上海交通大学硕士学位论文

A UAV Motion Planning Framework with Global and Local (Re-)Planning for 3D Environments

硕 士 研 究 生:	Rémy Hidra
学　　　　号:	119034990010
导　　　　师:	Prof. Wenxian Yu
副　导　师:	Prof. DongYing Li, Prof. DanPing Zou, Prof. Olivier Simonin
申 请 学 位:	专业型硕士
学　　　　科:	电子与通信工程
所 在 单 位:	电子信息与电气工程学院
答 辩 日 期:	2021 年 7 月 9 日
授予学位单位:	上海交通大学

Dissertation Submitted to Shanghai Jiao Tong University
for the Degree of Master

A UAV MOTION PLANNING FRAMEWORK WITH GLOBAL AND LOCAL (RE-)PLANNING FOR 3D ENVIRONMENTS

Candidate:	Rémy Hidra
Student ID:	119034990010
Supervisor:	Prof. Wenxian Yu
Assistant Supervisor:	Prof. DongYing Li, Prof. Dan-Ping Zou, Prof. Olivier Simonin
Academic Degree Applied for:	Professional Master of Engineering
Speciality:	Electronic and Communication Engineering
Affiliation:	School of Electronic Information and Electrical Engineering of SJTU
Date of Defence:	July 9, 2021
Degree-Conferring-Institution:	Shanghai Jiao Tong University

Abstract

Unmanned Aerial Vehicles are rapidly growing in popularity, with applications in various fields. This versatility creates the need for a flexible navigation solution, with different constraints, like high-speed aggressive control, operating in unknown or dynamic environment and hardware limitations like low memory and processing unit for on-board computations. A lot of state-of-the-art works introduce motion planning frameworks without considering the entire planning pipeline. This create sub-optimal behaviors and reduce the versatility of a solution. Additionally, very few works present a flexible and adaptable benchmarking method to evaluate a framework against the state-of-the-art.

In this thesis, we present multiple novel strategies for global and local planning. On the contrary of most works, we especially focus on building a re-planning global planner, using the Phi* path planning algorithm. Thus, our framework is robust against large map modifications, by implementing online global re-planning. We also describe a new method for local planning, using a motion primitive fast generation and a sequential optimization. This allows our planner to provide a flexible response in real-time, which is ideal in partially known environments. Then, we introduce our implementation architecture, and we discuss various interesting details, not generally relevant in more specialized works. Finally, our pipeline is evaluated against a re-usable benchmark with reproducible results. We believe other authors could use this to compare future methods more easily. Our work is also presented in a re-planning scenario, to simulate its behavior in a partially known environment, with a limited vision field around the UAV.

Our method provide a safe conservative approach for motion planning in partially known environment in large scale maps, where efficiency and computational speed is the issue.

Keywords— UAV, Motion planning, Global-local planner, Re-planning

Acknowledgements

I would like to thank my supervisors at the Shanghai Jiao Tong University: Prof. Daniele Sartori, Prof. Danping Zou and Prof. Dongying Li. They guided me with support during theses two years of hard work. Because of the unfortunate pandemic, I was helped a lot by Prof. Olivier Simonin, from the CITI Lab of Lyon. He took the time to advise me and provided me with lab equipment to experiment and improve my results.

Moreover, I want to thank the administration of the INSA Lyon and SJTU, who collaborated to make my double degree as smooth as possible.

I would not have come this far without my friend Claire Laverne, who helped me when times were rough and uncertain.

Lastly, I want to thank my parents, who always supported me, and without whom I would not be where I am now.

Contents

List of Figures

Chapter 1

Introduction

In the last decades, we have seen some tremendous progress in many fields of robotics. Thanks to a strong increase in the processing power of computer chips, improved bandwidth in wireless communications, and the rise of artificial intelligence, we witnessed more and more innovative applications around robotics, using real-time sensor processing and computer vision. On the one hand, while Unmanned Aerial Vehicles (UAVs) are slowly entering the consumers market, with interesting applications in aerial photography and acrobatic demonstrations, all require some sort of human pilot. On the other hand, autonomous driving is a very active area of research, with real-world applications progressively entering the consumers market. In this work, we are interested in autonomous UAV navigation. We define this type of navigation as almost completely self-governing, without needing a human interaction for basic movements. Similarly to autonomous driving, the operator should only specify basic objectives, for instance the goal location. Such operations should not depend on the type of the UAV. It could be generalized to swarms of micro UAVs (MAVs), which could coordinate themselves autonomously.

Impressive possible autonomous UAVs applications range from search and rescue missions [1], automated last mile fast delivery, cinematography [2], and even animated 3D display [3]. Each of these applications put the UAV in various environments, where communication between other mobile robots or a ground station may be necessary for coordination. In many cases, the robot does not have a access to a ground truth representation of the world around it. It should also be prepared to evolve in a dynamic or unknown environment, where recent mapping data will be quickly outdated, and its planning strategy should evolve accordingly. This issue of dynamic environments has been widely studied in the field of autonomous driving [4]. However, UAVs may evolve in denser 3D environments, with a smaller processing unit. This shows the need for a powerful framework, capable of performing an efficient motion planning in 3D at a high frequency while utilizing the most amount of mapping data as possible.

Most of recent work is splitting the problem into the mapping and motion planning problems. The mapping problem, commonly referred to as Simultaneous Localization and Mapping (SLAM), is generally solved using computer vision techniques, machine learning, estimators and advanced 3D data representations [4]. Some recent papers combine mapping and planning in a perception aware planning model [5] [6]. Motion planning is an active area of research, bringing together graph search, optimization methods, probabilistic samplers and geometric navigation [7].

Introduced by Richter and Charles in [8], a typical framework for motion planning in robotics is to split the problem in two parts. First, a global planner quickly builds a path on a large map between a starting point and a goal point often described by the operator. Since it is manipulating a large volume, some concessions have to be done to keep a processing time relatively low. Additionally, it is not desirable to plan the robot behavior precisely in a long time in the future, especially in a dynamic environment, where the path could be blocked when the UAV reaches it. Usually, planning will only be done on a geometric level, only considering static obstacle avoidance and ignoring dynamic constraints of the UAV. Such algorithms have been widely used for many years to solve various problems. Moreover, incremental versions of algorithm may be used, in order to re-use previously computed and still relevant data, in a re-planning scenario.

Second, a local planner will take as input the UAV current position and the closest waypoint in the path computed by the global planner. It will compute a feasible trajectory, which is defined as a time-parameterized path. It should consider the higher order derivative constraints of the robot and the dynamic environment. In other words, the local planner outputs a time-parameterized curve, usually in three or four dimensions, continuous up to a certain order. According to Mellinger and Kumar in [9], who prove the differential flatness of a quadrotor system, we can parameterize the trajectory of a UAV using four equations instead of six: the three positional coordinates (x, y, z) and the yaw angle θ. The local planner should ideally consider obstacle avoidance in the dynamic environment, with moving objects and the sensors uncertainty. Depending on the speed of the UAV, the local planner needs to output a safe feasible trajectory at a high frequency, fast enough for the UAV to avoid new obstacles in real-time.

The environment may be fully known, or partially known. It may also be static, as in the mapping data, once acquired, will always be correct, or dynamic, where obstacles will move potentially along the planned path of the UAV. This creates the need for an efficient mapping data structure, adapted for the problem statement. Certain works introduce multiple structures for multiple tasks in their navigation framework [10]. Additionally, in a partially known or dynamic environment, we need a motion planner capable of interpreting small amounts of new data about its environment, in an online way. Its response should be as close as possible to real-time, to avoid undesirable jerk and a potential crash. Therefore, a lot of authors work in building re-planning algorithms, which are more efficient in finding a valid trajectory in incremental steps, progressively as each new information about the environment appears.

1.1 Problem statement

Formally, we define the problem as the following. The configuration space C represents all the possible configurations which can be taken by the robot, considering all its dynamic constraints. With n degrees of freedom, the configuration space is a manifold of dimension n. Thanks to the robot's sensors, we can detect obstacles in the environment, defined as $C_{obs} \subseteq C$. Then, we can derive the free space, accessible by the robot, as $C_{free} \subseteq C \setminus C_{obs}$. Depending on the problem formulation, in partially known environments, we usually do not have access to all the environment information. In that case, we may want to introduce the concept of unknown space $C_{unknown} = C \setminus (C_{obs} \cup C_{free})$. A lot of work ignore the concept of unknown space and simply take an optimistic approach by considering it as free space, or a conservative approach by treating it as an obstacle. The motion planning problem can then be formulated as finding a continuous path $\tau : [0, 1] \to C_{free}$ such that $\tau(0) = q_{start}$ and $\tau(1) = q_{goal}$ with $q_{start}, q_{goal} \in C_{free}$ respectively the user defined starting and goal configuration.

In this thesis, we will consider the start q_{start} and goal q_{goal} already known and considered safe to takeoff or land. Depending on the experiment, the map of the environment is considered known or partially known. Although we will not consider dynamic environments, where obstacles may quickly move, we are interested in building a framework for partially known environments, where the robot will acquire progressively new knowledge about its environment and update its path planning accordingly. This method could model how the UAV may use external mapping data, e.g. from satellites, robots or other remote sensors. Additionally, the path planning problem is considered in a continuous 3D world, in continuous time. Our objective is to implement a solution as functional as possible, thus we will focus on improving time performances, and model a simulation as realistic as possible.

1.2 Contributions

In this work, not only do we introduce a new motion planning framework, we are also presenting our methodology to select each method. We consider multiple new strategies for a full motion planning solution, examine each one independently, and then evaluate our entire pipeline to locate its weaknesses and improve it. By focusing on all the components of our algorithm, we believe that we are more capable of building smarter interactions, improving data usage and increasing the overall quality of our framework.

First, most global planning works are very theoretical and are not interested in UAV applications, which makes them not adapted to UAV 3D global re-planning, especially when introducing real-time and scalability constraints. Second, even if plenty of local planners have been successfully implemented in real-world setups, in cluttered dynamic environments, most are still vulnerable to large scale map modifications, which would require real-time global re-planning. Generally, authors do not implement a strong relationship between the global and local planner. Finally, very few works evaluate their motion planning framework in a large base of environments, as implementation setups can vary, which makes it hard to find a common simulation platform.

The contributions of this work are:

- a new global planning algorithm, Phi* 3D, with a strong focus on fast efficient 3D re-planning in a partially known environment;
- an evaluation of our global planning algorithm, especially against more common solutions in most UAV frameworks;
- a novel local planner solution, based on motion primitive fast generation and a sequential optimization;
- experimental data showing the superiority of our local planner strategy compared to more classic motion primitive planners;
- a full description of our pipeline software architecture, fully implemented in simulation and easily adaptable to a real-world implementation, with an in-depth a various minor details commonly not introduced in other works;
- an evaluation of our full pipeline, in multiple reproducible small 3D environments, ideal to efficiently compare to other UAV planning works;
- an evaluation of our full pipeline against a re-planning scenario, in a large scale 3D environment, with an artificial limited vision field around the UAV.

The rest of this thesis is organized as follow. In chapter 2, we introduce common methods used in other works to solve the motion planning problem. Then, in chapter 3, we present our novel global planner based on our Phi* 3D algorithm and we evaluate its performances against another path finding algorithm. We describe our local planner in chapter 4, where we compare it with other common motion primitive based methods. In chapter 5, we introduce our full implementation architecture, while discussing some interesting details. Finally, in chapter 6, we present some results to evaluate the performances of our entire pipeline in a benchmark re-usable in other works for comparison. We finish this thesis by discussing our solution in chapter 7.

Chapter 2

Related works

In this chapter, we introduce various tools and systems used to solve the motion planning problem. We present in section 2.1 mapping structures with different advantages. Then, in section 2.2, we describe simple path finding techniques, commonly used as geometric 2D or 3D global planner. Finally, in section 2.3, we explain trajectory generation algorithms through local planning implementations.

2.1 Mapping

In motion planning, mapping systems are structures and algorithms used to process external sensors data and accurately represent the environment around the robot. The noisy data processing is commonly done by SLAM algorithms, which not in the scope of this work. With clean map data, we must use a good map structure representation. Depending on the application, some structures present more advantages than others.

A simple method to discretize 3D space is to use *voxels* [12] and segment the environment in a fixed 3D grid. This method is simple to implement on low-level hardware and is in random access. Thus, it provides a very fast reading and writing duration with direct addressing. However, this structure is also very rigid. It is highly inefficient to insert additional data without discarding already recorded voxels. Finally, the granularity of the grid is determined by its fixed resolution which exponentially increases its memory consumption. Voxel grid structures are ideal for small environments of fixed size. In [11],

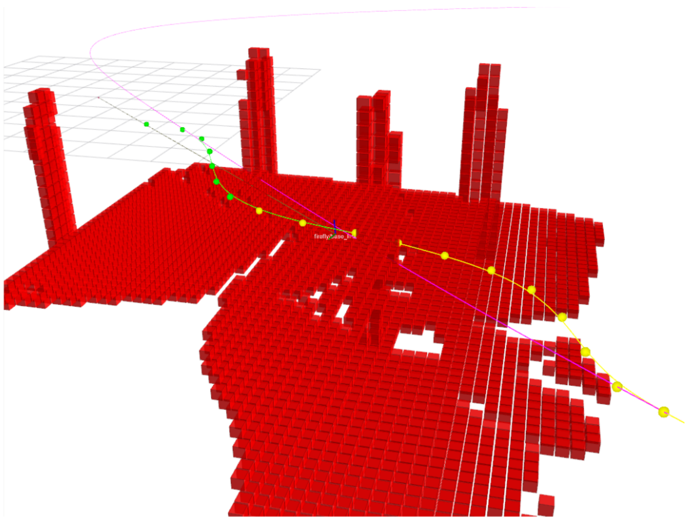


Figure 2.1: In [11], Usenko et al. structure the space in a voxel sliding window, to provide efficient direct access, while maintaining scalability.

the authors introduces a sliding voxel map, as shown in figure 2.1. Because of its structure it allows for a fast access and helps to discard irrelevant mapping data for current local planning.

Octomaps and *Octrees* [13] are a versatile solution, where the environment is divided in recursive nodes and each node is the parent of the eight sub-regions inside it. The graph structure helps to quickly insert new data in the map. Moreover, checking if a point is occupied can be done efficiently by discarding automatically multiple levels of depth in the graph. This allows to keep a good resolution on obstacles, without sacrificing memory. Octomaps are a good compromise between data access performances and memory consumption.

Kd-trees [14] are another mapping structure commonly used with UAV path planning [15]. The advantages are the very fast data transfer, at nearly the sensor rate, and the efficient nearest neighbor query. However, it performs poorly in random data access, thus it is computationally expensive for search based planners because of all the collision checks.

Finally, a useful tool is to extract the *Euclidean Distance Transform* (EDT) of a mapping representation [16]. The EDT of a specific point in space corresponds to its distance to the nearest obstacle. In other words, $EDT(x \in C_{obs}) = 0$ and $0 < EDT(x \in C_{free}) \leqslant L$, where L is a max distance parameter usually fixed in the implementation for performance issues. EDT has a fast direct access, but generally needs to be generated from an already existing mapping representation and is computationally expensive. EDT are a common tool used with Octomaps in robotics.

2.2 Global path planning

Plenty of works already exist in global planning. Because of a lot of simplifications, the problem can be solved using simple geometric path planning algorithms. In other words, during that phase, we often will not consider dynamic constraints of the UAV. We can usually split classical methods between sampling based methods, graph search based methods and artificial intelligence based methods [7] [17].

2.2.1 Sampling based methods

Sampling or probabilistic based methods are popular path finding methods, because they sample a continuous n-dimensional space. Since it is not necessary to discretize the environment beforehand, with the right sampling, it is possible to build much shorter paths than with a more naive grid discretization. The *Probabilistic Road Map* (PRM) [18] and *Rapidly exploring Random Tree* (RRT) [19] algorithms are two major milestones in the sampling path finding field. PRM is sampling points in the space, which are connected to nearby graph nodes. Then, a graph search algorithm, like Dijkstra or A* is applied to find the shortest path. RRT is more efficient, as it is building a tree rooted at the starting configuration q_{start}. It creates and connects a node to the tree only if it belongs to the free space and if it is in line of sight with other nodes. When the graph is able to be connected to the goal configuration q_{goal} a straight line connection is made and a path can be instantly retrieved, by going backward through the parent of each node. An evolution of RRT is *RRT** [20], which introduces rerouting the graph, with the nodes near a sample point (fig. 2.3, middle). This does not increase a lot the computational time, but makes the algorithm asymptotically optimal. In other words, with an infinite amount of samples in the free configuration space, the algorithm would retrieve the optimal shortest path if it exists, as shown in figure 2.3. Other variations worth mentioning are *RRG* [20], *RRT*-Smart* [21] and *Informed-RRT** [22]. Some functional UAV motion planning frameworks uses RRT* [8] and Informed-RRT* [23].

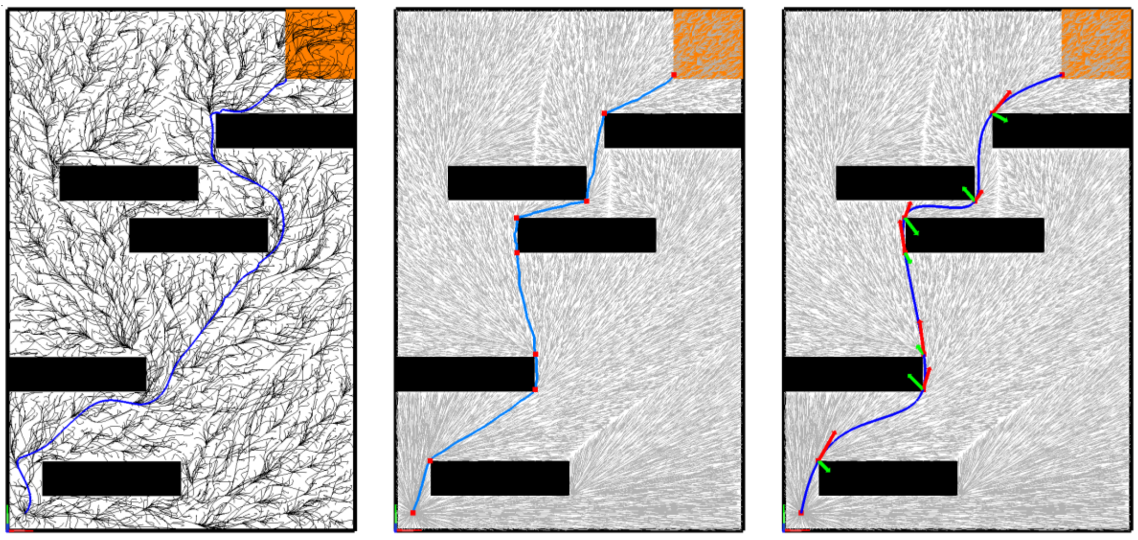


Figure 2.2: In the work of Richter et al. [8], application of a RRT* for UAV offline navigation in 2D. The orange square is the global goal. From left to right: Inefficient planning using RRT* with a polynomial steer function; Fast straight line RRT* planning, but infeasible by a UAV; Improvement of the previous solution using joint fast polynomial optimization to ensure dynamic feasibility.

The drawback of such a method is obviously the high computational cost when the number of samples increases, especially in higher dimensional spaces, where more samples are needed to attain a satisfying result. Because of this, in a lot of works, global planners run offline and are not made for real-time applications in dynamic environments. This is not ideal in exploration mission or environment where large scale changes may happen, because the local planner is generally not designed to recompute important path deviation.

Artificial Potential Field (APF) [24] uses a potential field model, where obstacles are repulsive. The start point and the obstacles have the highest potential, the goal has the lowest. By minimizing the potential from the start, we can find a short path to the goal. This method does not discretize the space and can find a continuous path, which can improve the feasibility. However, it can be expensive to compute, and the optimization may get stuck in a local minimum.

Most sampling based methods are asymptotically optimal, but they tend to converge slowly. Especially in 3D, the sampling space becomes exponentially larger, which makes it not suitable for an online path planning algorithm. Additionally, the classic RRT algorithm cannot quickly adapt to dynamic changes in the environment. Recent work, introduce *RRTx* [25], an RRT* based global planner, able to react to instantaneous changes in the environment. Re-planning with sampling based path finding algorithm is an active area of research for UAV motion planning [26].

2.2.2 Graph search based methods

Graph search based methods are another type of global path planning algorithm. Usually, the continuous environment has to be discretized as a graph using, for example, a *n-dimensional grid*, a *visibility graph* [27], or a *Voronoi graph* [28].

Depth First Search (DFS) and *Breadth-First Search* (BFS) are common graph search algorithms [29]. They are very classic, easy to implement, and often used in most computer science graph search problems. BFS is optimal if all edges of the graph have the same cost. DFS does not guarantee optimality but is faster. Both DFS and BFS are too slow to be applied in an online context.

Dijkstra [29] is a common fast algorithm applied for shortest path finding applications.

It is based on exploring at each iteration the un-visited node with the shortest path. Then, a generalization of Dijkstra is the *A** algorithm [30]. It introduces a heuristic cost function $h(s)$ which tries to evaluate a potential cost to the goal for the node s. A* is optimal depending on the choice of the heuristic, with the ideal heuristic being the actual length of the shortest path between s and s_{goal}. Dijkstra is equivalent to A* with a null heuristic $h(s) = 0$. A* is an efficient, greedy algorithm widely used in various areas of robotics, video-games and computer graphics. Multiple variations can be mentioned, like *Anytime Repairing A** (ARA*) [31], *Jump Point Search* (JPS) [32] and *Hybrid A** [33].

However, since those methods need to be applied on a discrete graph, they are not widely popular among the robotic community, as they are not allowing turns at any angle. For example, a path on a grid will only allow turns at multiples of 45°. Any-angle solutions may be found using additional post processing techniques, like filtering all unnecessary nodes between two nodes which have a line of sight relationship. Some works also introduce any-angle variations of the A* algorithm. *Theta** [34] introduces the concept of linking a node, not to its neighbor, but to the parent of its neighbor if it is in line of sight. If the map is discretized as a grid, the triangle inequality guarantees the Theta* path will be shorter than traditional A*.

Because of the predefined graph they are built on, graph search algorithms can be extended for re-planning scenarios. The *D** algorithm is a popular incremental variation of A*. *Field D** [35], *D* Lite* [36] and *Focussed D** [37] are any-angle incremental variations, which use linear interpolation to approximate a straight line path when possible.

2.2.3 Artificial intelligence

More recent works introduced new path planning algorithms based on artificial intelligence and machine learning [17]. In [38], authors implement a Genetic Algorithm (GA) based method, using *Simulated Anneal Arithmetic* (SAA) and *Ant Colony Optimization* (ACO), both popular genetic algorithms. In a classic genetic algorithm, a population of potential path planning solutions is first generated using a set of random features [39]. Using an objective function (or fitness function), each candidate is evaluated, so that only the best features are kept. Mutations are done to mix features and build a population performing better in the next iteration. GA showed great performances in various works in the last few years [39]. Additionally, GA is robust against local minimum, as new random variations are introduced at each mutation, allowing to focus on the entire range of possible solutions.

ACO methods are simulating a group of ants, leaving pheromone on their path and attracting more ants. Iteratively, as the ants starts to follow the same path, the algorithm converges towards a path solution [40]. While the optimization is quite slow, the algorithm can be easily parallelized, and allow for great application with UAV motion planning [39].

Artificial Neural Networks (ANN) are a common machine learning method, based on how information travels in interconnected human neurons. With a GPU architecture, because their computations are parallel, ANN can perform quite fast. They are used in UAV motion planning in order to approximate complex system dynamics and functions, thus reducing the complexity of non linear optimization problems [41].

Artificial intelligence based methods are quite new in the field of motion planning. A lot of solutions are not mature enough to be implemented in a real-time environment. However, most are parallelizable, thus ideal on GPU embedded hardware or for offline computation. Additionally, some more versatile algorithms can be scaled to a multi-UAVs problem.

2.3 Local path planning

Once a geometrical path has been computed, we need a local planner to refine it into a feasible trajectory. In other words, we must generate a continuous time-parameterized curve, which considers the dynamical constraints of the UAV and the moving obstacles in the surroundings. Because of the collision avoidance feature, it is generally optimal if the local planner can run at the same frequency as the UAV sensors. Optimization methods are very popular to generate trajectories [7]. After setting the minimization problem, they rely on optimizing specific curve parameters. Transformations can be applied to the problem to make it solvable. Then, we can use a standard mathematical optimization method to solve the problem and obtain the trajectory. Depending on the type of curve, the setting of the problem and the map data structure, we may want to either use a hard constrained or a soft constrained optimization. However, a major drawback of optimization is the computation cost. An alternative is to use a motion primitive based local planner.

2.3.1 Optimization based

In [8] and [9], authors introduce a hard constrained polynomial curve optimization method. As said above, we can reduce the system of a UAV as a system of 3D coordinates and a yaw angle [9]. In other words, a trajectory can be defined as four polynomial curves. If the curve is too long, the computational cost will increase exponentially. Instead, we can define a trajectory by a piece-wise polynomial curve. This problem can be solved using a classic quadratic programming (QP) optimization method [9]. We can set hard constraints in the QP for the maximum velocity and acceleration of the robot. Finally, we can control the continuity of the path in higher derivatives by using higher order polynomials. High order continuity is needed to ensure no sudden accelerations, which are not practically feasible and would degrade the execution of the trajectory. However, the optimization does not feature a collision avoidance system. [8] implements a collision check after each local planner execution. If a collision happened, a new waypoint is added in the middle of the previous segment and the local planner is executed again. This is a simple technique, but it highly increases the computation cost, as we have no guarantee over the number of executions of the local planner.

The *CHOMP* algorithm [42] is a useful tool for soft constrained trajectory optimization. It starts from a straight line trajectory, then, using gradient descent, it optimizes to find a collision free smooth path. It is efficient enough to run in real-time. However, the optimization outputs a set of discrete waypoints, which is not ideal to consider all the dynamical constraints.

In [23], we see an implementation in a real UAV of the method seen in [8], as a soft constrained optimization. It generate a continuous trajectory, based on polynomial curves, to respect dynamical constraints. This also avoids numerical differentiation errors. To solve the collision avoidance problem, instead of recomputing the entire optimization, the authors integrate a collision avoidance condition in the unconstrained quadratic programming problem. They add a collision cost function, which is the integral over the line segment of a potential function. Since the mapping representation of obstacles is discrete, they can sample the integral at discrete points along the segment. Inspired by the CHOMP algorithm, the potential function is defined as the inverse of a Euclidean Distance Transform (EDT) [42]. It is null in a wide free space, high inside an obstacle, and rising at a close distance to an obstacle. This function can be tuned to set the clearance distance to avoid obstacles. Thanks to this definition, the trajectory converges to a safe distance from each obstacle. This algorithm can be implemented incrementally in real-time and provided great results in simulation and in real-world UAV experiments.

Because of the work by Richter and Charles [8], optimizing continuous polynomial

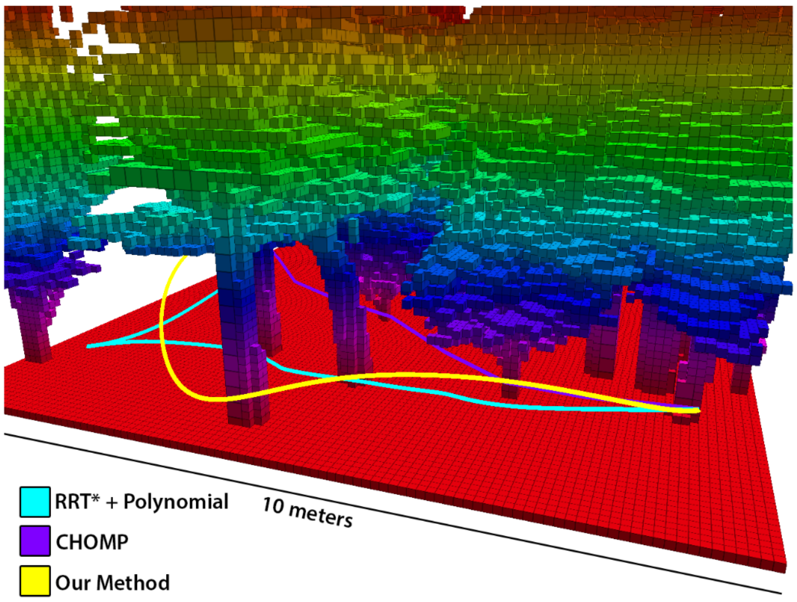


Figure 2.3: In the work of Oleynikova et al. [23], they compare their local polynomial optimization method to older methods, like the CHOMP algorithm [42] and an RRT* path smoothed with polynomial optimization [8], in a small 3D environment.

curves is popular in the research community. In [11], the authors explore a new method for trajectory generation. They introduce a B-splines trajectory optimization technique and a local 3D circular buffer for fast mapping data storage and access. The B-spline offers interesting properties. It only needs to be expanded up to the fifth order to ensure high derivative continuity, which makes it more efficient than polynomial curves. It is also easier to turn the problem into an unconstrained optimization, which is faster to solve. In a B-spline, the curve does not necessarily go through all the control points. Thus, it is harder to add geometrical constraints to the path, for instance, if the global planner ensures a lot of collision avoidance. This technique has been implemented on a real-world UAV, for local obstacle avoidance with only sensors data. It performs well and achieve seamless re-planning in an unknown environment.

2.3.2 Motion primitive library

Another method for local planning is to use a motion primitive library. [15] explores this in an exploration framework implementation and [43] and [44] implement this in classic 3D planners. Instead of using a computationally expensive trajectory optimization technique as a local planner, authors used a finite motion primitive library. The primitives are forward-arcs based curves, at different final angles and final velocities. First, the library is regenerated for the UAV starting position and velocity. Then, a cost is given to each primitive, given the collision probability and the closeness to the local goal. Once the lowest cost primitive is selected, the UAV can execute the path. This algorithm is very fast, because the cost function has to be run on a predictable number of trajectories, but the algorithm cannot invent new paths out of the library. In practice however, authors report that is not a problem for good motion planning. The algorithm is not implemented in 3D, which can make it inefficient when operating in complex multi-floors environments. Overall, not a lot of work has been done implementing a motion primitive based local planner in a UAV motion planning setup. This is mainly due to a lot of pre-existing efficient works in optimization based planner.

Most local planner are not reliable in an indoor unknown cluttered environment. This

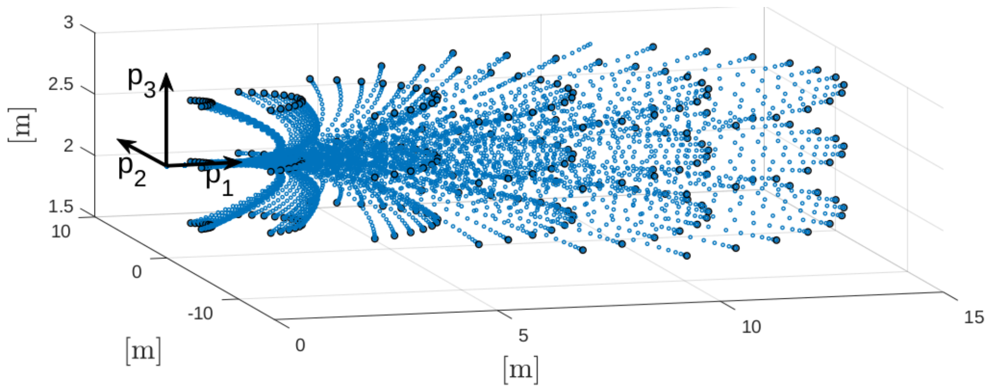


Figure 2.4: From Ryll et al. [43], example of a motion primitive library, sampled over a range of final heading angles, final velocities and final altitudes. All primitives are dynamically feasible for the UAV.

is due to the small re-planning horizon of the local planner, which is not able to face large unexpected changes in the map. For example, if the UAV faces a dead-end situation, the local planner will not find a feasible solution. In that case, the global planner need to be executed again, by re-using all the previous environment data gathered before.

Chapter 3

Proposed global planner solution

In this chapter, we present our solution to the UAV global motion planning problem in a partially known 3D environment. This chapter is organized as follow. In section 3.1, we describe our global planner implementation, based on the any-angle Phi* path finding algorithm. And, in section 3.2, we evaluate our algorithm against Theta*, another any-angle solution. We present the architecture of the planner in figure 3.1.

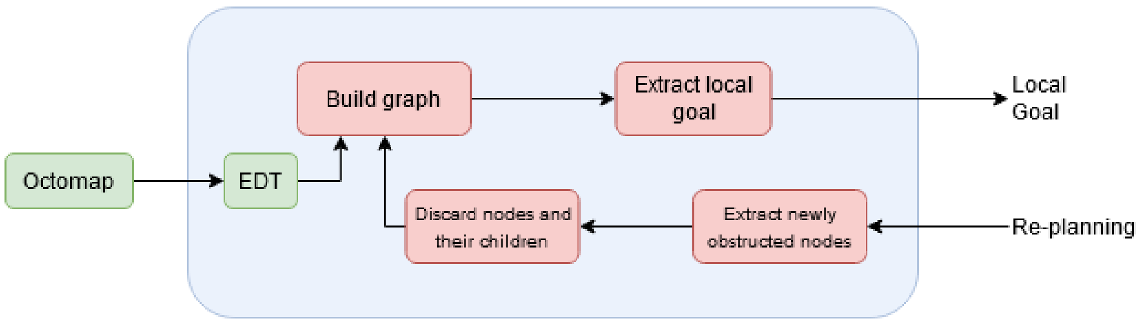
3.1 Phi* 3D Global planner

As explained in section 2.2, the goal of a global planner is to quickly find a geometric path between the robot and the main objective. In order to be computationally cheap, it can ignore dynamic constraints of the UAV. In a dynamic and partially known environment, it is also important to use incremental re-planning. Incremental algorithms allow to re-use previously computed data. Even if they may be slower than classic path planners, they can scale more easily to larger maps.

In order to combine an incremental solution with an any-angle algorithm, which provides shorter paths, we decided to examine Phi* [45], a fairly new any-angle incremental path finding algorithm based on Theta* [34] and developed by Nash and Koenig. Since the original implementation is only in 2D, we extended its definition to be usable in a 3-dimensional environment.

3.1.1 Theta* 2D

Theta* is a simple extension of A* in an any-angle algorithm [34]. Each node s has a global parent ($parent(s)$), and to explain Phi*, we must introduce the notion of local parent ($local(s)$) [45]. In classic A*, we have $parent(s) = local(s)$, the global path is equivalent to the local path. With Theta*, before linking a node s' to its neighbor s, as we



Octomap
EDT
Build graph
Extract local goal
Local Goal
Discard nodes and their children
Extract newly obstructed nodes
Re-planning

Figure 3.1: Architecture of our global planner.

would in A*, we first must check if the global parent of its neighbor $parent(s)$ is in line of sight with the current node s'.

- If we do not have a line of sight relation (*path 1* in the work of [45]), we simply link s' to its neighbor, so $parent(s') = local(s') = s$.
- If we have a line of sight relation (*path 2*), we can link s' to $parent(s)$, in order to have a straight line path when possible. We have $parent(s') = parent(s)$, and we can keep track of the underlying structure of the graph using the local path $local(s') = s$.

When the algorithm stops, we can backtrack from the goal, using the global parent of each node, to build the geometrical path solution.

In a re-planning setup, when receiving new obstacles, it is very hard to infer a behavior on a Theta* generated path. The quickest technique is to detect an obstruction in the path, discard the entire graph and re-start the Theta* computation from scratch. This method is as fast as the initial Theta* computation for the same graph complexity, and does not re-use previously done computations, which can create a lot of inefficiencies, especially in large cluttered maps.

3.1.2 Phi* 2D

Phi* solves the incremental problem, by adding constraints before linking a node to its neighbor or neighbor's parent. The algorithm is fully defined as in [45] in appendix A. With Phi*, the path planning graph must always hold the property 1.

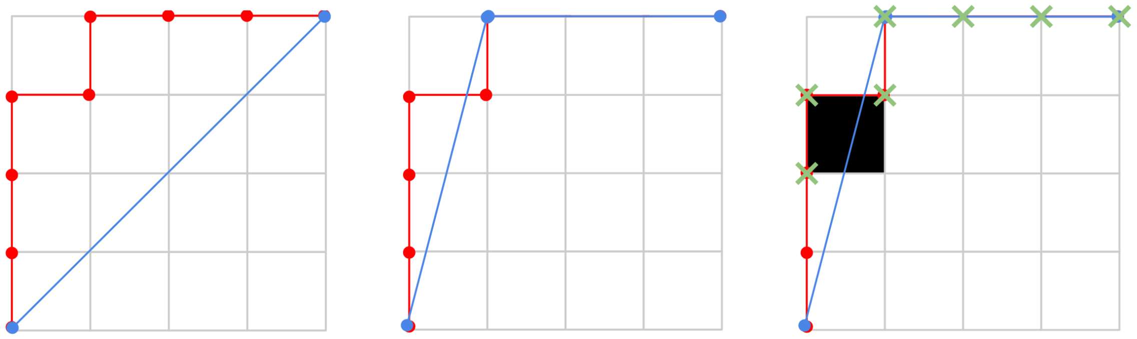
Property 1 *Any cell crossed by the global path must always have at least one corner which belongs to the local path.*

If property 1 is verified, later, during re-planning, if an obstacle is detected in a cell, we can check all of its corners. If one of them belongs to the local path, we know that it may obstruct the global path. We can discard and reset all of its children in the graph. After this cleaning step, we can guarantee that the local and global graphs are usable and not obstructed. Then, we run the path finding algorithm if we need to find a new solution to reach the goal, we explain this algorithm in figure 3.2. In this process we only discarded outdated graph data and we kept previously computed nodes which are still relevant. Compared to Theta*, Phi* is more efficient. When re-planning in 2D, it is several orders of magnitude faster than Theta*, which has to start the algorithm from scratch again.

To achieve property 1, we must define an angle range associated with a node. Each node s should record the upper bound $ub(s)$ and the lower bound $lb(s)$ of its angle range $[lb(s), ub(s)]$. As in [45], we define $\Phi\left(s, p, s'\right)$ as the small angle $\angle(s, p, s') \in [-180°, 180°)$.[1] When expanding the node s', local children of s, if the algorithm considers a *path 2* expansion, not only must it check the line of sight between s' and $parent(s)$, it must also ensure that $\Phi\left(s', parent(s), s\right) \in [lb(s), ub(s)]$. Intuitively, $[lb(s), ub(s)]$ represent the range over which the straight line between $parent(s)$ and s can bend to still keep the property 1 verified.

In other words, if a node expands using *path 2*, it must update its angle range to account for the already existing global path constraints between it and its parent. If we assume we expanded a node s' using *path 2*, so $local(s') = s$ and $parent(s') = parent(s)$. To hold property 1, we must ensure that when expanding a new node s'' from s', we have $\Phi\left(s'', parent(s'), s'\right) \in [lb(s'), ub(s')]$. From the point of view of s', this is equivalent to

[1] We provide a Python implementation of 2D Φ in appendix C.



(a) Theta*: A simple line of sight check is necessary to ensure a link between nodes.

(b) Phi*: The global path must always be in a cell whose at least one corner belong to the local path.

(c) Phi* Incremental: Before running the path finding algorithm again, we ensure the graph only contains valid information by discarding corners of the obstacles and their children.

Figure 3.2: Example of execution of Theta* and Phi* on a 2D grid. The global path is in blue, while the local path is in red. In black, we draw a newly detected obstacle, to demonstrate the re-planning step. The nodes crossed out are discarded.

limiting the angle range of s to the angle range defined by the crossbar centered at s'. Recursively, this ensures that the crossbars, by which the global path is crossing cells are always part of the crossbars of the local path nodes. So we have the update rule described in equation 3.1, where $crossbar(s')$ is the set of all point along the crossbar of s'. [2]

$$\left[lb(s'), ub(s')\right] = [lb(s), ub(s)] \cap \left\{ \Phi\left(s', parent(s), s''\right) \middle| s'' \in crossbar(s') \right\} \tag{3.1}$$

If the line of sight or the angle range conditions are not verified, a node is expanded using *path 1*. Then, we can initialize its angle range as $[lb(s'), ub(s')] = [-45°, 45°]$.

Lastly, [45] introduces one last case, when the results of *path 1* and *path 2* is identical, that is to say, when s', s and $parent(s)$ are aligned and $\Phi\left(s', parent(s), s\right)$ is a multiple of 45°. In case of such a path tie, [45] says it is slightly better to expand using *path 1*, as it produces slightly better path lengths.

The algorithm previously described generates the path finding graph. With incremental Phi*, when a new obstacle is detected, we can simply check the corners of the newly obstructed cells. For each cells which belong to the graph, we discard it and all its children along the local path. This updates the graph in the most efficient way as possible. We can simply run the graph search algorithm again to obtain a new valid path finding solution.

3.1.3 3D Generalization

Because we want to apply our motion planning algorithm to an autonomous UAV system, it can be useful to implement our global planner in 3D. We have not found other works implementing the incremental Phi* algorithm in 3D.

The key difference between Phi* 2D and 3D, is how to maintain the property 1. In 2D, we had to make sure the global path was always crossing a crossbar centered on a node of the local path. Similarly, we extend the definition of a crossbar in 3D, as the three squares intersecting in their center the node in the three axis directions (horizontally and vertically orthogonal to $\vec{x}$ and $\vec{y}$), as presented in figure 3.3. In other words, we guarantee the property 1, if and only if the global path always cross the crossbar of a local path node.

[2]We omitted the offset of angle ranges for clarity. See appendix A for the full algorithm.

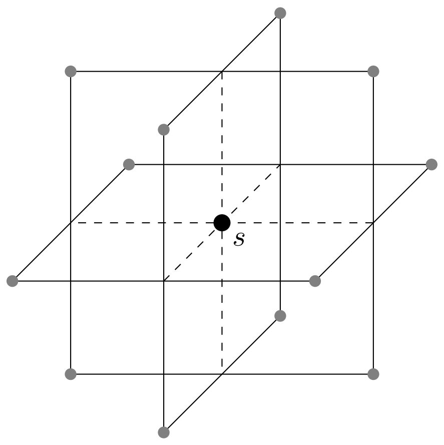


Figure 3.3: Crossbar in 3D of a node s.

To ensure this, we must also extend our definition of angle ranges. Analogically, $\Phi\left(s', p, s\right)$ now returns a couple of angles (φ, θ), similar to spherical coordinates angles.[3] On the one hand, φ is similar to the value returned by the 2D version, it corresponds to the angle projected in the $(\vec{x}, \vec{y})$ plane. On the other hand, θ is the angle $\angle(s, p, s')$ in the vertical plane. Additionally, each node must store two angle ranges instead of one to support the new dimension: $\left[lb_\varphi(s), ub_\varphi(s)\right]$ and $\left[lb_\theta(s), ub_\theta(s)\right]$.

Similarly, to expand a node s' with *path 2*, we must ensure a line of sight relation to $parent(s)$, $\varphi\left(s', parent(s), s\right) \in \left[lb_\varphi(s), ub_\varphi(s)\right]$, and $\theta\left(s', parent(s), s\right) \in \left[lb_\theta(s), ub_\theta(s)\right]$.

When expanding a node s' with *path 2*, we still have $local(s') = s$ and $parent(s') = parent(s)$. We must also update its angle ranges accordingly. As in 2D, we can take the intersection of the angle ranges of s and the angle range of the crossbar centered on s', as described in the equation 3.4.

$$S_\varphi = \left\{\varphi\left(s', parent(s), s''\right) \middle| s'' \in crossbar(s')\right\} \tag{3.2}$$

$$S_\theta = \left\{\theta\left(s', parent(s), s''\right) \middle| s'' \in crossbar(s')\right\} \tag{3.3}$$

$$\begin{cases} [lb_\varphi(s'), ub_\varphi(s')] & = [lb_\varphi(s), ub_\varphi(s)] \cap S_\varphi \\ [lb_\theta(s'), ub_\theta(s')] & = [lb_\theta(s), ub_\theta(s)] \cap S_\theta \end{cases} \tag{3.4}$$

Lastly, if the *path 2* conditions are not verified, we expand the node with *path 1.* We can then initialize the angles ranges as the minimum and maximum crossbar angles as described in equation 3.5. We can notice that, by applying equation 3.5 in the $(\vec{x}, \vec{y})$ plane, we obtain the result from the previous section, $\left[lb(s'), ub(s')\right] = [-45°, 45°]$.

$$\begin{cases} [lb_\varphi(s'), ub_\varphi(s')] & = [\min S_\varphi, \max S_\varphi] \\ [lb_\theta(s'), ub_\theta(s')] & = [\min S_\theta, \max S_\theta] \end{cases} \tag{3.5}$$

In a re-planning scenario, Phi* 3D is not very different than Phi* 2D. We extract the eight corners of the newly obstructed cell and discard the ones that are part of the local path. We also discard all their local children. This makes the graph valid again and usable. Then, we run the path finding algorithm on the updated graph to quickly generate a new solution.

[3]We provide a Python implementation of 3D Φ in the appendix C.

3.1.4 Local goal extraction

Once the global planning Phi* algorithm completes, we obtain a 3D geometrical path, from the start to the goal position. Our local planner needs a simplified and localized version of this path. A common method is to extract a local objective point G_L, relative to the current position of the UAV [23] [15]. The local goal should belong to the geometrical path and be close to the UAV, ideally at a distance h we can specify. We call h the local planner horizon. It should also help the UAV get closer to the global goal. Because the local goal is on the geometrical path, it is guaranteed to be safe and unobstructed.

A naive method is to use the grid discretization done during the Phi* planning. First, we select the node of the path closest to the UAV, then we move forward along the path, until we reach a node at a distance of at least h from the other node. The position of this node is the local goal. This method is simple, but may raise some issues. Unless we up-sample the number of nodes along the path, we are limited by the resolution of the Phi* grid. Since we may want to specify an horizon distance dependant on the UAV velocity, it is important to have a good granularity on the value of h.

The method we chose is to rely on mathematical projections. First, we select the segment of the path the closest from the UAV. Then, we compute the projection of the robot's position on the segment. Since we know the distance between the UAV and the segment, this is equivalent to solving a line-sphere intersection problem. Once we have the projection, we move it forward along the path by a distance h. If we go beyond the segment limit, we project the excess distance along the next segment, until we attain a point on the path. This algorithm only uses linear operations and square roots, which makes it efficient. It also does not rely on any external path planning parameter, which makes it very versatile.

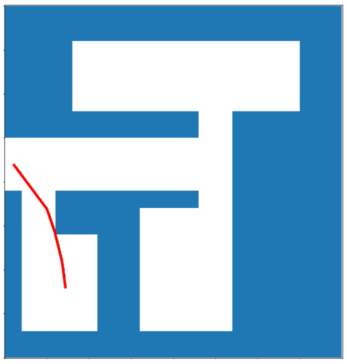
In our implementation, the horizon h is proportional to the UAV velocity and is clamped by a minimum and maximum value. This makes the UAV move even if its velocity is null. It gradually accelerates, until it reaches the maximum value.

3.2 Theta* and Phi* 2D comparison

In this section, we evaluate Phi* to show its strength against a more classic global planning algorithm. Because Phi* is an incremental evolution of Theta*, it is natural to compare both methods. Here, we only focus on the 2D implementations. Both solutions have been implemented in Python, using Numpy [46] and Matplotlib. To benchmark the algorithms, we use Occ-Traj120 [47], a large database of various small 2D maze-like environments. The database is made of 400 different maps and provide 120K trajectories for all of them. We use the trajectories provided to extract a start and goal points. The map grids provided are quite small to provide meaningful data, so we up-sampled them by 40, which increases a lot the complexity of the path finding problem.

To generate obstacles in a re-planning scenario, we needed a method to quickly create organic obstacles in a 2D discretized grid environment. We decided to generate perlin noise [48] on top of the grid and threshold it to obtain a mask. Then, we invert the state of the grid cells according to the mask. This gives us randomly placed blobs of obstacles in the map, as shown in figure 3.4. Note that with this method, we may not obstruct the current path solution. We may also remove other obstacles which may create new shorter paths to the goal. We execute this strategy multiple times, to obtain a multiple successive re-planning scenarios.

For each couple of map and start/goal points, we evaluate in 2D both Phi* and Theta* 12 times, using 5 re-planning scenarios. In total, we produce 130 data points. We present our result in figure 3.5.


(a) Initial planning environment

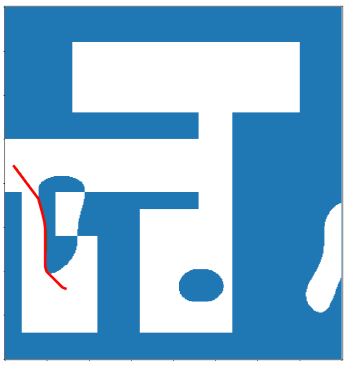
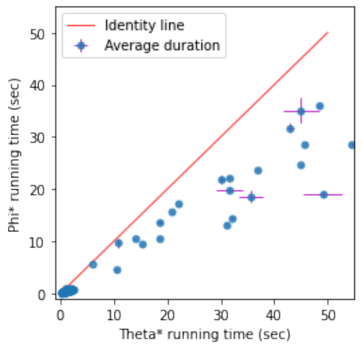
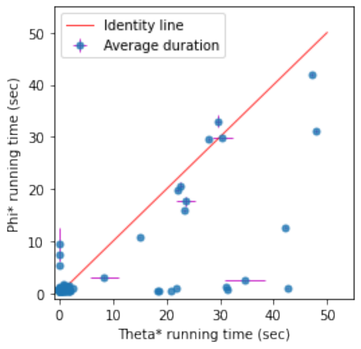

(b) First re-planning environment

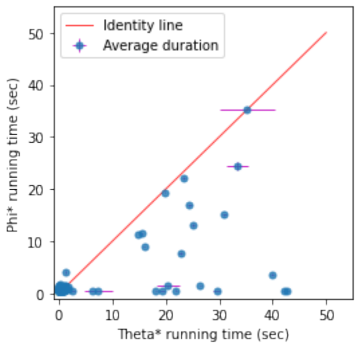
Figure 3.4: Example of a 2D environment, with a randomly generated re-planning scenario. In blue, we have obstacles. A path solution between a start and goal points is given by Phi* in red.



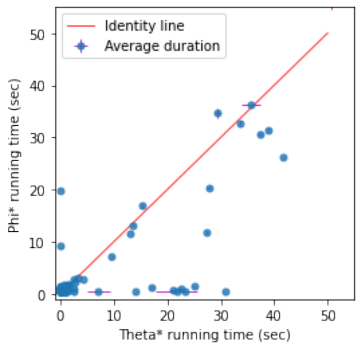
(a) Initial planning



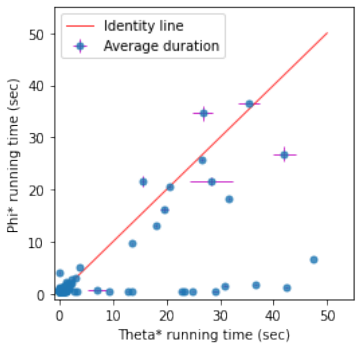
(b) 1st re-planning



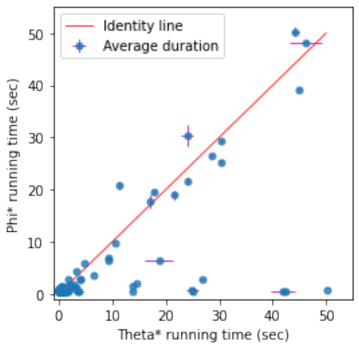
(c) 2nd re-planning



(d) 3rd planning



(e) 4th re-planning



(f) 5th re-planning

Figure 3.5: Comparison of the running time between Phi* and Theta*, in a 2D environment, with 5 re-planning scenarios. Phi* algorithm duration is shown in the Y axis, while Theta* is in the X axis. Each point is a map and start/goal scenario, reproduced 12 times. If the point is below the identity line, Phi* is performing faster than Theta* for this task.

We can see that, most of the time, Phi* is performing better than Theta*. Especially, at the re-planning step, Phi* computation time can drop close to zero, while Theta* still has a non-negligible running time. Interestingly, Phi* appears to perform better than Theta* even during the initial planning. According to the original paper, this could be due to the path tie condition, explained in section 3.1.2, which avoid unnecessary line of sight checks when points are aligned.

Similarly, we examine the geometric path length difference at the initial planning. To compute the path length, we simply use the sum of the euclidean distance between all

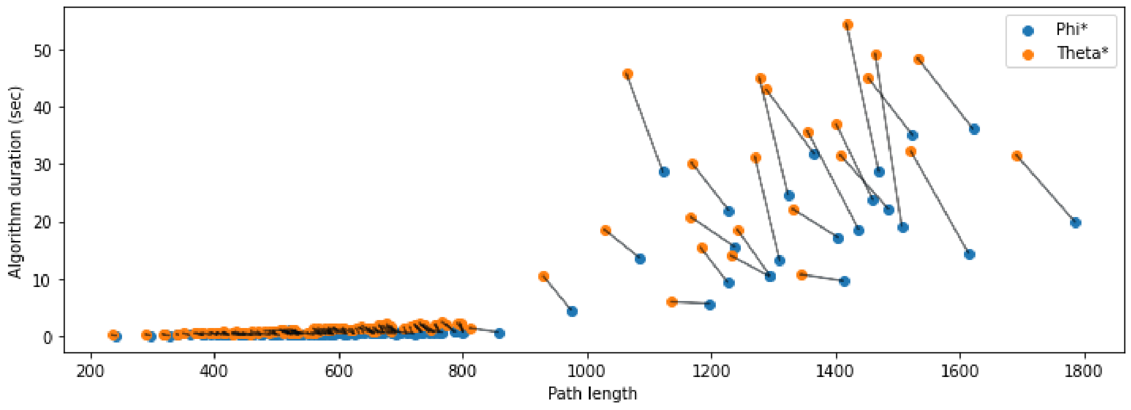


Figure 3.6: Lengths of path produced in a 2D environment by Theta* and Phi* 2D, with their running time. Each scenario produces a point for Theta* and for Phi*, which are linked in the graph.

the neighbor nodes along the path. We plot each data point of the path length with the algorithm running time in figure 3.6. We can see that the path generated by Phi* is always longer, which is logical because of the triangle inequality. Additionally, the running time of Phi* is always inferior or equal to the one of Theta*.

Theses results show that Phi* is an efficient algorithm for incremental global planning. It can quickly generate a path in an environment, and execute a re-planning scenario without discarding relevant data. Even if the path length is increased compared to a Theta* planning, we believe it is worth the gain in computation time and scalability in a large and partially known environment.

Chapter 4

Proposed local planner solution

In this chapter, we introduce our solution to the local planning problem for a UAV in a partialy known 3D environment. In section 4.1, we present our local planner, an hybrid motion primitive and optimization trajectory generation system. In section 4.2, we evaluate our method in two comparisons, one against a B-spline optimization algorithm and another against a naive motion primitive based algorithm.

4.1 Motion primitive based local planner

Once we have a local goal close to the UAV, we use a local planner to generate a time-parameterized trajectory $\Gamma : t \in \mathbb{R} \mapsto (p, v, a) \in \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3$. It should be dynamically feasible, that is, it should respect the dynamic constraints of the UAV hardware. The local trajectory should also be collision free and optimal with respect to the local goal. Lastly, it should be continuous in higher derivatives, in order to avoid undesirable jerk motion. We decided to build a novel hybrid method, inspired by [15], based on motion primitives and sequential optimization. Its architecture is presented in figure 4.1.

4.1.1 Motion primitive generation

A key part of a motion primitive based planner is the primitive generation. We use the work done in [44] as a trajectory generation method. Assume we have a UAV with a starting state $\Gamma(0) = (p_0, v_0, a_0)$, respectively the initial position, velocity and acceleration along one axis, T_f the total time needed to fly along a primitive and a goal $\Gamma(T_f) = \left(p_f, v_f, a_f\right)$ respectively the objective position, velocity and acceleration. The trajectory Γ must be dynamically feasible. For clarity, we only consider the generation along one axis.

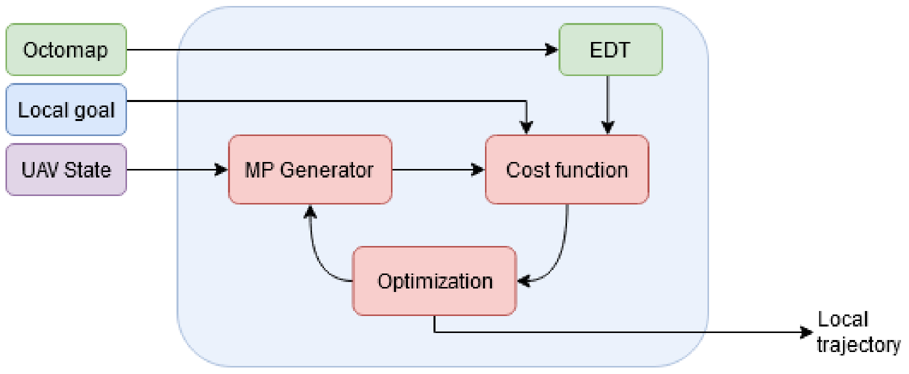


Figure 4.1: Architecture of our local planner.

[44] fix the constraint of having a thrice differentiable trajectory and build a cost function to minimize jerk along the path. By analytically optimizing the objective function, authors finds a solution as in equation 4.1.

$$\Gamma(t) = \begin{bmatrix} \frac{\alpha}{120}t^5 + \frac{\beta}{24}t^4 + \frac{\gamma}{6}t^3 + \frac{a_0}{2}t^2 + v_0 t + p_0 \\ \frac{\alpha}{24}t^4 + \frac{\beta}{6}t^3 + \frac{\gamma}{2}t^2 + a_0 t + v_0 \\ \frac{\alpha}{6}t^3 + \frac{\beta}{2}t^2 + \gamma t + a_0 \end{bmatrix} \tag{4.1}$$

Where α, β, γ are constants dependant on the final objective state $s(T_f)$. [44] actually solves the optimization problem for a fully or partially defined final state. In other words, we can constrain some components of our trajectory and leave others free. For example, the solution for one axis where the final velocity v_f and acceleration a_f are defined, but not the final position is given in equation 4.2

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{T_f^3} \begin{bmatrix} 0 & 0 \\ -12 & 6T_f \\ 6T_f & -2T_f^2 \end{bmatrix} \begin{bmatrix} v_f - v_0 - a_0 T_f \\ a_f - a_0 \end{bmatrix} \tag{4.2}$$

We can see that the method given by [44] is very efficient. The trajectory generation only requires to perform simple linear operations for each axis, which can be done very quickly. We can generate a large library of dynamically feasible trajectories using various constraints from the initial state of the UAV.

In our implementation, we fixed the final z_f coordinate in order to easily move the UAV up or down if needed. As in [15], we fixed the acceleration to zero $(a_{x_f}, a_{y_f}, a_{z_f}) = (0, 0, 0)$. Finally, from an objective velocity norm v_f and yaw angle θ_f, we deduce the fixed velocity components $(v_{x_f}, v_{y_f}, v_{z_f}) = (v_f \cos(\theta_f), v_f \sin(\theta_f), 0)$. Note that the vertical final velocity is zero, as we do not want to create vertical oscillations of the UAV along the path. The position along the x and y axis is left free. We now have a simple function $g(\theta_f, v_f, z_f) = \Gamma$ to generate a single trajectory.

Lastly, [44] adds more constraints during the generation, in order to discard primitives with too high accelerations. In an aggressive scenario requiring high velocities, we may add a system like that.

4.1.2 Cost function

Once we have a collection of trajectories, we need a method to evaluate them and select the best one. We build a cost function inspired by [43]. This function outputs a single scalar $E(\Gamma)$, lower for better trajectories. Since we already have dynamic feasibility constraints in the motion primitive generation algorithm, we do not need to include this constraint in our cost function. Therefore, we just need to optimize the proximity to the local goal G_L, and ensure the trajectory is collision free.

The end point cost E_{ep} represent the distance between the local goal G_L and the final state position of the trajectory. Additionally, we define the end point direction cost E_{dir} as the distance between the local goal direction and the final direction of the UAV on the trajectory. This cost helps to reduce oscillation when following a straight line on the global path. Both costs are defined in equations 4.3 and 4.4, where G_L and $G_{L_{dir}}$ are respectively the local goal position and unit vector direction.

$$E_{ep} = \left\| G_L - p(T_f) \right\| \tag{4.3}$$

$$E_{dir} = \left\| G_{L_{dir}} - \frac{v(T_f)}{\left\| v(T_f) \right\|} \right\| \tag{4.4}$$

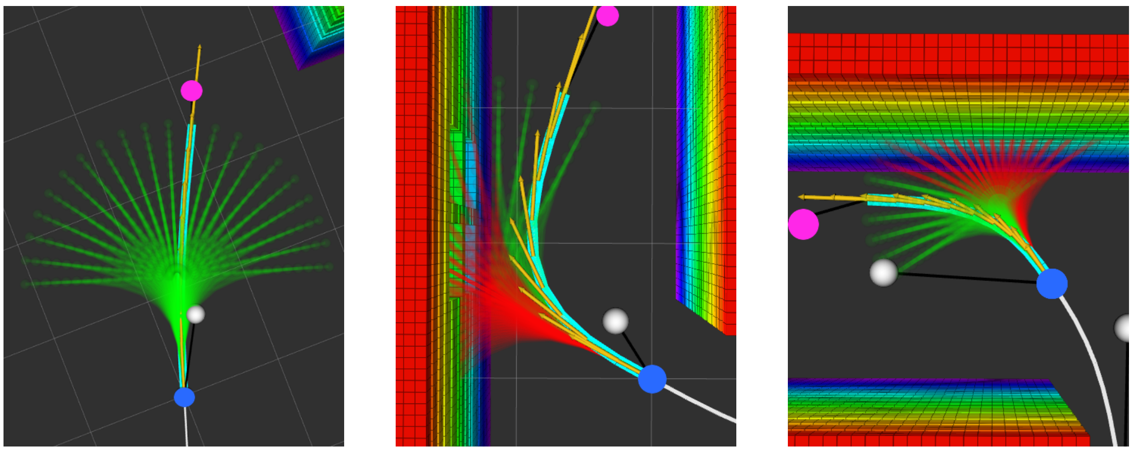
We also need to penalize trajectories which are too close from obstacles. A common technique used in [43] and [23] is to integrate along the path using a euclidean distance transform function (EDT) [16]. The EDT is a fast direct access structure storing the distance of a point to the nearest obstacle. $EDT(x) = 0$ if x is in an obstacle and $0 < EDT(x) \leqslant L$ otherwise, with L a maximum limit set in the implementation to improve performances. We invert the EDT, in order to penalize a lot the trajectories close or in obstacles, and almost not trajectories further away. Moreover, we multiply by the norm of the final velocity $\left\| v(T_f) \right\|$. This allows to choose unsafe trajectories only by keeping a very slow speed. We define the collision cost E_c in equation 4.5 and 4.6, with d_{safety} a safety radius around obstacles and δ a negligible quantity to avoid division by zero. We approximate E_c using an finite sum of N coefficients. N corresponds to the number of points sampled along the trajectory. We set it proportional to the distance between the initial and the final position of the trajectory. Thus, longer trajectories will have more points sampled, as they present a bigger risk of intersecting an obstacle.

$$d(t) = \max(EDT\,(p(t)) - d_{\text{safety}}, \delta) \tag{4.5}$$

$$E_c = \int_0^{T_f} \frac{\left\| v(T_f) \right\|}{d(t)}\, \mathrm{d}t \approx \left(\sum_{n=0}^{N} \frac{1}{d\left(T_f \frac{n}{N}\right)} \right) \left\| v(T_f) \right\| \frac{T_f}{N} \tag{4.6}$$

Using theses definitions, we can compute the total cost E of a generated motion primitive, as shown in equation 4.7. w_{ep}, w_{dir}, w_c are weight coefficients used to tune the objective function. We present how this function performs to rate and select the best trajectory in figure 4.2.

$$E\,(\Gamma) = w_{ep}\, E_{ep} + w_{dir}\, E_{dir} + w_c\, E_c \tag{4.7}$$


(a) In a simple setup with no obstacles, the best primitive is the one closest to the local goal.

(b) With obstacles, all the overlapping primitives have to be discarded.

(c) Even in sharp turns, a large range of final yaw angles allows to have a feasible valid trajectory.

Figure 4.2: Motion primitive libraries generated with different starting parameters and environments. It shows a top-down view of a 3D environment, with obstacles. The primitives are sampled uniformly over a range of final yaw angles θ_f and final velocities v_f, with a fixed altitude z_f here. The starting position is the blue circle, while the local goal is the purple circle. Primitives with a high cost are in red and ones with a low cost are in green. We highlighted the selected trajectory with the smallest cost in cyan. Additionally, velocity vectors are sampled along the selected trajectory and displayed as arrows in yellow.

On the contrary of methods like [23], we do not need to include additional terms to verify the dynamic feasibility. This constraint is already verified because of the trajectory generation method, which ensures this analytically. This allows us to have a faster cost function execution.

4.1.3 Sequential 1D optimization

Using the cost function, we can attribute a value to each motion primitive, which we can try to minimize in order to select the best trajectory.

A naive selection algorithm used in [43] and [15], is to generate a motion primitive library (MPL), by sampling $g(\theta_f, v_f, z_f)$ along some possible input values (θ_f, v_f, z_f). Then, we compute the cost of all the primitive in the MPL, and we select the one with the lowest cost. This strategy is quite inefficient. First, we have a discrete set of primitive to choose, making certain ideal trajectories impossible to select. We can solve this problem by increasing the size of the MPL, but this also increases the running time of the planner. Second, this method spends time on computing the cost of trajectories even though they could be easily discarded with a smarter selection algorithm.

We decided to use an optimization method, by merging our trajectory generation algorithm and our cost function as $f(\theta_f, v_f, z_f) = E\Big(g(\theta_f, v_f, z_f)\Big) = E(\Gamma)$. We are optimizing f to obtain the input $(\theta_f^*, v_f^*, z_f^*)$ for which we have the lowest cost E^*. After an analysis of the objective function f, we decided to implement a sequential 1D optimization method. Various reasons motivate this choice.

1. We did not manage to compute the gradient of f analytically, so it was ideal to use a gradient-less optimization method, which is usually applied in one dimension.

2. 1D optimization can be much faster to compute than N-dimensional optimization. We get very satisfying results even after a sequence of multiple 1D optimization over different parameters.

3. We assumed that the minimums relative to each parameter are independent of each other. That is, when setting two parameters constants, the minimum value when varying the third parameter should not be influenced by the value of the other two parameters.

This last hypothesis is not necessarily true and is highly dependent on the environment. We made our optimization more robust by optimizing multiple times the same parameter.

We chose the *Brent's method* [49], as it is an efficient and fast general 1D optimization algorithm, widely implemented in most programming languages. The objective function used does not need to be differentiable. We do not need initial values, but we must provide the algorithm with boundary values for the optimized parameters.

From empirical observations and analysis of the objective function f in various environments, we build the following algorithm for trajectory optimization. We have θ_0, v_0 and z_0 respectively the initial yaw angle, velocity, and altitude of the UAV. We also specify the boundary values used to initial the algorithm at each iteration.

- We optimize along the yaw angle $\theta_1^* = \underset{\theta}{\operatorname{argmin}} f(\theta, v_1, z_0)$ with $\theta \in \Big[\theta_0 - \frac{\pi}{2}, \theta_0 + \frac{\pi}{2}\Big]$ and $v_1 = \max\Big(0.1, v_0 - 1/T_f\Big)$.

- Then, using the parameter θ_1^* we just computed, we optimize along the velocity norm $v^* = \underset{v}{\operatorname{argmin}} f(\theta_1^*, v, z_0)$ with $v \in \Big[v_0 - 4/T_f, v_0 + 1/2T_f\Big]$. [1]

[1]We omitted the full expression for clarity, but we ensure v stays positive and non zero.

- To make our optimization robust against the independent parameters hypothesis, we optimize a second time along the yaw angle $\theta_2^* = \underset{\theta}{\operatorname{argmin}} f(\theta, v^*, z_0)$ with $\theta \in [\theta_1^* - 0.4\pi, \theta_1^* + 0.4\pi]$.
- Finally, we optimize along the altitude $z^* = \underset{z}{\operatorname{argmin}} f(\theta_2^*, v^*, z)$ with $z \in [z_0, z_G]$ and z_G the local goal altitude.

Each optimization is done independently using Brent's method. Note that this algorithm does not guarantee to find a minimum. However, empirically, we found this method is worth the time performance gain and ease of implementation. Thanks to it, we can obtain a satisfying local trajectory $g(v^*, \theta_2^*, z^*) = \Gamma$ to reach the surroundings of the local goal.

4.2 Naive and optimization based local planner comparison

Our sequential optimization algorithm may appear not ideal, especially when looking at work like [15] and [43]. In those work, authors implement a naive primitive selection algorithm, where a large library of motion primitive is generated, by sampling uniformly N_p primitives, and each one of them rated with a cost function. Then, the primitive of least cost is selected. This algorithm is very simple to implement and robust against local minimum. However, it limits the possible available trajectories, which can create sub-optimal trajectories if N_p is not high enough. The complexity of the algorithm is $O(N_p)$, which makes the algorithm too slow if a good precision is needed, while having an expensive cost function. On the contrary, with an optimization based method, the convergence towards a solution may be much faster, but the optimization may get stuck in a local minimum.

To evaluate both methods, we implemented both local planners in Python, using the cost function described in section 4.1.2. We ran them over various environments to generate multiple trajectories. By using a conservative global planner, we do not observe any critical issue with the trajectories generated. Thus, we are mostly interested in time performances

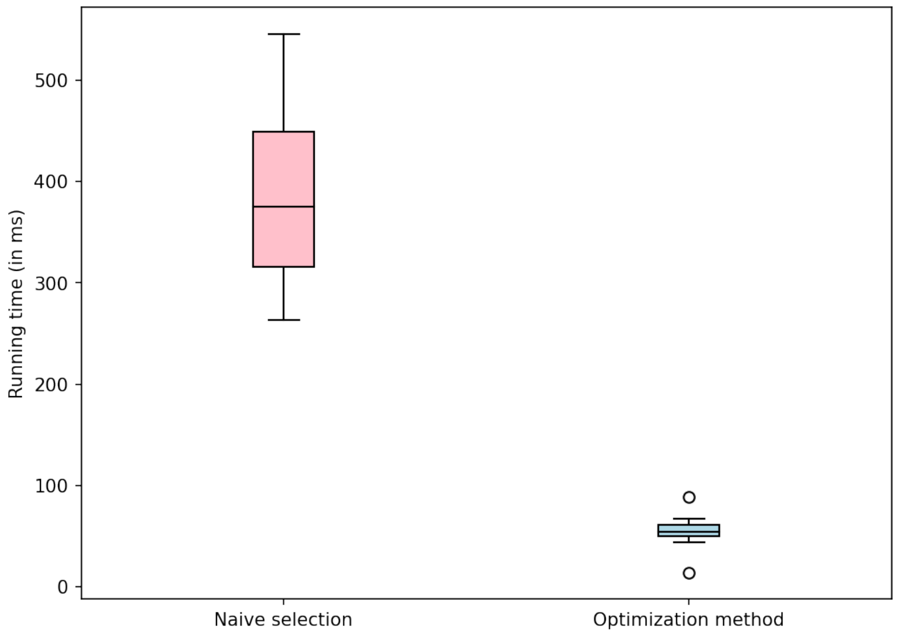


Figure 4.3: Running time distributions between local planners based on naive selection and Brent's method optimization ($N = 21$).

of both algorithms. The naive method running time is bounded by the number of samples N_p and the optimization based method running time is bounded by the precision required in the optimization algorithm. In our pipeline, we can set a low precision limit and still obtain a very satisfying result.

We present our results in figure 4.3. In the naive method, we sample $N_p = 315$ primitives, along a wide range of final velocities v_f, final yaw angles θ_f and final altitudes z_f. We have a mean running time of **378ms** for the naive method and **55.1ms** for the sequential optimization method. We can see that that the optimization outperform greatly the naive method. With the naive method, we can reach up to 550ms running time, which is quite slow. In a local planner, time performance is very valuable to allow for fast reactivity in a potential dynamic environment. It is also important for the reasons explained in section 5.4, where we need a precise initial UAV state estimation for the local planner to avoid undesirable jerk. This can be estimated using complex estimators and algorithms, though the best method is to minimize the local planner running time to minimize the state estimation error.

Therefore, we can safely say that our sequential 1D optimization method is more efficient in time performances than a naive trajectory selection. It does not waste time on computing the cost function with easily discarded primitives. However, as with all optimization based local planner like [23] and [11], it is vulnerable to local minimum. We can solve this problem by using a more conservative global planner. But we may want to improve this optimization to obtain a more robust local planner and rely less on global path safety mechanisms.

Chapter 5

Implementation

In this chapter, we describe our solution and various minor implementations specific points, which were interesting in the elaboration of our solution. In section 5.1, we describe the architecture of our entire pipeline. Then, from section 5.2 to section 5.6, we look at multiple problems we had to solve to successfully implement a realistic UAV 3D motion planning pipeline in simulation.

5.1 Software architecture

We implemented our entire pipeline in *ROS* [50], using the *PX4* firmware [51]. To communicate with the firmware, we are using the *MAVLINK* offboard protocol, through the *MAVROS* node. In a simulation setup, we use the *Gazebo* simulator [52] with PX4 SITL mode (software in the loop). Each simulated environment we used also had a ground truth Octomap, which is distributed by an *Octomap server* globally over a ROS topic. This server can also be connected to a point cloud sensor to generate an Octomap on the fly, which makes it adaptable to a real-world experiment. We present our architecture in detail in figure 5.1.

We built a *global planner node*, which retrieve the current starting position of the UAV from MAVROS sensor data and the global goal coordinates. From the Octomap, it generates a EDT and execute the initial Phi* 3D planning. From the geometrical path, the global planner then broadcast the local goal position and direction.

We also have a *local planner module*, which listens for request of trajectory generation. It also passively updates its environment structure, an Octomap and EDT, similar to the

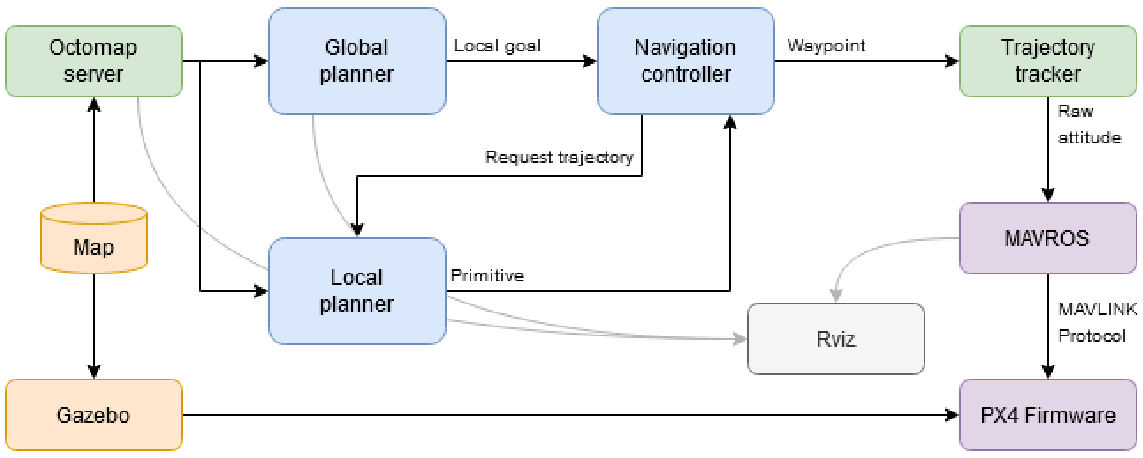


Figure 5.1: Architecture of our pipeline in simulation.

one used by the global planner. When receiving a trajectory generation request, the local planner execute the algorithm of section 4.1. It generates a solution using the sequential 1D optimization, which quickly generates trajectories and compute their cost. This provides us with an ideal local feasible trajectory.

Then, we have a *navigation controller module*, which objective is to control the UAV from a high level. This module receives the local goal from the global planner and request a local trajectory to the local planner. Meanwhile, it must ensure the continuity of the UAV motion. At a high rate (> 50Hz), the navigation controller samples the current local trajectory and sends the waypoint state to a *low level trajectory tracker*. When it starts to reach the end of the trajectory, the module request a new trajectory to the local planner and replace the current primitive when the local trajectory generation is done.

Finally, a low level trajectory tracker is useful to connect to MAVROS and convert a sequence of state waypoints in low level UAV attitude commands. Our pipeline uses the implementation of [53], which is inspired by the work of [54]. Most of the nodes are plotting visual content to a *Rviz* visualization interface.

This entire pipeline is quite heavy, especially in a simulation setup. However, we believe it scales well in a real-world hardware implementation. We mostly used Python to develop all of our custom modules, using common librairies like Numpy [46] and Scipy [55] for optimization. We also implemented our local planner in C++, as explained in section 5.5.

5.2 Collision checks

Like a lot of UAV related works, we chose to represent the mapping of the environment using an Octomap [13], as described in section 2.1. Because of their popularity Octomaps have a lot of well maintained library implementations, which also feature EDT generation. We used two collision check methods. The first one is simply a point EDT check, with a potential safety radius added. This verifies the validity of a point in the environment. The second method is a ray cast, to check if any obstacles are along a straight line. For example, this is useful to check if two points are in line of sight in the Phi* algorithm. To implement this algorithm, we sample points regularly along the straight line, and we check their EDT, with a safety radius at least equal to half the sampling distance. This gives us a simple and efficient algorithm for ray cast checking with a flexible safety radius around the straight line.

5.3 Motion primitive execution duration

In section 4.1, we mention the T_f parameter, which is a fixed constant representing the duration the UAV will take to execute one motion primitive. On the one hand, by choosing a high T_f, we make spatially longer primitives, which may fail more easily. On the other hand, with a smaller T_f, we increase the granular control of the local planner and allow for a much finer optimization. However, the drawback of a small T_f is that we still need the time necessary to generate the next future trajectory. To avoid failures, we need T_f much higher than the average generation duration of a trajectory.

5.4 Initial local planner state estimation

To start a local planner trajectory generation, we need to send the initial UAV state at the time the trajectory will be actually executed. This future initial state is unknown and may be estimated. If we provide the state of the UAV at the time of the trajectory request, we must hope that the generation algorithm will be fast enough to not have a too large offset between the provided UAV initial state and its actual state. An offset too big could result in undesirable jerk or in a critical crash.

Thus, we must request a new trajectory when the UAV is as close as possible to finish the execution of the previous primitive. We request the generation of a new trajectory a fixed time T_g before reaching the end of the previous one. Therefore, to account for the trajectory generation time, we must have T_g at least higher than the worst generation duration, but as close to zero as possible.

Using previous states and T_g, we can apply a linear interpolation to estimate the future state value. Since the state values are noisy due to the UAV sensors, we may have better results using a Kalman filter estimator.

However, in our implementation, by simply using a the UAV position at the time of the request, the expected velocity according to the current primitive at the end of the trajectory and a null acceleration, we already get satisfying results. We have a small offset in the expected and real initial position, which may create a small jerk, but it is generally unnoticeable.

5.5 C++ Local planner optimization

As we have seen in section 4.2, we can reach good time performances for the local planner by using an optimization based primitive selection algorithm. However, as stated in section 5.4, we really need the local trajectory generation duration to be as close as possible to zero. And without improving too much our algorithm, we can easily improve greatly the time performances by implementing our local planner as a C++ ROS module, instead of using Python. C++ is a compiled programming language, while Python is interpreted on-the-fly, thus we can expect a very fast execution. Moreover, C++ has plenty of popular optimization libraries implemented and well maintained, which makes it an ideal choice for real-time optimization algorithms, especially in robotics and embedded systems.

To re-implement our module, we can use to our advantage the distributed logic structure of ROS, which allows us to create language agnostic modules, which should be able to

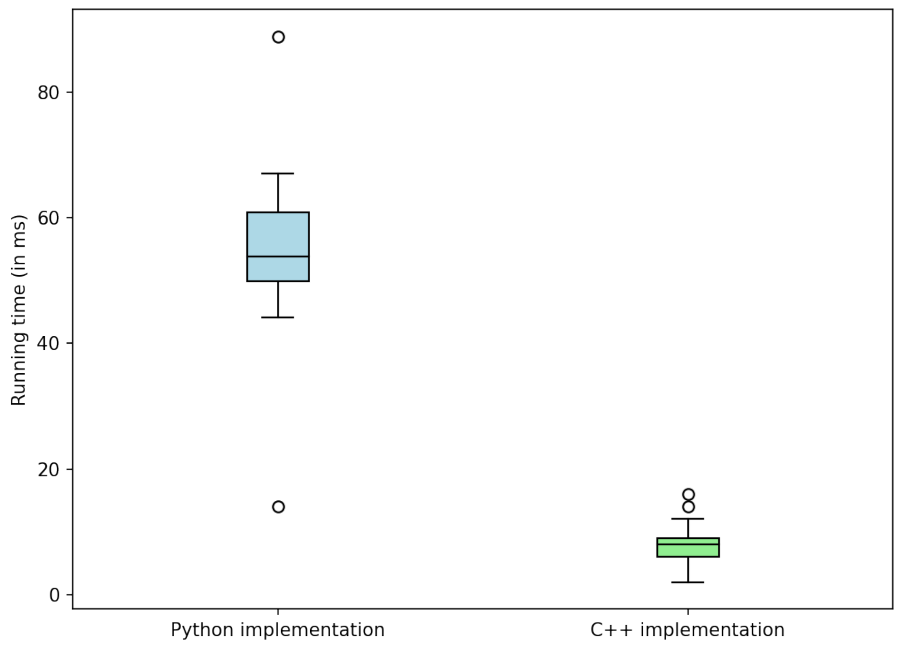


Figure 5.2: Running time distributions between local planners implemented in Python and in C++ ($N = 21$).

communicate normally. Thus, we can create our local planner in C++, while the rest of our pipeline will stay mostly the same. This saves us development and testing time. Because C++ is very popular in robotics, we have a C++ version for most of the libraries we used, for example to manipulate the Octomap and the EDT [13] or the generation of primitive [44]. To use the Brent's optimization method, we use Scipy's implementation in Python [55], while we used Boost's implementation in C++ [56].

To compare the planner performances, we ran the generation of various trajectories in different environments and we recorded the running time of each algorithms. We obtain the results in figure 5.2. We have an average running time of 55.1 ms for the Python implementation and 8.00 ms for the C++ implementation. Additionally, we can see that the standard deviation is much smaller with the C++. Thus, we obtain a great increase in performances, which is what we expected. Moreover, the running time of the C++ implementation is close or inferior to the duration of a loop of our main navigation controller. In other words, the controller can request a trajectory, sleep for the rest of the loop, then receive a new optimized trajectory during the next loop, in order to send an updated waypoint. This should minimize as much as possible the initial state estimation explained in section 5.4 and help avoid potential fast moving obstacles in a dynamic environment.

In practice, we noticed that the controller was looping between 2 to 4 times before receiving an answer from the local planner. This increased delay is probably due to input-output processing, communication protocols and the Python interruption process, which may not be efficient. This could be optimized, by converting the entire pipeline to C++ and by using distributed processes at the OS level instead of ROS. However, from our experiments, we believe that this is good enough for us. The UAV can correctly minimize jerk, and safely avoid obstacles in most scenarios.

5.6 Start/Goal inversion

A major drawback to the implementation of Phi* 3D described in section 3.1 is the added number of collision checks, which is much higher in 3D than in 2D, making the algorithm not as efficient. A small optimization we found, was to invert the start and the goal positions. In other words, the global planner will now start from the goal position and look for the UAV in the environment. A few reasons motivate this choice.

First, if the UAV position changes and we need to recompute the Phi* graph, it is impossible to adapt the algorithm to change the starting point, without re-building the entire graph. It is, however, easy to change only the goal position. The only action needed, is to recompute the $h(s)$ value on nodes in the *open set*. Thus, in a re-planning scenario where the UAV position must change, it is more efficient to simply move the goal position.

Second, when re-planning, Phi* will rebuild the part of the graph located between the new obstacle and the graph goal. In other words, in order to perform efficiently, we should ideally minimize the distance between the new obstacles and the goal. In most setups, we can expect new environment data to be centered around the UAV, because of the robot sensors. Thus, it makes sense to invert the start and goal position, to optimize re-planning when changes are detected close to the UAV. This should help avoid failures if the global planner takes too much time to compute a solution.

Chapter 6

Pipeline evaluation

In this chapter, we focus on explaining our implementation through quantitative results. In section 6.1, we benchmark our solution through repetitive testing. Then, in section 6.2, we evaluate the efficiency of our re-planning algorithm through an unknown environment scenario.

6.1 Solution evaluation

To evaluate the performance of our full pipeline, we must use a reproducible test platform, to easily repeat identical tests, over multiple maps and configurations. Ideally, testing should be automatic. In this section, we are only interested to evaluate our global and local planner, how they interact with each other and if our pipeline is resilient to working in different environments. Therefore, we consider the environment fully known. We also do not require large scale maps, as repetitive quick testing is our priority. Lastly, because of our implementation, we require environment as Gazebo world with a ground truth Octomap.

This type of data, especially in the context of repetitive testing for UAV motion planning evaluation, is not widely available. In our research, we have not found a lot of motion planning works evaluating their work over various environments in a 3D simulation setup. We decided to use the *forest_gen* repository [1], built by the Autonomous System Lab of the ETH Zurich [23]. The repository includes 9 different maps built as procedural forests in a Gazebo world, with their corresponding Octomap. Each map is 10x10m with a 0.2 trees/m^2 density. They also provide a list of 100 possible start and goal points configurations for each map. The configurations points are considered enough far away from obstacles. This gives us 900 different test cases. Since some uncertainty is introduced in the Gazebo simulation, we also repeat each individual test 10 times to obtain a good performance average, totaling in 9000 test cases. We automate the execution of the pipeline 9000 times on all the configurations.

In a classic test, first, the pipeline and the simulation are started. Then, the global planner is tasked with finding a global path solution. Once it succeed, the navigation controller uses the extracted local goal to request an optimal primitive trajectory to the local planner, which it then samples regularly and send state waypoints to the trajectory tracker. The tracker communicate with low level components to make the UAV move forward. Once the navigation controller reaches the end of the primitive, it requests a new local trajectory. When the UAV is close the global goal, the simulation is interrupted and the collected motion planning data is written in a file, with a success flag. Over one test, we collect as much data as possible about the motion planning of the UAV. Especially, we

[1]`https://github.com/ethz-asl/forest_gen`

are interested at the success rate of the pipeline and the causes of failure. A success is when the UAV reaches a certain distance to the goal. We distinguish three types of failure:

- Global planner failures are due to Phi* 3D not being able to find a valid path between the start and the goal. To not run the algorithm for unnecessarily too long, we also introduce a timeout after a few minutes of running time.
- Local planner failures are generally due to the sequential optimization algorithm getting stuck in a local minimum.
- Unknown failures are minor failures due to unexpected errors in other components of our system. They can be easily avoided by repeating the test.

Note that, according to the structure of the maps, Phi* should always be able to find a solution in a reasonable amount of time. However, we recorded a high amount of local planner failures. We infer it must be due to local minimums appearing when the UAV is too close to a tree. When deciding which side to choose to avoid the tree, the optimization algorithm does not follow the global path. So even if the obstacle avoidance works, the UAV is not along the global path anymore, which results later in a crash and a local planner failure. To eliminate this problem, we increase the safety margin of the global planner. That way, the global path stays at a larger distance from the trees, which makes the local planner more robust. However, by increasing this margin, we also reduce the number of feasible paths found by Phi*, which makes it incapable of finding global path solution sometimes. For future work, we may be able to improve our algorithm by introducing a Phi* re-planning when the local planner is stuck in a local minimum.

Table 6.1: Success results of running our pipeline over 8979 test cases.

Result type	Quantity	Percentage
Successes	8515	**94.83** %
Global planner failures	74	**0.8241** %
Local planner failures	390	**4.343** %
Total test cases	8979	

After running our pipeline on the 9000 test cases and removing the unknown failures, we obtain the results in table 6.1. We can see that even though we get a few failures, most of the test cases succeed. Most failures are because of the local planner. This is obtained using a safety margin in the global planner of 0.8 m, in a discrete 3D grid of 0.2 m resolution. They can be adjusted to reduce the failures of the local planner, but that may increase the global planner failures. From those results, we can say that our pipeline performs well to safely bring the UAV to the goal point.

Additionally, it is relevant to examine the path quality by comparing the length of the generated trajectory, with the theoretical best possible path length. To compute the theoretical shortest path, for a specific configuration, we execute an A* algorithm with a weight on the heuristic, to make it close to optimal. We also apply it on a very fine discrete grid, with a resolution of 0.1 m which is equal to the resolution of the Octomap. The algorithm may take a long time to complete, but by summing the distance between each node of the final path, we should find a very satisfying estimation of the shortest distance between the start and goal point.

Similarly, if the test was successful, we can compare this shortest distance to the actual distance of the real trajectory followed by the UAV. For a given configuration, we compute the average of the sum of the distance between of all the sampled positions of the UAV during the entire test case. We present our results using the relative difference between the theoretical shortest distance and the practical distance in figure 6.1. We have an average relative difference of **63.78** %. However, as we can see on the histogram, the mean of the distribution is influenced by a few outliers above 150 %. Most test cases are around less than 100 %. We actually see a peak at around 40 %.

Because of the UAV size and the obstacle avoidance margin, we cannot reach the optimal path. Therefore, the results we have are expected and satisfying. The very large differences (above 100 %) are probably due to the global planner safety margin, which is too high and makes the global planner unable to find the normal solution. Thus, it has to take largely different paths, which makes them much more inefficient than the optimal path.

Theses results show that our pipeline is resilient and safe for the UAV. It is conservative, in that it may sacrifice path length for safety.

6.2 Re-planning in an unknown environment

One of the main goal of our pipeline is to integrate a re-planning component. We implemented it into our global planner, using the Phi* 3D incremental algorithm. By extracting the node of the path planning graph impacted by the new obstacles, we can discard the outdated part of the graph and run the path finding again, to find a new solution for the global planner. Note that, since our local planner computes trajectories online as soon as needed, it is also robust to new obstacles appearing in the environment.

To evaluate our pipeline re-planning capabilities, we decided to use a method similar to the work in [23]. As in section 6.1, we use the *forest_gen* repository, which also provides a large forest map, of 50x50m, with its corresponding Octomap. We can use it to simulate a larger environment than in the previous experiment. In a large environment, it may not be scalable to store the entire map of the world, especially if only a small part of it is useful for the motion planning of the UAV. Similarly, if the UAV evolves in an unknown environment, it will only be able to navigate using its sensors, which generally only detect obstacles in the surroundings of the UAV. Therefore, it seems interesting to simulate the behavior of the UAV, with a limited vision field.

As it is harder to simulate the behavior of sensors, we simplify our solution, by using an artificial sphere of radius R_{vision} centered at the UAV. Every obstacle located inside the sphere is fully detected by our system, even if it is hidden behind other obstacles, and the rest of the environment is unknown. As the UAV will move forward, the sphere will move too, and the motion planning system will be able to detect more relevant and previously unknown obstacles, which should update the trajectory. This system is not necessarily

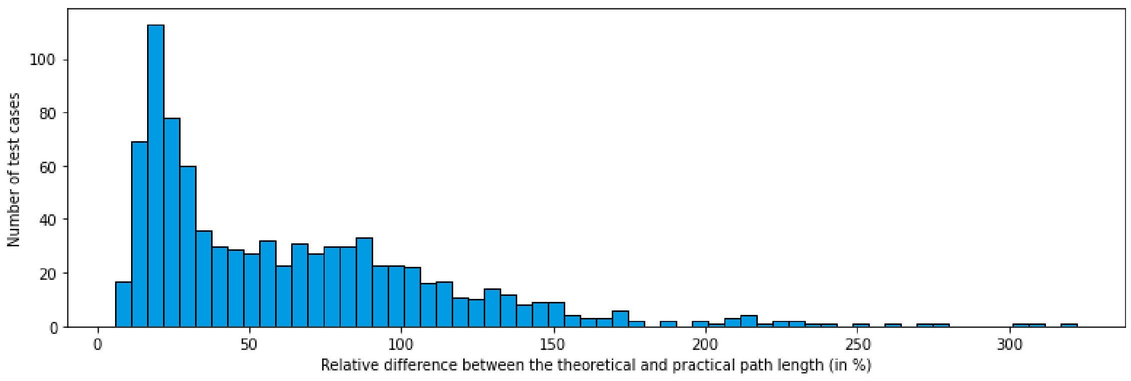
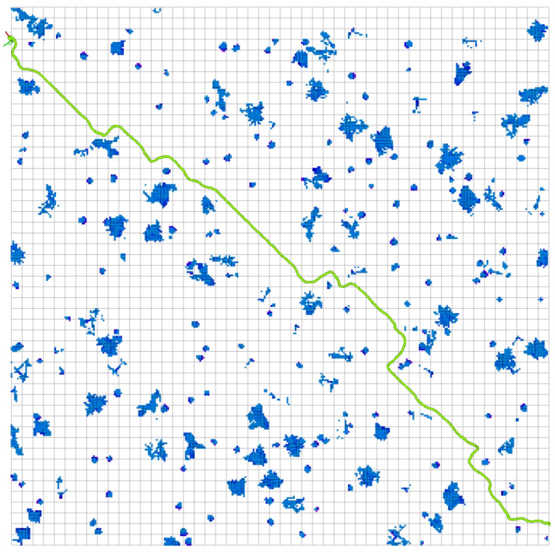


Figure 6.1: Histogram of the average relative difference between the theoretical and practical trajectory path length, over 883 samples.

realistic to simulate the behavior of real sensors, however we believe that it will give us a good approximation of the robustness of our pipeline. Additionally, this method simulate well a sliding map window, used in systems like [15] and [11] to avoid wasting unnecessary resources to store a very large environment map.

We set $R_{\text{vision}} = 15\,\text{m}$, though eventually we could set this parameter much lower by optimizing the global planner running time, using C++ and a bigger processing unit for example. Because we cannot update the global planner continuously, we update it every time the UAV moves a step of 5 m. In other words, every 5 m, we extract all of the Phi* graph nodes obstructed by newly added obstacles. Those nodes are discarded, along with their children. Since the global goal is actually the origin of the Phi* graph, as explained in section 5.6, we only discard a small portion of the graph located in the vision sphere around the UAV. Then, the path planning algorithm is ran again, using only still relevant environment data and the current position of the UAV. A new path solution should be quickly found. Note that we cannot guarantee that the UAV, which is still moving using the previous valid global path, will be along the new solution path, once the algorithm stops. However, since, the algorithm is quite fast and re-use previous graph structure, this is generally not a problem. A smaller radius R_{vision} may help reduce the running time and ensure the UAV is on the new path.

To compare our re-planning algorithm, we also run the pipeline using a full knowledge of the environment. In that case, no re-planning will happen, as the best global planning solution will be found by the initial execution of Phi* 3D. Both scenarios are executed on the same map, with the start located at one corner, and the goal located at the other corner. We present the resulting trajectory of the UAV in the figure 6.2. We obtain a trajectory of 84.77 m using the simulated limited vision field, against a 80.38 m length with a fully known environment map, resulting in a **5.46** % increase. The length difference seems minor enough to say that our re-planning system is quite efficient and does not increase a lot the path length. By looking at the trajectory itself though, we can see that


(a) Trajectory with full environment knowledge. No re-planning. Trajectory length: 80.38 m.

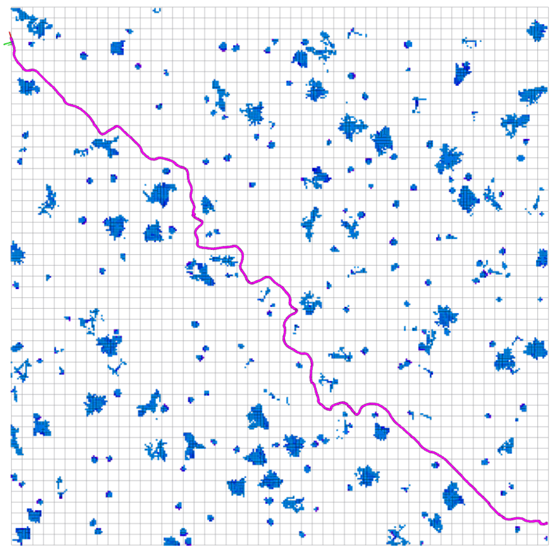

(b) Trajectory with a limited vision field around the UAV, with frequent re-planning of the global planner. Trajectory length: 84.77 m.

Figure 6.2: Comparison of the full navigation pipeline trajectory, with full knowledge of the environment and with only a limited vision field of a sphere of radius $R_{\text{vision}} = 15\,\text{m}$ around the UAV. The global planner re-plan the path with new information every 5 m moved. The start is located at the bottom right corner, while the goal is at the top left corner.

the limited horizon made the global planner choose a different less optimal path, which created a lot of sharp unnecessary turns. However, even if it may not be as optimal, we can see that the conservative parameters in the global planner made our local planner able to safely avoid all the obstacles, without risking the UAV safety.

Chapter 7

Discussion

In this chapter, we are looking into our system limitations and potential future improvements. We describe in section 7.1 the main current vulnerabilities encountered. Then, in section 7.2, we examine how to make our pipeline robust in a dynamic environment. Finally, in section 7.3, we introduce a few methods to implement our system in a real-world UAV.

7.1 System vulnerabilities

As we explained in section 4.1.3, our sequential optimization method is, on the one hand, very fast, but, on the other hand, it is not very robust against local minimum. This weakness is due to the 1D sequential algorithm, which does not consider the whole shape of the cost function at the same time. Additionally, depending on the environment, the cost may present more local minimums. A common case is with a small obstacle in front of the UAV, like a tree. Each side is a local minimum, with the side taken by the global path being the global minimum. If the tree is in front and close to the UAV, a small offset in the initial optimization parameters will generate a trajectory going through the wrong side of the tree. Because our solution is fast, we may be able to improve our method by executing our optimization with multiple initial parameters, and choosing the best one. We could also sample a few primitives in a fixed range, like in the naive method described in section 4.3, which is not vulnerable to local minimum. Then, this approximation could be improved using the sequential optimization.

We made the choice of sacrificing path quality for performances in the local planner. But we were able to compensate this issue with the global planner. Indeed, if the obstacles are far enough from the UAV, local minimums may still be present, but they will not be as much accessible as before. Thus, an easy solution is to increase the margin of safety, which corresponds to the minimal distance between a Phi* graph node and its closest obstacle. The minimal margin of safety cannot be null, as it needs to consider the size of the UAV itself. It needs to be higher to increase the local planner safety. However, this increased safety not only reduces the efficiency of the generated paths, but it also makes some solution non-existent. Indeed, if obstacles are too close to each other, Phi* will not be able to pass through them, as the safety margin will not be verified. We are able to partly solve this problem by increasing the Phi* grid resolution, giving the algorithm more granularity to find solutions. But this increases a lot the computational cost.

7.2 Dynamic environment limitations

Even though it was not a requirement for this work, we can discuss the application of our pipeline in a dynamic environment. We mostly applied it in fully known or partially known static environment, where obstacles do not move and the acquired data about the environment will always be true. In a more dynamic environment, obstacle may move unpredictably, possibly close to the UAV, modifying already stored data.

Technically, our global planner is able to react to new obstacles added in the map. Thus, it is able to handle a dynamic scenario. However, the re-planning step can be computationally expensive and may not be fast enough for a dynamic situation. For example, in a warehouse environment, with objects moving everywhere with a good certainty, dynamic changes can happen quite fast, possibly close to the UAV, which should trigger a re-planning strategy and compute a solution at almost real-time. For very fast response though, we should rely on the local planner, which let the global planner some time to react, but our pipeline is currently not efficient enough. A common and easy improvement could be to implement the global planner in C++. As we have seen in section 5.5, we can expect a strong performance increase. Overall, re-implementing the entire pipeline in C++ could be very beneficial, as it will increase all node rates in the system.

Even if our local planner performs well in running time, it may also not be fully adapted to work in a dynamic environment. A useful feature for dynamic local planners [23] is a continuous data input, to continuously update the local trajectory to face new obstacles, even if a valid local trajectory has already been found before. This paradigm is a lot different to our solution, and is not easy to integrate, especially with efficiency constraints. Additionally, this sort of pipeline is particularly adapted to optimization based planner, as the algorithm can be started again with the previous solution and it quickly converges continuously to a better solution as new data arrives. We have not seen any works implementing a continuous re-planning local planner using a motion primitive technique. Because of this, our local planner will not be very efficient in highly dynamic environment, where quick unexpected obstacles avoidance is needed.

7.3 Real-world implementation

Even though it was an initial objective when starting this thesis, we were not able to conduct the implementation of this pipeline in a real-world UAV. However, we still did some research over implementation methods. We would have implemented our pipeline on a Parrot Bebop 2 platform, which does not feature a lot of on-board sensor, apart from a simple camera. The firmware is closed, thus, we can only input a limited set of commands, like the current preferred direction toward the objective, using *bebop_autonomy* a ROS driver made for this UAV. To implement a localization solution, we decided to use a visual SLAM software. A popular library is OpenVSLAM, which works with any monocular camera, for offline mapping and online localization [57]. From the Bebop driver, we can extract the video feed to OpenVLSAM and perform localization. After pre-processing lightly the image, by increasing brightness and contrast, we manage to obtain good localization result in a non-ideal environment. To control the UAV, no work exists to perform trajectory tracking, especially with the high level Bebop driver. To solve this, we examined implementing a simple PID controller, from the waypoint states of the custom navigation controller. Future work could be done to implement our complex pipeline in a real-world UAV.

Finally, even if we can implement the data pipeline in a real-world UAV, the motion planning algorithm may not work. Implementing with real-world hardware introduces a lot of undesirable noise, in the UAV state estimation and the obstacles description. This could cause unwanted jerk, especially when switching the current sampled primitive. It could also create crashes, as poorly detected obstacles may be located in the planned path. Without improving the localization and mapping methods, which are not in the scope of this work, it is possible to increase the safety of both planners to make the UAV follow a more conservative trajectory. However, it may generate sub-optimal paths.

Chapter 8

Conclusion

In this thesis, we introduced a motion planning pipeline for UAV navigation in a 3D partially known environment. We first described our global planner module, which goal is to generate a simple collision free geometrical path, without considering dynamic constraints of the robot. Our planner uses our novel 3D generalization of the Phi* path finding algorithm. It is a natural improvement of Theta* and A*, to include re-planning capabilities in an any-angle solution. As an incremental algorithm, it is able to re-use previously computed data if it is still relevant, to reduce the total number of computations. In 2D, we saw a very large improvement against Theta*, especially at re-planning. This planner ultimately extract a local goal close to the UAV to easily generate of local trajectory feasible by the robot.

Then, we present a local planner, which uses the local goal, to generate a local time-parameterized trajectory. It should be collision free and dynamically feasible by the UAV. While most state-of-the-art works implement an optimization based planner, we built a fast motion primitive generator along with a sequential 1D optimization. The former ensures a fast primitive generation, with dynamically feasible feature. The latter computes an optimal primitive as the local trajectory, by minimizing a cost function. We implement this function as a collision avoidance tool and to follow the local goal as much as possible. This algorithm is fast and efficient. After a correct tuning of the cost function, it computes trajectories of good quality. After evaluating the time performance of our method, we found it outperforms greatly more classic motion primitive library based algorithms. Even if our planner is quite vulnerable to local minimum, it is very fast and could easily be improved to be made more robust.

We evaluated our entire pipeline through a group of reproducible small tests, using a fully known static 3D environment. Our pipeline was implemented using ROS, MAVROS, PX4, MAVLINK, Octomap, Gazebo for the simulation and Rviz for the visualization. Over 900 unique test cases, with 10 repetitions each, we measured a success rate of 94.83 %. Over those successes most path length are not too far from the optimal distance, which cannot be reached because of the UAV safety radius. We obtain a relative difference between the theoretical optimal path length and the practical obtained length of 63.78 %. According to those results, our pipeline is quite robust and can be applied to various situations. It mostly performs conservatively, and sacrifices path efficiency for UAV safety. We also evaluate our pipeline in a re-planning scenario, by constraining the vision field of the obstacle detection system to a sphere of a relatively small radius, centered at the UAV. When the UAV moves, new obstacles appear and an online global planner re-planning is triggered. We obtain good results, not far from the optimal path, which could be improved by decreasing the running time of the global planner, with a C++ implementation for example.

This pipeline is satisfying for our setup, but it could be greatly improved. Reducing the local planner vulnerabilities to local minimum would allow to decrease the safety margin of the global planner. It would increase the success rate of the system, while overall improving the paths lengths. Moreover, it would be very beneficial to implement the pipeline in a real-world UAV. We would expect various issues to rise, because of noisy localization of the UAV and the obstacles. However, it would be necessary to evaluate the feasibility of this work in a real scenario.

In conclusion, the proposed method works great in a simulation setup. We explored various motion planning strategies, with global and local planning. We compared each technique by progressively increasing in complexity, from 2D to 3D and from a known to a partially known environment. The pipeline we implemented constitute a good basis for future work, to implement a more robust framework on a real-world UAV.

Bibliography

[1] Nikolaos Stefas, Haluk Bayram, and Volkan Isler. "UAV landing at an unknown location marked by a radio beacon". In: (2019).

[2] Jesus Capitan, Arturo Torres-Gonzalez, and Anibal Ollero. "Autonomous cinematography with teams of drones". In: *Workshop on Aerial Swarms. IEEE International Conference on Intelligent Robots and Systems (IROS)*. Vol. 1. 2019, pp. 1–3.

[3] Javier Alonso-Mora et al. "Object and animation display with multiple aerial vehicles". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 1078–1083.

[4] Beipeng Mu et al. "Information-based active SLAM via topological feature graphs". In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE. 2016, pp. 5583–5590.

[5] Luca Bartolomei, Lucas Pinto Teixeira, and Margarita Chli. "Perception-aware Path Planning for UAVs using Semantic Segmentation". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2020)(virtual)*. 2020.

[6] Davide Falanga et al. "Pampc: Perception-aware model predictive control for quadrotors". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 1–8.

[7] Lun Quan et al. "Survey of UAV motion planning". In: *IET Cyber-systems and Robotics* 2.1 (2020), pp. 14–21.

[8] Charles Richter, Adam Bry, and Nicholas Roy. "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments". In: *Robotics research*. Springer, 2016, pp. 649–666.

[9] Daniel Mellinger and Vijay Kumar. "Minimum snap trajectory generation and control for quadrotors". In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 2520–2525.

[10] Jesus Tordesillas et al. "Real-time planning with multi-fidelity models for agile flights in unknown environments". In: *2019 international conference on robotics and automation (ICRA)*. IEEE. 2019, pp. 725–731.

[11] Vladyslav Usenko et al. "Real-time trajectory replanning for MAVs using uniform B-splines and a 3D circular buffer". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 215–222.

[12] Franklin Samaniego et al. "UAV motion planning and obstacle avoidance based on adaptive 3D cell decomposition: Continuous space vs discrete space". In: *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*. IEEE. 2017, pp. 1–6.

[13] Armin Hornung et al. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees". In: *Autonomous robots* 34.3 (2013), pp. 189–206.

[14] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[15] Matthew Collins and Nathan Michael. "Efficient Planning for High-Speed MAV Flight in Unknown Environments Using Online Sparse Topological Graphs". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 11450–11456.

[16] Pedro F Felzenszwalb and Daniel P Huttenlocher. "Distance transforms of sampled functions". In: *Theory of computing* 8.1 (2012), pp. 415–428.

[17] Yuncheng Lu et al. "A survey on vision-based UAV navigation". In: *Geo-spatial information science* 21.1 (2018), pp. 21–32.

[18] Thierry Siméon, J-P Laumond, and Carole Nissoux. "Visibility-based probabilistic roadmaps for motion planning". In: *Advanced Robotics* 14.6 (2000), pp. 477–493.

[19] Steven M LaValle et al. "Rapidly-exploring random trees: A new tool for path planning". In: (1998).

[20] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *The international journal of robotics research* 30.7 (2011), pp. 846–894.

[21] Jauwairia Nasir et al. "RRT*-SMART: A rapid convergence implementation of RRT". In: *International Journal of Advanced Robotic Systems* 10.7 (2013), p. 299.

[22] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 2997–3004.

[23] Helen Oleynikova et al. "Continuous-time trajectory optimization for online UAV replanning". In: *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2016, pp. 5332–5339.

[24] Ahmed Hussain Qureshi and Yasar Ayaz. "Potential functions based sampling heuristic for optimal path planning". In: *Autonomous Robots* 40.6 (2016), pp. 1079–1093.

[25] Michael Otte and Emilio Frazzoli. "RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning". In: *The International Journal of Robotics Research* 35.7 (2016), pp. 797–822.

[26] Devin Connell and Hung Manh La. "Dynamic path planning and replanning for mobile robots using rrt". In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2017, pp. 1429–1434.

[27] Der-Tsai Lee. *Proximity and reachability in the plane.* University of Illinois at Urbana-Champaign, 1978.

[28] Franz Aurenhammer. "Voronoi diagrams—a survey of a fundamental geometric data structure". In: *ACM Computing Surveys (CSUR)* 23.3 (1991), pp. 345–405.

[29] Thomas H Cormen et al. *Introduction to algorithms.* MIT press, 2009.

[30] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[31] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. "ARA*: Anytime A* with provable bounds on sub-optimality". In: *Advances in neural information processing systems* 16 (2003), pp. 767–774.

[32] Daniel Harabor and Alban Grastien. "Online graph pruning for pathfinding on grid maps". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 25. 1. 2011.

[33] Dmitri Dolgov et al. “Path planning for autonomous vehicles in unknown semi-structured environments”. In: *The international journal of robotics research* 29.5 (2010), pp. 485–501.

[34] Alex Nash and Sven Koenig. “Any-angle path planning”. In: *AI Magazine* 34.4 (2013), pp. 85–107.

[35] Dave Ferguson and Anthony Stentz. “Using interpolation to improve path planning: The Field D* algorithm”. In: *Journal of Field Robotics* 23.2 (2006), pp. 79–101.

[36] Sven Koenig and Maxim Likhachev. “D^* lite”. In: *Aaai/iaai* 15 (2002).

[37] Anthony Stentz et al. “The focussed d^* algorithm for real-time replanning”. In: *IJCAI*. Vol. 95. 1995, pp. 1652–1659.

[38] Qi Zhang, Jiachen Ma, and Qiang Liu. “Path planning based quadtree representation for mobile robot using hybrid-simulated annealing and ant colony optimization algorithm”. In: *Proceedings of the 10th World Congress on Intelligent Control and Automation*. IEEE. 2012, pp. 2537–2542.

[39] Yijing Zhao, Zheng Zheng, and Yang Liu. “Survey on computational-intelligence-based UAV path planning”. In: *Knowledge-Based Systems* 158 (2018), pp. 54–64.

[40] Ugur Cekmez, Mustafa Ozsiginan, and Ozgur Koray Sahingoz. “A UAV path planning with parallel ACO algorithm on CUDA platform”. In: *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE. 2014, pp. 347–354.

[41] Joseph Horn, Brian Geiger, and Eric Schmidt. “Use of neural network approximation in multiple-unmanned aerial vehicle trajectory optimization”. In: *AIAA guidance, navigation, and control conference*. 2009, p. 6103.

[42] Matt Zucker et al. “Chomp: Covariant hamiltonian optimization for motion planning”. In: *The International Journal of Robotics Research* 32.9-10 (2013), pp. 1164–1193.

[43] Markus Ryll et al. “Efficient trajectory planning for high speed flight in unknown environments”. In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 732–738.

[44] Mark W Mueller, Markus Hehn, and Raffaello D’Andrea. “A computationally efficient motion primitive for quadrocopter trajectory generation”. In: *IEEE Transactions on Robotics* 31.6 (2015), pp. 1294–1310.

[45] Alex Nash, Sven Koenig, and Maxim Likhachev. “Incremental Phi*: Incremental any-angle path planning on grids”. In: (2009).

[46] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585 (2020), 357–362.

[47] Tin Lai, Weiming Zhi, and Fabio Ramos. “Occ-Traj120: Occupancy Maps with Associated Trajectories”. In: *CoRR* (2019).

[48] Ken Perlin. “An image synthesizer”. In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296.

[49] Richard P. Brent. “An algorithm with guaranteed convergence for finding a zero of a function”. In: *The Computer Journal* 14.4 (1971), pp. 422–425.

[50] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org.

[51] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. “PX4: A node-based multi-threaded open source robotics framework for deeply embedded platforms”. In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 6235–6240.

[52] Nathan Koenig and Andrew Howard. “Design and use paradigms for gazebo, an open-source multi-robot simulator”. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 3. IEEE, pp. 2149–2154.

[53] Jaeyoung Lim. *mavros_controllers - Aggressive trajectory tracking using mavros for PX4 enabled vehicles*. Mar. 2019. URL: https://doi.org/10.5281/zenodo.2652888.

[54] Taeyoung Lee, Melvin Leok, and N Harris McClamroch. “Geometric tracking control of a quadrotor UAV on SE (3)”. In: *49th IEEE conference on decision and control (CDC)*. IEEE. 2010, pp. 5420–5425.

[55] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272.

[56] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[57] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. “OpenVSLAM: A versatile visual SLAM framework”. In: *Proceedings of the 27th ACM International Conference on Multimedia*. 2019, pp. 2292–2295.

Appendix A

Phi* 2D algorithm

The algorithm Phi* 2D is from the work of Nash et al. [45]. We describe it in section 3.1.2 and in the algorithm 1.

Algorithm 1: Phi* 2D

```
Function Main:
    Initialize()
    ComputeShortestPath()
    if $g(s_{goal}) \neq \infty$ then
        return "Path found"
    else
        return "Path not found"

Function Initialize:
    $open = closed = \emptyset$
    InitializeVertex($s_{start}$)
    InitializeVertex($s_{goal}$)
    $g(s_{start}) = 0$
    $parent(s_{start}) = s_{start}$
    $open.insert(s_{start}, g(s_{start}) + h(s_{start}))$

Function InitializeVertex($s$):
    $g(s) = \infty$
    $parent(s) = NULL$
    $local(s) = NULL$
    $lb(s) = -\infty$
    $ub(s) = \infty$

Function ComputeShortestPath:
    while $open.topKey() < g(s_{goal}) + h(s_{goal})$ do
        $s = open.pop()$
        $closed = closed \cup \{s\}$
        foreach $s' \in nghbr_{vis}(s)$ do
            if $s' \notin closed$ then
                if $s' \notin open$ then
                    InitializeVertex($s'$)
                UpdateVertex($s$, $s'$)

Function UpdateVertex($s$, $s'$):
    $g_{old} = g(s')$
    ComputeCost($s$, $s'$)
    if $g(s') < g_{old}$ then
        if $s' \in open$ then
            $open.remove(s')$
        $open.insert(s', g(s') + h(s'))$

Function ComputeCost($s$, $s'$):
    if $lineofsight(parent(s), s')$ and
      $\Phi(s, parent(s), s') \in [lb(s), ub(s)]$ and
      $\angle(s', parent(s))$ is not a multiple of $45°$ then
        /* Path 2 */
        if
          $g(parent(s)) + distance(parent(s), s') < g(s')$
        then
            $parent(s') = parent(s)$
            $g(s') =$
              $g(parent(s)) + distance(parent(s), s')$
            $local(s') = s$
            $l =$
              $\min_{s'' \in nghbr_{cross}(s')}(\Phi(s', parent(s), s''))$
            $h =$
              $\max_{s'' \in nghbr_{cross}(s')}(\Phi(s', parent(s), s''))$

            $\delta = \Phi(s, parent(s), s')$
            $lb(s') = \max(l, lb(s) - \delta)$
            $ub(s') = \min(h, ub(s) - \delta)$
    else
        /* Path 1 */
        if $g(s) + distance(s, s') < g(s')$ then
            $parent(s') = s$
            $g(s') = g(s) + distance(s, s')$
            $local(s') = s$
            $lb(s') = -45°$
            $ub(s') = 45°$
```

We consider a 2D grid, with 8 neighbors cells. We use the following notation. $open$ is a priority queue, where $open.insert(s, x)$ insert an element s with the key x; $open.remove(s)$ removes the element s; $open.pop()$ removes and return the element with the smallest key; $open.topKey()$ returns the smallest key in the queue or ∞ if it does not exists. $lineofsight(s, s')$ checks if a straight line from s to s' is free from obstruction in the environment. $distance(s, s')$ returns the euclidean distance between the nodes s and s'. Finally, $nghbr(s)$ is the set of all the 8 neighbors of the node s if they exists, $nghbr_{vis}(s) \subseteq nghbr(s)$ is the subset of the neighbors of s which are in line of sight with s, and $nghbr_{cross}(s)$ is the set of crossbar neighbors of s. Crossbar neighbors are the 4 nodes located at the top, bottom, left and right of the node.

Appendix B

Phi* 3D algorithm

Phi* 3D is our generalization of the work of Nash et al. [45] to a 3D environment. Main differences are explained in section 3.1.3. In pseudo-code, the algorithm is the same as in appendix A, except for the `ComputeCost` function, described in algorithm 2. We use the same notations as in appendix A. Therefore, $nghbr(s)$ represent the 26 possible neighbors of a node s in a 3D grid and $nghbr_{cross}(s)$ is the set of the 12 corners of the three squares crossing the node s. In figure 3.3, it corresponds to the grey points. In other words, if the cell s is located at `(0, 0, 0)` in a 3D grid with a size length of 1, the set of its crossbar neighbors will be $nghbr_{cross}(s)$ = `{ (0,1,1); (0,1,-1); (0,-1,-1); (0,-1,1); (1,1,0); (-1,1,0); (-1,-1,0); (1,-1,0); (1,0,1); (-1,0,1); (-1,0,-1); (1,0,-1) }`.

Algorithm 2: Phi* 3D

```
Function ComputeCost($s$, $s'$):
    if $lineofsight(parent(s), s')$ and $\varphi(s, parent(s), s') \in [lb_\varphi(s), ub_\varphi(s)]$ and
      $\theta(s, parent(s), s') \in [lb_\theta(s), ub_\theta(s)]$ and $(s', s, parent(s))$ are not aligned then
        /* Path 2 */
        if $g(parent(s)) + distance(parent(s), s') < g(s')$ then
            $parent(s') = parent(s)$
            $g(s') = g(parent(s)) + distance(parent(s), s')$
            $local(s') = s$
            $l_\varphi = \min_{s'' \in nghbr_{cross}(s')}(\varphi(s', parent(s), s'')$
            $l_\theta = \min_{s'' \in nghbr_{cross}(s')}(\theta(s', parent(s), s'')$
            $h_\varphi = \max_{s'' \in nghbr_{cross}(s')}(\varphi(s', parent(s), s'')$
            $h_\theta = \max_{s'' \in nghbr_{cross}(s')}(\theta(s', parent(s), s'')$
            $\delta_\varphi = \varphi(s, parent(s), s')$
            $\delta_\theta = \theta(s, parent(s), s')$
            $lb_\varphi(s') = \max(l_\varphi, lb_\varphi(s) - \delta_\varphi)$
            $lb_\theta(s') = \max(l_\theta, lb_\theta(s) - \delta_\theta)$
            $ub_\varphi(s') = \min(h_\varphi, ub_\varphi(s) - \delta_\varphi)$
            $ub_\theta(s') = \min(h_\theta, ub_\theta(s) - \delta_\theta)$
    else
        /* Path 1 */
        if $g(s) + distance(s, s') < g(s')$ then
            $parent(s') = s$
            $g(s') = g(s) + distance(s, s')$
            $local(s') = s$
            $lb_\varphi(s') = \min_{s'' \in nghbr_{cross}(s')}(\varphi(s', s, s'')$
            $lb_\theta(s') = \min_{s'' \in nghbr_{cross}(s')}(\theta(s', s, s'')$
            $ub_\varphi(s') = \max_{s'' \in nghbr_{cross}(s')}(\varphi(s', s, s'')$
            $ub_\theta(s') = \max_{s'' \in nghbr_{cross}(s')}(\theta(s', s, s'')$
```

Appendix C

Angles computations

Here, we provide the implementation for the computation of angles. The implementation is done in Python, thus, performance was not a priority.

The following is the implementation of the Φ function for Phi* 2D, as described by [45].

```
import math

# Computes the angle formed by three Phi* graph nodes.
# a, b, c are the position of the three nodes as
# defined in the original paper.
# Each point is defined as a tuple of 2D coordinates [x, y].
# Returns the angle in degrees.
#
# Python implementation by Remy Hidra.
def phi_2d(a,b,c):
    angle = - math.atan2(a[1]-b[1], a[0]-b[0])
            + math.atan2(c[1]-b[1], c[0]-b[0])
    angle *= 180. / math.pi

    # Set the angle between [-180; 180]
    if angle > 180:
        angle = - (180 - (angle % 180))

    return angle
```

From our 3D generalization explained in section 3.1.3, we developed a new version of this function. Φ now returns a tuple (φ, θ) which is made of φ the component in the $(\vec{x}, \vec{y})$ plane, and θ the component in the vertical plane along $\vec{z}$.

```
import math

# Computes the angles formed by three Phi* 3D graph nodes.
# a, b, c are the position of the three nodes.
# Each point is defined as a tuple of 3D coordinates [x, y, z].
# Return the angles in degrees
#
# Python implementation by Remy Hidra.
def phi_3d(a,b,c):
    p = - math.atan2(a[1]-b[1], a[0]-b[0])
        + math.atan2(c[1]-b[1], c[0]-b[0])
    p *= 180. / math.pi
    if p > 180: # Set phi between [-180; 180]
        p = - (180 - (p % 180))

    d1 = math.sqrt((a[0]-b[0]) ** 2 + (a[1]-b[1]) ** 2)
    d2 = math.sqrt((c[0]-b[0]) ** 2 + (c[1]-b[1]) ** 2))
    t = - math.atan2(a[2]-b[2], d1)
        + math.atan2(c[2]-b[2], d2)
    t *= 180. / math.pi
    if t > 180: # Set theta between [-180; 180]
        t = - (180 - (t % 180))

    return p, t
```