods lies in their approach to motion planning. Search-based methods aim to find optimal solutions, prioritizing optimality, while sampling-based methods focus on finding feasible solutions, emphasizing practicality and efficiency. Nevertheless, obtaining the optimal solution in a reasonable time becomes extremely challenging in high-dimensional spaces. In cases where computing the optimal solution becomes computationally intractable, the randomized approach is more suitable. However, if computational time allows, the deterministic approach is always preferable, as it guarantees optimal solutions. Additionally, search-based methods can become more efficient in high-dimensional spaces when a heuristic function is taken into account [8]. However, the use of weighted heuristics often results in suboptimal solutions and does not necessarily lead to a reduction in planning time [9].

All the proposed planning methods model the UAV as a sphere. This choice facilitates the creation of a simple configuration space (C-space) by inflating the obstacles with the

dimensions of the robot. Consequently, the robot can be treated as a single point in C-space, simplifying collision checks. However, the spherical model assumption is very conservative, as it disregards numerous trajectories whose feasibility depends on the robot orientation. An alternative that considers an ellipsoid model is proposed by Liu et al. [9]. This model allows the quadrotor to pass through gaps that are smaller than its diameter with nonzero pitch or roll angles.

In the presence of aerodynamic effects, such as strong winds, most existing methods for planning quadrotor trajectories will not attempt to deviate from a determined plan, even if it is risky, under the assumption that a robust controller can counteract any aerodynamic perturbations. Wang et al. [17] introduce an innovative trajectory planning strategy designed to generate a safe and efficient path in unknown environments with aerodynamic disturbances. They utilized a real-time Gaussian process to model the errors induced by these aerodynamic effects.

In this thesis, the focus revolves around the design of a planner that can generate dynamically feasible, collision-free, globally optimal, and complete trajectories in real-time. The complexity of this task lies in considering system dynamics, input constraints, and collision avoidance. Two distinct approaches are explored: the first utilizes a forward propagation method meticulously designed for quadrotor dynamics, while the second addresses the kinodynamic motion planning problem for arbitrary system dynamics.

Contributions

The primary contribution of this thesis is the development of a trajectory planner and generator for fast and aggressive maneuvering of UAVs while satisfying vehicle dynamics, state constraints, input constraints, and avoiding obstacles. The proposed advanced planner and trajectory generator will offer a solution for UAVs to navigate efficiently in complex and challenging scenarios without collisions, especially when the vehicle is required to use its full maneuvering capabilities, and to react in real time to changes in the operational environment. This will lead to improvements in both efficiency and safety compared to existing approaches.

Specifically, this thesis delves into an in-depth analysis of the planning problem, presenting two key solutions as its main contributions. The first solution, drawing inspiration from the work presented in [8], introduces a search-based planner specifically tailored for quadrotors. This planner utilizes the forward-propagation approach to compute motion primitives. Subsequently, the effectiveness of this approach is demonstrated through numerical examples. The second solution, inspired by [15], goes beyond the quadrotor

domain, providing a versatile search-based planner. This algorithm integrates a database of offline precomputed solutions into the search-based planning framework, effectively alleviating the computational load associated with solving Two-Point Boundary Value Problems (TPBVPs). This method is validated through rigorous testing that involves both numerical examples and experimental tests.

Structure of the thesis

The thesis unfolds across several chapters, each addressing distinct aspects of the kinodynamic motion planning problem and proposing innovative solutions:

- Chapter 1 introduces the path planning problem accompanied by a presentation of the general framework that utilizes graph search algorithms to address this challenge. In addition, the chapter delves into an exploration of the dynamics of the system.
- Chapter 2 focuses on the formalization of the kinodynamic motion planning problem. Furthermore, an innovative technique employing motion primitives is introduced to mitigate the computational complexity inherent to this problem.
- Chapter 3 delves into the proposal of a search-based planner that uses motion primitives computed through a forward propagation method to solve the deterministic shortest trajectory problem specifically tailored for quadrotors. Moreover, numerical examples are proposed to demonstrate the effectiveness of the proposed approach. This chapter is largely inspired by the work presented in [8].
- Chapter 4 introduces a novel search-based algorithm that integrates a database of offline precomputed solutions. This method is designed to be applicable to a wider range of systems beyond quadrotors. To underscore the efficacy of this approach, the chapter provides two numerical examples. This chapter draws significant inspiration from the work proposed in [15].
- Chapter 5 presents two experiments designed to demonstrate the effectiveness of the approach introduced in Chapter 4. The focus is on navigating real-world cluttered environments, providing practical validation for the proposed solution.
- Chapter 6 presents conclusions and exposes potential avenues for future research, as well as exploring the potential applications of these approaches.

1 | Preliminary

In this chapter, graph search techniques for path planning are presented alongside the system dynamics. Specifically, Section 1.1 delineates the path planning problem and formulates it as an optimization problem. Section 1.2 introduces graph search algorithms for path planning, with a detailed exploration of the A* search algorithm framework and its characteristics. In Section 1.3, the dynamics of a generic nonlinear system are introduced and tailored for quadrotors, accompanied by an introduction to their differential flatness property.

1.1. Path planning

Path planning, is a fundamental problem in robotics. Its primary objective is to determine a sequence of valid configurations that enables a robot to move from an initial position to a designated destination while avoiding obstacles. This is a critical aspect of robotics, as it ensures that vehicles can navigate and operate safely and effectively in various environments.

The primary purpose in a standard path-planning problem is to identify a route with minimal cost between two nodes (or vertices). This path begins at the starting configuration, denoted as $\boldsymbol{q_S}$, and ends at the goal configuration, denoted as $\boldsymbol{q_G}$. Therefore, path planning represents an optimization problem featuring a cost function and some constraints, denoted as $J(\cdot)$ and $\phi(\cdot)$, respectively. This problem can be formulated as proposed in [7]:

$$\begin{aligned} &\min_{P} J(P) \\ &\text{s.t.} \;\; \phi(P). \end{aligned} \tag{1.1}$$

In the previous formulation, P is a path in $\mathbb{R}^m$ characterized by a starting point $\boldsymbol{q_S}$, a target point $\boldsymbol{q_G}$, and a sequence of n connected waypoints that the vehicle follows to reach its destination $P := \langle \boldsymbol{q_S} = \boldsymbol{q_0} \rightarrow \boldsymbol{q_1} \rightarrow \ldots \rightarrow \boldsymbol{q_N} \rightarrow \boldsymbol{q_{N+1}} = \boldsymbol{q_G} \rangle$. A path is composed of successive segments, with each segment generated by two consecutive waypoints along the path.

Typically, constraints $\phi(P)$ include conditions that the path must satisfy, such as constraints on initial and final configurations, as well as clearance from obstacles. Such constraints can be expressed as:

$$\begin{aligned} &\boldsymbol{q_S}, \boldsymbol{q_G} \in P, \\ &P \in \mathcal{C}^{free}, \end{aligned}$$

where $\mathcal{C}^{free}$ is the collision-free subspace of a given configuration space $\mathcal{C} \in \mathbb{R}^n$. The configuration space, also known as C-space, is the set of all possible configurations of the vehicle.

In the context of path planning, a "node" typically refers to a specific point or location within the environment. Nodes are elements of the set $\mathcal{V}$, which represents the set of vertices in a graph. Each node corresponds to a position that a robot might occupy within the configuration space. An "edge" in this context is a connection or link between two nodes. Edges are elements of the set $\mathcal{E}$, representing the set of edges connecting the vertices in the graph. These edges signify possible transitions or movements between different positions in the environment.

A common scenario within path planning involves addressing the deterministic shortest-path problem. This specific problem aims to identify the shortest path, characterized by minimizing the total distance between two points. In the given scenario, if the environment in which the vehicle operates is discretized into a graph, denoted as $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the deterministic shortest-path problem can be formulated as a graph search problem. The objective is to find a path that connects the starting point $\boldsymbol{q_S}$ to the goal point $\boldsymbol{q_G}$ within the configuration space. The problem is captured within the formulation presented in [7]:

$$\begin{aligned} \min_{P} \quad & \sum_{i=0}^{N-1} \|\boldsymbol{q_{i+1}} - \boldsymbol{q_i}\|_p \\ \text{s.t.} \quad & \boldsymbol{q}_0 = \boldsymbol{q_S}, \, \boldsymbol{q_N} = \boldsymbol{q_G}, \\ & \boldsymbol{q_i} \in \mathcal{V}\,, \text{ i} = 0,...,\text{N}, \\ & e(\boldsymbol{q_i}, \boldsymbol{q_{i+1}}) \in \mathcal{E}, \text{ i} = 0,...,\text{N-1}. \end{aligned} \tag{1.2}$$

The first constraint imposes that the path begins at the starting configuration $\boldsymbol{q_S}$ and terminates at the goal configuration $\boldsymbol{q_G}$. The second constraint ensures that all waypoints along the path belong to the set of vertices $\mathcal{V}$ and the final constraint guarantees that each waypoint along the path is connected to its adjacent neighbor. The function $e(\cdot, \cdot)$ represents the directional edge from one configuration to the other. To find a collision-free path, only the graph in the collision-free space is considered such that $\mathcal{G}(\mathcal{V}, \mathcal{E}) \in \mathcal{C}^{free}$.

Several algorithms can be leveraged to address this problem, including graph search al-

gorithms and sampling-based algorithms. The following section examines fundamental search-based algorithms for path planning, specifically the Dijkstra and A* algorithms.

1.2. Graph search algorithms

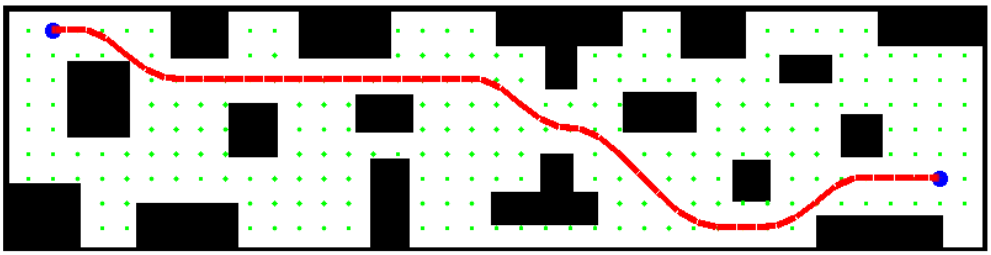
Graph search algorithms and sampling-based algorithms are two fundamental approaches employed in path planning to navigate environments. Graph search algorithms, such as Dijkstra's or A*, operate by systematically exploring nodes in a graph representation of the environment to find the optimal path. Their advantage lies in the ability to guarantee optimality, ensuring that the generated path is the most efficient. However, their main disadvantage is the computational demand, particularly in large and complex environments, which can limit real-time applications.

On the other hand, sampling-based algorithms, like the Rapid Exploring Random Trees (RRT) algorithm, focus on randomly sampling the configuration space and building a tree structure to connect sampled points. These algorithms excel at handling high-dimensional spaces and complex environments. They are computationally efficient, but their drawback is the lack of optimality guarantees.

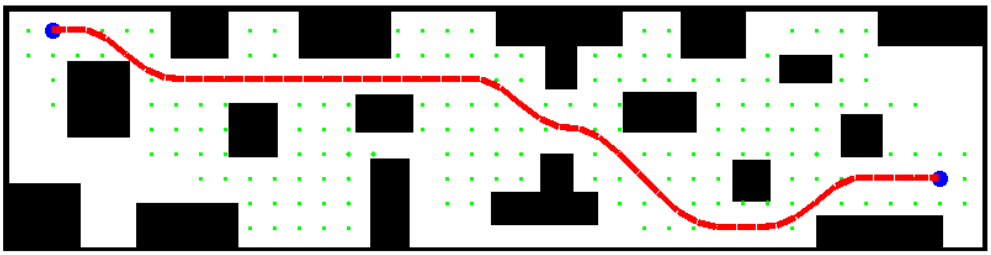
In the context of this thesis, which attempts to address the complexities of the kinedynamic motion planning problem, it has been decided to use graph search algorithms. A detailed explanation of this choice will be provided in the subsequent chapter, in which the kinodynamic motion planning problem will be introduced. Therefore, the following section will delve into a comprehensive exploration of the A* algorithm, shedding light on its main components.

1.2.1. A* algorithm

Dijkstra's [7] algorithm serves as a fundamental graph search approach to determine the shortest paths from a starting node to all other nodes in a graph, thereby generating a shortest-path tree. In this context, this thesis leverages the A* algorithm, an improved variant of Dijkstra's algorithm, to address Problem 1.2. Unlike Dijkstra's algorithm, the A* algorithm only identifies the shortest path from an initial node to a defined goal node. This trade-off is essential when employing a specific goal-directed heuristic to guide the search, which allows for better performance and speeds up the search. In the case of Dijkstra's algorithm, the complete shortest-path tree is constructed, treating each node as a potential goal. However, this approach precludes the use of goal-specific heuristics. Incorporating a goal-oriented heuristic into the A* algorithm involves the inclusion of a



(a) Dijkstra algorithm.



(b) A* algorithm.

Figure 1.1: Motion planning problem solved using the Dijkstra algorithm (a) and the A* algorithm (b). The blue dot on the right represents the starting position, whereas the one on the left indicates the final position. The red curve is the optimal trajectory. The green dots are the expanded nodes.

heuristic function that estimates the cost of reaching the goal from a given node, commonly referred to as the *cost-to-goal* value. The integration of this heuristic function is crucial as it can reduce the number of vertices to be expanded during the search phase compared to the uninformed exploration conducted by Dijkstra's algorithm.

Figure 1.1 illustrates a comparison between solving a motion planning problem using the Dijkstra algorithm and employing the A* algorithm. Remarkably, the use of a heuristic function is shown to decrease the number of expanded nodes (green dots). Additionally, it is noteworthy that despite this reduction in computational load, the resulting trajectory remains identical. This is attributed to the admissibility and consistency of the heuristic function, ensuring that the A* algorithm identifies the optimal trajectory.

It is noteworthy to highlight the incremental planning characteristic shared by these algorithms. In this approach, the complete graph is not established from the beginning;

instead, it is dynamically constructed on the fly during the search process. This dynamic construction allows the algorithms to adapt flexibly to the specifics of the motion planning task at hand.

Properties

A* algorithm possesses the following properties:

- **Completeness**: A* algorithm is complete, meaning it will always find a feasible solution if one exists. In the cases where no solution is present, it will return FAIL within a finite time.
- **Optimality**: A* algorithm is optimal when using an admissible heuristic, guaranteeing that it finds the path with minimum cost among all possible paths from the start to the goal.
- **Efficiency**: A* is efficient when an effective heuristic is used, as it can significantly reduce the number of nodes explored during the search.

Heuristic function

In order to enable these improvements, the heuristic function must exhibit the following two properties:

- **Admissible**: the heuristic should not overestimate the actual cost to reach the goal.

  $$h(v_i) \leq h^*(v_i).$$

  If the heuristic function is admissible, the A* algorithm guarantees finding the least-cost path from the starting point to the goal, thus ensuring an optimal solution. Typically, when a heuristic closely approximates the true *cost-to-goal* value, the algorithm operates faster, as it tends to explore fewer vertices.

- **Consistent**: for every node in the graph v_i and for each successor v_j of v_i, the estimated *cost-to-goal* from v_i must always be less than or equal to the estimated *cost-to-goal* from v_j plus the step cost to reach v_j. In addition, the heuristic value for the goal node is always zero.

  $$h(v_i) \leq h(v_j) + c(v_i, v_j),$$
  $$h(v_g) = 0.$$

A consistent heuristic is also admissible, however, the converse is not always true [7]. Dijkstra's algorithm can be viewed as a special case of A* where $h(v_i) = 0$ for all nodes.

Configuration space

Graph search algorithms play a fundamental role in navigating complex spaces, and their utility extends notably to the domain of configuration spaces. The configuration space, denoted as $\mathcal{C}$, is the set of all possible configurations of the vehicle, with its topological characteristics and dimension varying according to the particular system in focus. As an example, the configuration space of a rigid free-flying body corresponds to the Special Euclidean group of dimension three, denoted as $SE(3)$. In general, the configuration space dimension is defined by the minimum number of DoFs needed to completely specify a vehicle configuration. The free configuration space $\mathcal{C}_{free}$ represents the set of collision-free configurations, that is, $\mathcal{C}_{free} := \mathcal{C}/\mathcal{C}_{obs}$ where $\mathcal{C}_{obs}$ is a subset of $\mathcal{C}$ comprising the configurations that lead to collisions.

The state of the dynamic system, denoted $\boldsymbol{x}$, includes the configuration $\boldsymbol{q}$, first-order derivatives, and potentially higher-order derivatives, depending on the specific problem. Thus, $\boldsymbol{x} = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \ldots] \in \mathcal{S}$ and $\boldsymbol{q} = [x, y, z] \in \mathcal{C}$. The free state space, represented as $\mathcal{S}_{free}$, constitutes a subset of the state space $\mathcal{S}$. It delineates the free region of the state space, encapsulating not only obstacle-free configurations $\mathcal{C}_{free}$, but also constraints related to the dynamics of the system $\mathcal{D}_{free}$. These constraints include maximum velocity $\bar{v}_{max}$, acceleration $\bar{a}_{max}$, and potentially higher-order derivatives. Thus,

$$\begin{aligned} &\mathcal{S}_{free} = \mathcal{C}_{free} \times \mathcal{D}_{free} \\ &\text{where } \mathcal{D}_{free} = [-\bar{v}_{max}, \bar{v}_{max}]^3 \times [-\bar{a}_{max}, \bar{a}_{max}]^3 \times \ldots. \end{aligned}$$

The workspace, denoted as $\mathcal{W}$, is a subset of the 2D or 3D Euclidean space where the vehicle operates. The process of checking for collisions between the vehicle and the obstacles within the workspace can be complex and non-intuitive. To address this complexity, path planning is performed in the configuration space, subsequently enabling the determination of the optimal collision-free path within the workspace.

Graph representation

The configuration space $\mathcal{C}$ can be represented by a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of vertices representing different configurations, and $\mathcal{E}$ is the set of edges connecting these configurations. In this graph representation, each vertex corresponds to a unique configuration, while the edges delineate feasible transitions between these configurations,

often influenced by permissible movements or transformations of the system. This graph model serves as a valuable framework for employing graph search algorithms to navigate and explore the configuration space efficiently, aiding in the determination of optimal paths or sequences of configurations that adhere to specific criteria or constraints imposed by the system.

An occupancy grid is a special type of graph that is used to represent the robot workspace as a discrete grid. It divides the physical space into a grid of cells, where each cell represents a small portion of area. These cells store information about the occupancy status, typically indicating whether a particular portion of space is occupied by an obstacle, free for traversal, or unknown. This information about the environment can be collected from sensors in real time or loaded from prior knowledge. Within an occupancy grid map, each cell can be assigned a value that signifies the likelihood of occupancy. For instance, cells might be labeled as occupied if they contain obstacles, free if they are accessible, or unknown if there is insufficient information available about that region. Typically, values of 0 denote free space, 100 mark obstacles, and -1 signify unknown regions. This grid-based representation acts as a pivotal framework, especially in the domain of robotic navigation and path planning. Each grid cell directly translates into a graph vertex, while the interconnections between cells are dictated by the grid layout, linking adjacent cells in an organized manner. Consequently, this graph structure facilitates efficient path planning, enabling robots to maneuver through their environment effectively. In this thesis, the environment will be represented using the described occupancy grid map.

Once the graph is created, the A* algorithm can be applied to find the optimal path, and the planned path in the configuration space will also be safe in the original workspace. Thus, planning in the C-space is equivalent to planning in the original workspace.

Nodes attributes

The following notation is used in the A* pseudocode 1.1 presented in the next section. For each node in the graph, denoted $v \in \mathcal{V}$, the attributes reported in Table 1.1 are defined.

Priority queue and closed list

Both a priority queue $\mathcal{Q}$, also known as an open list, and a closed list $\mathcal{Z}$ are used in the A* algorithm. The priority queue collects the neighboring nodes of the expanded nodes, serving as candidates for the next expansion. The closed list stores the already expanded nodes that will not be considered again. The functions listed in Table 1.2 and Table 1.3 are used to manage the closed list and the priority queue, respectively.

Table 1.1: Notation employed for each vertex v in the graph

$g(v)$	*start-to-state* value of element v
$h(v)$	*state-to-goal* value of element v
$f(v)$	priority value of element v
Succ(v)	one-step successors of element v
Pred(v)	one-step predecessors of element v

Table 1.2: Function employed to manage the closed list.

Add(v)	add the element v to the closed list $\mathcal{Z}$

A* pseudo-code

The pseudo-code for the A* algorithm is presented in Algorithm 1.1 and Algorithm 1.2. In the `AStar` function, lines 2 to 7 encompass the algorithm initialization phase. Specifically, lines 2 and 3 establish the priority queue $\mathcal{Q}$ and the closed list $\mathcal{Z}$ as empty sets, while lines 4 to 7 initialize the g and h values for all nodes to infinity. Subsequently, the starting vertex v_s is assigned with an initial g-value of zero. Its heuristic cost is then computed and the node is placed into the initially empty priority queue. The main steps of the algorithm are written in the while loop from lines 12 to 36. The main loop continues until the priority queue is empty or the goal is reached. During each iteration, a node v with the highest priority value ($f(v) = g(v) + \epsilon h(v)$) is dequeued. If the dequeued node is the goal, the algorithm terminates, and the optimal path is found. Otherwise, the algorithm expands the node v by generating its successors s and updating their $g(s)$ and $h(s)$ values. These successors are then enqueued based on their priority value $f(s)$. Upon reaching the goal node v_g, the optimal path calculated by the `AStar` function is reconstructed by the `RecoverPath` procedure by backtracking from the goal to the start using the predecessors list and the recorded g values.

Furthermore, ϵ represents the weight of the heuristic function and can be adjusted according to the specific purpose. For example, the Dijkstra algorithm can be considered as a special case of the A* algorithm with ϵ set to 0. In certain scenarios, setting ϵ to a value greater than 1 can speed up the computation, even though it may render the heuristic value inadmissible, possibly leading to a non-optimal solution.

Algorithm 1.1 A* algorithm. Given the start node v_s and the goal node v_g, the A* algorithm computes the optimal path for Problem 1.2.

```
1: function AStar(u_s, g_s, ϵ)
2: Q ← ∅
3: Z ← ∅
4: for all v ∈ V do
5:    g(v) ← ∞;
6:    h(v) ← ∞;
7: end for
8: g(v_s) ← 0;
9: h(v_s) ← getHeuristic(v_s);
10: f(v_s) ← g(v_s)+ϵh(v_s);
11: Q.Push(v_s);
12: while Q ≠ ∅ do
13:    v ← Q.Top();
14:    Q.Pop();
15:    Succ(v) ← getSuccessors(v)
16:    for all s ∈ Succ(v) do
17:       Z.Add(s)
18:       if v ∉ Pred(s) then
19:          Pred(s) ← Pred(s) ∪ {v}
20:       end if
21:       h(s) ← getHeuristic(s);
22:       g_tmp ← g(v) + cost(v, s);
23:       if g_tmp < g(s) then
24:          g(s) ← g_tmp;
25:          f(s) = g(s) + ϵh(s);
26:          if s ∈ Q then
27:             Q.Update(s, f(s))
28:          else
29:             Q.Insert(s, f(s));
30:          end if
31:       end if
32:    end for
33:    if v = v_g then
34:       return RecoverPath(v);
35:    end if
36: end while
37: return Failure;
38: end function
```

Table 1.3: Functions employed to manage the priority queue.

Push(v)	add the element v at the end of $\mathcal{Q}$
Top()	return the element with the highest priority
Pop()	delete the element with the highest priority
Insert(v, f)	insert the element v with priority f in $\mathcal{Q}$
Remove(v)	remove the element v from $\mathcal{Q}$
Update(v, f)	update the priority value f of element v
ShiftUp()	preserve the list order when inserting or updating an element
ShiftDown()	preserve the list order when inserting or updating an element

Algorithm 1.2 Function to recover the optimal path computed by the A* algorithm.

```
1: function RecoverPath(v, Pred(v))
2: P ← ∅;
3: PredList(v) ← Pred(v)
4: for all p' ∈ PredList(v) do
5:     p ← arg min_p' (g(p') + cost(p', v));
6:     P ← ⟨p, P⟩;
7:     v ← p;
8: end for
9: return P;
10: end function
```

1.3. System dynamics

The dynamics of a generic nonlinear system are described by a set of ordinary differential equations that govern its motion and evolution over time, given by:

$$\dot{x}(t) = f(x(t), u(t)), \tag{1.3}$$

where $u(t) \in \mathcal{U}$ denotes the control input, and $x(t) \in \mathcal{S}$ is the state. The sets $\mathcal{S}$ and $\mathcal{U}$ represent the state space and the control space, respectively. A system can be subjected to physical constraints arising from its dynamics. These constraints, expressed as functional equalities and/or inequalities, restrict the range of values that can be assumed by control and/or state variables.

1.3.1. Quadrotor dynamics

In this thesis, all examples and simulations will be carried out using quadrotor UAVs, whose dynamics are exposed in this section and drawn from the insights provided in [11].

The quadrotor is modeled as a rigid body system with six degrees of freedom (DOFs), including position and orientation within a three-dimensional space $\mathbb{R}^3$. Three DOFs pertain to linear translations along three mutually perpendicular inertial axes, and three DOFs correspond to the rotation of the body frame $\mathcal{B}$ with respect to the inertial frame $\mathcal{W}$, described by the rotation matrix $\mathbf{R}_{\mathcal{B}}^{\mathcal{W}}$. A model of the quadrotor along with the coordinate systems considered, namely the world inertial frame, denoted as $\mathcal{W}$, and the body attached frame, represented as $\mathcal{B}$, are illustrated in Figure 1.2.

Euler angles are employed to define the roll ϕ, pitch θ, and yaw ψ angles. The robot's angular velocity, represented by $\omega_{\mathcal{BW}}$, denotes the angular velocity of the body frame $\mathcal{B}$ with respect to the world frame $\mathcal{W}$. Its components in the body frame are denoted as p, q, and r, and these values can be directly related to the derivatives of the roll, pitch, and yaw angles.

$$\omega_{\mathcal{BW}} = p\mathbf{x_B} + q\mathbf{y_B} + r\mathbf{z_B}.$$

The state vector of a quadrotor can be represented by the position and velocity of its center of mass, the Euler angles and the angular velocity:

$$\boldsymbol{x} := [x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \dot{z}, p, q, r]^T.$$

Each rotor operates with an angular velocity ω_i, generating both a force, F_i, and a moment, M_i, according to the following formulas:

$$\begin{aligned} F_i &= k_F \omega_i^2, \\ M_i &= k_M \omega_i^2. \end{aligned} \tag{1.4}$$

The control input of the quadrotor system is denoted as $\mathbf{u} = [u_1, u_2, u_3, u_4]^T$, where the first component indicates the net body force, and the subsequent three components represent the body torques. According to 1.4, the control input components can be written as:

$$\begin{aligned} u_1 &= k_F\omega_1^2 + k_F\omega_2^2 + k_F\omega_3^2 + k_F\omega_4^2, \\ u_2 &= Lk_F\omega_2^2 - Lk_F\omega_4^2, \\ u_3 &= -Lk_F\omega_1^2 + Lk_F\omega_3^2, \\ u_4 &= k_M\omega_1^2 + k_M\omega_2^2 + k_M\omega_3^2 + k_M\omega_4^2, \end{aligned}$$

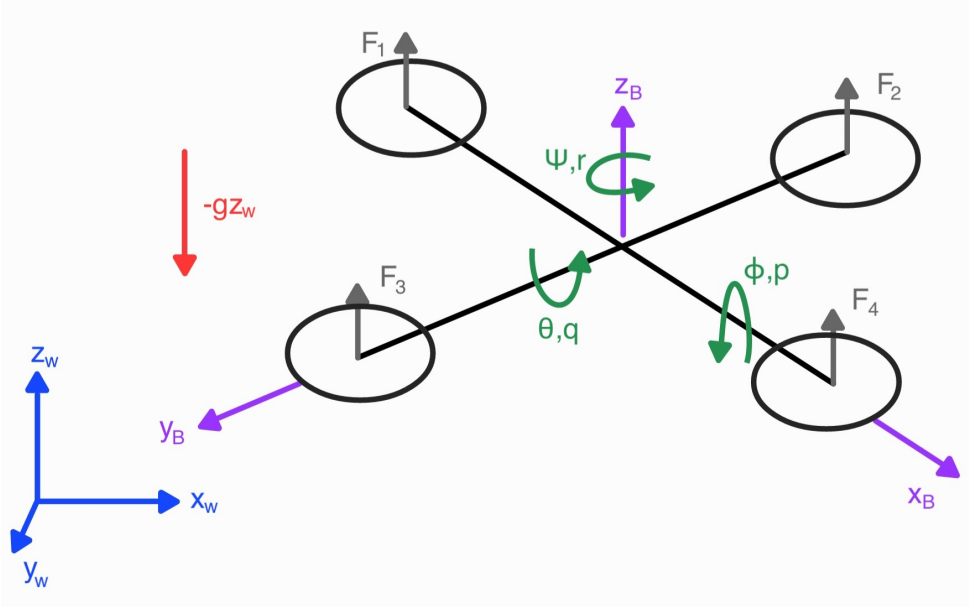


Figure 1.2: Quadrotor model and reference frames.

where L is the distance from the center of the rotors to the center of the quadrotor.

The mass-normalized differential equations of motion governing the acceleration of the center of mass are:

$$\boldsymbol{a} = f\boldsymbol{z_B} - g\boldsymbol{z_W}, \tag{1.5}$$

where f is the thrust input u_1 normalized by the quadrotor mass m, resulting in units of acceleration. It is important to observe in Figure 1.2 that the motors generate a force that is perpendicular to the quadrotor's x-y plane, thus directed along the $\boldsymbol{z_B}$ body axis. $g\boldsymbol{z_W}$ is the gravity acceleration in the inertial frame, aligned with the negative $\boldsymbol{z_W}$ axis.

The selected model neglects aerodynamic effects, which are taken into account in more complex models [17]. Nonetheless, this simple and symmetric rigid model captures the essential dynamics and is suitable for solving trajectory generation problems.

Differential flatness

Differential flatness is a valuable property that can greatly simplify the generation and planning of trajectories for complex systems. A system can be defined differentially flat if there exists a subset of the output, called flat output, such that the state and input can be defined as functions of the flat output and a finite number of its derivatives. More

precisely, a nonlinear system

$$\dot{x} = f(x, u), \quad x \in \mathbb{R}^n, u \in \mathbb{R}^m$$
$$y = h(x), \quad y \in \mathbb{R}^m$$

is differentially flat if there exists a flat output

$$\sigma = \sigma(x, u, \dot{u}, \ldots, u^{(l)}), \quad \sigma \in \mathbb{R}^m$$

such that the state x and the input u can be defined as smooth functions of this flat output and its derivatives as follows:

$$x = h_x(\sigma, \dot{\sigma}, \ddot{\sigma}, \ldots, \sigma^{(l)}),$$
$$u = h_y(\sigma, \dot{\sigma}, \ddot{\sigma}, \ldots, \sigma^{(l)}).$$

The complete state of the quadrotor system is inherently nonlinear and typically challenging to directly compute. As demonstrated in [11], the quadrotor with the four inputs is a flat system, characterized by the following flat output:

$$\sigma = [x, y, z, \psi]^T. \tag{1.6}$$

Therefore, the flat output can be used to explicitly express the state of the system and the control input in terms of σ and a finite number of its derivatives. The translational components of the state of the system can easily be derived from the first three elements of σ. To demonstrate that the rotation $\mathbf{R}_{\mathcal{B}}^{\mathcal{W}}$ is a function of the flat outputs σ, a modification of Equation 1.5 can be considered (see [11] for further details). Therefore, since both the translational part of the state and orientation are dependent on flat outputs σ, their respective derivatives are also functions of σ and its derivatives. Moreover, differential flatness implies that any trajectory in the state of the flat outputs is flyable if their derivatives are correctly bounded.

2 | Kinodynamic motion planning

Path planning considers the geometric aspect of the problem while addressing constraints that are primarily kinematic in nature, typically arising from the presence of obstacles in the environment. However, while path planning effectively produces collision-free paths, it may not guarantee their feasibility when executed by the actual systems. This aspect holds particularly true for scenarios involving agile autonomous vehicles, where additional limitations on the vehicle's movement that arise from its dynamics or nonholonomic constraints - constraints that cannot simply be integrated to impose a restriction on the position components - must be considered. Consequently, there is a strong preference for motion planning approaches that incorporate the vehicle's dynamics. This is the central focus of a contemporary direction in motion planning research commonly known as kinodynamic motion planning. Kinodynamic planning goes beyond collision avoidance, integrating the system's dynamics and input constraints. This approach ensures that the resulting trajectory not only avoids collisions but also remains feasible and executable by the vehicle. Fundamentally, the essence of kinodynamic motion planning lies in the task of determining the control inputs for a vehicle. These inputs enable the system to smoothly transition from an initial configuration to a desired target configuration, all while respecting its dynamical constraints and avoiding obstacles in the environment.

Kinodynamic motion planning considers both the kinematics and dynamics of a system simultaneously, requiring a direct exploration of the system's state space. This approach significantly increases the complexity of motion planning by introducing higher-dimensional spaces and differential constraints imposed by the system's dynamics. These complexities pose notable challenges, particularly evident when it comes to real-time trajectory planning. While online kinematic path planning requires minimal computational effort, the shift to online kinodynamic motion planning turns into a considerably more computationally demanding task, especially for complex systems.

This chapter is dedicated to formalizing the kinodynamic motion planning problem. Section 2.1 formulates the kinodynamic motion planning problem as an optimization problem. In Section 2.1.1, the problem is reformulated specifically for the quadrotor system. Next,

in Section 2.2, a technique is introduced to alleviate the computational complexity of kinodynamic motion planning for complex systems by employing motion primitives. Finally, Section 2.3 introduces the fundamental components of a search-based kinodynamic planner.

2.1. Problem formulation

Within the domain of motion planning applications, the resolution to the planning problem extends beyond mere feasibility and collision avoidance. In practical scenarios, the planning problem involves the fulfillment of additional requirements, such as reaching the goal in minimal time, minimizing control effort, and maximizing safety. This has redirected the focus from solely designing collision-free and feasible trajectories to the pursuit of optimal solutions. In light of these considerations, the kinodynamic motion planning problem is cast as an optimization problem in which the objective is to find the input functions $u(t)$ and the corresponding state trajectories $x(t)$ that minimize a cost function while adhering to specified constraints. The formulation of an optimal control problem is as follows:

$$\min_{x(t),u(t)} J = \int_0^T g(x(t), u(t))\, dt \tag{2.1a}$$

$$\text{s.t. } \dot{x} = f(x(t), u(t)), \tag{2.1b}$$

$$x(t) \in \mathcal{S}_{free}, \quad \forall t \in [0, T] \tag{2.1c}$$

$$u(t) \in \mathcal{U}, \quad \forall t \in [0, T] \tag{2.1d}$$

$$x(0) = x_i, \tag{2.1e}$$

$$x(T) = x_f. \tag{2.1f}$$

In this formulation, J is the cost function we want to minimize, where g is an arbitrary function, depending on the specific motion planning problem being examined. Equation 2.1b enforces the system's dynamics. Conditions 2.1c and 2.1d outline the state constraints and the control input constraints. The initial and final conditions in 2.1e and 2.1f correspond to the boundary constraints. In general, the decision variables $x(t)$ and $u(t)$ are continuous functions and finding a solution to the optimal control problem is challenging.

In this thesis, the specific objective is to compute a trajectory guiding the vehicle from an initial state x_i to a defined final state x_f, striking a balance between minimizing the total time T and minimizing the effort of the trajectory. Therefore, the cost function in 2.1a

takes the form:

$$J = J_e + J_T = \int_0^T \|u(t)\|^2 \, dt + \rho T, \tag{2.2}$$

Here, the integral term J_e represents the effort of the trajectory, interpreted as an upper bound on the average of a product of the inputs to the system [12]. The control action $u(t)$ can be replaced by velocity $v(t)$, acceleration $a(t)$, jerk $j(t)$, or snap $s(t)$, each corresponding to the objective of achieving minimum velocity, minimum acceleration, minimum jerk and minimum snap trajectories, respectively. The parameter $\rho \geq 0$ in the second term governs the trade-off between the duration of the trajectory T and its effort, determining their relative importance in the optimization process. In practice, when multiple solutions are available, all capable of achieving a common high-level goal, this cost function serves as a valuable tool to assess the aggressiveness of the input of these trajectories [13].

This thesis considers state spaces incorporating first-order derivatives (velocity components), as well as higher-order derivatives depending on the particular problem at hand, including acceleration, jerk, and snap. Consequently, the solution to the kinodynamic motion planning problem is a trajectory that simultaneously provides a collision-free path and profiles for velocity, acceleration, and higher-order derivatives. A trajectory is formally represented as a time-parameterized function that delineates a vehicle's desired states throughout time. Leveraging the formulation proposed in [9], the trajectory of the vehicle in this study is described as a piece-wise polynomial function. This representation involves expressing the trajectory, denoted as $\Gamma(t)$, as a composite function comprising N segments:

$$\Gamma(t) = \begin{cases} \Gamma_1(t - t_0), & t_0 \leq t < t_1 \\ \Gamma_2(t - t_1), & t_1 \leq t < t_2 \\ \Gamma_3(t - t_2), & t_2 \leq t < t_3 \\ \vdots & \\ \Gamma_N(t - t_{N-1}), & t_{N-1} \leq t < t_N. \end{cases} \tag{2.3}$$

Each segment of the trajectory Γ_i can be obtained through various methods. Firstly, by forward integration of the state equation in 1.3, achieved by applying a control action $u(t)$ over a duration $\Delta t_i = t_{i+1} - t_i$. This method will be explored in Chapter 3. Alternatively, each segment can be computed by solving a Two-Point-Boundary-Value problem (TPBVP). This method will be explored in Chapter 4. A visual representation of this trajectory is shown in Figure 2.1.

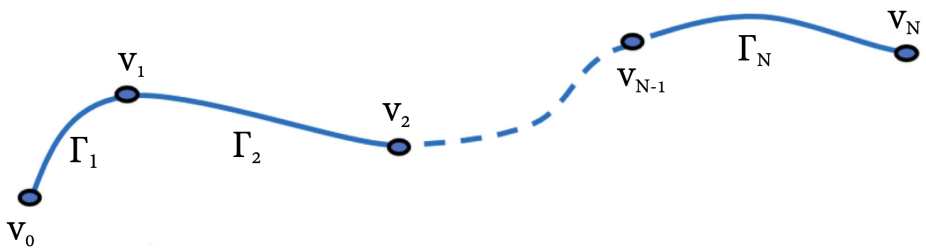


Figure 2.1: Piece-wise polynomial trajectory.

2.1.1. Quadrotor problem formulation

As explained in Section 1.3.1, the quadrotor dynamics are differentially flat, with flat output $\sigma = [x, y, x, \psi]$. Therefore, for the differentially flat system, the objective function can be rewritten as:

$$J(x(t), u(t)) = J(h_x(\sigma, \dot{\sigma}, \ddot{\sigma}, \ldots, \sigma^{(l)}), h_u(\sigma, \dot{\sigma}, \ddot{\sigma}, \ldots, \sigma^{(l)})) = J(\sigma, \dot{\sigma}, \ddot{\sigma}, \ldots, \sigma^{(l)})$$

The quadrotor inputs can be written in terms of the q^{th} derivative of the position (x, y, z). Therefore, to obtain a trajectory with low control effort, the q^{th} derivative of the position can be minimized. This is equivalent to minimizing the q^{th} derivative of the first three entries of the flat output. In quadrotor systems, the yaw angle ψ is typically negligible when the goal is to obtain a smooth trajectory and, as such, it is omitted. Consequently, the control effort term J_e of the objective function in 2.2 can be reformulated as:

$$J_e(\sigma, \dot{\sigma}, \ddot{\sigma}, \ldots, \sigma^{(l)}) = \int_0^T (x^{(q)}(t))^2 \, dt + \int_0^T (y^{(q)}(t))^2 \, dt + \int_0^T (z^{(q)}(t))^2 \, dt$$

The choice of the derivative q being minimized results in different trajectories. Specifically, for $q = 1$, the trajectory minimizes velocity. For $q = 2$, it minimizes acceleration. For $q = 3$, it minimizes jerk. Finally, for $q = 4$, the trajectory minimizes snap.

These steps result in the decomposition of the quadrotor dynamics into three mutually orthogonal inertial axes. Each axis can be treated as a double or triple integrator, as performed in various works such as [8] and [11].

Reformulating the trajectory expression in 2.3 for the quadrotor case involves representing it as a piece-wise polynomial function with respect to the flat outputs. The differential flatness inherent in quadrotor systems enables the construction of control inputs based on 1D time-parameterized polynomial trajectories independently specified along each axis of the flat output vector $\sigma = [x, y, z, \psi]^T$. Consequently, a trajectory $\Gamma(t)$ encompasses four

dimensions, denoted respectively as $\Gamma_x(t)$, $\Gamma_y(t)$, $\Gamma_z(t)$, and $\Gamma_\psi(t)$. The omission of the yaw angle results in the exclusion of $\Gamma_\psi(t)$ from the optimization formulation.

Following [7], $x(t)$ is a dynamical system state, consisting of position and its $(m-1)$ derivatives (velocity, acceleration, jerk, snap, etc.) in a three-dimensional space.

$$x(t) = [p(t)^T, \dot{p}(t)^T, \ddot{p}(t)^T, \ldots, p^{(m-1)}(t)^T]^T.$$

Position along each axis can be formulated as an n-th order polynomial, given by:

$$p(t) = c_n \frac{t^n}{n!} + c_{n-1} \frac{t^{n-1}}{(n-1)!} + \ldots + c_1 t + c_0 \tag{2.4}$$

The additional polynomial states along a single axis can be expressed in terms of the position $p(t)$ and its derivatives as follows:

$$\begin{aligned}
v(t) = \dot{p}(t) &= c_n \frac{t^{n-1}}{(n-1)!} + c_{n-1} \frac{t^{n-2}}{(n-2)!} + \ldots + c_2 t + c_1, \\
a(t) = \ddot{p}(t) &= c_n \frac{t^{n-2}}{(n-2)!} + c_{n-1} \frac{t^{n-3}}{(n-3)!} + \ldots + c_3 t + c_2, \\
j(t) = \dddot{p}(t) &= c_n \frac{t^{n-3}}{(n-3)!} + c_{n-1} \frac{t^{n-4}}{(n-4)!} + \ldots + c_4 t + c_3, \\
s(t) = \ddddot{p}(t) &= c_n \frac{t^{n-4}}{(n-4)!} + c_{n-1} \frac{t^{n-5}}{(n-5)!} + \ldots + c_5 t + c_4.
\end{aligned}$$

The complete state of the system dynamics in a three-dimensional space can be conveniently represented through individual polynomials along the three axes. For instance, the trajectory segment $\Gamma_i(t)$ can be expressed as $\Gamma_i(t) = [p_x^i(t), p_y^i(t), p_z^i(t)]^T$, where $p_k^i(t)$, with $k = x, y, z$ denotes the polynomial 2.4 corresponding to the x, y, and z-axes for the i-th segment.

2.2. Kinodynamic planning with motion primitives

Solving kinodynamic motion planning problems requires a significant computational load, especially for complex systems. This complexity arises from the method's incorporation of higher-dimensional spaces and differential constraints resulting from the system's dynamics. One potential strategy to reduce the computational complexity of kinodynamic motion planning for a nonlinear, high-dimensional system is based on a discretization, which involves choosing a finite set of motion primitives. Motion primitives, as implied by their name, represent a collection of precomputed motions tailored for specific dynamic

systems. These precalculated solutions address sub-problems, which can be concatenated to form a complete trajectory that effectively resolves the motion planning problem. These primitives, when properly interconnected, constrain the admissible trajectories of the system to a family of time-parameterized curves. As a result, instead of solving an optimal control problem in a continuous and high-dimensional space, the dynamics of the system are restricted to switching between a finite number of motion primitives [4]. This discretization leads to an approximation, meaning that not all trajectories, which are potential solutions to the vehicle's equations of motion, can be executed by the discretized dynamical system. Consequently, the computed solution is inherently suboptimal [1]. To address this suboptimality, a more refined discretization can be employed.

2.3. Search-based kinodynamic planning

Addressing kinodynamic motion planning challenges involves a careful consideration of algorithmic choices, with search-based algorithms and sampling-based approaches emerging as key contenders in this domain. Sampling-based methods, oriented towards finding feasible solutions, contrast with deterministic methods that strive for optimality. The choice between these approaches significantly influences the efficiency and quality of motion planning solutions. Sampling-based approaches offer a valuable advantage in terms of computational speed, providing solutions in short timeframes. This characteristic makes them particularly suitable for addressing problems in high-dimensional spaces, which is common in scenarios involving complex and nonholonomic systems. Graph search methods may struggle to find optimal solutions in a reasonable time in such complex spaces, making randomized approaches more suitable. However, the drawback of randomized approaches lies in their unpredictability, especially when fast online replanning is required. Trajectories generated by a sampling-based method during consecutive re-planning periods may exhibit significant variations, making them less suitable for applications where path consistency is crucial [7]. In contrast, deterministic planning approaches are characterized by their ability to generate higher-quality paths and are especially preferable when computational time permits. The deterministic approach prioritizes optimality, a critical factor in scenarios where finding the best possible solution is crucial. In addition, the incorporation of heuristic functions within the search-based framework serves as a powerful tool to accelerate the search process, enabling a more efficient exploration of the state space. The adaptability of search-based algorithms in dynamic scenarios positions them as robust solutions for future advancements in this thesis. Consequently, the A* method is specifically chosen to address the kinodynamic motion planning problem.

Transitioning to a more detailed exploration, in the following sections the fundamental components of a search-based kinodynamic planner will be explained.

2.3.1. Global planner and local planner

Kinodynamic motion planning for autonomous vehicles is composed of two main components: a local planner and a global planner. The local planner is responsible for producing a valid trajectory between two states of the system, neglecting collisions with obstacles (it operates independently of the environmental map). Specifically, within the graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the set of vertices denoting different states, and $\mathcal{E}$ is the set of edges connecting these states, the local planner's role is to generate feasible transitions between these states, essentially creating the edges of the graph. The local planner effectively acts as a local trajectory generator, specialized in computing feasible trajectories without considering obstacles. For the local trajectory to be valid, it must satisfy criteria that ensure the adherence to dynamic constraints and input constraints.

Complementing the local planner, the global planner is the overall algorithmic process responsible for solving the motion planning problem. It explores the state space of the system, taking into account the map of the environment and the obstacles. Specifically, the global planner generates a trajectory from the start state to the goal state through multiple calls to the local planner. It is essential to note that the local trajectories generated by the local planner do not inherently exist within the free configuration space $\mathcal{C}^{free}$. The verification of this aspect is subsequently performed by the collision-checking module of the global planner.

The local planner assumes a crucial role as a subroutine in search-based algorithms. In each iteration of the algorithm, the local planner is responsible for establishing connections between the current node under expansion and its successors to construct a graph structure. Two possible techniques used by local planners to create local trajectories connecting the current node and its successors are as follows:

- *State-based steering*: involves computing a local trajectory that originates from an initial state, denoted as x_i, and efficiently guides the system towards the final state, denoted as x_f. This technique achieves an exact connection between the two states, addressing the inverse problem of precisely interpolating between x_i and x_f. This interpolation corresponds to the resolution of a TPBVP. This exact interconnection, known as *exact optimal steering*, is a requirement for most search-based algorithms that guarantee optimality.
- *Forward propagation*: generates new states by applying a feasible control input

to an already generated state over a specified time interval. This is performed through the integration of the equations of motion and avoids solving a TPBVP. From a computational point of view, this approach proves to be significantly more convenient than attempting to solve the TPBVP numerically.

Both techniques will be employed in this thesis. Specifically, *forward propagation* will be extensively elucidated in Chapter 3, providing a detailed exploration and explanation. On the other hand, Chapter 4 will focus on *state-based steering* method, exploring its implementation and complexities within the research context.

2.3.2. Heuristic function design

In search-based algorithms, the heuristic function plays a crucial role, acting as a guiding factor that significantly influences the efficiency of the search process. The heuristic function provides an estimate of the cost from a given state to the goal state, offering a heuristic value that helps the algorithm make informed decisions during the exploration of the search space. A well-designed heuristic function can greatly expedite the search process by directing the algorithm towards more promising trajectories. The effectiveness of a heuristic function lies in its ability to strike a balance between accuracy and computational efficiency. A heuristic that accurately reflects the true cost or distance to the goal provides the algorithm with valuable insights, enabling it to make more informed choices. However, it is equally important for the heuristic to be computationally feasible, to ensure that the algorithm can navigate the search space in a timely manner.

When planning in Euclidean spaces, opting for the Euclidean distance as heuristic function is straightforward, often calculated as the straight-line distance between two points. For instance, in the case of a translating rigid body on a 2D plane, the shortest distance between two points is the straight line connecting them. However, as the complexity of the state space increases, especially when involving the first and/or second order derivatives of the position, computing a meaningful heuristic becomes more challenging. Intuitively, the *cost-to-go* can be estimated using the maximum speed constraint v_{max} as follows:

$$T_{min} = w\frac{\|p_g - p\|_\infty}{v_{max}},$$

where w is a weight. This heuristic establishes a lower bound on the minimum achievable time to reach the goal position p_g from a given position p. As proven in [7], it is both admissible and consistent. The significant advantage of this minimum-time heuristic is its rapid computation, which has a minimal impact on the overall trajectory planning time. Furthermore, it accounts for velocity constraints. However, the downside is that it

evaluates the *cost-to-go* without considering control effort, making it a loose lower bound on the optimal cost. In [8], an alternative heuristic function is proposed, incorporating control effort for a more accurate *cost-to-go* estimate. The trade-off is its increased computational complexity. Nevertheless, this thesis opts for the minimum-time heuristic, as the development of a more sophisticated heuristic function lies beyond the thesis's scope.

2.3.3. Collision checking

Collision checking is a crucial aspect in search-based algorithms, particularly in the domain of motion planning. Its primary importance lies in ensuring the feasibility and safety of generated trajectories. This involves verifying whether a proposed trajectory connecting two states, $x_i \in \mathcal{S}_{free}$ and $x_f \in \mathcal{S}_{free}$, collides with obstacles in the environment, essentially determining if the trajectory remains entirely within the free state space $\mathcal{S}_{free}$. Furthermore, the collision detection module is invoked multiple times during the planning process, emphasizing the need for it to be efficient in order to prevent any degradation in the overall performance of the planning algorithm.

Throughout this thesis, numerical examples leverage an occupancy grid map as a representation of the environment (see Section 1.2.1 for further details). Collision checks are performed at multiple discrete instances along the trajectory to assess whether the trajectory is obstacle-free or if it collides with obstacles.

3 | Search-based kinodynamic planning with online motion primitives generation

When search-based methods are employed for optimal kinodynamic planning, the core challenge is to develop an efficient local planner to compute the optimal trajectory between two states. This involves computing a trajectory from an initial state, labeled x_i, to guide the system towards a goal state, denoted as x_f. While holonomic vehicles can readily establish this connection through straightforward straight-line paths, the task becomes considerably more intricate for many other dynamic systems, necessitating the solution of a TPBVP. Generating optimal trajectories for such systems inherently poses significant challenges. However, certain dynamic systems, such as double or triple integrators [6], benefit from efficient solutions that facilitate the system's transition from one state to another. Therefore, this chapter focuses on addressing these specific systems, drawing considerable inspiration from the work presented in [7]. The focus will involve exploring the generation of motion primitives using a forward-propagation approach. This method involves the generation of new states by applying a feasible control input to an already generated state over a specified time interval. This is executed through the integration of the equations of motion and avoids solving a TPBVP. From a computational point of view, this approach proves to be significantly more convenient than attempting to solve the TPBVP numerically.

Within this chapter, Section 3.1 explains the theory behind motion primitive generation utilizing the forward propagation approach. This is complemented by illustrative examples that demonstrate its practical application. Subsequently, Section 3.2 elaborates on the discrete representation of the state space achieved through motion primitives. Section 3.3 addresses kinodynamic motion planning by utilizing motion primitives. Additionally, it introduces modifications to the classical A* algorithm to handle motion primitives. Lastly, Section 3.4 proposes two examples designed to evaluate the effectiveness and performance of the proposed methodology.

3.1. Motion primitives generation

Following [8], the differential flatness property of quadrotor systems allow us to construct control inputs by utilizing 1D time-parameterized polynomial trajectories, which are independently specified for each of the three position axes. Hence, the subsequent analysis considers polynomial state trajectories denoted as $x(t) = [p(t)^T, v(t)^T, a(t)^T, j(t)^T, s(j)^T]^T$, where the position can be mathematically expressed as:

$$p(t) = c_k \frac{t^k}{k!} + c_{k-1} \frac{t^{k-1}}{(k-1)!} + ... + c_1 t + c_0 \in \mathbb{R}^3, \tag{3.1}$$

where $C = [c_0, \ldots, c_k] \in \mathbb{R}^{3\times(k+1)}$ is a polynomial trajectory parametrization. Then, velocity as well as all other derivatives can be straightforwardly obtained as its derivative. To generate the polynomial trajectories in 3.1, the following linear time-invariant system in state space form can be employed:

$$\begin{aligned} \dot{x} &= Ax + Bu, \\ \dot{x} &= \begin{bmatrix} 0 & I_3 & 0 & \dots & 0 \\ 0 & 0 & I_3 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & I_3 \\ 0 & \dots & \dots & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ I_3 \end{bmatrix} u. \end{aligned} \tag{3.2}$$

The control input $u(t)$ is confined within the set $\mathcal{U} = [-u_{\max}, u_{\max}] \subset \mathbb{R}^3$, where $u_{\max}$ denotes the maximum limit of the control input in a three-dimensional space.

The dynamic model described in Equation 3.2 is employed to create a series of motion primitives. This process involves a lattice discretization of the control input set $\mathcal{U} = [-u_{\max}, u_{\max}]$, denoted as $\mathcal{U}_\mathcal{L} = \{u_1, \ldots, u_L\} \subset \mathcal{U}$. In this context, each control vector $u_l \in \mathbb{R}^3$ defines a short-duration motion for the system. To create this discretization $\mathcal{U}_\mathcal{L}$, a practical approach involves selecting a discrete number of samples z along each axis within the permissible control input range $[0, u_{max}]$. This choice of samples results in a discretization step $du = \frac{u_{max}}{z}$, which, in turn, generates a total of $N = (2z+1)^3$ motion primitives [7]. It is worth noting that during this stage, input constraints and affine state constraints are intentionally excluded. Instead, the focus is on generating motion primitives that are feasible with respect to the system's dynamics. This process involves solving for each of the three spatial axes independently. Input and state constraints will be addressed in a later phase, subsequent to the generation of motion primitives.

Following [7], given an initial state denoted as $x_i = [p_i^T, v_i^T, a_i^T, \ldots]^T$, a motion primitive with a duration of $\tau > 0$ is generated by applying a constant control input $u(t) \equiv u_l \in \mathcal{U}_\mathcal{L}$ for $t \in [0, \tau]$, ensuring that:

$$u(t) = p^{(n)}(t) = \sum_{i=0}^{k-n} c_{i+n} \frac{t^i}{i!} \equiv u_l.$$

The constancy of the control input requires that all coefficients dependent on time must be identically zero, which means

$$c_{(n+1):k} = 0.$$

This leads to a control input that remains constant throughout the motion.

$$u_l = c_n.$$

The integration of the control function $u(t) = u_l$ from an initial condition x_i yields to the following polynomial expression for the position:

$$p(t) = u_l \frac{t^n}{n!} + \ldots + a_i \frac{t^2}{2} + v_i t + p_i.$$

Equivalently, the resulting trajectory of the linear time-invariant system in Equation 3.2 is:

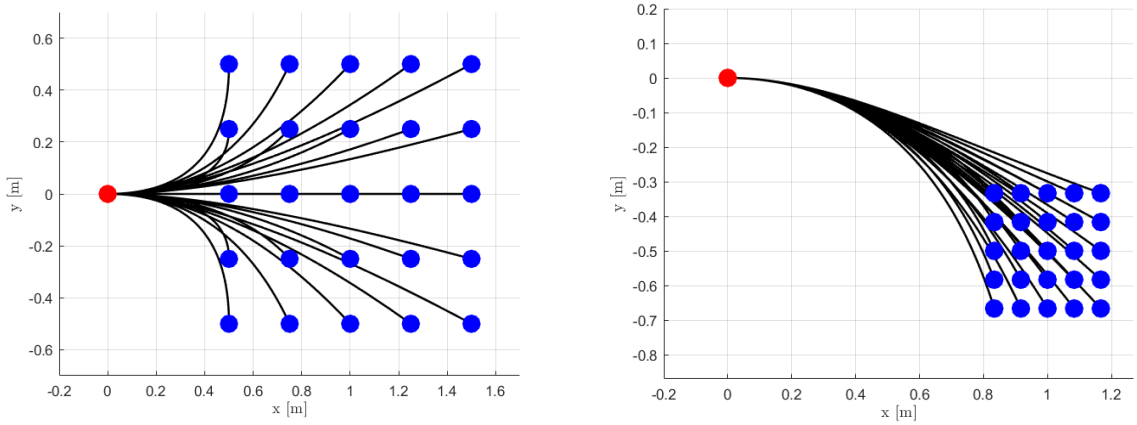
$$x(t) = e^{At} x_i + \left[\int_0^t e^{A(t-\beta)} B \, d\beta \right] u_l = F(t) x_i + G(t) u_l.$$

Figure 3.1 presents an illustration of the system trajectories that emerge from this approach. Given that both the duration τ and the control input u_l are fixed, the overall cost of the motion primitive, incorporating both the per-axis control effort and the specified duration, is:

$$J = (\|u_l\|^2 + \rho)\tau. \tag{3.3}$$

The motion primitive which connects two states x_i, x_j with $x_j = F(\tau)x_i + G(\tau)u_{ij}$ is optimal according to the cost function in Equation 3.3 [8]. The generated motion primitives adhere to the system's dynamics in Equation 3.2, but do not incorporate information regarding the system's dynamic constraints, which encompass velocity limitations and constraints on higher-order derivatives. Consequently, after the computation of the motion primitive, one must verify that the computed trajectory is feasible for the vehicle. This involves verifying that the state variables fall within the permissible limits.

In the following, some examples of practical interest are provided.



(a) Acceleration-controlled system.

(b) Jerk-controlled system.

Figure 3.1: Generate 2D motion primitives for an acceleration-controlled system (a) and a jerk-controlled system (b) starting from the initial state $(x_i, y_i) = (0, 0)$. The red dot denotes the initial state position, whereas the blue dots represent the final state positions (x_f, y_f). The black lines depict the computed trajectories resulting from various control inputs. The initial velocity and acceleration are $v_i = [1, 0]^T$ and $a_i = [0, -1]^T$(applicable to figure (b) only).

Examples

Example 1 (velocity-controlled system): Starting from an initial state, which is characterized solely by its position, $x_i = p_i^T$, a trajectory is generated by applying a constant velocity input for a duration τ, where $u(t) = v(t) \equiv u_l$. The integration of the control action from the initial condition results in the polynomial expression for the position as follows:

$$p(t) = u_l t + p_i.$$

Example 2 (acceleration-controlled system): Starting from an initial state, which is characterized by its position and velocity, $x_i = [p_i^T, v_i^T]^T$, a trajectory is generated by applying a constant acceleration input, where $u(t) = a(t) \equiv u_l$. The integration of the control action from the initial condition results in the polynomial expression for the position as follows:

$$p(t) = \frac{u_l}{2} t^2 + v_i t + p_i.$$

Example 3 (jerk-controlled system): Starting from an initial state, which is characterized by its position, velocity and acceleration, $x_i = [p_i^T, v_i^T, a_i^T]^T$, a trajectory is generated by applying a constant jerk input, where $u(t) = j(t) \equiv u_l$. The integration of the control

action from the initial condition results in the polynomial expression for the position as follows:

$$p(t) = \frac{u_l}{6}t^3 + \frac{a_i}{2}t^2 + v_i t + p_i.$$

Example 4 (snap-controlled system): Starting from an initial state, which is characterized by its position, velocity, acceleration, and jerk, $x_i = [p_i^T, v_i^T, a_i^T, j_i^T]^T$, a trajectory is generated by applying a constant snap input, where $u(t) = s(t) \equiv u_l$. The integration of the control action from the initial condition results in the polynomial expression for the position as follows:

$$p(t) = \frac{u_l}{24}t^4 + \frac{j_i}{6}t^3 + \frac{a_i}{2}t^2 + v_i t + p_i.$$

3.2. Graph construction

Motion primitives create a finite lattice discretization within the state space, providing a systematic framework to explore the various possible states and transitions. When considering all control inputs $u(t)$ within the discretized control set $\mathcal{U}_L$ and applying each of them for a duration τ to an initial state x_i, it results in the creation of $N = (2z+1)^3$ distinct motion primitives. Here, z represents the discrete number of samples selected along each axis within the permissible control input range $[0, u_{max}]$, as defined in Section 3.1. These primitives, illustrated in Figure 3.1, generate a multitude of final states, totaling $N = (2z+1)^3$. The iterative application of these control inputs leads to the gradual construction of a graph, denoted as $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Here, $\mathcal{V}$ represents the set of vertices and $\mathcal{E}$ is the set of edges connecting these vertices. Each edge within this graph represents a transition between two states, forming a structured framework that encapsulates the system's dynamics and feasible movements. A visual representation of this graph is provided in Figure 3.2

Expanding upon this discussion, it is crucial to note that the state space undergoes discretization as a result of the discretization of inputs. This discretization of control inputs leads to the creation of state lattices that are discretized not only in position but also in velocity and higher-order derivatives. However, the resulting grid in position is nonuniform in general, as demonstrated in [7]. This non-uniformity in the positional grid signifies that the discretization, particularly in position, may not adhere to a uniform or regular distribution across the state space. This irregularity could potentially influence the resolution and accuracy of the state space representation, impacting the efficacy of the motion planning algorithms utilized within this discretized space. In the next chapter, a method that avoids the issue of non-uniform discretization in the state space will be introduced.

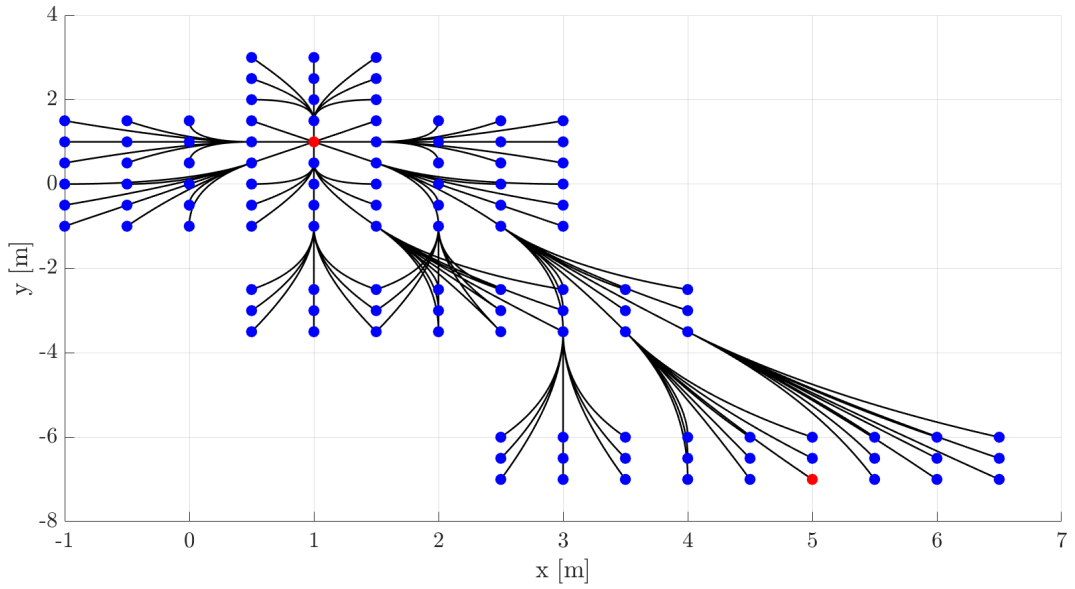


Figure 3.2: Graph generated from a starting point (red dot) located at coordinates $(x_i, y_i) = (1, 1)$ to a goal point (red dot) positioned at coordinates $(x_f, y_f) = (5, -7)$.

3.3. Kinodynamic motion planning

In the context of the quadrotor system under consideration, the effort of a trajectory can be expressed as

$$J(C) = \int_0^T \|u(t)\|^2 \, dt = \int_0^T \|p^{(n)}(t)\|^2 \, dt,$$

where $C \in \mathbb{R}^{3\times(k+1)}$ is a polynomial trajectory parametrization. Therefore, the motion planning problem arises from reformulating Problem 2.1 and incorporating the dynamics in Equation 3.2. Given an initial state $x_i \in \mathcal{S}_{free}$ and a goal region $\mathcal{S}_{goal} \subset \mathcal{S}_{free}$, the objective is to find a polynomial trajectory parametrization D and a time T such that

$$\min_{C,T} \; J(C) + \rho T \tag{3.4a}$$

$$\begin{aligned}
\text{s.t.} \quad & \dot{x}(t) = Ax(t) + Bu(t), && (3.4b)\\
& x(t) \in \mathcal{S}_{free}, \quad \forall t \in [0, T], && (3.4c)\\
& u(t) \in \mathcal{U}, \quad \forall t \in [0, T], && (3.4d)\\
& x(0) = x_i, && (3.4e)\\
& x(\tau) \in \mathcal{S}_{goal}. && (3.4f)
\end{aligned}$$

In this formulation, Equation 3.4b characterizes the system's dynamics. Conditions 3.4c and 3.4d outline the state constraints and the control input constraints. The initial and final conditions in 3.4e and 3.4f correspond to the boundary constraints.

The generation of motion primitives enables the conversion of Problem 3.4 from an optimal control problem to a graph-search problem, which can subsequently be solved using a graph-search algorithm. Based on [7], an effective method involves incorporating additional constraints that define the control input $u(t)$ as piece-wise constant within intervals of duration τ, mirroring the duration of each motion primitive. The overall trajectory duration becomes $T = N\tau$, where the integer N represents the total number of concatenated motion primitives required to form the final trajectory. Expressing this control input constraint can be achieved through the formulation:

$$u(t) = \sum_{k=0}^{N-1} u_k.$$

In this equation, each $u_k(t)$ belongs to the discretized control input set $\mathcal{U}_\mathcal{L}$ and is valid within the interval $[k\tau, (k+1)\tau]$. This constraint rigorously ensures that the control trajectory is a composite sequence derived from admissible controls found within the set $\mathcal{U}_\mathcal{L}$. Finally, Problem 3.4 can be reformulated into a graph-search problem. Within this context, the objective is to determine, given an initial state $x_i \in \mathcal{S}_{free}$, a goal region $\mathcal{S}_{goal} \in \mathcal{S}_{free}$, and a finite set of motion primitives each with a duration $\tau > 0$, the optimal sequence of control inputs u_k, $k = 0, ..., N-1$ such that:

$$\begin{aligned}
&\min_{u_k} \left(\sum_{k=0}^{N-1} \|u_k\|^2 + \rho N \right) \tau \\
\text{s.t. } & x_{k+1}(t) = F(t)x_k(\tau) + G(t)u_k, \forall\, t \in [0, \tau], \\
& x_{k+1}(t) \in \mathcal{S}_{free}, \forall\, k = 0, ..., N-1,\ t \in [0, \tau], \\
& u_k \in \mathcal{U}_L, \forall\, k = 0, ..., N-1, \\
& x_{k+1}(0) = x_k(\tau), \forall\, k = 0, ..., N-1, \\
& x_0(0) = x_i, \\
& x_N(\tau) \in \mathcal{S}_{goal}.
\end{aligned}$$

Such problem is solved via search-based motion planning algorithm.

3.3.1. A* algorithm

To solve the previously discussed graph search problem, the A* algorithm, explained in Section 1.2.1, is used. The primary adaptation of this algorithm lies within the `GetSuccessors` function, which will be explained in the next section. Additionally, a significant modification is found within the `RecoverPath` function (Algorithm 1.2). Specifically, this function is replaced by the `RecoverTraj` function (Algorithm 3.1), which is explicitly crafted to manage motion primitives while retrieving the optimal trajectory computed by the A* algorithm. It begins by initializing the trajectory as an empty set P in line 2. The subsequent steps involve the recovery of the predecessors list of node v (line 3). The core of the procedure spans lines 4 through 9: here, for each node p' within the predecessors list of node v, the corresponding index of the control action utilized to transition from node p' to node v is obtained. Subsequently, the `forwardAction` function (Algorithm 3.2) computes the primitive guiding the vehicle's transition from node p' to node v. This derived primitive segment is then added to the set P. Ultimately, the optimal trajectory is determined by concatenating all the trajectory segments present within the set P.

The `forwardAction` procedure (Algorithm 3.2) employs the `PrimitiveStateControl` function to calculate the coefficients of the motion primitive polynomial trajectory. This trajectory is obtained by applying a constant control input u_{idx} to the state v over a duration of τ. This computation applies the methodology outlined in Section 3.1, particularly leveraging the expressions provided in the examples within that section.

Algorithm 3.1 Function to recover the optimal trajectory computed by A* algorithm.

```
1: function RecoverTraj(v, Pred(v))
2: P ← ∅;
3: PredList(v) ← Pred(v)
4: for all p' ∈ PredList(v) do
5:     idx ← PredActIdx(p')
6:     pr ← forwardAction(p', idx)
7:     P ← ⟨pr,P⟩;
8:     v ← p;
9: end for
10: return P;
11: end function
```

Successor nodes

The central function within the A* algorithm is the `getSuccessors` procedure, which is outlined in Algorithm 3.3. This crucial procedure aims to propagate states and plays a

Algorithm 3.2 Function to compute the motion primitive pr derived by applying the input u_{idx} corresponding to the index idx within the lattice discretization of the control input $\mathcal{U}_L$ to the current node v for a duration τ.

1: **function** *forwardAction*(v, idx)
2: pr $\leftarrow$ *PrimitiveStateControl*(v, u_{idx}, τ);
3: **return** Pr;
4: **end function**

fundamental role in exploring the free state space while constructing the connected graph, as discussed in Section 3.2. The core of the algorithm resides in lines 5 through 16. For every control action u_l within the control input set $\mathcal{U}_L$, a motion primitive is calculated by applying the chosen control action u_l to the current node v for a duration τ. Then, this calculated trajectory is evaluated for its dynamic feasibility. If feasible, the corresponding successor node is determined and inserted into the successors list of v. Additionally, the cost of the primitive trajectory is computed and stored. On the contrary, if the computed trajectory is found to be dynamically infeasible for the vehicle, it is eliminated from the feasible motions that the vehicle can execute.

Algorithm 3.3 Given the current node v and the lattice discretization of the control input $\mathcal{U}_L$, find the set of successors Succ(v), their associated cost SuccCost(v) and their action index SuccActIdx(v).

1: **function** *GetSuccessors*(v, $\mathcal{U}_L$,τ)
2: Succ(v) $\leftarrow \emptyset$;
3: SuccCost(v) $\leftarrow \emptyset$;
4: SuccActIdx(v) $\leftarrow \emptyset$;
5: **for all** $u_l \in \mathcal{U}_L$ **do**
6: pr $\leftarrow$ *PrimitiveStateControl(v,u_l,τ)*;
7: **if** *isDynamicallyFeasible(* pr) **then**
8: $s_f \leftarrow$ pr(τ);
9: Succ(v) $\leftarrow$ Succ(v) $\cup$ $\{s_f\}$;
10: SuccCost(v) $\leftarrow$ SuccCost(v) $\cup$ $\{(\|u_l\|^2 + \rho)\tau\}$;
11: idx $\leftarrow$ *getIndex(u_l)*;
12: SuccActIdx(v) $\leftarrow$ SuccActIdx(v) $\cup$ {idx};
13: **end if**
14: **end for**
15: **return** Succ(v), SuccCost(v), SuccActIdx(v) ;
16: **endfunction**

3.4. Numerical examples

In this section, two examples of a high-level trajectory generator that uses motion primitives are presented. The primary goal is for a quadcopter to move from a predefined initial state to a target final state, with a focus on striking a trade-off between trajectory duration and effort along the trajectory. In both examples, the quadrotor can operate in a 2D $40m \times 10m$ rectangular virtual environment.

3.4.1. Acceleration-controlled system

In this example, the quadrotor is modeled as a double integrator:

$$\begin{cases} \dot{x}(t) = v_x(t) \\ \dot{y}(t) = v_y(t) \\ \dot{v}_x(t) = a_x(t) = u_x \\ \dot{v}_y(t) = a_y(t) = u_y. \end{cases}$$

Velocity and acceleration (control input) are bounded within $v_x, v_y \in [-2, 2]m/s$ and $a_x, a_y \in [-2, 2]m/s^2$, respectively. The control input set is discretized taking into account the following 9 control inputs:

$$[u_x \ u_y] = \begin{bmatrix} -2 & -2 \\ -2 & 0 \\ -2 & 2 \\ 0 & -2 \\ 0 & 0 \\ 0 & 2 \\ 2 & -2 \\ 2 & 0 \\ 2 & 2 \end{bmatrix}.$$

During each iteration of the A* algorithm, motion primitives are generated by applying each control input within this set to the current node under expansion for a duration τ. The duration of the primitives can be customized by the user and is set to a fixed value of $\tau = 1s$ in this example.

The vehicle is initialized with the state $(x, y, v_x, v_y) = (2, 1, 0, 0)$ in a simulated indoor environment, and the goal state is set to $(x, y, v_x, v_y) = (38, 7, 0, 0)$. The algorithm is developed in MATLAB, and the simulations are executed on a personal computer with

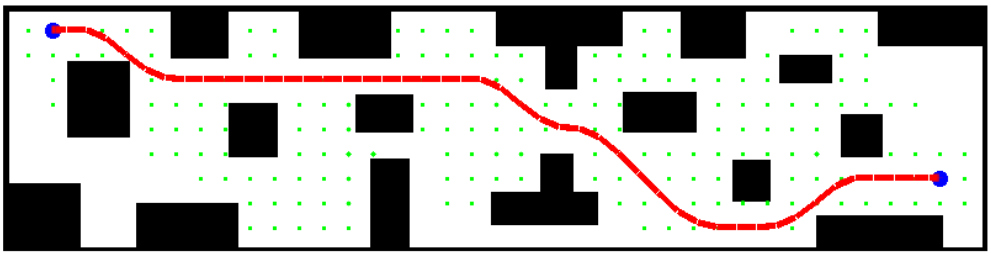


Figure 3.3: Simulation result for acceleration-controlled system. The blue dot on the right represents the starting position, whereas the one on the left indicates the final position. The red curve is the optimal trajectory. The green dots are the expanded nodes.

an IntelCore i7 processor running at 2.30 GHz and 16GB RAM. In Table 3.1 some relevant results are reported.

Table 3.1: Relevant results for the acceleration-controlled system.

Algorithm runtime	8964.06 ms
Number of expanded states	794

Figure 3.3 shows the simulation results for the acceleration-controlled system. The optimal trajectory in red shows the vehicle's ability to reach the target state navigating through the environment without any collisions with obstacles.

Figures 3.4 and 3.5 depict the velocity and actuation profiles corresponding to the optimal trajectory shown in Figure 3.3. These figures provide a clear visualization of how the trajectory satisfies both the velocity and the input constraints. Furthermore, the system model used introduces discontinuities in the acceleration profile, as observed in Figure 3.5. To address this issue, a more extensive exploration of the state space becomes necessary, as exemplified in the next example.

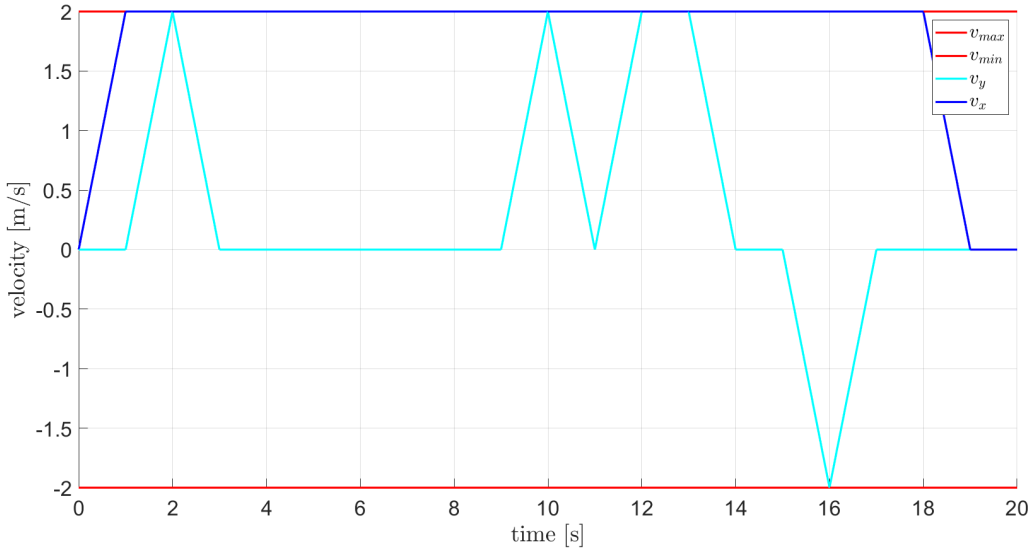


Figure 3.4: Velocity profile for the optimal trajectory in Figure 3.3. The blue and cyan lines depict the velocities along the x- and y-axes. The red lines show the velocity limits.

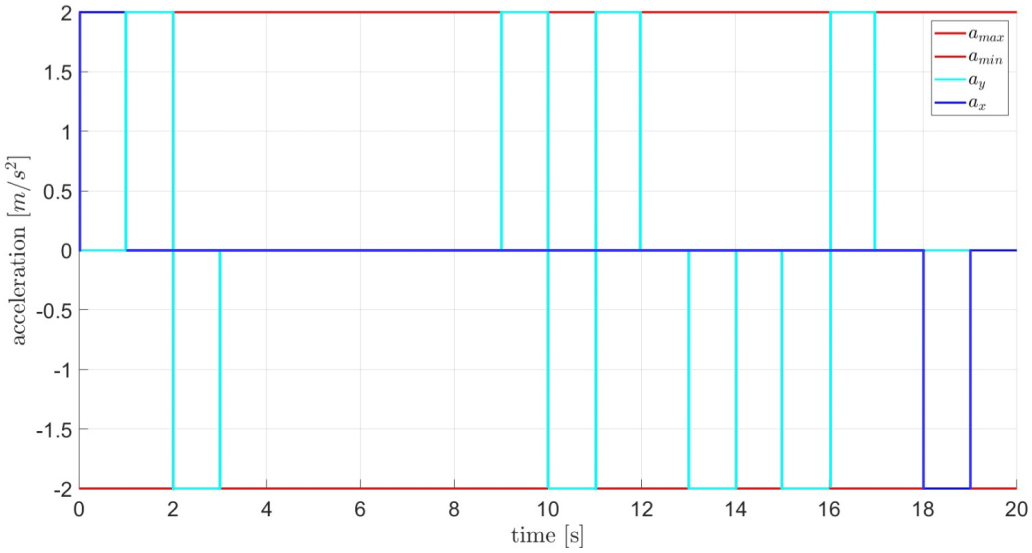


Figure 3.5: Actuation (acceleration) profile for the optimal trajectory in Figure 3.3. The blue and cyan lines depict the accelerations along the x- and y-axes. The red lines show the actuation limits.

3.4.2. Jerk-controlled system

In this example, the quadrotor is modeled as a triple integrator:

$$\begin{cases} \dot{x}(t) = v_x(t) \\ \dot{y}(t) = v_y(t) \\ \dot{v}_x(t) = a_x(t) \\ \dot{v}_y(t) = a_y(t) \\ \dot{a}_x(t) = j_x(t) = u_x \\ \dot{a}_y(t) = j_y(t) = u_y. \end{cases} \tag{3.5}$$

Velocity, acceleration, and jerk (control input) are constrained within $v_x, v_y \in [-1, 1]m/s$, $a_x, a_y \in [-1, 1]m/s^2$ and $j_x, j_y \in [-1, 1]m/s^3$, respectively. The control input set is discretized taking into account the following 9 control inputs:

$$[u_x \ u_y] = \begin{bmatrix} -1 & -1 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \\ 0 & 0 \\ 0 & 1 \\ 1 & -1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

During each iteration of the A* algorithm, motion primitives are generated by applying each control input within this set to the current node under expansion for a duration τ. The duration of the primitives can be customized by the user and is set to a fixed value of $\tau = 1s$ in this example.

The vehicle is initialized with an initial state $(x, y, v_x, v_y, a_x, a_y) = (2, 1, 0, 0, 0, 0)$, while the target state is defined as $(x, y, v_x, v_y, a_x, a_y) = (38, 7, 0, 0, 0, 0)$. The algorithm is implemented in MATLAB, and the simulations are executed on a personal computer equipped with an Intel Core i7 processor operating at 2.30 GHz and 16GB of RAM. In Table 3.2 some relevant results are reported.

Analyzing the results presented in Table 3.1 and Table 3.2, a clear increase in the algorithm's runtime becomes evident when implementing the jerk-controlled system. This substantial increase is primarily attributed to the augmented graph cardinality, directly influencing the computation time within the A* algorithm. The increase in computa-

Table 3.2: Relevant results for the jerk-controlled system.

Algorithm runtime	82438.77 ms
Number of expanded states	3080

tion time is related to the increase in the number of successor nodes to be checked at each iteration of the A* algorithm. Specifically, the number of expanded states increases significantly from 794 nodes for the acceleration-controlled system to 3080 nodes for the jerk-controlled system, showcasing the expanded search effort. Furthermore, if higher-order derivatives were included in the state space, both the algorithm's runtime and the number of expanded states would likely experience further increments. For enhanced search speed, considering code optimization techniques and transitioning to a C++ implementation could be beneficial.

Figures 3.7, 3.8 and 3.9 provide the velocity, acceleration, and actuation profiles for the optimal trajectory depicted in Figure 3.6. These figures vividly illustrate the adherence to velocity, acceleration, and input constraints.

In this particular example, the system model employed ensures a continuous acceleration profile. Additionally, compared with the previous example, the velocity profile is considerably smoother.

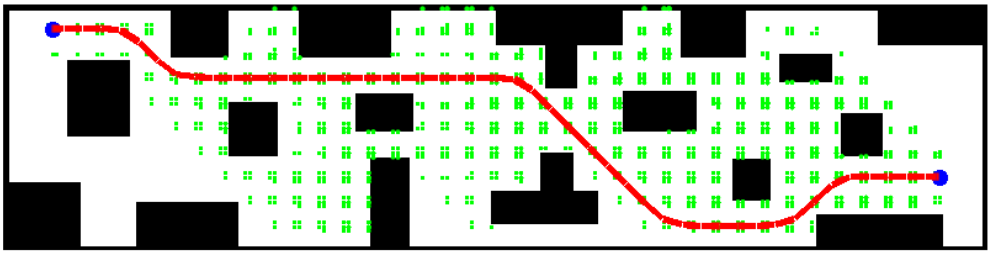


Figure 3.6: Simulation result for jerk-controlled system. The blue dot on the right represents the starting position, whereas the one on the left indicates the final position. The red curve is the optimal trajectory. The green dots are the expanded nodes.

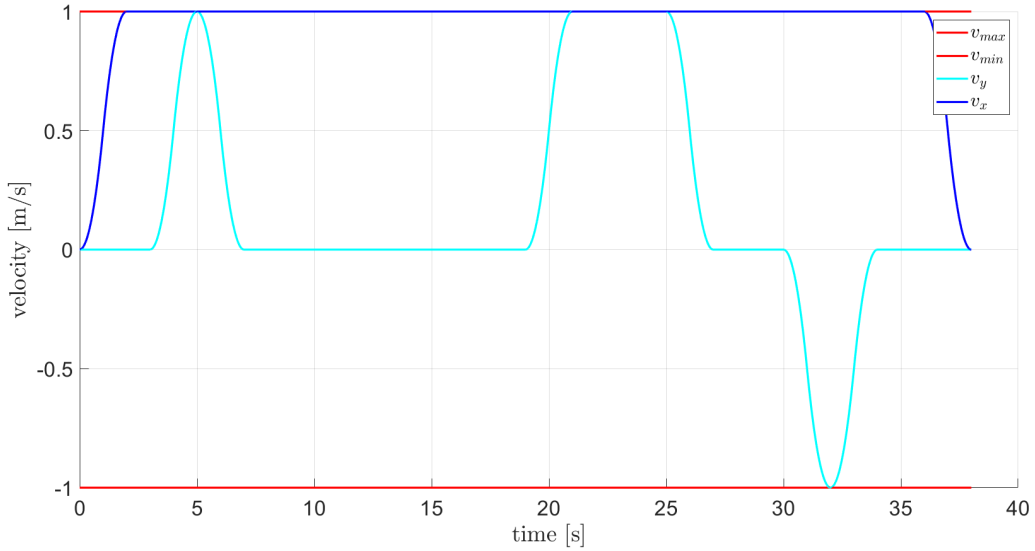


Figure 3.7: Velocity profile for the optimal trajectory in Figure 3.6. The blue and cyan lines depict the velocities along the x- and y-axes. The red lines show the velocity limits.

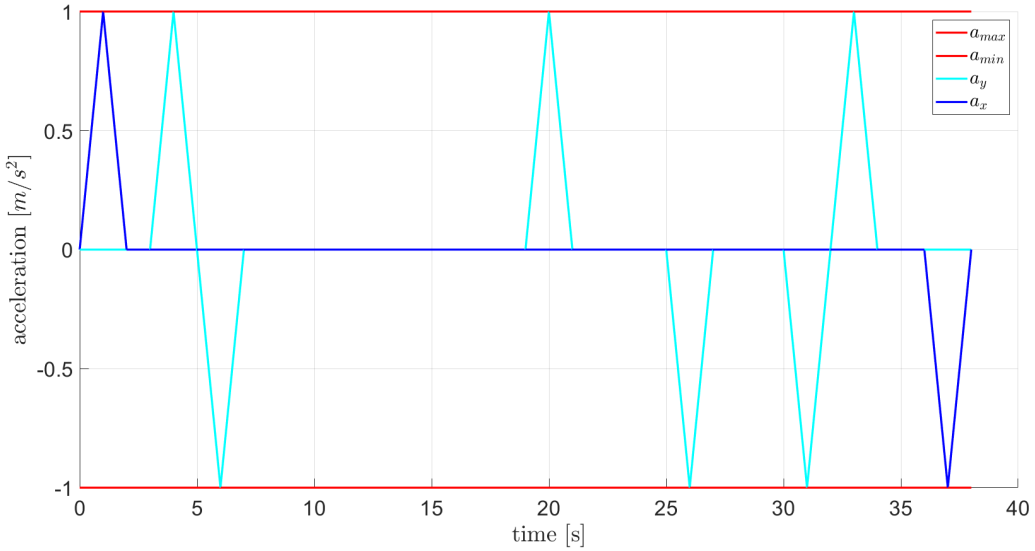


Figure 3.8: Acceleration profile for the optimal trajectory in Figure 3.6. The blue and cyan lines depict the accelerations along the x- and y-axes. The red lines show the acceleration limits.

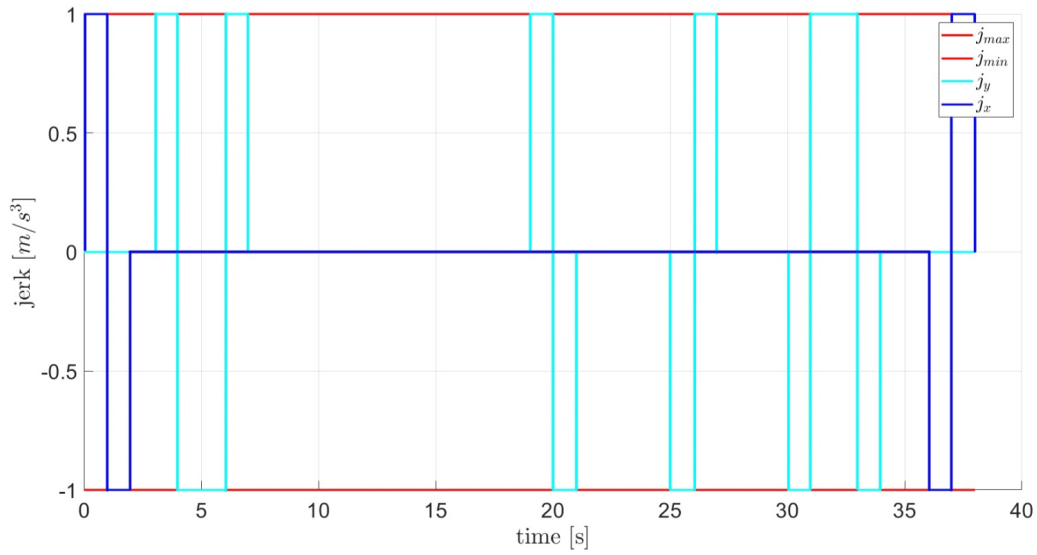


Figure 3.9: Actuation (jerk) profile for the optimal trajectory in Figure 3.6. The blue and cyan lines represent the jerks along the x- and y-axes. The red lines show the actuation limits.

3.5. Conclusion

In this chapter, a search-based planner that leverages motion primitives has been introduced as a solution explicitly tailored for quadrotors, aiming to address the complexities of the kinodynamic motion planning problem. The central focus was on generating motion primitives using a forward-propagation approach to circumvent the numerical challenges associated with solving TPBVPs.

In conclusion, the presented search-based motion planning approach demonstrates good performance in navigating quadrotor UAVs through complex and cluttered environments. In particular, this approach stands out for its ability to generate trajectories in real time, producing dynamically feasible, collision-free, resolution-optimal, and complete trajectories. However, this approach presents some limitations. The real-time computation of motion primitives at each iteration of the planner, while computationally tractable for straightforward systems with available analytical solutions, becomes intractable for more complex systems. Furthermore, the method's use of motion primitives with fixed duration hinders the overall objective of generating a minimum-time trajectory. The fixed duration of a motion primitive connecting two states remains unchanged, even if a shorter-duration trajectory compliant with dynamics and input constraints is feasible. In the next chapter, a method that efficiently addresses both of these issues will be introduced.

4 | Search-based kinodynamic planning with motion primitives library

Chapter 3 delves into the complexities of addressing kinematic motion planning problems, focusing in particular on straightforward systems such as double and triple integrators. The simplicity of these systems enables the calculation of motion primitives through forward propagation of the equations of motion, eliminating the necessity of solving a TPBVP. For these simple systems, performing real-time computations of the motion primitives within each iteration of the A* algorithm is relatively straightforward, thanks to the immediate availability of analytical solutions. However, this task becomes considerably more intricate for more complex dynamic systems. Addressing the computation of a motion primitive for these systems involves solving a TPBVP, which is non-trivial and time-consuming. Consequently, in such cases, performing real-time computations of the motion primitives during each iteration of the A* algorithm becomes intractable. The considerable time required to compute the solution slows down trajectory planning, making it unsuitable for dynamic environments where rapid and frequent replanning of the trajectory is crucial.

This chapter builds upon the methodology proposed in [15], providing an effective approach for addressing kinodynamic motion planning problems for complex systems. Specifically, Section 4.1 explains the generation of motion primitives by employing state-based steering techniques and delving into essential concepts regarding the invariance properties of the system. Additionally, it introduces the notion of a motion primitive library, accompanied by a comprehensive example illustrating the process of computing this library for a simple system. Section 4.2 delves into the intricate realm of search space design, exploring methodologies and considerations inherent to this crucial aspect of motion planning. Section 2.2 addresses kinodynamic motion planning by integrating the motion primitive library within the A* algorithm. This integration forms a pivotal part of the methodology, enhancing the algorithm's capabilities in handling complex systems. Finally, a numerical

validation is presented to show the efficacy of the proposed methodology.

4.1. Motion primitives generation

This section explores the generation of motion primitives through the application of state-based steering technique, drawing considerable inspiration from the work presented in [16]. More specifically, it involves computing a trajectory that originates from an initial state, denoted as x_i, and efficiently guides the system towards the final state, denoted as x_f, by establishing an exact connection between the two states. State-based steering effectively tackles the inverse problem of precisely interpolating between states x_i and x_f, which corresponds to the resolution of a TPBVP spanning from the initial state to the final state. This exact interconnection between the two states, known as *exact optimal steering*, guarantees the optimality of the algorithm. On the contrary, the approach described in Chapter 3 aims to achieve optimality without solving a TPBVP. Instead, it computes optimal trajectories analytically. Although motion primitives can be analytically computed in many straightforward scenarios, such as systems with linear dynamics and quadratic costs, including several aerospace applications such as double or triple integrators [6], traditional optimal control methods such as the Pontryagin minimum principle [7], [13] become computationally infeasible for solving more complex problems. Consequently, in the case of more complex systems, the resolution of a TPBVP becomes a required step.

The primitive motion that originates in the initial state x_i and arrives at the final state x_f is the solution to the following constrained optimal control problem. This problem focuses on determining the optimal duration τ for the trajectory, along with the corresponding optimal control input $u(\cdot)$.

$$\min_{u(\cdot),\tau} J = \int_0^\tau \gamma(x(t), u(t))dt \tag{4.1a}$$

$$\text{s.t. } \dot{x}(t) = f(x(t), u(t)), \tag{4.1b}$$

$$x(t) \in \mathcal{S}_{free} \text{ for } t \in [0, \tau], \tag{4.1c}$$

$$u(t) \in \mathcal{U} \text{ for } t \in [0, \tau], \tag{4.1d}$$

$$x(0) = x_i,\ x(\tau) = x_f, \tag{4.1e}$$

where J represents the cost function that needs to be minimized, typically consisting of both a boundary term and an integral term. The boundary term typically pertains to the trajectory's duration, while the integral term includes the control input, in order to

penalize excessive control efforts. The search for an optimal solution involves ensuring the satisfaction of various equality and inequality constraints. In particular, Equation 4.1b characterizes the dynamics of the system. The condition 4.1d outlines the control input constraints, where $u(t)$ represents the control inputs vector, and $\mathcal{U}$ denotes the admissible set of control inputs. The condition 4.1c describes the state constraints, which include restrictions on velocity and higher-order derivatives. In this context, $x(t)$ denotes the system's state vector. The initial and final conditions in 4.1e correspond to the boundary constraints.

Typically, the decision variables $(x(t), u(t))$ are continuous functions, posing a challenge in solving Problem 4.1. However, the differential flatness of the quadrotor system allows to rewrite Problem 4.1 using a finite parametrization. Subsequently, the problem can be reformulated as a nonlinear program and solved using NLP techniques. In this way, optimization over states and control inputs is avoided. This process will be better explained in the example provided in Section 4.1.3.

4.1.1. Invariance properties

Dynamic systems exhibit a fundamental characteristic: translation invariance. This property is a common feature in autonomous vehicle systems and has been extensively explored in previous works [5], [1], [15]. This property implies that if the dynamics of the system 1.3 and the initial conditions are expressed in a new set of coordinates obtained by a simple translation of the original one, the state evolution resulting from the application of a control input $u(\cdot)$ is a translated version of the original coordinates [16]. In other words, the use of the Euclidean norm in the cost function 2.2 ensures invariance under translation. As a consequence, the optimal primitive for a specific problem remains the same, regardless of the coordinate frame in which the problem is formulated [13]. Furthermore, the satisfaction of algebraic path constraints and the value of the cost function remain unaffected by the translation of the coordinate system. This provides the designer with the flexibility to formulate the motion primitive generation problem in the most suitable reference frame.

In specific instances, certain dynamic systems exhibit more robust invariance properties. Within such systems, it becomes possible to derive optimal solutions for two distinct sets of boundary conditions from one another. This capability arises when the initial and final configuration pairs are connected through an affine transformation that conserves distances. This extension of the invariance property goes beyond mere translation, including rotation and mirror reflection. As a result, rotated and symmetric motion primitives

share the same cost and satisfy identical algebraic path constraints [16]. For example, quadrotors exhibit the characteristic of being invariant to horizontal plane translations as well as rotations about a vertical axis [4]. This implies that all trajectories can be rigidly translated to start from any arbitrary point and rotated about the vertical axis to initiate with an arbitrary yaw angle. Furthermore, quadrotor motion primitives corresponding to boundary conditions that are symmetric with respect to both the x-axis and the y-axis are symmetric.

An issue in the offline trajectory generation process is the large storage memory required. In this context, the properties mentioned above can be leveraged to reduce the required storage memory. For the remaining part of this work, the essential properties of translational invariance and symmetry of motion primitives will serve for the assembly of motion primitive sequences into a complete trajectory.

4.1.2. Motion primitives library

Solving a TPBVP for a generic nonlinear system is a challenging task that requires the use of a nonlinear solver. Consequently, the online computation of motion primitives at each iteration of the planning algorithm becomes computationally intractable. This issue arises from the intention to apply the kinodynamic motion planning procedure in real-time scenarios, particularly in dynamic environments where vehicles must rapidly replan their trajectories. To enable real-time application, the algorithm must achieve exceptionally short runtimes, and the heavy computational load resulting from solving numerous TPBVPs precludes this possibility. In addressing this issue, a practical solution has been presented in [15] and [1], which introduces a library of motion primitives. One notable feature of this innovative approach is that transcribing the system dynamics into a motion library shifts all challenges related to solving nonconvex optimization problems, handling model nonlinearities, ensuring constraint satisfaction, and more, to the offline phase. In this phase, the use of iterative and computationally intensive algorithms does not pose problems. The remainder of this chapter adopts this efficient approach, drawing substantial inspiration from [16].

In this study, the database of motion primitives is generated offline by solving Problem 4.1 numerically for a suitable number of boundary conditions, obtained by uniformly gridding the continuous state space. In particular, for every pair of initial and final states (x_i^i, x_f^i), $i = 1, ..., n$, taken as the grid points of the uniform discretized grid, a TPBVP is solved. This process yields the optimal time history of the vehicle states $x_i^*(t)$, the optimal control input $u_i^*(t)$, the optimal trajectory duration τ_i^*, and the optimal cost C_i^*. All of these

elements minimize the cost function J and satisfy all constraints.

The resulting quantities for each trajectory are stored in a data structure that can be conveniently organized as a Look-Up Table (LUT). This collection of optimal trajectories is subsequently employed repetitively online; when the planner requires an edge connecting two nodes, it can simply select a suitable trajectory from the library. This approach transfers the generation of motion primitives to the offline phase, eliminating the requirement to repeatedly invoke a non-linear solver during the online planning stage. It is worth noting that in cases where a vehicle model is not available, trajectories can be obtained directly from experimental flight data [4]. Therefore, primitives are by design compatible with the vehicle dynamics.

In the following example, we will investigate the practical application of this method to quadrotors, leveraging their ready availability for experimental testing within the Aerospace Systems and Control Laboratory (ASCL) at Politecnico di Milano. However, it is essential to recognize that employing this method with such a straightforward system does not yield significant advantages compared to the approach outlined in Chapter 3. In this scenario, selecting the trajectory required by the planner from the library is essentially equivalent to computing the analytical solution in real time. Nevertheless, it is worth mentioning that this method offers wide versatility and can be employed with a broad range of dynamical systems governed by differential equations and subject to analytical constraints, provided that the design of edges can be formulated as a TPBVP.

4.1.3. Example

Consider a planning scenario that involves a quadrotor UAV operating in a 2D configuration space. The system is characterized by a 4D state space (x, y, v_x, v_y), including position (x, y) and velocity (v_x, v_y) along the x and y axes,in addition to a 2D actuation space $(a_x, a_y) = (u_x, u_y)$, constituted by the linear accelerations along the respective axes. The quadrotor's dynamics is described by the following set of equations:

$$\begin{cases} \dot{x}(t) = v_x(t) \\ \dot{y}(t) = v_y(t) \\ \dot{v}_x(t) = a_x(t) = u_x \\ \dot{v}_y(t) = a_y(t) = u_y. \end{cases}$$

The library of motion primitives is computed by solving the following TPBVP for every combination of initial state $(x_i, y_i, v_{xi}, v_{yi})$ and final state $(x_f, y_f, v_{xf}, v_{yf})$.

$$\min_{u_x(\cdot),u_y(\cdot),\tau} J = \int_0^\tau (u_x(t)^2 + u_y(t)^2)dt + \rho\tau \tag{4.2}$$

$$\begin{aligned}
\text{s.t. } & \dot{x}(t) = v_x(t), \\
& \dot{y}(t) = v_y(t), \\
& \dot{v}_x(t) = a_x(t) = u_x, \\
& \dot{v}_y(t) = a_y(t) = u_y, \\
& v_x(t) \in [-1, 1] \text{ for } t \in [0, \tau], \\
& v_y(t) \in [-1, 1] \text{ for } t \in [0, \tau], \\
& u_x(t) \in [-3, 3] \text{ for } t \in [0, \tau], \\
& u_y(t) \in [-3, 3] \text{ for } t \in [0, \tau], \\
& x(0) = x_i,\ y(0) = y_i, \\
& v_x(0) = v_{xi},\ v_y(0) = v_{yi}, \\
& x(\tau) = x_f,\ y(\tau) = y_f, \\
& v_x(\tau) = v_{xf},\ v_y(\tau) = v_{yf},
\end{aligned} \tag{4.3}$$

where ρ is a weight on the time that can be adjusted by the user to suit the specific problem at hand.

Leveraging differential flatness, a polynomial parametrization of the flat output is assumed. In particular, a third-order polynomial is used to express the positions along both the x and y axes:

$$x(t) = \frac{C_1}{6}t^3 + \frac{C_2}{2}t^2 + C_3 t + C_4,$$

$$y(t) = \frac{B_1}{6}t^3 + \frac{B_2}{2}t^2 + B_3 t + B_4.$$

The velocities along both axes are derived from the respective positions as their first derivatives:

$$v_x(t) = \frac{C_1}{2}t^2 + C_2 t + C_3,$$

$$v_y(t) = \frac{B_1}{2}t^2 + B_2 t + B_3.$$

In a similar manner, the accelerations (control inputs) are the second derivatives of the positions:

$$a_x(t) = u_x(t) = C_1 t + C_2,$$

$$a_y(t) = u_y(t) = B_1 t + B_2.$$

As a result of the polynomial representation, the optimization variables for the optimal trajectory along the x-axis are represented by coefficients C_1, C_2, C_3, and C_4 while for the trajectory along the y-axis, they are represented by coefficients B_1, B_2, B_3, and B_4. Additionally, the duration of the primitive τ is a shared optimization variable for both one-dimensional trajectories, as they are required to satisfy the final boundary conditions simultaneously. This allows the cost function in 4.2 to be reformulated in terms of the optimization variables:

$$J = \int_0^\tau [(C_1 t + C_2)^2 + (B_1 t + B_2)^2] dt + \rho\tau. \tag{4.4}$$

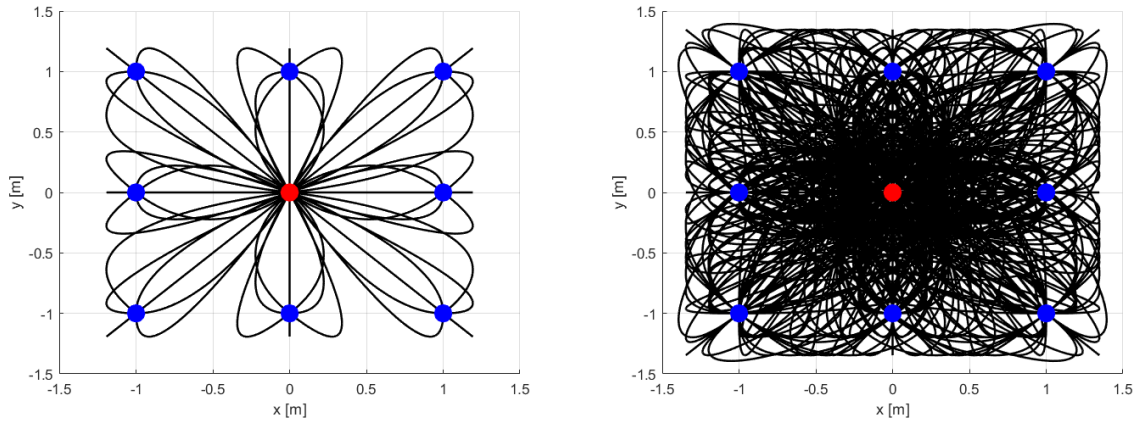
Likewise, the constraints can be expressed as functions of the optimization variables. Finally, the constrained optimization problem in 4.3 can be discretized in terms of the selected optimization variables as follows:

$$\begin{aligned}
\underset{C_i, B_i, i=1:4}{\text{minimize}} \; & J = \int_0^\tau [(C_1 t + C_2)^2 + (B_1 t + B_2)^2] dt + \rho\tau \\
\text{s.t. } & x(t) = \frac{C_1}{6} t^3 + \frac{C_2}{2} t^2 + C_3 t + C_4, \\
& y(t) = \frac{B_1}{6} t^3 + \frac{B_2}{2} t^2 + B_3 t + B_4, \\
& v_x(t) \in [-1, 1] \text{ for } t \in [0, \tau], \\
& v_y(t) \in [-1, 1] \text{ for } t \in [0, \tau], \\
& u_x(t) \in [-3, 3] \text{ for } t \in [0, \tau], \\
& u_y(t) \in [-3, 3] \text{ for } t \in [0, \tau], \\
& x(0) = x_i, \; y(0) = y_i, \\
& v_x(0) = v_{xi}, \; v_y(0) = v_{yi}, \\
& x(\tau) = x_f, \; y(\tau) = y_f, \\
& v_x(\tau) = v_{xf}, \; v_y(\tau) = v_{yf}.
\end{aligned} \tag{4.5}$$

Consequently, optimization over states and control inputs is avoided. Subsequently, the discretized problem in 4.5 is solved using the built-in MATLAB function `fmincon`.

Then, the database is constructed starting from an initial state position at $(x_i, y_i) = (0, 0)$. It relies on a low-resolution uniform square grid, where $(x_f, y_f) \in [-1, 0) \cup (0, 1] \times [-1, 0) \cup (0, 1]$, and each cell has a size of one meter. The choices for both initial and final velocities, v_x and v_y, are made from a set of three values: $\{-1, 0, 1\} m/s$.

Figure 4.1a shows a subset of motion primitives. These primitives are distinguished by tra-



(a) Subset of motion primitives.

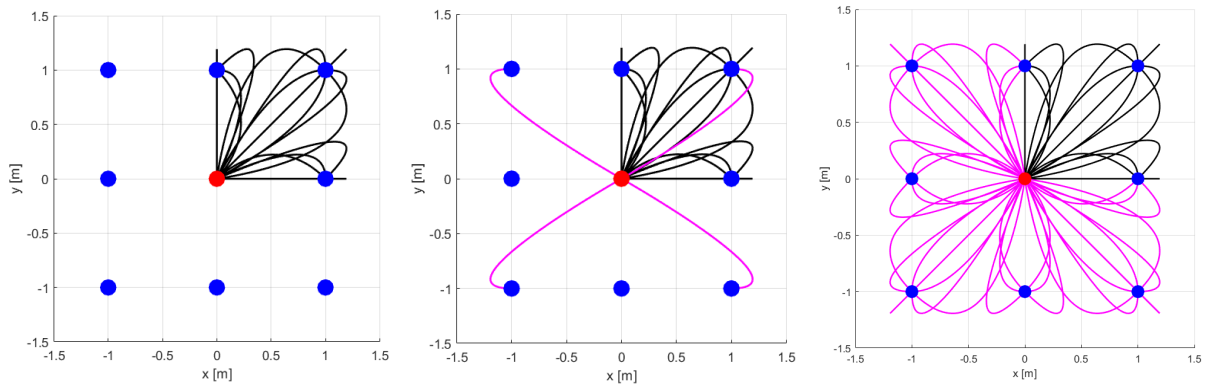
(b) Complete set of motion primitives.

Figure 4.1: Subset (a) and complete set (b) of motion primitives computed for a 4D state space (x, y, v_x, v_y). Red dot corresponds to the initial state position (x_i, y_i), while blue dots correspond to the final state positions (x_f, y_f). The black lines represent the computed trajectories for different final velocities v_x and v_y.

jectories that originate from the steady state, defined as $(x_i, y_i) = (0, 0)$, $(v_{xi}, v_{yi}) = (0, 0)$. Figure 4.1b illustrates the complete collection of motion primitives, with trajectories originating from the position $(x_i, y_i) = (0, 0)$ and velocities selected from the three available values of $\{-1, 0, 1\} m/s$.

Thanks to the property of translation invariance introduced previously (4.1.1), it is possible to maintain a small database size while effectively covering the entire space in which the vehicle operates. A key consequence of the system dynamics' invariance is the ability to treat all trajectory primitives as equivalent classes and select a prototype for each one, starting at a reference position. In practice, without any loss of generality, the initial position can be set as $(\hat{x}_i, \hat{y}_i) = (0, 0)$ during the database construction phase. Subsequently, the motion primitives can be easily translated to match any other initial position (x_i, y_i) simply by recentering the coordinate system around (x_i, y_i) [16].

Upon observation of Figure 4.1, it becomes evident that, within the context of the straightforward dynamic system at hand, a symmetry property holds with respect to both the x-axis and the y-axis. More precisely, it is necessary to keep only the trajectories located within the first quadrant in the database, as all other trajectories can be generated through straightforward mirroring relative to the x-axis and y-axis. Figure 4.2 illustrates the steps involved in generating a complete database starting from a smaller set of reference motion primitives. Figure 4.2a shows the collection of reference trajectories, which pertain to the first quadrant. In Figure 4.2b, a single reference trajectory (depicted by



(a) (b) (c)

Figure 4.2: Steps to build a complete motion primitives database (c) starting from a smaller set of reference trajectories (a) through mirror reflection relative to the x and y axes for all the trajectories within the reference set (b).

the magenta line) is subject to mirroring operations along the x and y axes. Figure 4.2c presents the complete set of primitives resulting from these mirroring operations. It is worth noting that to simplify the representation, only primitives originating from the steady state $x_i = y_i = v_{xi} = v_{yi} = 0$ are considered. To generate the entire database, all velocities within the set $\{-1, 0, 1\}m/s$ must be taken into account for the initial state.

A close examination of Figure 4.2 reveals a rotational invariance in the trajectories. This property derives from the fact that quadrotors exhibit an inherent invariance to horizontal plane translations and rotations about a vertical axis. By taking this property into account, it becomes possible to further reduce the size of the database.

4.2. Search space design

Motion primitives establish a finite lattice discretization within the state space. These predefined motions encapsulate a range of possible actions or trajectories that a system can undertake, enabling efficient and rapid planning by leveraging a precomputed set of feasible maneuvers. In the realm of search-based planning, leveraging motion primitives means making sure the planner's search space matches well with the precalculated motions stored in the database. To maximize the utility of this precomputed database, the search space within the planner must be carefully designed. It is imperative that this space aligns with the region where the motion primitives are generated. To facilitate connectivity between nodes during planning iterations, the planner's search space

is uniformly gridded based on the constructed motion primitives' region. This strategic alignment ensures that when the planner needs to establish connections between different nodes, it can efficiently access the corresponding optimal trajectories within its predefined search space. One crucial aspect often considered in this context is the trade-off between granularity and computational complexity. A finer grid within the search space can offer increased accuracy and precision in motion planning, but comes at the expense of higher computational demands due to an expanded number of grid cells and increased connectivity between nodes. On the contrary, a coarser grid reduces computational load but might sacrifice planning accuracy by limiting the diversity and precision of available trajectories.

4.3. Kinodynamic motion planning

In the domain of search-based kinodynamic motion planning, this thesis represents a significant breakthrough by integrating a library of precomputed motion primitives directly into the conventional A* algorithm. This integration eliminates the necessity of solving complex and time-consuming TPBVPs in real time during the connection of nodes in the planning process. This integration allows the planner to access a precomputed database of feasible and optimized motion primitives representing various permissible trajectories. Rather than recalculating these trajectories during each node connection, the planner efficiently retrieves and utilizes these pregenerated motion primitives. This method significantly improves the computational efficiency of the planning process while preserving the ability to generate high-quality trajectories.

4.3.1. A* algorithm

To address the kinodynamic motion planning problem, the proposed methodology is based on the use of the A* algorithm, detailed in Section 1.2.1. The primary adaptation of this algorithm lies within the `GetSuccessors` function, which will be explained in the next section. Additionally, a significant modification is introduced within the `RecoverPath` function (Algorithm 1.2). Specifically, this function is replaced by the `RecoverTrajLibrary` function (Algorithm 4.1), which is specifically crafted to manage the motion primitives library while retrieving the optimal trajectory computed by the A* algorithm. The process starts by initializing an empty set P for the trajectory at line 2. Following this, the procedure involves retrieving the predecessors list of node v at line 3. The main operation unfolds from lines 4 to line 8: for each node p' within v's predecessors list, the function extracts the trajectory allowing the transition from node p' to node v from the motion

primitive database. Subsequently, this resulting primitive segment is incorporated into the set P. Ultimately, the optimal trajectory is formed by combining all segments of the trajectory contained within the set P.

Algorithm 4.1 Function to recover the optimal trajectory computed by A* algorithm.

```
1: function RecoverTrajLibrary(v, Pred(v))
2: P ← ∅;
3: PredList(v) ← Pred(v)
4: for all p' ∈ PredList(v) do
5:     pr ← L(p', v);
6:     P ← ⟨pr,P⟩;
7:     v ← p;
8: end for
9: return P;
10: end function
```

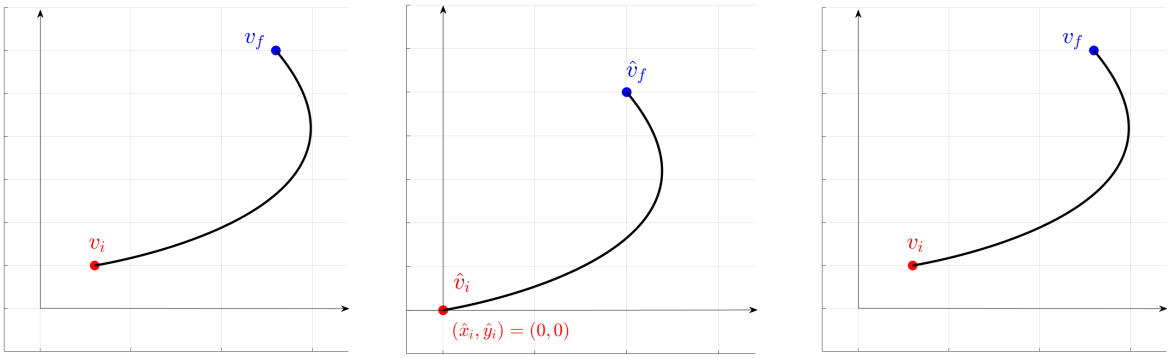
Successor nodes

Successor nodes consist of a collection of predefined trajectories that connect initial states to defined final states within a meticulously defined grid. The motion primitives database simplifies the process of determining a node's successors by simplifying the computation of the required steering action during online motion planning. Instead of solving TPBVPs in real-time, this process involves searching for a motion primitive within a pre-computed Look-Up Table (LUT), resulting in a substantial acceleration of the motion planning solution.

When provided with an initial state and a final state, the `GetSuccessorsLibrary` procedure in 4.2 is employed to query the motion primitives database. This process involves a sequence of geometric transformations, as depicted in Figure 4.3. Specifically, for each edge connecting two nodes, v_i and v_f, the respective trajectory and its associated cost are recovered using the following approach: First, the original pair of initial and final states (v_i, v_f) (Figure 4.3 a) are translated so that the resulting initial state $\hat{v}_i$ has the position corresponding to the origin of the motion primitive library $(\hat{x}_i, \hat{y}_i) = (0, 0)$ (Figure 4.3b). Subsequently, a database query is executed to retrieve the trajectory connecting these transformed nodes, along with the associated cost. Finally, to recover the trajectory linking the actual pair of boundary values, v_i and v_f, the inverse of the initial translation is applied. Therefore, the edge that connects the initial required boundary conditions is determined (Figure 4.3c).

Throughout this process, it is important to maintain trajectory continuity. Specifically, the selected primitive must ensure the continuity of all state variables, encompassing not

only position but also velocity, and, where applicable, higher-order derivatives, at each node. This attention to continuity is crucial to achieve a smooth and reliable trajectory.



(a) original nodes. (b) translated nodes. (c) inverse translation.

Figure 4.3: Steps involved in the geometric translation.

Algorithm 4.2 Given the current node v and the motion primitive library $\mathcal{L}$, including all initial states s_i, final states s_f, connecting primitives pr, and their associated effort costs c, find the set of successors $\text{Succ}(v)$ and their cost $\text{SuccCost}(v)$.

function *GetSuccessorsLibrary*(v, $\mathcal{L}$)
$\text{Succ}(v) \leftarrow \emptyset$;
$\text{SuccCost}(v) \leftarrow \emptyset$;
for all $s_i, s_f \in \mathcal{L}$ **do**
  $\mathcal{L}$.*isContinuous*(v,s_i $\mathcal{L}$);
  $\mathcal{L}$.*Translate*(v,s_i,s_f $\mathcal{L}$);
end for
for all pr $\in \mathcal{L}$ **do**
  if *isCollisionFree*(pr) **then**
    $\text{Succ}(v) \leftarrow \text{Succ}(v) \cup \{s_f\}$;
    $\text{SuccCost}(v) \leftarrow \text{SuccCost}(v) \cup \{c + \rho\tau\}$;
  end if
end for
return $\text{Succ}(v)$, $\text{SuccCost}(v)$;
end function

4.4. Numerical examples

This section proposes two examples from a virtual environment to demonstrate the effectiveness of the described methodology. We shall explore the practical application of this approach to quadrotors in the following cases, making use of the fact that they are available for experimental tests in the Aerospace Systems and Control Laboratory (ASCL) at Politecnico di Milano University. However, it is important to note that this approach is highly adaptable and may be used for a variety of dynamical systems governed by differential equations and subject to analytical constraints, as long as the edge design can be expressed as TPBVP.

In the following examples, the goal is to navigate a quadrotor from an initial state to a final state, with an emphasis on achieving a balance between trajectory duration and the effort expended along the trajectory. In each example, the quadrotor can operate in a 2D rectangular virtual environment measuring 40m $\times$ 10m.

4.4.1. Acceleration-controlled system

Consider a planning scenario involving a quadrotor UAV operating in a 2D configuration space. The system is characterized by a 4D state space (x, y, v_x, v_y), including position (x, y) and velocity (v_x, v_y) along the x and y axes, in addition to a 2D actuation space $(u_x, u_y) = (a_x, a_y)$, constituted by the linear accelerations along the respective axes. The quadrotor's dynamics can be represented by the following set of equations:

$$\begin{cases} \dot{x}(t) = v_x(t) \\ \dot{y}(t) = v_y(t) \\ \dot{v}_x(t) = a_x(t) = u_x \\ \dot{v}_y(t) = a_y(t) = u_y. \end{cases}$$

Leveraging quadrotor differential flatness, a polynomial parametrization of the flat output is assumed. In particular, a third-order polynomial is used to express the positions along the x and y axes.

$$x(t) = \frac{C_1}{6}t^3 + \frac{C_2}{2}t^2 + C_3 t + C_4,$$

$$y(t) = \frac{B_1}{6}t^3 + \frac{B_2}{2}t^2 + B_3 t + B_4.$$

The velocities and accelerations along both axes are straightforwardly obtained by taking the first and second derivatives of the respective positions. Following the methodology described in the example in Section 4.1.3, motion primitives are obtained solving the

following discretized problem:

$$
\begin{aligned}
&\underset{C_i,B_i,i=1:4}{\text{minimize}}\ J = \int_0^\tau \|u(t)\|^2\, dt + \rho\tau \\
\text{s.t. } & x(t) = \frac{C_1}{6}t^3 + \frac{C_2}{2}t^2 + C_3 t + C_4, \\
& y(t) = \frac{B_1}{6}t^3 + \frac{B_2}{2}t^2 + B_3 t + B_4, \\
& \|v(t)\| \in [-1.5\sqrt{2}, 1.5\sqrt{2}],\ t \in [0,\tau], \\
& \|a(t)\| \in [-4.5\sqrt{2}, 4.5\sqrt{2}],\ t \in [0,\tau], \\
& x(0) = x_i,\ y(0) = y_i, \\
& v_x(0) = v_{xi},\ v_y(0) = v_{yi}, \\
& x(\tau) = x_f,\ y(\tau) = y_f, \\
& v_x(\tau) = v_{xf},\ v_y(\tau) = v_{yf}.
\end{aligned}
\tag{4.6}
$$

where the weight on the time is chosen as $\rho = 10$, in order to obtain a faster trajectory, while simultaneously minimizing overall control effort. To enhance the realism of the example, coupled constraints for velocities and accelerations are introduced. Specifically, limits are defined for the total velocity $\|v\| = \sqrt{v_x^2 + v_y^2}$ and the total acceleration (control input) $\|u\| = \|a\| = \sqrt{a_x^2 + a_y^2}$. The maximum overall velocity and acceleration are set as $v_{max} = 1.5\sqrt{2} m/s$ and $a_{max} = 4.5\sqrt{2} m/s^2$, respectively.

To solve the discretized optimization Problem in 4.6, the built-in MATLAB function `fmincon` [10] is employed as a nonlinear programming solver. Among the available solvers, the Sequential Quadratic Programming algorithm (SQP) is selected.

The database is assembled starting from an initial state position at $(x_i, y_i) = (0, 0)$. It relies on a uniform square grid, where $(x_f, y_f) \in [-3, 0) \cup (0, 3] \times [-3, 0) \cup (0, 3]$, and each cell has a size of one meter. The choices for both initial and final velocities are made from a set of three values: $\{-1.5, 0, 1.5\} m/s$. Then, motion primitives are calculated for every combination of initial state $(x_i, y_i, v_{xi}, v_{yi})$ and final state $(x_f, y_f, v_{xf}, v_{yf})$ by solving Problem 4.6. By leveraging the symmetry property, generating the reference trajectories (equivalent to 1/4 of the complete database) requires approximately 21 minutes. Fortunately, this timeframe poses no issue for the motion planning problem, as the resolution of the TPBVPs is conducted offline. The complete database comprises a total of 1701 motion primitives. The numerical details concerning the generation of the database are presented in Table 4.1.

In the virtual indoor environment, the vehicle is initialized with the state $(x, y, v_x, v_y) =$

Table 4.1: Database details for the acceleration-controlled system.

Generation time	$\approx$ 21 min
Number of primitives	1701

$(2, 1, 0, 0)$, and the destination is set at the state $(x, y, v_x, v_y) = (38, 7, 0, 0)$. Simulations are executed on a personal computer equipped with an IntelCore i7 processor running at 2.30 GHz with 16GB RAM, and the algorithm is implemented in MATLAB. In Table 4.2 some relevant results related to motion planning simulations are reported.

Table 4.2: Motion planning results for the acceleration-controlled system.

Algorithm runtime	1410.60 ms
Number of expanded states	11

When comparing Table 3.1 and Table 4.2, an interesting observation emerges: the algorithm's runtime exhibits a reduction for the current problem. This reduction comes primarily from the database of motion primitives employed, which includes trajectories spanning 1, 2, or 3 meters in length, exceeding the length of trajectories generated within the analogous example discussed in Chapter 3. Consequently, this results in a reduced number of segments that must be concatenated to obtain the overall trajectory. As a consequence, the algorithm requires fewer iterations and expands a lower number of nodes to compute the optimal trajectory. These insights highlight the crucial role of discretization within the library, which includes position, velocity, and higher-order derivatives, in influencing the performance of the algorithm. The discretization is closely tied to the particular problem at hand, requiring thoughtful consideration and tailored selection. Moreover, as the grid resolution increases, augmenting the precision of the state space, a corresponding increase occurs in the state space's cardinality. This tends to decelerate the search process, highlighting the trade-off between precision and computational efficiency.

Figure 4.4 shows the simulation result for the acceleration-controlled system. The optimal trajectory in red shows the vehicle's ability to reach the target state navigating through the environment without any collisions with obstacles. Figures 4.5 and 4.6 illustrate the velocity and actuation profiles for the optimal trajectory presented in Figure 4.4, providing a clear demonstration of the satisfaction of the velocity and input constraints. It is worth noting that the velocity profile displays jerky acceleration behavior. This is a result of the velocity at each node being constrained to precisely match one of the values in the database. This observation underscores the importance of carefully selecting the

velocity discretization step when aiming for a smoother velocity profile. Furthermore, in this example, continuity has been enforced only on the velocity components, resulting in a discontinuous acceleration profile. To address this issue, a more extensive exploration of the state space is required, as exemplified in the subsequent example.

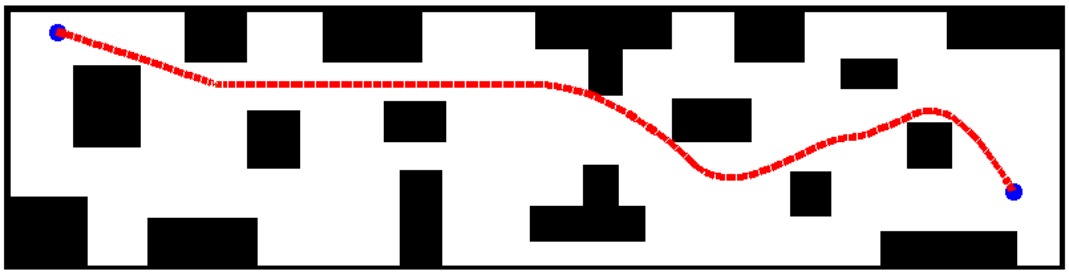


Figure 4.4: Simulation result for acceleration-controlled system. The blue dot on the right represents the starting position, while the one on the left indicates the final position. The red curve is the optimal trajectory.

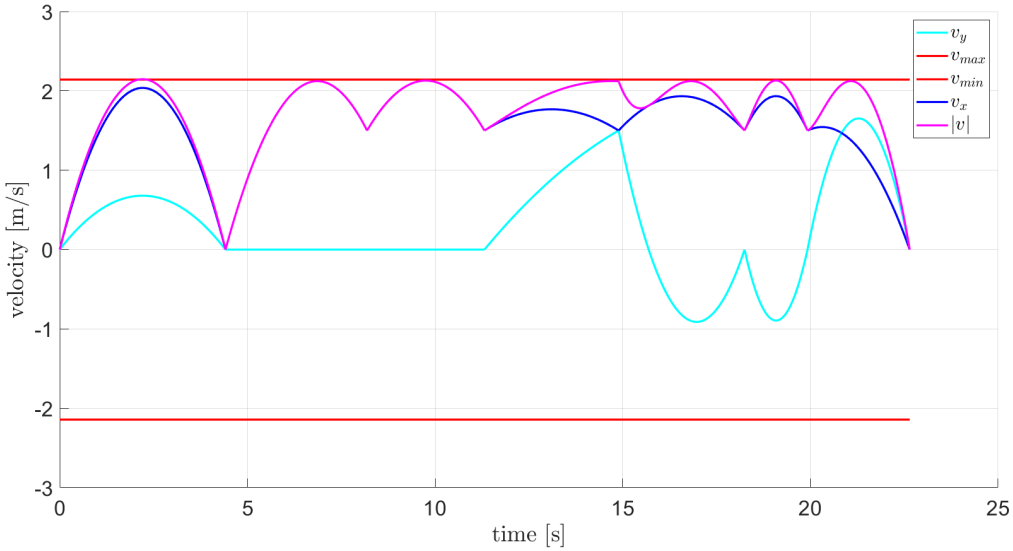


Figure 4.5: Velocity profile for the optimal trajectory in Figure 4.4. Blue and cyan lines depict the velocities along the x- and y-axes. Magenta curve represents the combined velocity. Red lines show the velocity limits.

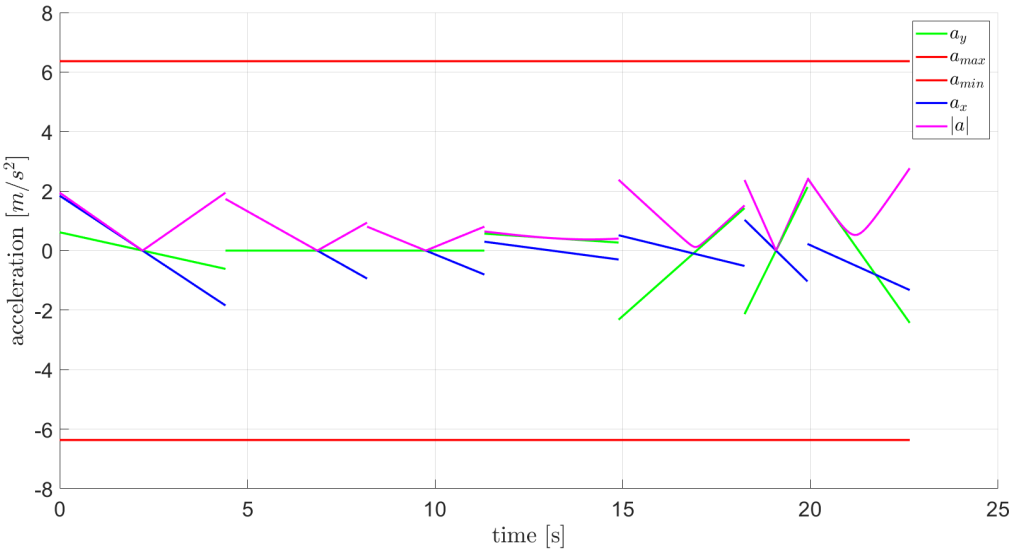


Figure 4.6: Actuation (acceleration) profile for the optimal trajectory in Figure 4.4. Blue and cyan lines depict the accelerations along the x- and y-axes. Magenta curve represents the combined acceleration. Red lines show the actuation limits.

4.4.2. Jerk-controlled system

Consider a planning scenario involving a quadrotor UAV operating within a 2D configuration space. The system is characterized by a 6D state space $(x, y, v_x, v_y, a_x, a_y)$, including position (x, y), velocity (v_x, v_y) and acceleration (a_x, a_y) along the x and y axes, in addition to a 2D actuation space $(j_x, j_y) = (u_x, u_y)$, constituted by the linear jerks along the respective axes. The quadrotor's dynamics is described by the following set of equations:

$$\begin{cases} \dot{x}(t) = v_x(t) \\ \dot{y}(t) = v_y(t) \\ \dot{v}_x(t) = a_x(t) \\ \dot{v}_y(t) = a_y(t) \\ \dot{a}_x(t) = j_x(t) = u_x \\ \dot{a}_y(t) = j_y(t) = u_y. \end{cases}$$

Leveraging quadrotor differential flatness, a polynomial parametrization of the flat output is assumed. In particular, a fifth-order polynomial is used to express the positions along the xx- and y-axes.

$$x(t) = \frac{C_1}{120}t^5 + \frac{C_2}{24}t^4 + \frac{C_3}{6}t^3 + \frac{C_4}{2}t^2 + C_5 t + C_6,$$
$$y(t) = \frac{B_1}{120}t^5 + \frac{B_2}{24}t^4 + \frac{B_3}{6}t^3 + \frac{B_4}{2}t^2 + B_5 t + B_6.$$

Obtaining velocities, accelerations, and jerks along both axes involves straightforwardly computing the first, second, and third derivatives of the respective positions. Applying a methodology similar to that outlined in Section 4.1.3, motion primitives are computed

by solving the following discretized problem:

$$
\begin{aligned}
&\underset{C_i,B_i,i=1:4}{\text{minimize}}\, J = \int_0^\tau \|u(t)\|^2\, dt + \rho\tau \\
\text{s.t. } & x(t) = \frac{C_1}{120}t^5 + \frac{C_2}{24}t^4 + \frac{C_3}{6}t^3 + \frac{C_4}{2}t^2 + C_5 t + C_6, \\
& y(t) = \frac{B_1}{120}t^5 + \frac{B_2}{24}t^4 + \frac{B_3}{6}t^3 + \frac{B_4}{2}t^2 + B_5 t + B_6, \\
& \|v(t)\| \in [-1.5\sqrt{2}, 1.5\sqrt{2}],\ t \in [0,\tau], \\
& \|a(t)\| \in [-3\sqrt{2}, 3\sqrt{2}],\ t \in [0,\tau], \\
& \|j(t)\| \in [-15\sqrt{2}, 15\sqrt{2}],\ t \in [0,\tau], \\
& x(0) = x_i,\ y(0) = y_i, \\
& v_x(0) = v_{xi},\ v_y(0) = v_{yi}, \\
& a_x(0) = a_{xi},\ a_y(0) = a_{yi}, \\
& x(\tau) = x_f,\ y(\tau) = y_f, \\
& v_x(\tau) = v_{xf},\ v_y(\tau) = v_{yf}. \\
& a_x(\tau) = a_{xf},\ a_y(\tau) = a_{yf}.
\end{aligned}
\tag{4.7}
$$

where the weight on the time is chosen as $\rho = 10$, in order to obtain a faster trajectory, while simultaneously minimizing overall control effort. Similarly to the previous example, coupled constraints are considered for velocity, acceleration, and jerk.

To solve the discretized optimization problem, the built-in MATLAB function `fmincon` [10] is employed as a nonlinear programming solver. From the available solvers, the Sequential Quadratic Programming algorithm (SQP) is the chosen approach.

The database construction starts from an initial state positioned at $(x_i, y_i) = (0, 0)$. This process is based on a uniform square grid, with the final state position coordinates (x_f, y_f) residing within the region $[-4, 0) \cup (0, 4] \times [-4, 0) \cup (0, 4]$, where each grid cell spans one meter. The process of selecting initial and final velocities involves choosing among a set of three predefined values: $\{-1.5, 0, 1.5\} m/s$. Similarly, the selection of initial and final accelerations involves selecting from a set of three specified values: $\{-3, 0, 3\} m/s^2$. Subsequently, the motion primitives are computed for every combination of initial states $(x_i, y_i, v_{xi}, v_{yi}, a_{xi}, a_{yi})$ and final states $(x_f, y_f, v_{xf}, v_{yf}, a_{xf}, a_{yf})$ by solving Problem 4.7. Exploiting the symmetry property, the creation of the reference trajectories (equivalent to 1/4 of the entire database) requires approximately 20 hours. Fortunately, this time frame does not pose any issue for the motion planning problem, as the resolution of the TPBVPs is conducted offline. The complete database comprises a total of 46656 motion

primitives. The numerical details regarding the database generation can be found in Table 4.3.

Table 4.3: Database details for the jerk-controlled system.

Generation time	$\approx$ 20 hours
Number of primitives	46656

Upon comparing Table 4.1 and Table 4.3, a significant increase is evident in both the time needed for database generation and the quantity of primitives included within it. This increase can be attributed to the incorporation of a wider state space, which now encompasses acceleration within the database discretization. However, its impact on the motion planning problem remains minimal, as the database generation process takes place offline.

In the virtual indoor map setting, the vehicle is initialized with the state $(x, y, v_x, v_y) = (2, 1, 0, 0)$, and the destination state is set to $(x, y, v_x, v_y) = (38, 7, 0, 0)$. Simulations are performed on a personal computer equipped with an Intel Core i7 processor running at 2.30 GHz, with 16GB RAM. In Table 4.4 are reported some relevant results pertaining to motion planning simulations.

Table 4.4: Motion planning results for the jerk-controlled system

Algorithm runtime	5353.08 ms
Number of expanded states	15

Upon examining the results reported in Tables 4.2 and 4.4, a slight increase in the execution time of the algorithm and the expanded states can be noticed. This increase can be attributed to the adoption of a larger state space, thus increasing the overall cardinality of the graph. With a wider state space, the search algorithm encounters a potentially larger number of states eligible for expansion during the search process. For improved search speed, considering code optimization techniques and transitioning to a C++ implementation could be beneficial.

Figure 4.7 shows the simulation result for the jerk-controlled system. Figures 4.8, 4.9 and 4.10 depict the velocity, acceleration, and actuation profiles associated with the optimal trajectory shown in Figure 4.7. These figures offer a clear illustration of how the velocity, acceleration, and input constraints are met. It is important to note that the velocity profile

exhibits jerky acceleration behavior. This behavior arises from the constraint demanding that the velocity at each node matches exactly one of the values in the database. This observation highlights the significance of a meticulous choice in the velocity discretization step when trying to obtain a smoother velocity profile.

Upon reviewing Figure 4.9, it becomes evident that the acceleration profile no longer shows discontinuities. This can be attributed to the parametrization of the trajectory as a fifth-order polynomial.

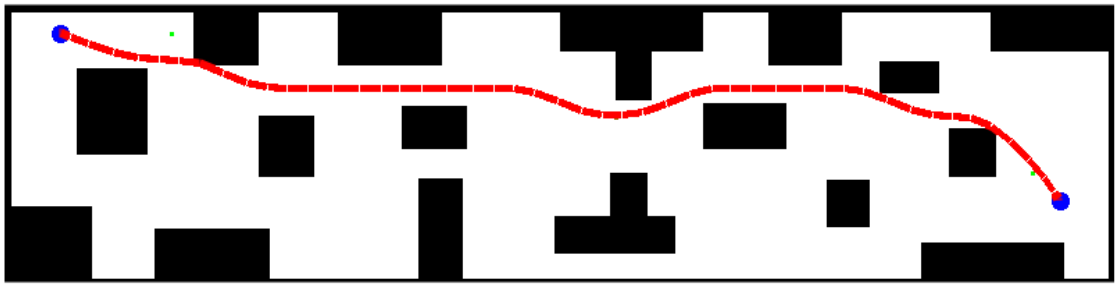


Figure 4.7: Simulation result for jerk-controlled system. The blue dot on the right represents the starting position, while the one on the left indicates the final position. The red curve is the optimal trajectory.

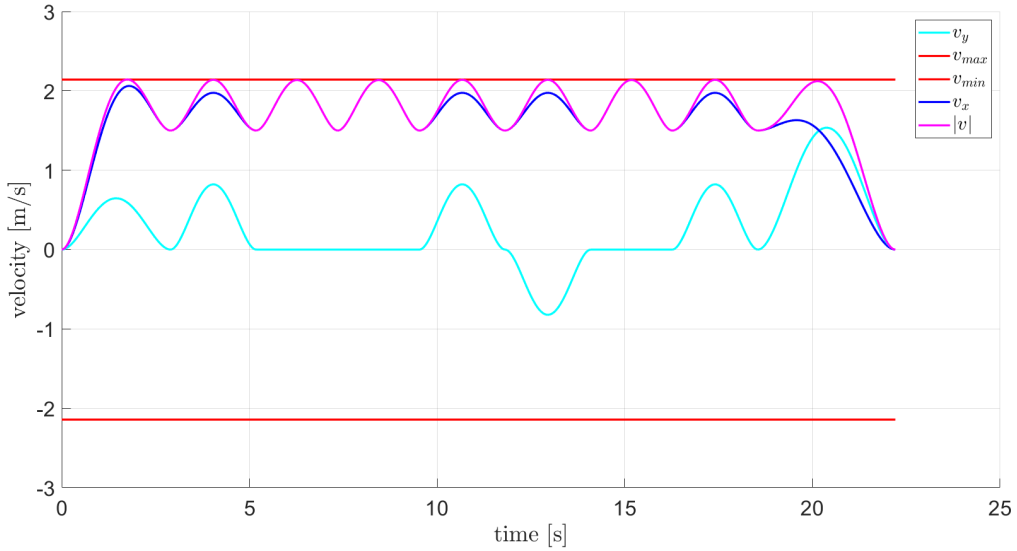


Figure 4.8: Velocity profile for the optimal trajectory in Figure 4.7. Blue and cyan lines depict the velocities along the x- and y-axes. Magenta curve represents the combined velocity. Red lines show the velocity limits.

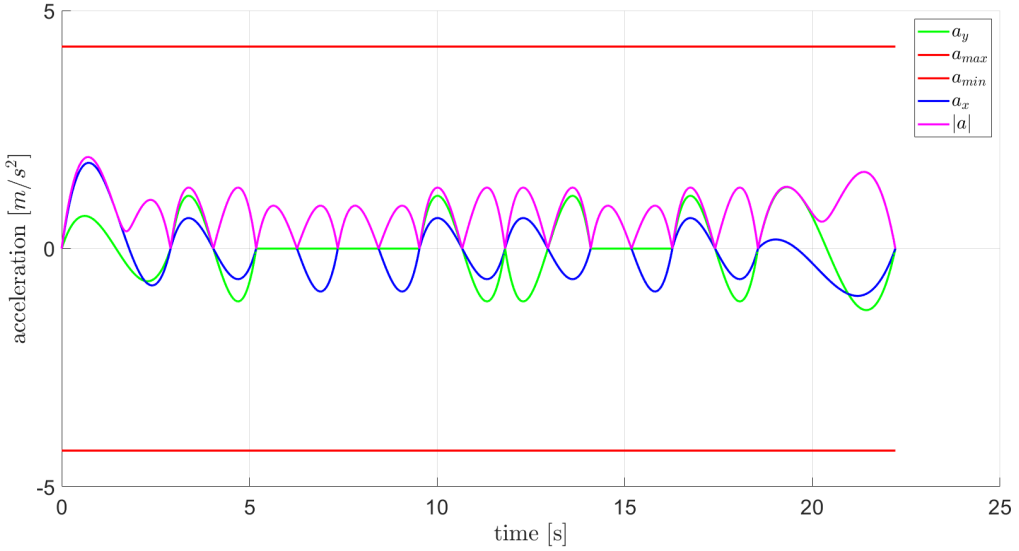


Figure 4.9: Acceleration profile for the optimal trajectory in Figure 4.7. Blue and cyan lines depict the accelerations along the x- and y-axes. Magenta curve represents the combined acceleration. Red lines show the acceleration limits.

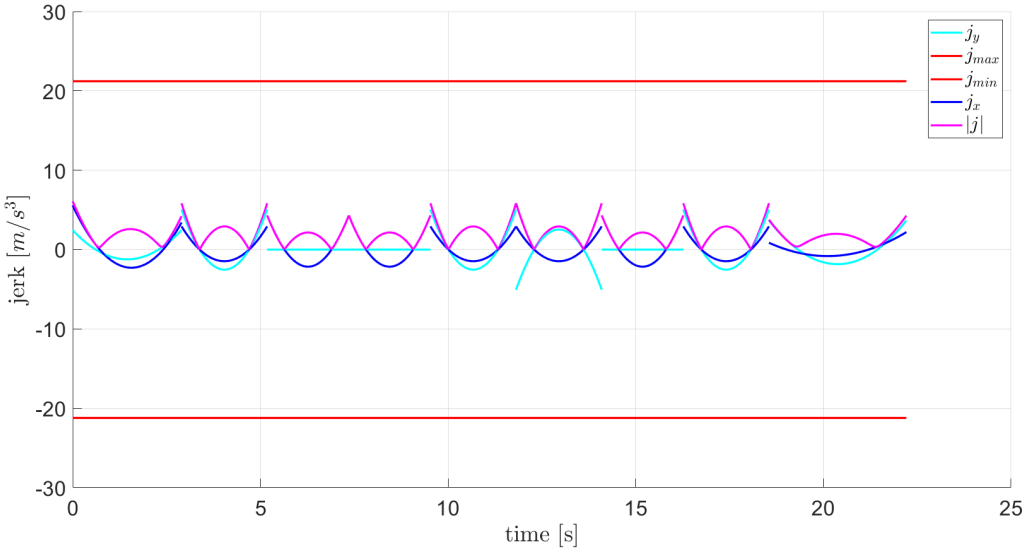


Figure 4.10: Actuation (jerk) profile for the optimal trajectory in Figure 4.7. Blue and cyan lines depict the jerks along the x- and y-axes. Magenta curve represents the combined jerk. Red lines show the actuation limits.

4.5. Conclusion

In this chapter, a search-based planner was introduced that innovatively incorporates a database of offline precomputed solutions. A core emphasis was placed on the creation of a motion primitive library, achieved through the resolution of multiple TPBVPs and leveraging the invariance properties inherent in dynamical systems. Despite the inherent challenges of planning with systems with arbitrary dynamics, the integration of a database of offline precomputed solutions within the search-based framework emerges as a strategic solution, effectively mitigating the computational load associated with this complexity.

In conclusion, the proposed advanced planner and trajectory generator, with its distinctive characteristics, stands out for its ability to navigate UAVs effectively through complex and challenging scenarios. Its prowess lies in the real-time generation of dynamically feasible, collision-free, resolution-optimal, and complete trajectories. What distinguishes this approach is not just its efficacy for quadrotors but its versatility to address a diverse range of dynamic systems, showcasing a level of adaptability and efficiency that goes beyond conventional planning methods.

5 | Experimental results

This chapter showcases the efficacy of the approach introduced in Chapter 4 for navigating real-world cluttered environments, demonstrated through a series of experiments. The experiments involve conducting simulations and real flight tests using the ANT-X quadrotor platform (depicted in Figure 5.1), within the Aerospace Systems and Control Laboratory (ASCL) of Politecnico di Milano University.

Figure 5.1: Quadrotor platform used for the experiments.

5.1. Problem setup

The operational area for the vehicle within the environmental map covers the region where $x \in [-5, 5]m$ and $y \in [-2, 2]m$. This map is populated with two rectangular obstacles measuring $0.4 \times 1.5 \times 2$ meters each. Positioned in the (x, y) plane (ground plane), the center of the first obstacle is located at $(x, y) = (-1.5, 0.5)$, while the center of the second obstacle is at $(x, y) = (1.5, -0.5)$. The obstacles are inflated by $0.3m$ to account for the radius of the drone and prevent potential collisions. This virtual environment is visualized in

Figure 5.2. The start position is set at $(x_{hi}, y_{hi}, z_{hi}) = (-3.5, 0.5, 1)$, while the goal region

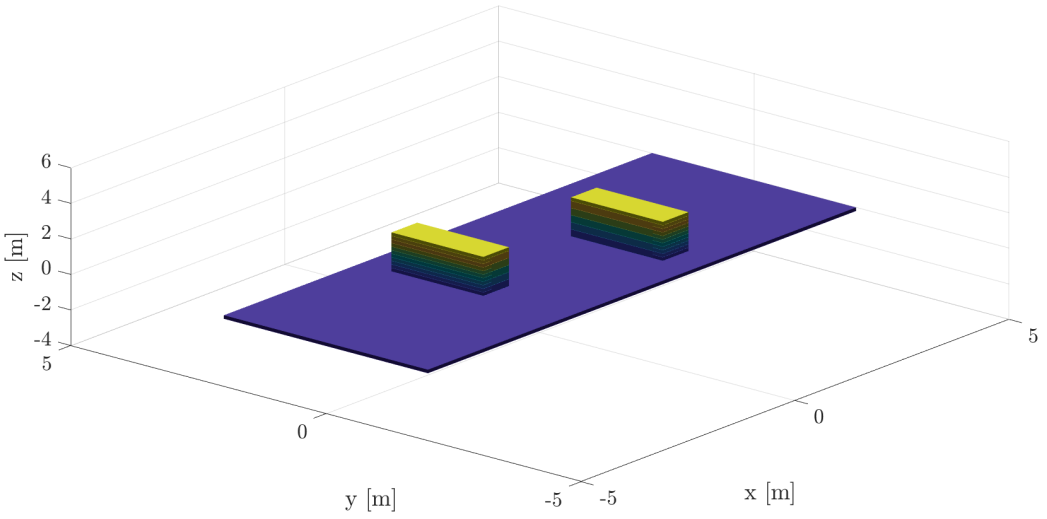


Figure 5.2: Virtual indoor environment used for the experiments.

is defined as a square with a side length of $0.5m$, centered at $(x_{hf}, y_{hf}, z_{hf}) = (3, -0.5, 1)$. The trajectory starts and ends with zero velocity and acceleration. Throughout the planning process, a constant altitude of $1m$ is maintained. The planning method is the one described in Chapter 4.

Attention must be paid to the change of reference frame adopted in this chapter. In both the simulations and the experimental tests, the NED reference frame is used. In contrast to the xyz coordinate system used so far, the N-axis aligns with the y-axis, the E-axis aligns with the x-axis, and the D-axis points downward, in correlation with the inverse direction of the z-axis represented in Figure 5.2.

Subsequent sections will showcase the results of simulations and real-world experiments derived from two distinct runs, each leveraging a different databases of motion primitives. In particular, in the first example, the upper and lower limits for total velocity, acceleration, and jerk (control input) are defined as $v_{max} = 1.5\sqrt{2}m/s$, $a_{max} = 4.5\sqrt{2}m/s^2$ and $j_{max} = 50\sqrt{2}m/s^3$, respectively. Consequently, this results in the total velocity and acceleration being constrained within $v \in [-v_{max}, v_{max}]m/s$ and $a \in [-a_{max}, a_{max}]m/s^2$ respectively, and the overall control input is bounded within $u \in [-j_{max}, j_{max}]m/s^3$. The construction of the database starts from an initial state positioned at $(x_i, y_i) = (0, 0)$. This process is based on a uniform square grid, with the final state position coordinates (x_f, y_f) residing within the region $[-1.5, 0) \cup (0, 1.5] \times [-1.5, 0) \cup (0, 1.5]$, where each grid cell spans half a meter. The process of selecting initial and final velocities involves choos-

ing among a set of three predefined values: $\{-1.5, 0, 1.5\}m/s$. Similarly, the selection of initial and final accelerations involves selecting from a set of three specified values: $\{-4.5, 0, 4.5\}m/s^2$.

In the second example, the limits for total velocity and acceleration remain consistent with the first example ($v_{max} = 1.5\sqrt{2}m/s$, $a_{max} = 4.5\sqrt{2}m/s^2$). However, the maximum jerk (control input) is lower, set at $j_{max} = 15\sqrt{2}m/s^3$, intended to produce a smoother and less aggressive trajectory. The process of constructing the database begins from an initial state positioned at $(x_i, y_i) = (0, 0)$. This construction method follows a uniform square grid framework, with the final state position coordinates (x_f, y_f) lying within the region $[-2, 0) \cup (0, 2] \times [-2, 0) \cup (0, 2]$, where each grid cell spans half a meter. Within this grid, the velocities and accelerations at the nodes can assume values from the sets $-1.5, 0, 1.5m/s$ and $-4.5, 0, 4.5m/s^2$, respectively.

5.1.1. Simulations setup

Before conducting real-world experiments, simulations are performed using the ANT-X simulator. The simulations encompass the following phases:

1. Start: quadrotor starts and remains in the initial position $(x_i, y_i, z_i) = (-3.5, 0.5, 0)$ (drone on the floor) for $5s$.
2. Take off: the quadrotor moves to the initial hovering position at the coordinates $(x_{hi}, y_{hi}, z_{hi}) = (-3.5, 0.5, 1)$. This phase lasts for a duration of $10s$.
3. Hovering: the quadrotor hovers at the initial hover position at the coordinates $(x_{hi}, y_{hi}, z_{hi}) = (-3.5, 0.5, 1)$ for $5s$.
4. Trajectory tracking: the quadrotor follows the planned trajectory.
5. Hovering: the quadrotor hovers at the final hover position, which is defined as a square region with a side length of $0.5m$ centered at $(x_{hf}, y_{hf}, z_{hf}) = (3, -0.5, 1)$ for $10s$.
6. Landing: the quadrotor moves to the final position which is defined as a square region with a side length of $0.5m$ centered at $(x_f, y_f, z_f) = (3, -0.5, 0)$. The duration of this phase is $10s$.
7. Hovering: the quadrotor hovers in the final position (drone on the ground) for $5s$.

5.1.2. Real-world experiments setup

After completing the simulation phase, real-world experiments are conducted within the Aerospace Systems and Control Laboratory (ASCL) at Politecnico di Milano University, employing the ANT-X quadrotor platform shown in Figure 5.1. Figure 5.3 shows the indoor environment in which the experiments are carried out.

Figure 5.3: Indoor environment used for the experiments.

5.2. Experimental test 1

In this section, both simulations and real-world experiments are executed for the example employing the first motion primitive database explained in Section 5.1.

5.2.1. Simulation 1 results

Figure 5.4 illustrates the planned trajectory for the first simulation, with the right-side colorbar indicating the velocity along the trajectory. It can be seen that our system can successfully reach the goal without hitting any obstacle.

Figure 5.5 and Figure 5.6 show the position and velocity profiles, together with their respective errors. In these plots, the dashed lines represent the setpoints, whereas the solid lines depict the actual position and velocity of the quadrotor model. In particular, the maximum position error is around $21cm$ along the east axis (x-axis in Figure 5.2), while the maximum velocity error is approximately $0.46m/s^2$ along the north axis (y-axis in Figure 5.2).

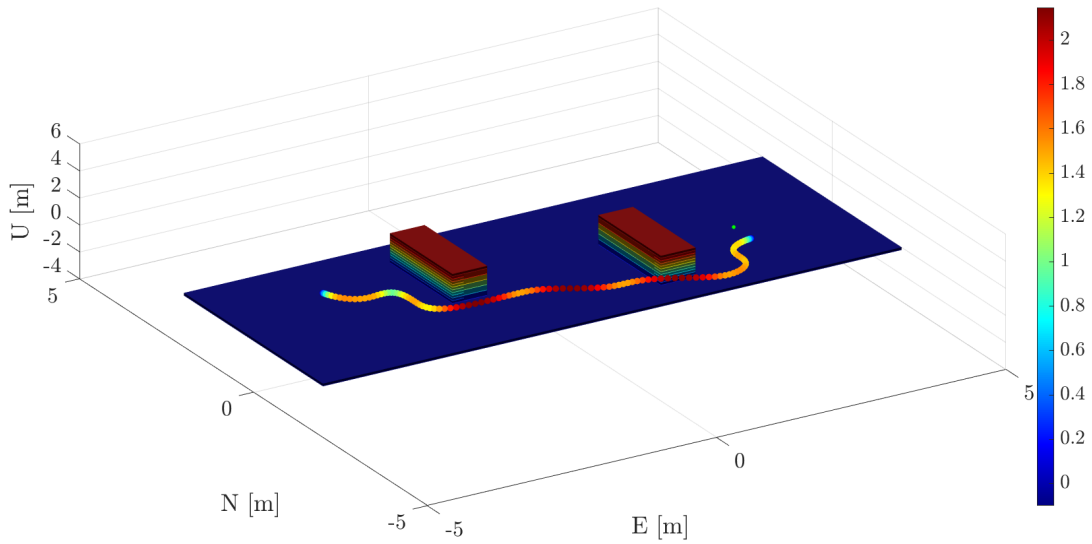


Figure 5.4: Planned trajectory for Simulation 1. The color bar on the right indicates the velocity along the trajectory.

In Figure 5.7, the thrust demands on the four motors of the quadrotor are shown. Remarkably, the graph reveals distinct peaks, with maximum and minimum thrust requirements reaching approximately 60% and 32% of the total thrust, respectively. These peaks coincide with the quadrotor's transitions between different motion primitives, necessitating elevated control inputs to effectuate the change.

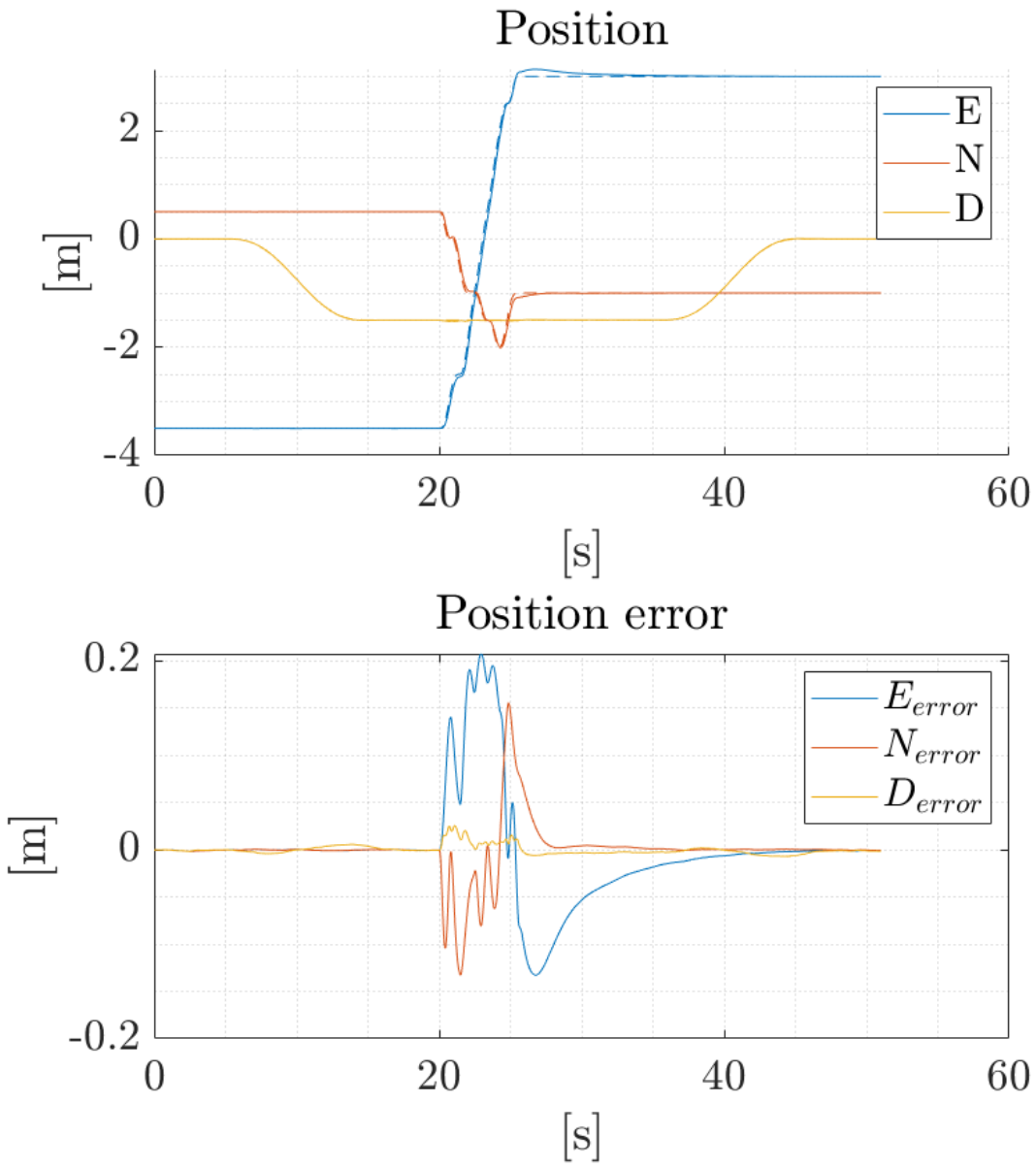


Figure 5.5: Position profile and position error for Simulation 1. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities.

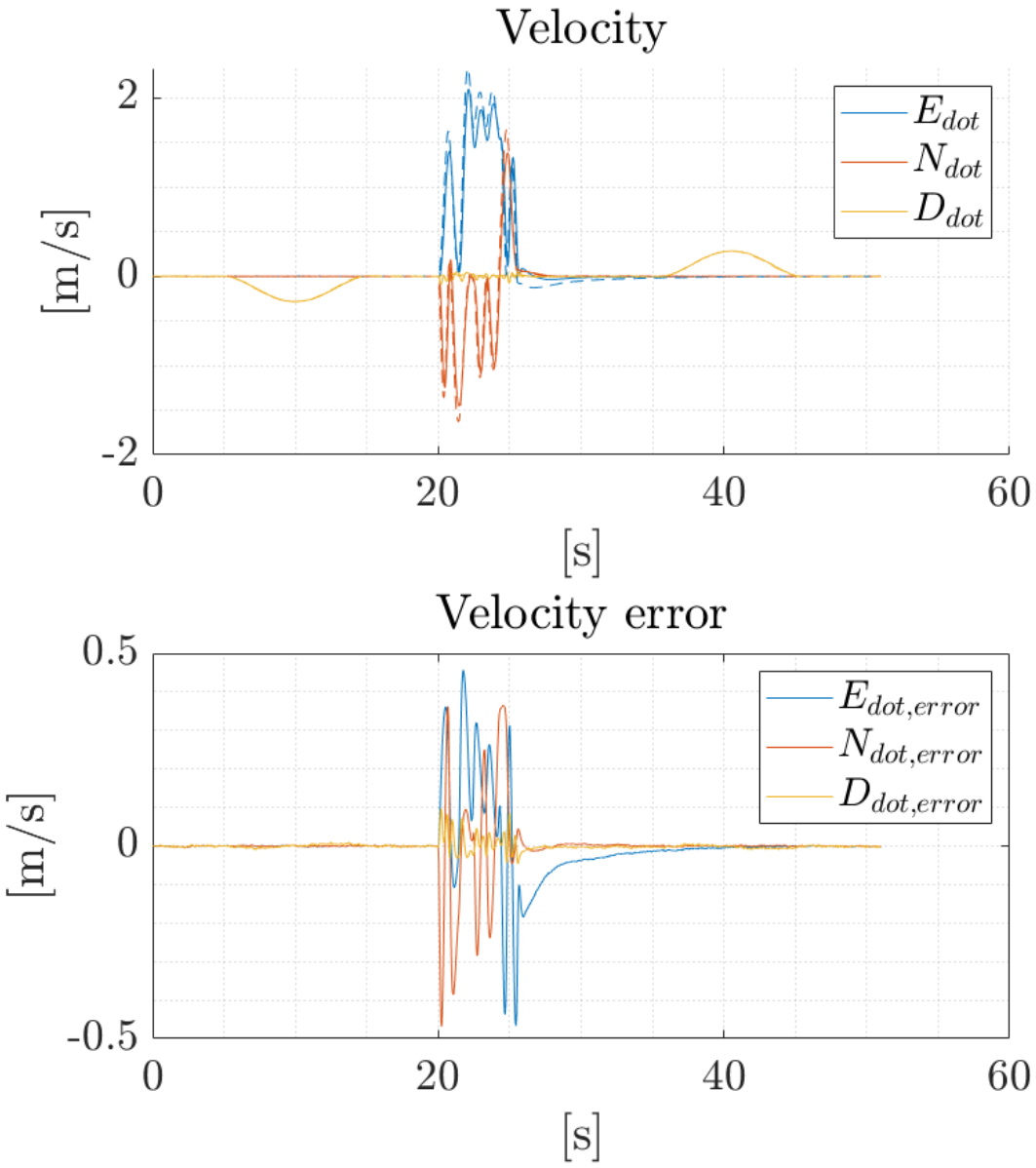


Figure 5.6: Velocity profile and velocity error for Simulation 1. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities.

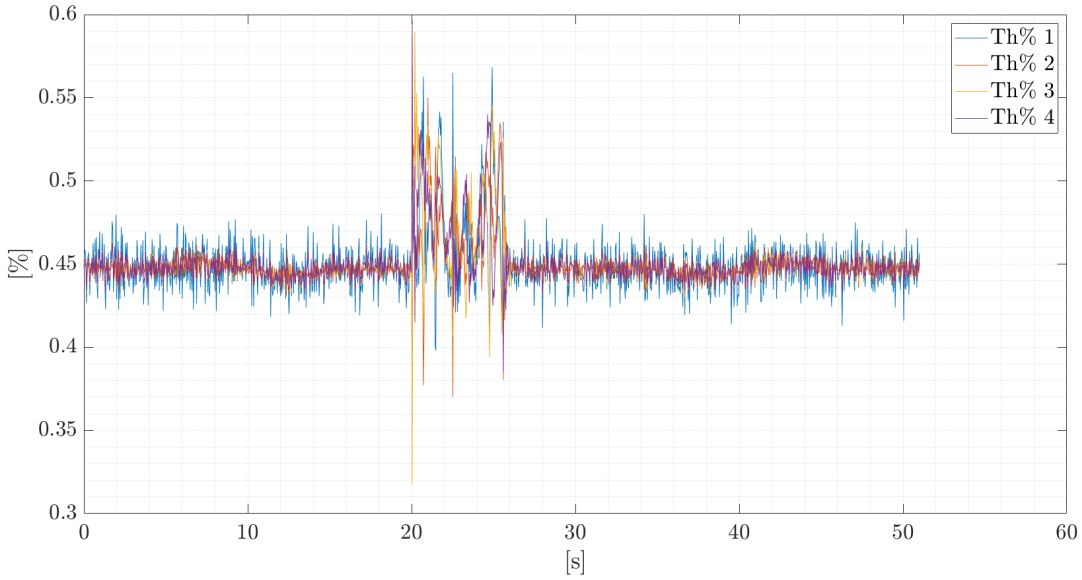


Figure 5.7: Motors thrust for Simulation 1.

5.2.2. Real-world experiment 1 results

Figure 5.8 shows the trajectory in the (E, N) plane. The dashed curve represents the position setpoint and the solid curve depicts the actual trajectory tracked by the quadrotor in real-world Experiment 1.

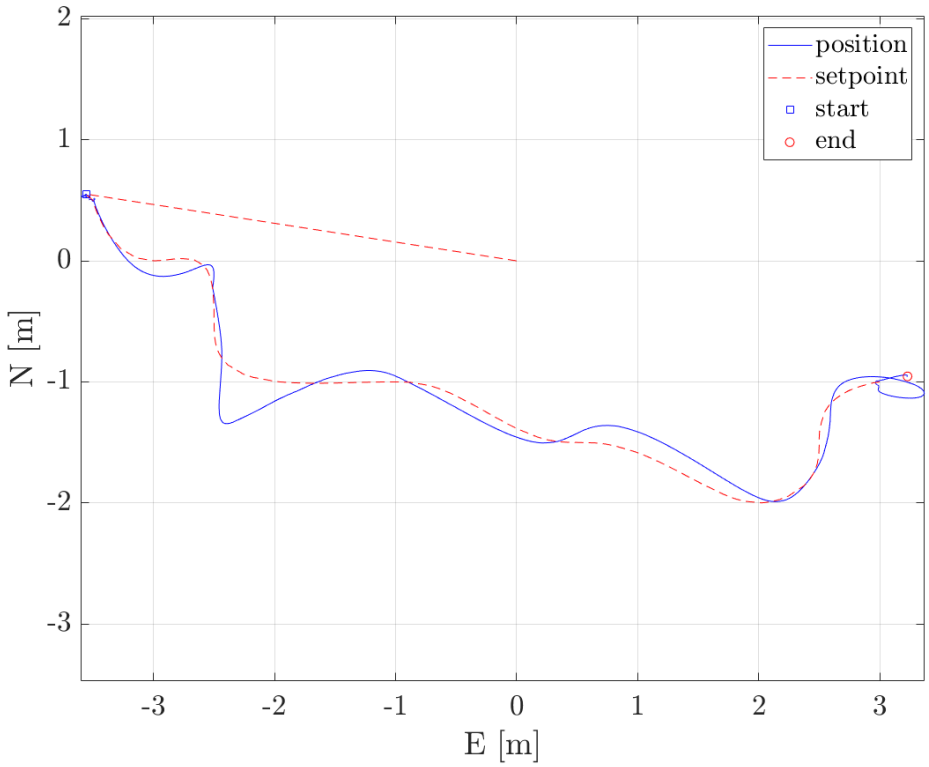


Figure 5.8: Position setpoint (dashed curve) and corresponding trajectory tracked by the quadrotor (solid curve) in real-world Experiment 1.

Figures 5.9 and 5.10 present the position and velocity profiles along the three axes, alongside their respective setpoints. Upon closer examination of these graphs, it becomes evident that the position, particularly along the North axis, is inadequately tracked. Furthermore, the velocity, along both the east and north axes, also demonstrates suboptimal tracking performance. Poor tracking performance can be attributed to the aggressiveness of the planned trajectory, which requires abrupt changes in velocity characterized by accelerations up to $6m/s^2$ and demanding high control inputs. This, in turn, is linked to the discretization of the motion primitives database. In this particular simulation, the library includes trajectories of lengths $0.5, 1$, or $1.5m$, resulting in high control inputs to achieve the specified velocities at nodes ($0m/s$ or $2m/s$) over short distances. To mitigate this problem, several strategies can be considered, all centered around the refinement of the

motion primitives library's discretization. The most effective approach involves introducing additional velocity and acceleration values at the nodes ($0, 1, 2m/s$ and $0, 3, 4.5m/s^2$, for example) to produce smoother and less aggressive trajectories. Although effective, this approach comes at the cost of a substantial increase in the number of allowed motions, demanding extensive computation time, typically several days, to resolve all TPBVPs and generate the motion primitives database. Alternatively, a quicker solution to implement involves incorporating longer trajectories into the database. This modification extends the distance available to the vehicle to reach the prescribed speed at the nodes, resulting in a lower acceleration and reduced control input. Furthermore, the short length of the motion primitives ($0.5, 1, 1.5m$) included in the database leads to a planned trajectory lacking smoothness, characterized by numerous curves and abrupt changes in direction, as evidenced in Figure 5.8. These considerations underscore the critical role of database discretization in addressing specific problems, necessitating thoughtful consideration and tailored selection. In the next simulation, a different database including longer motion primitives ($0.5, 1, 1, 5, 2m$) is adopted, aiming to correct some of the above-mentioned issues.

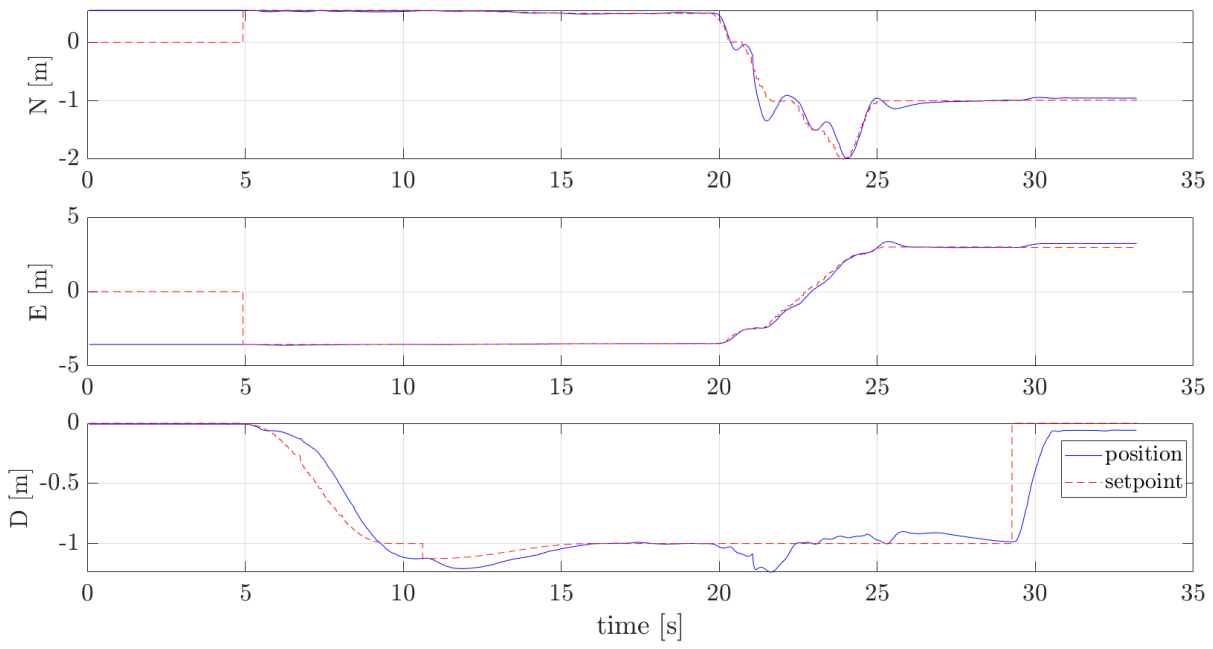


Figure 5.9: Profile of the position setpoint (dashed curve) and profile of the position tracked by the quadrotor (solid curve) in real-world Experiment 1.

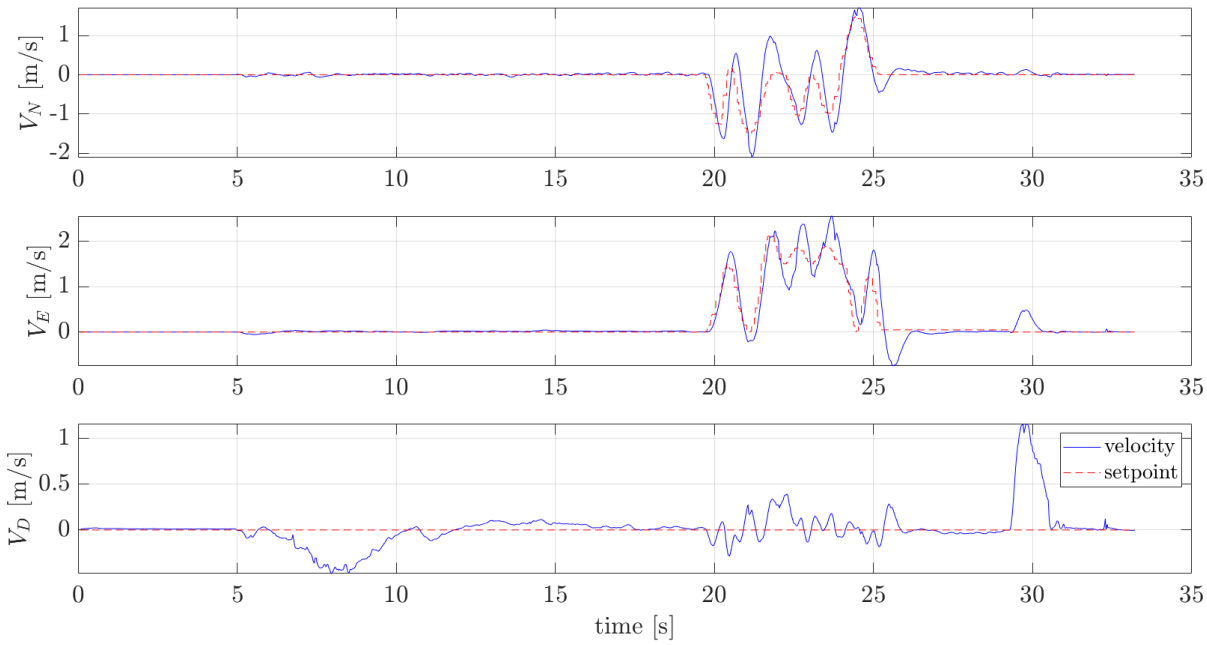


Figure 5.10: Profile of the velocity setpoint (dashed curve) and profile of the velocity tracked by the quadrotor (solid curve) in real-world Experiment 1.

5.3. Experimental test 2

In this section, both simulations and real-world experiments are executed for the example employing the second motion primitive database explained in Section 5.1

5.3.1. Simulation 2 results

Figure 5.11 visually represents the planned trajectory for the second simulation, with the right-side colorbar indicating the velocity along the trajectory. It can be seen that our system successfully navigates to the goal without hitting any obstacles. In Figures 5.12

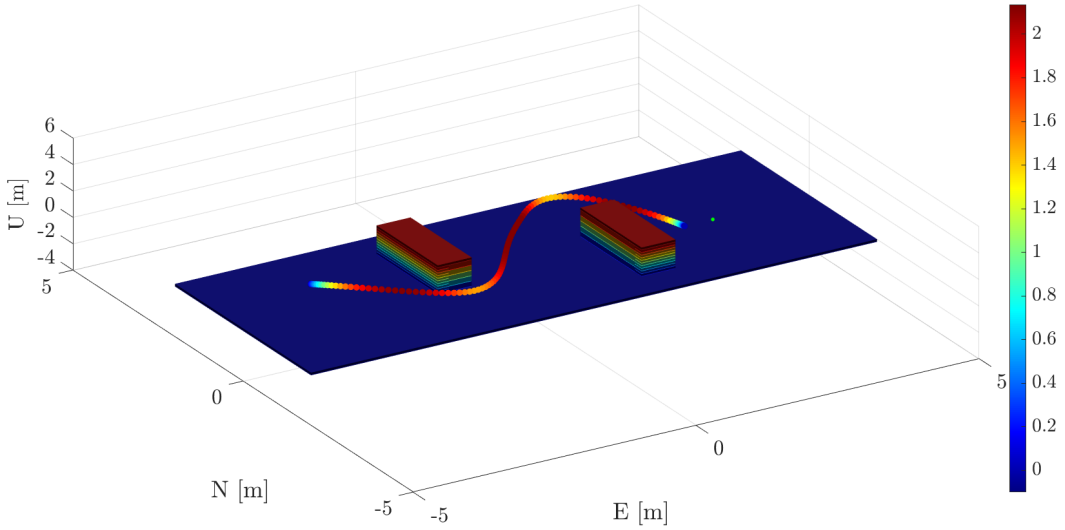


Figure 5.11: Planned trajectory for Simulation 2. The colorbar on the right indicates the velocity along the trajectory.

and 5.13, position and velocity profiles are presented, accompanied by their respective errors. The dashed curves represent the setpoint, while the solid curves depict the actual position and velocity of the quadrotor model. In particular, the maximum position error is approximately $21cm$ along the North axis, and the maximum velocity error is close to $0.45m/s$ along the same axis. This provides insights into both position and velocity performance, highlighting the accuracy of the quadrotor model in trajectory tracking.

Figure 5.14 shows the thrust required by the four motors. In particular, the maximum and minimum thrusts required are approximately 50% and 41% of the total thrust, respectively. Moreover, the graph shows peaks significantly lower than those in Figure 5.7, resulting in a less aggressive trajectory.

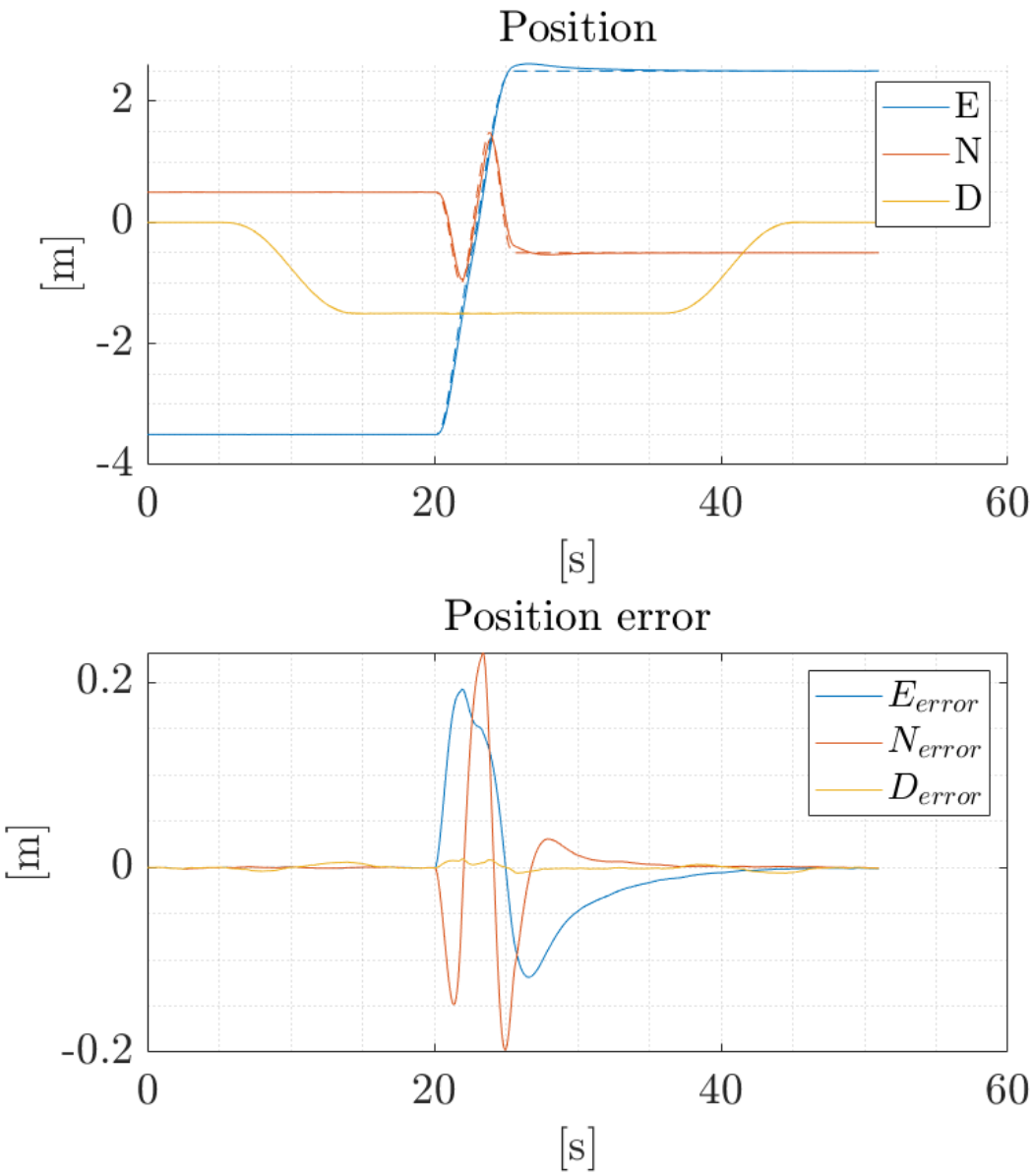


Figure 5.12: Position profile and position error for Simulation 2. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities.

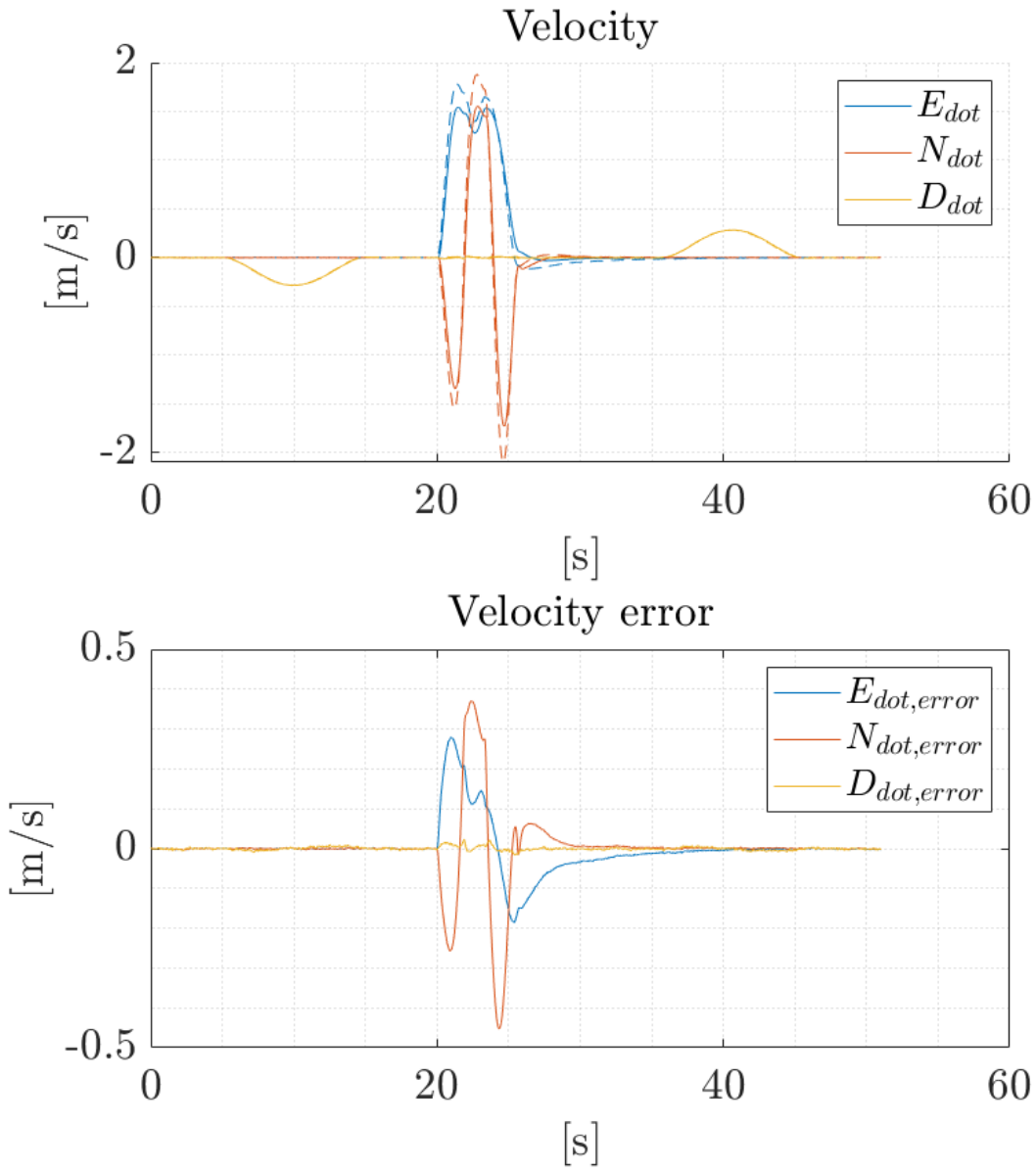


Figure 5.13: Velocity profile and velocity error for Simulation 2. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities.

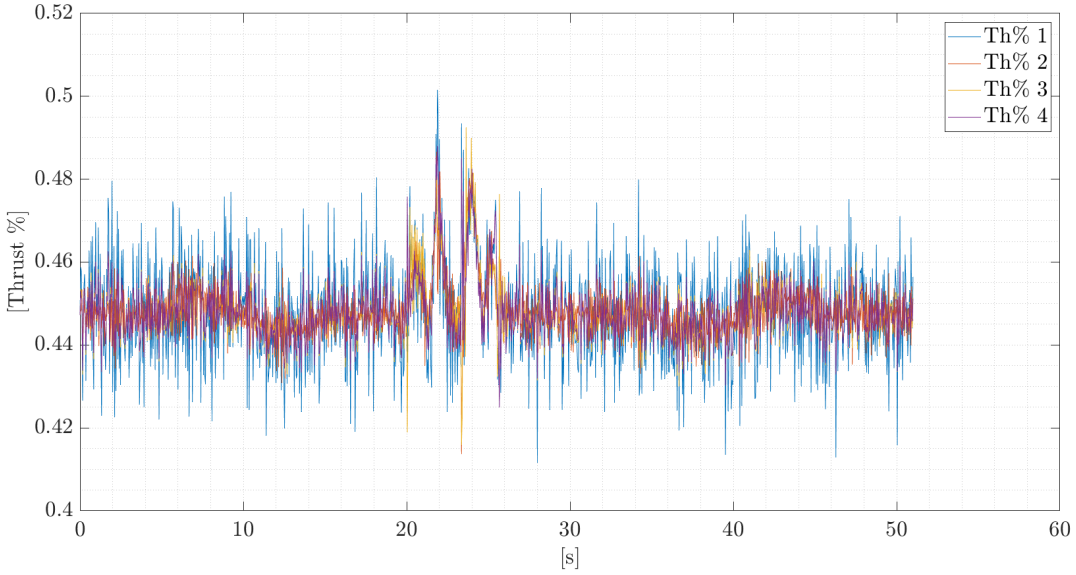


Figure 5.14: Motors thrust for Simulation 2.

5.3.2. Experimental test 2 results

Figure 5.15 shows the planned trajectory in the plane (E, N). The dashed curve depicts the position setpoint, while the solid curve represents the actual trajectory tracked by the quadrotor in real-world Experiment 2. A comparative analysis with Figure 5.8 related to Experiment 1 reveals a notable improvement in tracking performance.

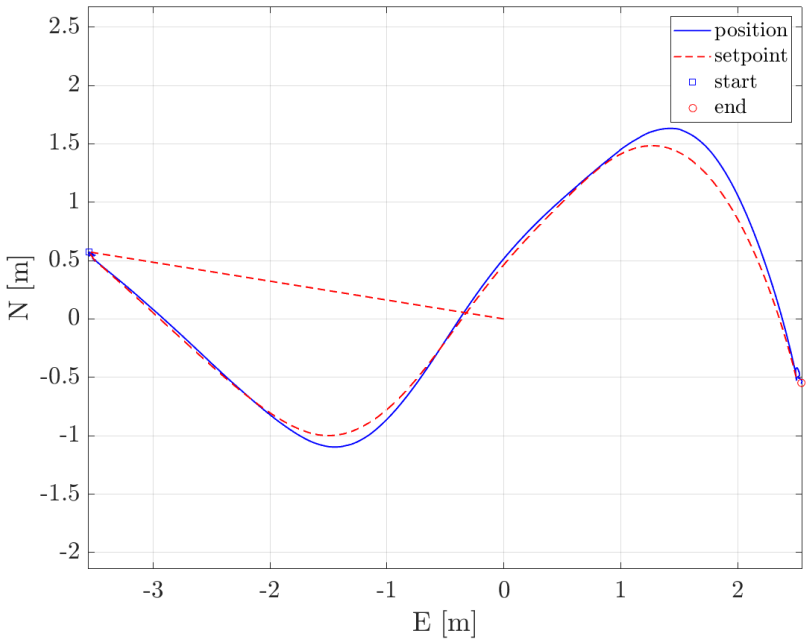


Figure 5.15: Position setpoint (dashed curve) and corresponding trajectory tracked by the quadrotor (solid curve) in real-world Experiment 2.

Figures 5.16 and 5.17 show the position and velocity profiles across the three axes, accompanied by their respective setpoints. Upon examination of these graphs, it becomes evident that both the position and the velocity along all the axes are well tracked. This improved tracking performance, compared to Example 1, can be attributed to the reduced aggressiveness of the planned trajectory. In this specific scenario, the maximum acceleration achieved is approximately $4.5 m/s^2$ and the control inputs are also significantly lower. This achievement is obtained by utilizing a different motion primitives database. In particular, longer trajectories are incorporated into the database for this experiment, enabling the vehicle to achieve the selected velocities at the nodes with reduced control effort. This strategic selection of the database discretization contributes to the overall enhancement of the tracking.

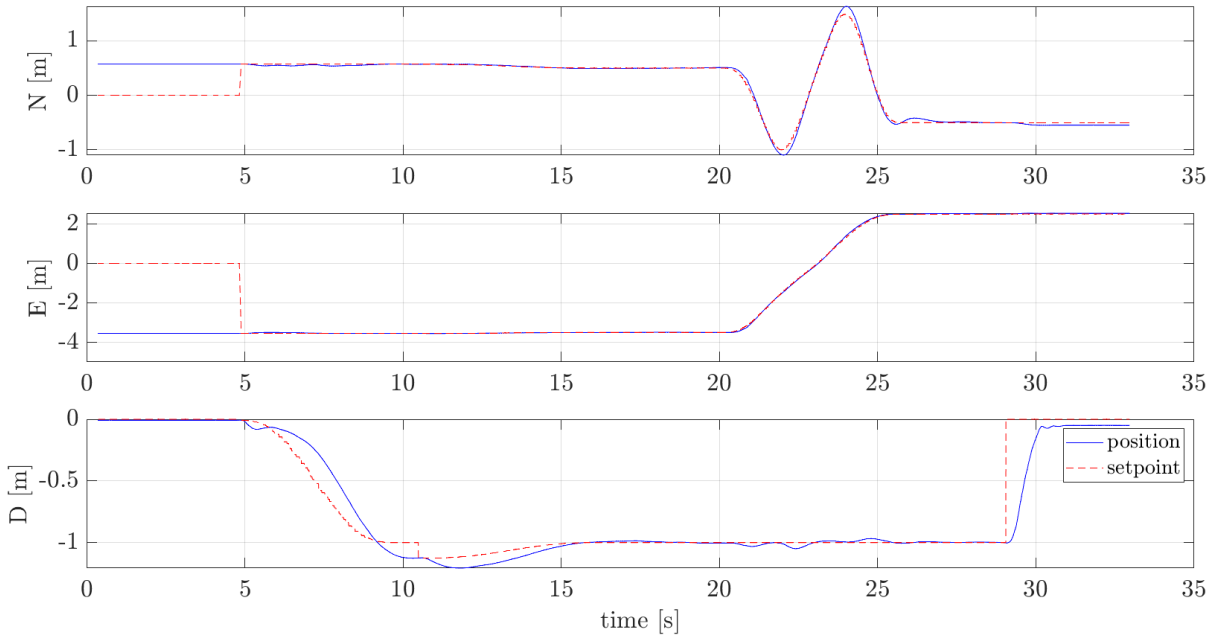


Figure 5.16: Profile of the position setpoint (dashed curve) and profile of the position tracked by the quadrotor (solid curve) in real-world Experiment 2.

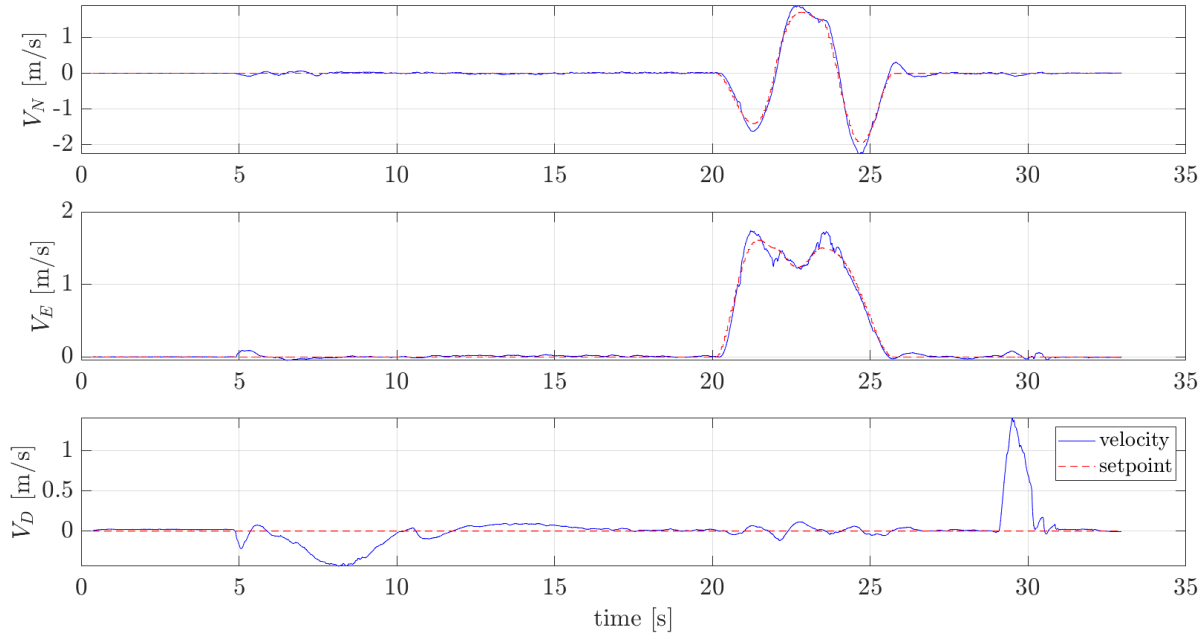


Figure 5.17: Profile of the velocity setpoint (dashed curve) and profile of the velocity tracked by the quadrotor (solid curve) in real-world Experiment 2.

6 | Conclusions and prospective works

6.1. Conclusions

In summary, this thesis addresses trajectory planning and generation for UAVs, especially in the context of fast and aggressive maneuvering in complex and cluttered environments. Here, the focus extends beyond mere obstacle avoidance to include compliance with vehicle dynamics, as well as the fulfillment of dynamic and input constraints. The implementation of two different methods has provided valuable insights and solutions, each with its unique strengths and contributions.

The first method, a search-based planner tailored for quadrotors, demonstrated exceptional performance in addressing the kinodynamic motion planning problem for quadcopters. Moreover, this approach has applicability beyond quadrotors, extending to systems with straightforward dynamics and existing analytical solutions. Using motion primitives and a forward propagation approach, this methodology excels in generating real-time and dynamically feasible trajectories for quadrotor UAVs.

The second method introduced a search-based planner that incorporates a database of offline precomputed solutions, showcasing efficiency and applicability beyond quadrotors to include arbitrary dynamical systems. The integration of a library of motion primitives emerged as a strategic solution, effectively mitigating the computational challenges associated with planning with systems with arbitrary dynamics. This approach, demonstrated through both numerical examples and experimental tests, not only broadens the horizons of applicability but also highlights its adaptability and efficiency in handling diverse dynamic systems.

In conclusion, the solutions presented in this thesis offer a comprehensive and versatile framework for UAV navigation in complex and potentially hostile environments, especially when the vehicle is required to use its full maneuvering capabilities and react in real time to changes. The numerical examples and the real-world experiments conducted vali-

date the performance and efficacy of both methods, emphasizing their ability to generate resolution-complete, collision-free, resolution-optimal, and dynamically feasible trajectories. Furthermore, this work contributes not only to the field of quadrotor navigation but also marks a substantial step forward in the advancement in the field of motion planning for arbitrary dynamic systems. This will lead to improvements in both efficiency and safety compared to existing approaches.

6.2. Methods comparison

In this comparative analysis, the method introduced in Chapter 3 (referred to as the first method) is compared with the method presented in Chapter 4 (referred to as the second method), and the main advantages and disadvantages of each approach are outlined.

- The first method calculates motion primitives in real-time at each iteration of the planner. This makes it computationally tractable for relatively straightforward systems, such as quadrotors, where analytical solutions are available. However, for more complex systems lacking analytical solutions, solving complex and time-consuming TPBVPs becomes necessary, making real-time computation of motion primitives intractable. The second method efficiently addresses this challenge by introducing a motion primitive library, which extends the motion planning solution to any arbitrary dynamic system.
- The forward propagation technique employed by the first method precludes the creation of a motion primitive database, which requires an exact connection between the states. This limitation is effectively overcome by the second method, which employs a state-based steering method. Moreover, most of the search-based planners that have a guarantee on optimality require the nodes to be connected exactly and optimally.
- The first method employs motion primitives with fixed duration, hindering the overall objective of generating a minimum-time trajectory. This is due to the fixed duration of a motion primitive connecting two states, which remains unchanged even if a shorter-duration trajectory that complies with dynamics and input constraints is feasible. In contrast, the second method adopts minimum-time motion primitives, optimizing their duration. This strategic choice enables the generation of a final trajectory that is optimal in terms of time.
- The analytical solutions employed in Chapter 3 do not incorporate information on the dynamic constraints of the system, such as maximum velocity, accelerations, and

actuation limits. Consequently, after the computation of the motion primitive, one must verify that the computed trajectory is feasible for the vehicle. In contrast, the approach presented in Chapter 4 integrates information about the system's dynamic constraints directly into the database of motion primitives, eliminating the necessity for a subsequent feasibility verification step.

6.3. Impact and applications

As UAVs are gaining popularity for a wide range of applications, it is crucial to ensure their operational safety and efficiency. The advanced planner and trajectory generator proposed in this thesis provide a solution for UAVs to navigate efficiently in complex and challenging scenarios without collisions.

The results of this thesis can be applied to various fields, both in the commercial and military sectors. Commercial applications such as delivery services or infrastructure inspections can benefit from safe, efficient and agile UAVs. On the other hand, military operations often involve navigating through complex and challenging terrains, where advanced planning is essential. Beyond immediate applications, the thesis itself could contribute to the field of robotics and autonomous systems by addressing complex challenges in trajectory planning and control. This could lead to new insights, methodologies, and techniques that can be applied to other domains as well.

6.4. Prospective works

Building on the foundations laid in this thesis, future efforts will expand the focus beyond the current scope of kinodynamic motion planning in known environments. A natural development involves extending the proposed methodologies to tackle the challenges posed by unknown and dynamic environments. This extension aims to allow the system to dynamically re-plan trajectories, using real-time data to navigate and avoid newly detected obstacles.

In the context of employing a database of motion primitives, the robustness of the offline phase is highly dependent on the accuracy of the model employed. To address modeling errors and improve trajectory tracking, the application of Iterative Learning Control (ILC) emerges as a promising avenue. The implementation of ILC proposed in [2] requires a meticulous two-stage training process, involving learning in simulations and real flights, to systematically construct a comprehensive database of motion primitives. This library is subsequently referenced in real time, with the primitives exploited by an intelligent con-

trol mechanism during maneuvers required for specific segments of a trajectory. Crucially, the construction of the library is supported by a meticulous and extensive phase of experimental testing, ensuring the system's adaptability across diverse conditions. This precise testing minimizes potential modeling uncertainties that could otherwise compromise planning performance. As a result, the robustness of the system is maintained, providing a reliable basis for the real-time use of motion primitives in response to dynamic trajectory requirements.

In pursuit of enhanced real-world applicability, particularly in dynamic environments demanding rapid replanning, the optimization of the proposed algorithms can be achieved through their implementation in C++. This strategic choice not only aligns with industry standards, but also lays the foundation for improved computational efficiency, contributing to the perfect integration of these algorithms into the operational framework of real vehicles.

Bibliography

[1] C. L. Bottasso, D. Leonello, and B. Savini. Path planning for autonomous vehicles by trajectory smoothing using motion primitives. *IEEE Transactions on Control Systems Technology*, 16(6):1152–1168, 2008. doi: 10.1109/TCST.2008.917870.

[2] E. Camci and E. Kayacan. Learning motion primitives for planning swift maneuvers of quadrotor. *Autonomous Robots*, 43:1733–1745, 2019. doi: https://doi.org/10.1007/s10514-019-09831-w.

[3] M. Dharmadhikari, T. Dang, L. Solanka, J. Loje, H. Nguyen, N. Khedekar, and K. Alexis. Motion primitives-based path planning for fast and agile exploration using aerial robots. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 179–185, 2020. doi: 10.1109/ICRA40945.2020.9196964.

[4] E. Frazzoli, M. Dahleh, and E. Feron. Robust hybrid control for autonomous vehicle motion planning. In *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*, volume 1, pages 821–826, 2000. doi: 10.1109/CDC.2000.912871.

[5] E. Frazzoli, M. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. *Journal of guidance, control, and dynamics*, 25(1):116–129, 2002.

[6] S. Karaman and E. Frazzoli. Optimal kinodynamic motion planning using incremental sampling-based methods. In *49th IEEE Conference on Decision and Control (CDC)*, pages 7681–7687, 2010. doi: 10.1109/CDC.2010.5717430.

[7] S. Liu. *Motion Planning For Micro Aerial Vehicles*. PhD thesis, University of Pennsylvania, 2018.

[8] S. Liu, N. Atanasov, K. Mohta, and V. Kumar. Search-based motion planning for quadrotors using linear quadratic minimum time control. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2872–2879, 9 2017. doi: 10.1109/IROS.2017.8206119.

[9] S. Liu, K. Mohta, N. Atanasov, and V. Kumar. Search-based motion planning for

aggressive flight in se(3). *IEEE Robotics and Automation Letters*, 3(3):2439–2446, 2018. doi: 10.1109/LRA.2018.2795654.

[10] MathWorks. Find minimum of constrained nonlinear multivariable function - matlab fmincon. `https://it.mathworks.com/help/optim/ug/fmincon.html`, (n.d.).

[11] D. Mellinger and V. Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 5 2011. doi: 10.1109/ICRA.2011.5980409.

[12] M. W. Mueller, M. Hehn, and R. D'Andrea. A computationally efficient algorithm for state-to-state quadrocopter trajectory generation and feasibility verification. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3480–3486, 11 2013. doi: 10.1109/IROS.2013.6696852.

[13] M. W. Mueller, M. Hehn, and R. D'Andrea. A computationally efficient motion primitive for quadrocopter trajectory generation. *IEEE Transactions on Robotics*, 31(6):1294–1310, 7 2015. doi: 10.1109/TRO.2015.2479878.

[14] A. E. Ross and M. Pavone. A real-time framework for kinodynamic planning in dynamic environments with application to quadrotor obstacle avoidance. *Robotics and Autonomous Systems*, 115:174–193, 2019.

[15] B. Sakcak, L. Bascetta, G. Ferretti, and M. Prandini. Sampling-based optimal kinodynamic planning with motion primitives. *Autonomous Robots*, 43(7):1715–1732, 2019. doi: https://doi.org/10.1007/s10514-019-09830-x.

[16] B. Sakçak. *Optimal Kynodynamic Planning for Autonomous Vehicle*. PhD thesis, Politecnico di Milano, 2017.

[17] Y. Wang, J. O'Keeffe, Q. Qian, and D. Boyle. Kinojgm: A framework for efficient and accurate quadrotor trajectory generation and tracking in dynamic environments. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 11036–11043. IEEE, 5 2022. doi: 10.1109/ICRA46639.2022.9812352.

List of Figures

1.1 Motion planning problem solved using the Dijkstra algorithm (a) and the A* algorithm (b). The blue dot on the right represents the starting position, whereas the one on the left indicates the final position. The red curve is the optimal trajectory. The green dots are the expanded nodes. 10
1.2 Quadrotor model and reference frames. 18

2.1 Piece-wise polynomial trajectory. 24

3.1 Generate 2D motion primitives for an acceleration-controlled system (a) and a jerk-controlled system (b) starting from the initial state $(x_i, y_i) = (0, 0)$. The red dot denotes the initial state position, whereas the blue dots represent the final state positions (x_f, y_f). The black lines depict the computed trajectories resulting from various control inputs. The initial velocity and acceleration are $v_i = [1, 0]^T$ and $a_i = [0, -1]^T$(applicable to figure (b) only). 34
3.2 Graph generated from a starting point (red dot) located at coordinates $(x_i, y_i) = (1, 1)$ to a goal point (red dot) positioned at coordinates $(x_f, y_f) = (5, -7)$. 36
3.3 Simulation result for acceleration-controlled system. The blue dot on the right represents the starting position, whereas the one on the left indicates the final position. The red curve is the optimal trajectory. The green dots are the expanded nodes. 41
3.4 Velocity profile for the optimal trajectory in Figure 3.3. The blue and cyan lines depict the velocities along the x- and y-axes. The red lines show the velocity limits. 42
3.5 Actuation (acceleration) profile for the optimal trajectory in Figure 3.3. The blue and cyan lines depict the accelerations along the x- and y-axes. The red lines show the actuation limits. 42

3.6 Simulation result for jerk-controlled system. The blue dot on the right represents the starting position, whereas the one on the left indicates the final position. The red curve is the optimal trajectory. The green dots are the expanded nodes. 44
3.7 Velocity profile for the optimal trajectory in Figure 3.6. The blue and cyan lines depict the velocities along the x- and y-axes. The red lines show the velocity limits. 45
3.8 Acceleration profile for the optimal trajectory in Figure 3.6. The blue and cyan lines depict the accelerations along the x- and y-axes. The red lines show the acceleration limits. 45
3.9 Actuation (jerk) profile for the optimal trajectory in Figure 3.6. The blue and cyan lines represent the jerks along the x- and y-axes. The red lines show the actuation limits. 46

4.1 Subset (a) and complete set (b) of motion primitives computed for a 4D state space (x, y, v_x, v_y). Red dot corresponds to the initial state position (x_i, y_i), while blue dots correspond to the final state positions (x_f, y_f). The black lines represent the computed trajectories for different final velocities v_x and v_y. 56
4.2 Steps to build a complete motion primitives database (c) starting from a smaller set of reference trajectories (a) through mirror reflection relative to the x and y axes for all the trajectories within the reference set (b). 57
4.3 Steps involved in the geometric translation. 60
4.4 Simulation result for acceleration-controlled system. The blue dot on the right represents the starting position, while the one on the left indicates the final position. The red curve is the optimal trajectory. 64
4.5 Velocity profile for the optimal trajectory in Figure 4.4. Blue and cyan lines depict the velocities along the x- and y-axes. Magenta curve represents the combined velocity. Red lines show the velocity limits. 64
4.6 Actuation (acceleration) profile for the optimal trajectory in Figure 4.4. Blue and cyan lines depict the accelerations along the x- and y-axes. Magenta curve represents the combined acceleration. Red lines show the actuation limits. 65
4.7 Simulation result for jerk-controlled system. The blue dot on the right represents the starting position, while the one on the left indicates the final position. The red curve is the optimal trajectory. 69

4.8 Velocity profile for the optimal trajectory in Figure 4.7. Blue and cyan lines depict the velocities along the x- and y-axes. Magenta curve represents the combined velocity. Red lines show the velocity limits. 69
4.9 Acceleration profile for the optimal trajectory in Figure 4.7. Blue and cyan lines depict the accelerations along the x- and y-axes. Magenta curve represents the combined acceleration. Red lines show the acceleration limits. 70
4.10 Actuation (jerk) profile for the optimal trajectory in Figure 4.7. Blue and cyan lines depict the jerks along the x- and y-axes. Magenta curve represents the combined jerk. Red lines show the actuation limits. 70

5.1 Quadrotor platform used for the experiments. 73
5.2 Virtual indoor environment used for the experiments. 74
5.3 Indoor environment used for the experiments. 76
5.4 Planned trajectory for Simulation 1. The color bar on the right indicates the velocity along the trajectory. 77
5.5 Position profile and position error for Simulation 1. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities. 78
5.6 Velocity profile and velocity error for Simulation 1. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities. 79
5.7 Motors thrust for Simulation 1. 80
5.8 Position setpoint (dashed curve) and corresponding trajectory tracked by the quadrotor (solid curve) in real-world Experiment 1. 81
5.9 Profile of the position setpoint (dashed curve) and profile of the position tracked by the quadrotor (solid curve) in real-world Experiment 1. 82
5.10 Profile of the velocity setpoint (dashed curve) and profile of the velocity tracked by the quadrotor (solid curve) in real-world Experiment 1. 83
5.11 Planned trajectory for Simulation 2. The colorbar on the right indicates the velocity along the trajectory. 84
5.12 Position profile and position error for Simulation 2. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities. 85
5.13 Velocity profile and velocity error for Simulation 2. The dashed curve represents the setpoint, while the plain curve depicts the actual quantities. 86
5.14 Motors thrust for Simulation 2. 87
5.15 Position setpoint (dashed curve) and corresponding trajectory tracked by the quadrotor (solid curve) in real-world Experiment 2. 88

5.16 Profile of the position setpoint (dashed curve) and profile of the position tracked by the quadrotor (solid curve) in real-world Experiment 2. 89
5.17 Profile of the velocity setpoint (dashed curve) and profile of the velocity tracked by the quadrotor (solid curve) in real-world Experiment 2. 89

List of Tables

1.1 Notation employed for each vertex v in the graph 14
1.2 Function employed to manage the closed list. 14
1.3 Functions employed to manage the priority queue. 16

3.1 Relevant results for the acceleration-controlled system. 41
3.2 Relevant results for the jerk-controlled system. 44

4.1 Database details for the acceleration-controlled system. 63
4.2 Motion planning results for the acceleration-controlled system. 63
4.3 Database details for the jerk-controlled system. 68
4.4 Motion planning results for the jerk-controlled system 68