Budapest University of Technology and Economics
Faculty of Transportation Engineering and Vehicle Engineering
Department of Control for Transportation and Vehicle Systems

Trajectory design for autonomous drones navigating in a dynamic environment

Máté Palkovits

Scientific Conference of Students

Consultants:
Dr. Tamás PÉNI
Senior research fellow
SZTAKI Systems and Control Lab

Dr. Roland TÓTH
Senior research fellow / Associate prof.
SZTAKI Systems and Control Lab
Eindhoven University of Technology

Budapest, 2022

Összefoglaló

Kutatásom célja egy kis számításigényű, elosztott pályatervezési algoritmus kifejlesztése autonóm drónok számára, rögzített és mozgó akadályokkal zsúfolt térben történő gyors és biztonságos navigáció megvalósítására. Az eljárás lehetséges alkalmazási területe gyártórendszerek monitorozása, városi környezetben történő navigáció, illetve felderítési és mentési feladatok támogatása.

A tervezés kezdő lépése egy tervezési gráf létrehozása kizárólag a statikus objektumok figyelembe vételével. A drónok egyedi pályáinak megtervezése ezután kezdődik. A dolgozat első felében feltételezzük, hogy a mozgó objektumok pályája előre ismert. A drónok egymás után, előre meghatározott sorrendben tervezik a pályájukat, minden drón a sorrendben előtte állókat mozgó objektumnak tekinti. Minden drón két fő tervezési lépést hajt végre: elsőként, konstans pályamenti sebességet feltételezve egy módosított, idővel paraméterezett gráfkereső algoritmus (A*, Dijkstra) felhasználásával minimális idő alatt bejárható utat keres a kiindulási és a célpont között. A keresést több, előre rögzített pályamenti sebességre párhuzamosan futtatjuk le. A konstans sebesség feltételezése azért lényeges, mert így a keresés kis számítási idővel végrehajtható. Ugyanakkor, épp emiatt előfordulhat, hogy a vizsgált fix sebességértékek mellett nem létezik ütközésmentes útvonal. Ennek elkerülésére a keresés során megengedjük az ütközést, de alkalmas súlyfüggvény megválasztásával büntetjük. Az ütközésmentes pályát a tervezés második lépésében hozzuk létre: ebben a lépésben a korábban kapott gráfútra, pontosabban az arra illesztett folytonos spline görbére alkalmas sebességprofilt tervezünk, ahol az ütközések elkerülését szigorú korlátozásként írjuk elő. Mivel a térbeli pálya már adott, ez a tervezési lépés kevert egészértékű kvadratikus optimalizálási feladatként (MIQP) írható fel, amely hatékonyan megoldható a rendelkezésre álló szoftvercsomagok (pl. Gurobi) segítségével.

A dolgozat második felében a pályatervezési eljárást továbbfejlesztjük: képessé tesszük előre nem ismert mozgású akadályok kezelésére is. Feltételezzük, hogy az ismeretlen mozgású objektumok pozíciójáról megbízható mérési adat áll rendelkezésre. Ezt felhasználva, adott időlépésenként becslést adunk az objektum jövőbeli mozgására vonatkozóan. Ha ennek ismeretében ütközés feltételezhető, az érintett kvadkopter pályáját módosítjuk. Az újratervezés a korábban megtervezett útvonal lokális megváltoztatásával történik. Ennek során először több elkerülő útvonaljelöltet hozunk létre, melyek közül a későbbi ütközés valószínűségét és a többi kvadkopter mozgását figyelembe véve választunk.

Az algoritmust Python nyelven implementáljuk, működését először MuJoCo szimulációs környezetben vizsgáljuk. Ezt követően az implementációt Bitcraze Crazyflie 2.1 miniatűr kvadkopterekre is elvégezzük, az eljárás alkalmazhatóságát valós rendszeren is teszteljük, illetve demonstráljuk.

Abstract

The aim of my research is to create an efficient, low complexity trajectory planning algorithm for autonomous quadcopters that have to navigate at high speed in a cluttered environment, among static and dynamic obstacles. The possible field of application of the proposed procedure could be monitoring of production systems, navigation in an urban environment or support for surveillance and rescue tasks.

The algorithm starts with the construction of a planning graph that takes only the static obstacles into consideration. Then the drones start designing their trajectories one after another, in a predefined order. Each drone considers the others that precede it in the design sequence as moving obstacles. For simplicity, in the first part of the work, we assume that the trajectories of all moving obstacles are a-priori known. The drones perform two design steps: first, assuming constant velocity, a modified, time-parameterized shortest path algorithm (A*, Dijkstra) is applied to find a feasible route between the starting and the target points. By fixing the velocity, the complexity of path finding can be greatly simplified. We perform multiple searches in parallel with different velocity values to increase the probability of success, however it is still possible that a collision-free route cannot be found. To avoid the failure, we allow collisions during the search step, but penalize them by a suitably chosen cost function.

The collision-free trajectory is created in the second step of the planning algorithm: in this step, a velocity profile is designed for the previously constructed route, the collision avoidance is forced by strict constraints. Since the spatial trajectory is already given, this design step can be formulated as a mixed-integer quadratic optimization problem (MIQP), which can be solved efficiently by the available software packages (e.g. Gurobi).

In the second part of the work, we improve the trajectory planning procedure by making it capable of handling obstacles whose motion is not known a-priori. We assume only the position of these objects can be detected. Collecting this data in a finite time window, the future movement of the obstacles are predicted. If a collision is expected, the trajectory of the affected quadcopter is modified. For this, multiple evasive route candidates are created, from which one is selected based on the probability of a future collisions and the movement of the other quadcopters.

The algorithm is implemented in Python, and it is analyzed first in the MuJoCo simulation environment. The implementation is then carried out for Bitcraze Crazyflie 2.1 miniature quadcopters and the applicability of the procedure is tested and demonstrated in real flight experiments as well.

Contents

List of Figures

1 Introduction

Quadcopters are popular robotic platforms thanks to their simplicity and agility, and wide range of applications. Most research quadcopters are large enough to carry cameras and smartphone-grade computers, but they are also expensive and require a large space to operate safely. In our work we use miniature drones which can safely maneuver in small and dense environments but has reduced on board computational capacity and carrying weight. In the last years Unmanned Aircraft Systems (UAS) have been widely used in many applications for industrial use. Automated drones can be used for security, surveillance, emergency response and infrastructure inspection. Miniature quadcopters are great for inspecting hard to reach or hazardous indoor areas of a factory.

There are various techniques for navigating drones which can be generally classified as global path planning, local path planning, and hybrid. In our study we use a hybrid method which constructs a base trajectory for the drones to avoid a-priori known obstacles with global path planning and unknown obstacles with local path planning. There are great solutions which handles multiple drones in partially known environments [4], but they either assume that the drones themselves calculates their paths or they controlled by a ground PC with reference position or velocity commands. The first method expects a larger computational capacity onboard the drones and the later one demands continuous communication with the command PC.

Our method offers a solution where the computation is done by the ground control PC, but do not need to constantly communicate with the drones. We achieve this by constructing entire trajectories which sent in one data package to the drones to evaluate and follow them. We use time-dependent shortest path algorithm [3] in search for a time minimal path from start to goal positions between static obstacles. We formulate the problem of avoid the moving obstacles with movements known a-priori as a mixed-integer quadratic optimization problem (MIQP) [10], which constructs a velocity profile for the found path. We handle the unknown obstacles by generating various path candidates that diverge from the original path to find the most optimal from them for an evasive maneuver. This solution was inspired by methods created for ground vehicles where optional paths were generated between the sides of the road for obstacle avoidance tasks [13]. However, drones are not limited in this respect, so a more flexible solution was needed in order to take advantage of this freedom.

2 Problem formulation

A 3D navigation problem is considered, where multiple drones is required to navigate safely and time efficiently in a partially known environment $\mathcal{E} \subset \mathbb{R}^3$ defined as a three-dimensional Euclidean space. The $\mathcal{E}$ environment contains a set of n obstacles, these obstacles can either be static with known position $\mathcal{O}^s$ (defined in Section 3.2), dynamic with a-priori known motion trajectory (e.g. the other drones) $\mathcal{O}^k$(defined in Section 3.1), or dynamic with unknown motion $\mathcal{O}^u$ (defined in Section 3.4).

The main objective is to safely guide N_d number of drones from their starting positions which can be in any points of the environment unoccupied by an obstacle in $t = 0$ time: $p^{start} = \{p_1^{start}, p_2^{start}, \cdots, p_{N_\mathrm{d}}^{start}\} \subset \mathcal{E} \setminus \{\mathcal{O}^s, \mathcal{O}^k(0), \mathcal{O}^u(0)\},\ p_i^{start} \subset \mathbb{R}^3,\ i \in \mathbb{I}_1^{N_\mathrm{d}} = \{b \in \mathbb{Z} \mid 1 \leq b \leq N_\mathrm{d}\}$, to their goal positions which points only limited by the static obstacles: $p^{goal} = \{p_1^{goal}, p_2^{goal}, \cdots, p_{N_\mathrm{d}}^{goal}\} \subset \mathcal{E} \setminus \mathcal{O}^s,\ p_i^{goal} \subset \mathbb{R}^3,\ i \in \mathbb{I}_1^{N_\mathrm{d}}$. To safely navigate between these points the drones are executing trajectories $\tau : \mathbb{R} \to \mathbb{R}^3$, where the trajectory of the i-th drone is $\tau_i = [x_i(s_i(t)), y_i(s_i(t)), z_i(s_i(t))]^T,\ i \in \mathbb{I}_1^{N_\mathrm{d}},\ 0 \leq t \leq T_i$ which are expressed in Cartesian coordinates where $s_i(t) \in \mathbb{R}^+$ is the covered distance along the i-th path at t time and $T_i \in \mathbb{R}^+$ is the flight time of the i-th drone. These trajectories have to start in p^{start} and end in p^{goal} which means:

$$\tau_i(0) = p_i^{start},\ i \in \mathbb{I}_1^{N_\mathrm{d}} \tag{1}$$

$$\tau_i(T_i) = p_i^{goal},\ i \in \mathbb{I}_1^{N_\mathrm{d}} \tag{2}$$

Further requirements for τ_i are that T_i has to be minimal while τ_i has to avoid $\mathcal{O}^s$ and $\mathcal{O}^k$ in respect of (5). Every τ_i trajectory is constructed as an arch length parameterized spline $\mathcal{S}_i = [x_i(s), y_i(s), z_i(s)]^T$ and an arch length - time function $s_i(t),\ 0 \leq t \leq T_i$ [12].

We assume that the drones have onboard tracking controller, so they can follow their $\tau(t)$ trajectory if the following constrains are met:

$$0 \leq \dot{s}(t) \leq v_{max} \tag{3}$$

$$-a_{max} \leq \ddot{s}(t) \leq a_{max} \tag{4}$$

where $\dot{s}(t) \in \mathbb{R}$ is the velocity and $\ddot{s}(t) \in \mathbb{R}$ is the acceleration of a drone. Note that the drones can only fly forward along their paths as (3) shows.

The shortest distance between the i-th drone and the j-th obstacle at time t is given by $d(i, j, t),\ i \in \mathbb{I}_1^{N_\mathrm{d}},\ j \in \mathbb{I}_1^{N_\mathrm{o}} = \{b \in \mathbb{Z} \mid 1 \leq b \leq N_\mathrm{o}\}$ where N_o is the number of all the obstacles. A safety requirement for the drones is defined as keeping a safe distance $d_{\mathsf{safe}} > 0$ from all obstacles according to the following equation:

$$d(i, j, t) \geq d_{\mathsf{safe}} \quad \forall t \in \mathbb{R},\ i \in \mathbb{I}_1^{N_d},\ j \in \mathbb{I}_1^{N_o} \tag{5}$$

From this point we will refer the violation of (5) as collision.

While the drones fly, they generate a large, fast-moving volume of air underneath their rotors called downwash [7]. The downwash force is large enough to cause a catastrophic loss of stability when one drone flies underneath another. We model downwash constraints as inter-robot collision constraints by treating each robot as a vertical capsule with radius $r \in \mathbb{R}^+$ and height $h \in \mathbb{R}^+$ where the drone is at the center as it can be seen in Figure 1.

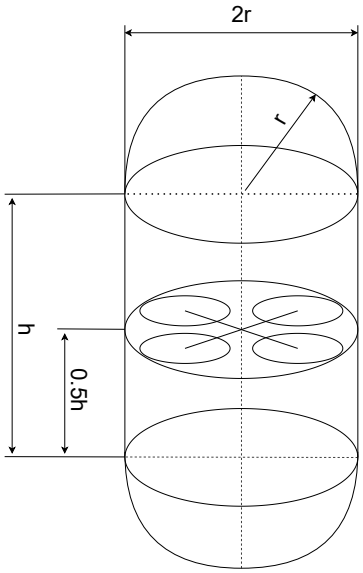


Figure 1: Convex capsule around the drone to help avoiding downwash during trajectory design

This way when the algorithm makes $\tau_i(t)$ it can handle the i-th drone as a sphere and avoid the other drones in respect to the downwash force while does not reduce its capability to fly close to $\mathcal{O} \setminus \mathcal{O}^k$.

3 Proposed solution

3.1 Main concept

The algorithm has an initial map of the environment containing information about the positions, shapes and sizes of $\mathcal{O}^s$. At first the algorithm constructs a graph $\mathcal{G} = \{V, E\}$ where the N_v number of vertex is defined as $V_i = \{p_{\text{v},i}, t_{\text{v},i}, c_{\text{c},i}, b_i\},\ p_{\text{v},i} \in \mathcal{E},\ t_{\text{v},i} \in t,\ c_\text{c} \in \mathbb{R}^+,\ a \in \mathbb{R}^+,\ i \in \mathbb{I}_1^{N_\text{v}} = \{a \in \mathbb{Z} \mid 1 \leq a \leq N_\text{v}\}$ (explained in detail in Section 3.3.1) and the $E_{i,j}$ edge defined as line segment between the $V_i,\ V_j$ vertices. The $\mathcal{G}$ graph is constructed in the $\mathcal{E}$ environment and avoids the $\mathcal{O}^s$ static obstacles as it can be seen in Figure 2. It is done by randomly generating N_v number of vertices in the in the flying area. We connect the elements of V with edges E using Delaunay triangulation. This ensures that the edges connect just the neighboring vertices and does not cross each other. After that we remove the elements of E those are in intersection with any $\mathcal{O}^s$. We explain this method in more detail in Section 3.2.

In the next step which is covered in Section 3.3 we create a τ for each drone sequentially. For the i-th drone we create a base path $\mathcal{P} \subset \mathcal{G}$ between p_i^{start} and p_i^{goal} with a graph search using a time-dependent shortest path algorithm [3]. We modify the algorithm to enable it to handle moving obstacles assuming they movement is known. To reduce the complexity of the graph search we assume that the velocity of the drones is constant, while we search for a time minimal path with no collisions with the moving obstacles. However, with a constant velocity the existence of a collision free path is not guaranteed therefore we perform the search with multiple velocities and do not forbid the collisions yet just penalize them. After the graph search creates $\mathcal{P}$ we need to simplify it as it shown in Figure 5 to create a faster and more direct path $\mathcal{P}^f \subset \mathcal{P}$ between p_i^{start} and p_i^{goal}. With this method the algorithm can create $\mathcal{P}^f$ even if a collision free path does not exist with a constant velocity. We fit a B-spline to the vertices of $\mathcal{P}^f$ which parameterized by the arc length of the spline. Finally, we use MIQP to calculate a velocity profile along the spline which guaranties the avoidance of $\mathcal{O}^k$. After τ_i is constructed for the i-th drone, it is added along with r and h to define $\mathcal{O}^k_{(i)} = \{\tau_i(t), r, h\}$. In case of n number of known dynamic obstacle apart from the drones the $\mathcal{O}^k$ set can be defined as $\mathcal{O}^k = \{\mathcal{O}^k_1, \cdots, \mathcal{O}^k_n, \cdots, \mathcal{O}^k_{n+i}\}$ after the construction of τ_i.

While the drones are flying, we assume to have information only about the positions and dimensions of $\mathcal{O}^u$, therefore we must predict their future positions to predict the possibility of a future collision. If a collision is predicted in the future we construct a number of possible evasive path candidates (see in Figure 13). These paths start ahead of the drone to allow enough time for the calculation and the communication and return to the original path after the obstacle is avoided. The most optimal path is selected based on the chance of future collisions, interference with other drones, and the time required to fly.

Our algorithm has three main parts: (i) the *scene construction*, responsible for the description of the static environment where the drones fly and the graph which the *trajectory planner* uses for the path search; (ii) the *trajectory planner* that designs the motion trajectories for the drones (iii) the *path checker* that predict possible collision with $\mathcal{O}^u$ and provides modification of the motion trajectories to avoid them.

Algorithm 1 main

construct $\mathcal{G}$ with *scene construction* based on $\mathcal{O}^s$ ▷ see in Algorithm: 2
for $i \in \mathbb{I}_1^{N_\mathrm{d}}$ **do**
  construct $\mathcal{P}_i$ with *modified A** graph search ▷ see in Algorithm: 3
  fit B-spline to $\mathcal{P}_i$ to obtain $\mathcal{S}_i$
  construct τ_i with *trajectory planner* based on $\mathcal{S}_i$ in respect of $\mathcal{O}^k$ ▷ see in Algorithm: 4
  $\mathcal{O}_i^k \leftarrow \{\tau_i(t), r, h\}$
  add $\mathcal{O}_i^k$ to $\mathcal{O}^k$
end for
Start executing trajectories
repeat
  measure current positions of the obstacles in $\mathcal{O}^u$ to obtain p_c
  predict future positions of the obstacles in $\mathcal{O}^u$ to obtain p_f
  for $i \in \mathbb{I}_1^{N_\mathrm{d}}$ **do**
    predict collision between i-th drone and $\mathcal{O}^u$ based on p_f ▷ see in Algorithm: 5
    if collision predicted **then**
      update τ_i with *path checker* in respect of $\mathcal{O}^u$ and $\mathcal{O}^k$
      update the trajectory in $\mathcal{O}_i^k$ with τ_i
    end if
  end for
until Trajectories are executed

3.2 Scene construction

The set of n static obstacle is given by $\mathcal{O}^s = \{\mathcal{O}_1^s, \cdots, \mathcal{O}_n^s\}$ and every obstacle is assumed to be a square pole described by $\mathcal{O}_i^s = \{x_{s,i}, y_{s,i}, a_{s,i}, b_{s,i}, h_{s,i}\},\ i \in \{1, \cdots, n\}$ where x and y are the coordinates of the middle of the base, and $a, b, h \in \mathbb{R}^+$ are the length, width and height of the obstacle increased by the d_{safe} safety distance and the r radius of the drones as:

$$a_{s,i} = a'_{s,i} + 2 \cdot (d_{safe} + r) \tag{6}$$

$$b_{s,i} = b'_{s,i} + 2 \cdot (d_{safe} + r) \tag{7}$$

$$h_{s,i} = h'_{s,i} + d_{safe} + r \tag{8}$$

where $a'_{s,i}, b'_{s,i}, h'_{s,i}$ are the true dimensions of the i-th obstacle. With this size expansion we ensure that the V vertices and E edges are constructed in a sufficient distance from the $\mathcal{O}^s$ static obstacles.

We randomly generate $V = \{V_1, \cdots, V_{N_\mathrm{v}}\}$ vertices within $\mathcal{E}$ environment, where $V_i = [x, y, z]^T,\ i \in \mathbb{I}_1^{N_\mathrm{v}}$ are expressed in Cartesian coordinates. For V vertices we define two criteria, the vertices needs to be outside of the static obstacles:

$$V \subset \mathcal{E} \setminus \mathcal{O}^s \tag{9}$$

and there has to be a minimal distance between them, because the short distances are just unnecessarily increase the complexity of the graph (e.g. in a drone arena with dimensions of 3m x 3m x 3m there is no need for the vertices to be within millimeters of each other):

$$d_\mathrm{T} \leq ||V_i - V_j||, \quad i \in \mathbb{I}_1^{N_\mathrm{v}},\ j \in \mathbb{I}_i^{N_\mathrm{v}} \tag{10}$$

where the $||\cdot||$ is the standard Euclidean norm of a vector in $\mathbb{R}^3$. We expand V with fix vertices V_0 which represent e.g. charging pads or target destinations. Because we assume all of the V_0 fix vertices are purposely placed at certain points only the criterion defined by (9) applies to them.

We connect the V vertices by E edges using Delaunay triangulation. This way only the neighboring vertices are connected with edges that do not cross each other. After the triangulation there will be edges witch, we need to leave because they goes through the $\mathcal{O}^s$ static obstacles.

Algorithm 2 scene construction

input: Number of vertices to be generated N_v
Threshold d_T
Fix vertices V_0
Static obstacles $\mathcal{O}^s$
Environment $\mathcal{E}$
output: A $\mathcal{G}$ graph for the planning algorithm which does not intersect with the static obstacles
$V' \leftarrow RandVertices(N_\mathrm{v}, d_\mathrm{T}, \mathcal{E} \setminus \mathcal{O}^s)$
$V \leftarrow V_0 \cup V'$
$E \leftarrow DelaunayGraph(V)$
$i \leftarrow 1$
repeat
    if $Intersect(E_i, \mathcal{O}^s) = True$ **then**
        $E \leftarrow E \setminus E_i$
    end if
    $i \leftarrow i + 1$
until $i > \#E$
$\mathcal{G} \leftarrow \{V, E\}$

We define the $RandVertices(N_\mathrm{v}, d_\mathrm{T}, \mathcal{E} \setminus \mathcal{O}^s)$ function to generate N_v number of vertices that satisfy (9-10). The $DelaunayGraph(V)$ function connects the V vertices based on Delanuay triangulation to construct the E edges of the graph. We define the Intersect() function to give a $True$ value if an edge intersects with $\mathcal{O}^s$ static obstacles to locate the intersecting E edges.

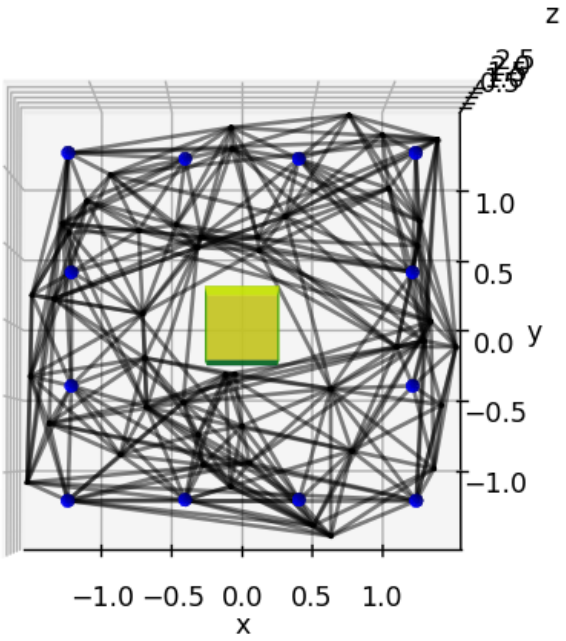


Figure 2: Generated graph.

3.3 Trajectory planner

In this section we show how to generate collision-free trajectories for the drones assuming we have information about the trajectories of the moving obstacles. We design the trajectory of the drones sequentially in a predefined order where each drone handles the previous ones as part of the $\mathcal{O}^k$ moving obstacles. The generation of one trajectory has two phases: *path planning*, *velocity planning*.

3.3.1 Path planning

The *path planning* based on a time dependent A* graph search [15] which we modified in a way that it can take into account $\mathcal{O}^k$ moving obstacles when calculating costs. This modification is based on that the algorithm has information about the trajectories, and r, h dimensions of $\mathcal{O}^k$ moving obstacles. When performing the path finding we assume the drones fly with a v_c constant velocity, which reduce the computational complexity, but there is a risk that a collision free path cannot be found. We increase the chance of finding a collision free path by performing multiple searches in parallel with different velocity values. Also, in this step we allow collisions during the search step, but penalize them by a suitably chosen cost function, to ensure the success of the path finding even in cases when a collision is unavoidable with constant velocities. In the algorithm the cost value of a V_j vertex which is neighboring to the V_i vertex is calculated as:

$$c_j = c_i + t_{\mathrm{e},j} + t_{\mathrm{g},j} + c_{\mathrm{c},j} \tag{11}$$

where $c_i \in \mathbb{R}^+$ is the cost assigned to the V_i vertex, $t_{\mathrm{e},j} \in \mathbb{R}^+$ the time needed for the drone to fly from V_i vertex to V_j vertex, $t_{\mathrm{g},j} \in \mathbb{R}^+$ is the time needed for the drone to fly to the goal position assigned to it in a straight line from the V_j vertex and $c_{\mathrm{c},j} \in \mathbb{R}^+$ a penalty which based on the possible collisions with $\mathcal{O}^k$ on the $E_{i,j}$ edge. The vertices contain information about their position p_v, which assigned to the during the scene construction; the t_v time needed for the drone to reach them from the start position, following the path

with a given constant velocity; their c_{c} cost value which needed for determining the most optimal path between the goal and start positions; the b index of the vertex before them in the found path. After the algorithm calculates the $t_{\mathrm{v},i}$ and $c_{\mathrm{c},j}$ values of V_j from V_i, if the new $c_{\mathrm{c},j}$ is less than the current cost of V_j, the values get updated.

Now we show how to calculate the cost values of the V_j vertices that are neighbouring to V_i vertex. In the algorithm t_{e} flight time cost of a V_j adjacent vertex of V_i vertex can be calculated as:

$$t_{\mathrm{e},j} = \frac{||p_{\mathrm{v},j} - p_{\mathrm{v},i}||}{v_{\mathrm{r}}},\ i \in \mathbb{I}_1^{N_{\mathrm{v}}},\ j \in \mathcal{A}_i \tag{12}$$

where v_{r} is the constant velocity and $\mathcal{A}_i$ denotes the set of index values of the adjacent vertices of the i-th vertex. The t_{g} time to fly from the neighbouring vertex to the p_k^{goal} goal position of the k-th drone can be calculated as:

$$t_{\mathrm{g},j} = \frac{||p_{\mathrm{v},i} - p_k^{goal}||}{v_{\mathrm{r}}},\ j \in \mathbb{I}_1^{N_{\mathrm{v}}} \tag{13}$$

We assign each vertex a t_{v} reaching time which represent the flight time to the vertex from the p_k^{start} start position with v_{r} constant velocity:

$$t_{\mathrm{v},j} = t_{\mathrm{v},i} + t_{\mathrm{e},j},\ i \in \mathbb{I}_1^{N_{\mathrm{v}}},\ j \in \mathcal{A}_i \tag{14}$$

note that $c_{\mathrm{t},1} = 0$ and $t_{\mathrm{v},1} = 0$. The $c_{\mathrm{c},j}$ collision cost based on the extent to which an edge leading to the j-th vertex from the i-th vertex cuts into the $\mathcal{O}^k$ obstacles, hence how difficult will be to avoid the obstacles with altering the velocity in the *velocity planning* step. We evaluate the trajectories of $\mathcal{O}^k$ in the $t_{\mathrm{v},i} \leq t \leq t_{\mathrm{v},j}$ time interval to get their positions in the given time window. When calculating the distances, the centers of the $\mathcal{O}^k$ obstacles defined by vertical line segments as:

$$p_{t,k} + \alpha \cdot \bar{h},\ t_{\mathrm{v},i} \leq t \leq t_{\mathrm{v},j},\ 0 \leq \alpha \leq 1, k \in \{1, \cdots, \#\mathcal{O}^k\} \tag{15}$$

where p_t points are at $0.5 \cdot h$ bellow of the positions of the obstacle and $\bar{h} = [0, 0, h]$ is a vertical vector pointing upwards with a h length shown in Figure 3.

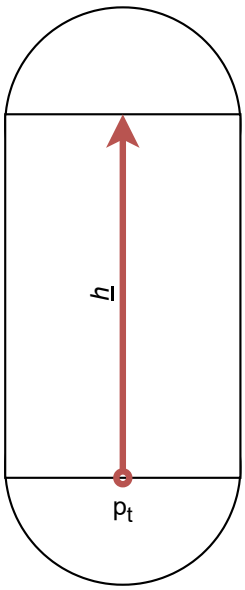


Figure 3: Center of an obstacle with the $\bar{h}$ vector

This h length is the height of the cylinder which we used to define the downwash effect (see Figure 1). The edges connecting the i-th vertex to the adjacent vertices can be defined by a line segments as:

$$p_{\mathrm{v},i} + \beta \cdot \bar{e}_j, \;\; i \in \mathbb{I}_1^{N_\mathrm{v}}, \;\; j \in \mathcal{A}_i, \;\; 0 \leq \beta \leq 1 \tag{16}$$

where $\bar{e}_j = p_{\mathrm{v},i} - p_{\mathrm{v},j}$ is a vector that points from the i-th vertex to the j-th adjacent vertex. We calculate the minimal distance $d_\mathrm{m} \in \mathbb{R}^+$ shown in Figure 4 between the k-th $\mathcal{O}^k$ obstacle and the edge which connects the V_i and V_j vertices in a given time grid as:

$$d_\mathrm{m} = \min_{\substack{0 \leq \alpha \leq 1 \\ 0 \leq \beta \leq 1 \\ t_{\mathrm{v},i} \leq t \leq t_{\mathrm{v},j}}} ||(p_t + \alpha \cdot \bar{h}) - (p_{\mathrm{v},i} + \beta \cdot \bar{e}_j)|| \tag{17}$$

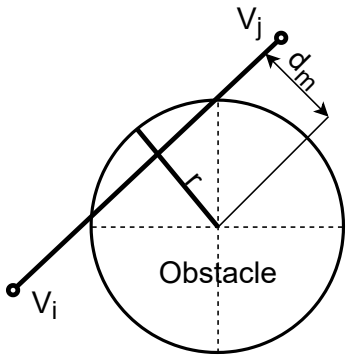


Figure 4: 2D representation of the minimal distance between the $E_{i,j}$ edge and an obstacle

If the d_m minimal distance is larger than the sum value of the radii of the drone $r_\mathrm{o} \in \mathbb{R}^+$ and the l-th $\mathcal{O}^k$ obstacle $r_\mathrm{o} \in \mathbb{R}^+$ plus the d_safe safety distance, then there is no collision and the l-th collision cost assigned to the j-th vertex is zero:

$$d_\mathrm{m} > (r_\mathrm{d} + r_\mathrm{o} + d_\mathrm{safe}) \Rightarrow c_{\mathrm{c},j,k} = 0 \tag{18}$$

However if it is less or equal, then the cost value is defined based on how deeply the edge connecting V_i and V_j vertices cuts into the safe zone of the l-th $\mathcal{O}^k$ obstacle:

$$d_m \leq (r_\mathrm{d} + r_\mathrm{o} + d_\mathrm{safe}) \Rightarrow c_{\mathrm{c},j,l} = c_{\mathrm{c,min}} + (1 - \frac{d_\mathrm{m}}{r_\mathrm{o} + d_\mathrm{safe}}) \cdot (c_{\mathrm{c,max}} - c_{\mathrm{c,min}}) \tag{19}$$

where $c_{\mathrm{c,min}} \in \mathbb{R}^+$ and $c_{\mathrm{c,max}} \in \mathbb{R}^+$ are heuristic values which define the minimal and maximal cost of a collision. The c_min tells the seriousness of the fact of the collision, while c_max tells the weight of the degree of the collision. The full cost of the collision assigned to the j-th vertex is the sum value that calculated for each moving obstacle:

$$c_{\mathrm{c},j} = \sum_{l=1}^{\#\mathcal{O}^k} c_{\mathrm{c},j,l} \tag{20}$$

Now we know how the algorithm calculates the cost values of the vertices, the Algorithm 3 shows the process of finding the time minimal path. The Algorithm 3 is performed

multiple times in parallel with different velocity values to increase the probability of finding a collision free path. At the end of the first phase, we will have a base path for the k-th drone as $\mathcal{P} = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\} \subset \mathcal{G}$ with $n \geq 2$ number of vertices where $\mathcal{P}_1 = p_k^{start}$ and $\mathcal{P}_n = p_k^{goal}$.

Algorithm 3 modified A*

input: planning graph $\mathcal{G}$; start and goal positions of the k-th drone p_k^{start}, p_k^{goal}; constant velocity v_c; moving obstacles with known trajectories $\mathcal{O}^k$; collision cost hyper parameters $c_{\mathrm{c,min}}$ and $c_{\mathrm{c,max}}$
output: a time minimal path $\mathcal{P}$
for $i \in \{1, \cdots, \#V\}$ **do**
    $c_{\mathrm{c},i} \leftarrow inf$
    $t_{\mathrm{v},i} \leftarrow inf$
    $b_i \leftarrow undefined$
end for
$c_{\mathrm{c},start} \leftarrow 0$
$t_{\mathrm{v},start} \leftarrow 0$
$Q \leftarrow V$
repeat
    $q \leftarrow$ vertex in Q with smallest c_c
    $Q \leftarrow Q \setminus q$
    for u adjacent to q **do**
        $i \leftarrow$ index of q vertex
        $j \leftarrow$ index of u vertex
        solve (12) with v_c; $p_{\mathrm{v},i}$; $p_{\mathrm{v},j}$ to obtain $t_{\mathrm{e},new}$
        solve (13) with v_c; $p_{\mathrm{v},j}$; p_k^{goal} to obtain $t_{\mathrm{g},new}$
        solve (14) with $t_{\mathrm{e},new}$; $t_{\mathrm{v},i}$ to obtain $t_{\mathrm{v},new}$
        solve (15-20) with $p_{\mathrm{v},i}$; $t_{\mathrm{v},i}$; $p_{\mathrm{v},j}$; $t_{\mathrm{v},new}$; $\mathcal{O}^k$; $c_{\min}$; $c_{\max}$ to obtain $c_{\mathrm{c},new}$
        solve (11) with $t_{\mathrm{e},new}$; $t_{\mathrm{g},new}$; $c_{\mathrm{c},new}$; c_j to obtain c_{new}
        if $c_{new} < c_j$ **then**
            update c_j with c_new
            update $t_{\mathrm{v},j}$ with $t_{\mathrm{v},new}$
            update b_j with i
        end if
    end for
until $q = V_{goal}$
$\mathcal{P} \leftarrow GetPath(V)$

where the *start* and *goal* are the indexes of the vertices in the position of p_k^{start}, p_k^{goal}, and the *GetPath* function is computes the path based on the b previous vertex indexes assigned to the vertices V. An example for the result of a path finding can be seen in Figure 8a.

Remark (Path simplification)**.** *We can further optimize the found path by exchanging the unnecessarily tortuous edge chains with single edges as shown in Figure 5. In the* path simplification *at first we construct a mini-graph* $\mathcal{G}^m = \{\mathcal{P}, E^s\}$ *where the vertices are the*

elements of $\mathcal{P}$ base path and E^s edges are constructed as:

$$E^s = \{[\mathcal{P}_1, \mathcal{P}_2], \cdots, [\mathcal{P}_i, \mathcal{P}_j], \cdots, [\mathcal{P}_{n-1}, \mathcal{P}_n]\},\ i \in \{1, \cdots, n-1\},\ j \in \{i+1, \cdots, n\} \tag{21}$$

which means $\mathcal{P}_i^r$ vertex have common edges with $\mathcal{P}_{i+1}, \cdots, \mathcal{P}_n$ vertices. We perform the previously introduced, modified A graph search again in $\mathcal{G}^m$ mini-graph to find a more efficient, final path $\mathcal{P}^f$.*

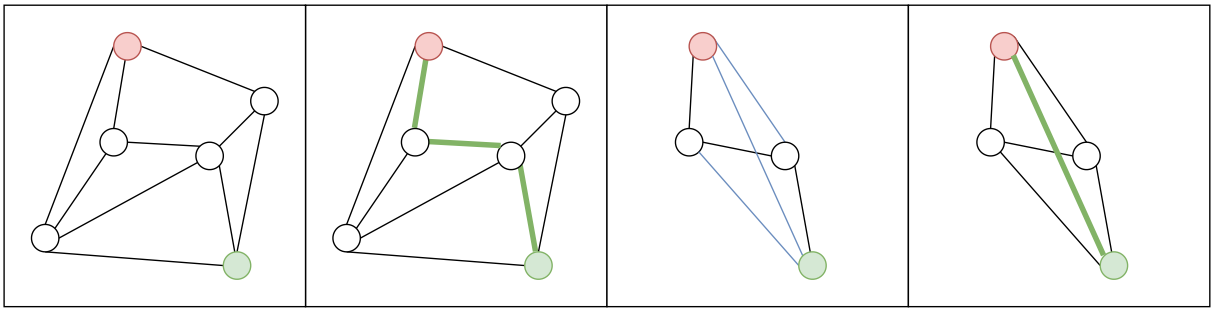
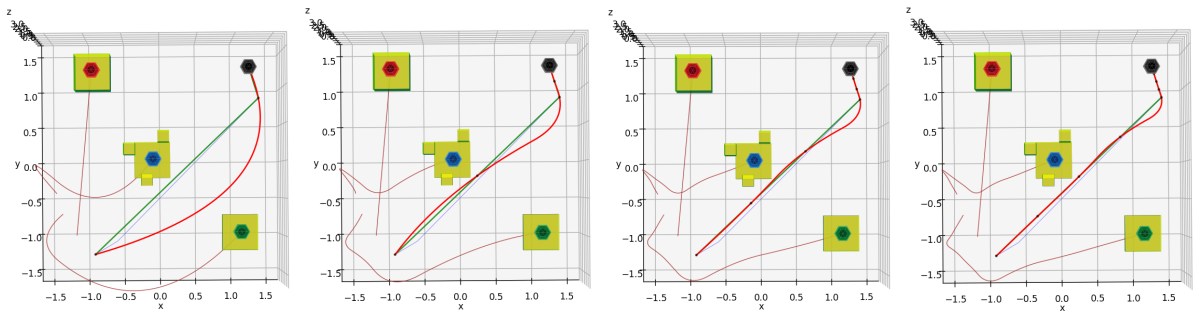


Figure 5: a. Base-graph with highlighted start and end points b. The path on the base-graph c. Generation of the mini-graph d. The simplified path

3.3.2 velocity profile generation

Before start the velocity profile generation we fit a B-spline $\mathcal{S} = [x(s), y(s), z(s)]$ parameterized by its arc length to the elements of $\mathcal{P}^f$. To reduces the deviation of $\mathcal{S}$ spline from the edges of the path we expand $\mathcal{P}^f$ with evenly distributed points along each edge as it shown in Figure 6.



(a) zero extra point (b) one extra point (c) two extra points (d) three extra points

Figure 6: Spline fitting with increasing extra point number

At this point we have a spline which avoids the $\mathcal{O}^s$ static obstacles and a v_c constant velocity. But the drones cannot follow the $\mathcal{S}$ spline only with the v_r reference velocity because: they need to accelerate and decelerate at the start and goal positions, furthermore with a constant velocity a collision free flight is not guaranteed. The goal of the *velocity planning* is to design a collision free velocity profile (see Figure 8b) with feasible accelerations. Since the *path planning* uses v_c while searching for the best path and calculating the possible collisions, if the velocity profile considerably differ from v_c, it is possible that collisions which not taken into account, has to be dealt with. Therefore, we

have to minimize the difference between the $v \in \mathbb{R}$ velocities and v_c as follows:

$$minimize: \sum_{i=0}^{t_\mathrm{H}} (v(i) - v_\mathrm{c})^2 \tag{22}$$

where $t_\mathrm{H} \in \mathbb{R}^+$ is the time horizon under the k-th drone must reach p_k^{goal} position. A sufficient value for t_H can be calculated as:

$$t_\mathrm{H} = \frac{L}{v_\mathrm{c}} \tag{23}$$

where L is the arch length of the 3D trajectory defined by $\mathcal{S}$. Because a acceleration is constrained and the drones start and end their movement in a hovering state t_H has a minimum value, which can be calculated as:

$$t_\mathrm{H} \geq \sqrt{\frac{L}{a_{max}}} \tag{24}$$

A moving obstacle can occupy the path of a drone for a significant amount of time, and this can result that the optimization software cannot find a feasible velocity profile. To handle this problem the *velocity planning* has to be repeatable with increasing t_H value. The optimization is done for a finite number of time instants $t_\mathrm{grid} = \{0, \cdots, t_\mathrm{H}\}$ and for a finite number of arch lengths $s_\mathrm{grid} = \{0, \cdots, L\}$ and the movements are determined by discrete integration. The first variable of the optimization is the covered path length $s(i) \in s_\mathrm{grid}, i \in \{0, \cdots, t_\mathrm{H}, t_\mathrm{H} + T_\mathrm{s}\}$ where T_s is the sampling time. The $s(i)$ is constrained by:

$$s(0) = 0 \tag{25}$$
$$s(t_\mathrm{H} + T_\mathrm{s}) = L \tag{26}$$
$$s(i+1) = s(i) + \dot{s}(i) \cdot T_\mathrm{s} + 0.5 \cdot {T_\mathrm{s}}^2 \cdot \ddot{s}(i),\ i \in t_\mathrm{grid} \tag{27}$$
$$0 \leq s(i) \leq L,\ i \in \{0, \cdots, t_\mathrm{H}, t_\mathrm{H} + T_\mathrm{H}\} \tag{28}$$

The second variable is the velocity $v(i),\ i \in \{0, \cdots, t_\mathrm{H}, t_\mathrm{H} + T_\mathrm{s}\}$, which constrained by:

$$v(0) = 0 \tag{29}$$
$$v(t_\mathrm{H} + T_\mathrm{s}) = 0 \tag{30}$$
$$v(i+1) = \dot{s}(i) + T_\mathrm{s} \cdot \ddot{s}(i),\ i \in t_\mathrm{grid} \tag{31}$$
$$0 \leq v(i) \leq v_\mathrm{max},\ i \in \{0, \cdots, t_\mathrm{H}, t_\mathrm{H} + T_\mathrm{H}\} \tag{32}$$

The third variable is the acceleration $a(i),\ i \in t_\mathrm{grid}$ acceleration which constrained by:

$$-a_\mathrm{max} \leq a(i) \leq a_\mathrm{max},\ i \in t_\mathrm{grid} \tag{33}$$

Finally we need a binary decision variable $b \in \{0, 1\}$ to help to formulate the constrains which used to declare how an obstacle should be avoided. To illustrate the meaning of b, let's take an example where one obstacle cross the path once in a $t_\mathrm{c} = \{t_\mathrm{c,start}, \cdots, t_\mathrm{c,end}\} \subset t_\mathrm{grid}$ time interval as it can be seen in the Figure 7. In this case we can define the decision for the avoidance direction using big-M formulation [2] as follows:

$$s(i) - b \cdot M \leq s_\mathrm{min}(i),\ i \in t_\mathrm{c} \tag{34}$$
$$s(i) + (1 - b) \cdot M \geq s_\mathrm{max}(i),\ i \in t_\mathrm{c} \tag{35}$$

where $s_{\min}(i), s_{\max}(i), \in s_{grid},\ i \in t_c$ give the minimum and maximum values of the occupied part of the path in a given time expressed in arch length and $M \in \mathbb{R}^+$ is a sufficiently big number for the big-M formulation. As it seen in (34) if $b = 0$ then the drone have to wait for the obstacle to cross the path, which means the arch length traveled by the drone $s(i),\ i \in t_c$ has to be less than the minimal value of the occupied part of the path $s_{\min}(i),\ i \in t_c$ in the time interval t_c. Furthermore if $b = 0$ the (35) can only be true because of the sufficiently big value of M. The same thought process can be applied in the case of $b = 1$ when the drone have to rush through the part of the path that will be occupied.

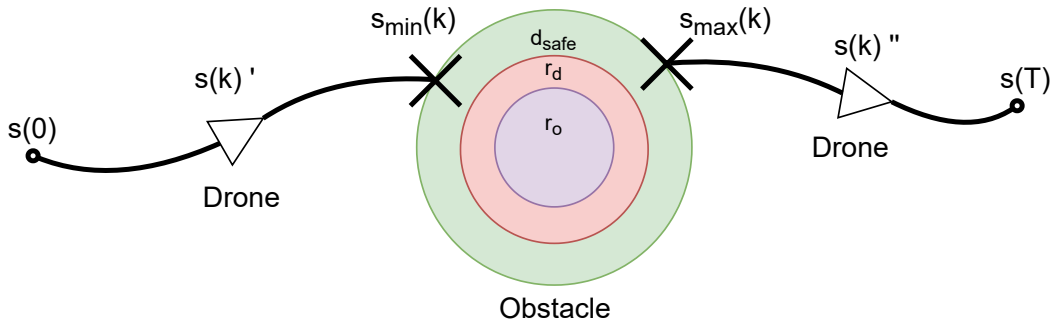


Figure 7: Example for a path crossing during $k \in t_c$ time where the drone has to decide to wait for the obstacle or rush trough before it

Since parts of the path can be occupied multiple times during the flight the (34-35) has to be expanded to handle b, t_c, $s_{\min}$ and $s_{\max}$ with multiple elements. First the cases of route occupations must be calculated before optimization in the t_{grid} time interval, in the process we obtain $t_{c,j},\ s_{\min,j},\ s_{\max,j},\ j \in \mathbb{I}_1^{N_b} = \{b \in \mathbb{Z} \mid 1 \leq b \leq N_b\}$ where N_b is the number of occupations cases. The extended constrains can be defined as follows:

$$s(i) - b_j \cdot M \leq s_{\min,j}(i),\ i \in t_{c,j},\ j \in \mathbb{I}_1^{N_b} \tag{36}$$

$$s(i) + (1 - b_j) \cdot M \geq s_{\max}(i),\ i \in t_{c,j},\ j \in \mathbb{I}_1^{N_b} \tag{37}$$

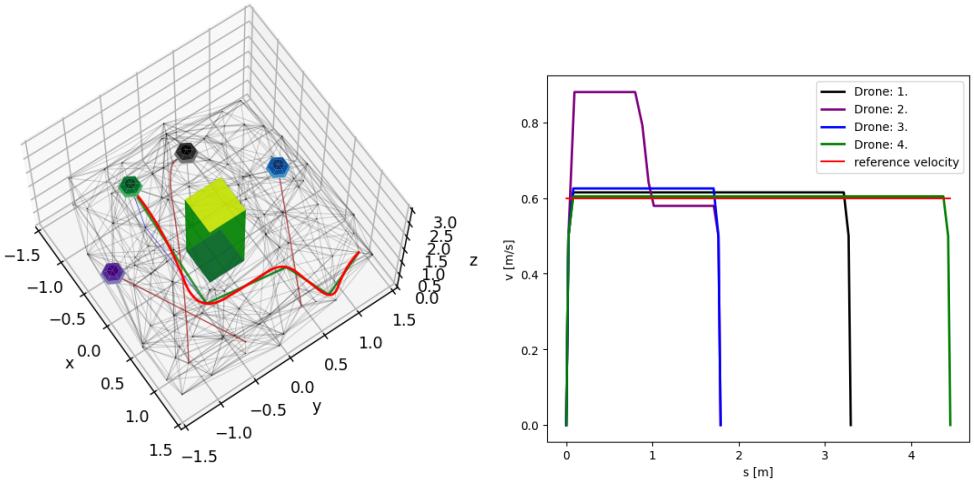
The optimization calculates the covered path length-time function $s(t_{\text{grid}})$ by solving (22) within (25-33), (36-37) constrains, this optimization problem can be formulated as:

$$\begin{aligned} \text{minimize: } & 22 \\ \text{subject to: } & (25-33), (36-37) \end{aligned} \tag{38}$$

The $s(t_{\text{grid}})$ is used with $\mathcal{S}$ spline to construct the $\tau_k(t) = [x(s(t)), y(s(t)), z(s(t))]$ trajectory for the k-th drone. The optimization can be formulated as a mixed-integer quadratic optimization problem which can be solved e.g. with Gurobi software [6].

Algorithm 4 trajectory generation

inputs: arch length parameterized spline $\mathcal{S}$; moving obstacles with known trajectories $\mathcal{O}^k$
output: trajectory for the k-th drone τ_k
solve (23) to obtain t_H
if (24) = False **then**
    increase t_H until (24) = True
end if
repeat
    $t_\mathrm{grid} \leftarrow \{0, \cdots, t_\mathrm{H}\}$
    calculate $N_\mathrm{b},\ t_{\mathrm{c},j},\ s_{\min,j},\ s_{\max,j},\ j \in \mathbb{I}_1^{N_\mathrm{b}}$ from $\mathcal{O}^k,\ \mathcal{S},\ t_\mathrm{grid}$
    set variables $s(i),\ i \in \{0, \cdots T_\mathrm{H}, T_\mathrm{H}+T_\mathrm{s}\};\ \dot{s}(i),\ i \in \{0, \cdots T_\mathrm{H}, T_\mathrm{H}+T_\mathrm{s}\};\ \ddot{s}(i),\ i \in \{0, \cdots T_\mathrm{H}\};\ b_j,\ j \in \mathbb{I}_1^{N_\mathrm{b}}$
    set constrains as (25-33), (36-37)
    set objective as (22)
    optimize (38) to obtain s
    increase t_H
until solution is feasible
construct τ_k from $s,\ \mathcal{S}$



(a) Result of the path finding

(b) Generated velocity profiles for the drones

Figure 8: Trajectory generation

3.4 *Avoiding unknown moving objects*

At this point the drones can safely navigate in an $\mathcal{E}$ environment filled with obstacles that either static or moving in an a-priori known trajectory. Unfortunately, in the $\mathcal{E}$ environment there could be N_u number of dynamic obstacles with unknown motion $\mathcal{O}^u = \{\mathcal{O}_1^u, \cdots, \mathcal{O}_{N_\mathrm{u}}^u\}$ (e.g. human workers or other robots) which also have to be avoided. We assume that these unknown obstacles can be enclosed in a capsule (see Figure 9) so each obstacle can be represented by three parameters $\mathcal{O}_k^u = \{p_{u,k}(t), r_{u,k}, h_{u,k}\},\ k \in \mathbb{I}_1^{N_\mathrm{u}} = \{b \in \mathbb{Z} \mid 1 \leq b \leq N_\mathrm{u}\}$.

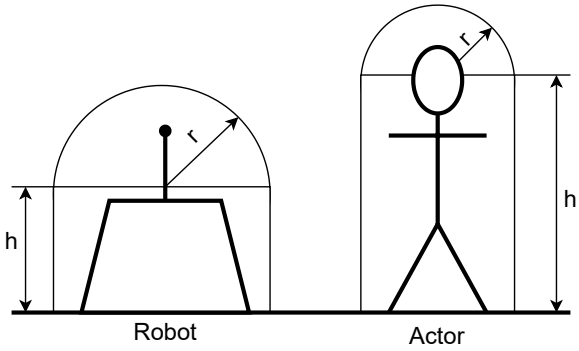


Figure 9: Robot and Human (Actor) enclosed in a convex capsule

Since we do not know the trajectories of the objects, we can only predict their movement based on their past positions. We measure the current positions $p_{\mathrm{p},k} \in \mathbb{R}^3,\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ of $\mathcal{O}_k^u,\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ obstacles and based on their positions in the previous measurement $p_{\mathrm{p},k} \in \mathbb{R}^3,\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ we calculate their future positions assuming a constant velocity motion. The $\bar{v}_k \in \mathbb{R}^3,\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ velocity vector of the k-th $\mathcal{O}^u$ obstacle is given by:

$$\bar{v}_k = \frac{p_{\mathrm{c},k} - p_{\mathrm{p},k}}{t_\mathrm{c} - t_\mathrm{p}},\ k \in \mathbb{I}_1^{N_\mathrm{u}} \tag{39}$$

where $t_\mathrm{p} \in \mathbb{R}^+$ is the measurement time of the previous position and $t_\mathrm{c} \in \mathbb{R}^+$ is the measurement time of the $p_{\mathrm{c},k}$ current position of the k-th $\mathcal{O}^u$ obstacle. We predict the future positions $p_{\mathrm{f},k}(t) \in \mathbb{R}^3, t_\mathrm{c} \leq t \leq (t_\mathrm{c} + t_\mathrm{H}),\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ in a suitably short time horizon, where the assumption of constant velocity can be valid. Assuming constant velocity in over a larger horizon would possibly cause invalid collision warning. The prediction for the $p_\mathrm{f}(t)$ future positions of the obstacle in a finite number of time instants $t_\mathrm{grid} = \{t_\mathrm{c}, \cdots, t_\mathrm{c} + t_\mathrm{H}\}$ is done by:

$$p_{\mathrm{f},k}(t) = \bar{v}_k \cdot (t - t_\mathrm{c}) + p_{\mathrm{c},k},\ t \in t_\mathrm{grid},\ k \in \mathbb{I}_1^{N_\mathrm{u}} \tag{40}$$

With the increasing value of t the accuracy of the prediction deteriorates, therefore we have to compensate for the uncertainty of the future position. When the future collisions are calculated the dimensions of the obstacles are "increased" based on how far to the future we predict their positions (see in Figure: 10). Therefore the $r_\mathrm{o}(t) \in \mathbb{R}^+,\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ radii and $h_{\mathrm{o},k}(t) \in \mathbb{R}^+,\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ heights of the $\mathcal{O}_k^u,\ k \in \mathbb{I}_1^{N_\mathrm{u}}$ obstacles are time dependents as:

$$r_{\mathrm{o},k}(t) = r_{\mathrm{o},k}(t_\mathrm{c}) + \gamma \cdot (t - t_\mathrm{c}),\ t \in t_\mathrm{grid},\ k \in \mathbb{I}_1^{N_\mathrm{u}} \tag{41}$$

$$h_{\mathrm{o},k}(t) = h_{\mathrm{o},k}(t_\mathrm{c}) + \gamma \cdot (t - t_\mathrm{c}),\ t \in t_\mathrm{grid},\ k \in \mathbb{I}_1^{N_\mathrm{u}} \tag{42}$$

where the γ is a tuning hyper parameter chosen a-prior to define the "growth" of the obstacle.

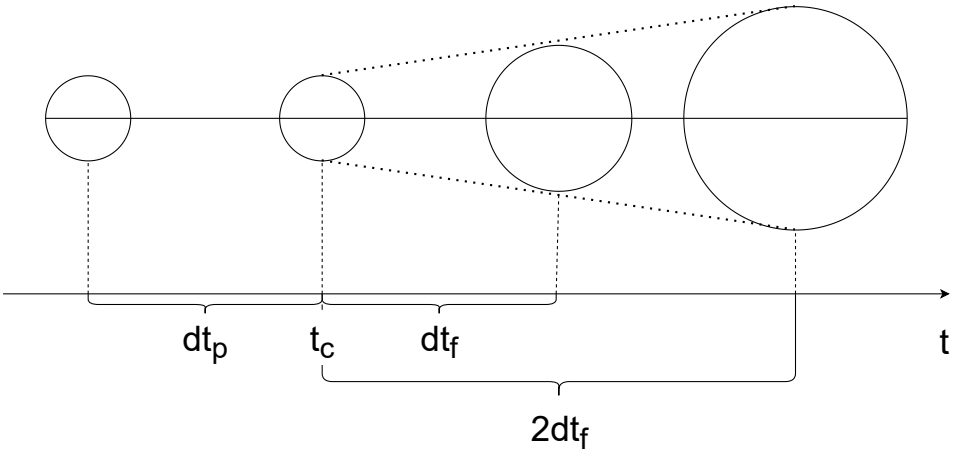


Figure 10: Prediction for the future positions of an obstacle

Now we have predicted positions for the $\mathcal{O}^u(t)$ from the t_c time moment, we can calculate the minimal distances between the drone and the obstacles in the $t_\mathrm{c} \leq t \leq (t_\mathrm{c} + t_\mathrm{H})$ time interval:

$$d_\mathrm{m}(t) = \min_{0 \leq \alpha \leq 1} ||(p_{\mathrm{f},k}(t) + \alpha \cdot \bar{h}_{\mathrm{o},k}(t)) - (p_\mathrm{d}(t))||,\ t \in t_\mathrm{grid},\ k \in \mathbb{I}_1^{N_\mathrm{u}} \tag{43}$$

Based on the result of $d_m(t)$ there is three cases that have to be distinguished: (i) the obstacles are a safe distance away so there are no need to modify the trajectory of the drone; (ii) there will be a collision in a $t_\mathrm{c} \leq t \leq (t_\mathrm{c} + t_\mathrm{m})$ time window where $t_\mathrm{m} < t_\mathrm{H}$ is a critically low time window which under an evasive maneuver is not possible due to the physical limitations of the quadcopter and the computational time of the algorithm; (iii) there will be a collision in a $(t_\mathrm{c} + t_\mathrm{m}) << t \leq (t_\mathrm{c} + t_\mathrm{H})$ time window therefore an evasive maneuver is possible.

In the first case there is nothing to do so the drone can continue its flight. The second case occurs, because even with the most efficient implementation there will be an idle time window for the calculations, the communication between the drone and the PC, and the physical reaction. So there is a critical time minimum during which an evasion is not possible, because the drone cannot react in time. Therefore, we have to define an emergency command for this event, which can be executed fast and safely (e.g. a stop moving or turn off command).

In the following we will explain our solution for the the third case by constructing an evasive trajectory for the drone by locally modifying its original path. From $d_m(t)$ minimal time distance we can see when the path will be occupied as:

$$d_\mathrm{m}(t) < (r_\mathrm{d} + r_\mathrm{o}(t) + d_\mathrm{safe}),\ t_\mathrm{o,min} \leq t \leq t_\mathrm{o,max} \tag{44}$$

where $t_\mathrm{o,min}$ and $t_\mathrm{o,max}$ are the minimal and maximal time values when the $d_m(t)$ minimal distance is less than the sum value of the r_d radius of the drone, r_o the radius of the obstacle and the d_safe safety distance. With these time values the occupied section of the path is between $s(t_\mathrm{o,min})$ and $s(t_\mathrm{o,max})$. There are two more important point on the path which we have to declare: the point which from the new path start to deviate from and the point where the new path reconnects with the original path. We want to start the evasive maneuver as soon as possible therefore $s(t_\mathrm{c} + t_\mathrm{a})$ is depends only from the t_a average value of the computation plus communication time. For a safe return to the original path, we

need a point in a sufficient distance from the obstacle. This d_{return} returning distance is calculated from the radius of the obstacle as:

$$d_{\text{return}} = \mu \cdot r_{\text{o}} \tag{45}$$

where $\mu = \mathbb{R}^+$ is a hyper parameter which we use to tune the d_{return} returning distance. These four points of the original path can be seen in Figure 11.

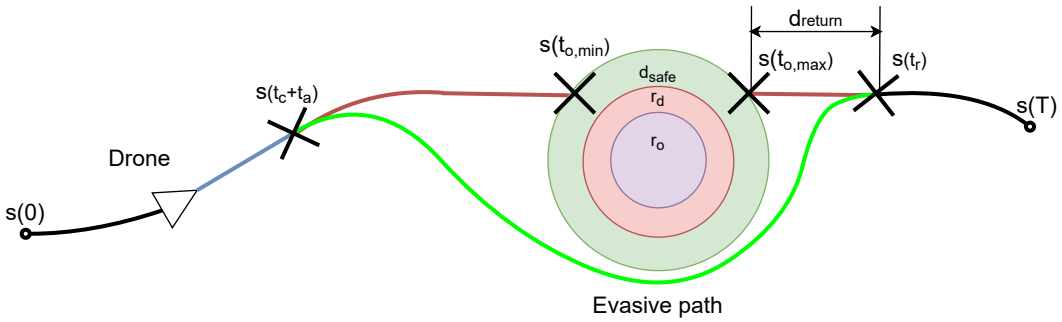


Figure 11: Important points of the original path in the process of evasive path generation

Now we know all the important point on the original path we can start to construct evading path candidates with different evading directions and deviations from the original path. For this we define two planes which are perpendicular to the line that connects the $s(t_{\text{o,min}})$ and $s(t_{\text{o,max}})$ points and also include one-one said point. The $P_1 = \{p_1, \bar{n}\} \subset \mathbb{R}^3$ and $P_2 = \{p_2, \bar{n}\} \subset \mathbb{R}^3$ planes can be defined with one of their points and their normal vector as:

$$p_1 = \{x(s(t_{0,\text{min}})), y(s(t_{0,\text{min}})), z(s(t_{0,\text{min}}))\} \tag{46}$$

$$p_2 = \{x(s(t_{0,\text{max}})), y(s(t_{0,\text{max}})), z(s(t_{0,\text{max}}))\} \tag{47}$$

$$\bar{n} = \frac{p_2 - p_1}{||p_2 - p_1||} \tag{48}$$

In these planes we generate N_{c} number of circle pairs $C_{1,\text{i}} = \{p_1, r_{\text{c},i}, \bar{n}\} \subset \mathcal{R}^3,\ i \in \{1, \cdots, N_{\text{c}}\}$ and $C_{2,\text{i}} = \{p_2, r_{\text{c},i}, \bar{n}\} \subset \mathbb{R}^3,\ i \in \{1, \cdots, N_{\text{c}}\}$ which $r_{\text{c},i} \in \mathbb{R}^+,\ i \in \{1, \cdots, N_{\text{c}}\}$ radii are different for each pair and their centers are p_1 and p_2 (see Figure 12). Now we choose randomly or in a given pattern N_p point pairs $p_{\text{c},1,\text{i}} \subset C_1,\ i \in \mathbb{I}_1^{N_{\text{p}}} = \{b \in \mathbb{Z} \mid 1 \leq b \leq N_{\text{p}}\}$ and $p_{\text{c},2,\text{i}} \subset C_2,\ i \in \mathbb{I}_1^{N_{\text{p}}}$ on the circles, one pair is defined as follows:

$$p_{\text{c},2,i} = p_{\text{c},1,i} + (p2 - p1),\ i \in \mathbb{I}_1^{N_{\text{p}}} \tag{49}$$

Finally we define the N_p number of path candidates which start at $p_{\text{s}} = \{x(s(t_{\text{c}} + t_{\text{a}})), y(s(t_{\text{c}} + t_{\text{a}})), z(s(t_{\text{c}} + t_{\text{a}}))\}$, go through a point pair $p_{\text{c},1,i},\ p_{\text{c},2,i}$ and return in $p_{\text{r}} = \{x(s(t_{\text{r}})), y(s(t_{\text{r}})), z(s(t_{\text{r}}))\}$.

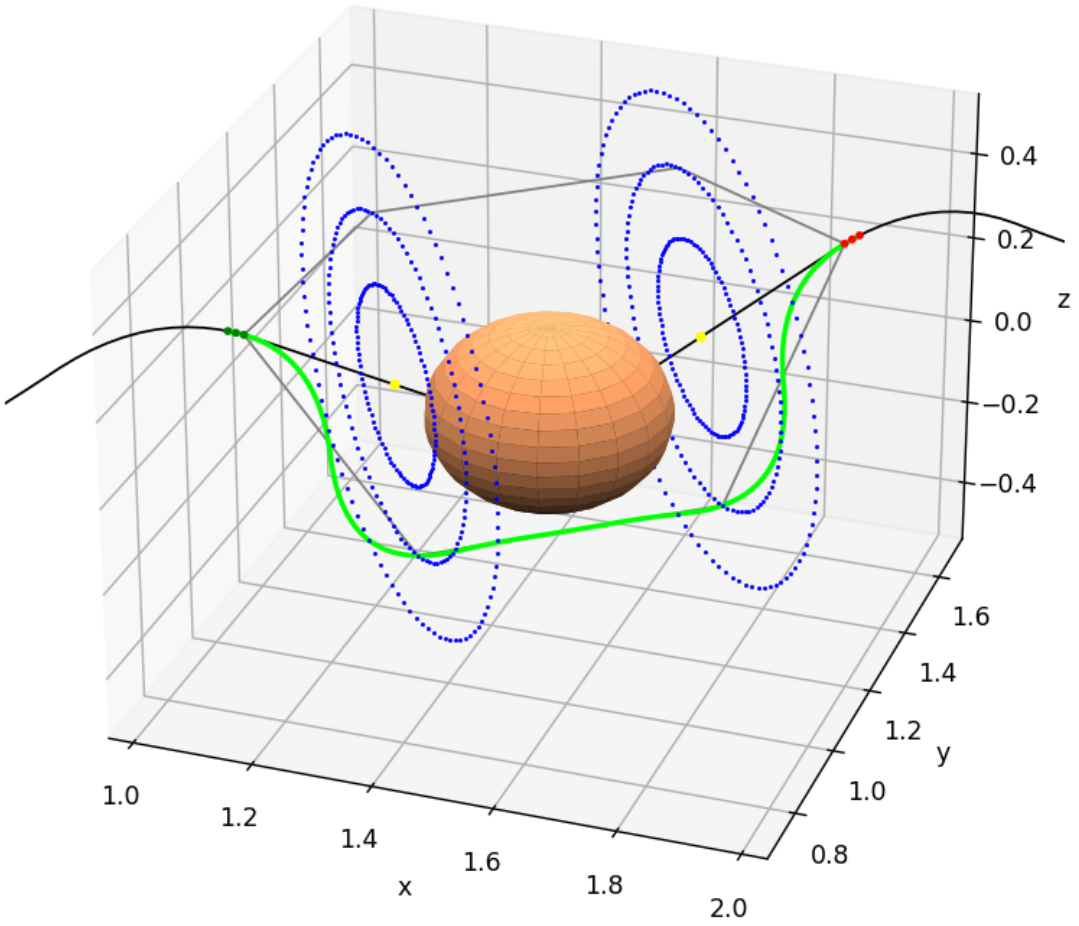


Figure 12: Construction of the evasive paths in one circle

To choose the best candidate we build a graph from the p_{s}, $p_{\mathrm{c},1,i}$, $p_{\mathrm{c},1,i}$, $i \in \mathbb{I}_1^{N_{\mathrm{p}}}$ and p_{r} points and run a similar graph search and velocity planning as in Section 3.3. The $\mathcal{G}^e = V^e, E^e$ graph (see Figure 13) is constructed for the search, it's vertices are containing the points defined above as:

$$V^e = \{p_{\mathrm{s}}, p_{\mathrm{c},1,i}, p_{\mathrm{c},2,i}, \cdots, p_{\mathrm{r}}\},\ i \in \mathbb{I}_1^{N_{\mathrm{p}}} \tag{50}$$

and the E^e edges are connecting p_{s} to $p_{\mathrm{c},1,i}$, $i \in \mathbb{I}_1^{N_{\mathrm{p}}}$, the $p_{\mathrm{c},2,i}$, $i \in \mathbb{I}_1^{N_{\mathrm{p}}}$ to p_{r} and the $p_{\mathrm{c},1,i}$, $p_{\mathrm{c},2,i}$, $i \in \{1, \cdots, N_{\mathrm{p}}\}$ pairs to each other in a way that every point pair with same index i is connected as:

$$E^e = \{p_{\mathrm{s}}, p_{\mathrm{c},1,i}\} \cup \{p_{\mathrm{c},1,i}, p_{\mathrm{c},2,i}\} \cup \{p_{\mathrm{c},2,i}, p_{\mathrm{r}}\},\ i \in \mathbb{I}_1^{N_{\mathrm{p}}} \tag{51}$$

Now we must check the edges of each path candidate whether it intersects with a static obstacle. Note that when we delete the edges intersecting with the $\mathcal{O}^s$ static obstacles, we need to remove all the edges that belongs they path candidate. So if any edge from the $[p_{\mathrm{s}}, p_{\mathrm{c},1,i}], [p_{\mathrm{c},1,i}, p_{\mathrm{c},2,i}], [p_{\mathrm{c},2,i}, p_{\mathrm{s}}]$, $i \in \mathbb{I}_1^{N_{\mathrm{p}}}$ edge triplet is intersecting with a static obstacle, the whole triplet is removed.

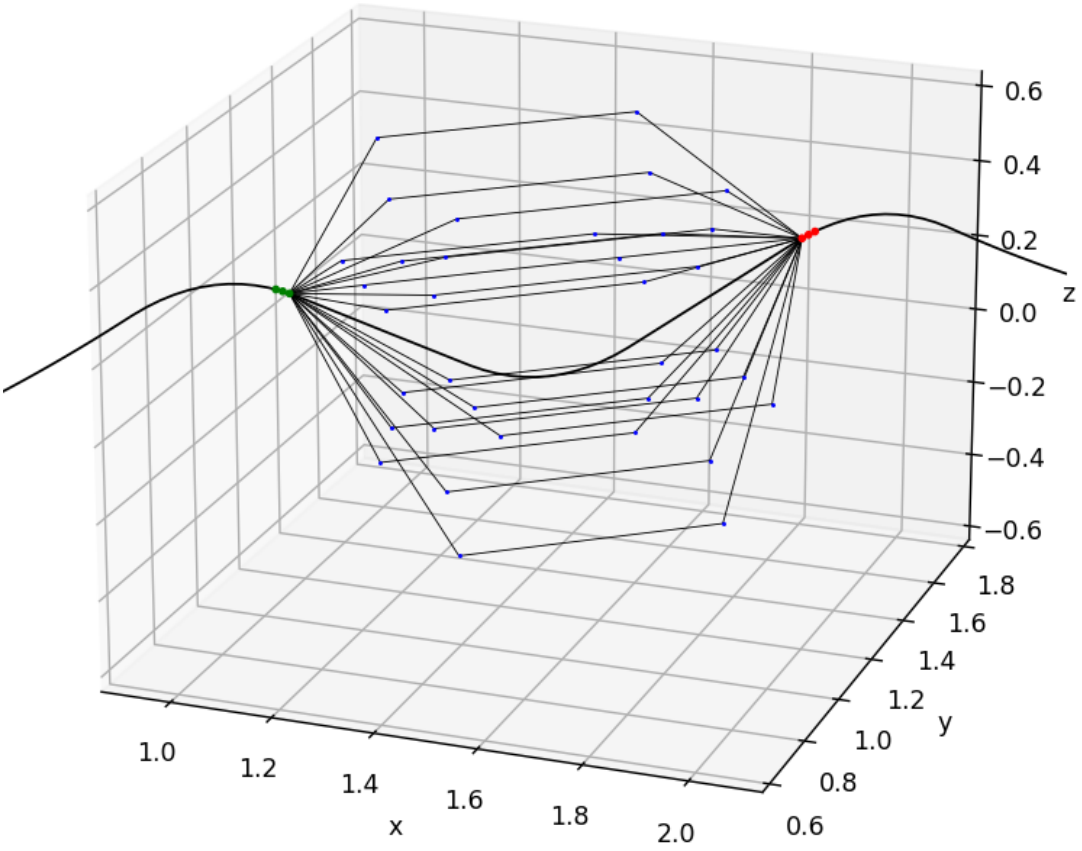


Figure 13: Graph made from the path candidates

The graph search is the same as it was in Section 3.3 with the extension of the predicted movement for the $\mathcal{O}^u$ obstacles, which for we assume a constant velocity motion $\bar{v}_k$. We fit a B-spline to the selected new path and connect it to the remaining part of the original path in the p_r returning point. The evasive trajectory is $\tau_\mathrm{e}(t) = [x_\mathrm{e}(s_\mathrm{e}(t)), y_\mathrm{e}(s_\mathrm{e}(t)), z_\mathrm{e}(s_\mathrm{e}(t))]$ where $s_\mathrm{e}(t) = \mathbb{R}^+$ is the arch length of the evasive path in a given time. Note that the evasive path will start at p_s so:

$$\tau_e(0) = p_s \tag{52}$$

The velocity planning is performed similarly as before with the following modifications: the alteration of the starting velocity constrain to mach the actual velocity of the drone in time $t_\mathrm{c} + t_\mathrm{a}$, since the execution of the evading trajectory will start during the flight:

$$\dot{s}_e(0) = \dot{s}(t_\mathrm{c} + t_\mathrm{a}) \tag{53}$$

The original trajectory is replaced by the modified one. After the drone starts to follow the evasive trajectory τ_e it will be handled like the original trajectory of that drone in case of a new evasive trajectory generation is needed. This is true even if more than one collision would be predicted in the original trajectory, because multiple obstacles are handled separately. The order of checking the $\mathcal{O}_i^k, k \in \mathbb{I}_1^{N_\mathrm{u}}$ obstacles is defined a-priori, and the evasive candidates are constructed to avoid the first detected obstacle. However, when we choose from the candidates, we take into account all of them to reduce the possibility for a future collision with them, i.e. the need for a new evasive maneuver.

Algorithm 5 path checker

inputs: $\mathcal{O}^s$ static obstacles; $\mathcal{O}^k$ moving obstacles; τ drone trajectories; γ hyper parameter for size increase; t_a average computation time; μ hyper parameter of returning
output: τ_e evasive trajectory
$p_\mathrm{p} \leftarrow \mathrm{pos}(\mathcal{O}^u(t_0))$
loop
    $t_\mathrm{c} \leftarrow$ current time
    $p_\mathrm{c} \leftarrow \mathrm{pos}(\mathcal{O}^u(t_\mathrm{c}))$
    construct prediction $p_\mathrm{f}(t)$ from p_p and p_c via (39 - 40)
    solve (41-42) with $dim(O^u)$ to obtain r_o, h_o
    for $i \in \mathbb{I}_1^{N_\mathrm{d}}$ **do**
        for $j \in \{1, \cdots, \#\mathcal{O}^u\}$ **do**
            solve (43) with $p_{\mathrm{f},j}(t),\ r_{\mathrm{o},j},\ h_{\mathrm{o},j}$ to obtain $d_\mathrm{m}(t)$
            evaluate (44) with $d_\mathrm{m}(t)$ to check collision
            if collision = True **then**
                $s(t_{\mathrm{o,min}}),\ s(t_{\mathrm{o,max}}) \leftarrow OccupiedPart(\tau_i,\ p_{\mathrm{f},j},\ r_{\mathrm{o},j},\ h_{\mathrm{o},j})$
                solve (46-49) with $s(t_{\mathrm{o.min}}),\ s(t_{\mathrm{o.min}},)$ to obtain $p_{\mathrm{c},1},\ p_{\mathrm{c},2}$
                $p_\mathrm{s} \leftarrow \tau_i(t_\mathrm{c} + t_\mathrm{a})$
                solve (45) to obtain d_return
                $p_\mathrm{r} \leftarrow GetReturnPoint(\tau_i, d_\mathrm{return})$
                construct V^e based on (50) with $p_\mathrm{s},\ p_{\mathrm{c},1},\ p_{\mathrm{c},2},\ p_\mathrm{r}$
                construct E^e based on (51) with $p_\mathrm{s},\ p_{\mathrm{c},1},\ p_{\mathrm{c},2},\ p_\mathrm{r}$
                $\mathcal{G}^e = \{V^e, E^e\}$
                run Alg: 3 with $\mathcal{G}^e$ to obtain $\mathcal{P}^e$
                $\mathcal{S}^e \leftarrow FitSpline(\mathcal{P}^e, \tau_i)$
                run Alg: 4 with $\mathcal{S}^e,\ p_\mathrm{s}, \tau_i(t_\mathrm{c} + t_\mathrm{a})$ according to (52-53) to obtain τ^e
                update τ_i with τ^e
            end if
        end for
    end for
    update p_p with p_c
end loop

In the algorithm the dim() and pos() gives the dimensions and positions of the obstacles. The *OccupiedPart* function calculates the distances between the future positions of an obstacle and the path of the trajectory, then gives the minimal and maximal arch lengths of the occupied path segment. The *GetReturnPoint* function selects the returning point to the original path which is d_return distance away from $s(t_{\mathrm{o,max}})$. The *FitSpline* function fits a B-spline to the selected evasive path and connects it with $\tau_i(t),\ t \in \{t_\mathrm{r}, \cdots, T_i\}$.

4 Simulation based analysis

The tests for the path planning and for the unknown obstacle avoidance are done separately with one-one question for each. As for the path planning, we want to know how the computation time increases with the number of static and moving obstacles. As for the evasive path generation, we want to test it for various scenarios to discover its efficiency and potential weaknesses.

They were carried out using an ASUS ROG Strix G513IH laptop with a 2.90 GHz AMD Ryzen 7 4800H CPU. The algorithm is implemented in Python and analyzed first using matplotlib [8] simulation and further examined in the MuJoCo simulation environment [14].

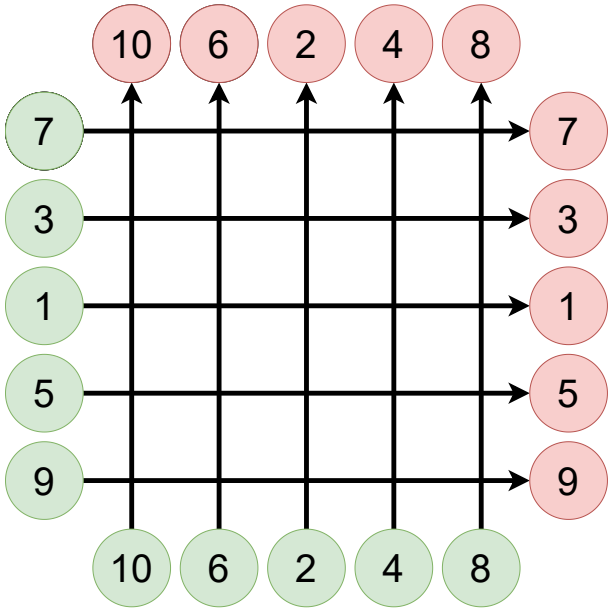
The two analyses use the same type of drones with the same capabilities and same dimensions. We assume the drones can achieve a maximum velocity of $v_{\max} = 1m/s$, a maximum acceleration of $a_{\max} = 0.4m/s$ and can be enclosed in a capsule with radius of $r = 0.1m$ and height of $h = 0.6m$.

4.1 Analysis of the trajectory planning algorithm

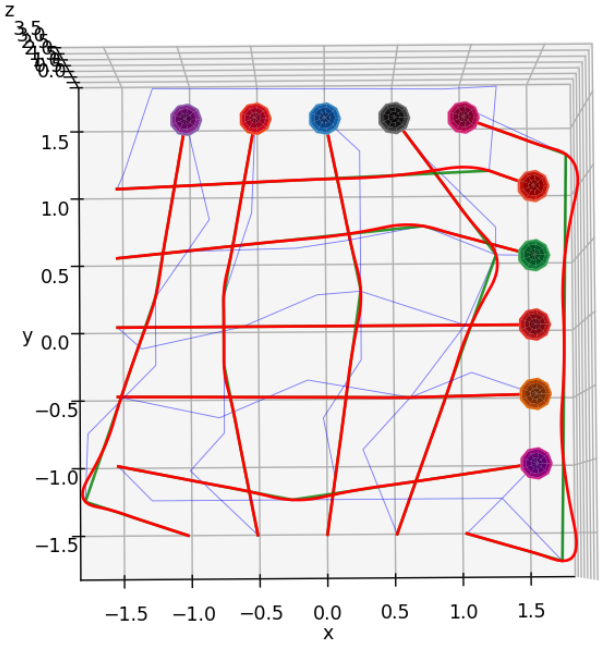
To test the computational time of more and more complicated tasks, we tested the Algorithms 3-4 in three different environment with increasing amount (0,4,13) of static obstacle $\mathcal{O}^s$ and in each cases we planned trajectories for ten drones. The placement of the static obstacles and the found paths can be seen in Figure 14.

The floor area of the test environment is $3.5m$ x $3.5m$ with a minimum and maximum flying height of $0.3m$ and $1.3m$. The dimensions of the static obstacles are $0.2m$ x $0.2m$ x $1.4m$, note that the obstacles are taller than the maximal flying height to force the drones to fly between them. As for the hyper parameters, the simulations were done with: $N_{\text{v}} = 100$, $d_{\text{T}} = 0.1$, $c_{\min} = 2$, $c_{\max} = 20$, $v_{\text{r}} = 0.4m/s;\ 0.5m/s;\ 0.6m/s$, $d_{\text{safe}} = 0.05m$.

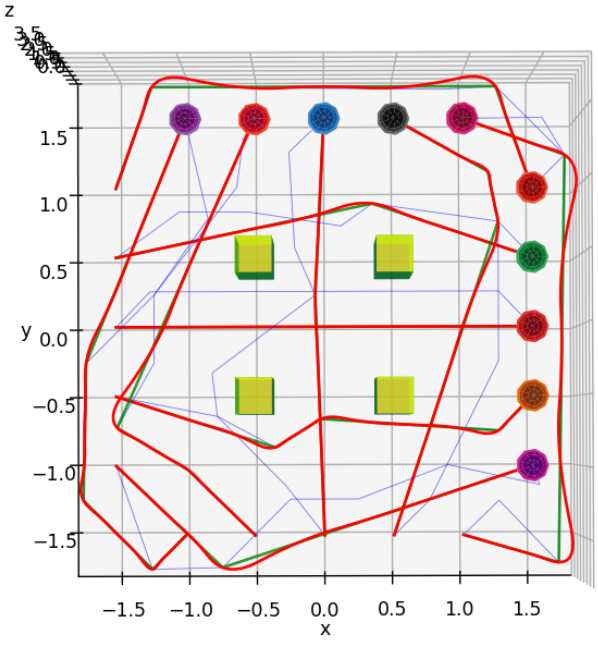
Since the goal of the analysis is the determination of the calculation time, we made an easily scalable pattern for the p^{start} and p^{goal} positions which can be seen in Figure 14a. We predict that the computational time for constructing a trajectory will increase with the number of the obstacles present in the environment. We used matplotlib [8] to show the trajectories of the drones (red lines) and the base paths before the simplification (thin blue lines), the videos showing the results of the MuJoCo simulations can be accessed with the links below Figures 15a-15c.



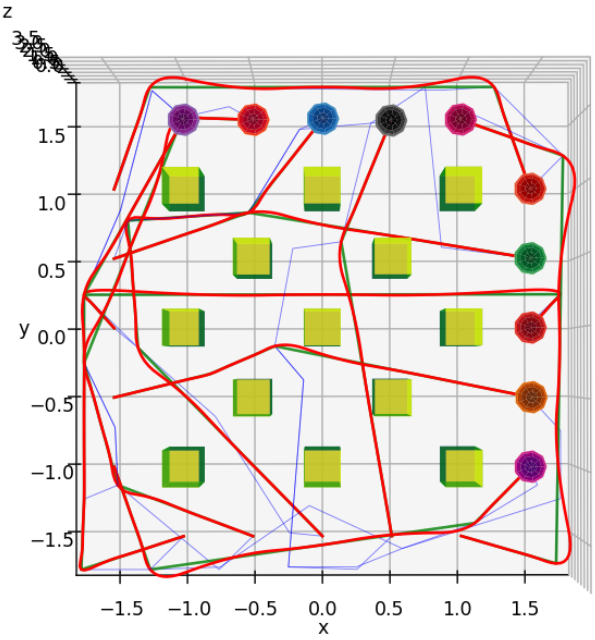
(a) Start and goal positions with drone order



(b) Found paths in an empty environment MujoCo simulation at https://youtu.be/VSlGnK8FVrc

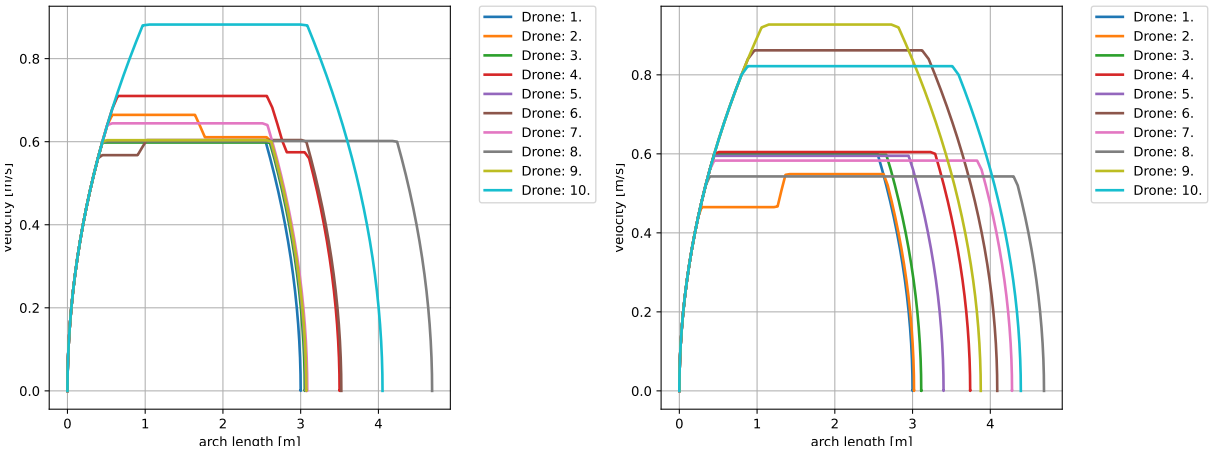


(c) Found path between four static obstacle MujoCo simulation at https://youtu.be/rpfzpxGNfa8

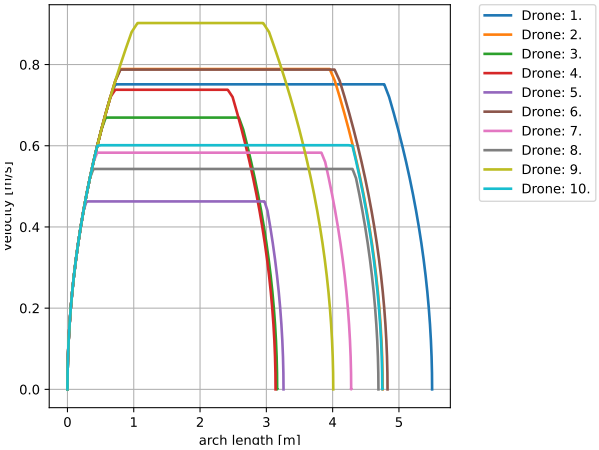


(d) Found path between thirteen static obstacle MujoCo simulation at https://youtu.be/ujIkgJTjnDM

Figure 14: Trajectories between different number of static obstacles generated with Algorithm 3



(a) Velocity profile among zero static obstacles (b) Velocity profile among four static obstacles



(c) Velocity profile among thirteen static obstacles

Figure 15: Velocity profiles generated with Algorithm 4

The chosen velocity was the available maximum $0.6m/s$ in all cases expect one case were among thirteen static obstacle, with two dynamic obstacle where the chosen velocity was $v_c = 0.5m/s$. From this we can assume that in sparse environments, the variety of velocity choices are less important than in dense environments. This need for a variety of velocities is clearer in the optimized velocity profiles seen in Figure 15, which shows the velocity of the drones compared to the distance traveled in their path.

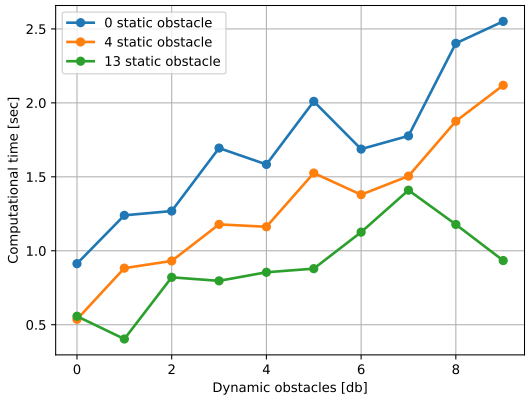
The Table 1 shows how much time the drones needs to fly from their start positions to their goal position on the paths found with the Algorithm 3 with constant velocity and the impact of the path simplification (discussed in Section: 3.3.1) that results in these flight times. The minimum flight time between the goal and end positions with the highest available constant velocity $v_c = 6m/s$ is 5sec, it can be seen in the first case when there are no obstacles blocking a straight passage with the first drone which do not have to deal with any obstacle. This time value is present in several places in the table, referring to a straight, undisturbed way between the obstacles It shows that with the increasing value of static obstacles the paths are got more complicated, which was to be expected. This increasing partner is not present in respect of the moving obstacles because the different drones did not plan their paths among the same moving obstacles relative to their local coordinate system. The second half of the table shows the importance of the

path simplification. It shows the percentage by which the duration of the path has been shortened. The average is a 16.815% reduction, but it can reach even a 72.904%.

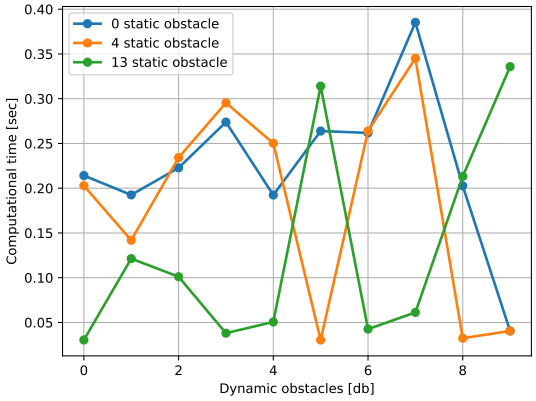
$\#\mathcal{O}^k$	Time of flight [sec]			Saved time [%]		
[pc]	$\#\mathcal{O}^s = 0$	$\#\mathcal{O}^s = 4$	$\#\mathcal{O}^s = 13$	$\#\mathcal{O}^s = 0$	$\#\mathcal{O}^s = 4$	$\#\mathcal{O}^s = 13$
0	5	5	8.748	16.912	10.413	8.867
1	5.1156558	5.029	7.865	6.526	50.282	12.496
2	5.0805	5.176	5.255	10.846	13.794	66.345
3	5.792	6.168	6.259	33.847	3.893	72.904
4	5.102	5.600	8.088	13.677	12.550	52.769
5	5.849	6.692	7.956	10.434	9.589	21.146
6	5.111	6.985	6.985	43.992	9.055	9.055
7	7.545	7.545	7.545	8.719	14.506	0.440
8	5.115	6.385	6.560	7.917	22.641	25.442
9	6.681	7.256	7.859	15.281	13.612	26.347

Table 1: Path lengths and time saving with path simplification

The results of the simulations just half met with our expectations. At first there is an expected overall increase in the computational time of Algorithm 3 as it can be seen in Figure 16a with the increasing dynamic obstacle numbers, but with the static obstacle number the results are the opposite. A possible explanation is with more obstacles there are less edges, hence less possible choices. Since we use relatively thin columns as obstacles (not e.g. wide walls) the Algorithm 3 do not have search for complex paths which would result in longer path finding. The Algorithm 3 should be further tested in maze like environments with increasing difficulty. The Figure 16b shows that the computational time of the Algorithm 4, which wildly varies but in average is much more faster than the path search.



(a) Computation times of the path finder Algorithm 3



(b) Computation time of the velocity profile planner Algorithm 4

Figure 16: Computational time of trajectories in different environments, where each point represents the computational time needed for one drone

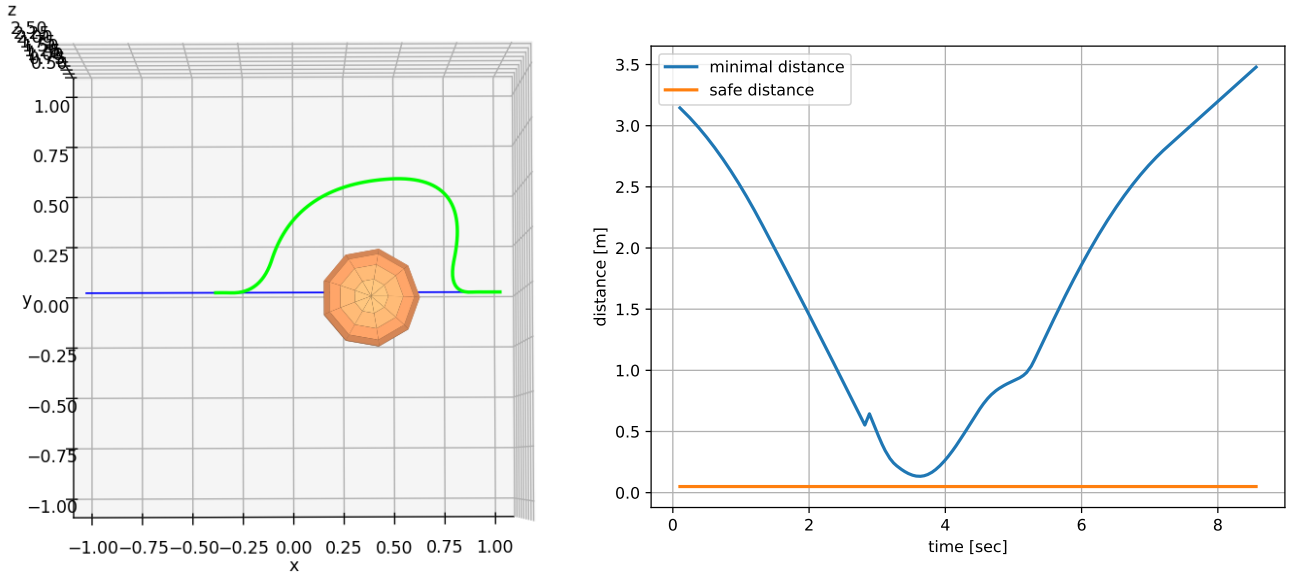
4.2 Analysis of the collision avoidance algorithm

In this section, we examine the limitations of the Algorithm 5 through different tasks. Since we could compose infinite number of different situations our focus was to examine how the algorithm handles simple problems. The hyper parameters of the *path checker* were the followings:$t_{\mathrm{H}} = 0.75sec$, $t_{\mathrm{a}} = 0.1sec$, $\gamma = 1.4$, $\mu = 0.5$, $N_{\mathrm{c}} = 4$, $r_{\mathrm{c}} = \{0.05m,\ 0.2m,\ 0.4m,\ 0.7m\}$, $N_{\mathrm{p}} = 32$ and the points for the candidate construction were equally distributed between and on the circles. During the tests the computational times for the path generation varied between $0.04 - 0.1sec$ and the average computational time was $0.0714sec$. The time the algorithm needs for a collision check is less than $0.001sec$.

4.2.1 Avoiding frontal and rear collision

In the first two cases, we examined the response of the algorithm when a column-shaped obstacle with a radius of $r_{\mathrm{o}} = 0.25m$ would have collided with the drone from the front or the back. During the tests the drone and the obstacle moved in a common straight line.

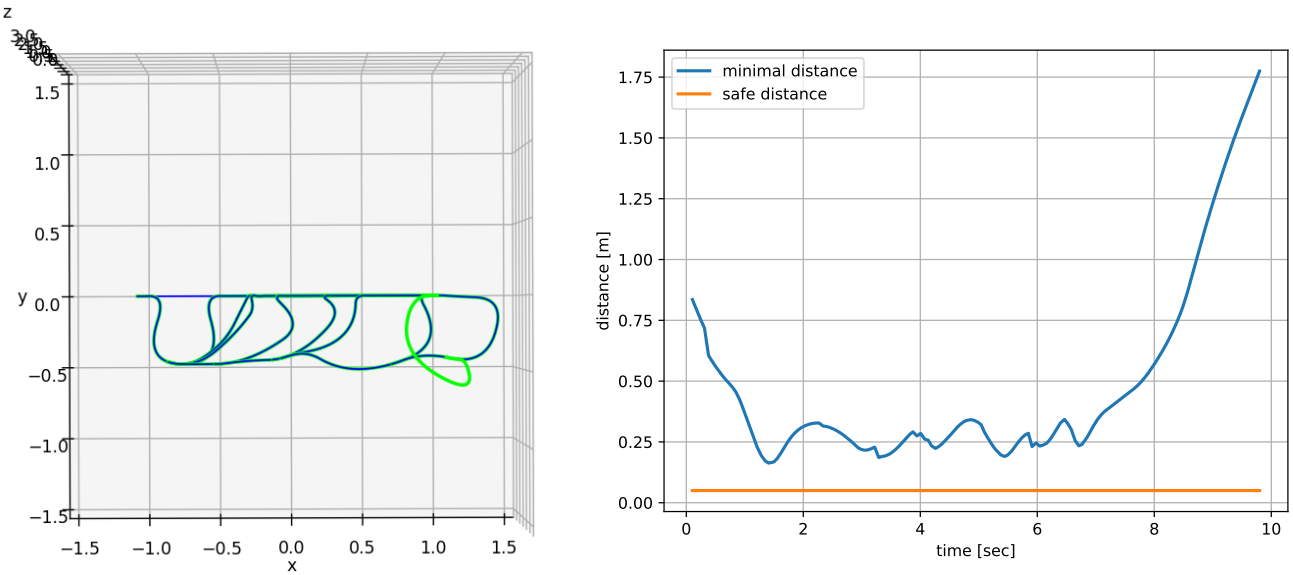
In the frontal tests the only limiting factor was the distance which from we started the obstacle. If the obstacle was started a sufficient distance the algorithm had designed a trajectory for a well-timed side motion which can be seen in the Figure 17a. The instant when the new trajectory was started can be seen around $3sec$ in Figure 17b since there is a little jump in the distance between the drone and the obstacle. For further developing the algorithm, we want to minimize this jump since it is a sign of a wrongly timed switch between the trajectories.



(a) The evasive path (b) Distance between the drone and the obstacle

Figure 17: Frontal collision avoidance

In cases when the drone and the obstacle moved in the same direction and the velocity of the obstacles was near to the velocity of the drone (which in this tests were $\pm 0.05 m/s$), a phenomenon occurred when the drone moved side by side to the obstacle. One such precedent can be seen in Figure 18a, which show that the drone kept trying to return to its original path.



(a) Constantly updated paths (b) Distance between the drone and the obstacle

Figure 18: Walk the dog effect

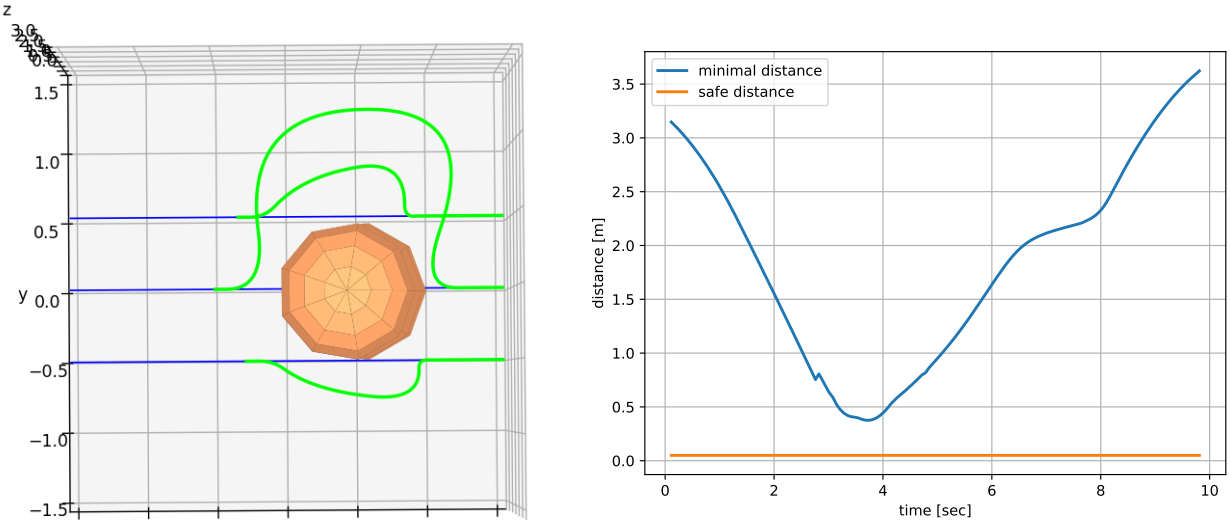
Since the algorithm first generates and evaluates the path candidates with constant velocity and just after that design the velocity profile the following could and did happen. When the drone started from behind the obstacle which was slightly slower than the drone (around $0.1 m/s$) the drone did not overtake or the obstacle or just slowed down, instead

hence it did an evasive maneuver which resulted in a reduced velocity in the direction of the movement of the obstacle therefor it stayed behind it.

4.2.2 Avoiding an obstacles with multiple drones

In this case the task was to avoid an column-shaped obstacle with a radius of $r_o = 0.25m$ with multiple drones which was starting positions were $0.5m$ distance from each other. The drones started at the same time and moved to the obstacle which came in their direction resulting a frontal collision.

At the first few tests the middle drone frequently stuck between a drone at the side and the obstacle, because the circle with the biggest radius did not ensure a path that would be steep enough to avoid the drone at the side in the same direction as the obstacle. However after the introduction of a new circle with bigger radius ($1m$) this problem was solved.



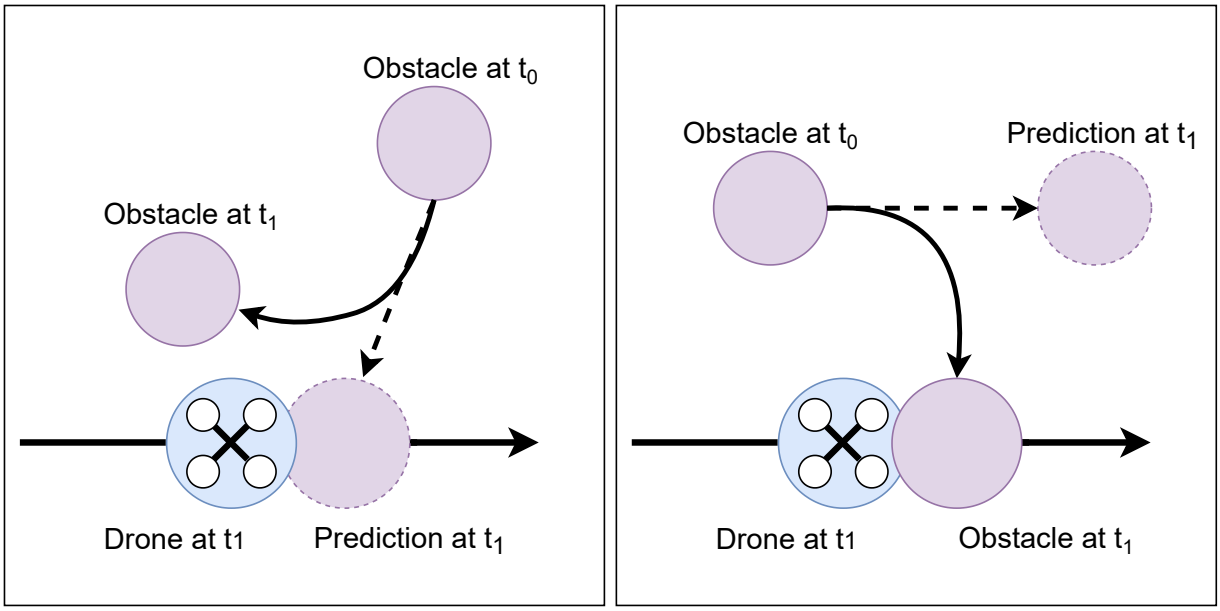
(a) Constantly updated paths

(b) Minimal distance between the drones and the obstacle

Figure 19: Problems with the assumption of constant velocity motion

4.2.3 Avoiding obstacle moving in a circular arc

The most common problem of the algorithm is caused by the assumption of a constant velocity for the obstacles. Although increasing the size of the obstacles during the prediction of their future positions meant to compensate for it, the larger we choose the γ value, the more amount of false collision warning will accrue, hence more evasive trajectory will be generated unnecessarily. However by not increasing the size of the obstacle when predicting its future position can result in a similar false collision warning see Figure 20a. Furthermore it can cause a more severe problem when there is collision in the future but the algorithm does not respond, see Figure 20b.



(a) False collision warning

(b) Not seen collision

Figure 20: Problems with the assumption of constant velocity motion

5 Real flight experiments

5.1 Hardware and software setup

The experimental setup consists of the Crazyflie drones, the Optitrack motion capture system (Optitrack image processing server and infrared cameras), and a ground control PC. The block diagram presenting the interconnection of the components is shown in Figure 21. The quadcopter is equipped with an IMU containing a 3D accelerometer, gyroscope, magnetometer and barometer, and it has two micro-controllers: a STM32F405 for running the light controller, and a nRF51822 for radio communication and power management. The quadcopter runs the original Bitcraze firmware, while on the server the Crazyswarm software platform is used to ease the implementation and configuration of high-level control components [11]. The onboard tracking controller of the Crazyflie drones which guarantees the precise tracking of the trajectories is a Cascaded PID controller[1]. Optitrack is a high precision motion capture system with sub-millimeter resolution. We use it to obtain precise pose measurement of the drones and unknown obstacles in real time.

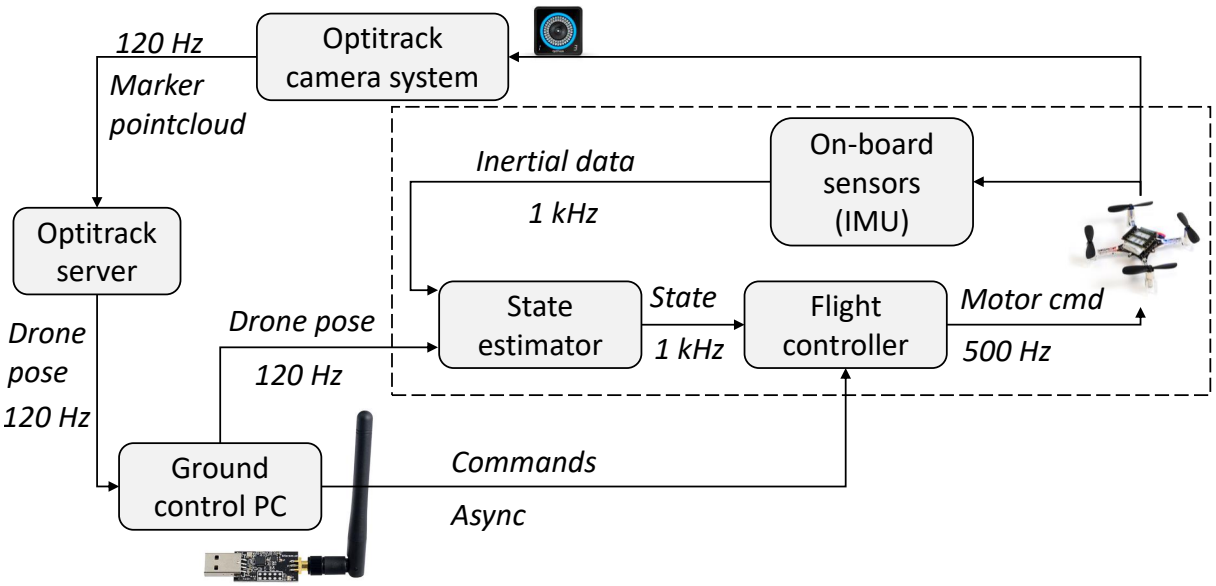


Figure 21: Block diagram of the experimental setup: indoor quadcopter navigation with internal and external measurement system [1].

5.2 Flight tests and results

The tests took place in a drone arena (see Figure 23) which floor area was $3m$ x $3m$, and the drones were limited to a minimal and maximal flying height of $0.2m$ and $1.3m$. The floor area and the maximal flying height was set to keep the drones in the area that the Optitrack system can see. The minimal flying height was set considering the pedestal of some obstacles and the safety distance which was $d_{safe} = 0.1m$ during the tests. We used

[1]Cascaded PID controller https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/controllers/#cascaded-pid-controller

four drone during the test, six static obstacle and a F1TENTH (see in Figure 22) car with a rod served as an obstacle with unknown movement.

Figure 22: F1TENTH with a rod

In the implementation the *scene construction* runs a-priori to the rest of the algorithm. With this method the markers which used to measure the positions of the obstacles can collected before flight.

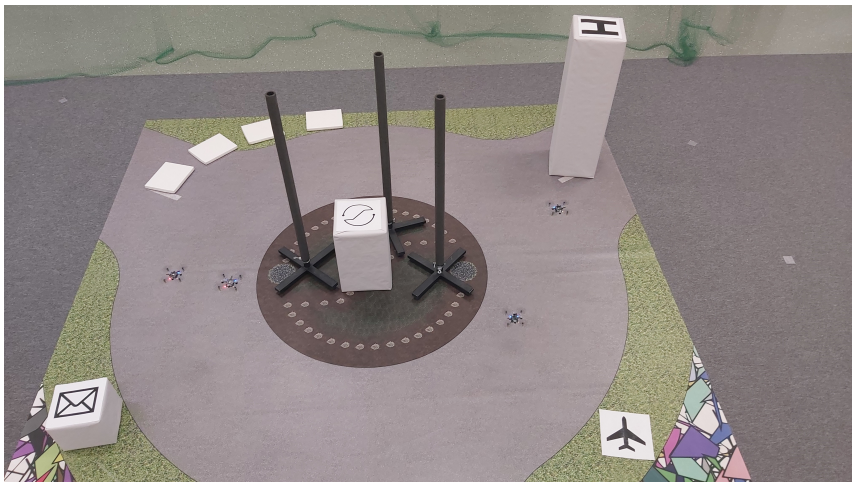


Figure 23: Drone arena

A video illustrating the simulation results is available at https://youtu.be/5vepASKlTwg

The drones path following accuracy during the real life experiment are shown in Figure 25 and the minimal distance between the drones during the flight can be seen in Figure 24. The drones kept a safe distance from each other during their flight.

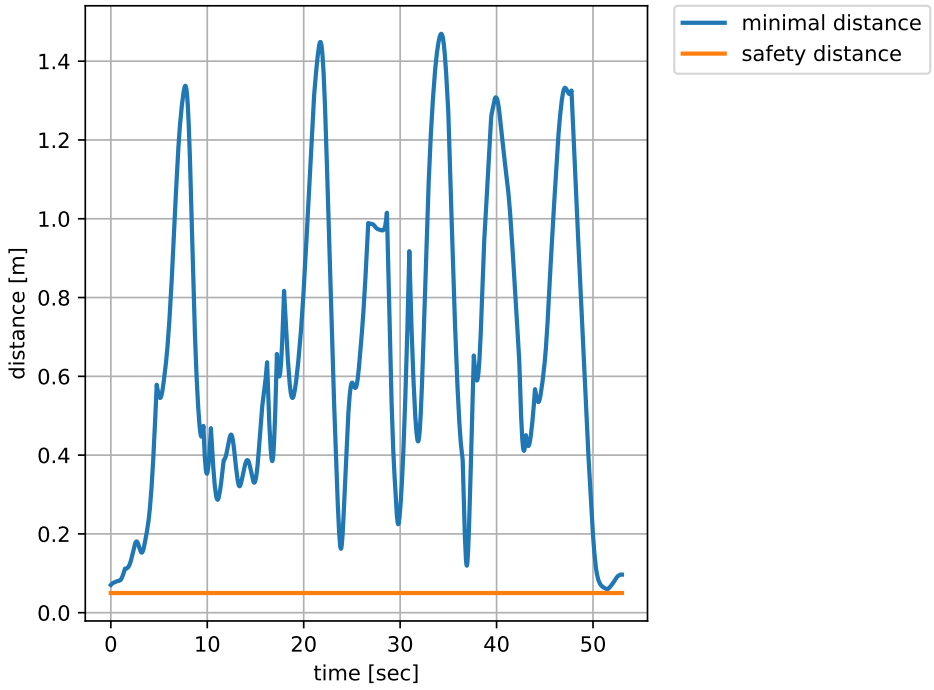


Figure 24: Minimum distance between the drones during the real flight experiment

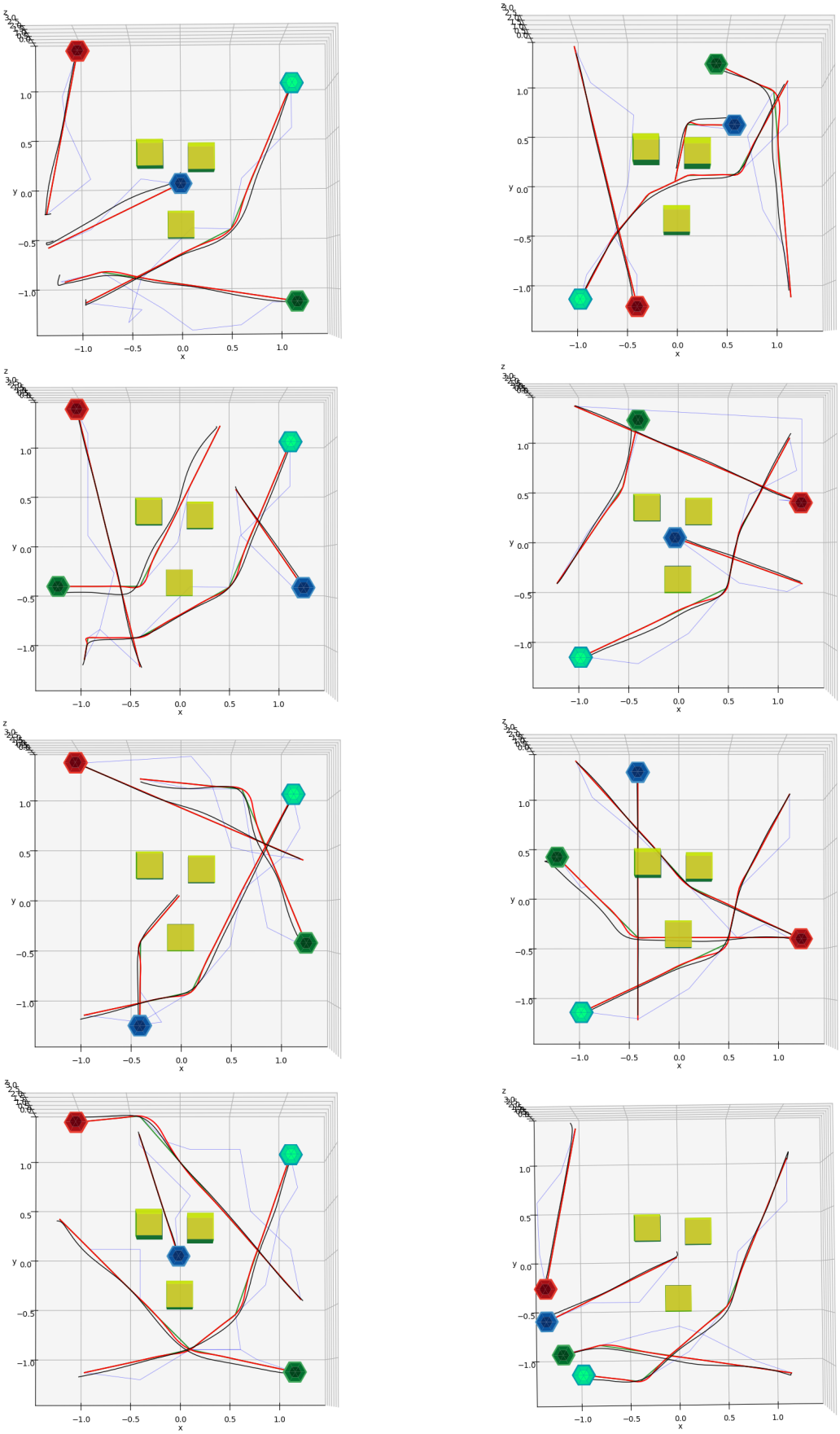


Figure 25: Expected (red) trajectories compared to the real (black) trajectories

6 Conclusions

We present a trajectory planning method for a group of drones, combining the advantages of the fast, graph-search based path planner and velocity profile optimization. We also complement the procedure by being able to modify the original trajectories to avoid unknown moving obstacles that are predicted to collide with the drones.

With the trajectory planning algorithm (Section 3.3) the drones can safely fly in dense environments among static and dynamic obstacles. This method is scales well with increasing number of drones and obstacles. The trajectory plan outputs have been tested and executed safely in numerous times on a team of 4 drones, as well as in simulations of up to 10 drones. Future improvements for the algorithm could be: replacing the A* with jump point search algorithm (JPS) which further optimize the process of the A* algorithm through the neighbour purging rule and the forced-neighbour judgement method, hence reducing the time and space costs of the algorithm [9]; in the velocity profile optimization step the constrains for the velocity and acceleration could be reformulated and expanded to a general 3D nonholonomic kinematic model [4], which helps to design a trajectory with more traceable velocity profile in case of sharp turns.

The algorithm for avoidance of obstacles with unknown movement 3.4 turned out to be expressly hyper parameter dependent. In an industrial, indoor environment this is not a problem because it can be fine-tuned for the obstacles that may occur during its operation, but in case of a fully unknown environment it would not be the best choice. To further improve the algorithm as it was mentioned in Section 4.2.3 a better movement prediction method should be developed [5].

In future work, we plan to apply the majority of the proposed enhancements and want to develop a method for the optimal setting of hyper parameters.

Acknowledgment

I would like to thank Tamás Péni and Roland Tóth at SZTAKI for their professional help and support during the research.

The research was supported by the European Union within the framework of the National Laboratory for Autonomous Systems. (RRF-2.3.1-21-2022-00002)47

References

[1] P. Antal, T. Péni, and R. Tóth. Nonlinear control method for backflipping with miniature quadcopters. *IFAC-PapersOnLine*, 55:133–138, 01 2022.

[2] Marco Cococcioni and Lorenzo Fiaschi. The big-m method with the numerical infinite m. *Optimization Letters*, 15(7):2455–2468, Oct 2021.

[3] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. A case for time-dependent shortest path computation in spatial networks. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 474–477, 2010.

[4] Taha Elmokadem and Andrey V Savkin. A hybrid approach for autonomous collision-free uav navigation in 3d partially unknown dynamic environments. *Drones*, 5(3):57, 2021.

[5] A. Elnagar and K. Gupta. Motion prediction of moving objects based on autoregressive model. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 28(6):803–810, 1998.

[6] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[7] Wolfgang Hönig, James A Preiss, TK Satish Kumar, Gaurav S Sukhatme, and Nora Ayanian. Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics*, 34(4):856–869, 2018.

[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[9] Yuan Luo, Jiakai Lu, Yi Zhang, Qiong Qin, and Yanyu Liu. 3d jps path optimization algorithm and dynamic-obstacle avoidance design based on near-ground search drone. *Applied Sciences*, 12(14), 2022.

[10] Daniel Mellinger, Aleksandr Kushleyev, and Vijay R. Kumar. Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams. *2012 IEEE International Conference on Robotics and Automation*, pages 477–483, 2012.

[11] James A. Preiss, Wolfgang Honig, Gaurav S. Sukhatme, and Nora Ayanian. Crazyswarm: A large nano-quadcopter swarm. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3299–3304, 2017.

[12] Christoph Sprunk. Planning motion trajectories for mobile robots using splines. 2008.

[13] Tim Stahl, Alexander Wischnewski, Johannes Betz, and Markus Lienkamp. Multilayer graph-based trajectory planning for race vehicles in dynamic scenarios. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, oct 2019.

[14] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IROS*, pages 5026–5033. IEEE, 2012.

[15] Liang Zhao, Tatsuya Ohshima, and Hiroshi Nagamochi. A* algorithm for the time-dependent shortest path problem. 01 2008.