Drone Geofencing

UNIVERSITÉ DE MONTPELLIER

Drone GeoFencing & Université de Lorraine

Pyroscan
Planification dynamique de trajectoires en deux phases

Auteur :

BIDOU Oscar
GIROUDEAU Rodolphe
MICHEL Fabien

28 août 2025

Résumé

Dans le cadre de mon Master 2 à l'Université de Lorraine et de mon alternance chez Drone Géofencing, j'ai conçu et implémenté un algorithme de planification de trajectoire pour drones évoluant en environnement 3D contraint. Le système repose sur une architecture bi-phases : un planificateur global inspiré de A^* et un planificateur local dynamique et statique, générant des points de passage associés à des vitesses et des *TimeStamp*, permettant le respect des contraintes physiques des drones.

Notre planificateur global pseudo-3D, reposant sur A^*, se révèle jusqu'à 15 fois plus rapide qu'un A^* classique pour des trajectoires de longueur similaire. Cette nouvelle méthode, inspirée d'un algorithme de JPS, permet un contournement rapide et à moindre coût des zones de restriction ou d'interdiction de vol. Le planificateur local sélectionne une primitive de mouvement conforme aux contraintes physiques et garantissant l'évitement des obstacles dans un temps négligeable. Cette méthode est extrêmement performante pour une planification online (dynamiquement) et offline (statiquement), et pourrait à terme permettre des vols en formation. Cette combinaison permet de déterminer efficacement une trajectoire entre deux points, dès lors qu'elle existe et que le drone est physiquement capable de respecter ces contraintes pour la durée impartie.

Nous avons ainsi adapté des algorithmes existants aux environnements de Drone Géofencing, afin d'en optimiser les performances. La pipeline complète génère une liste générique de `WayPointAction`, laquelle est ensuite convertie en commandes universelles, interprétables par n'importe quel kit de développement logiciel de drone (SDK).

Enfin, ce travail ouvre des perspectives vers la planification distribuée multi-agents et l'intégration de contraintes dynamiques (conditions de vent, dérives), pour répondre à des exigences accrues de temps réel, de robustesse, d'efficience énergétique et d'optimalité au sens de Pareto.

Table des matières

Chapitre 1

Structures d'accueil

Mon alternance se déroule au sein de Drone Geofencing, une entreprise nîmoise. Celle-ci m'a envoyé en mission de recherche du 1er mai au 10 septembre au sein du LIRMM (Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier) de l'Université de Montpellier.

1.1 Drone Geofencing

Drone Geofencing est une société par actions simplifiée (SAS) nîmoise qui développe une solution matérielle et logicielle pour la gestion d'une flotte ou d'un essaim de drones. L'entreprise dévelope un outil de gestion numérique permettant toutes les étapes de la planification de vol ainsi que la supervision des missions en temps réel. La société, créée en 2019, est composée de 18 personnes, dont 4 thésards en collaboration avec le LIRMM (Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier) dans le cadre du projet PyroScan.

Drone Geofencing produit deux interfaces numériques :

— Station**DRONE**, permettant la supervision en temps réel de la mission de vol ;
— Gesta**DRONE**, qui permet aux utilisateurs de planifier les missions de vol et de superviser les équipes de dronistes.

Gesta**DRONE** est composé de quatre modules distincts :

— module exploitant : centralisation et gestion de la documentation réglementaire ;
— module télépilote : suivi des compétences des dronistes et de l'ensemble de leurs activitées ;
— module matériel : suivi de la maintenance des drones et des charges utiles ;
— module demande de vol : automatisation des demandes d'autorisation de vol.

Station**DRONE** est également composé de quatre modules :

— cartographie : visualisation du plan de mission et de la position des drones ;
— pilotage de vecteur : d'un drone particulier de l'escadrille ;
— pilotage de capteur : prise de contrôle d 'une charge utile spécifique (ex. : boule optronique) ;
— vue multiplexe : retransmission en temps réel du flux vidéo des caméras embarquées ;
— pilotage de drone en essaim : prise de contrôle de l'essaim complet, ordres de types de vol, points de passage ou commandes d'urgence : RTH (Return-to-Home) et FTS (Flight Termination System).
— galerie multimédia : stocke les vidéos des caméras embarquées ;

Le backend des applications est également en développement : ManaDG qui permet la configuration des différents types/versions des commandes, des drones, des manettes de contrôle, et des scénarios personnalisés. En plus de ces solutions numériques, l'entreprise développe une nouvelle carte mère pour drones, Sygma**DRONE**, qui offrira une puissance accrue et une portée étendue.

Drone Geofencing regroupe en un seul service l'ensemble des besoins des entreprises : demandes d'autorisation de vol, préparation de la mission, supervision du matériel, conformité réglementaire des dronistes, pilotage et gestion de divers types de drones (quadricoptères, octocoptères, etc.). L'entreprise commercialise aussi des algorithmes d'IA (en collaboration avec l'équipe ICAR du LIRMM) pour la reconnaissance en temps réel des points d'intérêt. Dans le cadre du projet PyroScan, l'objectif est de détecter dynamiquement les feux de forêt à l'aide de caméras rotatives et de drones de reconnaissance. L'automatisation des missions passe également par le développement d'un système multi-agents (SMA) permettant de définir la composition de l'essaim et les interactions entre chaque type de drone.

Dans un premier temps, toute la gestion des flottes s'effectuera depuis une station sol. Avec le développement de Sygma**DRONE**, les drones pourront assumer une charge de calcul plus importante et gagner en autonomie.

1.2 LIRMM

Le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) est une unité mixte de recherche fondée en 1983, rattachée à l'Université de Montpellier et au CNRS. Il regroupe aujourd'hui près de 300 chercheurs, enseignants-chercheurs, ingénieurs et doctorants. Ses domaines d'excellence couvrent l'informatique fondamentale et appliquée, la robotique autonome et la microélectronique pour systèmes embarqués.

Encadrée par messieurs Giroudeau et Michel, respectivement des équipes MAORE et SMILE du LIRMM, j'ai été envoyé par Drone Geofencing en mission de recherche dans le laboratoire, dans le cadre du projet PyroScan.

1.2.1 Équipe MAORE

L'équipe MAORE (Methods, Algorithms for Operations REsearch) du département Informatique du LIRMM rassemble des spécialistes en optimisation combinatoire, théorie des graphes, programmation mathématique et programmation par contraintes, pour résoudre des problèmes d'optimisation discrète de façon exacte ou approchée. Ses principaux axes de recherche couvrent l'optimisation robuste, la recherche opérationnelle quantique, l'optimisation de réseaux et l'ordonnancement. MAORE collabore étroitement avec des partenaires industriels tels qu'Atoptima, La Poste, NanoXplore, Orange, la SNCF ou Total, contribuant ainsi à la valorisation de ses travaux dans des applications concrètes.

1.2.2 Équipe SMILE

L'équipe SMILE (Système Multi-agent, Interaction, Langage, Evolution) fédère ses travaux autour du paradigme des systèmes multi-agents (SMA), qu'elle utilise pour étudier, modéliser et implémenter des systèmes composés d'entités autonomes et intelligentes. Son expertise couvre à la fois la modélisation individuelle (comportements décisionnels, apprentissage, représentation des préférences) et la structuration collective (organisations, protocoles d'interaction), appliquée à des agents artificiels (robotique), à des modèles d'agents humains (jeux sérieux, simulations sociales) ou à des contextes mixtes. SMILE conçoit également des outils logiciels génériques pour la simulation et l'analyse de ces systèmes complexes.

Chapitre 2

État de l'art

2.1 Sujet de l'alternance

L'entreprise a pour objectif de concevoir un système multi-agents hétérogène traitant des images en temps réel et assignant certains drones ou groupes de drones à une mission particulière. La question d'un calcul de trajectoire décentralisé et en temps réel est centrale au problème.

L'objectif de cette alternance est de produire un code implémenté qui réponde aux exigences des différentes situations possibles.

Dans son état final, la solution devra permettre le calcul en temps réel, sur des systèmes embarqués, de trajectoires praticables par les vecteurs, en évitant des zones de restriction de vol connues en amont et des obstacles dynamiques détectés durant le vol. La solution doit être suffisamment modulaire pour permettre le calcul d'une trajectoire praticable par tout type de drone de l'opérateur, qu'il soit à voilure fixe ou tournante (quadrirotor/hexarotor). La solution devra donc prendre en considération les différentes caractéristiques physiques de vol propres à chaque drone.

Trois missions types sont prises en exemple :

— **Pyroscan** : Dans le cadre du projet Pyroscan, un drone ISR (drone de reconnaissance) survole une forêt à haut risque de départ de feu. Si de la fumée est détectée, un «drone in the box» est envoyé sur place pour prendre des images de plus près. D'un point de vue calcul de trajectoires uniquement, il faudra prendre en considération la distance des « drone in the box » (drone stocké dans une boîte propre où il recharge ses batteries) au point d'intérêt, l'état des batteries, imposer les zones de restriction et favoriser certaines altitudes.

— **17^{e} GA** : Le 17^{e} groupement d'artillerie demande une présentation technique des différents concurrents pour une offre de gestion d'une flottille de drones. Lors de cette présentation, 17 drones seront en vol avec des objectifs distincts par escadrille et une organisation propre. Dans un premier temps, les drones voleront en escadrilles ; dès qu'un premier drone est abattu, la formation se disperse. Il est donc nécessaire de gérer la génération de trajectoires dynamiquement, en prenant en compte les capacités physiques du drone mais aussi la trajectoire des autres appareils.

— **RIDEAU2FER** : Un consortium d'industriels, dont Drone Geofencing fait partie, propose à l'armée de terre le projet RIDEAU2FER. Le but est de larguer des détecteurs de vibrations par drone selon un schéma prédéfini. Ces détecteurs renvoient des informations en continu à une station au sol, laquelle peut déployer une escadrille de drones si nécessaire. La gestion du vol en formation, de la programmation de vol multiple et la résulation multi-objectif de la trajectoire sont donc des aspects important du calcul de trajectroires.

En résumé, la solution doit permettre de calculer en temps réduit (ordre de la seconde) une trajectoire physique adaptée aux caractéristiques de vol de chaque drone, tout en évitant les zones de restriction ou d'interdiction et en minimisant la distance parcourue. Elle doit également offrir la possibilité de programmer des vols en formation et, idéalement, de reprogrammer dynamiquement la trajectoire à moindre coût pour prendre en compte les obstacles détectés grâce au SLAM.

2.2 Calcul de trajectoires

2.2.1 Définition de la continuité d'une trajectoire en robotique

La notion de chemin praticable pour un drone sous-entend le respect de ses caractéristiques physiques (accélérations maximales, vitesse maximale, vitesse actuelle), et impose donc une condition de continuité sur la trajectoire. Il est nécessaire de différencier la notion de **chemin**, qui correspond à une série de points de passage

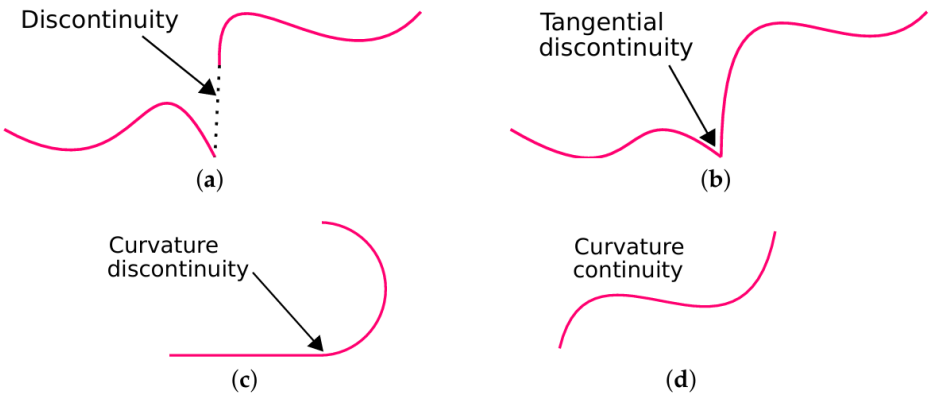


FIGURE 2.1 – De [33] - Continuité non paramétrique. (a) Segments de courbe discontinus. (b) Continuité G_0. (c) Continuité G_1. (d) Continuité G_2.

bruts, ne respectant aucune contrainte physique, et celle de **trajectoire**, qui correspond à une série de points de passage lissés, praticables par le drone.

En robotique, deux types de continuité d'une trajectoire sont classiquement distingués : la continuité géométrique, notée G^n, et la continuité paramétrique, notée C^n. La continuité géométrique d'ordre n garantit que la forme de la trajectoire est régulière et que ses dérivés géométriques (orientation, courbure, ...) le sont également, indépendamment de la paramétrisation. La continuité paramétrique d'ordre n assure que la fonction paramétrique $p(t)$ et ses dérivées jusqu'à l'ordre n sont continues. La continuité C^n inclut toujours la continuité G^n, $C^n \Rightarrow G^n$. C'est cette dernière qui nous intéresse afin de construire une trajectoire.

- La continuité C^0 ne garantit qu'une position ($p(t)$) continue : le drone doit alors parfois s'arrêter complètement pour changer de direction.
- Le niveau minimal de continuité pour un drone est C^1, c'est-à-dire une trajectoire à vitesse ($\dot{p}(t)$) continue mais dont l'accélération peut être discontinue. Cette discontinuité d'accélération entraîne un saut de la force centripète subie par le drone, risquant de compromettre sa stabilité. En équivalence géométrique, G^1 signifie continuité tangentielle avec une discontinuité de la courbure 2.1.
- Une trajectoire C^2 assure la continuité de l'accélération ($\ddot{p}(t)$), et donc une courbure continu (G^2) sur l'ensemble de la trajectoire.
- La continuité C^3 impose en plus la continuité du *jerk* (dérivée de l'accélération : $\dddot{p}(t)$) [33].

Si l'objectif physique principal est de générer une trajectoire dérivable au moins en C^2, de nombreux travaux ajoutent un terme de coût visant à minimiser le *jerk* (sans pour autant exiger formellement la continuité C^3) [22, 31, 32]. Pour des vols extrêmement agressifs à haute vitesse, la minimisation du *snap* (dérivée du *jerk*) devient également cruciale [29, 30, 36, 45].

2.2.2 Différentes méthodes de traitement d'obstacles

Les zones de restriction de vol sont des obstacles complexes en 2D (polygones), mais simples en 3D : elles sont délimitées par une altitude minimale z_{zmin} et une altitude maximale z_{max}. Pour accélérer les calculs d'intersection entre une droite et ces obstacles, ou pour déterminer si un point donné appartient à l'un d'eux, il est possible de les simplifier à l'aide de :

- Boîtes englobantes alignées sur les axes (axis-aligned bounding boxes, AABB) : la plus petite boîte alignée sur les axes qui contient tous les points de l'obstacle.
- Boîtes englobantes orientées (oriented bounding boxes, OBB) : la plus petite boîte, non alignée sur les axes, contenant tous les points de l'obstacle.
- Ellipsoïdes : la plus petite ellipsoïde enveloppant l'ensemble des points de l'obstacle.

Bien que ces approches offrent des temps de calcul très courts, de génération et de traitement, elles induisent une perte d'information trop importante, entraînant des contournements excessifs de par la nature des obstacles qui peuvent être extrêmement grands (ex : zone de restriction de vol d'un aéroport).

Pour un traitement plus fin de l'environnement 3D, des méthodes plus sophistiquées sont utilisées :

- Voxélisation [6, 15, 20, 22, 38] : division de l'espace en voxels - un petit cube élémentaire d'une grille 3D régulière. Pour un pas Δ le voxel (i, j, k) est le cube :

$$V_{i,j,k} = [i\Delta, (i+1)\Delta] \times [j\Delta, (j+1)\Delta] \times [k\Delta, (k+1)\Delta]. \tag{2.1}$$

- Octrees [19, 28, 43] : arbre hiérarchique composé de nœuds. Chaque nœud est soit une feuille, soit subdivisé en exactement huit nœuds fils. Cette subdivision a lieu si un obstacle est présent dans le cube

associé au nœud et que la profondeur maximale n'est pas atteinte. Cette représentation de l'environnement, affinée à proximité des obstacles, rend l'octree particulièrement efficace lorsque l'environnement est composé de grands espaces vides (voir figure 2.2).

Soit $C_0 \subset \mathbb{R}^3$ un cube initial et $D_{\max} \in \mathbb{N}$ une profondeur maximale. Un *octree* est un arbre enraciné (V, E, r) muni d'une application

$$C : V \longrightarrow \{\text{cubes dans } \mathbb{R}^3\}$$

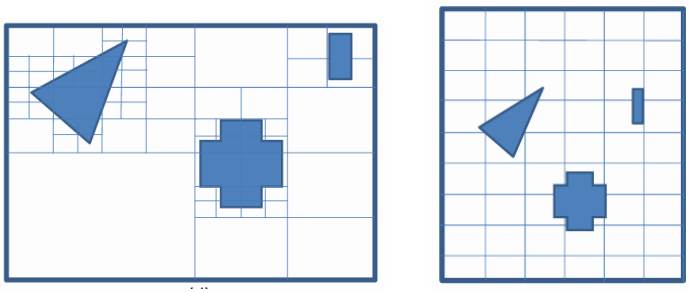
telle que $C(r) = C_0$.

$$v \text{ est subdivisé} \iff \big(\exists \text{ obstacle dans } C(v)\big) \wedge \big(\text{depth}(v) < D_{\max}\big).$$

Dans ce cas, v a exactement huit enfants $\{v_\varepsilon\}_{\varepsilon \in \{0,1\}^3}$ avec, si $C(v) = [a_x, b_x] \times [a_y, b_y] \times [a_z, b_z]$,

$$C(v_\varepsilon) = \prod_{i \in \{x,y,z\}} \Big[a_i + \tfrac{\varepsilon_i}{2}(b_i - a_i),\ a_i + \tfrac{\varepsilon_i + 1}{2}(b_i - a_i)\Big].$$

Un nœud v qui n'est pas subdivisé est une feuille.



(a) Quadtree (b) Voxelisation

FIGURE 2.2 – De [16] Représentation d'un environnement

Une variante des voxels est la création d'un "Euclidean Distance Transform" (EDT), où chaque voxel stocke la distance au point obstacle le plus proche, ce qui permet d'ajouter une force répulsive directement dans le coût de la trajectoire [22] et d'alimenter des algorithmes de planification par forces répulsives [38] ou de bande élastique pour obstacles dynamiques [8, 11].

Une solution pour réduire le nombre de voxels stockés est de garder en mémoire uniquement les voxels occupés par un obstacle. Cette méthode permet d'éviter de consommer de la RAM pour les zones vides. Comme précisé ci-dessus, les zones de restriction de vol peuvent être extrêmement larges et occuper un pourcentage important de la géocage (zone de mission). Elle ne fonctionne pas pour les EDT, où chaque voxel vide doit stocker la distance du plus proche obstacle. En principe, le temps de calcul d'un octree est bien inférieur à celui d'une voxélisation [28]. D'autant plus qu'en arrêtant l'approfondissement d'une branche lorsqu'elle est entièrement incluse dans un obstacle, le temps de calcul ainsi que le poids mémoire de l'octree sont significativement réduits . Comme précisé ci-dessus, les zones de restriction de vol peuvent être extrêmement larges et occuper un pourcentage important de la géocage (zone de mission). Ces importantes dimensions posent la question de l'environnement optimal (le plus rapide à construire et le moins coûteux en RAM). Bien que la création d'un graphe de voisinage est évidente sur une voxélisation, chaque voxel étant lié à ses voisins selon un déplacement Δ sur x, y et z (voir équation 2.1) ; la construction du graphe associé à un octree reste plus complexe. Deux types de graphes inhérents à un octree existent : le dual et l'edge corner [19].

Une méthode pour se passer de la construction du graphe inhérent à l'octree consiste à n'utiliser l'octree que pour valider rapidement la présence d'un point ou d'un segment. Les voisins d'un point donné sont exprimés selon un déplacement, un pas, défini au préalable. Cette méthode permet d'orienter la recherche des voisins. Plutôt que de considérer le pas comme un simple écart le long des axes x, y ou z, il peut se décomposer en :

— un déplacement le long de l'axe principal (direction départ-arrivée) projeté sur le plan XY ;
— des déplacements perpendiculaires à cet axe dans le plan XY ;
— un déplacement standard selon l'axe Z.

Bien que cette approche permette de réduire le nombre de voisins étudiés – et donc de réaliser un gain en performance – elle reste toutefois plus lente qu'un déplacement le long des axes avec un pas égal à la taille de l'espace divisé par $2^{\text{profondeur}}$. En définissant un voisin d'un point comme un déplacement axial à une distance

inférieure ou égale à la taille des plus petites feuilles de l'octree, on élimine la nécessité de tester la ligne de visée (LOS) vers les voisins. En effet, si un voisin ne fait pas partie d'une feuille occupée de l'octree, aucune feuille de plus petite taille ne peut intersecter le segment reliant le point courant à ce voisin — ce segment étant lui-même au plus grand que la plus petite feuille. Si la voxélisation et la création d'un octree permettent des modifications dynamiques à moindre coût, cela ne rentrera pas en compte, l'environnement étant dynamique uniquement localement et non globalement. Les zones de restriction de vol ne sont pas modifiées durant le vol, mais des obstacles aviaires peuvent apparaître avec le LIDAR du drone. Autrement dit, les modifications de l'environnement restent moindres proportionnellement aux zones de restriction de vol.

2.2.2.1 Compléxité et coût mémoire du traitement de l'environnement

Pour comparer correctement le coût mémoire et le temps de calcul entre une voxélisation et la création d'un octree, nous supposons que les voxels ont la même taille que les feuilles de profondeur maximale de l'octree. Autrement dit, le nombre de voxels occupés est égal au nombre de feuilles occupées de l'octree. Le paramètre de profondeur p définit indirectement la taille des voxels selon chaque axe : taille de l'axe divisée par 2^p • **Coût mémoire** : Pour un octree de profondeur p, sur un espace de dimension $\mathbb{R}^3$, où m est le poids en octets d'une feuille, le nombre total de nœuds N dans le pire des cas est de :

$$N_{octree} = \sum_{i=0}^{p} 8^i = \frac{8^{p+1} - 1}{7} \tag{2.2}$$

et le poids mémoire dans le pire des cas est de

$$M_{total,octree} = N_{octree} \cdot m \tag{2.3}$$

Dans le pire des cas, la voxélisation ne coûterait que :

$$M_{total,voxel} = 8^p \cdot m \tag{2.4}$$

• **Validation d'un point dans un octree** : Vérifier si un point situé dans l'octree est valide ou non nécessite de tester si la feuille contenant ce point est marquée comme valide. La complexité de cette opération est en $O(p)$, où p est la profondeur de l'octree.

• **Validation d'un point dans un modèle voxel** : Dans un modèle voxel, la validité d'un voxel se récupère en temps constant, soit $O(1)$, grâce à l'accès direct à la donnée de validité.

• **LOS segment–octree** : Pour détecter rapidement l'intersection entre un segment et les feuilles de l'octree (AABB), la méthode des *slabs* est couramment utilisée [5]. Cette méthode présente un coût constant $O(1)$ par feuille testée, pour un coût total dans le pire des cas en $O(R \log R)$, avec $R = 2^p$. Le facteur $\log R$ provient de la vérification des huit sous-feuilles (child nodes) de chaque nœud intersecté par le segment. Revelles et al. proposent une version de la méthode *slab* optimisée pour les octree, qui évite de tester systématiquement les huit fils d'un nœud intersecté, réduisant ainsi le nombre de tests [35].

• **LOS dans un monde voxel** : Dans un espace voxel 3D, l'algorithme développé par Amanatides et Woo permet de déterminer efficacement l'ensemble des voxels traversés par un segment [3]. Comme pour le test des *slabs*, le coût est en $O(1)$ par nœud visité, pour un coût total en $O(R)$, où R est le nombre de voxels traversés.

2.2.3 Différentes approches de calcul dynamique de trajectoires physiques

De nombreuses méthodes de calcul de trajectoires praticables existent, divisées en deux grandes approches.

La première approche consiste en un unique planner qui prend directement en compte les caractéristiques physiques du drone et de l'environnement et qui retourne une trajectoire allant du départ global au but global (Kinetic A^*, Kinetic RRT, Global Motion Planner, réseau de neurones) [8, 10, 23, 38, 40, 42, 45, 46]. Cette approche est souvent extrêmement coûteuse en raison du nombre énorme de possibilités dans un monde physique continu.

La seconde approche consiste à diviser le calcul en deux parties distinctes : un global planner et un local planner (ou lissage d'angle). Le global planner est un algorithme classique de calcul de trajectoires retournant des points de passage (Dijkstra, A^*, θ^*, D*, RRT, etc.) [13, 15, 18, 19, 25, 37]. La seconde étape peut consister en un simple lissage des angles, afin de rendre plus douces les transitions entre les points de passage. C^1 ou encore C^2 (courbes de Dubins, cycloïdes, hypocycloïdes) [6, 24, 26, 29, 31, 32, 33, 41, 44]. Elle peut également être un local planner, replanifiant la trajectoire pour chaque segment des points de passage définis par le global planner (splines cubiques, courbes de Bézier, motion planner) [20, 22, 30, 36, 37].

Le but final étant de trouver une solution légère en calculs pour obtenir une réponse dans l'ordre de la seconde sur l'embarqué, les solutions avec un unique planner ne peuvent pas être appliquées.

Le lissage simple des angles (courbes de Dubins, cycloïdes, hypocycloïdes) est une solution purement géométrique, ne garantit qu'une continuité G^n et non C^n. Afin de permettre un profils de vitesse pour le drone

sur cette trajectoire géométrique, il est possible de calculer la vitesse moyenne v_{mean} du drone nécessaire pour atteindre le point final dans la durée déterminée T. Si le drone conserve cette vitesse sur l'ensemble de la trajectoire, alors la dérivabilité G^n devient une dérivabilité C^n. Le rayon de courbure minimal, $R_{\min}$, du drone est alors

$$R_{\min} = \frac{v_{\text{mean}}}{a_{\max}},$$

avec $a_{\max}$ l'accélération maximale du drone.

Bien qu'extrêmement simpliste et très rapide à calculer (de l'ordre de la milliseconde), cette solution ne permet pas un calcul efficace d'une trajectoire. Les phases d'accélération et de décélération ne sont pas modélisées, l'accélération des drones étant si rapide qu'il pourrait être jugé inutile de les prendre en compte. Mais, les vitesses ascendante, descendante et horizontale des drones étant rarement les mêmes (le Tundra 2 de Hexadrone a une vitesse horizontale maximale de 18m/s pour une vitesse ascendante maximale de 7m/s et une vitesse descendante maximale de 5m/s [21]), il n'est pas possible de garantir une vitesse unique sur la trajectoire, rendant ces solutions inadatpées.

De même, les solutions purement géométriques telles que les courbes de Bézier et les splines cubiques n'assurent pas le respect du rayon de courbure minimal du drone, forçant une réévaluation de la vitesse sur la trajectoire.

À l'inverse, bien que plus coûteux, les local planners permettent le calcul d'une trajectoire géométrique avec un profil de vitesse adapté, en prenant en compte les différentes contraintes physiques des drones [20, 22, 30, 36].

Nous implémenterons ici uniquement le local planner défini par Hidra, qui offre une grande flexibilité des calculs et des paramètres, tout en étant nativement C^2 et conçu pour une exécution en temps réel [22].

2.2.4 Étude des solutions d'algorithme de calcul de chemins

Trois grandes familles d'algorithmes sont utilisées pour le calcul de chemin brut (chemin non praticable) en robotique : bio-inspirés (le plus connu étant les algorithmes génétiques GA), à base de graphes (A^*) et à base d'échantillonnage (RRT) [16]. Dans l'optique d'un calcul en temps réel, les algorithmes bio-inspirés sont d'office écartés en raison de leur coût trop élevé pour trouver une solution. Les solutions basées sur les graphes et la génération aléatoire sont les plus utilisées pour du calcul *online*.

A^* est l'algorithme le plus connu : il est extrêmement rapide et facile à implémenter. Cette méthode construit un graphe de recherche où les points à explorer (openset) sont les voisins des points déjà explorés (closeset) (voir annexe 1). Le point n_k sélectionné pour l'expansion est celui qui minimise l'heuristique [15, 20] définie par l'équation suivante :

$$\forall i, \quad h_i = w_s \,\text{distance}(n_i, \text{start}) + w_e \,\text{distance}(n_i, \text{end}), \quad \text{où } n_i \in \text{OpenSet} \tag{2.5}$$

où w_s et w_e sont les poids déterminant l'importance de chaque facteur. Il est impératif d'adapter ces poids à notre environnement : une bonne heuristique peut faire la différence entre un temps de calcul élevé et très faible.

Pour réduire le nombre de nœuds et accélérer la recherche, θ^* est une variante *any-angle* non native, où un nœud est relié directement à son grand-parent s'il est atteignable, modifiant a posteriori l'heuristique du point [19]. Anya est un algorithme nativement *any-angle* minimisant le nombre de nœuds explorés et offrant des gains allant jusqu'à dix fois la vitesse d'un A^* classique [18]. Mais Anya reste cantonné au 2D et son extension au 3D s'avère complexe et coûteuse. Afin de conserver un algorithme nativement *any-angle* dans un environnement 3D, Ruoqi [19] a proposé l'algorithme *Lazy* θ^*. Les voisins d'un point sont supposés connectables à leur grand-parent avant même leur exploration, ce qui permet une estimation de l'heuristique *any-angle* a priori, minimisant d'autant plus le nombre de voisins à explorer.

RRT génère des points aléatoires dans l'environnement puis les raccorde au point de l'arbre le plus proche, étendant ainsi l'arbre. L'algorithme s'arrête lorsqu'un point ajouté est à une distance ϵ du point final (voir annexe 2). RRT* réorganise dynamiquement l'arbre à chaque insertion, convergeant vers une solution optimale si tous les points possibles sont considérés [31, 36]. Les méthodes bidirectionnelles tirent parti d'une exploration simultanée depuis l'état initial et depuis l'objectif, ce qui permet de réduire considérablement la taille des espaces de recherche et d'accélérer la convergence vers une solution optimale. Par exemple, dans le cadre de la planification de chemin, RRT*-bidirectional (également appelé BI-RRT* ou RRT*Connect) fait croître deux arbres en parallèle, depuis le départ et depuis l'arrivée, et tente de les relier dès que leurs frontières se rencontrent, garantissant ainsi la propriété d'asymptotic optimalité tout en diminuant la complexité moyenne de la recherche [13, 25, 37].

Cette approche bidirectionnelle/non-dirigée peut être transposée à des recherches sur graphe discret : il est ainsi possible concevoir une version bidirectionnelle de l'algorithme A^*, qui exécute deux instances de la recherche heuristique (avant et arrière) et stoppe l'expansion dès que les deux fronts communiquent, réduisant de façon exponentielle le nombre de nœuds explorés dans des graphes de branchement important. De même, *Lazy* θ^*, en conservant son principe de chemins «au plus près de la ligne droite», gagne en efficacité en adoptant

une structure bidirectionnelle, puisqu'elle explore simultanément autour de la source et de la cible avant de lever les vérifications de visibilité au moment de la jonction [15].

Ni RRT* ni A^* ne garantissent une solution optimale en temps réel : RRT* converge vers l'optimum mais reste trop lent. De plus, aucun de ces algorithmes ne peut déterminer instantanément si un chemin existe, ce qui pousse A^* à explorer l'intégralité de l'espace et RRT à atteindre un nombre maximal d'itérations sans certitude de succès.

Le Hierarchical A^* (HA*) présenté à la GDC 2018 permet de tester en $O(1)$ si un passage existe entre deux nœuds, d'éviter les pièges et de fonctionner en 3D [2]. HA* peut être 100 à 1000 fois plus rapide qu'un A^* classique, mais exige un coût mémoire élevé pour stocker l'environnement voxelisé à chaque niveau hiérarchique.

L' environnement étant pseudo-3D, les obstacles sont complexes en 2D (plan xy) et délimités entre deux bornes ($z_{\min}$ et $z_{\max}$) sur la troisième dimension, générant de nombreux points redondants section 2.2.2. Pour optimiser A^*, il est possible de ne récupérer que les voisins dans le plan xy à z fixe. Si le voisin d'un point, appartient à un voxel (ou à une feuille d'octree) occupé, deux nœuds sont ajoutés (à condition qu'ils restent dans la géocage) :

— un nœud en $(x, y, z_{\max} + \epsilon_{\text{safe}})$;

— un nœud en $(x, y, z_{\min} - \epsilon_{\text{safe}})$.

Cette méthode limite l'explosion du nombre de points inhérente à la 3D, nous l'appellerons pseudo-3D A^*.

RRT souffre beaucoup du passage de 2 à 3 dimensions. Il n'existe pas de méthodes d'optimisation permettant de minimiser les points redondants, résultant en une explosion du nombre de points nécessaires pour mapper l'environnement.

Une méthode d'optimisation consiste à introduire une mécanique d'arrêt prématuré. À chaque point rattaché à l'arbre de recherche, la ligne de vue entre ce point et le point d'arrivée est calculée : si elle n'est pas obstruée, le point d'arrivée est directement relié au point actuel. Avec un calcul de LOS rapide, comme précisé dans section 2.2.2.1, le surcoût lié à l'augmentation du nombre de LOS est complètement compensé par la réduction du nombre de points à générer, valider et tester en LOS.

Que ce soit sur un octree ou sur des voxels, la vérification de la validité d'un point s'effectue en $O(p)$ pour l'octree et en $O(1)$ pour les voxels, ce qui reste négligeable. Le coût principal provient de deux facteurs : le calcul de la ligne de visée (LOS) et la gestion de l'openset. En 3D, l'openset peut devenir très volumineux : récupérer le minimum dans une liste non triée se fait en $O(N)$, alors que l'extraction répétée du minimum dans une liste triée coûterait $O(N \log N)$. Pour éviter ce surcoût, l'usage d'un tas binaire (*heap*) permet de récupérer le minimum en $O(1)$ et d'ajouter ou de supprimer un élément en $O(\log N)$.

Pour optimiser la recherche spatiale de voisins dans des algorithmes à pruning (RRT, RRT-bidirectionnel...),il est possible d'intégrer un k-d tree, un arbre binaire multidimensionnel qui partitionne récursivement l'espace. La recherche du plus proche voisin s'effectue en $\Theta(\log N)/O(N)$, et la recherche de tous les points dans un rayon r se fait en $O(\log N + m)$, où m est le nombre de voisins retournés.

Ainsi, la combinaison d'un tas pour l'openset et d'un k-d tree pour les requêtes spatiales permet de conserver chaque opération à coût logarithmique plutôt que linéaire, ce qui est crucial dès que N devient grand.

2.2.5 Étude des solution d'algorithme de calcul de trajectoires dynamiques

La grande majorité des articles sur le calcul de trajectoires dynamiques se basent sur le *Dynamic Window Approach* (DWA) [9, 10, 14, 20]. Cette approche dresse la liste des positions atteignables en fonction de la vitesse du drone et de ses capacités d'accélération sur un laps de temps donné. Elle assure une réactivité instantanée du drone : pour son état actuel, les états atteignables sont déterminés, puis une solution est calculée à partir d'une fonction de perte. DWA ne permet pas de projection temporelle sur la trajectoire, mais réalise uniquement un calcul instantané de la position au pas de temps suivant ; ce faisant, elle ne garantit pas la continuité C^2.

Une autre approche repose sur des algorithmes basés sur la bande élastique, qui déforment dynamiquement la trajectoire en temps réel. Les obstacles repoussent alors les points de la trajectoire selon une force répulsive [8, 11, 38]. Cette méthode ne garantit pas la continuité C^2 et peut poser problème lorsque les obstacles se déplacent trop rapidement.

Le local motion planner défini par Hidra [22] produit une trajectoire polynomiale quintique (de degré cinq) lisse $p(t)$, continue en C^2, tout en conservant un temps de calcul compatible avec une exécution en temps réel (ordre de la milliseconde) (voir figure 2.3). Un polynôme de degré 7 (septique) assurerait la minimisation du *snap*. Le local planner génère des primitives de mouvement (des fonctions polynomiales de degré 5) (voir équation 2.6), qu'il évalue ensuite à l'aide d'une fonction de coût (voir équation 2.7) pour chacun des trois axes $i \in \{x, y, z\}$, en fonction de trois paramètres : la vitesse v, le yaw θ (qui oriente la direction horizontale du vecteur vitesse final selon la direction souhaitée) et l'altitude z. La position $p_i(t)$ en $t \in T$ est alors donnée par le polynôme quintique :

$$p_i(t) = \frac{\alpha_i}{120}\, t^5 + \frac{\beta_i}{24}\, t^4 + \frac{\gamma_i}{6}\, t^3 + \frac{a_i}{2}\, t^2 + v_i\, t + p_i, \tag{2.6}$$

avec

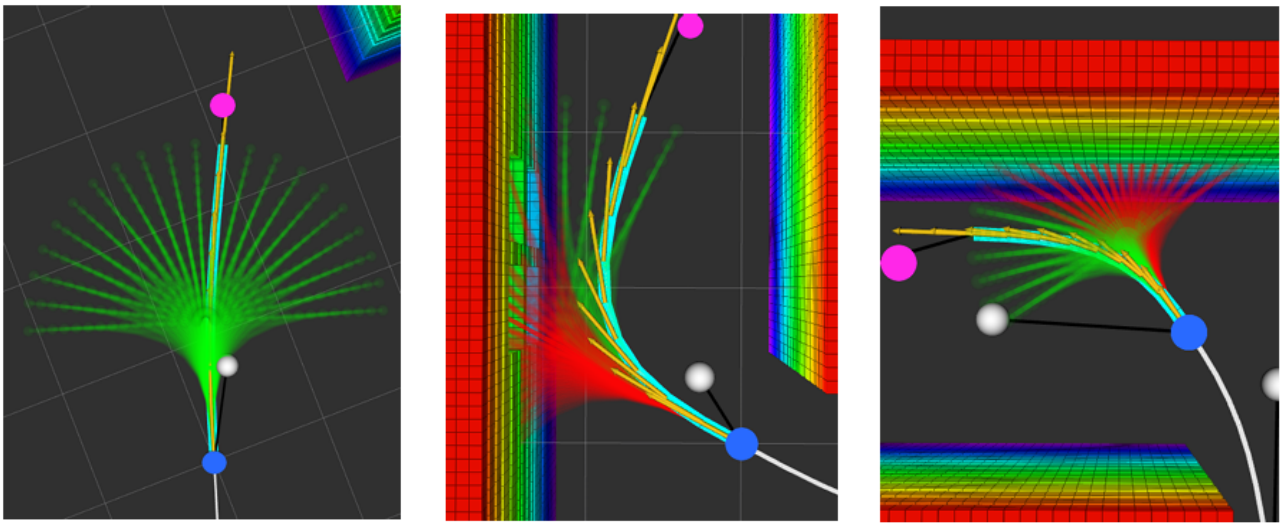
$$\begin{cases} p_i(0) = p_{0,i} \\ \dot{p}_i(0) = v_{0,i} \\ \ddot{p}_i(0) = a_{0,i} \end{cases} \qquad \begin{cases} p_i(T) = p_{f,i} \\ \dot{p}_i(T) = v_{f,i} \\ \ddot{p}_i(T) = a_{f,i} \end{cases}$$

La vitesse est donnée par la dérivée première de p_i, $\dot{p}_i$. De même, l'accélération est donnée par la dérivée seconde de p_i, $\ddot{p}_i$.

$$\dot{p}_i(t) = \frac{5}{120}\,\alpha_i\,t^4 + \frac{4}{24}\,\beta_i\,t^3 + \frac{3}{6}\,\gamma_i\,t^2 + a_i\,t + v_{1,i}.$$

$$\ddot{p}_i(t) = \frac{20}{120}\,\alpha_i\,t^3 + \frac{12}{24}\,\beta_i\,t^2 + \gamma_i\,t + a_i.$$

Pour déterminer les paramètres optimaux, quatre étapes d'optimisation sont réalisées : une première étape d'optimisation du yaw θ_1^*, puis de la vitesse v^* avec θ fixé ; une deuxième étape d'optimisation de θ_2^* avec v fixé ; enfin, une dernière optimisation de l'altitude z^* avec θ et v fixés (voire annexe 3).



(a) Dans une configuration simple sans obstacles, la meilleure primitive est celle la plus proche de l'objectif local.

(b) En présence d'obstacles, toutes les primitives qui se chevauchent doivent être écartées.

(c) Même dans les virages serrés, une large plage d'angles de lacet finaux permet d'obtenir une trajectoire faisable.

FIGURE 2.3 – De Hidra [22] Bibliothèques de primitives de mouvement générées avec différents paramètres de départ et environnements. La vue est une projection en plongée d'un environnement 3D, avec des obstacles. Les primitives sont échantillonnées de manière uniforme sur une plage d'angles de lacet finaux θ_f et de vitesses finales v_f, à une altitude fixe z_f. La position de départ est représentée par le cercle bleu, tandis que l'objectif local est le cercle violet. Les primitives de coût élevé sont en rouge et celles de coût faible en vert. La trajectoire sélectionnée, de coût minimal, est mise en évidence en cyan. De plus, les vecteurs vitesse sont échantillonnés le long de la trajectoire retenue et affichés sous forme de flèches jaunes.

Le local planner étant très modulable, il est possible d'ajouter toutes les contraintes physiques des drones, à savoir :

- une vitesse maximale horizontale : $v_{max,horizontal}$;
- une vitesse maximale ascendante : $v_{max,up}$;
- une vitesse maximale descendante : $v_{max,down}$;
- un rayon de courbure minimal : $R_i \geq R_{min} = \frac{\|\mathbf{v}_i\|^3}{\|\mathbf{v}_i \times \mathbf{a}_{max}\|}$;
- la durée totale de la trajectoire : T ;
- la vitesse initiale et d'arrivée : v_{init}, v_{goal} ;
- l'accélération initiale et d'arrivée : a_{init}, a_{start}

Il est également possible de modifier librement la fonction de coût des primitives, que Hidra définit comme :

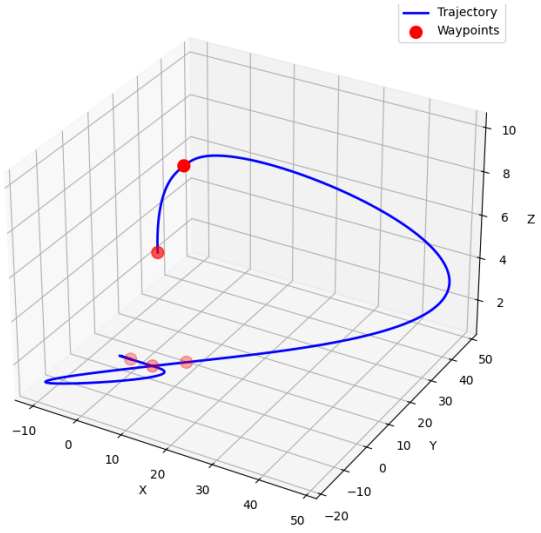
$$E = w_{\text{ep}} \cdot E_{\text{ep}} + w_{\text{dir}} \cdot E_{\text{dir}} + w_{\text{c}} \cdot E_{\text{c}} \tag{2.7}$$

où :

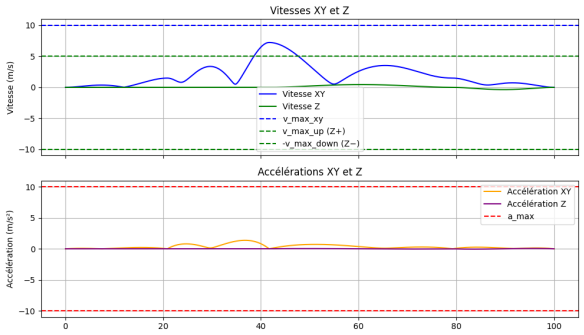
- w_{ep} est le poids du coût E_{ep} (distance entre l'état final et le point cible local) ;
- w_{dir} est le poids du coût E_{dir} (écart angulaire entre la direction de l'état final et celle du point cible local) ;

— w_c est le poids du coût E_c (coût de collision avec les obstacles).

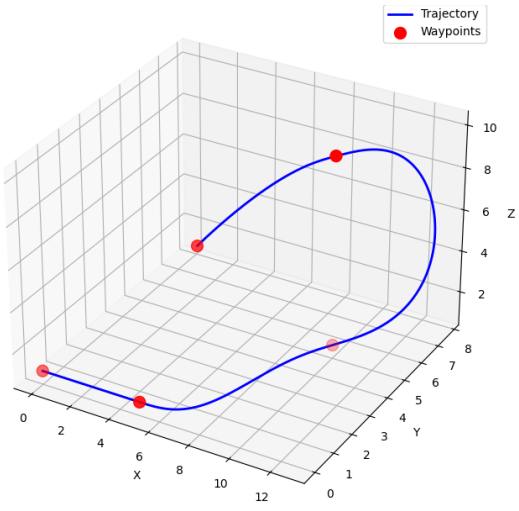
Il est possible d'ajouter un poids w_j associé à un coût E_j minimisant le *jerk* le long de la trajectoire. Lorsque la durée de la trajectoire est imposée, il est crucial de veiller à ce que le drone vole à la vitesse minimale nécessaire pour parcourir l'intégralité de la trajectoire dans le délai imparti ; sinon, le local planner aura tendance à privilégier les vitesses maximales pour respecter cette durée, ce qui peut conduire à des déplacements excessivement importants 2.4.



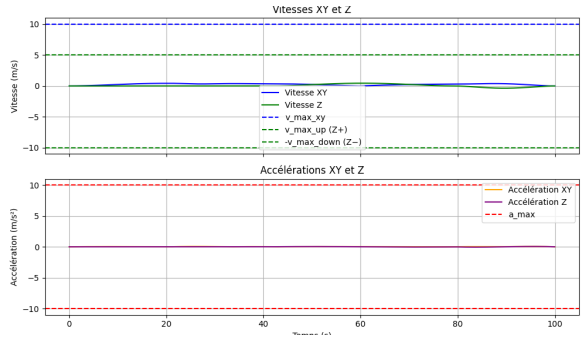
(a) Trajectoire avec $w_j = 0$



(b) Vitesse avec $w_j = 0$



(c) Trajectoire avec $w_j = 1$



(d) Vitesse avec $w_j = 1$

FIGURE 2.4 – Trajectoire et vitesse pour un poids $w_j = 0$ et $w_j = 1$

Hidra souligne les faiblesses de sa pipeline lors de vols avec un nombre important d'obstacles dynamiques. Bien qu'utilisant Φ^* [22], une variante de A^* spécialisée dans la replanification *any-angle*, il reste plus optimal de réagir uniquement avec le planner local.

2.2.6 Vol en formation, définition et spécificité

Le vol en formation est un sujet à part entière. L'objectif de Drone Géofencing est de permettre la constitution d'une formation en vol et le maintien de celle-ci lors de déplacements dynamiques. L'évitement d'obstacles mobiles est crucial pour garantir la cohésion de la formation dans un environnement réel. Les trois aspects du vol identifiés par Liu [27] sont ainsi nécessaires. Il est possible d'ajouter une exigence supplémentaire : le vol en formation d'escadrille. Chaque drone appartient à une escadrille distincte ; il importe donc de gérer à la fois la formation interne à chaque escadrille et la formation entre escadrilles.

Liu [27] propose trois approches existantes pour permettre le vol en formation :

1. Modèle maître-esclave [9] : le drone maître planifie et suit la trajectoire, tandis que les drones esclaves cherchent à minimiser l'écart de distance et d'angle par rapport au maître.

2. Maintien de la formation (VS) [39] : plus robuste que le modèle maître-esclave, cette approche consiste à minimiser la distance entre la position réelle de chaque drone et sa position désirée au sein de la formation.
3. Modèle basé sur le comportement [4] : inspiré du modèle des boids (vols d'oiseaux), des forces répulsives et attractives agissent sur les drones (répulsion s'ils sont trop proches, attraction vers le centre de gravité local des voisins, alignement des vitesses).

Les algorithmes de VS ne permettent pas une déformation flexible de la formation, les rendant inadaptés aux besoins.

Afin de permettre un vol en formation dynamique, nous introduirons un coût E_{form} de distorsion par rapport à la formation initiale, à l'image de la solution proposée par Cao pour le vol en formation avec DWA [9].

2.3 Conclusion de l'état de l'art

S'il n'est pas évident qu'elles sont méthodes de traitement d'obstacles et de calcul de chemin les plus performantes dans notre cas d'étude, il est néanmoins évident que s'appuyer uniquement sur un planificateur global cinétique n'est pas adapté au temps réel embarqué. Le défi consiste à implémenter les différentes méthodes et approches, puis à les tester dans un environnement contrôlé, le plus proche possible de la réalité. Toutes les méthodes existantes n'ont pas pu être codées ni testées, ni même traitées dans cet état de l'art ; nous avons donc décidé de nous concentrer uniquement sur les méthodes les plus prometteuses pour trouver une trajectoire dans l'ordre de la seconde.

Parmi les algorithmes de calcul de chemin les plus prometteurs figurent :
— les méthodes par génération aléatoire de points (RRT* et BI-RRT*) ;
— les méthodes fondées sur la création d'un arbre de recherche (pseudo-3D A^*, pseudo-3D Lazyθ^*, pseudo-3D BI-A^*, pseudo-3D BI-Lazyθ^*).

Nous supposons que les algorithmes de génération aléatoire de points permettent des temps de calculs inférieurs pour des distances de trajectoires supérieures à celles des méthodes basées sur un arbre de recherche. Nos tests ont pour objectif de définir la relation coût/bénéfice de chaque algorithme de recherche de chemin et de chaque méthode de traitement de l'environnement (voxel/octree).

Nous avons essayé de développer une solution ultra-simple pour le calcul de trajectoire statique. Dans un premier temps, nous avons implémenté successivement une approche basée sur les courbes de Dubins, puis sur les cycloïdes, puis sur les hypocycloïdes, sans succès. Nous avons toutefois réussi à implémenter une solution dynamique, celle proposée par Hidra [22], et l'avons modifiée pour prendre en considération toutes les caractéristiques physiques des drones.

Chapitre 3

Expérimentation

3.1 Méthodologie

Dans un premier temps, nous étudierons les performances d'un A^* classique et de la version pseudo-3D proposée dans section section 2.2.4.

Dans un second temps, nous testerons les différents algorithmes de LOS, suivi de l'étude de la performance des algorithmes de recherche de chemin. Pour tous les tests de LOS et de performance de chemin, nous utiliserons la même procédure et le même code de génération de l'environnement : celui-ci comprend une géocage, des obstacles et des points de passage. Les obstacles sont conçus pour être relativement proches des zones de restriction de vol réelles. Pour chaque test, qu'il s'agisse de l'algorithme LOS le plus performant ou du meilleur algorithme de calcul de chemin, nous avons mis en place une méthode de test rigoureuse permettant l'étude des cas extrêmes et garantissant la fiabilité des résultats. Stocker la graine aléatoire de chaque test permet une vérification *a posteriori* des cas extrêmes, rapide et fiable. Cette vérification est essentielle pour identifier précisément l'origine d'un problème. Si elle révèle une faiblesse du code, il est possible et simple de relancer les tests uniquement pour l'algorithme concerné. De même, lorsqu'une nouvelle approche doit être évaluée, il suffit de l'ajouter aux résultats existants sans réexécuter l'intégralité de la procédure.

Tous les tests sont effectués sur LENOVO, intel CORE i7 avec 24 Go de RAM.

3.1.1 Test A^* vs pseudo-3D A^*

Pour comparer les deux algorithmes, nous avons étudié trois cas réels, implémentés dans un monde voxelisé. Nous mesurerons la distance parcourue (d) par le chemin trouvé, la durée de calcul (t) ainsi que la taille de l'arbre de recherche (S_{tree}). Le premier cas de test est composé de trois obstacles similaires à des murs infranchissables par le haut ou par le bas, forçant un contournement horizontal. Le deuxième environnement de test est composé de nombreux petits obstacles, également infranchissables par le haut ou par le bas, mais nécessitant un contournement moins important. Le troisième environnement est constitué d'un seul obstacle très imposant, infranchissable par le haut ou par le bas, il impose un contournement conséquent et agit comme un piège pour les algorithmes de type A^*, qui doivent générer de nombreux points horizontalement avant de trouver une sortie.

3.1.2 Test LOS

Afin de déterminer quel algorithme LOS est le plus efficace, celui de [35] ou une méthode *slab* plus classique [5], il a été nécessaire de réaliser des tests à grande échelle. Pour couvrir un maximum de situations, les approches ont été évaluées sur un environnement dense (50 obstacles), selon différentes profondeurs d'octree (depth $\in \{5, 6, 7, 8, 9\}$). Cent points ont été générés pour 5000 paires ((x_i, x_j) avec $x_i, x_j \in P$ et $x_i \neq x_j$) de tests LOS distincts, tirées au hasard. Pour isoler l'impact de la profondeur de l'octree et du nombre d'obstacles, seules cinq générations distinctes de l'environnement ont été effectuées, une pour chaque valeur de n_{obs}. Sur chacune de ces générations, les profondeurs d'octree ont été testées successivement : on ne régénère pas un environnement neuf pour chaque combinaison, mais on fait varier la profondeur sur les mêmes environnements. Cinq métriques sont enregistrées :

— Le nombre de nœuds explorés par chaque algorithme LOS – N_{los}.
— Le temps de calcul de chaque LOS – t_{los}.
— *depth*.
— Seed aléatoire - permettant la vérification post test s
— L'algorithme utilisé – {simple, revelles}

Seule la distribution des temps de calcul t_{los} en fonction de la profondeur et de l'algorithme utilisé sera étudiée. La comparaison des distributions selon l'algorithme utilisé sera réalisée à l'aide du test de Kolmogorov–Smirnov (KS). En cas de rejet de l'hypothèse nulle, nous appliquerons ensuite le test de Mann–Whitney U (MWU), moins sensible (il rejette rarement, alors que le KS rejette très souvent).

Si les données comportent des valeurs extrêmes trop importantes, nous les étudierons via une loi de Pareto généralisée (GPD). La GPD offre une modélisation paramétrique des distributions à queue lourde : trois paramètres sont estimés :

— la valeur-seuil $\hat{\mu}$,
— le paramètre d'échelle $\hat{\sigma}$, qui mesure l'amplitude des dépassements au-delà du seuil,
— le paramètre de forme $\hat{\mathcal{E}}$, qui contrôle la «lourdeur» de la queue (vitesse de décroissance) [12].

Cette modélisation permet d'étudier les valeurs dites «extrêmes», leur fréquence et leur importance, et ce pour chaque distribution.

3.1.3 Test d'algorithme de chemin

La méthodologie développée va permettre de tester rigoureusement les différents algorithmes de recherche de chemin brut implémentés. Deux types d'environnements ont été définis. Dans le premier, chaque génération aléatoire instancie 20 «petits» obstacles, dans le second, 5 «grands» obstacles (chacun ayant un volume moyen neuf fois supérieur à celui des petits). Chaque génération affecte de manière aléatoire la configuration des obstacles et le positionnement des points. Cinquante générations distinctes ont été réalisées, chacune comprenant 11 points (soit 10 trajectoires). La seed aléatoire propre à chaque génération d'environnement est la même pour tous les algorithmes testés. Parmi ces algorithmes, on distingue deux approches : une représentation octree et une représentation voxel. Pour chaque algorithme de chaque environnement, il y a donc 100 environnements distincts avec 10 chemins à trouver par environnement.

Les données stockées sont :

— n_{obs}
— Temps de calcul de la construction de l'environnement — t_{env}
— Temps de calcul de l'algorithme — t_{algo}
— Distance de la trajectoire retournée — d
— Nombre d'échecs par génération (max. 10) — $N_{failure}$
— Nombre de nœuds de l'octree / nombre de voxels — N_{env}
— Nombre de points explorés par l'algorithme — N_{algo}
— Seed — s
— La profondeur *depth*

Les algorithmes testés sont : pseudo-3D-A^*, pseudo-3D-Lazy-θ^*, RRT, BI-RRT, pseudo-3D-BI-A^*, pseudo-3D-BI-Lazy-θ^*

3.1.4 Test du local motion planner

Pour le planificateur local, nous n'évaluerons que la durée moyenne de calcul pour chaque trajectoire, avec une contrainte de temps T non fixe (voir section 2.2.5) : le drone suit la trajectoire aussi rapidement que possible. Notre algorithme génère un polynôme entre chaque point, puis crée des points de passage toutes les $0.5s$. Imposer une contrainte de temps T trop importante fait augmenter linéairement le temps de calcul. Nous sélectionnerons les trajectoires et environnements produits par l'algorithme de recherche de chemin le plus performant, puis mesurerons les performances de notre planificateur local en fonction des caractéristiques du drone Tundra 2 d'Hexadrone [21] :

— Vitesse maximale horizontale : $v_{max,horizontal} = 35\text{m/s}$;
— une vitesse maximale ascendante : $v_{max,up} = 7\text{m/s}$;
— une vitesse maximale descendante : $v_{max,down} = 5\text{m/s}$;
— la vitesse initiale et d'arrivée : $v_{init} = 0\text{m/s}, v_{goal} = 0\text{m/s}$;
— l'accélération initiale et d'arrivée : $a_{init} = 0\text{m/s}, a_{start} = 0\text{m/s}$
— le *jerk* maximal, nous avons fixée la variation maximal du (*jerk*) à 2 : $jerk_i \approx \frac{\|\mathbf{a}_{i+1} - \mathbf{a}_i\|}{\Delta t} \leq 2$

Nous mesurerons le nombre d'échecs du planner $n_{failure}$ et la raison de l'échec $r_{failure}$, ainsi que le temps de calcul $t_{planner}$ nécessaire pour trouver la trajectoire associée à chaque chemin.

3.2 Résultats

3.2.1 Résultats A^* vs Pseudo-3D A^*

Sur les trois environnements de test, la méthode pseudo-3D A^* surpasse un A^* classique. Le tableau 3.1 présente la différence en pourcentage de pseudo-3D A^* à A^* des trois métriques mesurées dans section 3.1.1,

sur chacun des trois environnements. La nouvelle méthode proposée trouve un chemin de taille (d) similaire, ou très proche de celui trouvé par A^*. Pseudo-3D A^* a une durée de calcul (t) inférieure à A^* d'un facteur de 3,3 à 15 (voir tableau3.1). Ce gain important provient de la différence majeure du nombre de points visités par les deux méthodes (voir section 3.1.1). Comme annoncé dans la section 2.2.4, le temps de calcul d'une méthode à base d'arbre de recherche (par exemple : A^*) est, dans le pire des cas, en $O(N \log N)$, où N est le nombre de points de l'arbre. Pour le premier environnement, A^* construit un arbre (S_{tree}) de 1286 points, alors que notre méthode n'explore que 89 points, soit un arbre 13 fois plus faible. Similairement, sur le second et le troisième environnement A^* construit deux arbres de respectivement 88 et 77 points pour deux arbres de 15 et 14 points pour notre méthode ($\Delta S_{tree} = -486\%$, $\Delta S_{tree} = -450\%$). Pseudo-3D A^* explore beaucoup moins de points que la méthode classique, entraînant une diminution radicale du temps de calcul.

Cette méthode présente toutefois une faiblesse. L'altitude ne pouvant se modifier uniquement lorsque l'arbre rencontre un obstacle, il est possible que la solution ne soit jamais trouvée. Si le point d'arrivée est hors de vue de tous les points atteignables depuis l'altitude du point de départ et qu'il n'y a pas d'obstacles intersectant cette altitude, alors l'arbre de recherche ne s'étendra jamais à de nouvelles altitudes.

Une solution consiste à implémenter une approche bidirectionnelle / Connect de pseudo-3D A^*. La construction de deux arbres simultanés devrait résoudre le problème.

Environnement	$\Delta\ d$	$\Delta\ S_{tree}$	$\Delta\ t$
1	+2,39%	−1344,94%	−1519,33%
2	0,00%	−486,67%	−424,22%
3	0,00%	−450,00%	−336,71%

TABLE 3.1 – Performance : $\frac{\text{pseudo-3D}A^* - A^*}{A^*} * 100$

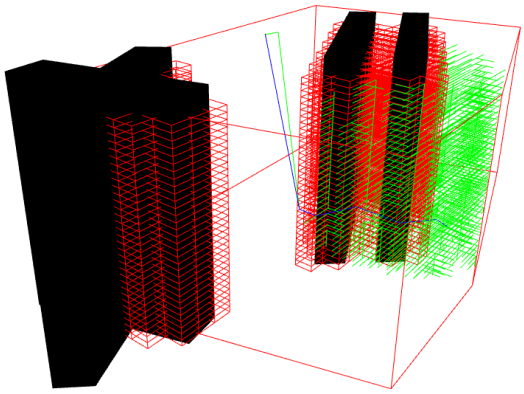


FIGURE 3.1 – A^* classique

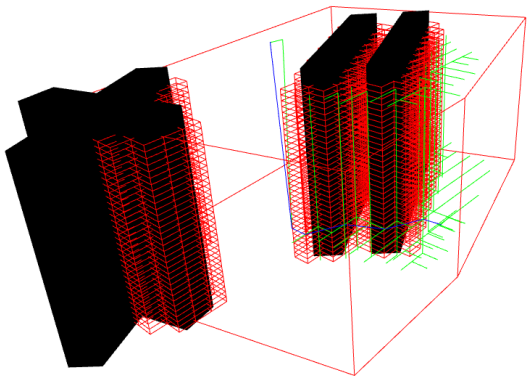


FIGURE 3.2 – pseudo-3D A^*

FIGURE 3.3 – Premier environnement de test, avec en bleu le chemin trouvé, en vert l'arbre de recherche, en noir les obstacles, en rouge les voxels occupés. L'altitude à été augmenté proportionnellement à la taille horizontal de la géocage.

3.2.2 Résultats LOS

Les distributions des temps de calcul (t_{los}) selon l'algorithme se sont révélées significativement différentes à tous les niveaux de profondeur, d'après le test de Kolmogorov–Smirnov (KS) et le test de Mann–Whitney U (MWU) (voir tableau 3.2). Le rejet de l'hypothèse d'égalité des distributions par le MWU confirme une divergence marquée (p-value«0,001 pour chaque profondeur, les distributions sont dissimilaires dans bien plus de 99,9 % des cas). La méthode de Revelles [35] présente des moyennes de performance de $0.33\,ms$, $0.37\,ms$, $0.43\,ms$ et $0.57\,ms$, alors que la méthode simple présente des moyennes de $0.36\,ms$, $0.41\,ms$, $0.49\,ms$ et $0.63\,ms$ (voir figure 3.4). Les distributions sont donc significativement différentes, avec une moyenne plus faible pour la méthode de Revelles que pour la méthode simple. Les différences restent de l'ordre de la microseconde, et n'ont en réalité aucun impact sur le temps de calcul d'un algorithme de recherche de chemin.

max_depth	KS_pvalue	MWU_pvalue
5	3.182558×10^{-37}	2.910208×10^{-18}
6	8.078142×10^{-49}	5.747122×10^{-20}
7	2.643381×10^{-54}	1.296388×10^{-22}
8	7.436639×10^{-52}	6.390766×10^{-22}

TABLE 3.2 – Tableau des valeurs de p-value pour différentes profondeurs maximales

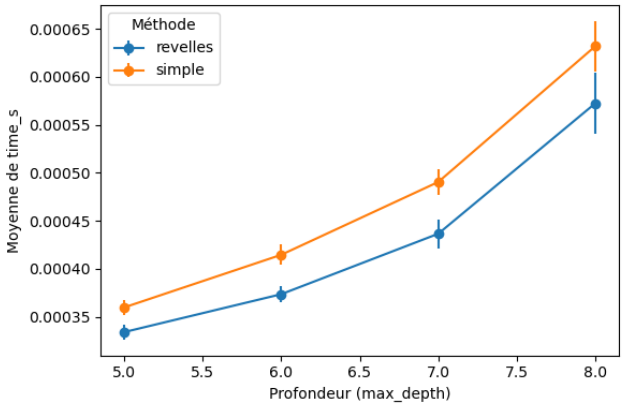


FIGURE 3.4 – Performance moyennes pour chaque valeur de *depth*

Parmi toutes les distributions de t_{los}, des valeurs extrêmes ont été observées, atteignant jusqu'à 1000 fois la moyenne. Nous avons évalué les paramètres de la loi de Pareto généralisée (GPD). Les résultats de la GPD montrent que la valeur-seuil $\hat{\mu}$ est très proche pour les deux algorithmes à chaque profondeur. En revanche, la croissance du paramètre d'échelle $\hat{\sigma}$ est plus marquée pour la méthode classique, ce qui traduit une fréquence plus élevée de dépassements importants à grande profondeur. La méthode de Revelles, dont la progression de $\hat{\sigma}$ est plus modérée, génère ainsi moins de cas extrêmes, confirmant sa meilleure performance dans les queues de distribution. À noter toutefois que le paramètre de forme $\hat{\mathcal{E}}$ croît plus rapidement pour la méthode de Revelles (et finit par dépasser celui de la méthode classique aux plus grandes profondeurs), signe d'une queue plus lourde relative au seuil (voir figure 3.5).

C'est l'algorithme de Revelles [35] qui sera utilisé pour calculer la LOS des différents algorithmes de calcul de chemin.

3.2.3 Résultats de la recherche de chemins

• **Construction de l'environnement** : La figure 3.6 montre clairement un coût de construction des 2 types d'environnements exponentiel, comme énoncé dans section 2.2.2.1. Cependant le temps de construction

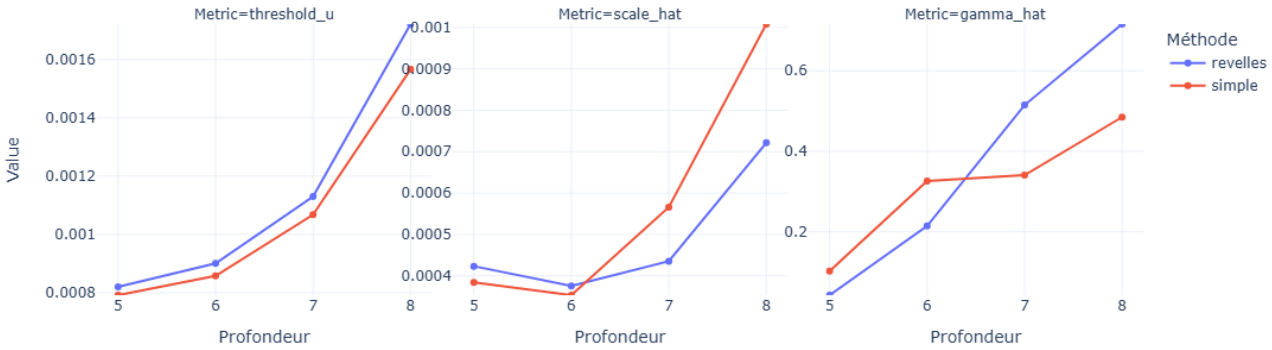


FIGURE 3.5 – Valeur des paramètres par méthode et par *depth*

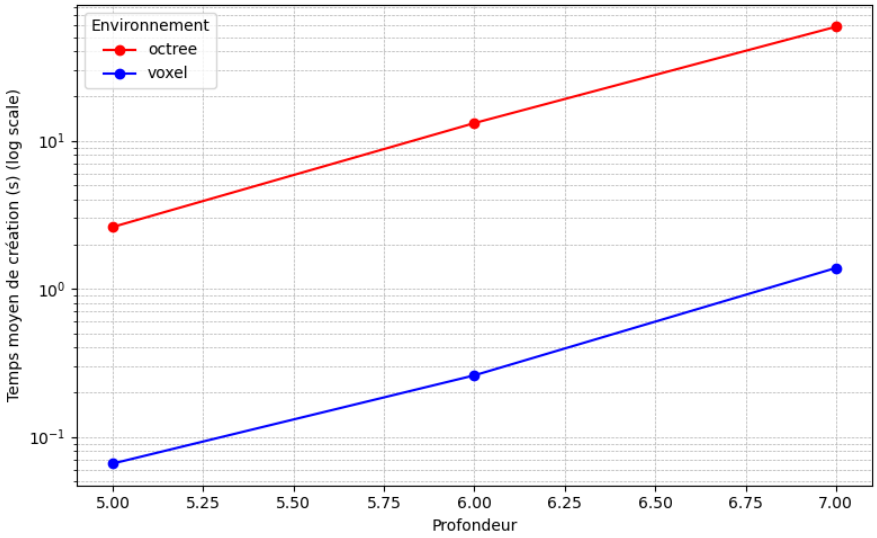


FIGURE 3.6 – Temps de constructions de l'environnement par niveau de *depth*

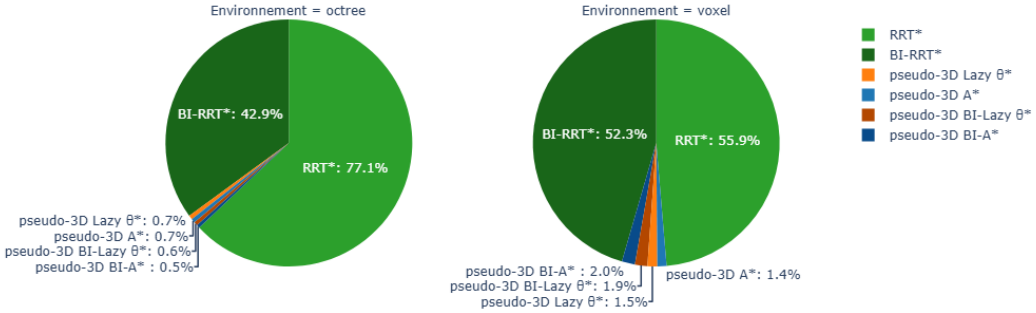


FIGURE 3.7 – Taux d'échèque par algorithme par environnement

de voxels est bien inférieur au temps de construction d'un octree (par un facteur proche de 40).

• **Calcul de chemin** : Comme nous l'avons rappelé dans la section 2.2.4, les algorithmes reposant sur une génération aléatoire de points, RRT* et BI-RRT*, ne peuvent en aucun cas garantir de trouver un chemin si celui-ci existe. En effet, ces algorithmes génèrent des points jusqu'à atteindre un nombre donné ; si aucune trajectoire n'est trouvée à l'issue de cette génération, ils s'arrêtent.

Dans nos tests, nous avons limité à 400 le nombre de points générés par RRT* et BI-RRT*. Cette limite arbitraire a conduit à un taux d'échec bien supérieur aux prévisions : RRT* atteint 77.1 % d'échecs sur un octree, toutes profondeurs confondues (voir figure 3.7 et tableau 3.3). Surprenamment, les versions pseudo-3D de A^* et de Lazyθ^* présentent un taux d'échec global inférieur de respectivement 0.6% et 0.4% à celui de leurs versions bidirectionnelles dans un environnement voxel. À partir de la profondeur 7, seul RRT* et BI-RRT* échouent toujours à trouver des chemins. La durée moyenne, pour les algorithmes autres que RRT* et BI-RRT*, en cas d'échec est de 10 secondes.

Pour l'analyse des temps de calcul moyens, nous excluons les chemins «directs», c'est-à-dire ceux où le point initial est directement relié au point final. En effet, tous nos algorithmes partagent le même code d'initialisation : en début d'exécution, si la LOS (ligne de vue) entre le point de départ et le point d'arrivée est dégagée, le code n'a pas besoin de s'exécuter ; le chemin se limite alors à départ–arrivée.

Inclure ces chemins directs introduit du bruit dans nos données et fait baisser artificiellement les moyennes et les variances, masquant ainsi toute différence pertinente entre les performances des algorithmes.

Les temps de calcul moyens des algorithmes augmentent avec la profondeur de l'environnement : une profondeur supérieure induit des pas de déplacement plus petits, ce qui nécessite l'exploration d'un plus grand nombre de points pour trouver un chemin. Par exemple, le pseudo-3D BI-A^* sur voxel nécessite près de vingt fois plus de temps entre les profondeurs 5 et 7 (voir figure 3.8 et tableau 3.3).

Tous les algorithmes sont nettement plus performants sur voxel que sur octree, à l'exception du pseudo-3D Lazy θ^*, dont le coût devient extrêmement élevé en profondeur 7 sur voxel (près de 50 % de coût supplémentaire par rapport à l'octree). Bien que le pseudo-3D A^* sur voxel surpasse sa version bidirectionnelle ainsi que le

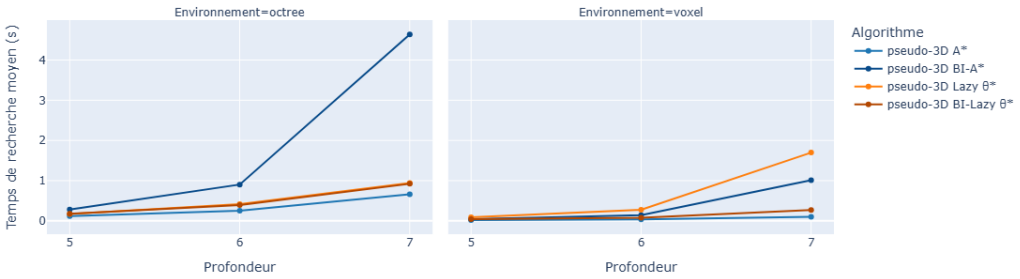


FIGURE 3.8 – Durée moyenne par algorithme et par profondeur

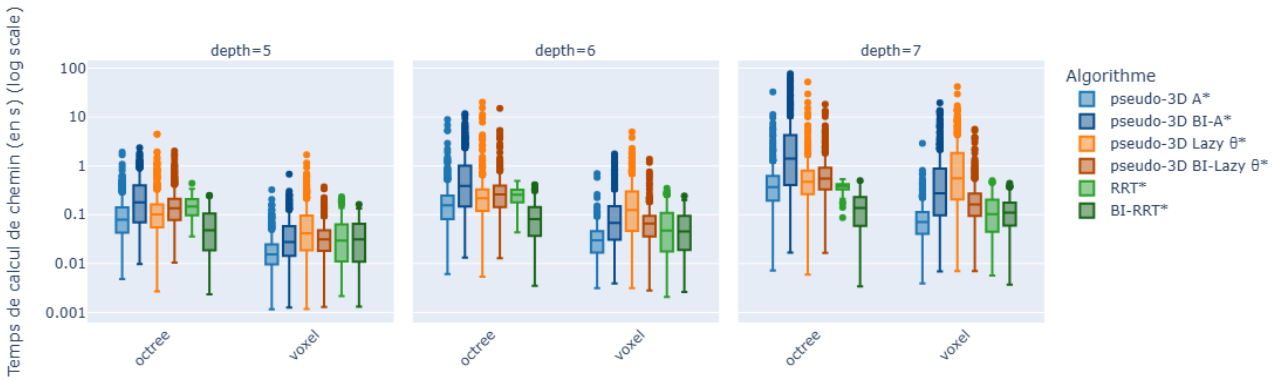


FIGURE 3.9 – Temps de calcul par environnement et pour chaque algorithme

pseudo-3D Lazy θ^*, ses durées moyennes (figure 3.8) et ses distributions (figure 3.9) restent comparables à celles du pseudo-3D BI-Lazy θ^*. Autrement dit, pseudo-3D BI-Lazy θ^* est tout aussi performant en temps de calcul que pseudo-3D A^*. Sur voxel en profondeur 7, pseudo-3D A^* a une moyenne de 0.099150 seconde, pseudo-3D BI-Lazy θ^* a une moyenne de 0.269869 (voir tableau 3.3).

Par souci de lisibilité, RRT* et BI-RRT* ne sont pas affichés : leur nombre de points générés étant constant, leur durée moyenne est quasi constante.

La distance parcourue par les algorithmes est mesurée en proportion de la distance euclidienne directe entre le point de départ et le point d'arrivée : les solutions «directes» ont donc une distance de 1,0. Cette distance décroît avec la finesse du maillage ; un environnement plus profond permet un contournement plus court des obstacles (voir figures 3.10 et 3.11). Les distances générées par RRT* et BI-RRT* sont nettement supérieures à celles des méthodes basées sur la construction d'arbres de recherche. En revanche, pour ces méthodes par arbre, les distances moyennes restent très similaires, quel que soit le type d'environnement (octree ou voxel) et la profondeur ; voir tableau 3.3. Les quatre algorithmes présentent une distance moyenne inférieure à 1,01 fois la distance euclidienne directe. Il n'y a pas de différence significative entre les distances des chemins générés par les différents algorithmes de création d'arbres de recherche. Toutes les distances parcourues restent faibles, aucune n'est excessivement élevée. Les environnements où un contournement important a été observé pour les quatre algorithmes ont été vérifiés manuellement, afin de s'assurer que ce contournement est bien dû à la configuration de l'environnement, et non à une faiblesse propre aux algorithmes.

3.2.4 Résultats du local motion planner

Suite aux résultats des tests de calcul de chemin, nous avons décidé d'évaluer les performances du planificateur local de mouvements dans un environnement voxel de profondeur 7, en utilisant un chemin global calculé par l'algorithme pseudo-3D A^*.

Pour les 100 environnements distincts (50 de chacun des deux types), avec 10 trajectoires par environnement (voir section 3.1.3), aucune trajectoire n'a échoué. Autrement dit, aucune primitive n'a généré de collision avec les voxels occupés ni avec la géocage : dès lors qu'un chemin global existe, le planificateur local trouve une trajectoire valide et respecte toutes les contraintes physiques (voir section 3.1.4).

Deux mesures de temps sont à prendre en compte : la durée moyenne de calcul de la meilleure primitive entre deux points, et celle nécessaire à la génération des points selon la primitive (un polynôme quintique) trouvée.

La durée moyenne de calcul de la meilleure primitive de mouvement est de 0.1 ms, avec une borne supérieure

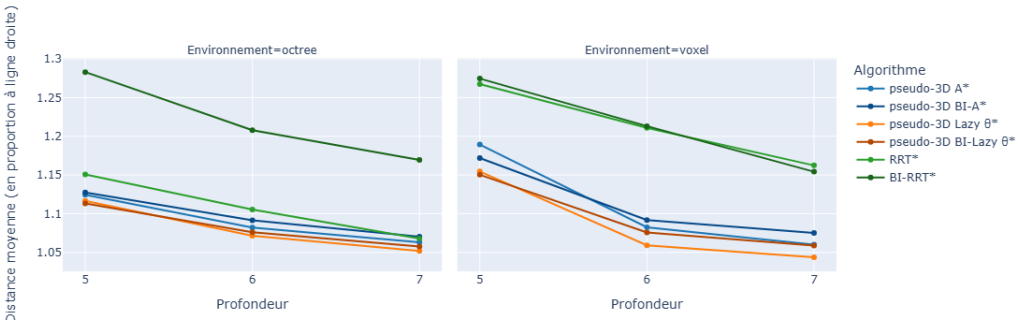


FIGURE 3.10 – Distance moyenne par algorithme et par profondeur

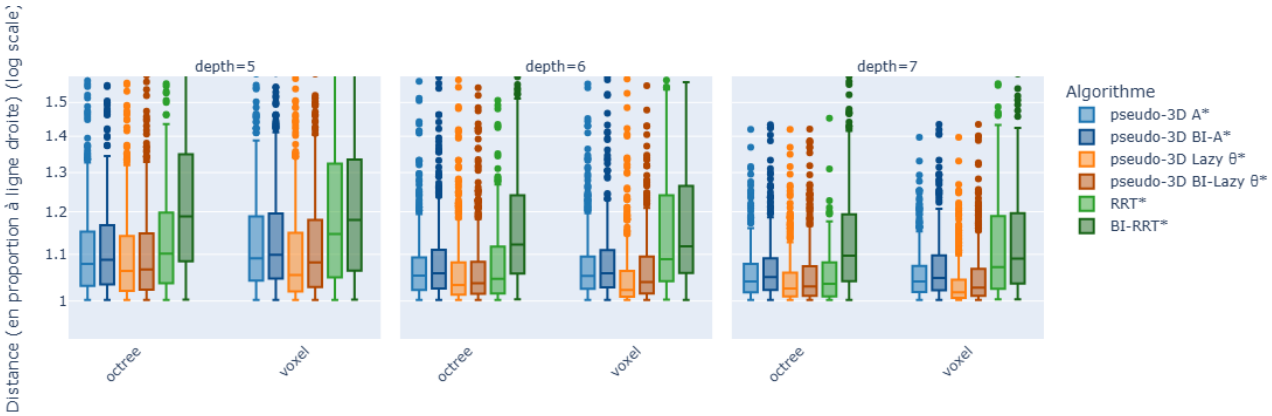


FIGURE 3.11 – Distance de la trajectoire par environnement et pour chaque algorithme

de l'intervalle de confiance à 99 % de 0.2 ms (soit 0.1 ms + 0.1 ms). Autrement dit, dans 99 % des cas, cette étape est effectuée en moins de 0.2 ms (voir figure 3.12), ce qui la rend négligeable dans le pipeline global.

La génération des points le long de la trajectoire, basée sur un polynôme de degré cinq, prend en moyenne 0.092 s, avec une borne supérieure à 99 % de 0.189 s (soit 0.092 s+0.096 s). Ainsi, dans 99 % des cas, la trajectoire complète est générée en moins de 0.189 s (voir figure 3.12), ce qui reste compatible avec les contraintes temps réel.

La figure 3.13 illustre un exemple de trajectoire générée par notre planificateur bi-étape : un chemin global pseudo-3D calculé par l'algorithme A^*, suivi d'un lissage local par primitives de mouvement sur chaque sous-chemin.

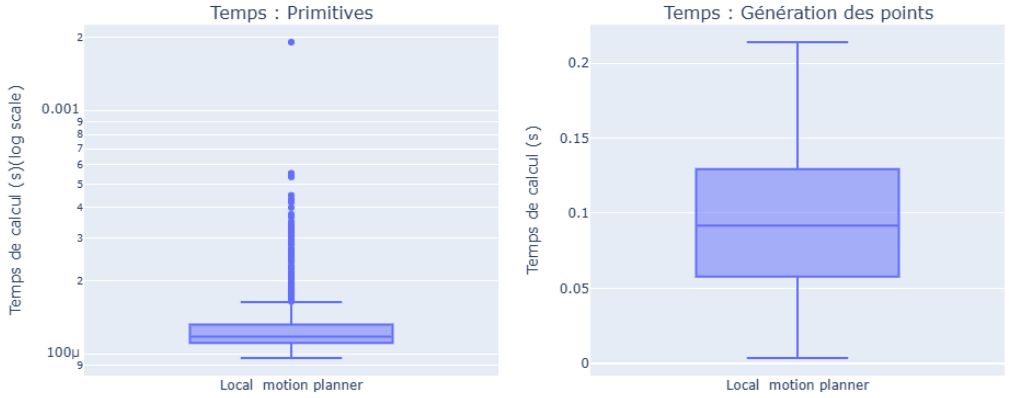
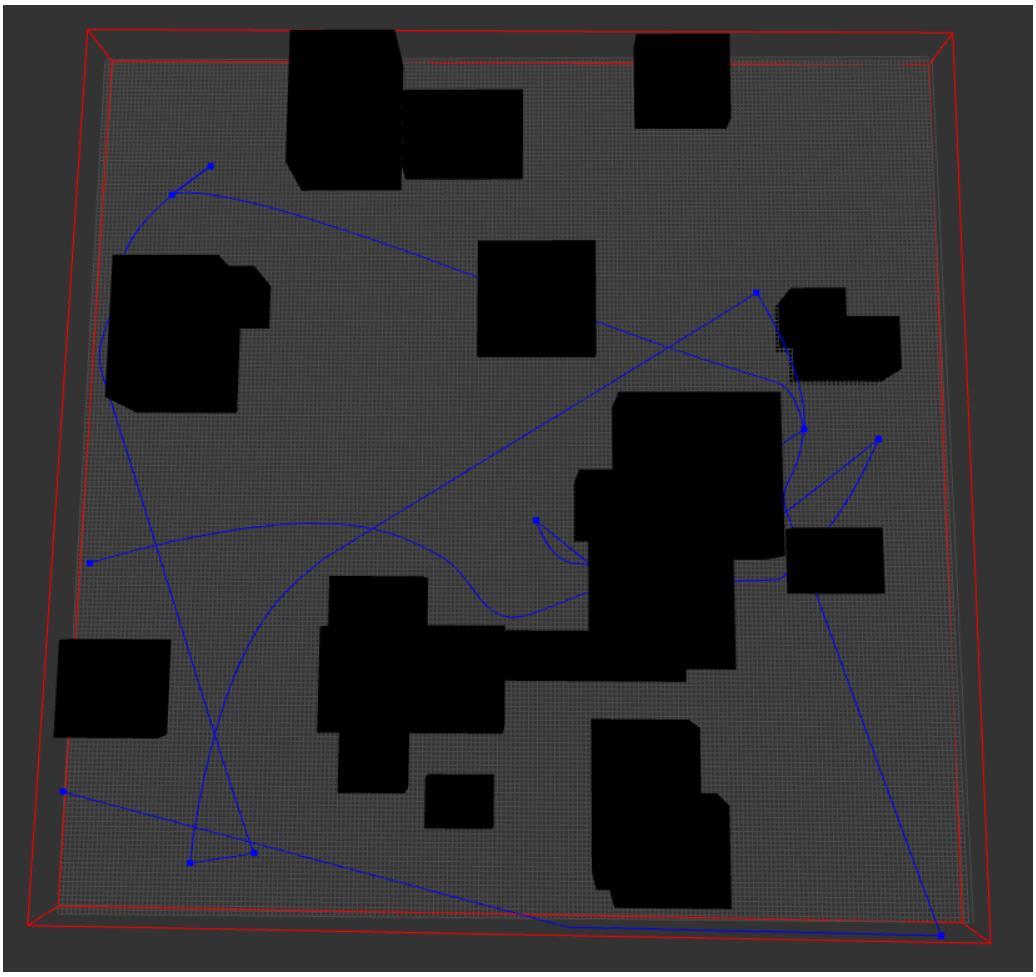


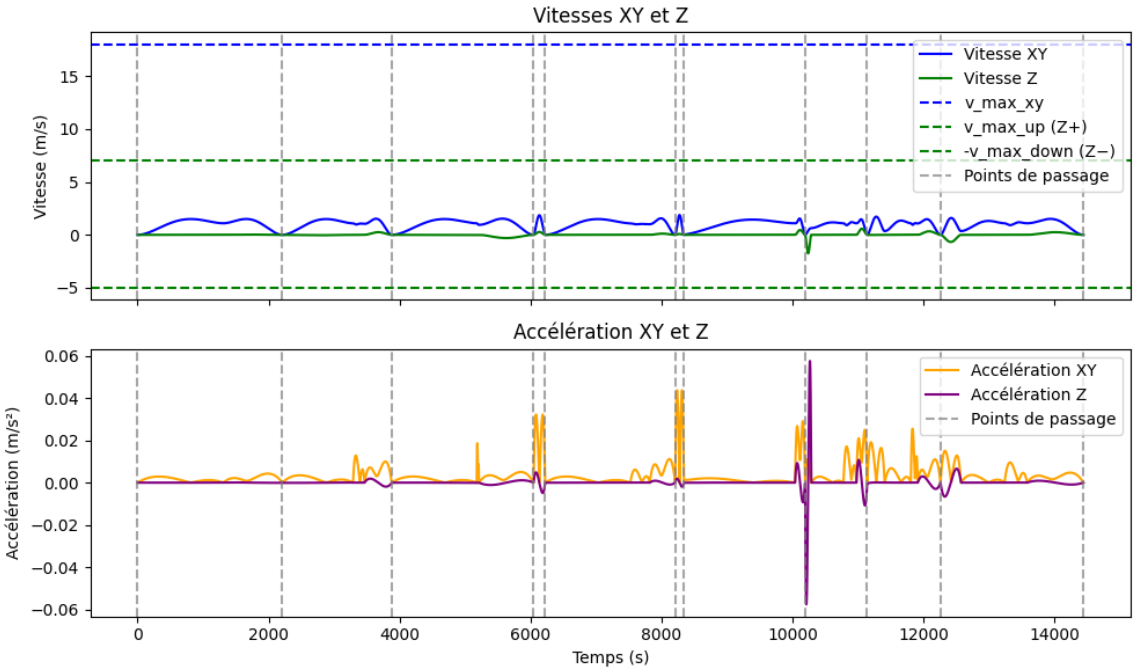
FIGURE 3.12 – Box plot du temps de calcul d'une trajectoire du local motion planner

3.3 Conclusion des tests

Bien qu'ayant optimisé au maximum le temps de construction de l'octree (voir section 2.2.2), la création d'un environnement voxel reste près de 40 fois plus rapide. Après une analyse a posteriori, nous avons constaté



(a) Trajectoire d'un Tundra 2 : en bleu la trajectoire, en noir les obstalces, en rouge la géocage



(b) Profils de vitesses et d'accélérations du Tundra 2 pour la trajectoire figure : 3.13a

FIGURE 3.13 – Trajectoire (figure 3.13a) et profils de vitesses/accélérations associés (figure 3.13b) d'un Tundra 2 avec vitesse et une accélération d'arriver à chaque point fixée à 0. Seed aléatoire : 513, 20 obstacles de type "petit", sur Voxel avec un chemin calculé par pseudo-3D A^*

qu'un bug dans le degré de précision était en partie responsable de la diminution du temps de création des voxels (voir figure 3.6). En effet, les octrees sont, dans le pire des cas, 4 fois plus précis que les voxels (soit moins d'un niveau de profondeur de différence). Même en tenant compte de cette différence, la création de voxels resterait de l'ordre de 10 fois plus rapide que celle d'un octree.

La durée moyenne des chemins trouvés penche également fortement en faveur des environnements sur voxels, malgré le travail d'optimisation de la LOS effectué sur octree (voir section 3.1.2 et section 3.2.2). Une solution théorisée consistait à n'utiliser qu'un quadtree (version 2D de l'octree) (voir figure 2.2) : nos obstacles étant simples en 3D (cantonnés entre deux bornes), il suffirait de vérifier la validité horizontale avec le quadtree avant de vérifier la validité de l'altitude par une simple comparaison aux deux bornes. Cependant, les calculs de LOS ne seraient plus aussi évidents qu'avec un octree. Faute de temps, nous n'avons pas pu approfondir cette idée.

Si la taille des chemins des algorithmes à base d'arbre de recherche reste très similaire, la durée de calcul est bien différente. Il est clair que les algorithmes pseudo-3D A^* et pseudo-3D BI-Lazyθ^* présentent des temps de calcul bien inférieurs à ceux des autres méthodes (ordre de la dixième de secondes contre ordre de la seconde) (voir section 3.2.3). Nous avons donc décidé de déployer la version la plus simple des deux algorithmes concurrents : pseudo-3D A^* avec comme traitement de l'environnement la voxélisation.

Les algorithmes à génération aléatoire (RRT* / BI-RRT*) ne sont pas du tout adaptés à un environnement 3D complexe. Bien que prometteurs sur le temps de génération des 400 points alloués, il est impossible de déterminer quand interrompre l'algorithme, rendant son implémentation en temps réel irréaliste.

Les performances de notre planificateur local basé sur les primitives de mouvement se révèlent extrêmement satisfaisantes pour la génération des primitives elles-mêmes, avec un temps de calcul moyen de seulement 0.1 ms pour trouver la meilleure primitive entre deux points (voir section 3.2.4). Cette performance est d'autant plus remarquable qu'aucun échec n'a été observé : dès lors que les contraintes physiques sont respectées, l'algorithme parvient systématiquement à générer une trajectoire valide.

En revanche, la génération des points le long de cette primitive s'avère bien plus coûteuse, avec un temps moyen de 0.1 s (voir section 3.2.4). En générant un point toutes les 0.5 s de trajectoire prévue, le nombre total de points devient rapidement très élevé sur les longues distances.

Au total, pour une trajectoire non directe dans un environnement voxelisé à une profondeur de 7, le temps moyen nécessaire pour trouver une trajectoire entre deux points est de 0.2 s, tandis que la construction de l'environnement prend en moyenne 1.2 s.

Profondeur	Environnement	Algorithme	Temps moyen (s)	Distance moyenne	Taux d'échec (%)
5	octree	BI-RRT*	0.117150	1.282933	46.55
		RRT*	0.219109	1.150815	63.67
		pseudo-3D A^*	0.188057	1.124318	1.04
		pseudo-3D BI-A^*	0.282405	1.127519	0.87
		pseudo-3D BI-Lazyθ^*	0.176315	1.113251	0.87
		pseudo-3D Lazyθ^*	0.310049	1.116611	1.05
	voxel	BI-RRT*	0.073776	1.274500	54.99
		RRT*	0.121569	1.267364	62.31
		pseudo-3D A^*	0.025984	1.189240	3.23
		pseudo-3D BI-A^*	0.047404	1.171881	4.61
		pseudo-3D BI-Lazyθ^*	0.044946	1.150258	4.27
		pseudo-3D Lazyθ^*	0.097871	1.154538	3.29
6	octree	BI-RRT*	0.145915	1.207919	36.49
		RRT*	0.311813	1.105361	71.79
		pseudo-3D A^*	0.311032	1.081968	0.86
		pseudo-3D BI-A^*	0.938602	1.091418	0.68
		pseudo-3D BI-Lazyθ^*	0.465439	1.075913	0.86
		pseudo-3D Lazyθ^*	0.595708	1.071538	0.87
	voxel	BI-RRT*	0.096755	1.213096	48.29
		RRT*	0.150880	1.210921	50.69
		pseudo-3D A^*	0.037144	1.082468	0.69
		pseudo-3D BI-A^*	0.147158	1.091763	1.03
		pseudo-3D BI-Lazyθ^*	0.089902	1.075624	1.05
		pseudo-3D Lazyθ^*	0.276422	1.059166	0.74
7	octree	BI-RRT*	0.226028	1.169476	**46.11**
		RRT*	0.401925	1.067760	**92.14**
		pseudo-3D A^*	0.658714	1.063156	0.00
		pseudo-3D BI-A^*	4.638813	1.070397	0.00
		pseudo-3D BI-Lazyθ^*	0.920793	1.057641	0.00
		pseudo-3D Lazyθ^*	0.941383	1.051952	0.00
	voxel	BI-RRT*	0.164348	1.154181	53.68
		RRT*	0.233057	1.162354	54.44
		pseudo-3D A^*	**0.099150**	**1.060085**	0.00
		pseudo-3D BI-A^*	1.013492	1.075048	0.00
		pseudo-3D BI-Lazyθ^*	**0.269869**	**1.058898**	0.00
		pseudo-3D Lazyθ^*	1.698125	1.043803	0.00

TABLE 3.3 – Résultats moyens par profondeur, environnement et algorithme

Chapitre 4

Conclusion & ouverture

4.1 Conclusion du travail effectué

Durant cette alternance, nous avons cherché et développé une solution répondant aux besoins de l'entreprise Drone Géofencing. Nous avons implémenté une nouvelle version d'un planificateur bi-phase de trajectoire dynamique, spécialement conçue pour répondre aux exigences du cahier des charges de l'entreprise. Cet algorithme s'appuie sur une architecture plus performante que celles décrites dans la littérature, en tirant parti de variantes optimisées d'algorithmes couramment utilisés. Plus précisément, nous avons développé une version améliorée de A^*, un pseudo-3D A^* finement ajusté, et révisé la génération de primitives de Hidra [22] afin de satisfaire les critères stricts de Drone Géofencing. Grâce à ces innovations, notre solution offre des gains de performance et de robustesse significatifs. Le code implémenté et intégré à Mana**DG** (le backend de l'entreprise) permet de trouver une trajectoire praticable par un drone en moins d'une seconde, et de transmettre la liste des `WayPointAction` à Station**Drone**. Seule la voxélisation peut dépasser ce délai (voir section 3.2.3). Toutefois, en stockant les voxels associés à une géocage donnée et un instant (*TimeStamp*) précis, il n'est plus nécessaire de refaire toute la voxélisation pour chaque trajectoire dans un même environnement.

La récupération des zones d'interdiction et de restriction de vol peut également être coûteuse en temps, en fonction de la taille et de la complexité de la géocage (parfois plusieurs dizaines de secondes). Une fois ces zones récupérées et les voxels correspondants générés puis stockés, ce coût n'est plus à supporter pour les calculs suivants.

La solution repose sur une architecture en deux étapes pour le calcul des trajectoires physiques, schématisée par la figure 4.1 :

— **Un planificateur global** (*global planner*) calcule une suite de points intermédiaires reliant les points de passage.
— **Un planificateur local** (*local planner*) s'exécute ensuite entre chaque paire de points intermédiaires pour générer des points associés à une vitesse et un timestamp, assurant ainsi une exécution fluide et physiquement réaliste de la trajectoire par le drone.

Cet algorithme bi-phase prend en compte les contraintes suivantes :

— vitesse maximale horizontale : $v_{\max,\text{horizontal}}$;
— vitesse maximale ascendante : $v_{\max,\text{up}}$;
— vitesse maximale descendante : $v_{\max,\text{down}}$;
— rayon de courbure minimal : $R_i \geq R_{\min} = \dfrac{\|\mathbf{v}_i\|^3}{\|\mathbf{v}_i \times \mathbf{a}_{\max}\|}$ | aussi appeler yaw rate/ taux de lacet ψ ;
— durée entre chaque point de passage : T ;
— vitesse initiale et finale à chaque point de passage : $v_{\text{init}}, v_{\text{goal}}$;
— accélération initiale et finale à chaque point de passage : $a_{\text{init}}, a_{\text{goal}}$;
— variation maximale du *jerk*, fixée à 2 : $jerk_i \approx \dfrac{\|\mathbf{a}_{i+1} - \mathbf{a}_i\|}{\Delta t} \leq 2$;
— obstacles statiques ;
— obstacles dynamiques potentiels, détectés par SLAM et modélisés sous forme d'ellipsoïdes.

Le temps moyen de calcul d'une trajectoire complète (hors temps de voxélisation) est de 0.2 s. En incluant la voxélisation (profondeur 7, soit $8^7 \approx 2$ millions de voxels), le temps moyen total est de 1.3 s. Nous avons évalué la performance de la génération de l'environnement du global planner et du local planner sur une machine plus puissante que celle de l'entreprise. Cependant, tous les constructeurs de drones commencent à intégrer des Jetson de NVIDIA comme processeur embarqué. Sur de l'embarqué en Rust, nous pouvons espérer des performances bien meilleures que celles relevées durant les tests chapter 3.

Comme précisé dans la section 2.2.4, l'heuristique définie par l'équation (2.5) dans une méthode de recherche

arborescente est essentielle. Pour notre planificateur global pseudo-3D basé sur A^*, nous avons choisi de fixer :

$$w_s = 1 \quad \text{et} \quad w_e = 2$$

Ainsi, notre planificateur global privilégie fortement la direction du point d'arrivée. Cette modification tue l'équivalence avec la distance parcourue réelle. L'algorithme trouve le chemin plus rapidement, mais pour une distance plus importante.

Concernant le planificateur local, nous avons modifié la fonction de coût d'Hidra (voir équation (2.7)) en ajoutant un terme pénalisant les fortes variations de *jerk* (section 2.2.5) :

$$E = w_{\text{ep}} \cdot E_{\text{ep}} + w_{\text{dir}} \cdot E_{\text{dir}} + w_{\text{c}} \cdot E_{\text{c}} + w_{\text{j}} \cdot E_{\text{j}} \tag{4.1}$$

où :

— w_{ep} est le poids du coût E_{ep}, représentant la distance entre l'état final et le point cible local : $w_{\text{ep}} = 1.0$;
— w_{dir} est le poids du coût E_{dir}, correspondant à l'écart angulaire entre la direction de l'état final et celle du point cible : $w_{\text{dir}} = 2.0$;
— w_{c} est le poids du coût E_{c}, mesurant la proximité des obstacles : $w_{\text{c}} = 0.0$;
— w_{j} est le poids du coût E_{j}, pénalisant les variations trop brusques de jerk : $w_{\text{j}} = 2.0$.

Au cours de nos tests, nous avons observé que l'introduction du coût de collision augmentait significativement le temps de calcul sans apporter de gain notable, du moins dans la version statique du code. Un simple rejet des primitives intersectant des voxels occupés s'avère suffisant pour permettre une grande performance de calcul de primitives. Notre algorithme peut tout aussi bien fonctionner dans de grands environnements principalement vides que dans des environnements densément occupés pour des vols très agressifs.

4.2 Ouverture

Faute de temps, nous n'avons pas eu l'occasion d'approfondir et de tester le vol en formation (section 2.2.6), un sujet vaste qui aurait nécessité plusieurs mois de travail supplémentaires.

Similairement, nous n'avons pas eu le temps d'aborder la problématique de la planification multi-trajectoires. Par exemple, si une trajectoire a déjà été calculée pour un premier drone, comment déterminer une trajectoire pour un second drone tout en prenant en compte le mouvement du premier ? Notre algorithme est censé éviter dynamiquement les drones trop proches, mais il n'est pas, en l'état, capable d'anticiper ou d'intégrer le déplacement d'autres drones dans sa planification. Une amélioration possible serait d'ajouter une contrainte temporelle lors de la génération des primitives : si à un certain *TimeStamp*, un drone se trouve à une distance inférieure à d_{safe} d'un autre drone (au même *TimeStamp*), alors cette primitive serait rejetée. Toutefois, cette méthode pourrait s'avérer extrêmement coûteuse en temps de calcul.

Actuellement, le planificateur global a été conçu pour trouver un chemin le plus rapidement possible, sans considérer le coût énergétique réel de ce chemin. Le coût pris en compte est purement théorique, défini a priori et indépendamment du planificateur local. Même si l'algorithme pseudo-3D A^* s'est révélé le plus performant parmi ceux testés, il serait pertinent de le combiner avec une heuristique plus réaliste, telle que la distance réelle (cf. équation 2.5) ou encore des facteurs environnementaux comme le vent, à l'instar de l'approche proposée par Zhao [44]. Une carte vectorielle 2D ou 3D des vents pourrait alors être intégrée dans l'heuristique pour favoriser les trajectoires minimisant à la fois la distance et le coût énergétique. Pseudo-3D A^* explorerait alors les chemins en tenant compte de ces nouvelles pondérations, retournant une trajectoire minimisant le coût énergétique théorique du drone.

In fine, notre pipeline retourne plusieurs métriques essentielles : le temps de calcul, la distance parcourue, le nombre d'échecs, et l'énergie consommée. Faute de données précises sur la consommation réelle des drones a posteriori, nous n'avons pas pu évaluer cette dernière. Dans le futur, si nous parvenons à estimer correctement l'énergie consommée et que la puissance de calcul le permet, il serait pertinent de générer un grand nombre de trajectoires entre deux points donnés, et de sélectionner celle qui se rapproche le plus de l'optimum de Pareto. L'algorithme EMOA^* [34] semble prometteur à cet égard pour une intégration en tant que planificateur global.

Le droniste NEODE Systems, créé et financé par MBDA, a fait le choix d'implémenter un réseau de neurones pour permettre le calcul dynamique de trajectoires multi-UAV et multi-objectifs. C'est une approche qui nous a semblé trop complexe à mettre en œuvre dans le temps imparti, et avec le peu de ressources disponibles en source ouverte.

Il pourrait toutefois rester pertinent de transformer le planificateur local en réseau de neurones. Ce réseau devrait alors être capable de prendre en compte les différents types d'UAV possibles, impliquant potentiellement l'utilisation de réseaux de neurones distincts pour chaque type d'UAV.

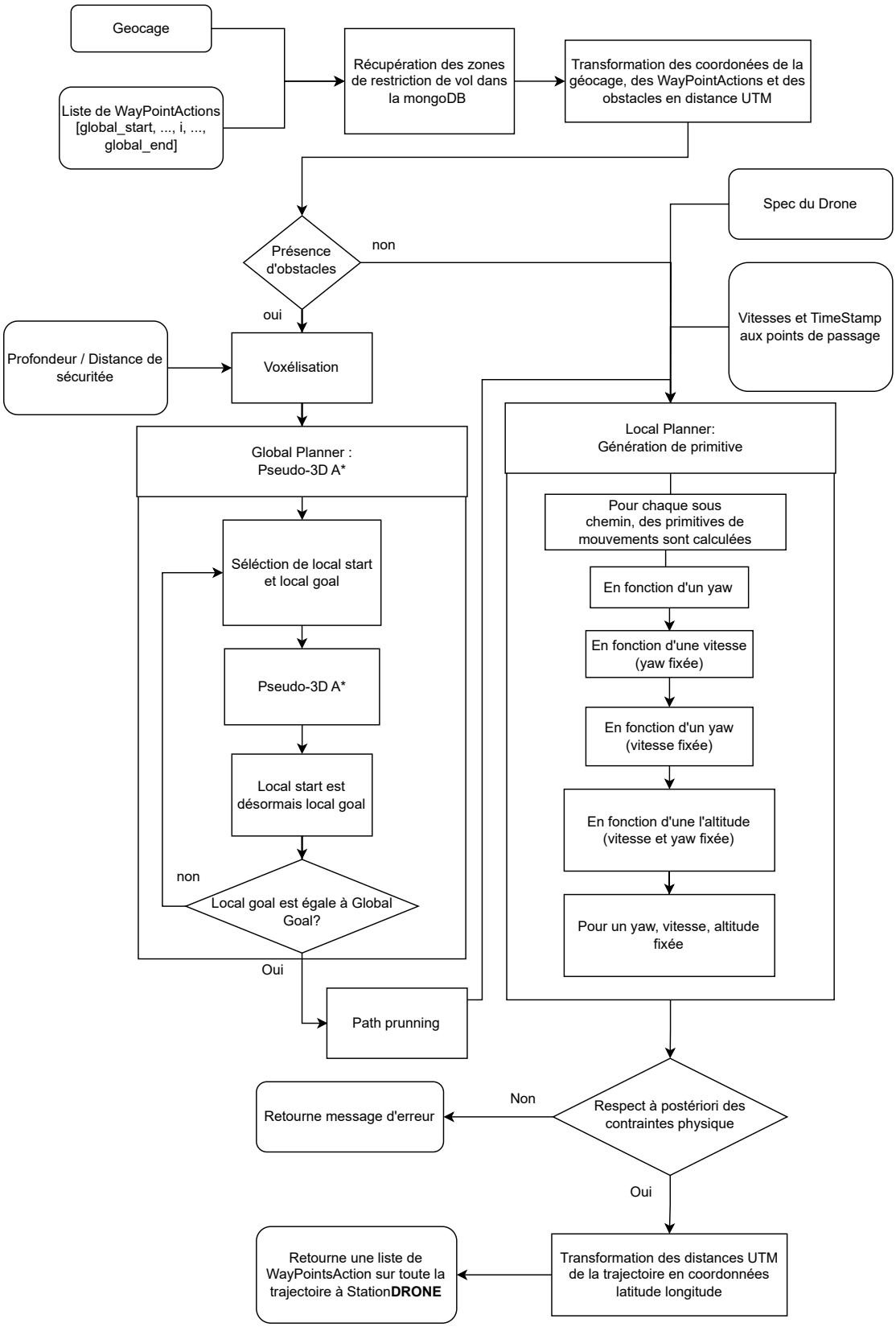


FIGURE 4.1 – Pipeline finale

Remerciements

Je tiens à remercier chaleureusement Drone Géofencing pour m'avoir offert l'opportunité de cette alternance, ainsi que l'Université de Lorraine pour m'avoir intégré à son Master. Je souhaite également exprimer ma profonde reconnaissance à Monsieur Dutech pour son suivi attentif et le déroulement optimal de mon parcours. Enfin, je remercie l'Université de Montpellier pour m'avoir ouvert ses portes et j'adresse une pensée particulière à Monsieur Giroudeau pour son soutien et ses précieux conseils.

Chapitre 5

Annexe

Algorithm 1 A^* pondéré selon l'Éq. (2.5)

$OpenSet \leftarrow \{s\}, \quad ClosedSet \leftarrow$
$g\_score[s] \leftarrow 0$
$f\_score[s] \leftarrow w_s \cdot g\_score[s] + w_e \cdot d(s,t)$ ▷ Eq. (2.5) : $h(n) = w_s \cdot d(n,s) + w_e \cdot d(n,t)$
while $OpenSet \neq$ **do**
    $n \leftarrow \arg\min_{x \in OpenSet} f\_score[x]$
    if $n = t$ **then return** reconstruct_path(n)
    end if
    $OpenSet \leftarrow OpenSet \setminus \{n\}, \quad ClosedSet \leftarrow ClosedSet \cup \{n\}$
    for all $m \in$ Neighbors(n) $\setminus ClosedSet$ **do**
        $tentative\_g \leftarrow g\_score[n] + d(n,m)$
        if $m \notin g\_score$ **or** $tentative\_g < g\_score[m]$ **then**
            $came\_from[m] \leftarrow n$
            $g\_score[m] \leftarrow tentative\_g$
            $f\_score[m] \leftarrow w_s \cdot g\_score[m] + w_e \cdot d(m,t)$
            $OpenSet \leftarrow OpenSet \cup \{m\}$
        end if
    end for
end while
return `chemin introuvable`

Algorithm 2 RRT

$T \leftarrow \{s\}$ ▷ arbre initial
while itérations restantes **do**
    $x_{\text{rand}} \leftarrow$ échantillon aléatoire
    $x_{\text{near}} \leftarrow \arg\min_{x \in T} \|x - x_{\text{rand}}\|$
    $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{near}}, x_{\text{rand}})$
    if collisionfree($x_{\text{near}}, x_{\text{new}}$) **then**
        add x_{new} et $(x_{\text{near}}, x_{\text{new}})$ à T
        if $\|x_{\text{new}} - t\| < \epsilon$ **then return** chemin depuis T
        end if
    end if
end while
return `échec`

Algorithm 3 Local Planner : optimisation séquentielle 1D (Hidra 2019, [22])

Note : $f(\theta, v, z)$ est la fonction de résolution polynomiale 1D évaluant le coût.

function LocalPlanner($\theta_0, v_0, z_0, T_f, z_G$)
  $v_1 \leftarrow \max\big(0.1,\ v_0 - 1/T_f\big)$
  $\theta_1^* \leftarrow \arg\min_{\theta \in [\theta_0 - \frac{\pi}{2},\, \theta_0 + \frac{\pi}{2}]} f(\theta,\, v_1,\, z_0)$
  $v^* \leftarrow \arg\min_{v \in [v_0 - 4/T_f,\, v_0 + 1/(2T_f)]}\ f(\theta_1^*,\, v,\, z_0)$
  $\theta_2^* \leftarrow \arg\min_{\theta \in [\theta_1^* - 0.4\pi,\, \theta_1^* + 0.4\pi]} f(\theta,\, v^*,\, z_0)$
  $z^* \leftarrow \arg\min_{z \in [z_0,\, z_G]}\ f(\theta_2^*,\, v^*,\, z)$
  return $(\theta_2^*,\, v^*,\, z^*)$
end function

Bibliographie

[1] Nour AbuJabal, Mohammed Baziyad, Raouf Fareh, Brahim Brahmi, Tamer Rabie, and Maamar Bettayeb. A comprehensive study of recent path-planning techniques in dynamic environments for autonomous robots. *Sensors*, 24(24) :8089, 2024.

[2] Benoit Alain. Hierarchical dynamic pathfinding for large voxel worlds, 2023. Vidéo YouTube.

[3] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Proceedings of the Eurographics Conference*, 1987 :3–10, 1987.

[4] Tucker Balch and Ronald C. Arkin. Behavior-based formation control for multi-robot teams. *IEEE Transactions on Robotics and Automation*, 14(6) :926–939, 1999.

[5] Tavian Barnes. Fast, branchless ray/bounding box intersections, part 3 : Boundaries, 2022.

[6] Salvatore Rosario Bassolillo, Gennaro Raspaolo, and Immacolata Notaro. Path planning for fixed-wing unmanned aerial vehicles : An integrated approach with * and clothoids. *Drones*, 8(2) :78, 2024.

[7] Sumana Biswas, Sreenatha G. Anavatti, and Matthew A. Garratt. Path planning and task assignment for multiple uavs in dynamic environments. *Aerospace Science and Technology*, 112 :106548, 2021.

[8] Thorsten Brandt and Thomas Sattel. Path planning for automotive collision avoidance based on elastic bands. *IEEE Transactions on Intelligent Transportation Systems*, 6(4) :390–400, 2005.

[9] Yanjie Cao and Norzalilah Mohamad Nor. An improved dynamic window approach algorithm for dynamic obstacle avoidance in mobile robot formation. *Sensors*, 24(3) :1012, 2024.

[10] Xucheng Chang, Jingyu Wang, Kang Li, Xinhui Zhang, and Qian Tang. Research on multi-uav autonomous obstacle avoidance algorithm integrating improved dynamic window approach and orca. *Aerospace Science and Technology*, 135 :109595, 2025.

[11] Thai-Vietdang Dang. Autonomous mobile robot path planning based on enhanced a* algorithm integrating with time elastic band. *Robotics and Autonomous Systems*, 167 :104364, 2023.

[12] Jesson J. Einmahl, John H. J. Einmahl, and Laurens de Haan. Limits to human life span through extreme value theory. *Journal of the American Statistical Association*, 113(522) :24–33, 2018.

[13] Honghui Fan, Jiahe Huang, Xianzhen Huang, Hongjin Zhu, and Huachang Su. Bi-rrt* : An improved path planning algorithm for secure and trustworthy mobile robot systems. *Robotics and Autonomous Systems*, 169 :104625, 2024.

[14] Dieter Fox and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1) :23–33, 1997.

[15] Atef Gharbi. Bi-directional adaptive enhanced a* algorithm for mobile robot navigation. *Sensors*, 24(6) :2897, 2024.

[16] Gopi Gugan and Anwar Haque. Path planning for autonomous drones : Challenges and future directions. *Unmanned Systems*, 11(2) :47–66, 2023.

[17] Daniel Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. *Artificial Intelligence*, 190 :1–19, 2011.

[18] Daniel Harabor, Alban Grastien, Dindar Öz, and Vural Aksakalli. Optimal any-angle pathfinding in practice. *Journal of Artificial Intelligence Research*, 56(1) :89–118, 2016.

[19] Ruoqi He and Chia-Man Hung. Pathfinding in 3d space : A*, *, lazy * in octree structure. *Applied Intelligence*, 45(2) :350–364, 2016.

[20] Yong He, Ticheng Hou, and Mingran Wang. A new method for unmanned aerial vehicle path planning in complex environments. *Aerospace Science and Technology*, 135 :109588, 2024.

[21] Hexadrone. https ://www.hexadrone.fr/produits/drone-tundra/.

[22] Rémy Hidra. A uav motion planning framework with global and local (re-)planning for 3d environments. Master's thesis, Shanghai Jiao Tong University, Department of Electronic Engineering, Shanghai, China, July 2021.

[23] Myung Hwangbo, James Kuffner, and Takeo Kanade. Efficient two-phase 3d motion planning for small fixed-wing uavs. *Proceedings of the IEEE International Conference on Robotics and Automation*, 2007 :2876–2881, 2007.

[24] Kristyna Janovska and Pavel Surynek. Multi-agent path finding in continuous environment. *Computers & Operations Research*, 155 :105349, 2024.

[25] Nan Li and Sang Ik Han. Adaptive bi-directional rrt algorithm for three-dimensional path planning of unmanned aerial vehicles in complex environments. *Sensors*, 24(4) :2124, 2024.

[26] Yucong Lin and Srikanth Saripalli. Path planning using 3d dubins curve for unmanned aerial vehicles. *Journal of Intelligent & Robotic Systems*, 74(1-2) :73–90, 2014.

[27] Yuanchang Liu and Richard Bucknall. A survey of formation control and motion planning of multiple unmanned vehicles. *Annual Reviews in Control*, 37(1) :153–171, 2013.

[28] Quentin Massonnat and Clark Verbrugge. Efficient octree-based 3d pathfinding. *Journal of Computational Geometry*, 13(1) :45–66, 2022.

[29] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. *Proceedings of the IEEE International Conference on Robotics and Automation*, 2011 :2520–2525, 2011.

[30] Chaojun Qin, Na Zhao, Yudong Luo, and Yantao Shen. Minimum snap trajectory planning and augmented mpc for morphing quadrotor navigation in confined spaces. *IEEE Transactions on Control Systems Technology*, 33(1) :112–125, 2025.

[31] Kemeng Ran, Yujun Wang, Can Fang, Qisen Chai, Xingxiang Dong, and Guohui Liu. Improved rrt* path-planning algorithm based on the clothoid curve for a mobile robot under kinematic constraints. *Robotics and Autonomous Systems*, 178 :104592, 2024.

[32] Abhijeet Ravankar, Ankit A. Ravankar, Yukinori Kobayashi, and Takanori Emaru. Shp : Smooth hypocycloidal paths with collision-free and decoupled multi-robot path planning. *IEEE Transactions on Robotics*, 32(4) :867–882, 2016.

[33] Abhijeet Ravankar, Ankit A. Ravankar, Chao-Chung Peng, Yukinori Kobayashi, and Yohei Hoshino. Path smoothing techniques in robot navigation : State-of-the-art, current and future challenges. *Sensors*, 18(9) :3170, 2018.

[34] Zhongqiang Ren, Carlos Hernández, Maxim Likhachev, Ariel Felner, Sven Koenig, Oren Salzman, Sivakumar Rathinam, and Howie Choset. EMOA* : A framework for search-based multi-objective path planning. *Artificial Intelligence*, 339 :104260, 2025.

[35] J. Revelles, C. Ureña, and M. Lastra. An efficient parametric algorithm for octree traversal. *Computer Graphics Forum*, 19(3) :91–101, 2000.

[36] Charles Richter, Adam Bry, and Nicholas Roy. Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. *Proceedings of the Robotics : Science and Systems Conference*, 2013 :1–8, 2013.

[37] Zhaokang Shenga, Tingqiang Song, Jiale Song, Yalin Liu, and Peng Ren. Bidirectional rapidly exploring random tree path planning algorithm based on adaptive strategies and artificial potential fields. *Robotics and Autonomous Systems*, 183 :104659, 2025.

[38] Yongbin Su, Chenying Lin, and Tundong Liu. Real-time trajectory smoothing and obstacle avoidance : A method based on virtual force guidance. *Robotica*, 42(4) :823–840, 2024.

[39] Kar-Han Tan, Anthony Lewis, and Anthony Lewis. Virtual structures for high-precision cooperative mobile robot control. *IEEE International Conference on Robotics and Automation*, 1996 :460–466, 1996.

[40] Francesco Trotti, Alessandro Farinelli, and Riccardo Muradore. A markov decision process approach for decentralized uav formation path planning. *Autonomous Robots*, 48(2) :285–303, 2024.

[41] Dongxiao Yang, Didong Li, and Huafei Sun. 2d dubins path in environments with obstacles. *Robotics and Autonomous Systems*, 61(12) :1328–1338, 2013.

[42] Jiangyi Yao, Xiongwei Li, Yang Zhang, Jingyu Ji, Yanchao Wang, and Yicen Liu. Path planning of unmanned helicopter in complex dynamic environment based on state-coded deep q-network. *Symmetry*, 14(5) :856, 2022.

[43] Junqiao Zhao, Qiaoyu Xu, and Sisi Zlatanova. Weighted octree-based 3d indoor pathfinding for multiple locomotion types. *ISPRS International Journal of Geo-Information*, 11(7) :358, 2022.

[44] Liang Zhao, Yong Bai, and Jeom Kee Paik. Global–local hierarchical path planning scheme for unmanned surface vehicles under dynamically unforeseen environments. *Ocean Engineering*, 280 :114193, 2023.

[45] Boyu Zhou, Fei Gao, Luqi Wang, Chuhao Liu, and Shaojie Shen. Robust and efficient quadrotor trajectory generation for fast autonomous flight. *IEEE Transactions on Robotics*, 35(2) :324–341, 2019.

[46] Hai Zhu, Francisco Martinez Claramunt, Bruno Brito, and Javier Alonso-Mora. Learning interaction-aware trajectory predictions for decentralized multi-robot motion planning in dynamic environments. *IEEE Robotics and Automation Letters*, 6(2) :2256–2263, 2021.