Politecnico di Torino

Politecnico di Torino

Master's Degree in Mechatronic Engineering

Spiral-Based Reactive Obstacle Avoidance for UAVs with PX4 SITL Integration

Relatore:

Dr. Stefano Primatesta

Candidato:

Alessandro Munafo'

Correlatore:

Dr. Riccardo Enrico

A.Y. 2024/2025

Abstract

The noticeable rise of drones, or UAVs, in business, rescue operations, and general applications necessitates increased safety due to their growing presence. As drones begin to operate alongside humans and existing structures, there is a need for preventing collisions with objects and/or people.

This thesis presents a potential solution to the obstacle avoidance problem, enabling drones to avoid possible collisions during navigation. The algorithm relies on an Archimedean spiral to identify an escape point in real time based on environmental data perceived by the depth Camera. During navigation, the algorithm continuously processes the point cloud information, generating a suitable escape point if an obstruction is detected within a safety cylindrical volume.

The implementation leverages the co-simulation between ROS 2 and PX4 Autopilot via uORB messages for seamless communication between high-level setpoint tracking and low-level flight controllers. Octomap processes the point cloud provided by the depth camera to retrieve a probabilistic 3D representation of the surrounding environment while maintaining reduced memory occupancy. Extensive Gazebo simulation validates system effectiveness across different scenarios.

Keywords: UAV, Collision Avoidance, ROS 2, PX4, Depth Camera, Real-time Systems, Autonomous Navigation

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

In recent years, the UAV market has been experiencing a rapid expansion, affecting multiple industries based on macro-drivers. Improvements in battery efficiency, autonomous systems, sensor miniaturization, and advanced communication technologies generate interest among players in the commercial, industrial, and civilian sectors. Autonomous flight, and in general, UAVs, are increasing their usage in various fields, as their applicability can be varied and highly efficient, or, in some cases, they are the only way to complete tasks that involve human risk in hazardous environments.

At the same time, some market trends are leveraging UAV technologies to achieve better efficiency; for instance, increasing demand from the e-commerce market can drive a rise in drone usage, particularly for last-kilometer delivery, leading to a more efficient service, reduced energy consumption (with a clear lower environmental impact), and increased accessibility at lower costs. While efficiency and profitability are important drivers, these technologies demonstrate their greatest value when deployed to address urgent humanitarian needs. Zipline's operations in Rwanda provide a compelling example of UAVs being utilized to transport essential medical supplies to hospitals in remote areas.

All the described application examples demonstrate the variety of advantages that UAVs offer, with autonomous navigation representing the most critical enabling technology in drastically reducing human intervention. In this context, safety plays a crucial role, particularly in low-altitude navigation operations, where proximity to people, infrastructure, and buildings can significantly limit the scalability of UAVs, not only for commercial deployment.

Waypoints navigation can be a challenging task for an autonomous entity due to complex environmental challenges, dynamic obstacles, unpredictable disturbances, and other factors. In most cases, a pre-loaded path from the origin to the destination point may not suffice, as it cannot account for dynamic obstacles encountered along the way. Additionally, although the locations of buildings could theoretically be known in advance, implementing this strategy would necessitate substantial memory usage to store information about all possible buildings, and there may also be instances where such knowledge is simply unavailable. For this reason, to enable UAV's safe autonomous navigation, a real-time obstacle avoidance approach is required to achieve a fully autonomous entity.

1.2 Problem Statement

While autonomous navigation offers clear advantages, implementing effective obstacle avoidance algorithms faces several challenges arising from algorithmic limitations, hardware constraints, and complexities in system integration. From an algorithmic perspective, many existing solutions risk becoming trapped in local minima or generating suboptimal trajectories that fail to achieve the desired objectives, particularly those related to local planners. Additionally, many available solutions focus on framing the obstacle avoidance problem in a 2D context, which simplifies algorithm complexity but comes at the cost of losing essential altitude information. In this context, resource limitations pose a significant bottleneck that restricts the deployment of advanced obstacle avoidance systems, particularly for small and medium-sized UAVs with restricted payload capacity and power. These systems typically rely on embedded computing solutions, which provide only a fraction of the processing power available in ground-based applications while simultaneously needing to manage various concurrent tasks, such as flight control, sensor processing, data management, communication, and navigation. Real-time obstacle avoidance algorithms are particularly resource-intensive, as perception sensors require complex processing that often exceeds the capabilities of standard onboard computers in UAVs.

Furthermore, carrying the necessary payload for adequate perception increase power consumption, thereby limiting the UAV's operational range. These challenges limit the broader application of UAVs, resulting in suboptimal performance, particularly in vision-based solutions, where a UAV's speed is closely tied to the processing capabilities of the perception system that generates point clouds. Addressing these limitations could enhance both safety and expand applicability into new fields.

1.3 Objectives

The primary goal involves adapting and integrating the spiral-based collision avoidance algorithm, initially developed by Azevedo et al. [7], into the ROS 2 framework. This integration involves establishing seamless communication with the PX4 flight stack through Software-in-the-Loop (SITL) simulation, utilising uORB messages to facilitate effective coordination between high-level navigation commands and low-level flight control systems.

The second objective centres on comprehensive validation and performance assessment of the integrated system through extensive simulation testing. The overall effectiveness of the spiral-based obstacle avoidance solutions, subject to different operational scenarios, was evaluated using Gazebo Sim with PX4 SITL.

Through these integration and validation efforts, this thesis contributes to the practical advancement of UAV autonomous navigation by demonstrating how existing, proven algorithms can be successfully adapted to modern robotics frameworks while providing a stable and documented platform that enables future algorithm developers to build upon this work without requiring a fundamental re-implementation of the integration layer.

1.4 Thesis Organization

This thesis work is organised into nine chapters, providing detailed coverage from the theoretical foundations to its integration and validation in a simulated environment.

- **Chapter 2** presents the state of the art of obstacle avoidance available solutions.
- **Chapter 3** details the theoretical background of the spiral-based obstacle avoidance algorithm, perception sensors and its processing methodologies.
- **Chapter 4** explains the overall system architecture and design decisions, describing the ROS 2 node structure, PX4 integration framework, and coordinate transformations required for accurate simulation implementation.
- **Chapter 5** focuses on the core implementation details of both ROS 2 nodes and their interaction with high-level flight control systems, representing the primary technical contribution of this work.
- **Chapter 6** provides insights about the simulation environment and its setup.
- **Chapter 7** shows the results obtained in this thesis work.
- **Chapter 8** discusses implementation achievements, limitations encountered during the development process, along with conclusions and future works.
- **Appendix A** provides quick-start setup procedures, enabling reproduction of the experimental framework.
- **Appendix B** integrates an overview of supporting algorithms used for main ROS 2 nodes and related resources.

Chapter 2

Literature Review

UAVs operating at low altitudes face a significant threat from non-cooperative obstacles, which include both static objects, such as buildings or trees, and dynamic ones, like other UAVs. This scenario poses a substantial risk of collision due to increased obstructions and reduced operational speeds compared to higher altitudes. Consequently, obstacle avoidance is a **critical capability** for low-altitude applications. To address this, various non-cooperative obstacle avoidance techniques have been developed. These methods can be divided into two main categories:

- **Global Path Planning:** This involves determining the optimal route between the starting point and the final goal. However, **it requires prior knowledge of the exact environment map**. Such methods can be computationally expensive to build and update and prone to errors due to map latency and dynamic actors that cannot be known a priori.
- **Local Obstacle Avoidance:** These techniques operate **without prior knowledge of the area map**. They continuously monitor the UAV's proximity and decide on the avoidance action based on the perception sensor feedback. Continuous sensing and real-time map updating, contribute to reducing the likelihood of latency issues. Local methods are generally preferred for onboard processing on small robotic devices due to their **low computational cost**. Therefore, they are ideal for quick responses, but at the same time, they may not find the optimal path or risk being trapped in dead-end situations.

A complete approach is often a combination of both strategies, using a low-level reactive layer for immediate safety and a high-level global planner for optimized long-term paths [11]. This review focuses on non-cooperative local obstacle avoidance methods, which can be further categorized into gap-based, geometric, repulsive force-based, and AI-based approaches.

2.1 Gap-based methods

Gap-Based methods are effective for collision-free UAV navigation by identifying and utilizing admissible gaps between obstacles. They are often computationally efficient and suitable for real-time applications, solving the avoidance problem for both static and dynamic obstacles [1], [12].

2.1.1 Open Sector

The **Open Sector (OS)** method, presented in [13], enables aerial robots to **navigate reactively without maps** by leveraging a 2D laser scan. It identifies open sectors (clear, wide angular arcs) and calculates a virtual target by incorporating a short-term memory of past actions to guide the robot and prevent it from being trapped in dead-end scenarios. Safety boundaries are applied within chosen sectors to ensure smooth, tangential, and safe travel around obstacles. Additionally, virtual walls are used to manage the laser scanner's blind spot. For robust operation, the system includes emergency actions for close obstacles and switches to a PF-IPA (Potential Field that Incorporates Past Actions) method if no open sectors are found or when approaching a waypoint closer than the look-ahead distance. It can achieve a **smooth trajectory at relatively high speeds** (e.g., 3 m/s on a physical system) and effectively handles multiple obstacles. A limitation is that it **does not account for the aerial robot's dynamics**, which may reduce its effectiveness in highly dynamic environments.

2.1.2 Nearness Diagram

The **Nearness Diagram (ND)** method, presented in [1], utilizes two primary diagrams: the Nearness Diagram from the central Point (PND) and the Nearness Diagram from the Robot bounds (RND). These diagrams serve as crucial tools for analyzing the relationships between the robot, the distribution of obstacles, and the goal location. The construction of both the PND and RND begins by processing sensory information, which is assumed to be available as depth point maps. This raw sensory data is used to divide the plane into sectors. For each sector i, a function `min_dist(i)` is computed, which determines the minimum distance to an obstacle point within that sector.

Each diagram is constructed as follows:

- PND : This diagram represents the nearness of obstacles as perceived from the robot's centre. Its values are directly derived from the `min_dist(i)` function for each sector. The PND is fundamental for identifying gaps in the obstacle distribution, which are defined as discontinuities in the PND between adjacent sectors. From these gaps, regions are then obtained, which are identified as valleys in the PND, formed by two contiguous gaps.

- RND : This diagram, in contrast, represents the nearness of obstacles from the robot's boundary, taking into account the robot's radius. It is computed using the `min_dist(i)` function and the robot's radius R, such that if `min_dist(i)` is less than or equal to R, the RND value is R, otherwise it is `min_dist(i)`. The RND is specifically used to evaluate the robot's safety by checking if any obstacles are within a predefined "security zone" around the robot's bounds.

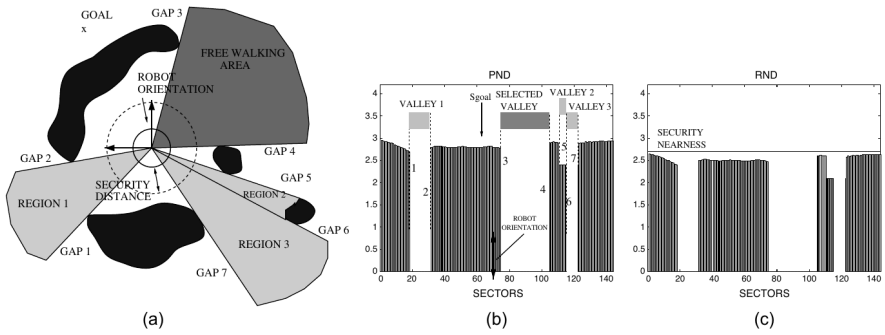

Figure 2.1: (a) Gaps, regions and free walking area. (b) PND. (c) RND. [1]

After these diagrams are constructed, the robot's environment is categorized into a set of "situations" with a corresponding "action" to perform. The situation identification process leverages PND and RND to evaluate the environment. This process is described in the decision tree shown in fig. 2.2

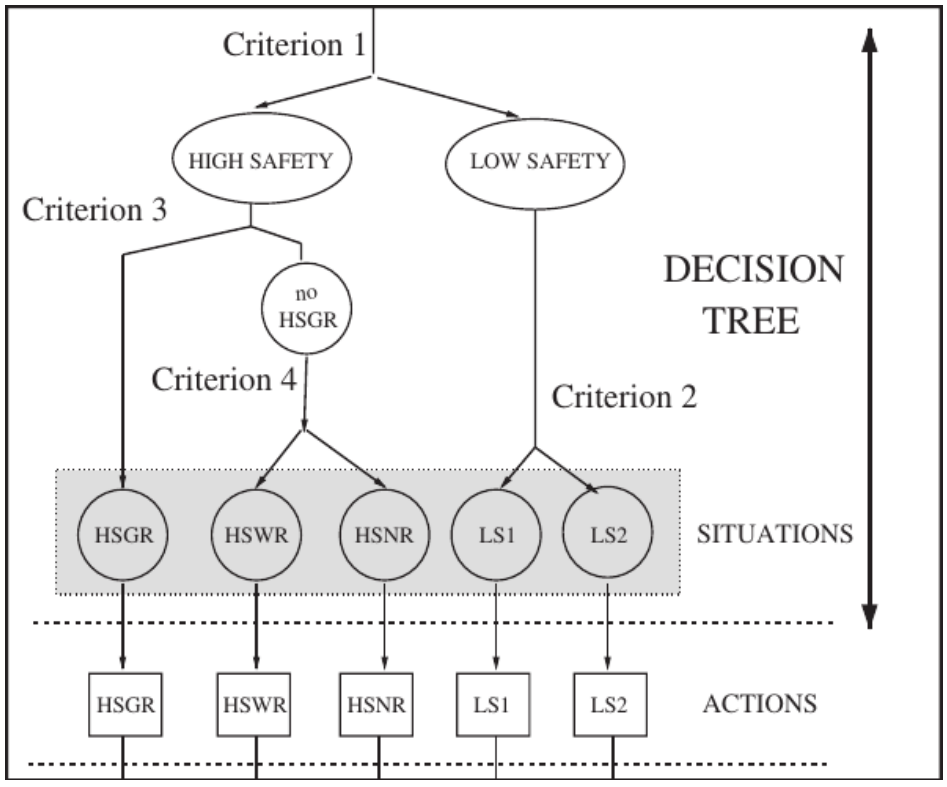

Figure 2.2: Nearness diagram architecture [1]

The decision tree navigation is based on the following criteria:

- **Criterion 1:** *Safety Criterion* - Checks leveraging RND, whether there are obstacles within a "security zone" around the robot's bounds leading to Low Safety (LS) if obstacles are present within the security zone or to High Safety (HS) otherwise
- **Criterion 2:** *Dangerous obstacle distribution criterion* - Leads to **Low Safety 1 (LS1)** if obstacles are located only on one side of the gap, or **Low Safety 2 (LS2)** if they are on both sides of the gap
- **Criterion 3:** *Goal within the free walking area criterion* - If the goal is within the free walking area, it leads to **High Safety Goal in Region (HSGR)**
- **Criterion 4:** *Free walking area width criterion* - Differentiates between wide and narrow free walking areas based on their angular width, leads to High Safety Wide Region (HSWR) if the selected free walking area is wide, to High Safety Narrow Region (HSNR) otherwise

For each of these five situations (LS1, LS2, HSGR, HSWR, HSNR), a specific action design is associated with producing the desired navigation behaviour. The associated actions are:

- **LS1 Action:** Calculates a motion direction that moves the robot away from the closest obstacle and toward the gap (closest to the goal) of the free walking area. The translational velocity is reduced proportionally to the distance to the nearest obstacle.
- **LS2 Action:** Centres the robot between the two closest obstacles on both sides of the gap in the free walking area while simultaneously moving the robot toward this gap. The translational velocity is also reduced in proportion to the proximity of the obstacle.
- **HSGR Action:** Directly drives the robot toward the goal location. The robot moves at maximum translational velocity.
- **HSWR Action:** Moves the robot alongside the obstacle towards the goal. The robot moves at maximum translational velocity.
- **HSNR Action:** Directs the robot through the central zone of the free walking area, computed as the bisector of the discontinuities defining the valley. The robot moves at maximum translational velocity.

In conclusion, it successfully navigates through cluttered environments, including avoiding common trap situations like U-shaped obstacles. However, a successful mission relies heavily on the quality of sensor data.

2.1.3 VFH and evolutions

One of the most effective algorithm is the **Vector Field Histogram (VFH)**, along with its subsequent evolutions (VFH+ and VFH*), which forms a cornerstone in the field of mobile robot obstacle avoidance. Developed to enable real-time, local navigation in unknown environments, this family of algorithms has continuously evolved to address inherent limitations and enhance reliability, particularly for fast-moving platforms like mobile robots and Unmanned Aerial Vehicles (UAVs).

The original VFH algorithm, introduced by Borenstein and Koren [14], was designed as a real-time local obstacle avoidance method for robots equipped with proximity sensors. It aimed to overcome some of the shortcomings of earlier potential field methods, such as susceptibility to local minima, enabling robots to travel at faster and more stable speeds.

VFH operates by processing a local environment map, often referred to as a "histogram grid," which is derived from certainty or occupancy grids. The algorithm employs a two-stage data reduction process to determine the robot's steering direction:

- Primary Polar Histogram Construction: A circular "active region" around the robot's momentary location within the two-dimensional map grid is transformed into a one-dimensional "primary polar histogram". Each active cell in the grid contributes as an "obstacle vector". The direction of this vector points from the active cell to the robot's centre, and its magnitude is directly proportional to the square of the cell's "certainty value" and inversely proportional to the square of its distance from the robot. This ensures that closer and more certain obstacles exert a stronger influence.

- Steering Direction Selection: From the primary polar histogram, the algorithm identifies "valleys" or openings that represent free space. The original VFH is highly goal-oriented, selecting the opening that most closely aligns with the target direction to define the robot's new steering command.

The VFH method was a notable advancement because it allowed robots to perform real-time obstacle avoidance at high speeds. Compared to earlier potential field methods, VFH was less prone to getting trapped in local minima and demonstrated greater stability, even when the robot travelled quickly. Despite its advantages, the original VFH algorithm had several weaknesses. It did not explicitly account for the robot's physical width, instead relying on an empirically determined low-pass filter for compensation. This filter was difficult to tune and could lead to the robot cutting corners. Furthermore, VFH neglected the robot's dynamics and kinematics, assuming it could change direction instantaneously. This fundamental assumption could result in the robot being incorrectly guided into obstacles, particularly if its turning radius

prevented an immediate course correction. The use of a fixed threshold for identifying free paths could also cause indecisive or oscillatory behaviour in environments with narrow openings. Being a purely local algorithm, it could sometimes make undesirable choices or lead the robot into dead-ends.

The first evolution, named VFH+, was introduced by Ulrich and Borenstein in 1998 [15] and addressed key limitations of the original VFH method, resulting in smoother robot trajectories and enhanced reliability.

It introduces a four-stage data reduction process (histogram grid, primary polar histogram, binary polar histogram, masked polar histogram) to compute the new direction of motion. One significant improvement is the explicit compensation for robot width. VFH+ analytically determines a low-pass filter and enlarges obstacle cells by the robot's radius (and a safety distance), effectively treating the robot as a point-like vehicle. This eliminates the complex tuning required for the original VFH's low-pass filter. Crucially, VFH+ takes into account the robot's trajectory, assuming movement along circular arcs and straight lines, preventing the algorithm from selecting paths that the robot's physical turning capabilities would make impossible, thereby avoiding collisions that the original VFH might not. To combat indecisive behaviour, VFH+ incorporates a threshold hysteresis using two thresholds (minimum and maximum) when constructing the binary polar histogram, which makes trajectories less oscillatory. Finally, VFH+ features an improved direction selection based on a cost function which considers goal-oriented behaviour, path smoothness, and steering command smoothness, allowing the robot to "commit" to a direction. This commitment prevents the robot from hesitating and potentially bumping into obstacles when faced with a single object in its path. VFH+ demonstrated safety at higher speeds (up to 1 m/s in Guidecane tests) and proved to be insensitive to its parameter values, simplifying implementation. It also gained the ability to detect when the robot is trapped in a dead-end situation.

Despite these significant enhancements, VFH+ remained a purely local obstacle avoidance algorithm. This inherent local nature meant that in some complex or ambiguous situations (e.g., a long, narrow corridor with obstacles), it could still make undesirable choices that might lead the robot into dead-ends or suboptimal paths that could have been avoided with a broader perspective.

To address the main limitation, **VFH*** [16] was implemented , combining VFH+ with the A* search algorithm to create a look-ahead tree and evaluate candidate directions. This approach overcomes the limitations of both VHF and VFH+. However, since VFH* does not function as a local planner, the working principles, advantages, and limitations of this method will not be discussed in this thesis.

Since its early 2D implementation, VHF algorithm, have been significantly extended and refined for three-dimensional environments, leading to the 3DVFH+ and its subsequent evolutions, culminating in 3DVFH* [3].

The 3DVFH+ algorithm, as introduced by Vanneste et al. [2], extends the histogram-based obstacle avoidance approach to 3D environments, addressing the needs of platforms such as UAVs.

It operates by first building a global map of the environment, typically utilizing an Octomap data structure. From this global map, the algorithm extracts local information within a bounding box around the robot to perform histogramic obstacle avoidance. This process involves constructing a 2D primary polar histogram from the 3D occupancy map, where the location of an active voxel is determined by its azimuth and elevation angles relative to the Vehicle Center Point (VCP). The size of the robot is compensated by enlarging active voxels in the histogram based on the robot's radius and safety radius. This primary polar histogram is then converted into a 2D binary polar histogram using two thresholds to distinguish real obstacles from measurement errors. Finally, the algorithm identifies openings in the binary histogram by using a moving window. It selects the optimal path by evaluating candidate directions with a cost function that considers the target angle, robot rotation, and previous selections. The chosen direction is then converted into a robot motion.

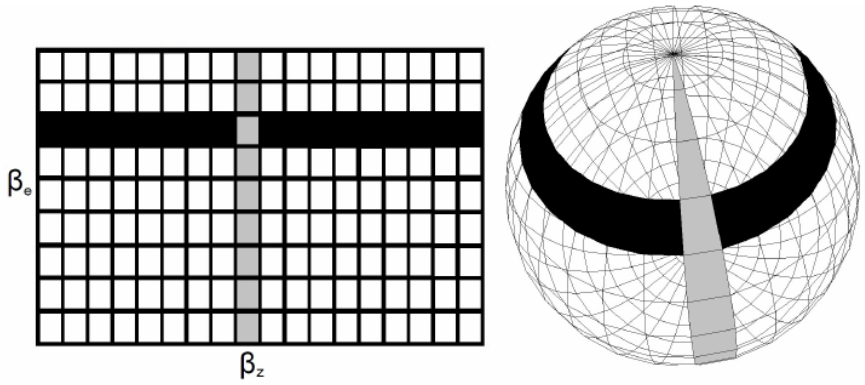

Figure 2.3: 2D Polar Histogram [2]

A key advantage of the 3DVFH+ is its reliance on a global map, which allows it to remember previously seen obstacles even if they are no longer within the UAV's immediate field of view. This characteristic positions it as an approach that lies between purely local and purely global methods. It can calculate a new robot motion with an average time of 300 µs, making it suitable for real-time applications. The algorithm explicitly accounts for the robot's physical characteristics, such as size and turning speed, which contribute to smoother trajectories.

The primary limitation of the 3DVFH+ is the significant computational overhead associated with building and maintaining a global map. This can be a considerable challenge for UAVs, which typically have limited onboard computing power and strict weight constraints. The algorithm's parameters are often chosen empirically, suggesting a need for configuration tailored to specific robot requirements.

To address the high computational demands associated with the global map in 3DVFH+, a localized version of the 3DVFH method was developed as part of the thesis [3]. This adaptation transforms 3DVFH into a purely local and reactive algorithm, eliminating the need for a persistent global map. Additionally, this version was updated with a memory strategy to incorporate previously encountered obstacles without incurring the computational costs associated with maintaining a full global map. Finally, further advancements were made by integrating these concepts with the A* search algorithm, resulting in the creation of the 3DVHF* approach.

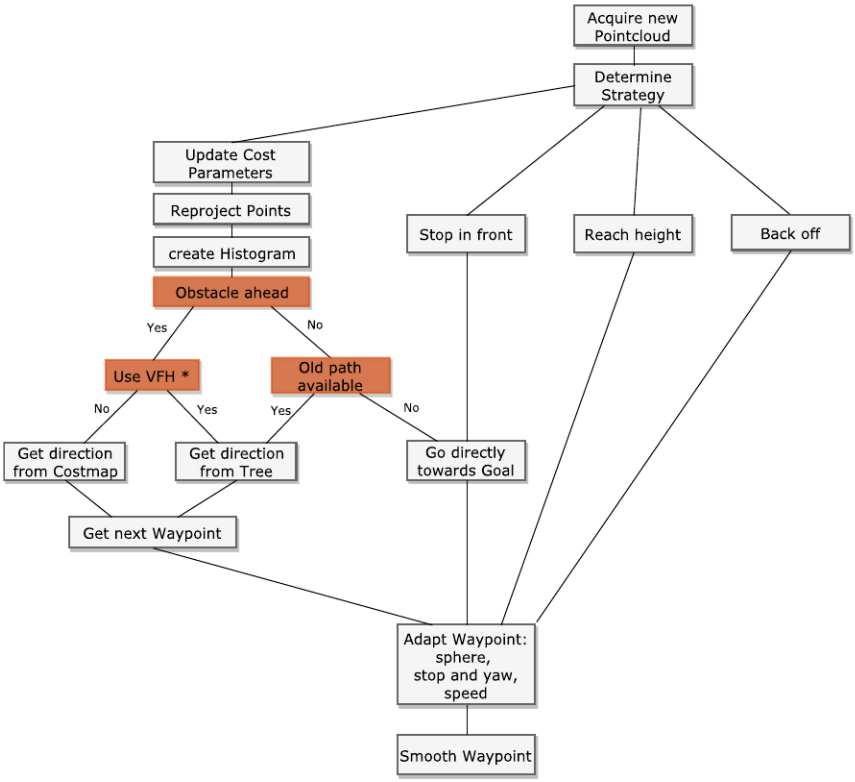

Figure 2.4: Decision-making pipeline for 3DVFH* [3]

fig. 2.4 illustrates the 3DVFH* decision-making pipeline, outlining the process from point-cloud data acquisition to the output of the next waypoint for the UAV. The figure emphasizes how the algorithm integrates concepts from both the 3DVFH+ and VFH* algorithms, incorporating a novel memory strategy.

2.2 Geometrics methods

2.2.1 Collision Cone

The first geometric method discussed is the Collision Cone approach, described in [17]; it distinguishes itself by relying on direct velocity information rather than position data to predict and avoid potential collisions. The basic working principle of the collision cone involves identifying a conic area in geometric space around an object where a collision is most likely to occur. The vertex of this cone is typically located at the robot's current position. This cone represents the range of velocities that, if the robot were to maintain, would inevitably lead to a collision with an obstacle. Once a potential collision is detected because the robot's velocity vector lies within this cone, evasive manoeuvres are executed by directing the UAV's velocity vector outside the collision cone region. These manoeuvres can involve changing speed, heading direction, or both, depending on the robot's capabilities and time constraints.

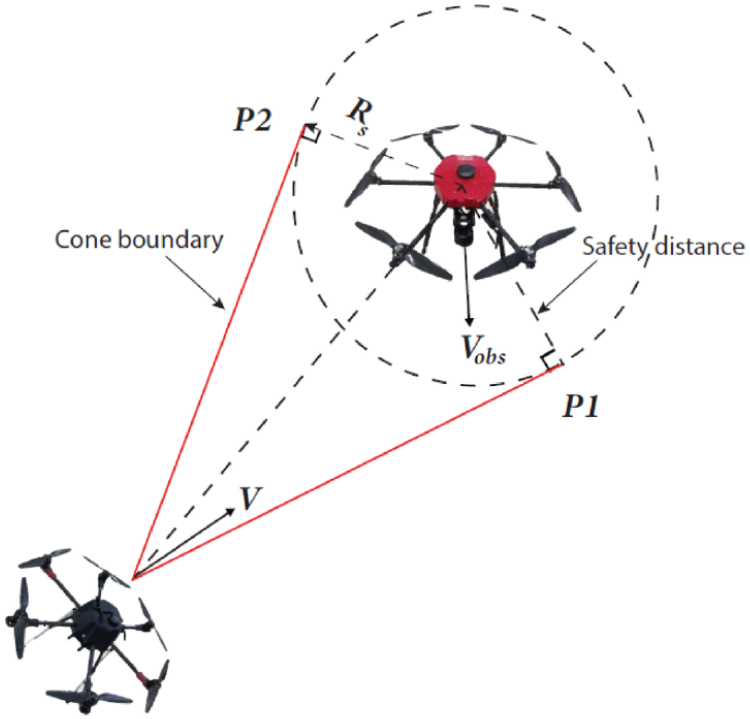

Figure 2.5: Collision Cone construction from [4]

A significant advantage of the collision cone approach is its applicability to dynamic environments with obstacles that have unpredictable trajectories and arbitrary shapes, a notable relaxation from limitations in other obstacle avoidance literature. The method's roots in well-established aerospace guidance theory, particularly for interception problems, contribute to its robustness and provide a strong theoretical foundation. Furthermore, it has low computational requirements for its core implementation and

has been extended to manage multiple obstacles and multi-UAV formations or swarms, offering a solution for complex multi-agent navigation problems. Despite its strengths, the collision cone approach has some limitations. For irregularly shaped objects, determining the precise angle required for computation can pose practical difficulties, and approximating these shapes with simpler forms, such as circles, results in an inexact collision cone compared to a theoretically exact one. Additionally, the fundamental method, particularly as presented in earlier works, does not inherently account for real-world complexities such as communication delays between agents, sensor noise, or environmental factors like wind and turbulence, especially when applied to multi-UAV systems. The accuracy of the system can also be critically dependent on the precision of sensing technologies, such as LiDAR, and filtering techniques. Some older collision cone techniques also did not initially consider time as a constraint for collision avoidance manoeuvres.

2.2.2 Velocity Obstacle

Another geometric collision avoidance algorithm is the Velocity Obstacle (VO), presented in [18], similar to the Collision Cone method, which can be considered a first-order model because it directly utilises velocity information to determine potential collisions. It maps the dynamic environment into the robot's velocity space. For a robot (actor A) and an obstacle (actor B), the first step is to conceptualize the obstacle in the robot's configuration space by treating the robot as a point and enlarging the obstacle by the robot's radius.

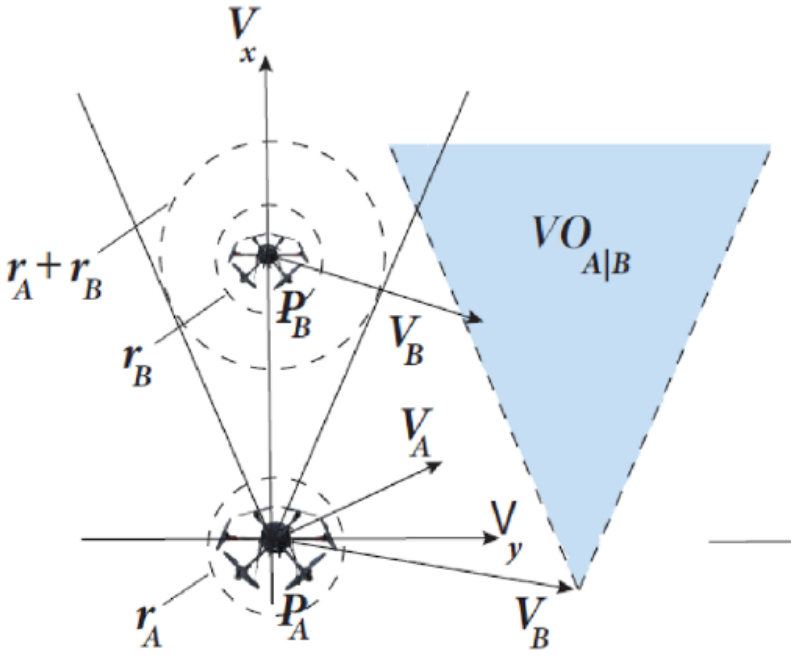

Figure 2.6: Velocity obstacle of UAV A induced by obstacle B [4]

A collision cone is then defined, representing the set of relative velocities between the robot and the obstacle that would result in a collision. Any relative velocity within this cone signifies a potential collision. To establish the velocity obstacle (VO), this collision cone is then translated by the absolute velocity of the obstacle V_B. This VO effectively partitions the robot's absolute velocities into those that would cause a collision and those that would avoid it. To avoid an obstacle, the robot selects a velocity that lies outside of its corresponding velocity obstacle. When multiple obstacles are present, the robot considers the union of all individual velocity obstacles. The method also incorporates a "time horizon" to focus on imminent collisions, modifying the VO to exclude collisions predicted to occur beyond this horizon. It computes a set of reachable avoidance velocities (RAV) by taking the velocities the robot can feasibly achieve given its constraints and subtracting the velocity obstacles, thus **ensuring both collision avoidance and dynamic feasibility**. A complete trajectory from start to goal is built as a sequence of these avoidance manoeuvres, computed at discrete time intervals.

A significant strength is its ability to unify the avoidance of both moving and stationary obstacles and to handle any number of moving obstacles by considering the union of their individual VOs. Crucially, it allows for the direct consideration of robot dynamics and actuator constraints by restricting the choice of avoidance velocities to those reachable by the robot's admissible accelerations.

Despite its strengths, the velocity obstacle approach has certain limitations. As a first-order method, it does not integrate velocities to yield positions as functions of time. This means its predictions about potential collisions rely on the assumption that the obstacle maintains its current shape and speed. Consequently, using the VO for predicting remote collisions may be inaccurate if the obstacle's actual trajectory is not a straight line, as the VO is based on a linear approximation of the obstacle's path. Furthermore, the method tends to generate conservative trajectories. Each segment of the path is designed to avoid all obstacles that are reachable within the specified time horizon, potentially excluding other feasible trajectories that involve avoiding an obstacle at a later point.

2.3 Repulsive force-based methods

Repulsive force-based algorithms are fundamentally based on the concept of modelling the drone's operational area as a potential field. Within this field, the target is conceptualized as generating an attractive force that pulls the drone towards its desired location. Conversely, any obstacles present in the environment create a repulsive force, which pushes the drone away from them. This mechanism resembles the interaction between electric charges, where the target functions as a positive charge, creating a gravitational field, while obstacles act like negative charges, producing repulsive fields. The cumulative sum of these attractive and repulsive forces determines the drone's subsequent movement or velocity [5].

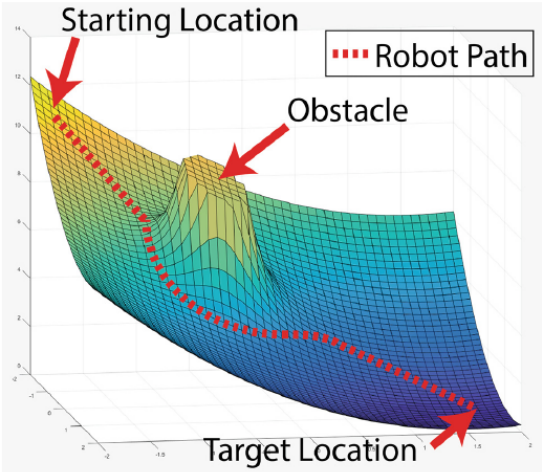

Figure 2.7: Example of potential field [5]

A significant advantage of potential field methods is their simplicity, elegance, and low computational demands, which enable real-time performance and make them a frequently used method for navigation. This simplicity also makes them easy to implement, contributing to a smooth path-generation process.

Despite their utility, traditional repulsive force-based algorithms have inherent limitations, especially in complex or cluttered environments. One major issue is the **local minima problem**, where the drone can become trapped in a position where opposing forces balance each other, resulting in a zero resultant force and preventing it from reaching the actual target. Another common limitation is the **Goal Not Reachable with Obstacles Nearby (GNRON)** problem, where the drone cannot approach a target if it is too close to an obstacle because the obstacle's strong repulsive force overrides the attractive force to the target. This occurs because the repulsive force increases significantly as the drone gets very close to an obstacle. In contrast, the attractive force may decrease as it approaches the goal, leading to repulsion dominating and pushing

the drone away from its intended destination. To address these shortcomings, extensive research has been conducted, including modifying repulsive force functions, incorporating past actions, introducing virtual target points [19], or employing methods like the regular hexagon-guided (RHG) method [20].

2.4 AI-based methods

AI-based methods for Unmanned UAV obstacle avoidance are categorised by their learning techniques, primarily into supervised and unsupervised learning algorithms. These methods leverage machine learning (ML) and Deep Reinforcement Learning (DRL) to process and analyze the large volumes of data generated by UAV sensing systems. Advancements in cloud computing, Graphics Processing Units (GPUs), and parallel computing have made it computationally feasible for these AI algorithms to process extensive data in real-time. Deep Reinforcement Learning, specifically, combines reinforced learning with deep neural networks, allowing a UAV to learn actions based on feedback in the form of rewards or penalties. A significant advantage of DRL is its suitability for applications that do not inherently have a pre-existing training dataset, as decisions are made based on real-time feedback from the environment. For instance, one DRL method utilises recurrent neural networks with temporal attention to retain relevant information from a monocular camera, thereby enabling better future navigation decisions in unknown indoor environments. Machine learning algorithms, on the other hand, are trained on images and sensor datasets to identify potential obstacles and hazards in the environment. The quality and size of the training dataset significantly impact their performance. Despite their advantages, AI-based methods come with limitations. Machine learning algorithms can be computationally expensive as their performance hinges on the scale and quality of the training dataset. DRL methods, while powerful, have limited applicability in some safety-critical environments because the UAV may need to repeatedly operate in the same workspace to collect sufficient data for effective learning.

The field of AI-based obstacle avoidance techniques is still considered to be in its infancy, indicating that ongoing research is needed to address challenges such as optimising energy consumption in AI models and exploring low-power hardware solutions for extended operational endurance in small UAVs. Due to these maturity and resource constraints, AI-based approaches were not actively considered for this thesis work.

Chapter 3

Background

3.1 Environment Perception

Perception in robotics refers to a system's ability to sense, process, and understand its surrounding environment, which is crucial for autonomous navigation and decision-making. For UAVs, perception sensors are a critical component of low-altitude navigation systems, enabling intelligent, real-time decisions such as navigation, path optimisation, localisation, motion planning, and, most critically, obstacle avoidance.

Perception sensors for UAV autonomous navigation are broadly categorised into visual and non-visual. Non-visual sensors operate on the principle of active sensing, emitting their energy to illuminate the environment and receive reflected signals. This category includes sensors such as LiDAR, radar, and ultrasonic sensors. LiDAR (Light Detection and Ranging) is highly valued for its precise and accurate obstacle detection, working as a Time of Flight (TOF) sensor to generate 2D or 3D point cloud data representing the UAV's proximity. LiDAR performs well in adverse weather conditions; however, 3D LiDAR sensors can be constrained by higher payload and computational requirements, potentially reducing flight time in small UAVs. Radar (Radio Detection and Ranging) systems are effective in obstacle avoidance and terrain mapping, particularly advantageous in adverse weather where image-based systems may struggle. Radar determines distance by measuring the time it takes for transmitted electromagnetic waves to return and can detect relative speed. However, radar can be computationally challenging, produce false readings in cluttered environments due to multiple path reflections, and may have limited range resolution, making it less suitable for detecting small, distant obstacles. Ultrasonic sensors use sound waves to determine the distance and direction of an obstacle. They are lightweight and inexpensive, making them suitable for UAVs; however, they have a limited detection range and can be affected by ambient noise and weather conditions, which can impact their performance and limit their ability to accurately measure obstacle size and shape.

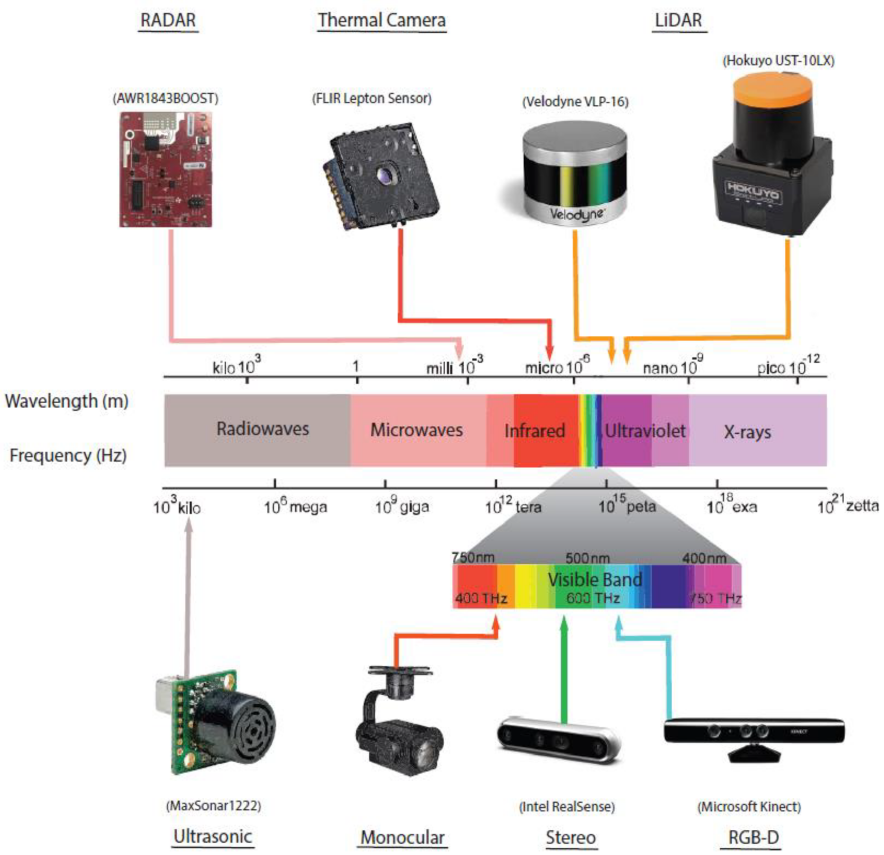

Figure 3.1: Perception sensors [4]

Visual sensing primarily uses passive sensors like cameras, capturing visual data in forms such as RGB, stereo, color infrared and thermal images. These sensors are known for their low cost, lighter payload, and wider field of view, providing significant information for obstacle detection, tracking, and depth perception. Monocular vision systems utilize a single camera to capture images and extract features, often relying on algorithms like Size Explanation Algorithm, Vanishing Point algorithm, Convolutional Neural Networks (CNN), or optical flow techniques to estimate depth from multiple frames. While low-cost and lightweight, monocular systems inherently lack direct depth perception and can be sensitive to lighting conditions. RGB-D cameras, on the other hand, capture both color and depth information simultaneously, typically using an infrared projector and camera to create a full 3D model of the scene.

Stereo vision systems are a key type of visual perception sensor that provide depth perception and spatial information, which is highly beneficial for UAVs. These systems operate by using two cameras placed a fixed distance apart to capture two images of the same scene from slightly different perspectives. By analyzing the disparity (the difference in position of a given object in the two images), stereo vision systems can compute the distance between objects. This capability allows UAVs to accurately detect and avoid obstacles in their path, thereby improving their safety and maneuver-

ability. Despite these advantages, employing stereo vision systems introduces certain challenges. They tend to add more computational cost and complexity to the hardware architecture of the UAV system. Moreover, studies have shown that while they can yield promising results, stereo vision systems exhibit vulnerability to poor calibration, particularly under varying lighting conditions, which can significantly impact their performance. The accuracy of depth estimation can also degrade as the distance to the object increases, making them more suitable for short-range obstacle detection.
Stereo vision, like other perception systems, generates a collection of 3D points that represent surfaces and objects, resulting in what is known as a point cloud.

3.2 Octomap Framework

Point clouds contain raw measurement points, but this approach is not memory-efficient. Additionally, point clouds do not distinguish between obstacle-free areas and unmapped regions, nor do they offer a method for probabilistically fusing multiple measurements. For these reasons, this thesis utilizes the Octomap Framework [6]to process and organize data obtained from a stereo camera.

The OctoMap framework is an open-source C++ library designed for generating volumetric 3D environment models, which is extensively applied in robotics applications. It is based on octrees and employs probabilistic occupancy estimation to represent occupied, free, and unknown areas within a 3D space. At its core, OctoMap utilises octrees, a hierarchical data structure for spatial subdivision in 3D space. Each node in an octree represents a cubic volume, known as a voxel, and this volume is recursively subdivided into eight sub-volumes until a predefined minimum voxel size is reached.

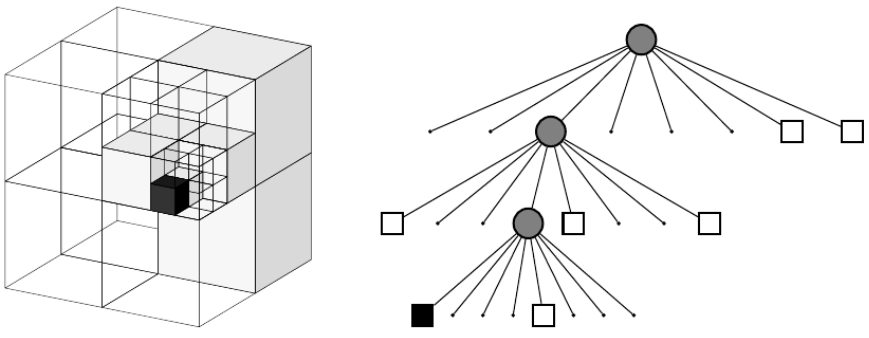

Figure 3.2: Octree spatial subdivision showing free (shaded white) and occupied (black) voxels with its volumetric representation (left) and hierarchical tree structure (right). [6]

Octrees overcome a significant drawback of fixed grid structures by delaying the initialization of map volumes until measurements are integrated, meaning the extent of the mapped environment does not need to be known beforehand, and the map only

contains measured volumes. A central property of OctoMap's approach is its ability to efficiently and probabilistically update occupied and free space. Occupied space is determined by the endpoints of distance sensors (like laser range finders), while free space corresponds to the observed area between the sensor and the endpoint. The hierarchical nature of octrees also allows for multi-resolution representations, as the tree can be queried at different depths to obtain broader subdivisions of the 3D space.

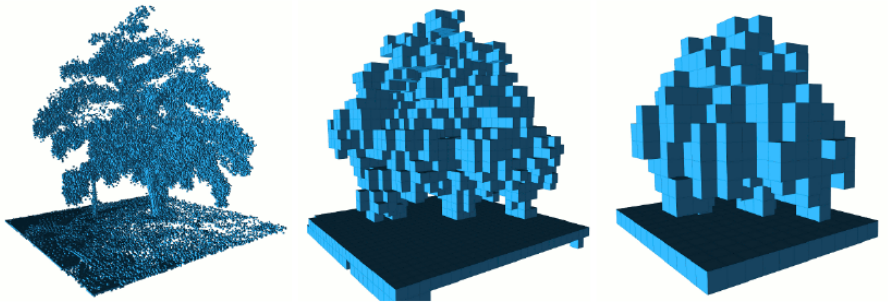

Figure 3.3: Resolution differences in Octree, 0.08m, 0.64m and 1.28m [6]

OctoMap integrates sensor readings using probabilistic occupancy grid mapping. This approach estimates the probability of a leaf node being occupied based on current and previous sensor measurements using a recursive update formula. Furthermore, OctoMap addresses the memory efficiency problem by utilising an octree map compression method. The framework introduces a compression method that locally combines coherent map volumes in both mapped free areas and occupied space, significantly reducing memory requirements. Compared to traditional 3D grids, which have large memory requirements and require their extent to be known beforehand, OctoMap's octree-based approach is much more compact, especially for large environments or fine resolutions. Implementation details, such as avoiding the explicit storage of node coordinates and voxel size (which can be reconstructed during navigation) and using a single child pointer to an array of eight pointers for inner nodes, further minimise memory overhead. For instance, leaf nodes only store mapping data and a null pointer, while inner nodes additionally store eight pointers. In practice, this design can save 60-65% memory compared to allocating eight pointers for each node.

Another key advantage of the Octomap Framework is its accessibility; it offers a variety of well-structured functionalities for data manipulation. When integrating individual range measurements, the `insertRay()` method is called (`insertScan()` for batch point cloud). This operation updates the endpoint of the measurement as occupied, while all other voxels along the ray from the sensor origin to the endpoint are updated as free. Ray intersection queries, executed through the `castRay()` method, are fundamental in robotics applications and hold particular significance for this thesis

work. This method allows for the simulation of "firing" a ray from the sensor's origin in a specified direction. The ray proceeds until it either intersects with an occupied volume or reaches a predefined maximum distance, effectively returning information on whether it encountered any occupied cells along its path.

3.3 Spiral-Based Escape Algorithm

After a deep-dive into the reviewed literature proposal, the algorithm that will be implemented in this thesis work is the Efficient Reactive Obstacle Avoidance Using Spirals for Escape algorithm, developed by Azevedo et al. [7]. This solution provides a reactive approach to collision avoidance, distinguishing itself by its focus on simplicity, robustness and efficiency in responding to unforeseen obstacles. The original article details implementations for both CPU and GPU map representations, enabling broader coverage of potential applications. However, within the scope of this thesis, the focus will be exclusively on the CPU-based implementation of the algorithm. The core pipeline of this algorithm involves three main stages:

- **Environment Perception and Representation:** Data from perception sensors, specifically a depth camera, is acquired and transformed into a global reference frame. For the CPU-based implementation, this environment is then efficiently represented using Octomaps, which provide an occupancy grid based on the octree data structure. The map insertion process in the CPU approach is optimized by directly accessing octree keys without fully converting occupied nodes into 3D Euclidean space until necessary.

- **Obstacle Detection:** The algorithm identifies threatening obstacles by checking for objects that intersect a cylindrical safety volume around the UAV. In the CPU approach, this cylindrical volume is geometrically approximated by firing n rays along the drone-waypoint direction, leveraging Octomap's efficient ray cast implementation. This search begins from the center of the safety volume and returns the nearest intersecting obstacle.

- **Avoidance Path Calculation:** Upon obstacle detection, the algorithm computes an escape trajectory. This is achieved by searching for valid escape points along an Archimedean spiral centred on the closest threatening object. This spiral is chosen for its continuous function, which moves away from the origin with linear angle growth, providing an approach that prioritises paths that minimally affect the original desired trajectory. The validity of an escape point is confirmed by ensuring a clear path from the UAV's current position to the escape point and then from the escape point towards the final goal along a predefined distance. If

no viable alternative path is found within the maximum number of iterations, a recovery procedure is triggered, instructing the drone to navigate to the previous waypoint to re-evaluate the problematic area with more comprehensive environmental knowledge.

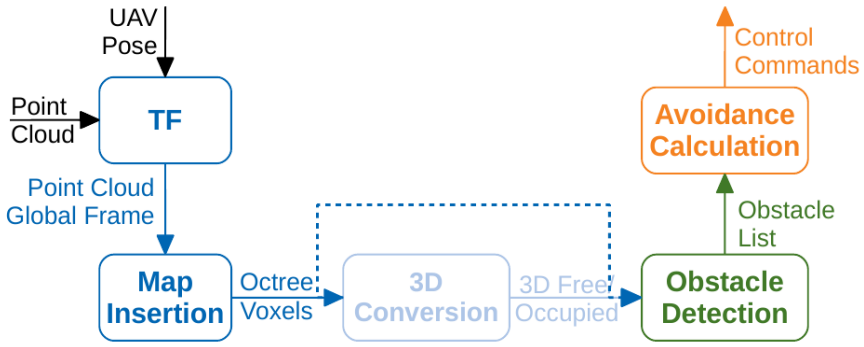

Figure 3.4: General algorithm pipeline [7]

The figure 3.4 presents a flowchart depicting the high-level pipeline, showcasing the flow of data and processes from initial sensor input to the generation of control commands. For each iteration, the process starts with Point Cloud data and the UAV Pose serving as initial inputs. This raw point cloud data, acquired from a depth camera, is first subjected to a transform operation to convert it into a global reference frame, ensuring the coherence of the data over time. Following the transformation, the data proceeds to the "Map Insertion" stage. For the CPU-based implementation, this step utilises the Octomap framework specifically to represent the environment. A distinguishing feature highlighted in fig. 3.4, through the explicit bypassing of a full "3D Conversion" block, is a crucial optimization: to accelerate processing, the map insertion process in the CPU approach neglects the conversion of occupied nodes into 3D Euclidean space at this early stage. Instead, the "Obstacle Detection" phase directly accesses the octree keys, performing the inverse conversion of 3D points only when necessary during the evaluation process.

The following central stage, shown in green, is "Obstacle Detection". Here, the algorithm identifies potentially threatening obstacles by checking for objects that intersect a cylindrical safety volume around the UAV. The outcome of this detection phase is an "Obstacle List".

Finally, if an obstacle is detected, the pipeline proceeds to the orange-coloured "Avoidance Calculation" stage. This module is responsible for computing an escape trajectory by searching for valid escape points along an Archimedean spiral. This spiral is specifically centred on the closest threatening object. The ultimate output of this entire pipeline is "Control Commands" that direct the UAV to execute the necessary

avoidance manoeuvre. It is essential to note that the low-level specifics of UAV control are considered beyond the scope of this particular work and are executed using the PX4 flight stack.

3.3.1 Obstacle detection

The method employed for obstacle detection is based on earlier works by Hrabar, and Azevedo et al. [11], [21], which involves implementing a cylindrical safety volume around the UAV to check for such threats. To leverage the Octomap's highly efficient raycast implementation, the cylinder is approximated by firing a series of n rays (dots in fig. 3.5) along the drone's intended path, specifically in the drone-waypoint direction.

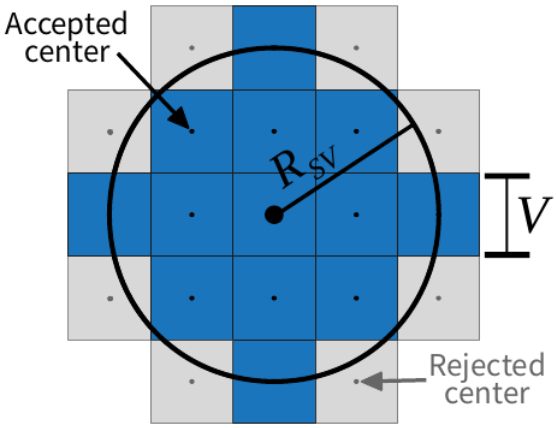

Figure 3.5: Cross section example [7]

The number of these rays is determined by the desired safety radius (R_{SV}) and the voxel size (V) of the Octomap representation. These rays are positioned in parallel and are spaced by V to ensure comprehensive coverage of all internal voxels within the designated safety volume. Although the external part of the cylindrical volume may not be entirely covered, any resulting tolerance is considered negligible, as the vehicle's subsequent motion will eventually encompass these areas. For a center calculation involving i horizontal steps and j vertical steps, a candidate center is deemed acceptable for firing a ray only if the geometric condition, given by Equation 3.1 , is met:

$$\sqrt{(i \cdot V)^2 + (j \cdot V)^2} \leq R_{SV} \Leftrightarrow \sqrt{i^2 + j^2} \cdot V \leq R_{SV} \tag{3.1}$$

Once validated, the starting positions of these rays, denoted as $\text{ray}_{\text{center}_{i,j}}$, are calculated relative to the UAV's current position r and the goal position p, using the normalized direction $d = \frac{p-r}{||p-r||}$ as per Equation 3.2:

$$\text{ray}_{\text{center}_{i,j}} = r + (i \cdot d_y, -i \cdot d_x, j) \cdot V, \quad \text{where } i, j \in \mathbb{Z} \tag{3.2}$$

The search for an obstacle with this ray-firing method initiates from the center of the safety volume and expands outwards up to its defined radius limit. The algorithm is designed to return the nearest obstacle that intersects this volume. The precise length of the cylinder, L, is dynamically calculated as the minimum value between a predefined maximum search range (L_{search}) and the sum of the distance to the current waypoint and the safety radius, as described in Equation 3.3:

$$L = \min(L_{\text{search}}, \|\mathbf{p} - \mathbf{r}\| + R_{SV}) \tag{3.3}$$

3.3.2 Avoidance path calculation

The avoidance calculation stage represents the crucial phase where the collision avoidance algorithm determines a safe path to circumvent detected obstacles. It leverages the closest identified object to compute a reactive avoidance trajectory.

The core of this process involves searching for a valid escape point, a method inspired by earlier research [11], [21], but distinctively employing an Archimedean spiral instead of an ellipse. The Archimedean spiral is chosen due to its continuous function that systematically moves away from its origin with a linear angle growth. This characteristic ensures that the search prioritises avoidance paths that deviate minimally from the UAV's original desired trajectory. The first valid point found along this spiral is considered the most optimal in terms of least deviation, facilitating collision-free operation.

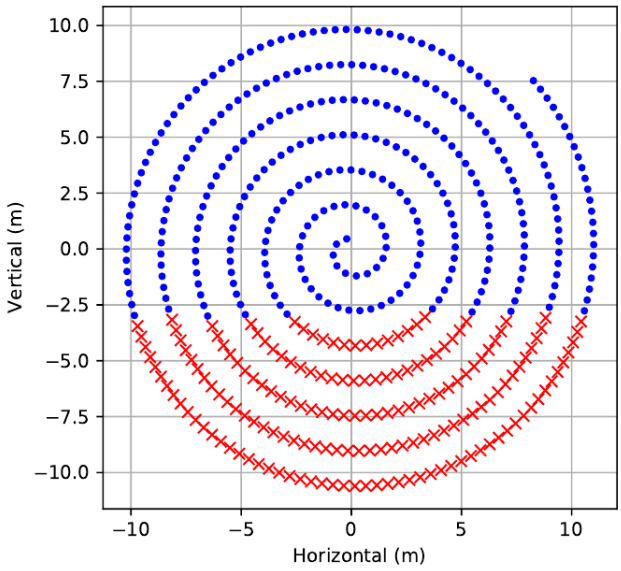

Figure 3.6: Archimedean spiral example [7]

When an obstacle, denoted as $o = (o_x, o_y, o_z)$, is detected, the spiral search for escape points starts. This spiral is centred on the obstacle. The search follows a

horizontal direction that is orthogonal to the UAV's current drone-waypoint direction, $d = \frac{p-r}{||p-r||}$, where r is the current UAV position and p is the goal position. The candidate escape points, denoted as $e = (e_x, e_y, e_z)$, are calculated using the following set of equations 3.4:

$$\begin{aligned} \text{radius}_{\text{hor}} &= a \cdot \theta \cdot \cos(\theta) \\ e_x &= o_x + \text{radius}_{\text{hor}} \cdot d_y \\ e_y &= o_y - \text{radius}_{\text{hor}} \cdot d_x \\ e_z &= o_z + a \cdot \theta \cdot \sin(\theta) \end{aligned} \tag{3.4}$$

A critical aspect of this calculation is the rejection of candidate escape points that would lead to very low-altitude flights. Specifically, if $e_z - o_z < -3m$, the candidate point is immediately discarded, and the algorithm continues its search. This preference for horizontal avoidance or altitude increase arises from the observation that the probability of encountering obstacles tends to rise with decreasing altitude. In the implemented approach, the constant arc length (l_{arc}) used was equal to the voxel size (V), and the winding separation (w) was set to $\pi \cdot V$, which implies that the spiral parameter $a = V/2$. As the spiral's radius increases, the angle step required to maintain a constant arc length naturally decreases. An iterative search approach is employed, where the current angle θ can be approximated based on the previous angle θ_{prev} by solving Equation 3.5:

$$\theta = \sqrt{\left(\frac{V}{2} \cdot \theta_{\text{prev}}\right)^2 + V^2} \cdot \frac{2}{V} = 2 \cdot \sqrt{\frac{\theta_{\text{prev}}^2}{4} + 1} \tag{3.5}$$

Once a candidate escape point e is generated, the algorithm performs a validation procedure. This involves re-applying the obstacle detection process to verify that the path is clear. This verification encompasses two segments: from the current UAV position to the candidate escape point and then from the escape point to the final goal point along a predefined length (e.g., 10 m). If the escape point satisfies these collision-free conditions, it is deemed valid and transmitted to the UAV as an intermediate goal point.

Should no valid escape point be found after iterating through a maximum number of candidates, a recovery procedure is initiated. In this scenario, the drone is instructed to navigate back to its previous waypoint, effectively gaining more knowledge about the environment before attempting to navigate the problematic area again. The UAV maintains a record of its waypoints to facilitate this recovery.

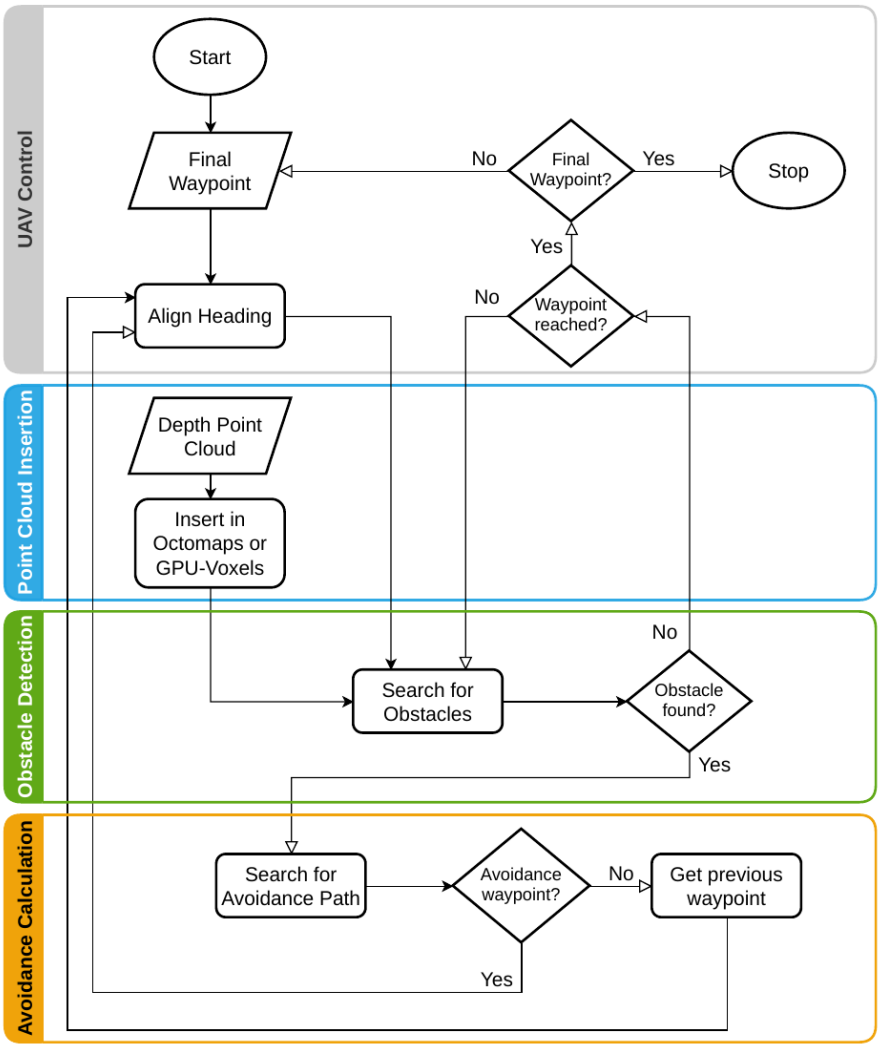

Figure 3.7: Algorithm flowchart [7]

Chapter 4

System Architecture and Design

4.1 Overall System Architecture

At the core of the autonomous UAV obstacle avoidance system are four distinct software components that collaborate through established communication protocols. This architecture illustrated in fig. 4.1 is characterized by a distributed design, where each primary actor plays a specialized role. The **Gazebo Simulator** is a powerful 3D environment that provides physics-based environmental simulation and realistic sensor models, including point cloud data from the stereo camera of the simulated environment. **ROS 2** (Robot Operating System 2) is responsible for processing perceived data, executing core avoidance algorithms, and commanding action, which is performed at a lower level by the **PX4** flight stack. Finally, the **Ground Control Station** (GCS) provides essential human oversight and intervention capabilities, allowing to monitor vehicle status, adjust parameters, and take manual control when necessary, often *being a prerequisite for arming the vehicle in simulation.*

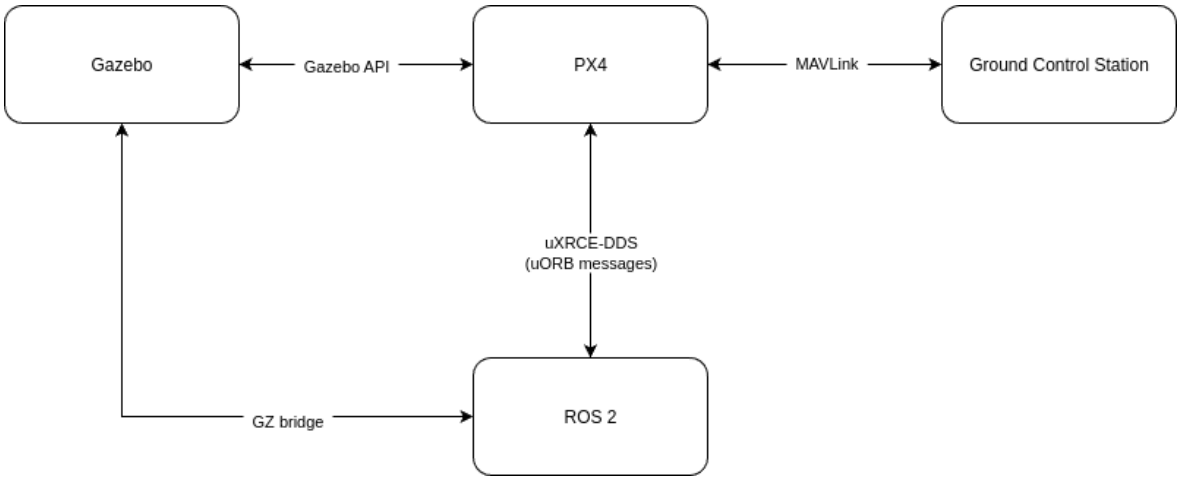

Figure 4.1: General communication architecture

The inter-component communication relies on three fundamental protocols. In dy-

namics simulation, Gazebo directly interfaces with PX4 using its native API, instead of MAVLink, for the core exchange of sensor data and actuator commands, unlike other simulators. Real-time command and telemetry exchange between ROS 2 and PX4 is facilitated by the uXRCE-DDS middleware, which bridges PX4's internal uORB messaging system with the ROS 2 environment. This uXRCE-DDS architecture involves a client running on PX4 and an agent running on the companion computer (which hosts ROS 2), enabling bi-directional data flow typically over serial or UDP links. Furthermore, to handle high-bandwidth data streams such as point clouds and camera imagery, a dedicated Gazebo-ROS 2 Bridge (implemented via the `ros_gz_bridge` package) allows for direct sensor data flow from the simulated environment to ROS 2's perception pipeline, thereby bypassing the flight controller.

This modular architecture ensures that each component can operate independently while maintaining real-time performance requirements critical for obstacle avoidance applications. The design also facilitates seamless transition from simulation-based development to real hardware deployment, as the same communication interfaces are maintained across both environments.

4.2 ROS 2 Framework

ROS 2 (Robot Operating System 2) is a modern, open-source middleware framework designed for developing distributed robotic applications. It serves as the communication backbone that enables complex robotic systems to coordinate multiple software components in real-time. The framework operates through individual software components called nodes that communicate through well-defined interfaces. Each node represents an independent process responsible for specific functionality. For example, in this thesis work, one node might manage the overall navigation algorithm, while another node focuses exclusively on processing point cloud data, detecting obstacles, and generating escape points. This distributed architecture allows the decomposition of complex systems into manageable, testable components that can be developed independently.

Communication in ROS 2 occurs through three primary patterns. Topics provide asynchronous message passing for continuous data streams, such as sensor readings. Services enable synchronous request-response interactions for immediate queries, and actions support long-running operations with feedback capabilities. One of ROS 2's key advantages over its predecessor (ROS 1, utilised in [7]) is real-time performance capability. The framework supports deterministic communication patterns essential for time-sensitive applications, allowing developers to specify message priorities and latency constraints. This is particularly crucial for UAV obstacle avoidance, where

delayed commands could result in collisions.

4.2.1 Node Architecture

The node architecture consists of three main ROS 2 nodes that communicate with each other through topics, as illustrated in fig. 4.2. Point cloud data is sourced from the simulated environment by leveraging the `ros_gz_bridge`, while drone information is retrieved from PX4 using the uXRCE-DDS middleware, which facilitates communication between ROS 2 and PX4 over ROS 2 topics.

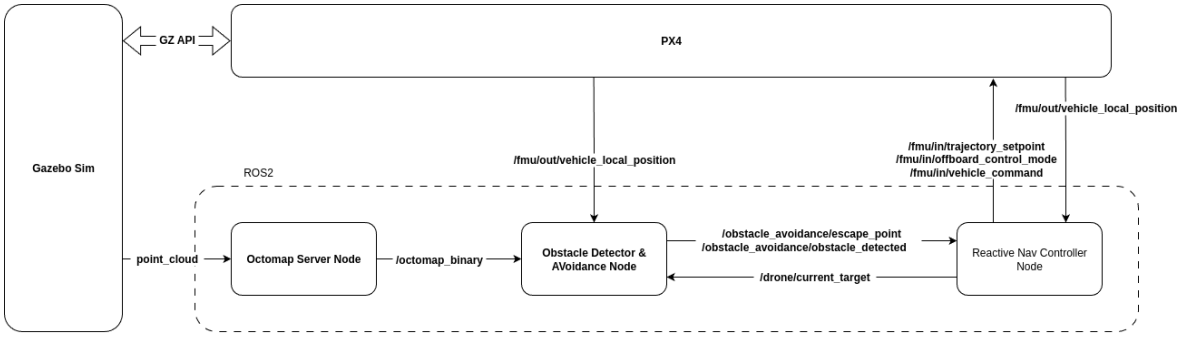

Figure 4.2: Nodes architecture

The **Octomap Server Node** is responsible for constructing and maintaining the Octree, providing a variety of information, including a three-dimensional representation of the surrounding space through the topic `/octomap_binary`.

The **Obstacle Detector and Avoidance Node** is responsible for maintaining an internal octree representation by processing data retrieved from the `/octomap_binary` topic. Using the octree information and the drone's pose provided by PX4 through the `/fmu/out/vehicle_local_position` topic, this node performs obstacle detection using the raycasting method described in 3.2. If an obstacle is detected, the node computes an avoidance manoeuvre based on the Archimedean spiral highlighted in 3.3.2. Additionally, the node communicates the retrieved information to the Reactive Navigation Controller Node via the `/obstacle_avoidance/escape_point` and `/obstacle_avoidance/obstacle_detected` topics. It also offers debugging features that can be accessed through RVIZ2, which is the primary visualization tool provided by the ROS 2 framework.

The last node, the **Reactive Nav Controller Node**, acts as the high-level mission coordinator and flight control interface for the UAV system. Its primary function is to manage the UAV's navigation state machine while also maintaining a list of target positions that guide the UAV through each desired waypoint. Additionally, it is responsible for requesting control actions through `/fmu/in/trajectory_setpoint` topic,

arming and disarming the UAV, requesting offboard control mode, and publishing the heartbeat required by PX4 through the topics `/fmu/in/offboard_control_mode` and `/fmu/in/vehicle_command`.

4.3 PX4 Autopilot

PX4 is an open-source autopilot flight stack that functions as the central control system for various unmanned robotic vehicles, including multicopters, fixed-wing aircraft, Vertical Takeoff and Landing (VTOL) vehicles, ground vehicles (rovers), and underwater vehicles. It operates on a real-time operating system (RTOS), such as NuttX. It provides essential capabilities, including stabilisation, safety features, pilot assistance for manual control, and the automation of tasks like takeoff, landing, and mission execution. A key aspect of PX4 is its strong integration with companion computers and robotics APIs, particularly ROS 2 and MAVSDK.A comprehensive PX4 system architecture is presented in fig. 4.3.

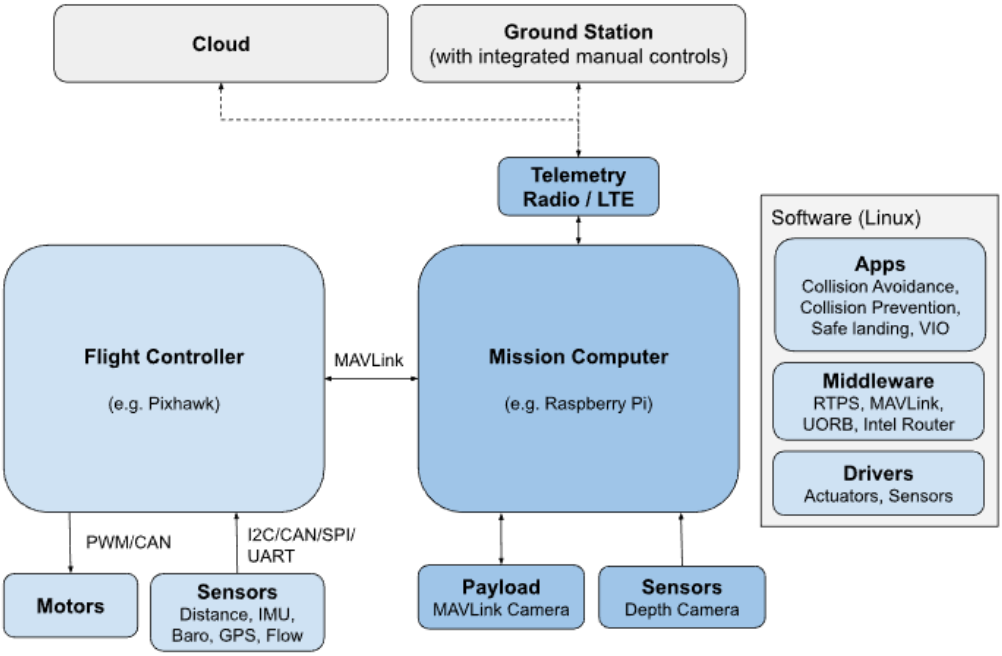

Figure 4.3: PX4 architecture [8]

At its core is present the flight controller running the PX4 flight stack, which can be considered as the "brain" of the unmanned vehicle. The diagram illustrates the critical hardware interconnected with the flight controller, including motors and Electronic Speed Controllers (ESCs) for vehicle movement, as well as various sensors such as GPS and IMUs that provide data for determining the vehicle's state and enabling autonomous control. A key element of this description is the companion computer, also referred to as a "mission computer" or "offboard computer," which operates alongside

the flight controller. This companion computer typically runs a Linux operating system, making it a superior platform for general software development, computer vision, networking, and other advanced features not directly handled by the flight controller. The diagram shows that the flight controller and companion computer are connected via a fast serial or IP link, usually communicating using the MAVLink protocol. Furthermore, interactions with ground control stations, such as QGroundControl, and cloud services are generally routed through the companion computer.

The key component of PX4 on which this thesis work relies is the Software In The Loop (SITL) simulation, which enables the PX4 flight code to interact with a software-based model of a vehicle within a simulated environment. In a SITL setup, the entire PX4 flight stack runs on a computer, which can be the same machine that hosts the simulator or a separate networked computer. This approach is highly effective and crucial in the PX4 context primarily because it offers a safe, rapid, and cost-efficient means to develop and thoroughly test changes to the PX4 firmware without requiring physical hardware. It also serves as an accessible entry point for new users who may not possess a real drone. PX4 SITL and the simulators operate in a lockstep fashion, ensuring they are synchronized to run at the same speed, which facilitates precise testing and debugging by allowing simulations to be run at various speeds or even paused for detailed code inspection.

4.3.1 Communication

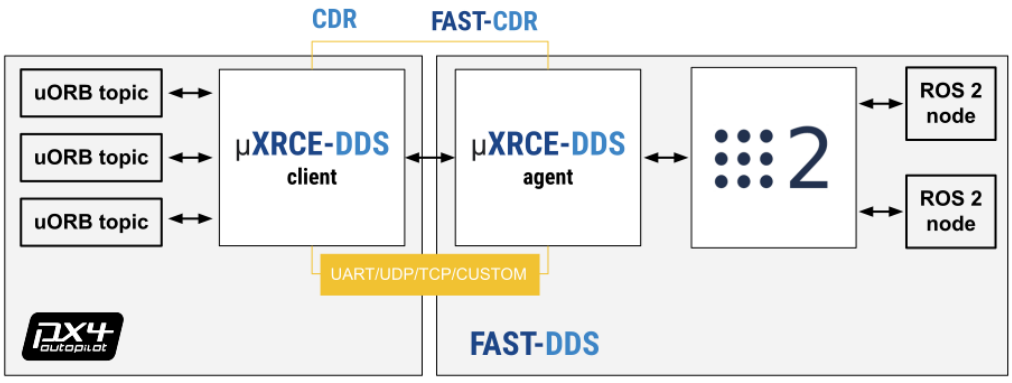

Figure 4.4: uXRCE-DDS middleware [9]

Within the PX4 autopilot system, uORB serves as the foundational asynchronous publish/subscribe messaging API for internal communication among its various software components, enabling inter-thread and inter-process data exchange. This architecture ensures that functionality is modular and reactive, with components updating instantly upon receiving new data. As previously anticipated, the uXRCE-DDS middleware acts

as the PX4-ROS 2/DDS Bridge, specifically designed to extend uORB's internal messaging to the external world, allowing uORB messages to be published and subscribed on a companion computer as if they were native ROS 2 topics.

This bridge operates through a client-agent architecture illustrated in fig. 4.4: a uXRCE-DDS client runs on the PX4 flight controller, while a uXRCE-DDS agent runs on a companion computer, communicating via serial or UDP links. The agent functions as a proxy, enabling the client to interact with the broader DDS data space where ROS 2 applications reside. The specific uORB topics exposed to ROS 2 (both publications from PX4, typically under `/fmu/out/`, and subscriptions to PX4, typically under `/fmu/in/`) are defined in the `dds_topics.yaml` file.

4.3.2 Offboard Control Mode

PX4 offers various flight modes, which are special operational states that dictate how the autopilot responds to user input and controls vehicle movement. Offboard control mode is one of the flight modes in which an external source, typically a companion computer running software like ROS 2 or MAVSDK, directly commands the vehicle's movement and attitude. This mode is fundamental for advanced applications, such as obstacle avoidance, since it allows an external system to provide high-level setpoints for position, velocity, acceleration, attitude, attitude rates, or thrust/torque rather than relying solely on PX4's internal controllers for high-level manoeuvres. In this thesis context, the output of the avoidance calculation and waypoint navigation is obtained by imposing a trajectory waypoint to reach, which is possible if the offboard mode is selected.

To request and maintain offboard control, PX4 requires a continuous "proof of life" signal from the external controller, which must be received at a rate of at least 2 Hz. This signal is conveyed either by streaming any of the supported MAVLink setpoint messages or, when using ROS 2, by continuously publishing the OffboardControlMode message. PX4 will only enable offboard control after receiving this signal for more than one second. Critically, if this signal ceases, PX4 will automatically regain control and switch to a predefined failsafe action, such as landing, after a configurable timeout.

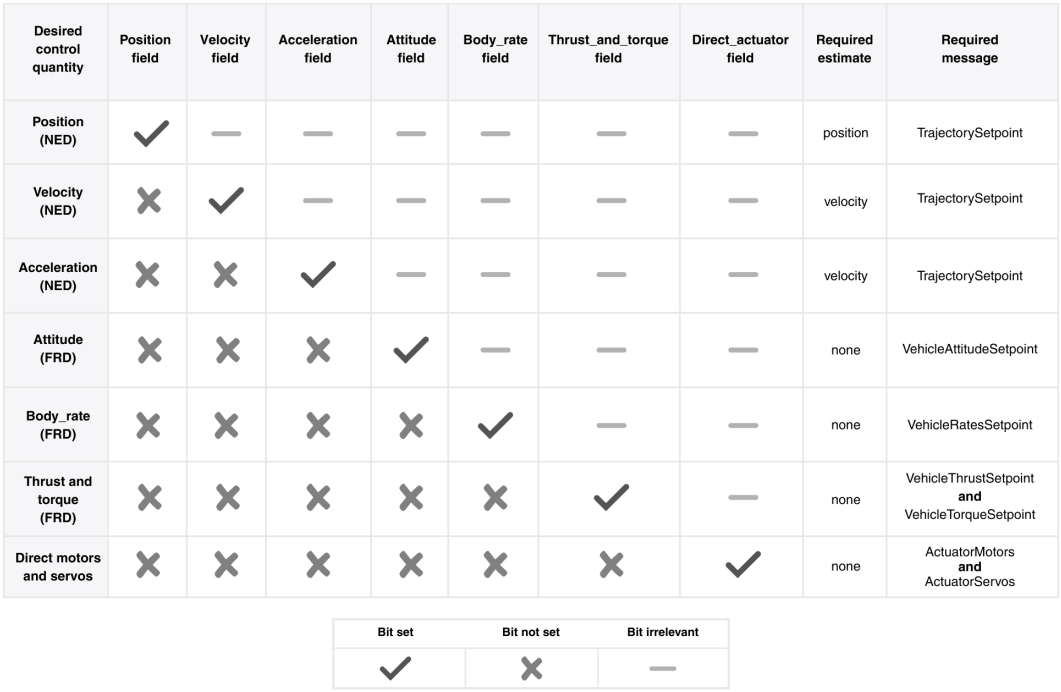

Figure 4.5: Offboard control mode

The OffboardControlMode ROS 2 message itself plays a crucial role beyond simply serving as a heartbeat. Its fields, as illustrated in fig. 4.5, are prioritised (from top to bottom) to dictate the level of the PX4 control architecture at which external setpoints are injected: position, velocity, acceleration, attitude, body rate, thrust and torque, and direct actuator. The first field with a non-zero value determines which valid estimate (e.g., position or velocity) is required by PX4 and which specific setpoint message (e.g., **TrajectorySetpoint**, VehicleAttitudeSetpo int, VehicleRatesSetpoint, VehicleThrust-Setpoint, VehicleTorqueSetpoint, ActuatorMotors, ActuatorServos) should be used to provide the actual setpoints. This allows the external system to bypass and disable lower-level internal PX4 controllers effectively. For example, if the velocity field is set, PX4 expects a valid position estimate and setpoints via the TrajectorySetpoint message.

4.4 Coordinate Transformations

In the domains of PX4 and ROS 2, a crucial aspect for coherent communication and vehicle control involves understanding and converting different coordinate frame conventions. PX4 predominantly utilises the NED (North, East, Down) frame for its local/world and body frame conventions, which corresponds to FRD (Front, Right, Down) for the body frame. In this system, the X-axis points North, the Y-axis points East, and the Z-axis points Down. Conversely, ROS 2 commonly employs the ENU (East, North, Up) standard, with its body frame often being FLU (Front, Left, Up).

Here, the X-axis points East, the Y-axis points North, and the Z-axis points Up.

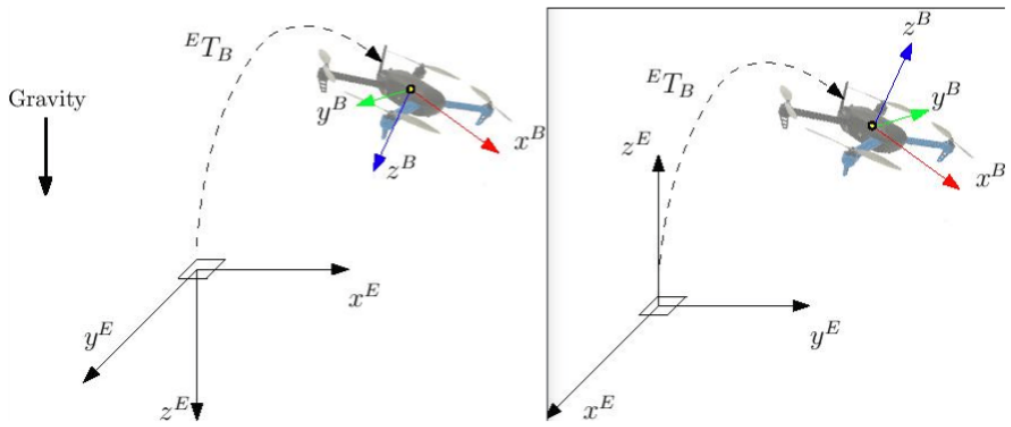

Figure 4.6: PX4 - ROS 2 frame conventions (FRD on the left/FLU on the right) [9]

The fundamental difference in these conventions necessitates explicit frame conversions when ROS 2 applications interact with PX4, as there is no implicit conversion handled automatically when topics are published or subscribed. To transform a vector from ENU to NED, two sequential rotations are performed: first, a $\pi/2$ rotation around the Z-axis (which is "Up" in the ENU frame), followed by a π rotation around the X-axis (which was the "East" direction but becomes the "North" direction after the first rotation). Conversely, to convert a vector from NED to ENU, the process involves a $\pi/2$ rotation around the Z-axis (which is "Down" in the NED frame), succeeded by a π rotation around the X-axis (which was "North" and becomes "East" after the initial rotation). It is important to note that these two resulting operations are mathematically equivalent. For body frame conversions, specifically from FLU to FRD or vice versa, a simpler π rotation around the X-axis (front) is sufficient.

These transformations are vital for various data types, including vectors found in setpoint messages and quaternions representing vehicle attitude. For example, all fields within the TrajectorySetpoint message, which define position, velocity, and acceleration, must undergo ENU to NED conversion before being transmitted to PX4. To facilitate these necessary conversions, a dedicated Python class was developed. This careful handling of coordinate frame conventions ensures the integrity and correctness of data exchanged between PX4 and ROS 2 systems for robust control and operation.

Chapter 5

Implementation Details

To achieve the objectives outlined in section 3.3, two primary ROS 2 nodes were developed and validated. The provided high-level algorithm, outlined in algorithm 1, describes the general system in which the Obstacle Detector and Avoidance Node, as well as the Reactive Navigation Controller Node, cooperate to enable autonomous obstacle avoidance for the drone. This cooperation occurs in parallel as the system runs continuously.

These core nodes interact with external components such as the PX4 Autopilot, a Depth Camera, and an Octomap Server. The Octomap Server, as specified in fig. 4.2 provides the 3D environment map to the Obstacle Detector and Avoidance Node via the `/octomap_binary` topic, which is then converted into an internal OcTree. Both the Obstacle Detector and Avoidance Node and the Reactive Nav Controller Node receive the drone's current position, velocity, and heading from the PX4 Autopilot through the `/fmu/out/vehicle_local_position` topic.

A crucial feedback loop is established in which the Reactive Navigation Controller Node publishes the drone's intended target waypoint to the Obstacle Detection and Avoidance Node via the `/drone/current_target` topic. This enables the perception system to direct its focus effectively. When an obstacle is detected, the Obstacle Detection and Avoidance Node communicates this status (True or False) to the Reactive Navigation Controller Node on the `/obstacle_avoidance/obstacle_detected` topic. If an obstacle is identified, the Obstacle Detection and Avoidance Node also publish the coordinates of a computed safe escape point (if available) on the `/obstacle_avoidance/escape_point` topic to the Reactive Navigation Controller Node. Lastly, the Reactive Navigation Controller Node sends velocity commands for flight execution to the PX4 Autopilot through the `/fmu/in/trajectory_setpoint` topic.Pseudocode related to all relevant methods of both nodes can be found in Appendix B.

Algorithm 1 High-Level System Architecture: Reactive Nav Controller Node and Obstacle Detector and Avoidance Node Cooperation

System Components:
- Obstacle Detector and Avoidance Node (Perception & Planning)
- Reactive Nav Controller Node (Navigation & Control)
- External: PX4 Autopilot, Depth Camera, Octomap Server

Communication Channels:
- `/octomap_binary` → Obstacle Detector and Avoidance
- `/fmu/out/vehicle_local_position` → Both Nodes
- `/drone/current_target` → Obstacle Detector and Avoidance
- `/obstacle_avoidance/obstacle_detected` → Reactive Nav Controller
- `/obstacle_avoidance/escape_point` → Reactive Nav Controller
- `/fmu/in/trajectory_setpoint` ← Reactive Nav Controller

while System is running **do**
  Parallel Node Execution:
  Obstacle Detector and Avoidance Node:
    Receive 3D environment map from octomap server
    Monitor drone position and current target
    Detect obstacles in safety volume
  if Obstacle detected in path **then**
      Compute escape point using spiral search
      Publish obstacle status: `True`
      Publish escape point coordinates
  else
      Publish obstacle status: `False`
  end if
  Reactive Nav Controller Node:
    Monitor vehicle status and position
    Manage waypoint mission execution
    Receive obstacle detection status
  if Obstacle status received **then**
    if Obstacle detected **then**
        Switch to AVOIDANCE mode
        Wait for escape point message
        Navigate to escape point
    else
        Continue normal MISSION mode
    end if
  end if
    Publish velocity commands to PX4
    Publish current target for perception feedback
end while

5.1 Obstacle Detector and Avoidance Node

This section presents a node responsible for processing environmental perception information provided by the Octomap Server Node and suggesting alternative paths when obstacles are encountered. Therefore, it can be considered the core algorithm implementing the concepts described in section 3.3.

The first point to address is how the environmental perception data are handled. This implementation relies on the `octomap_binary` topic provided by the Octomap Server node. The obstacle detection and avoidance node internally builds an Octree based on the information received. However, this may not be the most efficient method for processing perception data, as the onboard computer (the simulation computer in the context of this thesis) processes the Octree almost twice: first in the Octomap Server node and then again during the execution of the obstacle detection and avoidance node. This approach was chosen to enable a quick and reliable implementation of the core algorithm, which focuses more on the steps involved in obstacle detection and manoeuvre calculation.

The node continuously receives environmental data, along with the drone's current position and its target. Within a defined safety volume around the drone's projected path, the node actively detects obstacles. This detection is achieved by casting rays from the current position in the direction of the drone, following the approach shown in Figure fig. 3.5. The range of this search is determined by the parameters `safety_radius` and `search_max_length`. The safety radius is designed to ensure that the drone can physically travel in the desired direction, taking into account a safety margin to provide robustness against unknown or unpredictable noise factors. As both nodes are implemented in Python, ray casting is performed using the `castRay()` method from the C++ octomap library, accessed through a Python wrapper.

If any rays hit an occupied voxel in the octomap, an obstacle is considered detected, and a `hit_list` is generated. Upon detecting an obstacle, the node publishes True on `/obstacle_avoidance/obstacle_detected`. Subsequently, it computes a safe escape point by performing an Archimedean spiral search starting from the closest detected obstacle point as described in section 3.3.2. This search iteratively evaluates potential escape points, checking if a clear path exists to them from the drone's current origin using an additional ray cast. If a valid escape point is found within `max_iterations`, it is published on `/obstacle_avoidance/escape_point`. If no valid escape point is found after the maximum iterations, it can publish a None value, which will be managed later on by the Reactive Navigation Controller Node. The Obstacle Detector and Avoidance also provides visual markers for RViz2, which help debug and illustrate detected obstacles, the safety volume, ray casts, and the proposed escape point. If no obstacle is detected, the node publishes a false value for the `obstacle_detected`

Algorithm 2 Obstacle Detector and Avoidance Node - High Level Working Scheme

Initialize: Parameters (R_{safety}, L_{search}, V_{voxel}, N_{max_iter})
Initialize: Subscribers (Octomap, Drone Position, Drone Target)
Initialize: Publishers (Escape Point, Obstacle Status, Markers)
Initialize: Timers (Obstacle Detection, Marker Publishing)
while Node is running **do**
    Callback Processing:
    if Octomap message received **then**
        Update octree structure from binary message
    end if
    if Drone position message received **then**
        Convert NED to ENU coordinates
        Update current drone position and orientation
    end if
    if Drone target message received **then**
        Update target position
        Calculate direction vector: $\vec{d} = \frac{\vec{p}_{target} - \vec{p}_{drone}}{||\vec{p}_{target} - \vec{p}_{drone}||}$
    end if
    Obstacle Detection Timer:
    if All required data available **then**
        Clear visualization markers
        Calculate search length: $L = \min(L_{search}, ||\vec{p}_{target} - \vec{p}_{drone}|| + R_{safety})$
        Generate safety volume visualization
        ray_centers $\leftarrow$ CalculateRayCenters($\vec{p}_{drone}, \vec{d}$)
        hit_list $\leftarrow$ CastRaysFromCenters(ray_centers, $\vec{d}, L$)
        obstacle_detected $\leftarrow$ |hit_list| > 0
        Publish obstacle detection status
        if obstacle_detected **then**
            closest_hit $\leftarrow$ FindClosestHit(hit_list, $\vec{p}_{drone}$)
            escape_point $\leftarrow$ ComputeEscapePoint(closest_hit, $\vec{p}_{drone}, \vec{d}$)
            Publish escape point
        end if
    end if
    Marker Publishing Timer:
    Publish all visualization markers
end while

topic.The proposed algorithm is summarized in algorithm 2.

5.2 Reactive Navigation Controller Node

The Reactive Navigation Controller Node, handles high-level flight control, mission management, and dynamic reaction to obstacle detection. It continuously monitors the drone's vehicle status and local position from PX4 via `/fmu/out/vehicle_local_position` and `/fmu/out/vehicle_status`, ensuring the drone is armed and in offboard control mode, and sends commands to PX4 if necessary.

The node manages the execution of a predefined waypoint mission, using a state machine with DroneState values like INIT, READY, MISSION, AVOIDANCE, and LAND to govern the drone's overall behavior. It subscribes to the obstacle detection status on `/obstacle_avoidance/obstacle_detected` and escape point messages on `/obstacle_avoidance/escape_point` from the Obstacle Detector and Avoidance Node. When an obstacle is detected and an escape point is received via `escape_point_callback` the node transitions to AVOIDANCE mode. In this state, the received escape point becomes the drone's temporary target, overriding any mission waypoints to bypass the obstruction. If the escape point is deemed invalid (e.g., containing None values), the drone can switch to the LAND state. Conversely, if no obstacle is detected, the node continues in its normal MISSION mode, navigating towards the next waypoint in its list. The node calculates and publishes appropriate velocity commands to the PX4 Autopilot on `/fmu/in/trajectory_setpoint`. It also includes logic to calculate and command the drone's yaw based on its horizontal velocity, with measures to prevent rapid, unstable yaw changes. Crucially, the Reactive Navigation Controller Node constantly publishes the drone's current target (whether a mission waypoint or an avoidance point) to the Obstacle Detector and Avoidance Node via `/drone/current_target`, providing the necessary feedback for the perception system to focus its obstacle detection efforts. It also publishes its current operational state via `/drone/state`.

In contrast to what was proposed in [7], if the escape point calculation reaches the maximum number of iterations and returns none to Reactive Navigation Controller Node, the immediate action from the state machine's perspective is to transition instantly to the LAND state. This will cause the drone to halt its mission. This approach was intentionally designed to facilitate future work by integrating a global planner. In this scenario, the global planner could suggest the next course of action, whether it involves returning to the previous drone waypoint or providing a new waypoint based on updated information gathered during navigation. The proposed algorithm is summarized in algorithm 3.

Algorithm 3 Reactive Navigation Controller Node - High Level Working Scheme

Initialize: Parameters (R_{safety}, V_{max}, a_{accel}, a_{decel}, δ_{pos})
Initialize: Waypoint list $W = [w_1, w_2, ..., w_n]$
Initialize: Subscribers (Vehicle Position, Vehicle Status, Obstacle Status, Escape Point)
Initialize: Publishers (Offboard Control, Trajectory Setpoint, Vehicle Command)
Initialize: Timers (Main Control Loop, Offboard Mode)
drone_state ← INIT
while Node is running **do**
    Callback Processing:
    if Vehicle position message received **then**
        Convert NED to ENU coordinates
        Update $\vec{p}_{drone}$, $\vec{v}_{drone}$, ψ_{drone}
    end if
    if Vehicle status message received **then**
        Update arming and navigation states
        if not ready for offboard **then**
            Send arm and offboard mode commands
            drone_state ← INIT
        else
            drone_state ← READY (if was INIT)
        end if
    end if
    if Obstacle status message received **then**
        Update path_clear status
    end if
    if Escape point message received **then**
        avoidance_point ← escape_point
        drone_state ← AVOIDANCE
    end if
    Main Control Loop:
    Publish current drone state
    if $|W| = 0$ **then**
        drone_state ← LAND
    end if
    StateMachine(drone_state, W, avoidance_point)
    Offboard Mode Timer:
    Publish offboard control mode
end while

Chapter 6

Experimental Setup and Methodology

6.1 Simulation Environment

The development and validation of autonomous UAV obstacle avoidance algorithms require a robust simulation environment that accurately mimics real-world conditions while allowing for controlled and repeatable testing scenarios. Simulation-based testing offers several advantages, including risk-free algorithm development, precise control over environmental conditions, and the ability to systematically evaluate performance across varying complexity levels without hardware constraints or safety concerns.

Gazebo Simulator was chosen as the primary simulation platform due to its advanced physics engine capabilities, comprehensive sensor modelling, and native ROS 2 integration. This simulator provides a high-fidelity representation of the environment, which is essential for validating perception algorithms. Additionally, its modular architecture enables systematic testing of individual system components. The selection of Gazebo aligns with established practices in robotics research and ensures compatibility with the broader ROS 2 ecosystem used throughout the project.

This specific version of the Gazebo Simulator, Harmonic, was selected due to its proven compatibility with PX4 despite not being the recommended pairing for the ROS 2 Humble distribution. Because of this "non-default" pairing between ROS 2 and the Gazebo Simulator, it became necessary to rely on a specific `ros_gz_bridge`.

6.1.1 Vehicle Platform

The experimental platform is based on the Holybro X500 quadcopter frame, a robust and widely adopted research platform that provides an optimal balance between payload capacity, flight performance, and integration flexibility. The X500 features a 500mm motor-to-motor wheelbase with carbon fibre construction, ensuring structural rigidity while maintaining relatively lightweight characteristics essential for efficient

flight operations.

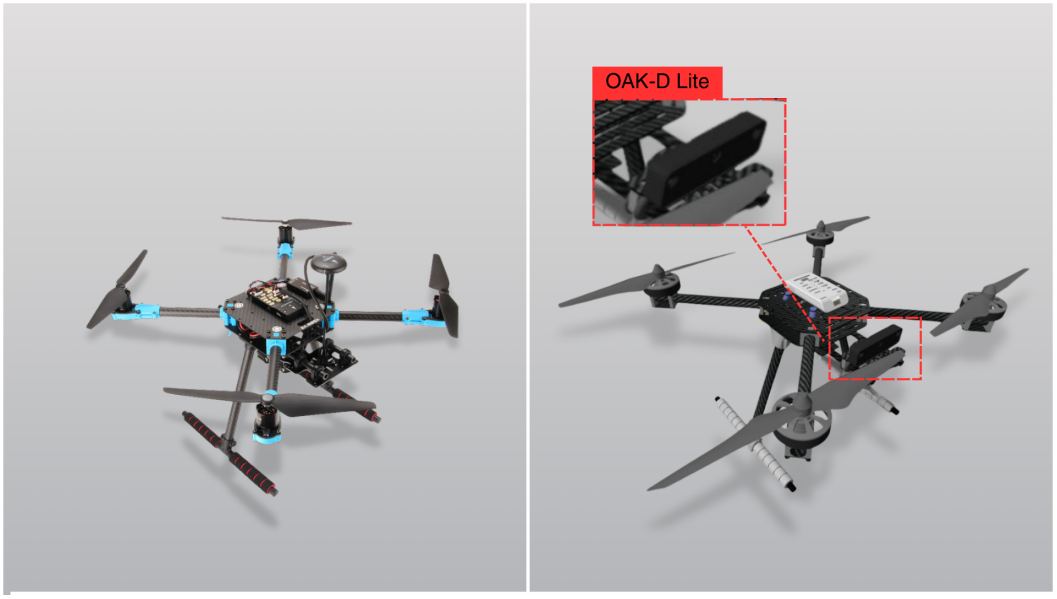

Figure 6.1: Holybro x500 (left) and its representation in Gazebo Sim (right) [10]

The frame features built-in vibration-dampening mechanisms and specific mounting points for sensor payloads, flight controllers, and companion computers, including the OAK-D Lite camera utilized in this thesis work.

6.1.2 Sensor Configuration

The perception system utilises a Luxonis OAK-D Lite camera (see fig. 6.1), which serves as a specialised tool for robotic vision by integrating a stereo depth camera and a high-resolution colour camera, along with on-device neural network and Computer Vision capabilities. This design approach enables real-time depth estimation directly on the device, thereby eliminating the need for additional computational demands on the flight computer. For depth perception, its stereo cameras feature global shutters and operate at a resolution of 480P (640x480), capable of up to 120 frames per second, while the central RGB camera offers a 13MP resolution.

Regarding its specifications, the camera is designed with a baseline separation of 75 mm between its stereo cameras, which is crucial for depth accuracy. It offers an ideal depth sensing range of 40 cm to 8 meters, extending to approximately 20 cm under extended 480P configurations. The Fixed-Focus variant of the OAK-D Lite is specifically recommended for applications with heavy vibrations, such as drones, highlighting its effectiveness in UAV obstacle avoidance. Although the stereo cameras are capable of generating 480P depth maps at high framerates, in the simulated scenario, a more conservative resolution of 640 $\times$ 480 at 30 Hz is employed for depth maps to balance detection accuracy with computational efficiency effectively.

Additionally, the OAK-D Lite incorporates an integrated BMI270 6-axis Inertial Measurement Unit (IMU), which is particularly suitable for drone applications. This IMU is comprised of a 16-bit tri-axial gyroscope and a 16-bit tri-axial accelerometer, providing precise measurements of acceleration and angular rate. This integrated IMU supports enhanced sensor fusion for improved pose estimation and serves as a complementary data source for vehicle state estimation, providing additional redundancy for attitude and motion sensing during critical obstacle avoidance manoeuvres, even though the primary navigation system relies on the PX4 flight controller's IMU.

The simulation of the sensor is managed through the configuration SDF file, which includes all the relevant information discussed earlier, along with the physical details of the sensor, such as a 3D model for visualization. Within the ROS 2 framework, camera integration requires establishing a spatial relationship between the coordinate frames of the point cloud origin (camera frame) and the world coordinate frame. This relationship is illustrated using the TF tree shown in fig. B.1.

6.1.3 World Modelling and Obstacle Placement

The design philosophy for the simulation world focuses on creating specific test environments that effectively validate obstacle avoidance capabilities while ensuring computational efficiency. Test worlds were developed to address key challenges faced by UAVs, concentrating on scenarios that assess the core functionality of the spiral-based avoidance algorithm. Each environment incorporates static obstacles that evaluate the algorithm's performance under varying levels of complexity.

Two primary test environments were created for validation purposes. The first, a simplified test world, features a simple setup with three obstacles positioned in the flight path. This controlled environment enables the verification of fundamental avoidance behaviour and the functionality of the spiral search. In this scenario, the algorithm's ability to detect obstacles, compute escape points, and execute successful avoidance manoeuvres is validated in an uncluttered setting.

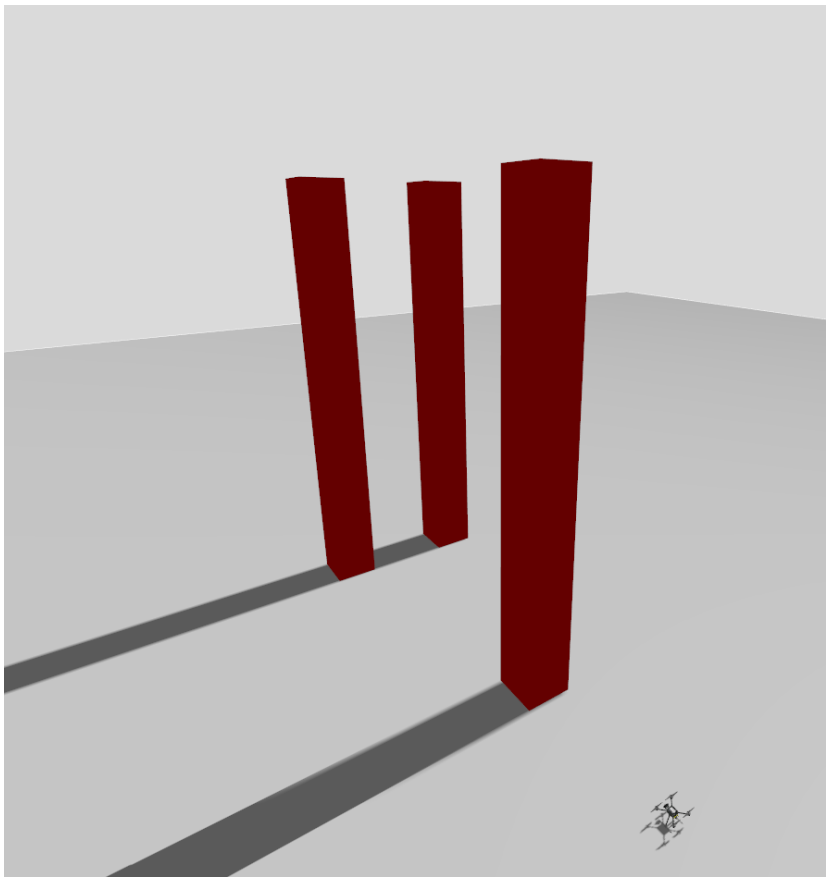

Figure 6.2: Simplified Simulated Scenario

The second environment, the cluttered test world, presents a significantly more challenging scenario than typical operational conditions. It contains multiple static obstacles of varying shapes and sizes distributed throughout the area. These include cylindrical structures that represent trees or poles, as well as rectangular blocks that simulate buildings or other structural elements. The increased obstacle density and spatial arrangement create multiple potential collision scenarios, requiring the algorithm to navigate through constrained passages and demonstrate robust avoidance capabilities.

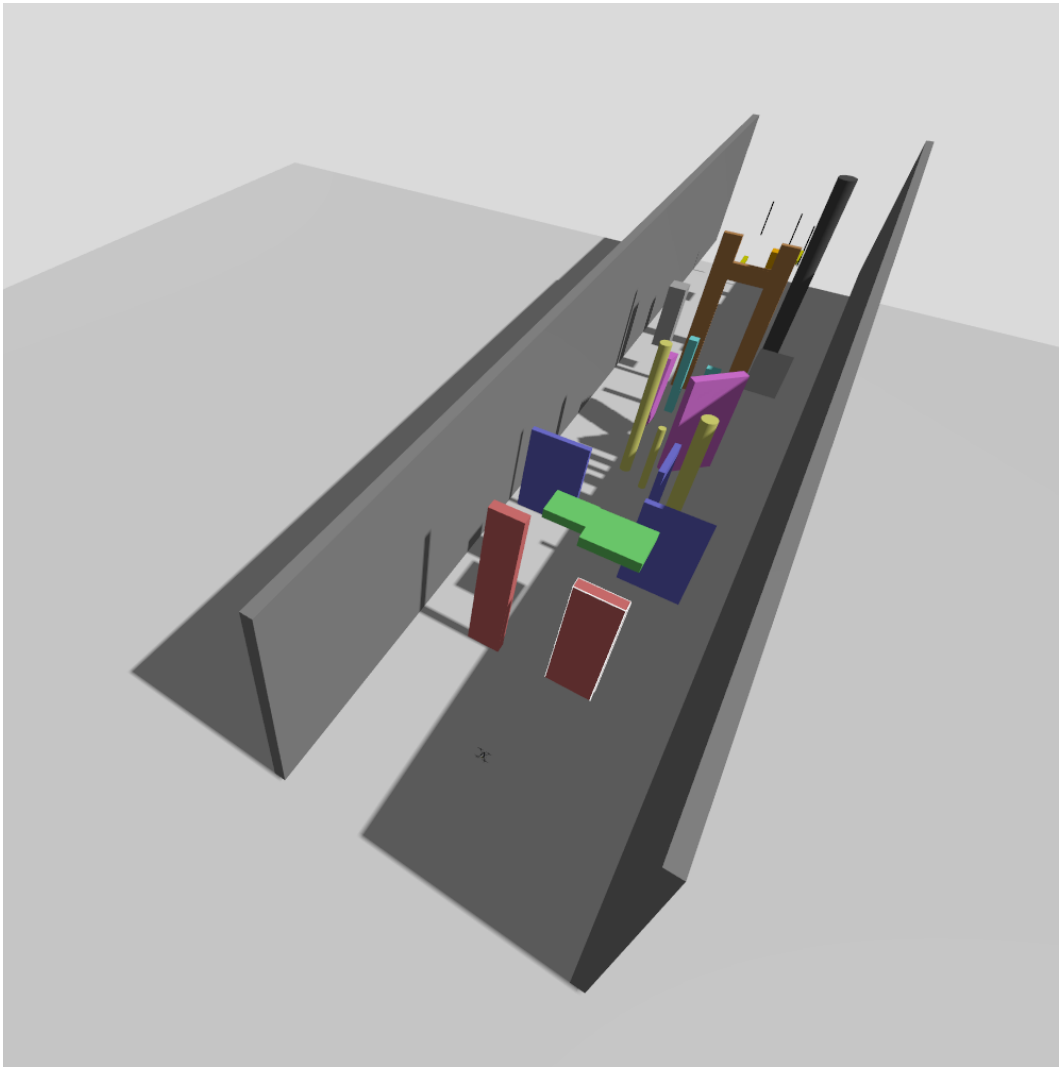

Figure 6.3: Cluttered Simulated Scenario

6.1.4 Physics Engine and Vehicle Dynamics

The Gazebo physics engine is configured with the appropriate temporal resolution and solver parameters to ensure stable and realistic vehicle dynamics simulation. The physics timestep is set to 4 ms (with a 250 Hz update rate) to strike a balance between computational efficiency and sufficient resolution for simulating the drone control system. The real-time factor is maintained at 1.0 to provide realistic timing behaviour for algorithm validation. This configuration is particularly suitable for testing obstacle avoidance algorithms, as it offers adequate resolution for the control frequencies commonly used in UAV navigation systems.

Vehicle dynamics modelling incorporates the specific mass properties, including a base frame weight plus additional payload mass for the OAK-D Lite camera and mounting systems. The vehicle model includes accurate mass distribution and inertial properties that influence flight stability and control response characteristics, ensuring

that the simulated behaviour of the vehicle closely matches the expected performance of the physical platform.

6.1.5 Development Platform Specifications

The primary development platform consists of a desktop computer equipped with an AMD Ryzen 9 7900X processor, which features 12 cores, 24 threads, and a base frequency of 4.7 GHz. It has 32 GB of DDR5 RAM and an NVIDIA GeForce GPU with 12 GB of GDDR6X memory. This configuration provides substantial computational resources, making it ideal for complex simulation scenarios, and enables detailed performance profiling and algorithm optimisation with excellent parallel processing capabilities. The development platform runs Ubuntu 22.04 LTS, offering a stable and well-supported environment for robotics development. For storage, it uses an NVMe SSD, providing high-speed I/O performance essential for data logging and processing large point clouds.

The core software environment is built on the ROS 2 Humble distribution, which acts as the middleware foundation for inter-process communication and system integration. The primary simulation platform is Gazebo Harmonic. The PX4 flight stack, version 1.15, serves as the foundation for flight control and navigation, integrated with the `px4_ros_com` bridge package to enable ROS 2 communication. QGroundControl version 4.4.2 functions as the ground control station software, facilitating mission planning and parameter configuration during simulations and testing phases. The software environment also includes specific versions of essential dependencies, such as Python 3.10 and NumPy 1.24.

The experimental framework incorporates comprehensive data collection and analysis capabilities, which are essential for systematic performance evaluation and algorithm optimisation. The primary data logging mechanism is ROS 2 bag recording, which captures all relevant system topics during experimental runs, including sensor data, algorithm outputs, and vehicle states. The ROS 2 bags infrastructure enables selective topic recording with configurable compression settings, allowing for the management of storage requirements while preserving data fidelity. Real-time monitoring capabilities utilize RQT (ROS Qt-based tools) for live system visualization and debugging during experimental execution. The RQT framework offers modular graphical interfaces, including real-time plotting of numerical data, topic monitoring for message inspection, parameter configuration for dynamic system tuning, and node graph visualisation for analysing system architecture. For time-series analysis, PlotJuggler is employed as the primary tool for offline data visualization and performance assessment. PlotJuggler enables comprehensive analysis of recorded bag files through advanced plotting capa-

bilities, such as multi-variable correlation analysis, statistical processing of time-series data, and customizable dashboard creation for systematic performance comparison.

Chapter 7

Results and Analysis

7.1 Spiral Implementation

As discussed in previous chapters, the obstacle avoidance node utilizes Archimedean spirals as described in Section 3.3.2. This mechanism is implemented within the Obstacle Detector and Avoidance Node detailed in Section 5.1. fig. 7.1 provides a graphical representation of the spiral computation using Rviz2.

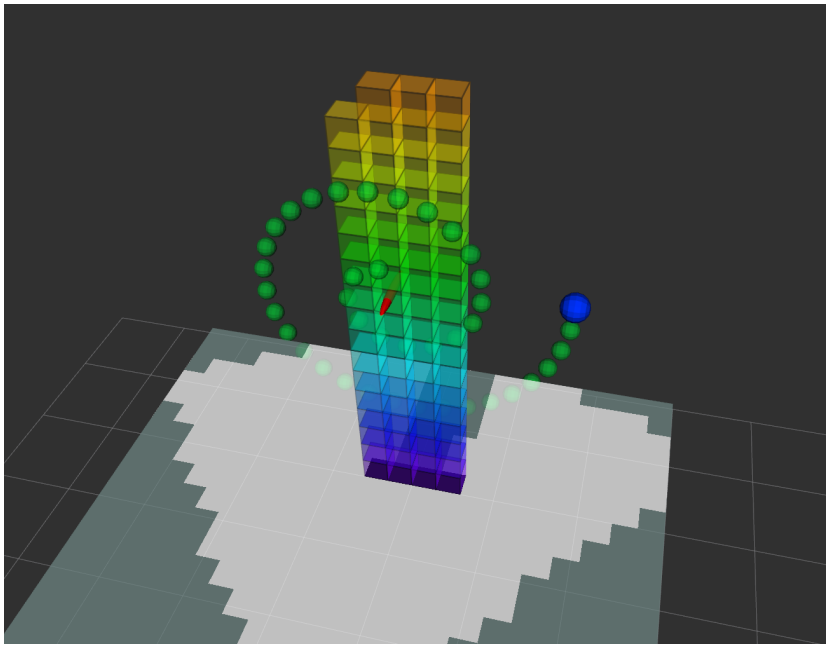

Figure 7.1: Spiral computation example

The green dots represent candidate escape points that were evaluated during the computation process, while the blue dot indicates a candidate escape point that provides a safe waypoint with minimal deviation from the current path. The red marker in the picture represents the closest collision point, meaning it is the one most likely to result in a collision. The Obstacle Detector and Avoidance Node initiate the com-

putation of escape points, starting the spiral from the closest point while moving on a plane that is normal to the line connecting the drone's current position and its current waypoint.

7.2 Simplified Scenario

This section presents results obtained from the simplified simulated scenario that serves as a foundational validation of the proposed drone navigation algorithm, demonstrating its core functionalities in a controlled environment before advancing to more complex operational conditions. This preliminary testing phase is essential for establishing the algorithm's baseline performance and ensuring that all fundamental components work together accurately. The simplified obstacle environment that forms the foundation of this testing scenario is illustrated in fig. 6.2.

The three blue rectangular obstacles are strategically positioned to create a clean and minimalist environment. This design allows for a focused evaluation of the core navigation algorithms without the complexities introduced by more realistic or cluttered scenarios.

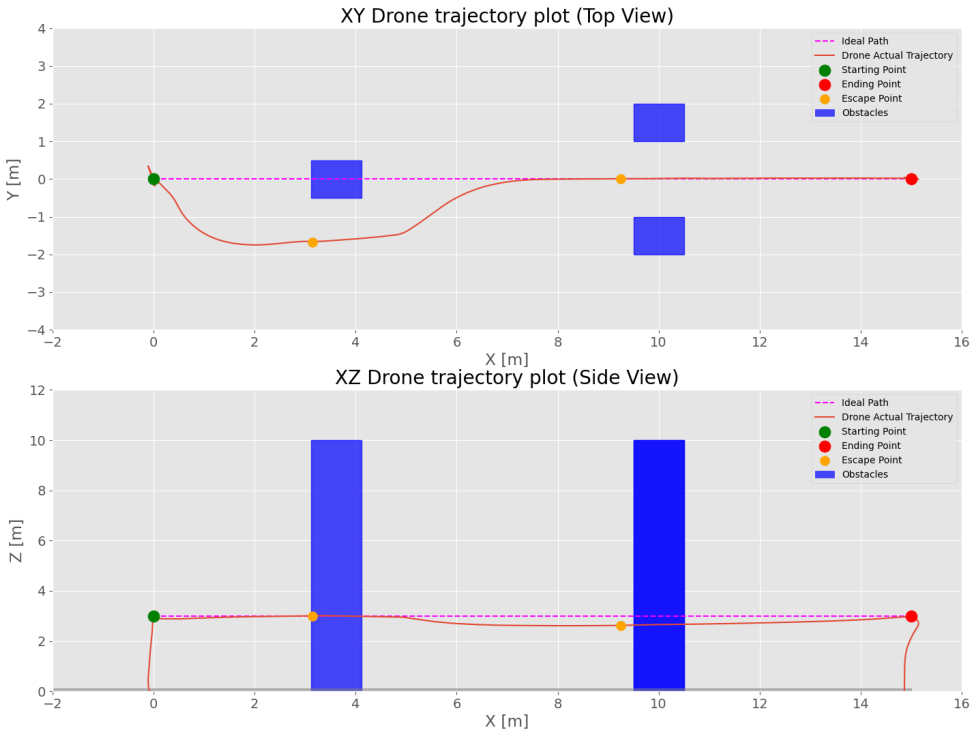

Figure 7.2: Top view and Side view

The spacing between the obstacles is carefully calibrated to ensure that viable paths

exist while still requiring the obstacle avoidance algorithm to navigate to the final goal without collisions successfully. fig. 7.2 presents the XY and XZ trajectory plots, which serve as the primary visualization for understanding the drone's horizontal and vertical navigation behaviour.

The XY and XZ plots clearly illustrate the ideal path, which is the connection between the starting point (green dot) and the ending point (red dot), along with the path followed by the drone as it reaches the final waypoint without colliding with any obstacles. This path is generated dynamically by the obstacle avoidance algorithm, which provides necessary escape points (orange dots) at runtime to successfully avoid the obstacles.

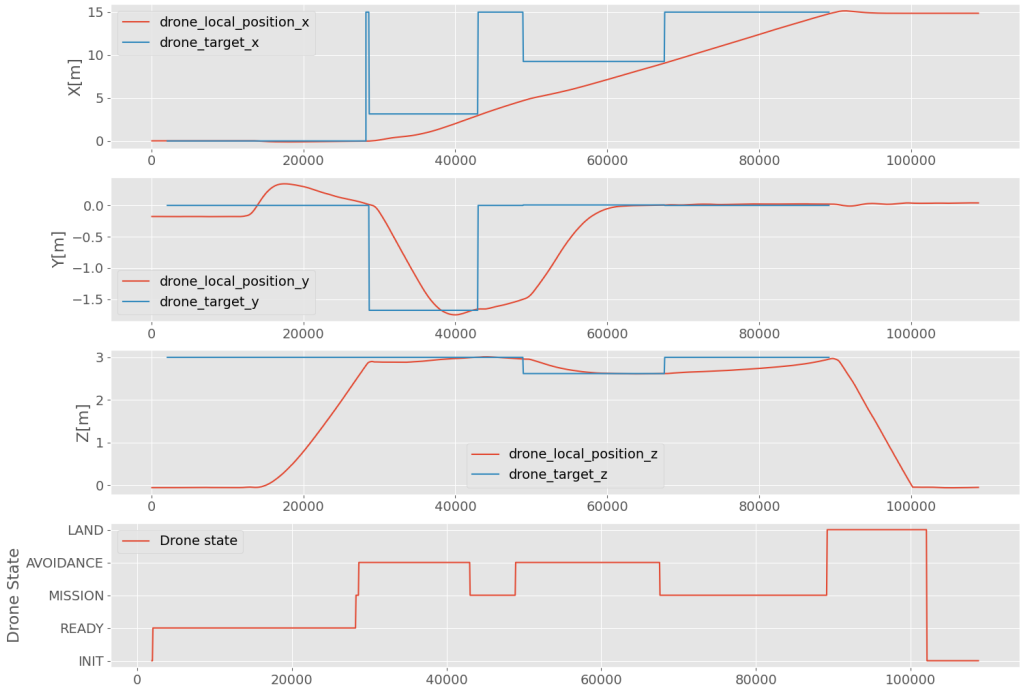

Figure 7.3: Positions and Drone state timeseries

fig. 7.3 illustrate a general temporal analysis of the drone's position tracking performance by comparing the actual trajectory (represented by the red lines) with the target waypoints (shown by the blue lines) across all three spatial dimensions. Additionally, the plot depicts the evolution of the drone's state over the simulation period. It is important to note that from the beginning of the simulation until the drone reaches the first waypoint (indicated by the green dot in fig. 7.2), the drone remains in the READY state. After approaching this point, the main simulation begins, and the drone navigates through the pre-defined list of waypoints. During this navigation, the

obstacle avoidance algorithm is active and prepared to intervene in the case of obstacle detection. If an obstacle is detected while the drone is in the normal navigation state (MISSION state), the state machine transitions to the AVOIDANCE state. An escape point (if available) is then assigned as the new target for the drone. Once the obstacle has been bypassed, the state machine returns to the MISSION state, resuming navigation from the next waypoint in the list. When the final waypoint is reached, the state machine transitions to the LAND state, enabling the PX4 to safely handle the drone's landing.

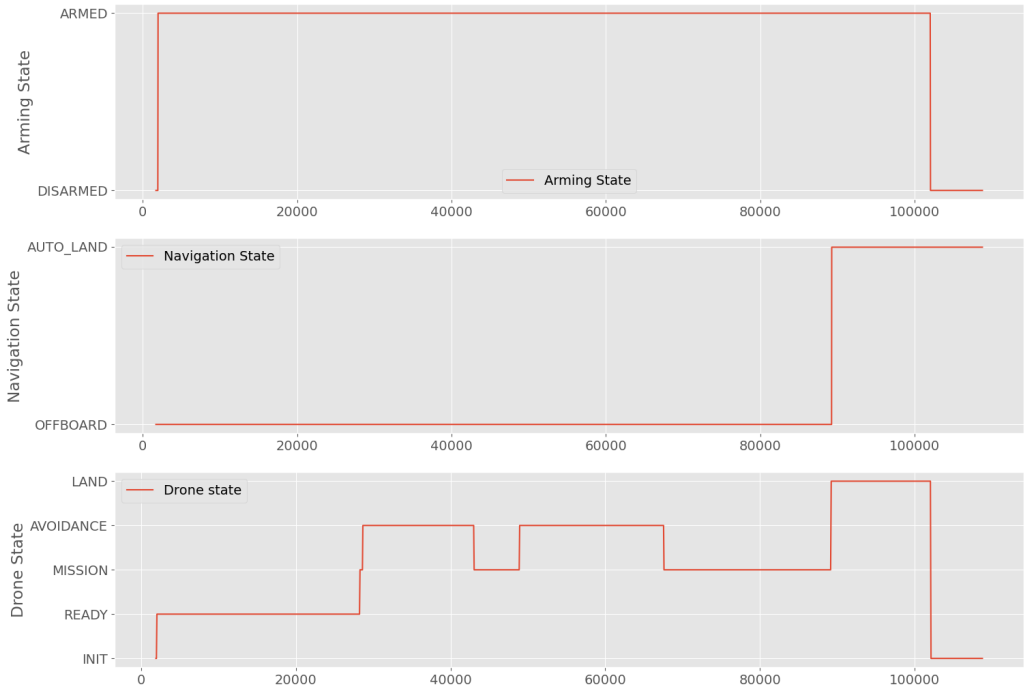

Figure 7.4: PX4/Avoidance state diagram

To ensure proper cooperation between PX4 and ROS 2 nodes, as discussed previously in 4.3.2, the PX4 must be in OFFBOARD mode, and the drone must be armed. fig. 7.4 illustrates that at the start of the simulation, while the avoidance node is in the INIT state, the drone is armed, and offboard control mode is activated. From this point onward, the Reactive Navigation Controller Node is responsible for controlling the drone. This configuration must remain unchanged throughout the drone's operations. Once the operations are complete and the last waypoint is reached, the state machine transitions to the LAND state, allowing PX4 to handle the landing operation safely using its Auto Landing Navigation state. After landing, the drone is disarmed.

7.3 Cluttered Scenario

The cluttered scenario, illustrated in fig. 6.3 features a comprehensive three-dimensional obstacle course that is 55 meters long and rises up to 12 meters in height. It is designed to systematically evaluate the navigation capabilities of autonomous drones across various challenges. The environment includes twenty-seven distinct obstacles, each with different geometric shapes, such as large rectangular barriers, cylindrical pipes, and thin wire-like structures. These obstacles are arranged in a progressive difficulty order to test increasingly advanced navigation skills.

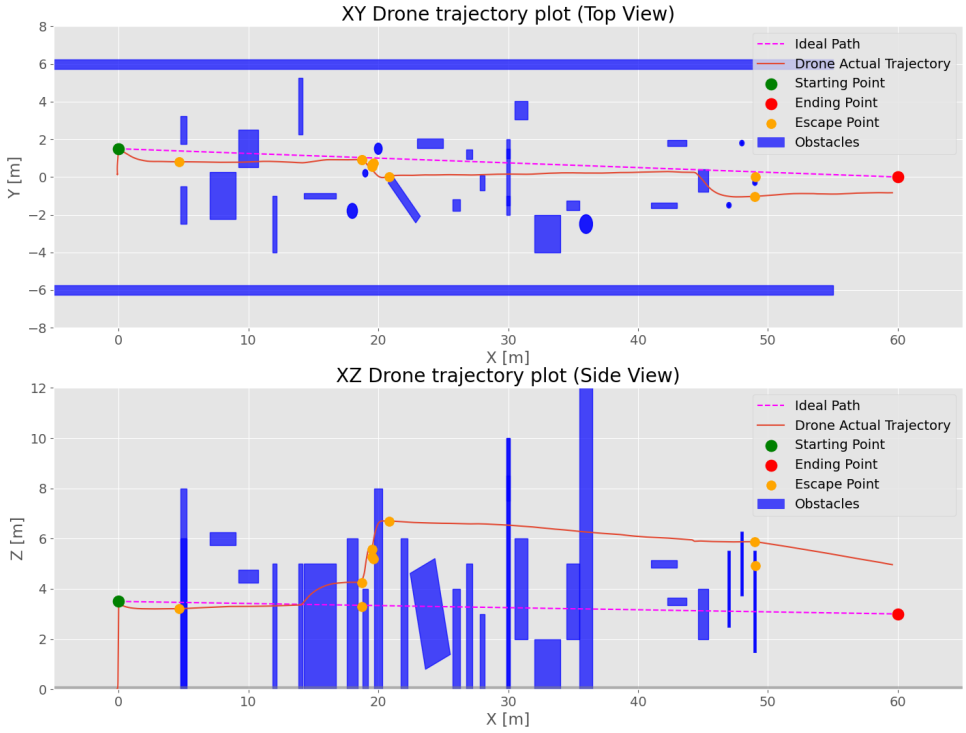

Figure 7.5: Top View and Side view

The XY and XZ plots shown in fig. 7.5 provide a top view and a side view of the simulation, respectively. The area around x = 20 m was quite cluttered, leading to multiple escape point calculations. However, all obstacles were successfully avoided, allowing the drone to reach the final waypoint without any collisions.

Furthermore, fig. 7.6 depicts the drone's position in relation to its target, as well as its operational state throughout the mission. This provides valuable insight into the drone's performance during its flight. Meanwhile, fig. 7.7 illustrates both the requested velocities—derived from the TrajectorySetpoint message—and the current state of the drone.

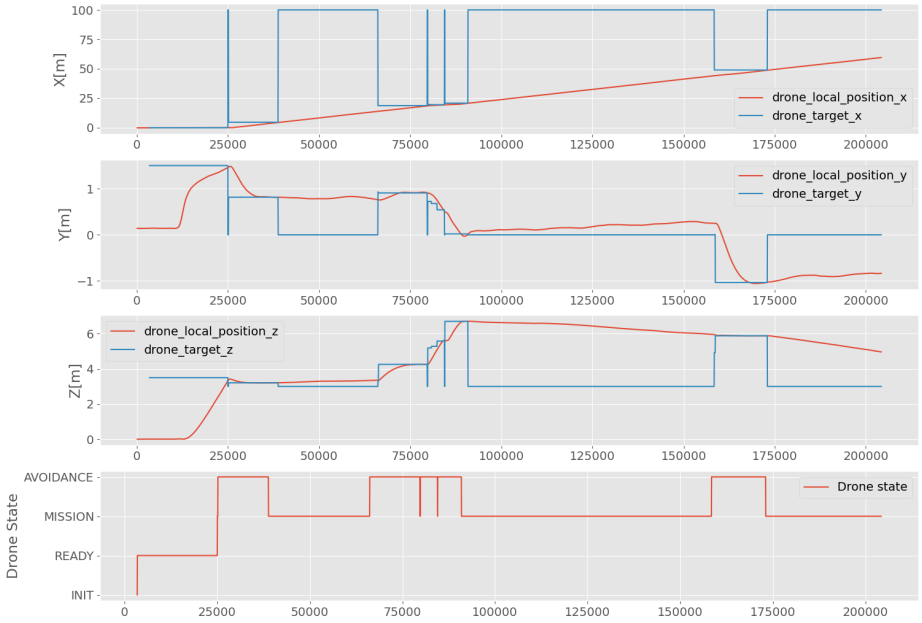

Figure 7.6: Position and Drone state timeseries

Figure 7.7: Requested velocities

Chapter 8

Conclusions

This thesis work presents an effective implementation of a spiral-based reactive obstacle avoidance algorithm as a robust and real-time solution to the complex problem of preventing collisions during UAV navigation. The central contribution of this work lies in the successful adaptation and integration of this spiral-based algorithm within the modern ROS 2 framework, ensuring seamless communication with the PX4 flight stack via SITL simulation. The algorithm leverages depth camera data processed by Octomap for efficient probabilistic 3D environment representation with reduced memory occupancy. Its reactive approach fundamentally relies on an Archimedean spiral to identify real-time escape points derived from environmental data. The system's distributed architecture, encompassing Gazebo for high-fidelity simulation, ROS 2 for high-level command and data processing, and PX4 for low-level flight control, demonstrated a cohesive and modular design. The effectiveness of the proposed system was rigorously validated through Gazebo simulation testing across both simplified and cluttered scenarios. These tests unequivocally demonstrated the system's ability to detect and successfully avoid obstacles while guiding the drone to its target waypoint.

8.1 Future Work

Building upon the current achievements, several key areas have been identified for future work and improvement, addressing technical limitations and expanding the system's capabilities.

Firstly, to optimise environment perception processing, remove the Octomap Server Node and integrate its capabilities into the Obstacle Detector and Avoidance Node, as previously mentioned. Secondly, the current core ROS 2 nodes, including the Obstacle Detector and Avoidance Node and the Reactive Navigation Controller Node, are implemented in Python. Implementing them in C++ would offer superior performance and lower latency.

Thirdly, to fully leverage GPU capabilities, while the thesis primarily focused on a CPU-based spiral algorithm, the original Azevedo et al. paper discusses both CPU and GPU map representations. The GPU approach, specifically using GPU-Voxels with its Voxel Map storage method, offers faster point cloud insertion and updates. In real-world scenarios on resource-constrained hardware like the Jetson Nano, GPU-Voxels can outperform Octomap for map updates.

Finally, to enhance efficient path planning, the current system relies solely on a local planner. Consequently, if no valid escape point is identified in complex situations, the system defaults to an immediate landing. A viable next step is to implement a global recovery strategy by integrating a global planner with the existing local avoidance system. This hybrid approach combines a low-level reactive layer for immediate safety with a high-level global planner for optimized long-term paths, enabling the UAV to overcome local minima and make more informed decisions based on a broader understanding of the environment.

Bibliography

[1] J. Minguez and L. Montano, "Nearness diagram (ND) navigation: collision avoidance in troublesome scenarios," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 1, pp. 45–59, 2004.

[2] S. Vanneste, B. Bellekens, and M. Weyn, "3DVFH+: Real-time three-dimensional obstacle avoidance using an octomap," in *Proceedings of the 1st International Workshop on Model-Driven Robot Software Engineering*, vol. 1319 of *CEUR Workshop Proceedings*, (York, UK), pp. 91–102, CEUR-WS.org, 2014.

[3] T. Baumann, "Obstacle avoidance for drones using a 3dvfh* algorithm," master's thesis, ETH Zurich, Zurich, Switzerland, 2018. Available online.

[4] M. Z. Butt, N. Nasir, and R. A. Rashid, "A review of perception sensors, techniques, and hardware architectures for autonomous low-altitude UAVs in non-cooperative local obstacle avoidance," *Robotics and Autonomous Systems*, vol. 173, p. 104629, 2024.

[5] A. Woods and H. La, "Dynamic target tracking and obstacle avoidance using a drone," in *Intelligent Technologies and Engineering Systems*, Lecture Notes in Electrical Engineering, (Cham), Springer, 2015.

[6] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, pp. 189–206, 2013.

[7] F. Azevedo, J. S. Cardoso, A. Ferreira, T. Fernandes, M. Moreira, and L. Campos, "Efficient reactive obstacle avoidance using spirals for escape," *Drones*, vol. 5, no. 2, p. 51, 2021. Open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

[8] PX4 Development Team, "PX4 systems architecture." https://docs.px4.io/main/en/concept/px4_systems_architecture.html, 2024. Accessed: 07 July 2025.

[9] PX4 Development Team, "ROS 2 user guide | PX4 user guide." https://docs.px4.io/main/en/ros2/user_guide.html, 2024. Accessed: 07 July 2025.

[10] Holybro, "PX4 development kit X500V2." https://docs.holybro.com/drone-development-kit/px4-development-kit-x500v2, 2024. Accessed: 07 July 2025.

[11] S. Hrabar, "Reactive obstacle avoidance for rotorcraft uavs," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, (San Francisco, CA, USA), pp. 4967–4974, 2011.

[12] M. Mujahed, D. Fischer, and B. Mertsching, "Admissible gap navigation: A new collision avoidance approach," *Robotics and Autonomous Systems*, vol. 103, pp. 93–110, 2018.

[13] J. A. Steiner, X. He, J. R. Bourne, and K. K. Leang, "Open-sector rapid-reactive collision avoidance: Application in aerial robot navigation through outdoor unstructured environments," *Robotics and Autonomous Systems*, vol. 112, pp. 211–220, 2019.

[14] J. Borenstein and Y. Koren, "The vector field histogram—fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, 1991.

[15] I. Ulrich and J. Borenstein, "VFH+: Reliable obstacle avoidance for fast mobile robots," in *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, vol. 2, (Leuven, Belgium), pp. 1572–1577, IEEE, 1998.

[16] I. Ulrich and J. Borenstein, "VFH*: Local obstacle avoidance with look-ahead verification," in *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, vol. 3, (San Francisco, CA, USA), pp. 2505–2511, IEEE, 2000.

[17] A. Chakravarthy and D. Ghose, "Obstacle avoidance in a dynamic environment: a collision cone approach," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 28, no. 5, pp. 562–574, 1998.

[18] J. van den Berg, M. Lin, and D. Manocha, "Reciprocal velocity obstacles for real-time multi-agent navigation," in *2008 IEEE International Conference on Robotics and Automation*, (Pasadena, CA, USA), pp. 1928–1935, IEEE, 2008.

[19] Y. Zhang, "Flight path planning of agriculture UAV based on improved artificial potential field method," in *2018 Chinese Control and Decision Conference*, (Shenyang, China), pp. 1526–1530, IEEE, 2018.

[20] X. Fan, Y. Guo, H. Liu, B. Wei, and W. Lyu, "Improved artificial potential field method applied for AUV path planning," *Mathematical Problems in Engineering*, vol. 2020, p. 6523158, 2020.

[21] F. Azevedo, A. Oliveira, A. Dias, J. Almeida, M. Moreira, T. Santos, A. Ferreira, A. Martins, and E. Silva, "Collision avoidance for safe structure inspection with multirotor uav," in *2017 European Conference on Mobile Robots (ECMR)*, pp. 1–7, 2017.

Appendix A

System Installation Guide

The installation and setup process generally involves four main steps:

1. Installing ROS 2
2. Installing PX4
3. Setting up the Micro XRCE-DDS Agent and Client
4. Building and running a ROS 2 workspace

These steps are detailed in the PX4 ROS 2 Installation Guide. Additionally, this guide includes installation instructions for properly pairing ROS 2 Humble with the Gazebo Harmonic bridge. For further reference, you can also find the official guide here: ROS 2 Humble/ Gazebo Harmonic Bridge. An example of a topic that can be bridged is the following:

```
- ros_topic_name "depth_camera/points"
  gz_topic_name "depth_camera/points"
  ros_type_name "sensor_msgs/msg/PointCloud2"
  gz_type_name "gz.msgs.PointCloudPacked"
  direction "GZ_TO_ROS"
```

Listing A.1: Bridging configuration example for point cloud topic

To customize Gazebo worlds and models, refer to this comprehensive guide: Adding New Worlds and Models for PX4 SITL.

Octomap mapping package can be found at Octomap_mapping, it contains also the `octomap_server` package.

```
ros2 run octomap_server octomap_saver_node --ros-args -p octomap_path
    :=(path for saving octomap)
```

Listing A.2: Octomap server run example

In this step, the `/cloud_in` topic should be remapped to `/depth_camera/points`, which corresponds to the `ros_topic_name` specified in the bridge configuration example.

Appendix B

Algorithms and References

Algorithm 4 CalculateRayCenters

Require: $\vec{p}_{origin}$, $\vec{d}$ (normalized direction vector)
Ensure: List of ray center positions

ray_centers $\leftarrow$ []
max_range $\leftarrow \lceil R_{safety}/V_{voxel} \rceil + 1$
for $i = -$max_range to max_range **do**
    for $j = -$max_range to max_range **do**
        if $\sqrt{i^2 + j^2} \cdot V_{voxel} \leq R_{safety}$ **then**
            ray_center $\leftarrow \vec{p}_{origin} + [i \cdot d_y, -i \cdot d_x, j] \cdot V_{voxel}$
            ray_centers.append(ray_center)
        end if
    end for
end for
return ray_centers

Algorithm 5 CastRaysFromCenters

Require: ray_centers, $\vec{d}$, max_range
Ensure: List of obstacle hits

hit_list $\leftarrow$ []
for each ray_center in ray_centers **do**
    result $\leftarrow$ octree.castRay(ray_center, $\vec{d}$, max_range)
    if result indicates hit **then**
        hit_list.append({ray_center, end_point})
        Create hit visualization marker
    end if
end for
return hit_list

Algorithm 6 ComputeEscapePoint (Archimedean Spiral)

Require: $\vec{p}_{obstacle}$, $\vec{p}_{origin}$, $\vec{d}$
Ensure: Escape point or None
$a \leftarrow V_{voxel}/2$ ▷ Spiral parameter
$\theta_{prev} \leftarrow 0.0$
for iteration = 1 to N_{max_iter} **do**
  $\theta \leftarrow 2.0\sqrt{\theta_{prev}^2/4.0 + 1.0}$
  $\text{radius}_{hor} \leftarrow a \cdot \theta \cdot \cos(\theta)$
  $e_x \leftarrow o_x + \text{radius}_{hor} \cdot d_y$
  $e_y \leftarrow o_y - \text{radius}_{hor} \cdot d_x$
  $e_z \leftarrow o_z + a \cdot \theta \cdot \sin(\theta)$
  $\vec{p}_{escape} \leftarrow [e_x, e_y, e_z]$
  if $|e_z - o_z| > 2.0$ **then**
    $\theta_{prev} \leftarrow \theta$
    continue ▷ Skip points too far vertically
  end if
  $\vec{d}_{escape} \leftarrow \frac{\vec{p}_{escape} - \vec{p}_{origin}}{||\vec{p}_{escape} - \vec{p}_{origin}||}$
  ray_centers $\leftarrow$ CalculateRayCenters($\vec{p}_{origin}, \vec{d}_{escape}$)
  hit_list $\leftarrow$ CastRaysFromCenters(ray_centers, $\vec{d}_{escape}, L_{search}$)
  if |hit_list| = 0 **then**
    Create escape point visualization marker
    return $\vec{p}_{escape}$ ▷ Clear path found
  end if
  $\theta_{prev} \leftarrow \theta$
end for
return None ▷ No valid escape point found

Algorithm 7 StateMachine

Require: drone_state, waypoint_list W, $\vec{p}_{avoidance}$
Ensure: Updated drone state and published velocity commands

if drone_state = READY **then**
    Publish current target: $W[0]$
    if IsAtPosition($W[0]$) **then**
        W.pop(0)
        Publish new target: $W[0]$
        drone_state ← MISSION
    else
        UpdateVelocity($W[0]$)
    end if
else if drone_state = MISSION **then**
    if IsAtPosition($W[0]$) **then**
        if $|W| = 0$ **then**
            drone_state ← LAND
        end if
        W.pop(0)
        Publish new target: $W[0]$
    else
        UpdateVelocity($W[0]$)
    end if
else if drone_state = AVOIDANCE **then**
    if IsAtPosition($\vec{p}_{avoidance}$ **then**
        Publish target: $W[0]$
        drone_state ← MISSION
    else
        Publish target: $\vec{p}_{avoidance}$
        UpdateVelocity($\vec{p}_{avoidance}$)
    end if
else
    Send land command
end if

Algorithm 8 UpdateVelocity

Require: $\vec{p}_{target}$
Ensure: Published velocity setpoint

$\vec{e}_{pos} \leftarrow \vec{p}_{target} - \vec{p}_{drone}$
$d \leftarrow ||\vec{e}_{pos}||$
if $d < \delta_{pos}$ **then**
    $\vec{v}_{cmd} \leftarrow [0, 0, 0]$
    PublishVelocitySetpoint($\vec{v}_{cmd}$)
    return
end if
$\vec{d}_{norm} \leftarrow \vec{e}_{pos}/d$
$\vec{v}_{cmd} \leftarrow \frac{\vec{d}_{norm}}{||\vec{d}_{norm}||} \times V_{max}$
PublishVelocitySetpoint($\vec{v}_{cmd}$)

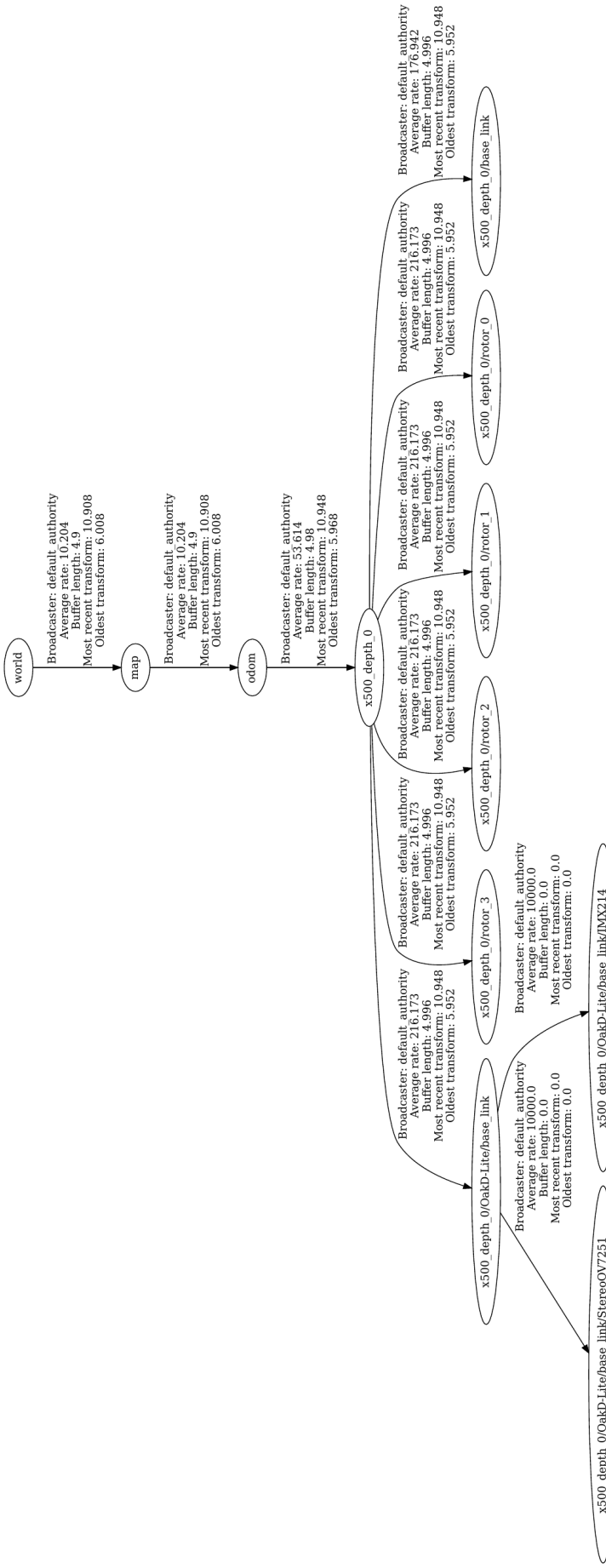

Figure B.1: Simulation TF tree