POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

A*-based Collision Avoidance for UAVs with ROS 2 and PX4 Integration

Supervisors

Dr. Stefano PRIMATESTA

Dr. Davide BITETTO

Dr. Gianluca RISTORTO

Candidate

Elena BERTA

October 2025

Abstract

This thesis, developed in collaboration with **MAVTech s.r.l.**, presents a modular architecture for **collision avoidance** in **UAVs** designed for outdoor applications. The system is implemented in **ROS2** and integrated with the **PX4 Autopilot** for flight operations. The primary objective is to enhance safety while maintaining operational continuity: avoidance maneuvers are triggered only when necessary, ensuring no alteration of system nominal conduct.

The approach combines an **A* local planner** operating on a **dynamic costmap**—fed by on-board sensors—with a **Bug**-like reactive behavior as a fallback strategy when local scheduling does not locate safe steps. A context-aware mode management system determines which algorithm to activate and when to apply its output to vehicle control.

The architecture emphasizes a clear separation between perception, planning, and control, promoting portability, reusability, and future scalability.

Development follows a two-phase strategy. In a first phase, the solution is implemented on a **terrestrial mobile robot** to consolidate the avoidance logic in a platform-agnostic context, without autopilot dependencies. In a second phase, the approach is transferred to the **drone**, favoring the optimization of A* for flight, while the integration of Bug behavior is postponed to future work. In this scenario, a state system is introduced to manage mission flow: it allows PX4 mission commands to proceed uninterrupted unless obstacles are detected, in which case the A* planner intervenes to ensure safe navigation. Integration with PX4 is achieved in **Offboard mode**, with careful attention to reference frame consistency and command interface alignment.

The evaluation covers the entire spectrum of scenarios in **simulation**, including challenging ones.

Overall, the work proposes a flexible and reusable solution for integrating collision avoidance into UAV platforms—compatible with real-world missions—while maintaining seamless integration within the ROS 2 and PX4 ecosystems.

Table of Contents

List of Tables

List of Figures

Acronyms

CRS Coordinate Reference System

DDS Data Distribution Service

DWB Dynamic Window Based

ENU East-North-Up

EPSG Geodetic Parameter Dataset (CRS identifier)

FLU Forward-Left-Up

FOV Field Of View

FRD Forward-Right-Down

GCS Ground Control Station

GPS Global Positioning System

IMU Inertial Measurement Unit

LTS Long Term Support

MAVLink Micro Air Vehicle Link

Nav2 Navigation2

NED North-East-Down

PX4 PX4 Autopilot

QoS Quality of Service

ROS 2 Robot Operating System 2

RViz ROS Visualization

SITL Software-In-The-Loop

TF2 Transform library for ROS 2

UAV Unmanned Aerial Vehicle

uORB micro Object Request Broker

UTM Universal Transverse Mercator

uXRCE-DDS Micro XRCE-DDS

VTOL Vertical Take-Off and Landing

YAML YAML Ain't Markup Language

Chapter 1

Introduction

In recent decades, unmanned aerial vehicles (UAVs), commonly known as drones, have become increasingly widespread thanks to advances in control, sensor, and communication technologies. Initially developed mainly for military applications, drones are now extensively used in civil and industrial contexts. Among the areas of greatest interest are environmental monitoring, hard-to-reach areas surveillance, precision agriculture, photogrammetry, infrastructure inspection, and, more recently, logistics and urban deliveries.

The ability to perform autonomous and safe operations is one of the key factors driving the large-scale adoption of these technologies. In this framework, collision avoidance represents a fundamental requirement for ensuring safe drone navigation in real-world scenarios characterized by uncertainty, dynamism and complexity. The absence of reliable obstacle avoidance systems severely limits the use of drones, relegating them to controlled environments or simple missions where routes are pre-planned and are not subject to unexpected changes.

Collision avoidance can be addressed in different ways. On the one hand, the choice of sensors is crucial for the quality and reliability of the collected information about the surrounding environment. On the other hand, the control logic, together with path-planning and obstacle-reaction algorithms, constitutes the core element that allows the drone to transform the gathered details into concrete actions.

Within this framework, the present research takes place with the aim of developing and validating a collision avoidance algorithm for drones operating in outdoor environments, designed to be modular and easily integrated into existing control architectures.

The work was carried out in collaboration with **MAVTech s.r.l.**, a company that designs and develops advanced UAV systems for professional applications. Founded as a spin-off of Politecnico di Torino, MAVTech combines academic research with industrial expertise and actively participates in national and international projects in autonomous aerial systems field. Operating in both Turin and

Bolzano, the company develops UAV solutions employed in real-world missions such as infrastructure inspection, precision agriculture, civil protection, and search and rescue. The cooperation with MAVTech ensured that the solution developed in this thesis was oriented towards real-world applicability, with the final objective of testing the algorithm on an actual drone under realistic operating conditions.

The research was structured in several complementary phases:

- **Preliminary analysis of the state of the art**, with particular focus on the most promising sensory technologies and algorithms;
- **Development of the algorithm** in *Python* within *ROS2* framework, including dedicated logic for edge-case management to ensure robustness;
- **Preliminary validation in simulation**, using *TurtleBot3* integrated with *Gazebo* and *RViz*, in order to verify the correct functioning of the proposed logic in controlled scenarios;
- **Adaptation and advanced testing** in the *PX4 SITL* (Software In The Loop) environment, again in combination with *Gazebo* and *RViz*, to realistically reproduce the dynamic behaviour of a UAV;
- **Real-world testing**, to verify the algorithm on a physical drone and evaluate its effectiveness in actual operating conditions.

Chapter 2

State of art

2.1 Sensors

Obstacle avoidance is one of the most important features for ensuring safety and reliability of UAVs. For this aim, various types of sensors have been studied and developed, each one with its own characteristics in terms of accuracy, robustness, cost and computational complexity.

In general, these sensors can be divided into two broad categories [1]:

- **Active sensors**: they emit a signal (electromagnetic or acoustic radiation) and measure its reflection on surrounding objects to estimate their position and distance. Among the most common ones there are radar, LiDAR and ultrasonic sensors;
- **Passive sensors**: these do not emit their own energy, but simply capture radiation already present in the environment or generated by objects naturally. This category includes optical cameras, infrared sensors and, more recently, event cameras.

The choice of the most suitable sensor depends on the operating scenario and the level of required autonomy: active sensors tend to guarantee greater robustness in difficult conditions, while passive sensors offer lighter and more economical solutions, but are highly subject to environmental conditions.

2.1.1 Radar

Radar is one of the most stable technologies for detecting obstacles. The operating principle is based on the emission of radio waves and the measurement of the return

time of the reflected signal, from which the distance and the size of the object can be calculated. There are two main configurations:

- **Continuous-Wave Radar (CW)**: it transmits continuously modulated frequency signals, ensuring a constant data flow;
- **Pulsed-Wave Radar (PW)**: it emits short, high-intensity pulses, both reducing energy consumption and improving accuracy.

Both types have blind zones, but in addition to distance, they also allow the speed and relative motion of obstacles to be estimated using the Doppler effect of the reflected signal [1].

Advantages: Operational in all weather conditions; Reliable detection even in dusty or smoky environments; Large operational range; High accuracy; Identification of moving obstacles;

Disadvantages: Computational complexity; High cost compared to cameras' one [1], [2].

2.1.2 LiDAR (Light Detection and Ranging)

LiDAR is a technology now widely used in autonomous systems, based on the emission of laser pulses and the measurement of the signal return time. The combination of transmitter and receiver allows high-resolution maps of the surrounding environment to be obtained, with 1D, 2D and 3D solutions depending on the level of required detail [1], [3].

Advantages: High accuracy in distance measurement; Ability to operate in low light conditions; Adaptability even to small UAV platforms; Lower cost than high-end radars' one [1].

Disadvantages: Significant weight and energy consumption; Relatively high cost; Sensitivity to reflective or transparent surfaces; Difficulty in reliably detecting fast-moving obstacles [1], [2].

2.1.3 Ultrasonic

Ultrasonic sensors operate by sending sound pulses and measuring the return time of the echo reflected by objects [1], [4]. The produced sound waves are outside the range audible to the human ear, between 20 and 50 kHz.

Advantages: Low cost; Lightweight; Easy to integrate; Suitable for short-range applications.

Disadvantages: Limited accuracy in noisy environments or on uneven surfaces; Reduced range [1].

2.1.4 Infrared

Infrared sensors base their operation on thermal radiation emitted by obstacles naturally.

Advantages: Lightweight and low cost; Ideal for short-range applications and for low-light environments.

Disadvantages: Sensitive to atmospheric conditions; Possible false positive and negative; Limited range [1].

2.1.5 Cameras (Monocular, Stereo, RGB)

Cameras are one of the most widely used solutions thanks to their low cost and ability to provide rich and versatile information regarding the environment.

- **Monocular Camera**: it provides two-dimensional images; depth is estimated using computer vision algorithms based on movement, shadows or machine learning techniques [1], [5];

- **Stereo Camera**: using two lenses arranged in parallel, it captures images from different perspectives, allowing three-dimensional reconstruction and direct depth estimation [1], [6];

- **RGB Camera (Red-Green-Blue)**: it captures colour images useful for object recognition and scene analysis, but does not provide direct information on distance [7].

Advantages: Low cost; Light weight; Compact size; Low energy consumption; Ability to recognize textures, contours and visual characteristics [1].

Disadvantages: Highly dependent on lighting conditions [2]; They requires advanced algorithms for depth extraction (in the case of monocular cameras) [1], [5].

2.1.6 Event Cameras

Event cameras are innovative sensors inspired by biological visual systems that work by capturing changes in brightness at the pixel level, instead of capturing frames at fixed intervals, with an high response rate. The output of an event camera is not an image but a stream of asynchronous events [8].

Advantages: Effective in low-light environments and for detecting fast-moving obstacles [9]; Low power consumption; Low data consumption compared to conventional cameras' one [8].

Disadvantages: Require advanced processing to reduce false positives; Need specific algorithms; Sensitive to environmental variations; Do not integrate well with other sensors types [9].

2.1.7 Multi-Sensor Fusion

Multi-sensor fusion is a technique that allows information from multiple heterogeneous sensors (e.g., radar, LiDAR, cameras, or ultrasonic sensors) to be integrated with the aim of obtaining a more complete, accurate, and robust perception of the surrounding environment. The combined use of active and passive sensors compensates for the intrinsic limitations of each device, improving the ability of drones to detect obstacles and plan safe trajectories [2], [10], [11].

To achieve this integration, various fusion algorithms are employed, ranging from the simplest to the most advanced methods. Among the most widely used are **Extended Kalman Filters (EKFs)**, which estimate the state of the system by reducing measurement noise [2], [11], and **Bayesian** approaches, which allow probabilistic management of the uncertainty associated with sensory data [2]. More recent techniques include **Particle Filter Fusion**, which can handle non-linear models and multimodal distributions [10], and **Deep Learning** methods based on neural networks, used to learn the optimal fusion model directly from the data [2].

Advantages: Redundancy and complementarity ensure greater robustness and accuracy [2], [10], [11].

Disadvantages: Computational complexity; Difficult calibration and synchronization; Sensitivity to dynamic environments [10].

Sensor	Operating Principle	Pros	Cons	Cost	FOV	Range	Weight
Pulsed-Wave Radar (PW)	Emits short, high-energy pulses to detect distance and velocity	Lower energy consumption, accurate, reduced interference	Processing complexity, small blind zone	$800 – $2,500	60° – 120°	10 – 100 m	150–300 g
Continuous-Wave Radar (CW)	Continuously transmits waves and detects Doppler shift for motion detection	Excellent for moving object detection	Not ideal for measuring distance unless modulated	$600 – $2,000	90° – 150°	5 – 50 m	100–250 g
Ultra-Wideband (UWB) Radar	Transmits wideband signals for high resolution and penetration	High accuracy, good noise resistance, can estimate impact points	Expensive, complex data processing	$1,500 – $3,000	120° – 180°	10 – 80 m	200–400 g
1D LiDAR	Measures distance in a single axis (usually vertical or frontal)	Lightweight, low-cost, ideal for altitude	Very narrow view, no lateral awareness	$50 – $200	~1° – 5° (single beam)	0.2 – 5 m	10–50 g
2D LiDAR (Planar)	Scans horizontally or vertically to create 2D maps	Affordable, obstacle contour detection	Lacks height/depth info, reflection-sensitive	$300 – $800	180° – 270°	5 – 20 m	150–300 g
3D LiDAR (Rotating or Solid-state)	Emits laser beams in multiple directions to reconstruct 3D maps	High precision, 360° perception	High cost, high energy consumption, heavy	$4,000 – $75,000	360°	50 – 200 m	0.5 – 1.5 kg
Ultrasonic sensor	Emits high-frequency sound waves and measures echo delay	Low-cost, lightweight, simple integration	Sensitive to noise, limited range	$10 – $100	15° – 30°	0.2 – 5 m	5–20 g
Monocular Camera	Captures 2D images from a single lens	Lightweight, low cost, texture detection	No direct depth, requires algorithms	$20 – $500	40° – 120°	10 – 100 m (depending on optics)	10–100 g
Stereo Camera	Two lenses capture images for depth estimation	Provides depth, useful for obstacle avoidance	Sensitive to lighting conditions, heavier than monocular	$150 – $1,500	60° – 120°	5 – 50 m	50–200 g
Infrared Camera	Detects emitted infrared radiation from objects	Works in darkness, low-cost, lightweight	Sensitive to weather, false positives	$50 – $1,000	20° – 90°	2 – 30 m	20–150 g
Event Camera	Captures changes in brightness at pixel level asynchronously	Low power, high temporal resolution, good in low light	Expensive, requires specific algorithms, low compatibility	$2,000 – $5,000	90° – 120°	10 – 50 m	100–250 g

Table 2.1: Summary table for sensors.

2.2 Algorithms

The development of obstacle avoidance algorithms for UAVs is one of the most active fields of research in mobile robotics. The ability to react to unexpected events, adapt to complex scenarios and plan safe trajectories is, in fact, an essential requirement for the real-world, large-scale use of drones. Many approaches have been proposed in the literature, varying in computational complexity, sensory requirements and level of guaranteed autonomy.

According to the reference literature, these algorithms can be organized in ascending order of complexity: from reactive methods, based on simple local rules, to deliberative planning approaches, then advanced perception and artificial intelligence algorithms, and finally hybrid and integrated solutions, which represent the most recent trends.

2.2.1 Reactive Approaches Based On Proximity Sensors

Reactive algorithms based on proximity sensors are one of the simplest and most immediate solutions for obstacle avoidance. They use measurements from short-range sensors, such as ultrasonic, infrared or two-dimensional LiDAR, to adjust the drone's trajectory in real time [3].

Among the most representative methods there are **Bug Algorithms** which are based on deterministic rules for obstacle avoidance and are particularly suitable for simple, structured environments [5]. Other approaches include the **Sense-and-Avoid** paradigm, in which the drone in *Avoid-and-Continue mode* (autonomous mode) detects the presence of an obstacle and plans an immediate evasive manoeuvre, while in *Brake mode* (manual mode) it prioritises safety by stopping flight when an obstacle is detected at a critical distance [1], [3]. Another widely used technique is **Vector Field Histogram (VFH)**, which constructs a directional histogram from sensory data, identifying the most suitable clear corridors for continuing the flight [12].

Advantages: Simple to implement; Low computational cost; Very fast reaction times.

Disadvantages: No long-term planning capability; Difficulty in managing complex environments or those densely populated with obstacles.

2.2.2 Reactive Approaches Based On Force Fields

Force field-based algorithms belong to the category of reactive methods, but introduce a more abstract model of the interaction between the drone and the

environment. The basic idea is to represent the drone as a particle immersed in a potential field: the destination exerts an *attractive* force, while obstacles generate *repulsive* forces that push the vehicle to maintain a safe distance.

One of the best-known methods is the **Artificial Potential Field (APF)**, which allows trajectories to be calculated quickly, avoiding collisions through the vector sum of attractive and repulsive forces. Although widely used for its simplicity and immediacy, the *APF* suffers from problems related to local minima, situations in which the drone can become "trapped" in sub-optimal configurations.

An evolution of this approach is the **Dynamic Window Approach (DWA)**, which takes into account the dynamics and kinematic limitations of the drone in generating trajectories. The *DWA* evaluates a set of possible speeds within a *"dynamic window"* and selects the one that maximises safety with respect to obstacles, while ensuring progress towards the target [1], [13].

Advantages: Fast response time; Low computational cost; Easy to implement even on platforms with limited resources.

Disadvantages: Risk of getting stuck at local minima; Difficulty in managing complex environments or those with moving obstacles.

2.2.3 Local Planning Approaches

Local planning approaches fall somewhere between purely reactive methods and global planning perspectives. The main objective is to generate safe flight trajectories in real time, using partial and dynamic information from on-board sensors. Unlike reactive methods, local planning does not simply react instantly to obstacles, but constructs partial representations of the surrounding environment that are constantly updated.

A common technique is to use a **Voxel Map**, a discrete three-dimensional representation of space, in which obstacles are modelled as occupied voxels. This local map allows the identification of free volumes and spatial constraints to plan trajectories within. Integration with interpolation methods, such as **B-Splines**, allows the generation of continuous, smooth and dynamically feasible trajectories, reducing sudden manoeuvres and unwanted oscillations [14].

Another example is the **Trajectory Generation Based on Object Avoidance (TGBOA)** method, which constructs alternative trajectories by minimising deviations from the planned route, balancing safety and progress towards the target [15].

Advantages: High responsiveness; Computational efficiency; Generation of smooth and dynamically achievable trajectories; Good adaptability to fast-paced environments.

Disadvantages: Lack of long-term memory; Difficulty in managing very large or complex scenarios; Dependence on the quality of local representation.

2.2.4 Global Planning Approaches

Global planning approaches aim to generate an optimal path from origin to destination, taking into account the entire configuration of the navigation space. Unlike local planning, which only considers the immediate surroundings of the drone, global planning requires a complete or near-complete map of the environment, on the basis of which optimised trajectories can be calculated.

The most commonly used classical algorithms are deterministic graph search methods, such as **Dijkstra**'s algorithm, which always guarantees the shortest solution, and the **A*** algorithm, which introduces heuristic functions to speed up the search while maintaining optimality. These methods have long been the standard in path planning, also for UAV applications.

At the same time, approaches inspired by natural processes and metaheuristic methods have been developed, such as **Ant Colony Optimisation (ACO)**, which is based on the behaviour of ant colonies in searching for minimum paths, and **Particle Swarm Optimisation (PSO)**, which simulates the collective behaviour of swarms. Although these methods do not always guarantee the optimal solution, they allow complex search spaces to be explored in a short time and are suitable for large-scale scenarios [10].

Advantages: Generation of globally optimal or near-optimal trajectories; Possibility of considering complex constraints (energy, time, safety); Ability to prevent local minima typical of reactive approaches.

Disadvantages: High computational complexity; Need for a complete or updated map; Poor adaptability to sudden changes in the environment.

2.2.5 Model Predictive Control Approaches

Model Predictive Control (MPC) is an advanced control technique based on the idea of predicting the future evolution of the system and calculating, at every instant, the sequence of commands that optimises a cost function within a limited time horizon. In practice, the drone calculates multiple possible trajectories, evaluates which is the best based on criteria such as distance from the target, energy consumption or collision risk, and applies only the first command in the sequence. In the next step, the process is repeated using the new sensory information available.

One of the most recent developments is **Data-Driven Risk-Aware**, which integrates probabilistic models to estimate uncertainty about the position of obstacles and uses stochastic constraints, such as *Distributionally-Robust Chance Constraints (DRCC)*, to ensure safe trajectories even in dynamic and uncertain environments [16].

Advantages: Predictive capability; Robustness in the presence of uncertainties; Integration of dynamic and kinematic constraints.

Disadvantages: High computational complexity; Need for accurate models; Difficulty of implementation on platforms with limited resources.

2.2.6 SLAM Approaches

Simultaneous Localization and Mapping (SLAM) is one of the most widely used techniques for enabling autonomous navigation of drones in indoor unfamiliar environments [10]. The basic idea is to allow the drone to build a map of the environment while simultaneously estimating its own position within it. In this way, even without GPS, the system is able to move safely and effectively.

There are several variants of SLAM, which differ in the type of used sensor. Some exaples are: **Visual SLAM** (monocular cameras, stereo cameras or RGB-D cameras), **LiDAR SLAM** (LiDAR), **Monocular-Inertial SLAM** (combination of a single monocular camera with IMU) [**\cite {article_6}**, 5].

Advantages: Autonomous navigation without GPS; High positioning accuracy; Ability to generate maps in real time; Flexibility in the use of different sensors.

Disadvantages: High computational complexity; Sensitivity to sensor noise; Difficulty in application in highly dynamic or unstructured environments.

2.2.7 Event Cameras Approaches

Event cameras, already introduced previously, produce asynchronous event streams instead of traditional images. This type of output, which has no frames and consists of point variations in brightness, is not compatible with common artificial vision algorithms, which are based on the processing of complete images. For this reason, it is necessary to develop dedicated algorithms capable of correctly interpreting asynchronous data and transforming them into information useful for navigation.

Several algorithms have been proposed, among the best known are **Polar Time Difference Surface (PTDS)**, clustering techniques based on **Depth-First Search (DFS)** [9] and bio-inspired architectures such as **CEF**, **LEM** and **DOT**,

which seek to replicate perception mechanisms similar to those of biological systems [8].

Advantages: Extremely fast reaction times; Greater robustness in handling fast-moving obstacles; Ability to operate in scenarios with lighting variations that are difficult to manage with traditional sensors.

Disadvantages: Need to develop complex, non-standardized models; Difficulty integrating with other vision algorithms; Still limited availability of specific frameworks and datasets.

2.2.8 Deep Learning Approaches

In recent years, the introduction of deep learning techniques has revolutionized visual data processing for drones, enabling significantly higher performance than traditional machine vision methods. Due to their light weight, versatility and low cost, cameras are the preferred sensor in these approaches: integrated on board numerous commercial UAVs, they provide a continuous stream of visual information that can be processed in real time for obstacle detection and avoidance [6].

On the perception side, **Convolutional Neural Networks (CNNs)** are widely used for visual feature extraction, enabling the recognition of shapes, contours and textures in the surrounding environment [17]. **Monocular Depth Estimation (MDE)** techniques, based on deep learning, also allow depth to be estimated from a single camera, overcoming one of the main limitations of traditional monocular systems [18]. At the same time, **Recurrent Neural Networks (RNN, LSTM)** allow the modelling of temporal sequences and the prediction of the movement of dynamic obstacles [17].

On the decision-making and avoidance side, **Reinforcement Learning (RL)** has taken on a central role. In this paradigm, virtual agents are trained in simulated environments, such as *AirSim*, to interact with obstacles and complex scenarios, learning safe and adaptive flight strategies [17]. Algorithms such as **Deep Q-Networks (DQN)**, **Proximal Policy Optimisation (PPO)** [19] and **Soft Actor-Critic (SAC)** have performed well in autonomously generating safe trajectories and managing dynamic and unpredictable scenarios [6], [8], [19].

Advantages: Ability to recognise and classify complex obstacles; Learning of adaptive avoidance strategies.

Disadvantages: Need for extensive datasets; High computational load.

2.2.9 Decision Trees Approaches

Decision Trees represents a simpler and more interpretable machine learning approach than deep learning, but it still finds application in drone collision avoidance. The basic idea is to build a model capable of choosing the most appropriate action based on the available information, progressively dividing the decision space through hierarchical rules.

A typical workflow involves creating a *training dataset* obtained from real or simulated flights, in which corrective manoeuvres are recorded and associated with the corresponding environmental situations. Classifiers such as **J48** or **Random Forest** are then trained on this data, enabling them to generalise the decision rules and implement them on board the drone [20].

Advantages: Computational lightness; Simplicity of implementation; Greater transparency and interpretability of decisions.

Disadvantages: Strong dependence on dataset quality; Limited generalization capacity; Poor effectiveness in complex and dynamic scenarios.

2.2.10 Fusion Algorithms Approaches

An increasingly popular approach to obstacle avoidance in drones is the fusion of heterogeneous algorithms, designed to balance computational lightness, accuracy and robustness. Methods that have been proposed in the literature combine monocular cues, deep learning-based depth estimation models and reactive local planning algorithms, with the aim of overcoming the limitations of individual approaches and ensuring safer navigation.

A significant example is the work of [5], which integrates traditional **Monocular Cues** (texture gradient, feature detection, edge detection), the **MiDaS network** for depth estimation, and a modified variant of **Bug0** for local planning. The different outputs are merged using a weighting system that combines the lightness of classic cues with the greater precision guaranteed by deep learning. The entire architecture has been designed to be compatible with low-power embedded platforms, making the solution scalable even to lightweight and inexpensive UAVs.

Advantages: Ability to balance efficiency and accuracy; Compatibility with lightweight and inexpensive platforms; Greater robustness thanks to the integration of different approaches.

Disadvantages: Reduced informativeness in very simple scenarios; Difficulty of generalisation in highly complex environments; Risk of latency introduced by the use of heavier deep learning models.

Algorithm	Category	Strategy	Pros	Cons	Complexity
SLAM (Simultaneous Localization and Mapping)	Mapping / Localization	Builds a map while estimating drone position in real time	GPS-free navigation, accurate, real-time mapping	High computational demand, sensitive to sensor drift	High
Deep Learning (End-to-End, CNN, RNN, RL)	Learning-based	Learns obstacle patterns and avoidance strategies from data	Adaptive, capable of learning from dynamic scenarios	Training required, high computational resources	High
Decision Trees (J48, Random Forest)	Supervised Learning	Classifies navigation decisions based on obstacle attributes	Fast, interpretable, efficient	Training data-dependent, limited adaptability	Medium
VFH	Reactive / Local Planner	Creates a histogram grid from sensor data to determine safe directions for navigation	Fast, lightweight, suitable for dynamic	Can struggle in narrow or cluttered spaces, local minima risk	Medium
Vector Field Methods (APF, DWA, Force Field)	Reactive	Uses virtual forces or windows for path selection	Quick response, lightweight processing	Can get stuck in local minima	Low-Medium
Sense-and-AVoid	Reactive / Rule-Based	Uses simple rules to detect obstacles and alter path in real time	Simple to implement, low-cost, fast response	Limited in complex or cluttered environments	Low
Geometric method	Analytical / Model-Based	Estimates collision risk and safe angles using geometric projections, map-based	Deterministic, no training required, interpretable	Rigid assumptions, not adaptive to dynamics	Medium
PID-Based Reactive Avoidance	Control / Reactive	Uses a PID controller to adjust velocity or direction based on sensor input	Real-time control, simple tuning, compatible with basic hardware	Not predictive, can oscillate or fail in dynamic scenes	Low-Medium
Bug Algorithms (Bug 0, 1, 2)	Reactive / Proximity	Follows obstacle edges until path is clear	Simple, real-time	Not globally optimal, can be slow; always need to know where the target is	Low
SAD (usa DL) / MDE / Superpixel Depth	Vision-based	Estimates depth from camera images using ML or geometry	Rich visual data, no active sensor needed	Lighting-dependent, algorithmically heavy	Medium-High
Event Camera Algorithms (PTDS, CEF, LEM, DOT)	Vision / Neuromorphic	Detects changes in brightness for ultra-fast perception	Ultra-low latency, high motion accuracy	Requires specialized processing and fusion	High
Model Predictive Control (MPC)	Predictive Planning	Optimizes trajectory over time using prediction models, map-based (usato come controllore)	Forward-looking, optimal control, robust to noisy data	Heavy computational cost, complex setup	High
Data-Driven Risk-Aware MPC	Predictive / Learning-Based	Learns system behavior and risk from past data to improve MPC trajectory planning, map-based	Adapts to real-world dynamics, accounts for uncertainty, improves safety	Requires high-quality training data, more compute-intensive	High
Multi-Sensor Fusion (Bayesian, Kalman)	Sensor Fusion	Combines multiple sensor inputs for accurate decisions	Improved reliability and precision	Requires complex processing and calibration	High
Algorithm Fusion (MidasNet + Bug 0 etc.)	Hybrid	Combines ML-based depth with reactive navigation	Fast and accurate, adaptable	Complex implementation	High
Air Bumper Framework	Collision-Tolerant	Detects physical collisions and recovers dynamically	Effective in confined spaces, robust	Not for high-speed or dynamic obstacles	Medium
Local Planning (Voxel + B-Spline, TGBOA)	Local Planner	Creates short-term voxel maps and optimizes paths	Fast, trajectory-optimized, adaptive	No memory, not suited for fast obstacles	Medium
Global Planning (A*, Dijkstra, Swarm, Ant)	Global Planner	Plans entire path based on known map and cost functions	Optimized paths, avoids local traps	Computationally expensive, not reactive	High
Fuzzy Logic Control	Rule-Based / Adaptive	Applies fuzzy rules to handle obstacle avoidance under uncertainty	Handles uncertainty, interpretable logic	Requires expert tuning, lacks learning	Medium
Genetic Algorithms	Hybrid	Uses a genetic algorithm to evolve feasible trajectories, incorporating ray casting for obstacle avoidance and emergency landing detection	Highly flexible, supports multiple objectives, robust in complex maps, works on embedded hardware (e.g. Raspberry Pi)	Slower than reactive methods, not guaranteed to find optimal path, requires parameter tuning, assumes static environment	High
Sampling-Based Motion Planning (RRT, PRM)	Probabilistic Planning	Explores configuration space using random sampling	Good for high-dimensional space, scalable	Non-optimal, slow in dynamic scenes	High

Table 2.2: Summary table for obstacle-avoidance and navigation algorithms.

Chapter 3

Algorithmic and Sensory choices

The first design decision addressed in this work concerned the selection of algorithms to be used for path planning and obstacle avoidance. In this context, the **A*** algorithm was chosen, thanks to its ability to generate optimal trajectories even in complex scenarios. Unlike traditional global planner approaches that operate on predefined static maps, in this work A* was applied to a local and dynamic map, constructed and updated in real time from the drone's on-board sensors. This map remains constantly centred on the UAV and evolves as perceptions change, ensuring a consistently coherent and up-to-date representation of the surrounding environment.

To support the path planner, an hybrid algorithm belonging to the **Bug** family has also been introduced, designed as a backup strategy for managing situations in which A* encounters operational difficulties or fails to produce a valid trajectory. The integration of the two methods allows the optimality of global planning to be combined with local responsiveness, improving the overall robustness and reliability of the navigation system.

The sensor chosen was a **2D LiDAR**, which represents a good compromise between performance and operational requirements. This type of sensor offers a reliable behavior even in outdoor environments, ensuring good quality and density of the acquired data. Although heavier and more energy-intensive than cameras, LiDAR has been adopted not only for obstacle avoidance, but also for other perception and navigation tasks, which justifies its use as a robust and versatile solution within the overall system architecture.

The primary objective of the architecture developed is to implement a logic of obstacle avoidance in outdoor scenarios that operates selectively: the module intervenes only when an obstacle is detected, while for the rest of the time the drone

proceeds autonomously along the waypoints predefined by the *mission*, without external intervention on the trajectory control. This combines the safety guaranteed by obstacle avoidance with operational efficiency, reducing calculation times and energy consumption. This approach reflects a fundamental design principle: maximising the simplicity and linearity of the UAV's behavior, intervening with corrective strategies only when strictly necessary.

3.1 A*

The A* algorithm plays a prominent role in the literature on path planning and graph search. It was introduced in 1968 by Hart, Nilsson and Raphael at the Stanford Research Institute, at a time when artificial intelligence was taking its first steps and facing with the practical problems of autonomous navigation. A* was conceived as a synthesis between the completeness and optimality of *Dijkstra*'s algorithm and the speed of informed approaches based on the use of *heuristics*. This combination made it possible to develop a method capable of calculating optimal paths while reducing the number of nodes to be explored, a fundamental aspect especially in complex or large environments.

The principle behind A* consists of assigning each node a value that takes into account both the actual cost of the path travelled up to that point and an estimate of the remaining cost to reach the destination. In this way, the algorithm does not simply search for the shortest solution in absolute terms, but guides the search towards promising regions of the space, significantly speeding up the process.

From a formal point of view, the behaviour of A* is governed by the function:

$$f(n) = g(n) + h(n)$$

where $g(n)$ represents the actual cost accumulated from the initial node up to n, while $h(n)$ is a heuristic estimate of the minimum remaining cost to reach the goal. Heuristics are therefore functions that guides informed research: depending on the characteristics of the problem, they can be defined, for example, as *Euclidean*, *Manhattan* or *diagonal* distance between the current node and the target. If the heuristic is admissible, that is, it never overestimates the actual cost, the algorithm is guaranteed to find an optimal solution. In addition, if the heuristic is also consistent, i.e. it respects the triangular inequality

$$h(n) \leq c(n, n') + h(n')$$

for each pair of adjacent nodes n and n', then the algorithm achieves maximum efficiency. In this relationship $c(n, n')$ indicates the transition cost between two adjacent nodes and represents the "expenditure" associated with the shift: it can

coincide with the geometric distance between two points, but also model other magnitudes, such as travel time, energy consumed or a risk factor related to environmental conditions.

The operational functioning of A* involves the management of two sets: an *open list*, which contains the nodes to be explored, ordered according to the value of the function $f(n)$, and a *closed list*, which collects the nodes that have already been analysed. At each iteration, the node with the minimum value of $f(n)$ is selected, its neighbours are expanded by calculating their costs through the sum of the accumulated cost $g(n)$ and the transition cost $c(n, n')$, and the lists are updated accordingly. The process is repeated until the objective is achieved, at which point the optimal path is reconstructed by tracing back the predecessors.

One of the strengths of A* is its *generality*: the structure of the algorithm is independent of the application domain, as long as an appropriate heuristic is defined. This has made it a benchmark in a wide variety of sectors. In mobile robotics and drones, it is used to generate safe trajectories on maps that are continuously updated by on-board sensors.

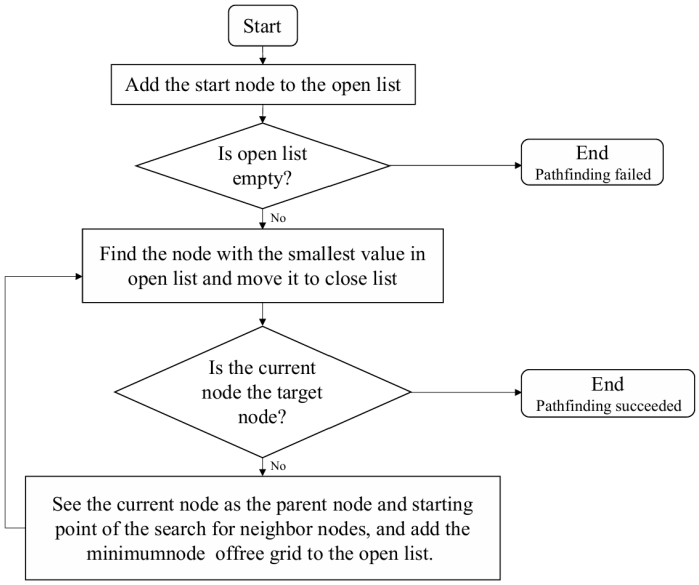

Figure 3.1: A* algorithm flow chart [21]

3.2 Bug

The Bug family of algorithms was introduced in the 1980s as one of the first formal strategies for addressing the problem of obstacle avoidance in mobile robotics.

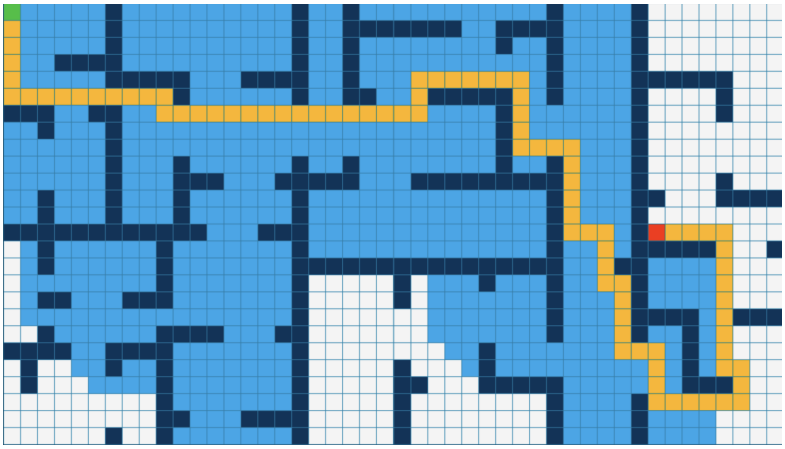

Figure 3.2: A* algorithm visualization [22]

Unlike many planners, Bugs do not rely on a predefined map of the environment, but operate exclusively on the basis of local information provided by on-board sensors. The robot therefore has no prior knowledge of the surrounding space: navigation is constructed step by step, reacting in real-time to the detection of obstacles.

The idea behind this family of algorithms is intuitive: the robot initially proceeds in a straight line towards the goal and, upon first contact with an obstacle, follows its contour until it satisfies an exit rule that allows it to resume its journey towards the goal.

The three fundamental concepts are:

- **Goal** (G), known in its absolute or relative position;
- **Hit Point** (H) which corresponds to the first interaction with the obstacle;
- **Leave Point** (H), i.e. the position on the edge from which to resume the path towards the goal.

The overall trajectory is therefore composed of straight segments alternating with *boundary-following* sections, along the edges of the obstacles.

A hybrid Bug navigation algorithm combining Bug0 and Bug2 is employed in this work. For completeness, also a brief summary of Bug1 is provided below.

3.2.1 Bug0

Bug0 represents the simplest version of the family: the robot moves towards the goal until it encounters an obstacle, at which point it activates the bypass phase.

The exit rule is very basic and can be summarised in the condition:

$$d(P, G) < g(H, G)$$

where $d(P, G)$ is the distance between the current position P and the goal, and $d(H, G)$ is the distance between the point of impact and the goal. Once this inequality has been verified, the robot leaves the edge and resumes its journey towards the goal.

Bug0 is extremely simple but not comprehensive: in complex scenarios, such as concave obstacles, it may fail to find a valid path.

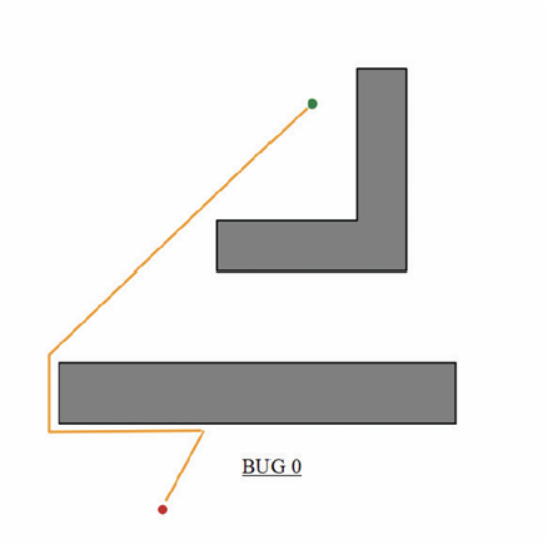

Figure 3.3: Bug0 example [5]

3.2.2 Bug1

Bug1 algorithm introduces a more structured strategy to determine the exit point. After reaching an impact point H, the robot travels along the entire contour of the obstacle, memorising the minimum distance from the goal:

$$d_{\min} = \min_{P \in \text{edge}} d(P, G)$$

The exit point L is then chosen as the point on the boundary that satisfies:

$$d(L, G) = d_{\min}$$

After completing the circumnavigation, the robot returns to L and from there resumes its march towards the goal.

Bug1 is complete: if a path exists, the algorithm will find it. However, the need to travel the entire perimeter of each obstacle often makes the paths much longer than necessary.

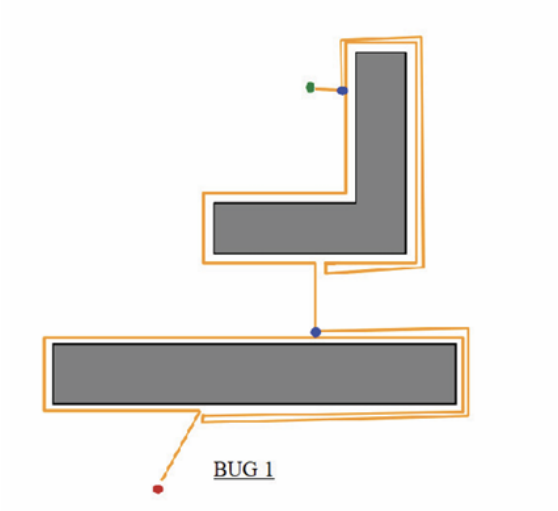

Figure 3.4: Bug1 example [5]

3.2.3 Bug2

Bug2 algorithm improves efficiency by introducing the so-called *M-line*, i.e. the straight line connecting the starting point S with the goal G. After encountering an obstacle, the robot starts following the edge; the exit rule is that the robot can leave the obstacle when:

$$P \in \text{M-line} \ \wedge \ d(P, G) < d(H, G)$$

In other words, the robot leaves the edge as soon as it finds itself on the M-line at a point closer to the goal than the point of impact.

This choice significantly reduces the number of complete paths around obstacles, improving efficiency compared to Bug1. Bug2 is also complete, but may encounter difficulties in particular geometric configurations.

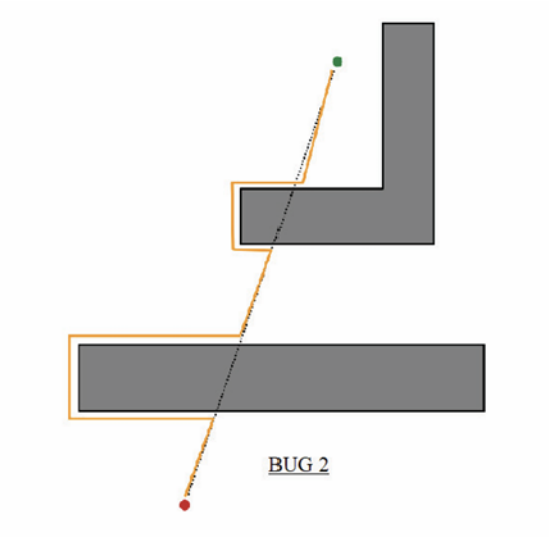

Figure 3.5: Bug2 example [5]

Chapter 4

Software Tools and Simulation Environments

The communication architecture adopted in this work is modular and distributed: each component plays a well-defined role, but at the same time it collaborates in real time with the others, guaranteeing scalability, robustness and flexibility to the overall system.

The logical core is **ROS 2**, which handles data processing, scheduling execution and obstacle avoidance algorithms, as well as transmission of high-level commands. The simulation environment is entrusted to **Gazebo**, which reproduces realistic three-dimensional scenarios and virtual sensors, while **RViz** acts as a visualization tool, allowing to monitor the state of the system and the evolution of the information processed by the nodes.

Low-level control is delegated to **PX4 Autopilot**, the flight stack that translates the received commands into concrete actions on the drone's actuators. Information exchange between ROS 2 and PX4 occurs via *uXRCE-DDS* middleware, which connects PX4's internal messaging system *(uORB)* with the ROS ecosystem, ensuring low-latency two-way communications. For managing high-intensity data flows, *Gazebo–ROS 2 Bridge (ros_gz_bridge)* is used, allowing sensory measurements to be transferred directly from the simulator to the perception framework without going through the flight controller. In this way, a more rapid and efficient elaboration of available resources is reached.

Completing the architecture is the **Ground Control Station (GCS)**, implemented via **QGroundControl**, which ensures supervision and human intervention. Through this interface, it is possible to monitor the status of the drone, modify operational parameters, and, if necessary, take manual control of the UAV.

The overall structure, illustrated in Figure 4.1, shows how these modules interact with each other.

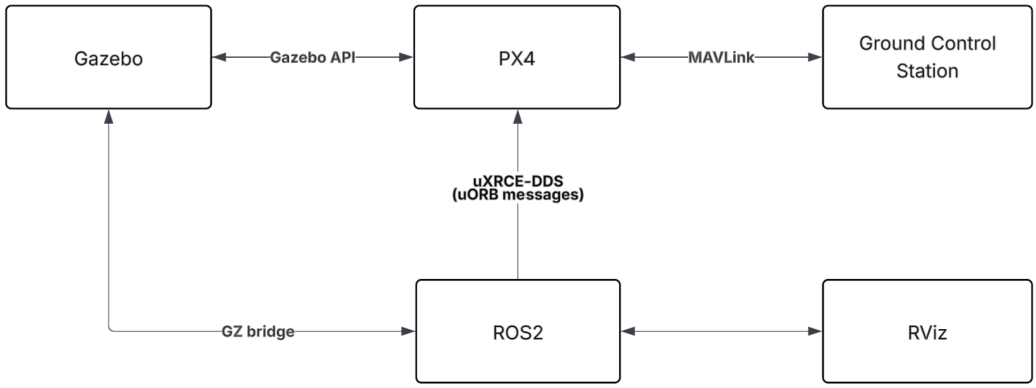

Figure 4.1: General Communication Architecture

4.1 ROS2 Framework

ROS 2 (Robot Operating System 2), [23], is currently one of the most widely used and established software platforms to develop complex robotic systems. It is an open-source framework designed to simplify the creation and integration of applications in which different software modules, such as sensors, perception algorithms, planners and controllers, must cooperate within a single architecture.

The principle behind ROS 2 is *modularity*: instead of building a monolithic system, each functionality is encapsulated in an independent software unit called **node**. The set of running nodes forms a *Computation Graph*, in which each node can communicate with the others by exchanging information. This approach ensures flexibility and reusability, since it allows individual components to be replaced, updated, or integrated without having to modify the entire system.

One of the most significant improvement over *ROS 1* is the adoption of *DDS (Data Distribution Service)* middleware as the communication layer. DDS not only offers configurable *Quality of Service (QoS)* to manage latency, update frequency and reliability, but also introduces an automatic discovery mechanism: once started, each node announces its presence on the network and detects the ones that are already active, establishing transparent connections with those that publish or subscribe to the same topics. This eliminates the need for manual configuration or a central coordinator (master), making ROS 2 inherently scalable and suitable for distributed scenarios, Figure 4.2.

The ROS 2 **Humble Hawksbill** distribution was used in the present work, released in May 2022 and classified as *Long Term Support (LTS)*, with updates guaranteed until 2027. Humble is currently one of the most adopted versions in

both the academic and industrial fields, thanks to its stability and compatibility with the most advanced navigation packages, in particular the Nav2 framework, used in this project (it will be explained in section 5.1).

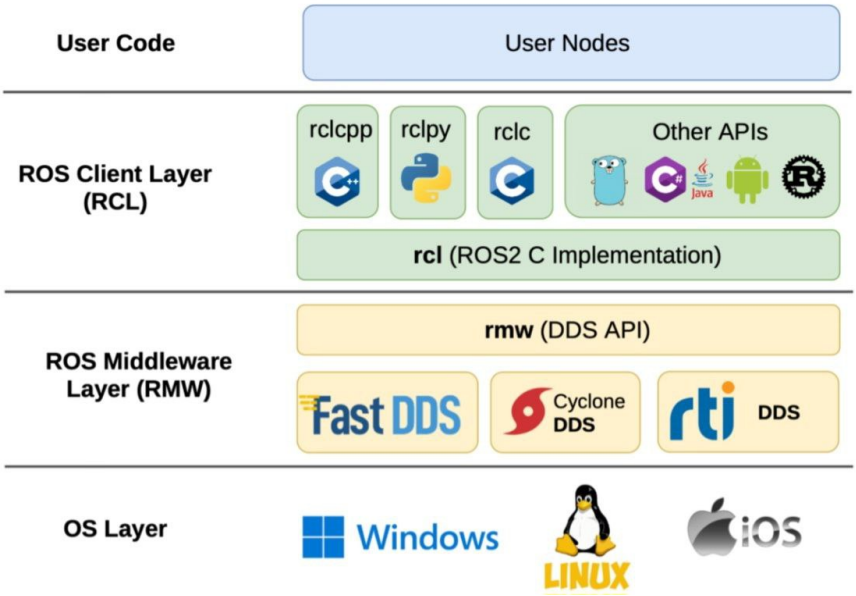

Figure 4.2: ROS2 Layers [24]

4.1.1 Node

The node concept is the basis for the operation of ROS 2. Each unit can be seen as an independent process that performs a very specific task, like reading data from a sensor, building a map of the environment, planning a path, or sending motion commands to the actuators. Even if an autonomous robot may seem a single complex entity, it is nothing more than the result of the cooperation of many nodes that work together and continuously exchange information.

A central aspect is that these nodes do not necessarily have to be on the same machine. In fact, thanks to DDS middleware, they can be distributed across multiple devices, for example an on-board computer mounted on the drone, a networked development workstation or even external edge systems, and communicate with each other in a completely transparent way.

4.1.2 Communication between Nodes

Communication between nodes in ROS 2 is articulated in three complementary modes, which allow to cover the entire spectrum of possible interactions in a robotic

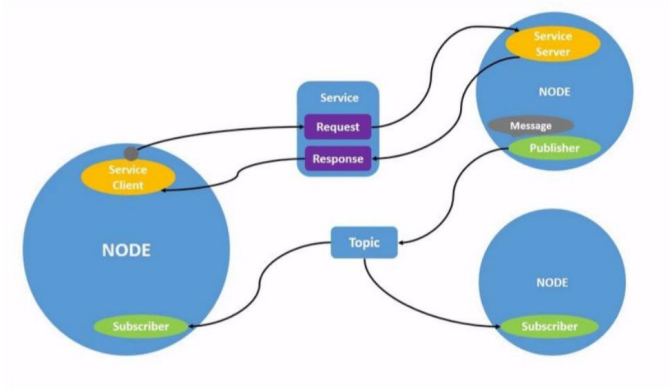

Figure 4.3: Node [24]

system.

The most widespread form is the one based on **Topics**, Figure 4.4, which implement a *publish/subscribe* model: a node can publish a data flow on a specific channel, while all nodes subscribed to that channel receive them in real time. This asynchronous and scalable mechanism is ideal for the continuous transmission of information such as sensory measurements or control commands.

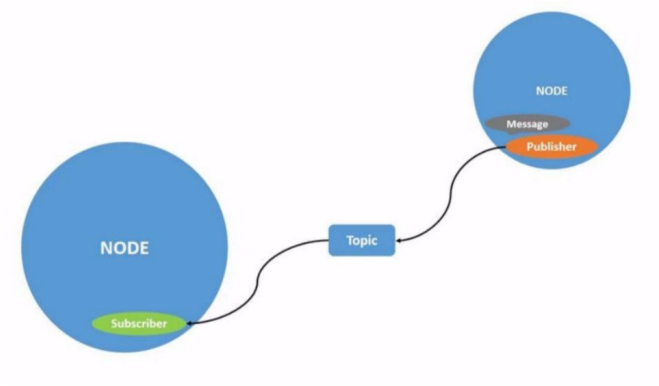

Figure 4.4: Topic [24]

Alongside this model, ROS 2 provides **Services**, Figure 4.5, structured according to the *request/response* paradigm. They are suitable for point interactions, where a node sends a specific request and receives an immediate response, similar to a remote function call.

Finally, to manage longer-duration tasks that require continuous monitoring,

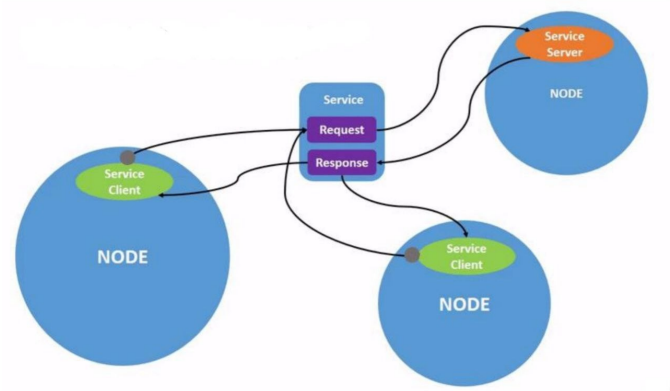

Figure 4.5: Service [24]

Actions are used, Figure 4.6. Through this mechanism, a node can send a goal, receive periodic feedback on progress, and obtain a final result, with the possibility of interrupting or modifying the in progress task.

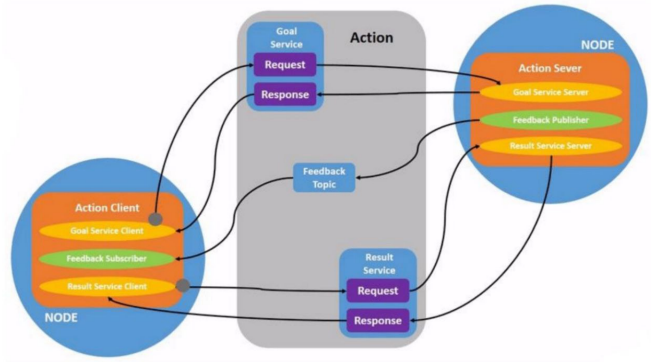

Figure 4.6: Action [24]

Data exchanged through topics, services or actions is formalized into typed **messages**, defined in *.msg*, *.srv* or *.action* files. This strongly typed approach ensures interoperability and reliability between components produced by different developers.

4.2 Gazebo

Gazebo, [25], represents one of the reference simulators in robotics, designed to offer a realistic three-dimensional environment in which analyzing and verifying the behavior of robots and algorithms before their application in the field. The simulator

allows to model complex virtual worlds, populated by robots, obstacles, and objects, within which the performance of a system can be observed and evaluated under controlled but highly plausible conditions.

One of Gazebo's strengths lies in its ability to reproduce physical dynamics accurately, thanks to integration with several physics engines (including ODE, Bullet and DART). In this way, it is possible to simulate forces, torques, frictions, and collisions, returning a behavior that faithfully reflects the real one. Added to this is the availability of a vast library of virtual sensors, such as LiDAR, monocular and stereo cameras, IMU and GPS, capable of generating data similar to that collected by physical sensors, an essential element for the validation of perception and planning algorithms.

The connection with ROS 2 is done through dedicated plugins, which allow the simulator to be integrated within the framework architecture. In this way, the measurements of the simulated sensors can be published as ROS topics, while the commands generated by the control nodes are sent to the robot models, thus reproducing the perception–decision–action cycle characteristic of an autonomous system.

4.3 RViz

RViz, [26], is the official ROS 2 3D visualization tool and is designed to provide the user with an intuitive and interactive representation of the state of the robotic system. Unlike Gazebo, which simulates physical and sensory dynamics, RViz focuses on visualizing the data produced by ROS nodes, allowing real-time monitoring of what the robot perceives and how it interprets its surroundings.

The graphical interface of RViz allows a great variety of information to be displayed: maps generated in real time, point clouds from LiDAR sensors, images from monocular or stereo cameras, trajectories planned by navigation algorithms and even the status of the robot's actuators. Each data is shown as a selectable and configurable information level, offering a high degree of customization depending on the needs of the experimentation.

A further strength is the ability to visualize multiple data streams simultaneously, favouring comparative analysis and diagnostics. In this sense, RViz is not only a debugging and analysis tool, but also a support to the demonstration of results, since it makes complex processes visible and understandable .

Thanks to these characteristics, RViz is an essential complement to Gazebo, offering a clear and detailed vision of the information that guides the robot's behavior.

4.4 PX4 Autopilot

PX4, [27], is an open-source *flight stack* that represents one of the reference standards for the control of remotely piloted aerial vehicles. Created to guarantee reliability and versatility, it is now used not only on multicopter drones and fixed-wing aircraft, but also on hybrid VTOL platforms, land rovers and underwater vehicles.

The core of the system is made up of the **flight controller**, which runs the PX4 firmware on a real-time operating system (typically *NuttX*) and which can be considered to all intents and purposes the "mind" of the vehicle: it receives measurements from the on-board sensors (IMU, GPS, magnetometers, barometers), and translates control commands into actions on the actuators, ensuring stabilization, safety, and automatic functions such as take-off, landing or position holding.

The controller is often supported by a **companion computer**, an external computing unit that runs generic operating systems, such as Linux, and which allows to implement advanced features that cannot be managed directly by the flight controller: visual processing, network communications, complex mission planning. In this context, **Offboard mode** is inserted: a flight mode of PX4 that enables the companion computer to send high-level commands (position, speed, acceleration or attitude setpoint) directly to the controller. This mode is fundamental for obstacle avoidance applications, since it permits to integrate external perception and planning algorithms, leaving the tasks of stabilization and immediate control of the engines to PX4. For safety reasons, Offboard activation requires continuous reception of a *"proof of life"* signal at a minimum frequency of 2 Hz: in the case of an interruption, PX4 immediately disables external control and activates a *failsafe procedure* (such as automatic landing or return to the take-off point).

For code development and validation, PX4 also makes **Software-In-The-Loop (SITL)** simulator available, which allows the entire flight stack to be run on a computer without resorting to physical hardware. In this way, it is possible to simulate the dynamics of a vehicle and safely test both the PX4 firmware and the external integrated algorithms via ROS 2.

Internal communication between controller modules occurs through the *uORB publish/subscribe* architecture, while data exchange with ROS 2 is made possible by the *uXRCE-DDS middleware*, which extends uORB messages to the DDS domain, Figure 4.7. This mechanism, implemented according to a client–agent architecture, ensures data transparency: messages published by PX4 become available as a ROS 2 topic and vice versa.

A crucial aspect in the integration between ROS 2 and PX4 relates to the *reference systems conventions* adopted by the two environments. PX4 uses the **NED (North–East–Down)** system for the world frame and **FRD (Front–Right–Down)**

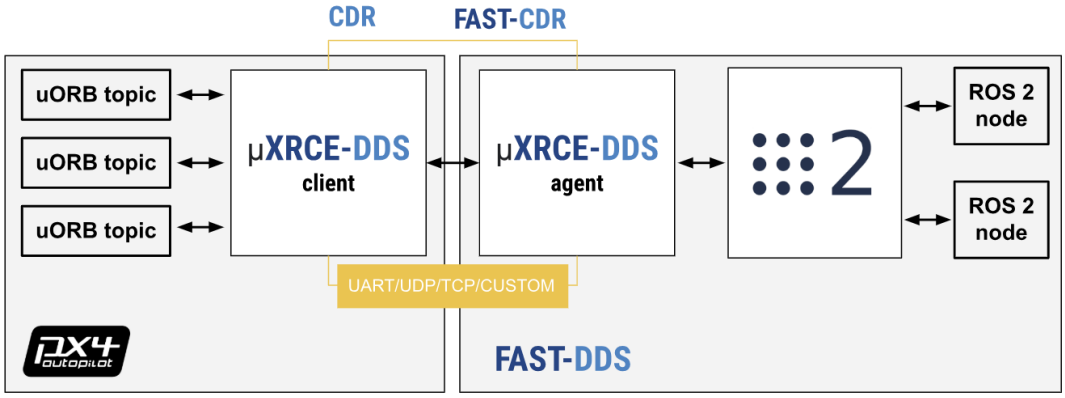

Figure 4.7: uXRCE-DDS communications middleware [27]

for the body frame as the standard: in this convention the X axis points north (or forward in the body), the Y axis eastward (or rightward in the body), and the Z axis downward. ROS 2, in contrast, adopts the **ENU (East–North–Up)** convention for the world and **FLU (Front–Left–Up)** for the body: here the X axis is directed to the East (or forward in the body), the Y axis to the North (or left in the body), and the Z axis upwards. These differences necessitate explicit coordinate transformations whenever data such as positions, velocities, or attitude are exchanged between the two systems.

$$R_{\text{NED}\to\text{ENU}} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Rotation matrix from NED to ENU.

$$R_{\text{FLU}\to\text{FRD}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Rotation matrix from FLU to FRD.

The transformation of a vector from the NED system to the ENU one can be achieved by applying two successive rotations: first a rotation of $\pi/2$ around the Z axis (oriented downwards in the NED frame), followed by a rotation of π around the X axis, which initially coincides with the North direction and instead becomes East after the first transformation. It is relevant to underline that these two operations are, from a mathematical point of view, fully equivalent. On the other hand, regarding the conversion between the FLU and FRD body frames, the required operation is simpler and consists of a rotation of π around the X axis (forward).

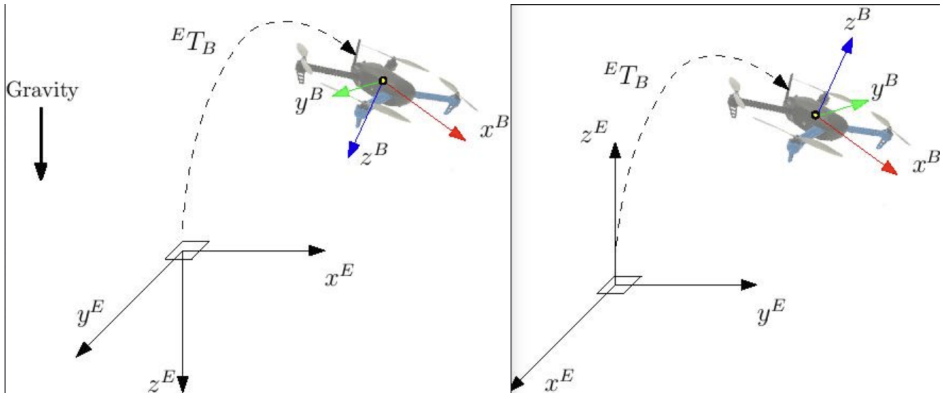

Figure 4.8: PX4- ROS 2 frame conventions (FRD on the left/FLU on the right) [27]

4.5 QGroundControl

QGroundControl (QGC) is one of the most popular and complete Ground Control Stations for PX4 and MAVLink systems. This is open-source, cross-platform software (Windows, macOS, Linux, Android, iOS), designed to provide the user with a unified vehicle monitoring, configuration and control interface.

Functionally, QGroundControl plays several key roles. Firstly, it allows the initial configuration of the drone, including the calibration of sensors (IMU, magnetometers, GPS) and the definition of flight parameters. Then, it offers a graphical interface for **mission planning** via *waypoints*, which can include instructions such as quota changes, payload trigger commands, or flyby areas for mapping missions.

During flight, QGroundControl acts as a real-time supervision tool, showing detailed telemetry (position, attitude, speed, battery status, GPS signal quality, etc.) and allowing the operator to take action if necessary. Among its most relevant features there is support for failsafe management, which allows to select automatic actions (for example, return to the starting point or immediate landing) when critical conditions are detected.

Another distinctive aspect is the integration with standard communication protocols such as **MAVLink**, which ensure interoperability with a wide range of autopilots and UAV systems. QGroundControl also supports advanced workflows: manual piloting via USB joystick or on-screen virtual sticks, RC transmitter setup and calibration, and simultaneous monitoring of multiple vehicles. It integrates smoothly with simulators like PX4 SITL, making it widely used in both research and operational applications.

In the context of this thesis, QGroundControl has been employed as the main interface for the supervision of the drone, allowing to monitor in real time the

state of the system during simulations, to intervene with any manual corrections or changes to the operating parameters and, above all, to define and assign mission waypoints to the vehicle, Figure 4.9.

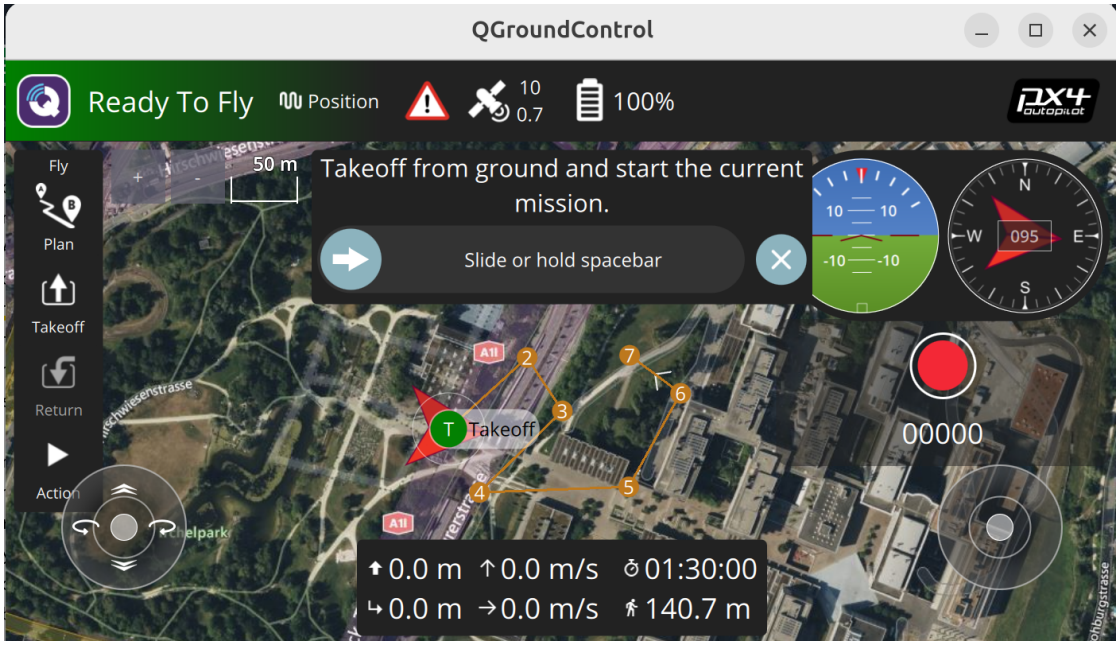

Figure 4.9: QGroundControl interface

Chapter 5

First Implementation

The first implementation of the obstacle avoidance system was developed and tested in simulation using the mobile robot **TurtleBot3 Burger**. This platform was chosen as a preliminary test bed as it allows a simplified but effective environment for validating perception and navigation logics. In this initial phase, the **PX4 flight controller**, whose integration was introduced only later within the simulation, with the objective of reproducing the dynamics of an UAV more faithfully, was not used.

The use of TurtleBot3 has made it possible to quickly and effectively verify the planning and obstacle avoidance logic, postponing the management of the complexities connected to flight control to the subsequent phases. Once the algorithms were consolidated in this simplified context, the architecture was transferred to a configuration more faithful to the application domain, based on the PX4, so as to evaluate the system behavior in conditions closer to the operational ones.

This chapter presents the local navigation and collision avoidance architecture developed for the simplified case of TurtleBot3 Burger in ROS 2, designed to combine reactive readiness and reliability in partially known scenarios. The solution integrates **A***-based local scheduling on rolling costmap (centered and integral with the robot provided by **NAV2**), a **DWB** local controller, always from NAV2, used as a *Controller Server* plugin, and a **Bug** responsive behavior that acts as a fallback in case of failed scheduling/control. The management of states and transitions is entrusted to an orchestration node, **NavigationManager**, while the achievement of the goal is monitored by a **GoalChecker**.

The complete pseudocodes of A* and Bug are given in this chapter (see respectively 3 and 2); the pseudocodes of GoalProjector, PathFollower, GoalChecker and NavigationManager are instead collected in Appendix A in order to keep the main discussion focused on the two pivotal algorithms. All configurable parameters (thresholds, gains, frequencies, topic and action server identifiers) are managed via

a **YAML file** loaded at startup, allowing non-intrusive calibration of the pipeline.

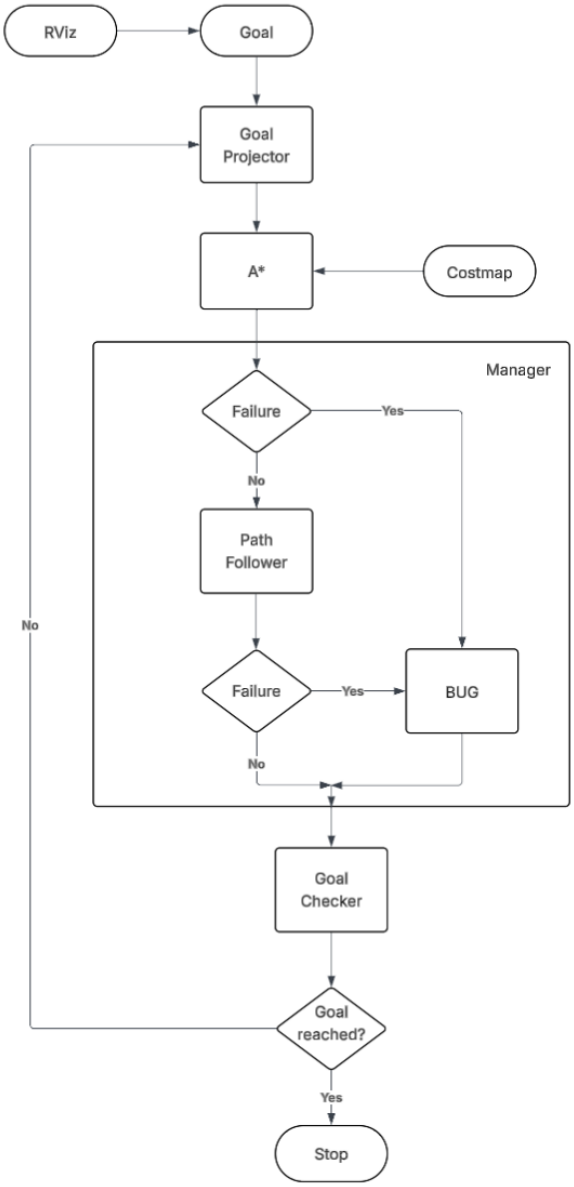

Figure 5.1: Simplified flowchart of Turtlebot nodes

5.1 NAV2

Nav2 (Navigation2) is the navigation stack for ROS 2 designed for mobile robots. It provides, in a **modular and modular way**, the fundamental functions of route planning, movement control, and recovery strategies, coordinating them via a *Behavior Tree*. The architecture is **plugin**-based: planning and control servers load interchangeable algorithms (eg. planners and local controllers), each supported by its own Costmap2D with layers that shape the environment around the robot. Nav2 exposes standard actions (e.g. NavigateToPose, ComputePathToPose, FollowPath), integrates with TFs, sensors (LiDAR/odometry), maps and localization, and is configurable via YAML files. In summary, it offers a robust and flexible framework for switching from lens pose to speed commands, adapting to different platforms and scenarios [28].

In the present work, NAV2 is employed for two essential functions:

1. **Local costmap (Costmap2D, rolling window)**: A robot-centric costmap maintained from 2D LiDAR and odometry, with layers (e.g., obstacle, inflation) that form the perception basis for local planning and control.

2. **Controller Server with DWB plugin**: Exposes the `FollowPath` action to track a `nav_msgs/Path`. The DWB (Dynamic Window Based) controller samples dynamically admissible (v, ω), forward-simulates short trajectories, and scores them with critics (path/goal progress, obstacle clearance, smoothness), selecting the command with the best weighted score. Using it as a plugin gives a mature, YAML-configurable controller without changing its internals.

5.2 Local Costmap, Occupancy Grid, Costs and Conversions

Local planning rests on a **"rolling" window** costmap centered on the robot: the grid translates together with the vehicle, so that it always remains in the center of the operational box. The costmap is published to `/local_costmap/costmap` in the format `nav_msgs/OccupancyGrid`, i.e. a two-dimensional regular grid described by a *resolution* r (meters per cell), by *size* `width` $\times$ `height`, by an **origin o** $= (o_x, o_y)$ and from a reference *frame* (`frame_id`). In all experiences the frame of the costmap is `odom` (parameter `global_frame` in NAV2), consistent with the TF chain `odom` $\rightarrow$ `base_link`. The same frame is maintained in derived messages (Path, PoseArray, PoseStamped) and set as `Fixed Frame` in RViz to avoid misalignments between data, display and control.

The costmap is updated in real time by merging the measurements of the LiDAR 2D (ray-tracing: the rays free the cells crossed and mark the impact cell as occupied)

with the **odometry** (which anchors spatially and temporally the grid). NAV2 can apply several *layers*; crucial among these is the **inflation layer**, which expands obstacles and creates a decreasing cost gradient with distance: a "bearing" which orients the controller to prefer trajectories with greater **clearance**.

5.2.1 Internal Costs and Conversion to Occupancy

Internally `costmap_2d` represents each cell with a *cost* $c \in \{0, \ldots, 255\}$ (type `uint8_t`) with special codes:

- 0=`FREE_SPACE` (free);
- 253=`INSCRIBED_INFLATED_OBSTACLE`;
- 254=`LETHAL_OBSTACLE`,
- 255=`NO_INFORMATION` (unknown).

Values $1 \ldots 252$ model a risk gradient (largely generated by the *inflation layer*).

When the costmap is exported as `nav_msgs/OccupancyGrid` for interoperability/display, the internal costs are mapped to a **occupancy estimate** in $-1/0$–100.

A remapping consistent with the special codes is:

Costmap → OccupancyGrid

- if $c = 255 \Rightarrow \text{occ} = -1$ (unknown);
- otherwise $\text{occ} = \text{round}\Big(\min(100, (c/254) \cdot 100)\Big)$.

In particular: $0 \to 0$, $254 \to 100$.

OccupancyGrid → Costmap

- if $\text{occ} = -1 \Rightarrow c = 255$ (unknown);
- otherwise $c = \text{round}\Big((\text{occ}\,/100) \cdot 254\Big)$.

In particular: $0 \to 0$, $100 \to 254$.

This distinction is important: the **internal costmap** is a field of *costs* used by planners/controllers; the **OccupancyGrid** is a *occupancy view* useful for interoperability and RViz. The selection of which cells participate in planning is not a property of the costmap itself but of the planner that consumes it; the corresponding criteria are introduced in section 5.3.

5.2.2 Inflation profile: Radius and Cost Decay

The *inflation layer* defines how far from the obstacle the "penalized" zone extends and how quickly its cost decays. Two YAML parameters govern the profile:

- **inflation_radius** (R): maximum radius of influence; beyond R the cost returns to zero.
- **cost_scaling_factor** (s): coefficient of the exponential decay; the higher the s, the steeper is the gradient (the cost drops in less space).

Indicating with r the minimum distance to the edge of the obstacle and with $r_{\text{inscribed}}$ the radius "inscribed" (depends on the robot footprint), the profile of the **internal costs** is

$$c(r) = \begin{cases} 254, & r \le r_{\text{inscribed}} \quad \text{(lethal zone/inscribed)}, \\ 1 + (254-1)\,\exp\Big(-s \cdot (r - r_{\text{inscribed}})\Big), & r_{\text{inscribed}} < r \le R, \\ 0, & r > R. \end{cases}$$

The `OccupancyGrid` view is obtained (for $r \le R$) as $\text{occ}(r) = \text{round}\Big(100 \cdot c(r)/254\Big)$. Two useful decay scale measures are

$$\Delta r_{1/2} = \frac{\ln 2}{s} \qquad \text{e} \qquad \Delta r_{95\%} = \frac{\ln 20}{s}.$$

These relationships help select s based on the "width" of the desired bearing: for the same R, s large produces a narrow, steep edge; s small produces a wider, softer edge.

5.2.3 Geometry: World ↔ Map and indexing

To align the continuous poses and the discrete grid, the usual conversions are used (in the `odom` frame):

- **world → map**: $m_x = \lfloor (x - o_x)/r \rfloor$, $m_y = \lfloor (y - o_y)/r \rfloor$ (valid if $0 \le m_x <$ `width` and $0 \le m_y <$ `height`);
- **map → world** (cell center): $x = o_x + (m_x + 0.5)\, r$, $y = o_y + (m_y + 0.5)\, r$;
- **indexing** of the vector `data` (row-major): `idx` $= m_y \cdot$ `width` $+ m_x$.

Keeping the `frame_id` (`odom` consistent in this work) along the entire pipeline - costmap, pose, path - avoids reference errors and allows scheduling and control to operate on the same metric and coordinate basis.

5.3 Goal Projector

The *GoalProjector* has the task of always making the target reachable for local planning. Since A* works within the local costmap window, it may happen that a global goal falls outside this box. The node then receives the target at `/goal_pose` from RViz, the current pose at `/odom`, and the local costmap at `local_costmap/costmap/`, and periodically posts on `/projected_goal` a projected goal that falls within the costmap. Everything happens in the `odom` frame and with consistent headers, so that planner and controller see the same reference.

The logic is intentionally simple and robust. With each new objective (or when the timer set by `public_rate` expires), the node considers the straight line that joins the current pose to the goal and samples it with a step equal to approximately half the resolution of the costmap: in this way, advancing *"in small jumps"* along the desired direction, it identifies the first valid point that falls within the `width` $\times$`height` limits. If the straight line is barred from obstacles, moderate angular deviations are tried around the main direction ($\pm 15°$ and $\pm 30°$), until a map-compatible candidate is found. If no direction generates a valid point, the node does not publish updates and logs the event in the logs.

The validity check is perfectly aligned with the one then used by A*: the continuous point (x, y) is converted into cell indices (mx, my) using the origin and resolution of the costmap; after checking the limits, the linear index `idx`=`my`$\times$`width`+`mx` allows to read the value in the `data` vector. In this system, a cell is *walkable* if `data == 0` (free) or `data == -1` (unknown/unknown); it is *not walkable* if `data > 0` (occupied or penalized by inflation). In this way the Goal Projector never proposes objectives that A* would reject.

Since the costmap is rolling and moves with the robot, the projected goal is updated periodically: as the robot advances, the local target "slips" with the window, keeping planning stable and continuous. When the node receives `goal_reached` on `/goal_reached`, it suspends publishing to prevent unnecessary activity. In summary, the Goal Projector introduces a simple receding horizon: it transforms a distant objective into a succession of local micro-goals, always compatible with the costmap, which A* can actually plan and the controller can follow.

5.4 A*

The node *AStarPlanner* accomplishes a local on grid schedule within the window of local costmap in the frame `odom`. Upon startup it declares the topics configurable by YAML and opens subscriptions to `/local_costmap/costmap`, `/projected_goal`, `/odom`, `/goal_reached`, as well as publishers to controller and RViz (`/astar_path` as `PoseArray`, `/astar_path_visual` as `Path`) and to the flag of outcome (`/astar_`

`failed` as `Bool`).

The planner works on the `nav_msgs/OccupancyGrid` published by the local costmap (scale −1/0–100), not on the internal structure `costmap_2d`; from the message reads *dimensions*, *resolution*, *origin* and `frame_id`, which is then propagated unchanged in the output messages to ensure consistency of reference; from odometry it constructs a `PoseStamped` of the current pose. When a new projected goal arrives, it saves it as a target; if the previous goal has already been reached, it ignores any new identical projections by 0.1 m (debouncing). The schedule is punctuated by a timer with period 1/`publish_frequency`: with each activation the node checks that costmap, pose and goal are available and that it is not active the goal flag reached, otherwise it comes out with a log.

The path calculation occurs with a classical A* on a **8-connected** graph. The initial pose and goal are first projected onto the grid using the functions *world→map*, explained in section 5.2; the search space coincides with the costmap window. The algorithm maintains the usual structures (`open_set`, `came_from`, `g_score`, `f_score`): the heuristic is the **Euclidean distance** to the goal cell (admissible in 8-connected) and the **edge cost** is the distance between adjacent cells (1 for orthogonal moves, $\sqrt{2}$ by diagonals). At each iteration, the node with the lowest `f_score` is selected. If it is the goal, the path is reconstructed by backtracking through predecessors; otherwise, the eight neighbors are generated, filtered for traversability as defined in section 5.3, and `f_scores` and predecessors are updated when an improvement is found. If the open set is exhausted, the search fails.

The discrete path is then transformed into a *continuous path*: for each cell the center is calculated in world coordinates (*map→world*) and a `PoseStamped` is composed with dimension `z=0`. The orientation of each pose is aligned to the direction to the next cell: the **yaw** is from `atan2(dy,dx)` and the quaternion is set by valuing `z = sin(yaw/2)` and `w = cos(yaw/2)`. Poses are published as `PoseArray` on `/astar_path` (controller input) and as `Path` on `/astar_path_visual` (for RViz). In parallel the node updates `/astar_failed`: `True` when there is no path (or the list of poses is empty), `False` when planning is successful. Upon receiving `/goal_reached` the planner deactivates and resets the internal goal, avoiding unnecessary recalculations.

Algorithm 1: AStarPlanner Node

1 **Subscribers:**
`/local_costmap/costmap` → COSTMAP_CALLBACK (OccupancyGrid)
`/projected_goal` → PROJECTED_GOAL_CALLBACK (PoseStamped)
`/odom` → ODOM_CALLBACK (Odometry)

`/goal_reached` → GOAL_REACHED_CALLBACK (Bool)
Publishers:
`/astar_path` ← PoseArray
`/astar_path_visual` ← Path
`/astar_failed` ← Bool
Parameters:
`publish_frequency`, `topic_astar_path`, `topic_astar_visual`, `topic_astar_failed`
`topic_costmap`, `topic_projected_goal`, `topic_odom`, `topic_goal_reached`
Initialization:
Create pubs/subs; start timer with period 1/`publish_frequency`
State: `costmap_ready`←False, `current_pose`←None, `global_goal`←None
`latest_path`←None, `goal_reached`←False
Callback COSTMAP_CALLBACK(OccupancyGrid msg):
Store map data/size/resolution/origin/frame_id; set `costmap_ready`←True
Callback ODOM_CALLBACK(Odometry msg):
Build PoseStamped from header+pose → `current_pose`
Callback PROJECTED_GOAL_CALLBACK(PoseStamped goal_msg):
if `goal_reached` **and** _POSES_EQUAL(goal_msg.pose, `global_goal`.pose) **then**
  return ▷ ignore identical projection after goal reached
end if
`global_goal`←goal_msg; `goal_reached`←False
Callback GOAL_REACHED_CALLBACK(Bool msg):
if msg.data **then**
  `goal_reached`←True; `global_goal`←None
end if
Timer thread TIMER_CALLBACK():
if `goal_reached` **then**
  return
end if
if not `costmap_ready` **or** `current_pose`=None **or** `global_goal`=None **then**
  return
end if
path_msg ← COMPUTE_PATH(`current_pose`, `global_goal`)
if *path_msg* is None **or** $|path\_msg.poses| = 0$ **then**
  publish Bool(True) on `/astar_failed`; **return**
else
  publish Bool(False) on `/astar_failed`
  PUBLISH_PATH_AS_POSE_ARRAY() on `/astar_path`
  Build Path for RViz and publish on `/astar_path_visual`
end if
Function _POSES_EQUAL(p_1, p_2, tol): **return** $\sqrt{(p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2} < tol$
Function WORLD_TO_MAP(x, y):

$mx = \lfloor (x - origin\_x)/resolution \rfloor$; $my = \lfloor (y - origin\_y)/resolution \rfloor$; **return** (mx, my)

Function MAP__TO__WORLD(mx, my):
$x = origin\_x + (mx + 0.5) \cdot res$; $y = origin\_y + (my + 0.5) \cdot res$; **return** (x, y)

Function IS__FREE(mx, my):
if $0 \leq mx < width$ **and** $0 \leq my < height$ **then**
$val \leftarrow$ `map_data`$[my \cdot width + mx]$; **return** $(val = 0)$ **or** $(val = -1)$
else
return False
end if
Function HEURISTIC(a, b): **return** $\sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}$
Function COMPUTE__PATH(start:PoseStamped, goal:PoseStamped) ▷ A* on 8-neighbors
$s \leftarrow$ WORLD__TO__MAP($start.x, start.y$); $g \leftarrow$ WORLD__TO__MAP($goal.x, goal.y$)
$open \leftarrow \{s\}$; $came\_from \leftarrow \emptyset$; $g\_score[s] \leftarrow 0$; $f\_score[s] \leftarrow$ HEURISTIC(s, g)
while $open \neq \emptyset$ **do**
choose $c \in open$ with minimal $f\_score[c]$
if $c = g$ **then**
return BUILD__PATH__MESSAGE(RECONSTRUCT(came_from,s,g))
end if
remove c from $open$
for each neighbor n of c (8-connected) **do**
if not IS__FREE(n) **then**
continue
end if
$\tilde{g} \leftarrow g\_score[c] +$ HEURISTIC(c, n)
if $\tilde{g} < g\_score[n]$ **then**
$came\_from[n] \leftarrow c$; $g\_score[n] \leftarrow \tilde{g}$; $f\_score[n] \leftarrow \tilde{g} +$ HEURISTIC(n, g)
add n to $open$
end if
end for
end while
return None
Function BUILD__PATH__MESSAGE(cells):
Create Path with frame_id; for each (mx, my) map to (x, y), estimate yaw, set quaternion; append
Update `latest_path_poses`; **return** Path
Procedure PUBLISH__PATH__AS__POSE__ARRAY():
Build PoseArray (stamp now, frame_id) from `latest_path_poses` and publish on `/astar_path`

5.5 Path Follower - DWB

The *PathFollower* node plays the role of a **bridge** between the geometric path produced by A* and the NAV2 local controller. At input it subscribes to `/astar_path` (`PoseArray`) and, for each update, reconstructs a `nav_msgs/Path`: it copies the header and encapsulates each `Pose` of the `PoseArray` into a `PoseStamped`, preserving the consistency of frames and timestamps. The path thus obtained is sent to the **action server** `/follow_path` by means of a `ActionClient` on the action `nav2_msgs/action/FollowPath`. In the goal message sets `controller_id = 'FollowPath'` and `goal_checker_id = 'goal_checker'`, participates in accepting the target and monitors the outcome: in case of `STATUS_ABORTED` it publishes `/controller_failed` = `True`, otherwise `False`. The node also exposes a publisher on `/cmd_vel` (used only by `public_stop()` to send a null `Twist`) and logs main states (server wait, path send, outcome).

From a control point of view, the node does not calculate speed commands directly: it delegates this responsibility to the *DWB (Dynamic Window Based)* controller, used as a *plugin* of NAV2's Controller Server and selected via `controller_id`. In short, DWB implements a variant of the *Dynamic Window Approach*: at each cycle it samples pairs (v, ω) within the dynamic window defined by the current speed and acceleration/braking constraints of the robot; for each sample it simulates forward (short horizon) the trajectory induced by the differential kinematic model; then evaluates and punctuates each trajectory by means of a set of critics (e.g. obstacle distance/footprint on the costmap, path alignment, goal proximity, regularity), typically combined in a function of the type

$$J(\tau) \;=\; \sum_i w_i \, C_i(\tau),$$

where C_i are the scores/penalties of the individual *critics* and w_i are the configurable weights in the YAML file [29]. The best-scored trajectory determines the `geometry_msgs/Twist` to be sent to the `/cmd_vel`. The Goal Checker defines the local termination condition, typically via position/orientation tolerances; upon its satisfaction, the `FollowPath` action concludes successfully.

In the overall architecture, `PathFollower` is a *thin orchestrator*: it takes the path A* (published to `/astar_path`), adapts it to the format expected from the Controller Server, and entrusts DWB with generating speed commands, publishing the outcome to `/controller_failed`. A repeating `True` on this topic is used by the `NavigationManager` to trigger the *fallback* when the controller fails to follow the path due to obstacles, dynamic constraints, or local inconsistencies. Under nominal conditions, the A* → PathFollower → DWB cycle guarantees a responsive path tracking, taking advantage of the local costmap (inflation included) and *critics* to maximize the clearance and keep the gear stable towards the goal.

5.6 Bug

The *BugNode* node implements a reactive obstacle avoidance behavior used as a **fallback** when A* does not locate a viable path or when the controller (DWB) fails to follow it. Unlike costmap-based modules, the Bug is not dependent on the local map: it operates directly on the 2D-LiDAR returns (`/scan`) and odometry (`/odom`), generating kinematic commands on `/cmd_vel`. Consequently, it subscribes directly to the goal provided by RViz (`/goal_pose`) rather than to the projected goal (`/projected_goal`). Turning the fallback *on* and *off* is done via the `/fallback_to_bug` (Bool messages) topic, while re-entry to the scheduled pipeline is notified to the manager by publishing `/bug0/abort` = `True`. All behavior is defined in the `odom` frame, and parameters are configurable from YAML files.

5.6.1 State Machine

Operation can be read as a *finite state machine* that alternates **goal tracking** and **wall-following**. Under normal conditions, the state *"go_to_goal"* orients the robot directly towards the received target at `/goal_pose`. The desired orientation is derived from the angle joining the current position to the goal position; the angular command applies a *proportional gain* (`kp_goal`) on the orientation error, saturated within `max_angular_speed`, while the linear velocity is set at a prudent value (`linear_speed_goal`). However, if LiDAR detects an obstacle in the direction of travel, an event that the code verifies with `is_obstacle_ahead()`, comparing the frontal measurement with `obstacle_front_threshold`, the node leaves direct tracking and moves on to the **side-choosing** stage.

The transition *"decide_side"* is intended to establish which flank to follow the contour of the obstacle. To do so, the node compares the distances measured by the LiDAR in two diagonal directions per side (e.g. $\pm 30°$ and $\pm 45°$); when the difference is modest (less than `follow_side_threshold`), the *left* side is conventionally preferred to avoid oscillation. Otherwise, the side is chosen as the one with the smaller diagonal distance (i.e., the closer obstacle). The decision is blocked for the next stretch (internal flag), so that frequent switching is avoided, and the system enters the state wall-following.

In *"follow_wall"* the robot maintains an almost constant distance from the obstacle. The code measures one *lateral distance* from the selected side and constructs an error against the required value `desired_wall_distance`; the control action is a proportional angular correction `kp_wall` $\times$ `error`, with a reinforcement factor (`wall_error_boost`) when the error exceeds a threshold (`wall_error_threshold`). The linear component depends on the *frontal risk level*: in

the presence of very close obstacles (below `obstacle_front_threshold` from the front, or below `obstacle_diag_threshold` on the diagonals), the node imposes `linear_speed_stop` and performs an emergency rotation (`emergency_angular_speed`) that moves the robot away from the wall. In a non-critical approach phase (`front` $<$ `slowdown_front_threshold`), it reduces the speed to `linear_speed_slow` while keeping the angular correction saturated; under nominal conditions, it proceeds at `linear_speed_wall`.

5.6.2 Exit Condition

Exiting the *"follow_wall"* state and returning to *"go_to_goal"* is allowed when at least one of two sufficient conditions occurs.

The first condition is the **visibility of the goal**. The `goal_visible()` function calculates the target angle in the robot frame, checks that it falls within the LiDAR's field of view, and compares the robot–goal distance with the LiDAR measure in the same direction, applying the `goal_visibility_margin`. If no LiDAR returns emerge prior to the goal, the direction is considered free and the node can abandon wall-following.

The second condition is **reentry onto the M-line**. The function `on_m_line()` evaluates the distance of the robot from the straight line joining `mline_start` to the current target and compares it with the threshold `mline_tolerance`. If this distance is below the threshold, and in the absence of frontal obstacles, direct pursuit of the target should be resumed.

These two exit rules, considered together, configure a *hybrid* policy that integrates the principles of **Bug0** (restart as soon as the direction towards the goal is free) and **Bug2** (abandonment of the contour upon re-entry onto the M-line), as discussed in section 3.2.

Upon fulfillment of any of the above-described conditions, the node publishes `/bug0/abort` = `True`, informing the NavigationManager that the fallback can be disabled and the scheduled pipeline (A* $\rightarrow$ DWB) can be restored. In addition, if the manager explicitly arranges to disable the fallback (`/fallback_to_bug` = `False`), the node performs a secure stop procedure, publishing a null Twist on `/cmd_vel` and handing control to the main modules.

Algorithm 2: BugNode

1 **Subscribers:**

`/goal_pose` → GOAL_CALLBACK (PoseStamped)
`/odom` → ODOM_CALLBACK (Odometry)
`/scan` → LASER_CALLBACK (LaserScan)
`/fallback_to_bug` → FALLBACK_CALLBACK (Bool)

Publishers:
`/cmd_vel` ← Twist
`/bug0/abort` ← Bool

Parameters (excerpt):
Wall following: `desired_wall_distance`, `kp_wall`, `wall_error_threshold`, `wall_error_boost`
Goal tracking: `kp_goal`, `goal_tolerance`, `goal_visibility_margin`
Speeds: `linear_speed_goal`, `linear_speed_wall`, `linear_speed_slow`, `linear_speed_stop`, `max_angular_speed`, `emergency_angular_speed`
LiDAR: `lidar_front_angle`, `lidar_diag_angles`, `lidar_total_fov`, `lidar_angle_range`
Bug logic: `mline_tolerance`, `mline_start`
Control: `control_frequency`

Initialization:
Create pubs/subs; set timer with period 1/`control_frequency`
State: `goal_pose`←nil, `position`←nil, `yaw`← 0
`laser_ranges`← [ ], `bug_active`←False, `goal_reached`←False
FSM: `state`←`go_to_goal`; `wall_follow_side`←nil; `wall_follow_side_locked`←False

Callback GOAL_CALLBACK(*msg: PoseStamped*):
`goal_pose` ← *msg.pose*; `goal_reached` ← False; `state` ← `go_to_goal`

Callback ODOM_CALLBACK(*msg: Odometry*):
`position` ← *msg.pose.pose.position*; `yaw` ← yaw from quaternion

Callback LASER_CALLBACK(*msg: LaserScan*):
`laser_ranges` ← *msg.ranges*

Callback FALLBACK_CALLBACK(*msg: Bool*):
`bug_active` ← *msg.data*

if not `bug_active` **then**
  PUBLISH_STOP()
end if

Timer thread LOOP():
if not `bug_active` **then**
  return
end if
if `goal_pose`=nil **or** `position`=nil **or** |`laser_ranges`| = 0 **then**
  return
end if
$d \leftarrow$ DISTANCE(`position`, `goal_pose.position`)
if $d <$ `goal_tolerance` **then**
  `goal_reached` ← True; PUBLISH_STOP(); **return**

```
22 end if
23 if state = go_to_goal then
24     if IS_OBSTACLE_AHEAD() then
25         hit_point ← position; state ← decide_side
26     else
27         twist ← MOVE_TO_GOAL()
28     end if
29 else if state = decide_side then
30     L ← |(GET_RANGE(lidar_diag_angles[0]) + GET_RANGE(lidar_diag_angles[1])) / 2|
31     R ← |(GET_RANGE(−lidar_diag_angles[0]) + GET_RANGE(−lidar_diag_angles[1])) / 2|
32     if |L − R| < follow_side_threshold then
33         wall_follow_side ← left
34     else
35         wall_follow_side ← (R < L) ? right : left
36     end if
37     wall_follow_side_locked ← True; state ← follow_wall
38 else if state = follow_wall then
39     if GOAL_VISIBLE() or (ON_M_LINE() and not IS_OBSTACLE_AHEAD()) then
40         state ← go_to_goal; wall_follow_side_locked ← False
41         publish Bool(True) on /bug0/abort
42     else
43         twist ← FOLLOW_WALL()
44     end if
45 end if
46 publish twist on /cmd\_vel
47 Function MOVE_TO_GOAL():
       ang ← atan2(y_g − y, x_g − x); e ← NORMALIZE_ANGLE(ang − yaw)
       twist.linear.x ← linear_speed_goal; twist.angular.z ← SATURATE(kp_goal · e, −max_angular_speed, max_angular_speed)
       return twist
48 Function FOLLOW_WALL():
       front ← GET_RANGE(lidar_front_angle); fl ← GET_RANGE(lidar_diag_angles[0]); fr ← GET_RANGE(−lidar_diag_angles[0])
       if side=right: side ← GET_RANGE(−lidar_diag_angles[1]); err ← desired_wall_distance − side
       else (side=left): side ← GET_RANGE(lidar_diag_angles[1]); err ← side − desired_wall_distance
       u ← kp_wall · err; if |err| > wall_error_threshold then u ← u · wall_error_boost
       if front < obstacle_front_threshold or fl < obstacle_diag_threshold
```

or $fr <$ `obstacle_diag_threshold`:
$twist.linear.x \leftarrow$ `linear_speed_stop`; $twist.angular.z \leftarrow \pm$ `emergency_angular_speed` (turn away)
else if $front <$ `slowdown_front_threshold`:
$twist.linear.x \leftarrow$ `linear_speed_slow`; $twist.angular.z \leftarrow$ SATURATE(u, $-$`max_ang.`, `max_ang.`)
else: $twist.linear.x \leftarrow$ `linear_speed_wall`; $twist.angular.z \leftarrow$ SATURATE(u, $-$`max_ang.`, `max_ang.`)
return $twist$

49 **Function** IS_OBSTACLE_AHEAD():
return GET_RANGE(`lidar_front_angle`) $<$ `obstacle_front_threshold`

50 **Function** GOAL_VISIBLE():
$\phi \leftarrow$ NORMALIZE_ANGLE($\text{atan2}(y_g - y,\ x_g - x) - yaw$); $deg \leftarrow \deg(\phi)$; $h \leftarrow$ `lidar_total_fov`$/2$
if $deg < -h$ **or** $deg > h$ **then return** False
$d_g \leftarrow$ DISTANCE(`position`, `goal_pose.position`); $d_L \leftarrow$ GET_RANGE(deg)
return $d_L \geq d_g -$ `goal_visibility_margin`

51 **Function** GET_RANGE($angle\_deg$):
if $|angle\_deg| >$ `lidar_angle_range` **or** $|$`laser_ranges`$| = 0$ **then return** ∞
$N \leftarrow |$`laser_ranges`$|$; $idx \leftarrow \text{clip}\left(\left\lfloor \frac{angle\_deg + \texttt{lidar\_angle\_range}}{\texttt{lidar\_total\_fov}} \cdot N \right\rfloor\right)$
return `laser_ranges`$[idx]$

52 **Function** ON_M_LINE():
if `hit_point`=nil **or** `goal_pose`=nil **then return** False
distance from `position` to the line `mline_start`→`goal_pose` $<$ `mline_tolerance` **return** True/False

53 **Function** DISTANCE(p_1, p_2): **return** $\sqrt{(p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2}$

54 **Function** NORMALIZE_ANGLE(θ): wrap $\theta \in (-\pi, \pi]$; **return** θ

55 **Function** SATURATE(v, a, b): **return** $\max(\min(v, b), a)$

56 **Procedure** PUBLISH_STOP(): publish $Twist()$ on `/cmd\_vel`

57 **Main:**
init rclpy; create node `BugNode`; spin; on interrupt: stop, destroy, shutdown

5.7 Goal Checker

The *GoalChecker* is an auxiliary node that provides the system with a binary, reliable and low latency signal on whether the local target has been reached. It operates in the `odom` frame and subscribes to the current robot pose from `/odom` and the projected goal from `/projected_goal`. It posts to `/goal_reached` a `std_msgs/Bool` message that is `True` when the robot is within a metric threshold of the goal, `False` otherwise.

The criterion is purely `geometric`: at the configurable sampling rate (`check_frequency`

Hz), the node computes the Euclidean distance

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

between the current position (`Odometry → pose.pose.position`) and the target (`PoseStamped → pose.position`). If $d <$ `goal_tolerance`, the goal is declared reached; otherwise it is not. To avoid chattering and redundant logs, the node maintains an internal flag (`already_reached`) that allows emitting the transition to `True` only once upon entering the tolerance region, and restoring `False` only if the robot subsequently moves back outside the threshold (with an explicit "reset" message in the log).

5.8 Manager

The *NavigationManager* supervises the local pipeline: planning, tracking, reactive fallback, and termination. It operates in the `odom` frame, is **event–driven**, and on each new target (`/goal_pose`) it resets its internal state (controller–failure counter, fallback flag, completion state) and restores the A* → DWB chain, so every goal starts from a clean condition.

The decision logic relies on three signals:

1. A* planning outcome at `/astar_failed`;
2. Controller outcome at `/controller_failed`;
3. Bug abort notification at `/bug0/abort`.

It enables the Bug (`/fallback_to_bug` = `True`) when planning fails (`True` on `/astar_failed`) or when the controller reports failures in sequence (`threshold = 3` on an internal counter), if the fallback is not already active. When the controller later reports success (`False` on `/controller_failed`), it resets the counter and, if active, disables the Bug (`/fallback_to_bug` = `False`). A Bug abort (`/bug0/abort` = `True`) likewise triggers a return to A* → DWB and resets the controller-failure counter.

The Goal Checker's `/goal_reached` has priority: at the first transition to `True` the manager deactivates nonessential behaviors (including the fallback), resets counters, and marks the goal as completed; subsequent failure notifications are ignored to prevent unwanted re-entries. This policy of hysteresis and idempotence yields clear state semantics: new goal ⇒ reset and main pipeline; repeated failures ⇒ fallback; Bug abort or controller success ⇒ return to the pipeline; goal reached ⇒ orderly shutdown.

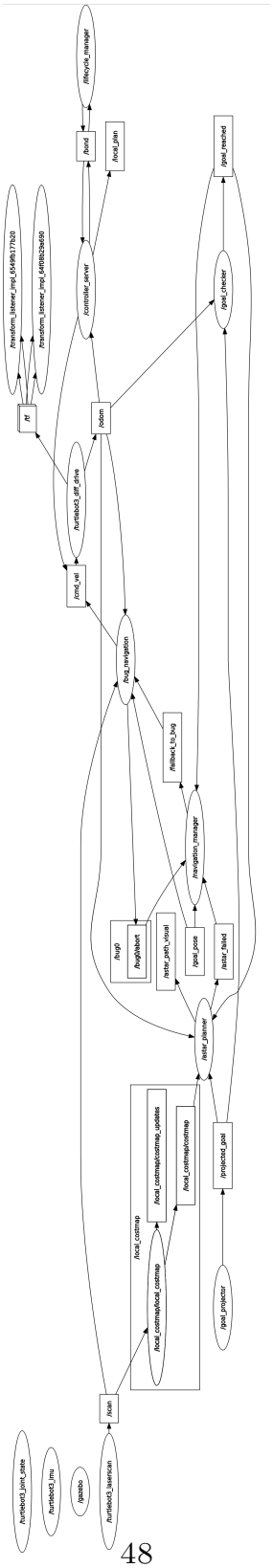

Figure 5.2: rqt_graph mobile robot

Chapter 6

PX4 Autopilot Integration

This chapter presents the integration of the **PX4 Autopilot** into the navigation pipeline, with the aim of validating obstacle avoidance on an aerial platform in a rigorous and reproducible way. In this scenario, it was deliberately chosen to focus the optimization on a single collision avoidance algorithm, A*, postponing the integration of the Bug to future work. That choice allowed a **state machine** to be designed that enables `Offboard` mode exclusively when needed, limiting intrusive interventions and preserving mission execution whenever conditions allow it.

Operational behavior is divided into two modes (Figure 6.1), selectable via the YAML `collision_prevention_in_mission` parameter:

- **COLLISION PREVENTION IN MISSION** (`True`): during the execution of the mission, upon detection of an obstacle the UAV enters the `HOLD`, without deviating from the planned trajectory.
- **COLLISION AVOIDANCE** (`False`): upon detection of an obstacle the system transits from `Mission` to `Offboard`, the drone is controlled via speed setpoints along a path generated by A* within the local costmap and, upon restoration of safe conditions, automatically re-enters `Mission` to continue the mission.

Compared to the terrestrial case, the substantial difference is that in the PX4 context an external local controller (like `FollowPath`) is not necessary to generate commands: the `Offboard` mode allows to directly send speed/position setpoints to the flight controller. As a result, the previously used DWB was removed, also in light of its performance already considered suboptimal in the terrestrial context (section 7.1).

The local costmap remains the one of Nav2 (25x25 m), but the parameters have been adapted to the drone scenario, so as to appropriately calibrate the cost profile and expected clearance in flight. In this configuration, the sensor used to create the

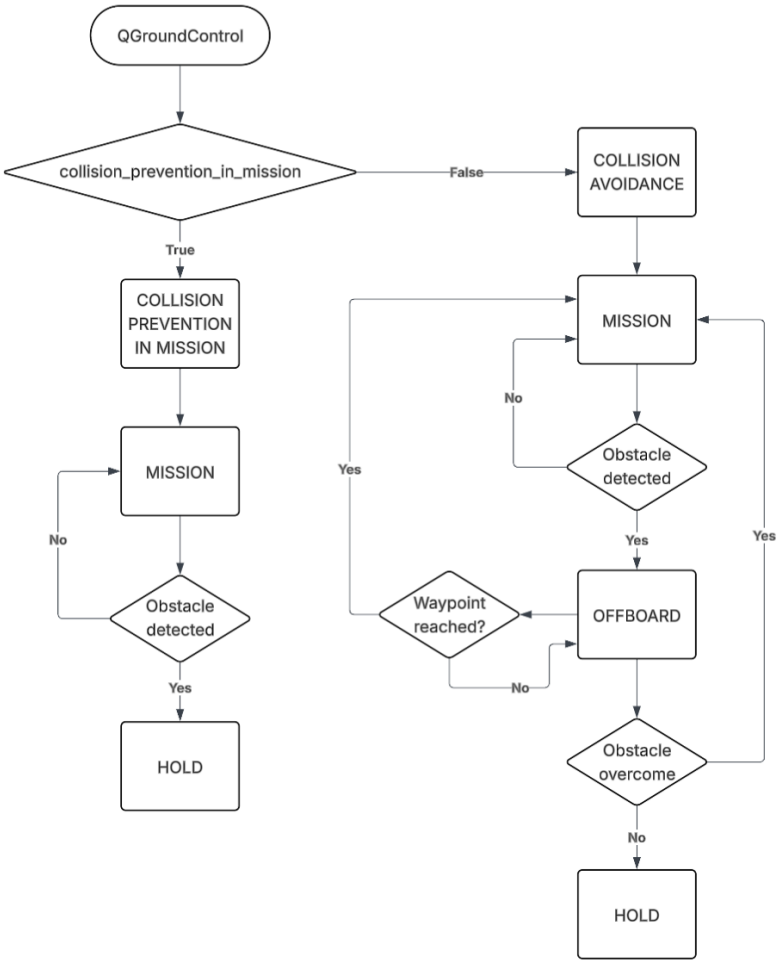

Figure 6.1: Drone modes flowchart

costmap is always a 2D LiDAR but with a field of view of 270°, chosen to ensure angular suitable coverage to the mission profile.

The nodes actually used are described below (Figure 6.2), highlighting the differences from the previous architecture and illustrating in detail the role of the state machine that governs entry and exit from the Offboard, as well as the transition between the two operating modes.

As before, the main pseudocodes are reported in the text while the others in Appendix B.

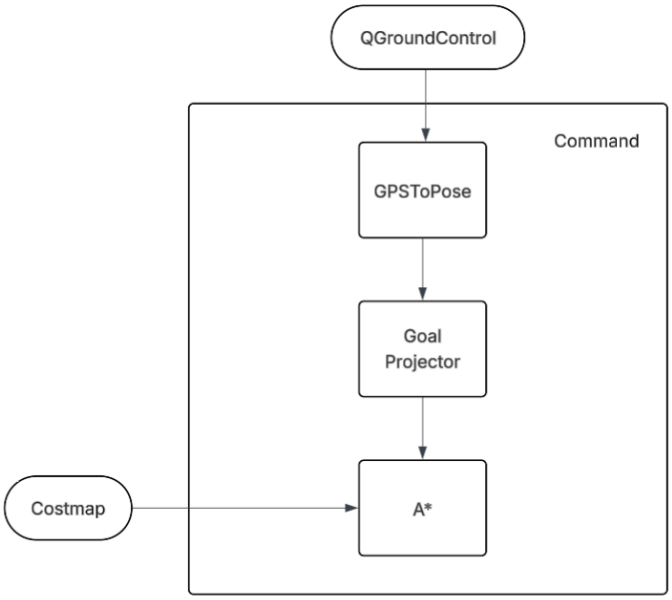

Figure 6.2: Simplified flowchart of drone nodes

6.1 GPSToPose

Compared to the pipeline employed on TurtleBot, this work introduces a new node that acts as a direct link between QGroundControl and PX4: mission waypoints are defined in QGroundControl and exposed by the autopilot on **PX4 topics**, while RViz remains solely a visualization tool. The node *GPSToPose* subscribes to `/fmu/out/home_position` (`px4_msgs/HomePosition` message) to acquire the *"home"* position and to `/fmu/out/position_setpoint_triplet` (`px4_msgs/PositionSetpointTriplet` message) to receive the current setpoint. Until the home is available, incoming setpoints are not processed; moreover, the node processes only POSITION-type setpoints, discarding the others. Subscriptions use a sensor-type *QoS profile* (`BEST_EFFORT`, `TRANSIENT_LOCAL`, `KEEP_LAST(10)`) on both topics, consistent with high-frequency flows and the need to receive the latest message already published.

The conversion from the **geodetic domain** (*latitude, longitude, altitude*) to the **local metric domain** is accomplished with **pyproj**. Starting from home longitude, the code recognizes the `UTM zone` (that is, the longitudinal band of 6° in which the vehicle is located) by calculating it as $32600 + int((lon_home + 180)//6) + 1$, and constructs the related planar reference system with $Proj(..., always\_xy = True)$.

In doing so, it automatically locates the "portion of the world" in an east-west direction and adopts the `EPSG` of the `326xx` family (northern hemisphere). Note that the current code assumes the northern hemisphere; a possible extension for the southern hemisphere would require the selection of the `327xx` family as a function of the latitude of the home. Both the home and the waypoint are projected in meters (x, y); local increments result

$$\Delta x = x_1 - x_0, \qquad \Delta y = y_1 - y_0, \qquad \Delta z = \mathrm{alt}_{\mathrm{wp}} - \mathrm{alt}_{\mathrm{home}}.$$

These quantities are published in the `odom` frame on two distinct topics, with consistent headers (timestamp and `frame_id`):

- `/goal_pose` (**ENU**) — `geometry_msgs/PoseStamped`: the node assigns $(x, y, z) = (\Delta x, \Delta y, \Delta z)$ and sets a neutral orientation ($w = 1.0$);
- `/goal_ned` (**NED**) — `geometry_msgs/PoseStamped`: the node applies planar and public permutation only $(x, y, z) = (\Delta y, \Delta x, \Delta z)$. An explicit orientation for this message is not set in the code.

This choice makes the same spatial target immediately usable for components both operating under the ENU convention and waiting for a NED representation, without introducing further transformations along the planning and control chain.

6.2 Goal Projector

Compared to the original version of the *GoalProjector*, the new implementation introduces an optimized behavior, while maintaining the same functional objective: to ensure that the planner always works with an achievable target within the local costmap window. Both nodes receive the global goal at `/goal_pose` and the costmap at `/local_costmap/costmap`, publishing at `/projected_goal` a projected target consistent with the local map limits; both respect the `odom` frame and construct the message as `PoseStamped` with planar position and neutral orientation. However, the differences emerge clearly in the choices of pose acquisition, in the sampling policy, in the cell acceptance criteria and in the management of critic cases.

First of all, the new node no longer subscribes to `/odom`, because in drone the integration with PX4 exposes a set of topics different from the ROS 2 standards and odometry is not available on the classic topic. That's why the current pose is derived via **TF2** by making the lookup of the `base_link` → `odom` transformation with timeout (2 s). This choice makes the node agnostic with respect to the state estimate source and still aligns all modules on the `odom` frame. For completeness, the transform itself is broadcast by a lightweight bridge that subscribes to `/fmu/`

`out/vehicle_odometry` (PX4), converts position/orientation from NED/FRD to ENU/FLU, stamps the transform with ROS time, and publishes `odom→base_link` on TF2. If the TF lookup fails, the current cycle is simply skipped, avoiding stale projections.

Even subscribing to the costmap adopts a more suitable QoS profile: `RELIABLE`, `TRANSIENT_LOCAL`, `KEEP_LAST(1)`, so it always receives the last "latching" useful and does not miss critical updates; the previous version was limited to a default numeric queue.

The projection procedure retains the basic idea (walking the straight line between pose and goal and, in case of block, attempting moderate angular deviations), but the new version refines the details. Sampling along the main direction is denser: the pitch is equal to a quarter of the map resolution ($res \cdot 0.25$ factor) and a minimum of 30 samples is imposed, against the half-cell pitch and the absence of a stringent minimum in the previous version. Lateral deviations are controlled by parameter (`side\_angles\_deg`); this facilitates the adaptation of the node to costmaps with different inflation or to more "cluttered" spaces. A further difference concerns the cell acceptance criterion: now a cell is considered practicable if $val \leq 5$ or $val = -1$ (unknown), whereas previously only 0 and -1 were accepted. Enlargement to small positive values (typical of light inflation) reduces false projection negatives near obstacles, while preserving a safety margin.

In terms of managing degenerate cases, the new version introduces an explicit *projection timeout mechanism*. If, despite sampling along the straight line and along the lateral directions, no valid point is found, the node starts an internal timer (`no_valid_since`) and, having exceeded the parametric threshold (`projection_timeout_sec`), publishes a `Bool(True)` flag to `/projection_timeout`, signaling downstream that the waypoint is temporarily unreachable. From that moment the search is suspended (`timeout_sent=True`) until a new goal or `/goal_reached` event, which resets the status and restores normal behavior, is received. The previous version did not provide this "local" failure signal nor did it implement a controlled suspension: in case of persistent impossibility of projection, it continued to attempt without an explicit diagnosis channel towards the controller.

6.3 A*

The *AStarPlanner* node performs local grid scheduling within the local costmap window (`odom` frame). The operational architecture follows the previous version: the node subscribes to `/local_costmap/costmap`, receives the local goal at `/projected_goal`, listens to `/goal_reached`, schedules with A* on 8-connected graph, and publishes the result both to the controller (`/astar_path` as `PoseArray`) and to RViz (`/astar_path_visual` as `Path`). As anticipated in the description

of section 6.2, the current pose is obtained via TF2 (timeout 0.5 s); the costmap `frame_id` is propagated in the output messages to ensure reference consistency.

The main innovation, compared to the "terrestrial" version, is twofold: **cost model** and **post–processing** of the route. Instead of a binary (free/unknown) walkability, here the walk cost incorporates the "softness" of the occupancy: cells with a value ≥100 are excluded (obstacle), while for intermediate values a nonlinear penalty is added to the weight of the arc

$$\text{penalty(cell)} \;=\; \exp\Big(\tfrac{\text{cost}}{40}\Big) - 1,$$

added to the Euclidean distance between adjacent cells. In this way the planner favors wide corridors and avoids trajectories close to obstacles.

The discrete path produced by A* is then refined in two steps. At first, a simplification by distance (parametric threshold `path_min_dist`) eliminates redundant points. Later, a planar smoothing using **cubic B-spline** (splprep/splev). Given the N poses $\{(x_i, y_i)\}_{i=1}^{N}$, we construct a parametric curve $\mathbf{S}(u) = \big(x(u),\, y(u)\big)$, with $u \in [0,1]$, in the standard form:

$$x(u) = \sum_j N_{j,3}(u)\, C_{x,j}, \qquad y(u) = \sum_j N_{j,3}(u)\, C_{y,j},$$

where $N_{j,3}(u)$ are cubic B-spline basis functions and $\mathbf{C}_j = (C_{x,j},\, C_{y,j})$ are the estimated control points.

The smoothing parameter `spline_smoothing = s` governs the interpolation/approximation trade-off: for $s = 0$ the curve passes through all samples, for $s > 0$ it approximates their cloud with controlled error. The smooth curve is then sampled uniformly in u with `spline_points` cardinality. In this code these parameters are set to 2 and 100 respectively. The orientation along the path is obtained from the tangent (in the code: a forward difference between adjacent samples),

$$\psi_i \;\approx\; \text{atan2}\Big(y(u_{i+1}) - y(u_i),\; x(u_{i+1}) - x(u_i)\Big),$$

and converted into a quaternion as

$$(z,\, w) \;=\; \Big(\sin(\psi_i/2),\, \cos(\psi_i/2)\Big).$$

The output to the controller is adapted to the aeronautical context: in addition to the paths for RViz (raw and smooth, over `/astar_path_visual` and `/astar_path_smooth`), the node publishes a `PoseArray` over `/astar_path` in a relative and NED-compatible form. The positions are expressed with respect to the first point $(\Delta x, \Delta y)$ and converted with planar permutation $(x_{NED}, y_{NED}) = (\Delta y, \Delta x)$; the angle is consistently recalculated ($\psi_{NED} = \pi/2 - \psi_{ENU}$). A minimal filter (0.3 m) avoids excessively dense samples, making tracking more stable.

The scheduling cycle is punctuated by a parametric frequency timer (`publish_frequency = 20Hz`). With each activation the node checks availability of costmaps, poses and goals, invokes A*, goes up the `came_from` predecessors to reconstruct the path and converts the cells into world coordinates via `map_to_world`. The `/astar_failed` flag explicitly signals the absence of solutions; the goal callback implements a *debounce*: if a projection arrives that is practically identical to the one just reached (within 0.1 m), it is ignored to avoid oscillations.

Algorithm 3: AStarPlanner Node

Subscribers:
`/local_costmap/costmap` → COSTMAP_CALLBACK (OccupancyGrid)
`/projected_goal` → PROJECTED_GOAL_CALLBACK (PoseStamped)
`/goal_reached` → GOAL_REACHED_CALLBACK (Bool)
Publishers:
`/astar_path` ← PoseArray
`/astar_path_visual` ← Path
`/astar_path_smooth` ← Path
`/astar_failed` ← Bool
Parameters:
`topic_costmap`, `topic_projected_goal`, `topic_goal_reached`
`topic_astar_path`, `topic_astar_visual`, `topic_astar_smooth`, `topic_astar_failed`
`publish_frequency` (Hz), `path_min_dist` (m), `spline_smoothing`, `spline_points`
Initialization:
Create pubs/subs (costmap QoS: RELIABLE, TRANSIENT_LOCAL, KEEP_LAST(1))
State: `costmap_ready`←False; `current_pose`←None; `global_goal`←None
`latest_path_poses`← [ ]; `goal_reached`←False
Start timer with period 1/`publish_frequency` → TIMER_CALLBACK
Init TF buffer + listener
Callback COSTMAP_CALLBACK(OccupancyGrid msg):
Store `map_data,width,height,resolution,origin,frame_id`; `costmap_ready`←True
Function GET_CURRENT_POSE(`target="odom"`, `source="base_link"`)→PoseStamped|None:
try TF lookup (timeout 0.5s) ⇒ build PoseStamped; else return None
Callback PROJECTED_GOAL_CALLBACK(PoseStamped msg):
if `goal_reached` **and** `global_goal`≠None **and** _POSES_EQUAL(msg.pose,`global_goal`.pose, 0.1) **then**
    return ▷ ignore identical projection after goal reached
end if

11 `global_goal`←msg; `goal_reached`←False
12 **Callback** GOAL_REACHED_CALLBACK(Bool msg):
13 **if** msg.data **then**
14 `goal_reached`←True; `global_goal`←None
15 **end if**
16 **Timer** TIMER_CALLBACK():
17 **if** `goal_reached` **or not** `costmap_ready` **then**
18 **return**
19 **end if**
20 `current_pose`←GET_CURRENT_POSE()
21 **if** `current_pose`=None **or** `global_goal`=None **then**
22 **return**
23 **end if**
24 $path\_raw \leftarrow$ COMPUTE_PATH(`current_pose`, `global_goal`)
25 **if** $path\_raw$ is None **or** $|path\_raw.poses| = 0$ **then**
26 publish Bool(True) on `/astar_failed`;
27 **return**
28 **else**
29 publish Bool(False) on `/astar_failed`
30 $raw \leftarrow [p.pose\ \forall p \in path\_raw.poses]$
31 $filt \leftarrow$ SIMPLIFY_PATH(raw, `path_min_dist`)
32 `latest_path_poses`← SMOOTH_PATH_WITH_BSPLINE($filt$, `spline_smoothing`, `spline_poi`
33 PUBLISH_PATH_AS_POSE_ARRAY() on `/astar_path`
34 Build RViz Paths and publish on `/astar_path_visual`, `/astar_path_smooth`
35 **end if**
36 **Function** _POSES_EQUAL(p_1, p_2, tol): **return** $\sqrt{(p_1.x - p_2.x)^2 + (p_2.y - p_1.y)^2} < tol$
37 **World/Map:**
WORLD_TO_MAP(x, y) : $mx = \lfloor (x - origin\_x)/res \rfloor$, $my = \lfloor (y - origin\_y)/res \rfloor$
MAP_TO_WORLD(mx, my) : $x = origin\_x + (mx + 0.5)res$, $y = origin\_y + (my + 0.5)res$
38 **Function** GET_COST(mx, my):
if in bounds return `map_data`[my·width+mx] else return 100
39 **Heuristic** $H(a, b) = \sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}$
40 **Function** COMPUTE_PATH(start:PoseStamped, goal:PoseStamped)→Path|None:
if start=None or goal=None return None
$s \leftarrow$ WORLD_TO_MAP($start$); $g \leftarrow$ WORLD_TO_MAP($goal$)
$open \leftarrow \{s\}$; $came\_from \leftarrow \emptyset$; $g\_s[s] \leftarrow 0$; $f\_s[s] \leftarrow H(s, g)$
41 **while** $open \neq \emptyset$ **do**
42 $c \leftarrow \arg\min\_o \in open f\_s[o]$
43 **if** $c = g$ **then**
44 **return** BUILD_PATH_MESSAGE(RECONSTRUCT(came_from,s,g))
45 **end if**

remove c from *open*; $N \leftarrow$ 8-neighbors of c
for each $n \in N$ **do**
$cost \leftarrow$ GET_COST(n);
if $cost \geq 100$ **then**
continue
end if
$pen \leftarrow e^{(cost/40)} - 1$; $\tilde{g} \leftarrow g\_s[c] + H(c, n) + pen$
if $\tilde{g} < g\_s(n)$ (default ∞) **then**
$came\_from[n] \leftarrow c$; $g\_s[n] \leftarrow \tilde{g}$; $f\_s[n] \leftarrow \tilde{g} + H(n, g)$; add n to *open*
end if
end for
end while
return None
Function BUILD_PATH_MESSAGE(cells):
Create Path (`frame_id`); for (mx, my): $(x, y) \leftarrow$ MAP_TO_WORLD; $z \leftarrow 0$
yaw from forward difference; set quaternion; append PoseStamped; update `latest_path_poses`; return Path
Function ENU_TO_NED_RELATIVE(dx, dy) $\rightarrow$ (x_ned, y_ned) with $x\_ned \leftarrow dy$, $y\_ned \leftarrow dx$
Procedure PUBLISH_PATH_AS_POSE_ARRAY():
Build PoseArray (stamp now, `frame_id`); for each pose in `latest_path_poses`:
compute (dx, dy) wrt first; ENU$\rightarrow$NED; convert yaw: $yaw\_ned = \pi/2 - yaw\_enu$; down-sample by 0.3 m; publish
Procedure SIMPLIFY_PATH($poses, min\_dist$):
keep-first; append pose only if distance from last $\geq$ `min_dist`; ensure last included
Procedure SMOOTH_PATH_WITH_BSPLINE($poses, smoothing, num\_points$):
if $|poses| < 4$ return poses; else compute B-spline (splprep, s =smoothing), sample num_points
rebuild poses with yaw from forward diffs; on exception return original

6.4 Obstacle Detector

The *ObstacleDetector* node performs a behavioral gating function for security management: it provides the decision signal which determines the entry into `OFFBOARD` mode when COLLISION AVOIDANCE is active, or the entry into `HOLD` mode when COLLISION PREVENTION IN MISSION is active. To avoid false positives that trigger unnecessary transitions, the controlled portion of space is only **the one in front of the drone**, aligned with the trajectory towards the global goal; furthermore, if the waypoint falls within the costmap, the verification corridor is truncated exactly to the waypoint, so that, in the absence of obstacles along that stretch, the UAV can continue in `MISSION` mode.

Operationally, the node implements light and responsive detection along the drone → goal direction. It works on the local costmap (`nav_msgs/OccupancyGrid`) and, at a parametric rate (`check_interval`), publishes a Boolean on `/obstacle_detected` that indicates the presence of occupied cells within a rectangular corridor centered on the trajectory. For diagnostic purposes, it also issues a marker at `/obstacle_area_marker` that displays the area actually tested in RViz. On the I/O plane, it subscribes to `/local_costmap/costmap` and `/goal_pose` uses TF2 to get the current pose in the costmap frame, and adopts QoS `RELIABLE`, `TRANSIENT_LOCAL`, `KEEP_LAST(1)` to have the last state latched. The main parameters are a corridor half-width (`half_width = 1.5`), control interval (`check_interval = 0.2`), and sampling step (`step_size = 0.1`).

The logic is geometric and operates entirely in the costmap frame. At each cycle the node: (i) computes the vector from the drone to the goal,

$$(\Delta x_{\text{goal}},\, \Delta y_{\text{goal}}),$$

its length

$$L = \sqrt{(\Delta x_{\text{goal}})^2 + (\Delta y_{\text{goal}})^2},$$

and the unit direction

$$\mathbf{d} = (d_x, d_y) = \left(\tfrac{\Delta x_{\text{goal}}}{L},\, \tfrac{\Delta y_{\text{goal}}}{L}\right);$$

(ii) constructs the perpendicular

$$\mathbf{p} = (-d_y,\, d_x);$$

(iii) defines a forward-aligned rectangle with half-width and effective length $L_{\max}$, initially set to L (drone→goal distance) and reduced only if, stepping by `step_size` along $\mathbf{d}$, the path exits the costmap bounds. Consequently, when the goal lies inside the window,

$$L_{\max} = L,$$

and the corridor ends at the target.

Inspection is cell-based: for each cell we retrieve the world coordinates of its center

$$(w_x,\, w_y),$$

form the displacement

$$\mathbf{r} = (w_x - x_{\text{drone}},\, w_y - y_{\text{drone}}),$$

and project its longitudinal and lateral components:

$$\text{forward} = \mathbf{r} \cdot \mathbf{d}, \qquad \text{lateral} = \mathbf{r} \cdot \mathbf{p}.$$

A cell triggers detection if `data[idx]` > 0 (obstacle or inflation) and it falls within the corridor,

$$0 \leq \text{forward} \leq L_{\max}, \qquad |\text{lateral}| \leq \texttt{half\_width}.$$

Free (0) or unknown (-1) cells do not trigger the flag, consistent with the use of inflation as a conservative barrier. Upon first hitting the node publishes `Bool(True)` to `/obstacle_detected` and stops scanning; in parallel, a red `LINE_STRIP Marker` (lifetime 1 s) draws the vertices of the controlled rectangle, from the edge near the drone to the front edge (at the goal or limit of the costmap).

6.5 Command

The *Command* node is the executive level of the collision avoidance pipeline: it interfaces with PX4 to decide the operating mode (`MISSION`, `OFFBOARD`, `HOLD`) and, when necessary, generates speed commands that guide the drone along the local path. The integration takes place via the PX4 topics: at the input law `/fmu/out/vehicle_status_v1` (navigation state), `/fmu/out/vehicle_local_position` (poses and speed), and the avoidance chain signals (`/astar_path, /goal_ned`, `/local_costmap/costmap`, `/obstacle_detected`, `/astar_failed`, `/projection_timeout`)¦ at the public output `/fmu/in/offboard_control_mode`, `/fmu/in/trajectory_setpoint` (speed), and `/fmu/in/vehicle_command` (change modes). The control loop spins at 20 Hz and enables only when `nav_state==3` (`MISSION`), which poses `mission_allowed=True`.

On the data plane, the node maintains currents (x, y, z) and (v_x, v_y), stores a local home on first reception and keeps the last received path at `/astar_path` as an ordered sequence of waypoints (x_i, y_i, z_i, ψ_i). The components (x_i, y_i) come from the `PoseArray`, ψ_i is obtained from the quaternion, while z_i is aligned to the Offboard dimension saved during handover (or, in the absence, to the current dimension). From the local costmap the node extracts the weight $\omega \in [0 - 100]$ of the central cell (**drone position**) to modulate the speed according to the proximity to obstacles; the values "unknown" (< 0) are mapped to `occupancy_unknown_as` (0 by default).

From path to velocity command

The PoseArray on `/astar_path` is already in the `OFFBOARD` compatible navigation frame, so it doesn't require transformations - each element is a waypoint (x_i, y_i, z_i, ψ_i) arranged along the track. At each control cycle the node:

- **selects** the current target `current_goal_index` and advances to the next

index when the planar distance

$$d_i \;=\; \sqrt{(x - x_i)^2 + (y - y_i)^2} \;<\; 0.2 \text{ m};$$

if the points are exhausted, it enters `HOLD`.

- **assumes** the waypoint yaw ψ_i as the **direction of motion** (local tangent of the path); it does not compute a heading error, but directly uses ψ_i as the reference.

- **scales** the speed magnitude by reading the central weight w. With $v_{\max} = 2.0$ m/s and `max_cell_weight`$= 60.0$, the scale

$$s \;=\; \max\Big(0,\; \min\Big(1,\; \frac{\texttt{max\_cell\_weight} - w}{\texttt{max\_cell\_weight}}\Big)\Big)$$

linearly reduces the speed down to a stop for $w \geq 60$.

- **projects** the command from the body frame to the navigation frame: it builds $\mathbf{v}^{(b)} = [\, v_{\max}\, s,\; 0\,]$ and rotates by ψ_i, yielding

$$v_x = (v_{\max} s)\cos\psi_i, \qquad v_y = (v_{\max} s)\sin\psi_i, \qquad v_z = 0.$$

- finally, it publishes a `TrajectorySetpoint` with unconstrained x, y positions (`NaN`), altitude fixed to `offboard_altitude`, `yaw` $= \psi_i$, and velocity $(v_x, v_y, 0)$.

Thanks to the parameters `inflation_radius = 2.0` and `cost_scaling_factor = 1.0`, the combination with `max_cell_weight = 60.0` results in the vehicle coming to a stop approximately 0.7 m from the obstacle. If s=0, in addition to sending zero speed the node forces `HOLD` for safety.

Transitions and security guards

In `OFFBOARD`, a `/projection_timeout` (inability to project a local goal for 0.5s) induces immediate `HOLD`; a persistent `/astar_failed` beyond `retry_duration=0.5s` causes `HOLD`; if `/obstacle_detected` returns `False` the avoidance is no longer necessary and re-enters `MISSION`. The position of the closest waypoint is saved in `/goal_ned`, the node verifies $\sqrt{(x - x_g)^2 + (y - y_g)^2} <$ `goal_reached_treshold`(0.5) and, upon satisfaction, release `OFFBOARD` by returning to `MISSION` so that PX4 marks the waypoint as achieved and can advance to the next one.

6.5.1 Finite State Machine

- **VERTICAL**
  Stabilization phase which can correspond to **take-off**, but also to the phase immediately following an entry into `HOLD` in which the drone stopped. The node enters in it when `mission_allowed = True` and the horizontal speed v_{xy} is less than `vertical_speed_threshold` (0.3 m/s). In particular, if the `HOLD` had been caused by an obstacle and the latter disappears, the UAV, once it has returned to `VERTICAL` (because it is stopped), is able to resume the mission: upon subsequent exceeding of the speed threshold, take-off/stabilization is considered completed and it moves on to `EVALUATE_AFTER_TAKEOFF`. No avoidance commands are yet issued in `VERTICAL`.

- **EVALUATE_AFTER_TAKEOFF**
  It waits for the first value of `/obstacle_detected` to assess the post-takeoff scene. If there are no obstacles, `MISSION` is reaffirmed; if there are obstacles, the choice depends on `collision_prevention_in_mission`: if active, it goes to `HOLD` (COLLISION PREVENTION IN MISSION), otherwise it prepares `OFFBOARD` for local avoidance (COLLISION AVOIDANCE).

- **MISSION**
  Nominal mode in which PX4 follows the mission. If such an obstacle is detected, `collision_prevention_in_mission = True` asks for `HOLD` at PX4; otherwise, it passes to `PREPARE_OFFBOARD` to take control and circumvent the obstacle.

- **PREPARE_OFFBOARD**
  Secure handover to external control: continuous publishing of `OffboardControlMode`, saving the current altitude to `offboard_altitude`, and sending, for 10 cycles, zero-speed setpoints (priming required by PX4). Once priming is complete, `VEHICLE_CMD_DO_SET_MODE` (`param2 = 6.0`) is sent to engage `OFFBOARD`.

- **OFFBOARD**
  The node drives the drone along `/astar_path` by converting waypoints into tangential speed commands and modulated by the costmap weight: the speed decreases as the weight increases until it stops at ≈ 0.7 m from the obstacle. Critical events (`/projection_timeout`, `/astar_failed` beyond `retry_duration`) induce `HOLD`; removing the obstacle (`/obstacle_detected==False`) results in a return to `MISSION`. Since `/goal_ned` is active, if the planar distance drops below `goal_reach_threshold`, the node releases `OFFBOARD` returning to `MISSION`, so that PX4 marks the waypoint as reached and continues. If the waypoints run out, it enter in `HOLD`.

- **HOLD**
  Safety stop: the node controls `Hold` mode at PX4 sending `VEHICLE_CMD_DO_SET_MODE` (`param2 = 4.0`, `param3 = 3.0`). The drone maintains its position until favorable conditions return (valid route, obstacles removed, new goal), avoiding oscillations between modes and preserving margin from obstacles.

Algorithm 4: Command Node

1 **Subscribers:**
`/fmu/out/vehicle_status_v1` → VEHICLE_STATUS_CB (VehicleStatus)
`/fmu/out/vehicle_local_position` → LOCAL_POS_CB (VehicleLocalPosition)
`/astar_path` → ASTAR_PATH_CB (PoseArray)
`/obstacle_detected` → OBSTACLE_CB (Bool)
`/goal_reached` → GOAL_REACHED_CB (Bool)
`/astar_failed` → ASTAR_FAILED_CB (Bool)
`/goal_ned` → GOAL_NED_CB (PoseStamped)
`/local_costmap/costmap` → COSTMAP_CB (OccupancyGrid)
`/projection_timeout` → PROJECTION_TIMEOUT_CB (Bool)

2 **Publishers:**
`/fmu/in/vehicle_command` ← VehicleCommand
`/fmu/in/offboard_control_mode` ← OffboardControlMode
`/fmu/in/trajectory_setpoint` ← TrajectorySetpoint

3 **Parameters:**
topics: `topic_vehicle_command, topic_offboard_mode, topic_setpoint, topic_vehicle_status`
`topic_local_position, topic_astar_path, topic_obstacle_detected, topic_goal_reached`
`topic_astar_failed, topic_goal_ned, topic_costmap, topic_projection_timeout`
scalars: `goal_reach_threshold` (m), `retry_duration` (s), `v_max` (m/s), `max_cell_weight`, `collision_prevention_in_mission` (bool), `vertical_speed_threshold` (m/s)

4 **State:**
PX4: `nav_state`, `mission_allowed`←False
Pose: (x, y, z, vx, vy); `local_pos_received`←False; home set once
Mode FSM: `state`∈ {VERTICAL, EVALUATE_AFTER_TAKEOFF, MISSION, PREPARE_OFFBOARD, OFFBOARD, HOLD}
Offboard: `mode_engaged`←False; `setpoint_counter`← 0; `offboard_altitude` (saved once)
Obstacles: `obstacle_detected`←False; `obstacle_evaluated`←False
A*: `goal_poses`$[(x, y, z, yaw)]$; `current_goal_index`← 0; `astar_failed`,

`astar_fail_time`
Costmap: width/height/res/ori/data; `costmap_ready`←False; `occupancy_unknown_as`← 0; `last_weight`← 0
Other: `goal_ned` (tuple or None), `projection_timeout`←False
Timer: $dt = 0.05\,\mathrm{s} \rightarrow$ CONTROL_LOOP()
Callback COSTMAP_CB(OccupancyGrid msg):
store $w, h, res, origin\_x, origin\_y, data$; `costmap_ready`←True
Function GET_CENTER_CELL_WEIGHT():
if costmap not ready ⇒ None; else read center (cx, cy), map to idx; unknown→`occupancy_unknown_as`
Callback VEHICLE_STATUS_CB(VehicleStatus msg):
`nav_state`←msg.nav_state; if msg.nav_state= 3 (MISSION) and not `mission_allowed` ⇒ set True
Callback LOCAL_POS_CB(VehicleLocalPosition msg):
update (x, y, z, vx, vy); `local_pos_received`←True; set home once
Callback GOAL_NED_CB(PoseStamped msg):
$(x, y, z) \leftarrow$ msg; $yaw \leftarrow 0$; `goal_ned`← (x, y, z); `goal_poses`← $[(x, y, z, yaw)]$; idx= 0
Callback PROJECTION_TIMEOUT_CB(Bool msg):
`projection_timeout`←msg.data
Callback ASTAR_FAILED_CB(Bool msg):
if `state`=`OFFBOARD` **then**
  if msg.data **and not** `astar_failed` **then**
    set `astar_failed`←True; `astar_fail_time`←now
  else
    reset flags
  end if
end if
Callback ASTAR_PATH_CB(PoseArray msg):
`goal_poses`←[]; for each pose: extract (x, y), z ←offboard_altitude or current z; yaw from quaternion; append
reset `current_goal_index`← 0, `astar_failed`←False, `astar_fail_time`←None
Callback OBSTACLE_CB(Bool msg): `obstacle_detected`←msg.data; `obstacle_evaluated`←True
Callback GOAL_REACHED_CB(Bool msg): `goal_reached`←msg.data
Timer CONTROL_LOOP():
if not `local_pos_received` **then**
  warn "Waiting for local position..."; **return**
end if
if not `mission_allowed` **then**
  return
end if
$v_{xy} \leftarrow \sqrt{vx^2 + vy^2}$

30 **if** `state`$\neq$ `VERTICAL`**and** v_{xy} < `vertical_speed_threshold` **then**
31 info "Enter VERTICAL"; `state`←VERTICAL; **return**
32 **end if**
33 **VERTICAL:** if `state`=VERTICAL and v_{xy} > `vertical_speed_threshold` ⇒ `state`←EVALUATE_AFTER_TAKEOFF; **return**
34 **EVALUATE_AFTER_TAKEOFF:**
35 **if not** `obstacle_evaluated` **then**
36 info "Waiting /obstacle_detected..."; **return**
37 **end if**
38 **if** `obstacle_detected` **then**
39 **if** `collision_prevention_in_mission` **then**
40 warn "CPIM ⇒ HOLD"; SET_HOLD_MODE(); `state`←HOLD
41 **else**
42 info "Prepare OFFBOARD"; `setpoint_counter`← 0; `offboard_wait_counter`← 0; `mode_engaged`←False; `state`←PREPARE_OFFBOARD
43 **end if**
44 **else**
45 info "No obstacle ⇒ MISSION"; SET_MISSION_MODE(); `state`←MISSION
46 **end if**
47 **return**
48 **MISSION:**
49 **if** `state`=MISSION **and** `obstacle_detected` **then**
50 **if** `collision_prevention_in_mission` **then**
51 warn "CPIM ⇒ HOLD"; SET_HOLD_MODE(); `state`←HOLD
52 **else**
53 info "Obstacle ⇒ PREPARE_OFFBOARD"; reset counters; `mode_engaged`←False; `state`←PREPARE_OFFBOARD
54 **end if**
55 **return**
56 **end if**
57 **PREPARE_OFFBOARD:**
58 PUBLISH_OFFBOARD_MODE();
59 **if not** `offboard_altitude_saved` **then** save `offboard_altitude`← z; flag True
60 **if** `setpoint_counter`<10 **then**
61 SEND_VELOCITY(0,0,0); ++counter; **return**
62 **end if**
63 **if not** `mode_engaged` **then**
64 SET_OFFBOARD_MODE(); `mode_engaged`←True; **return**
65 **end if**
66 `state`←OFFBOARD
67 **OFFBOARD:**
68 PUBLISH_OFFBOARD_MODE()
69 **if** `projection_timeout` **then**

```
70        error “Waypoint unreachable ⇒ HOLD”; SET_HOLD_MODE(); state←HOLD; reset flags; return
71    end if
72    if astar_failed then
73        t ← (now − astar_fail_time) (s)
74        if t > retry_duration then
75            warn “A* keeps failing ⇒ HOLD”; SET_HOLD_MODE(); state←HOLD; return
76        else
77            info “Retry waiting...”; return
78        end if
79    end if
80    if not obstacle_detected then
81        info “Obstacle cleared ⇒ MISSION”; SET_MISSION_MODE(); state←MISSION; reset flags; return
82    end if
83    if goal_ned≠ None then        computed=√((x − x_g)² + (y − y_g)²);
84
85    if d < goal_reach_threshold then
86        info “Goal NED reached”; SET_MISSION_MODE(); state←MISSION; reset alt flag; return
87    end if
88 end if
89 if goal_poses=∅or idx≥|goal_poses| then
90    warn “No path or completed”; state←HOLD; return
91 end if
92 (gx, gy, −, yaw) ← goal_poses[idx];  d ← √((gx − x)² + (gy − y)²)
93 if d < 0.2 then
94    ++idx; return
95 end if
96 w ← GET_CENTER_CELL_WEIGHT() or last_weight; last_weight← w
97 scale ← clip((max_cell_weight − w)/max_cell_weight, 0, 1)
98 v_b ← v_max · scale;  (v_x, v_y) ← (v_b cos yaw,  v_b sin yaw)
99 SEND_VELOCITY(v_x, v_y, 0, yaw);
100 if scale = 0 then
101    warn “scale=0 ⇒ HOLD”; SET_HOLD_MODE(); state←HOLD; reset alt flag; return
102 end if
103 HOLD:
104 if state=HOLD then
105    SET_HOLD_MODE(); info “Drone in HOLD”
106 end if
107 Publish helpers:
      PUBLISH_OFFBOARD_MODE(): set OffboardControlMode flags (position, velocity true); timestamp=NOW
```

SEND_VELOCITY(v_x, v_y, v_z, yaw): TrajectorySetpoint with velocity; position= (NaN, NaN, alt)
SET_MISSION_MODE(): VehicleCommand DO_SET_MODE (Mission)
SET_OFFBOARD_MODE(): VehicleCommand DO_SET_MODE (Offboard)
SET_HOLD_MODE(): VehicleCommand DO_SET_MODE (Hold)
PUBLISH_VEHICLE_COMMAND(cmd, params): fill VehicleCommand; publish

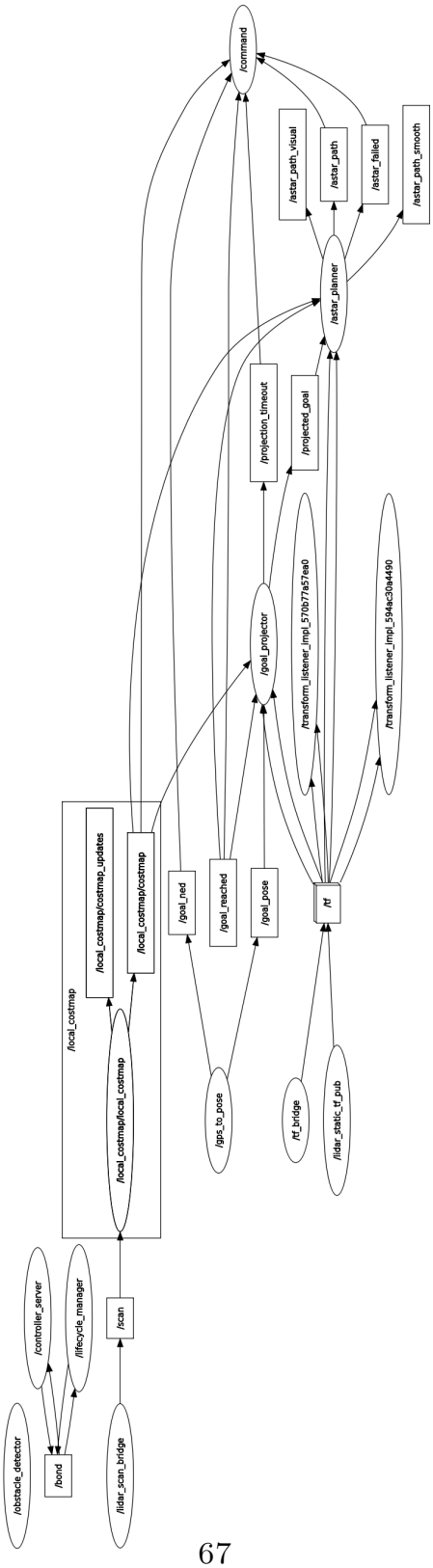

Figure 6.3: rqt_graph drone

Chapter 7

Simulations Results

7.1 Turtlebot

Simulations were initiated by means of a **launch file** that initializes in a coordinated way the entire pipeline (Gazebo, RViz, Goal Projector, A*, Path Follower, Bug, Goal Checker and Navigation Manager) ensuring time synchronization and reproducibility of the experiments. In addition to the created nodes, **Nav2's Controller Server** (`nav2_controller`) and **Lifecycle Manager** (`nav2_lifecycle_manager`) are also started, essential for the correct functioning of the plugins.

In this first set of tests, the methodological objective was to demonstrate the **neutrality** of the navigation logic with respect to the sensor used. Although the TurtleBot3 Burger's Gazebo model natively integrates 2D LiDAR with 360° FOV, the field of view was deliberately limited to 90° in tests. Such a configuration allows the pipeline to be evaluated also under representative conditions of narrow FOV sensors (for example, a camera), without changing the architecture of the nodes.

World 1

The first scenario consists of three single obstacles arranged in a plane such that at least one free cell of the grid remains between each pair, Figure 7.1. This spacing guarantees the connectivity of the search graph and, given the inflation parameters adopted, maintains an effectively viable corridor in the local costmap (6x6 m).

The Figure 7.2 reports three significant moments of the simulation. Coordination of the modules is observed in each frame. The red arrow identifies the goal projected by the Goal Projector: it is progressively translated forward within the local window as the robot advances, in order to offer A* an always reachable target. The green line represents the path calculated by A* on the OccupancyGrid published by the costmap; this map updates in real time as a function of sensory measurements and the path takes advantage of the narrow passages left free between obstacles.

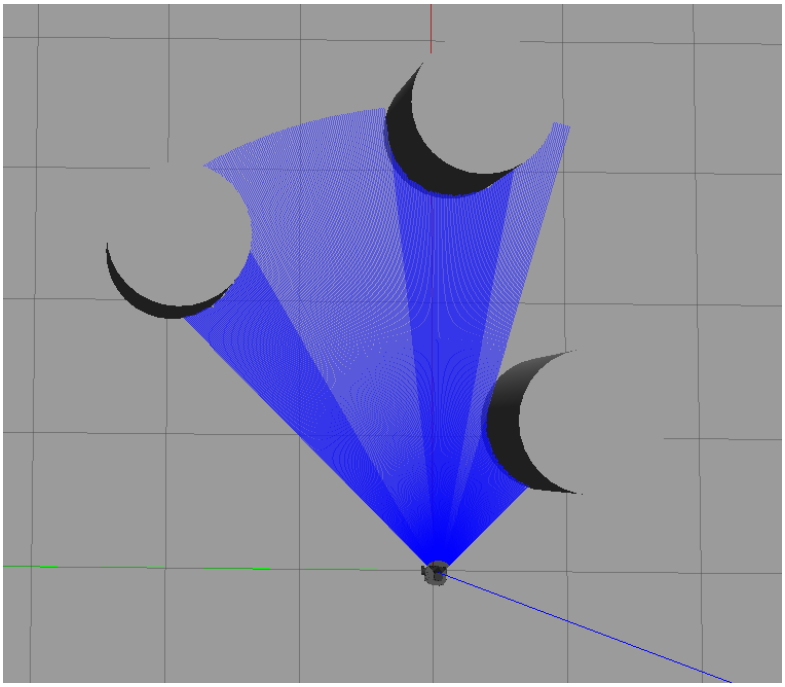

Figure 7.1: Turtlebot World 1 in Gazebo

Finally, the little blue segment, not clearly visible in these images, represents the trajectory actually followed by the DWB, which selects the speed commands within the kinematic constraints of the robot, favoring, among the eligible alternatives, those with greater clearance.

Although linear behavior is expected in this scenario — A* provides a valid path and the local controller generates commands without resorting to fallbacks — in some executions the DWB, integrated as a plug-in, accumulates more than three consecutive failures and activates Bug recovery. As an external component, the causes are not explored here; however, it is noted that, in the subsequent implementation with the drone, this component has been removed. However, the use of the Bug does not alter the outcome of the simulation: the robot still reaches the objective while avoiding obstacles. In a variant of the same environment, in which the obstacles are so close together that they leave no practicable corridor for A*, this fallback instead proves decisive in achieving the target.

World 2

The second scenario features a long wall that exceeds the sensor's field of view, Figure 7.3. As a result, the local costmap is "cut" by the obstacle: even with the goal projected correctly inside the window, A* does not have a walkable corridor

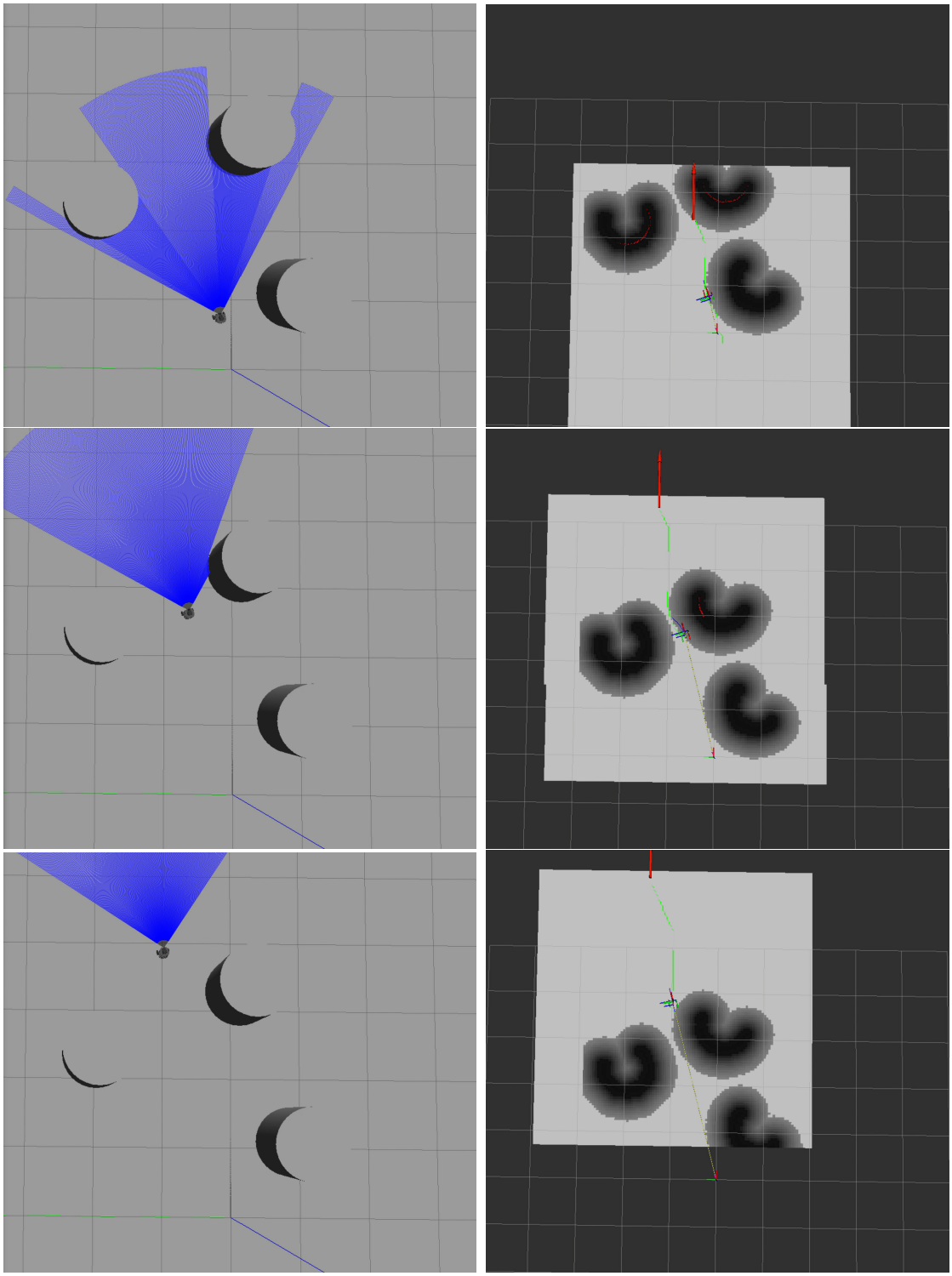

Figure 7.2: Turtlebot simulation in World 1 - Gazebo view on the left, RViz view on the right

and cannot find a valid path. In this condition, the Bug intervenes, which takes control and, as soon as the frontal obstacle is detected, selects a side on which to bypass it while maintaining a regulated distance, until the costmap highlights a useful step again and A* returns to produce a valid path.

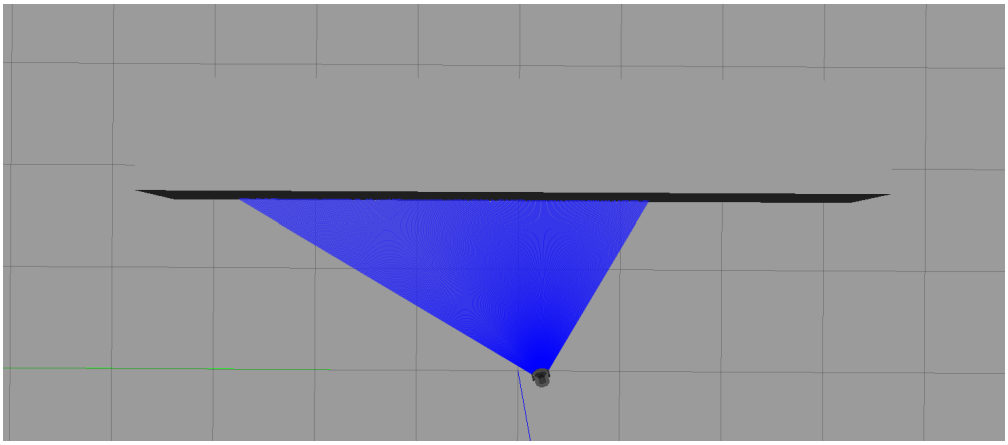

Figure 7.3: Turtlebot World 2 in Gazebo

The Figure 7.4 shows three moments of the simulation. In the first frame, the impossibility of planning is observed: the wall interrupts local connectivity and the Navigation Manager activates fallback. In the second frame, a green line appears (path A*) which, however, is not usable: it is the residual display of a path calculated before the robot perceived the side part of the wall; as soon as the costmap is updated with the obstacle, that path is no longer valid and the system continues with the Bug. In the third frame, after following the contour for a stretch, the robot recovers a gap in the costmap: A* returns to planning a coherent path (updated green line) and the DWB resumes the chase (blue segment), closing the fallback cycle and re-establishing the nominal pipeline.

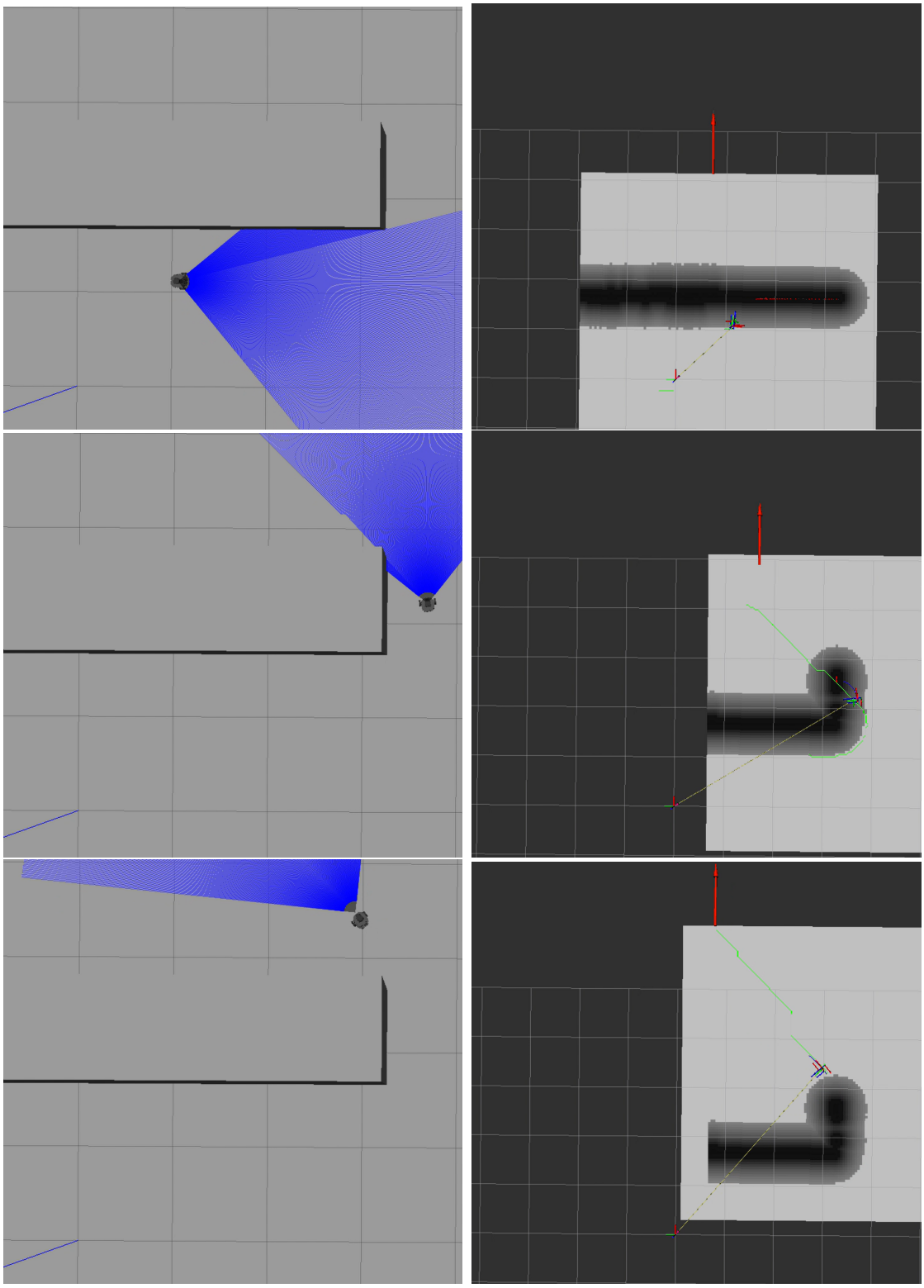

Figure 7.4: Turtlebot simulation in World 2 - Gazebo view on the left, RViz view on the right

7.2 Drone

Also for the execution of simulations with the drone, a boot infrastructure based on ROS 2 launch files was set up, responsible for the coordinated commissioning of only the ROS components: the developed nodes, Gazebo and RViz. The vehicle integrated with the PX4 Autopilot and QGroundControl are instead started separately inside the respective dedicated containers, so as to keep the separation between the ROS 2 stack and the autopilot/ground station clear.

The aircraft used is the `gz_x500_lidar_2d`, a multirotor equipped with 2D LiDAR with a field of view of 270°.

All tests were conducted assuming a mission speed of 5 m/s; consequently, the set of parameters was calibrated to such dynamic regime. Operating at higher speeds is likely to require parameter re-optimization to preserve safety margins and behavioral quality (tracking stability, compliance with minimum obstacle clearances, robustness to sensing delays).

In the figures reported from now on, the rectangle highlighted in red represents the area of space monitored for obstacle detection: if an obstacle falls outside this corridor, the UAV continues the mission without changing its conduct. The black path is the discrete path generated by A*, while the red one is the trajectory smoothed using B-spline, which constitutes the reference in collision avoidance mode. However, it is crucial to observe that the drone does not faithfully follow the red line point-by-point: the controller mainly employs local path orientation (direction of travel) to calculate speed commands, without imposing a tight constraint of exact passage on the samples. This choice explains why, in some snapshots, the UAV can be in correspondence with cells with a value > 0 in the costmap: the margin distance is however guaranteed by the cost model and the speed modulation as a function of the weight of the central cell.

During the simulation, the user can assume **manual control** via joystick at any time and, subsequently, return to `MISSION` mode without compromising the internal state of the modules or the continuity of the collision avoidance algorithm: the system manages the handover transparently, resuming the mission profile when automatic control is restored.

World 1

The first scenario, shown in Figure 7.5, introduces an apparently simple case but already suitable for highlighting a critical aspect. The environment contains two $1 \times 1 \times 10m$ parallelepiped pillars arranged on the forward direction of the UAV: the first about $4m$ from the take-off point, the second about $40m$. Criticality results from the proximity of the first obstacle from the beginning of the mission:

the drone is immediately in a condition where the frontal free corridor is reduced, resulting in early activation of the detection chain and avoidance.

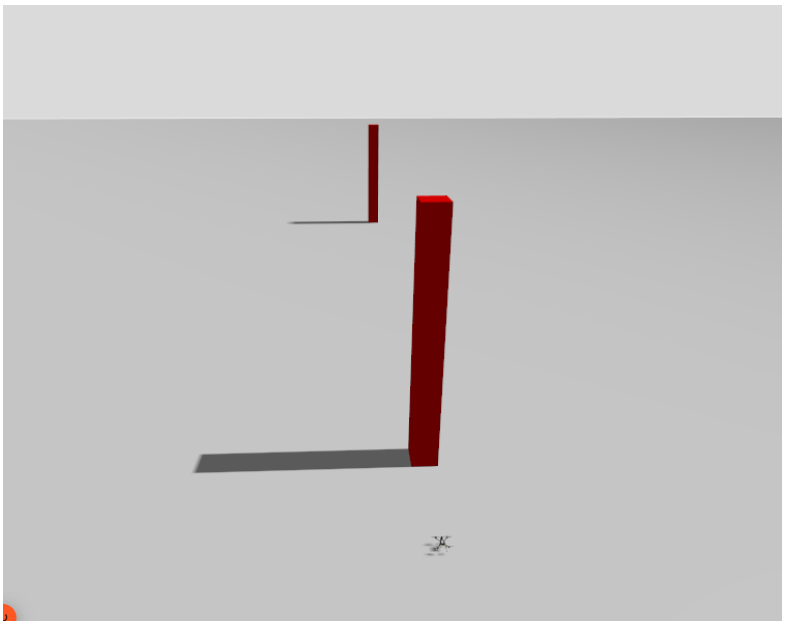

Figure 7.5: Drone World 1 in Gazebo

- **COLLISION PREVENTION IN MISSION**

  In case `collision_prevention_in_mission = True`, the drone, after `TAKEOFF`, enters in `MISSION` to enable the algorithm and immediately after passes into `HOLD`.

  Although the `HOLD` command is given instantly, the actual stop requires a finite time, proportional to the instantaneous speed: at 5 m/s, for example, the jerk-limited controller needs a few meters to dispose of the momentum. In these conditions, peculiar behavior is observed: the UAV continues for a distance "forward", passes the hold position captured by the autopilot (loiter setpoint), and then "reenters" returning slightly backwards until it stabilizes, Figure 7.6.

  This phenomenon is not deterministic and depends on a combination of factors: (i) the speed at the time of the switch (higher speed → larger stopping distance); (ii) the direction of motion relative to the hold point (if the switch occurs while the drone is still "going", overshoot is more likely); (iii) asynchronies between ROS 2 cycles, PX4 and simulator, which determine the precise instant at which the loiter point is "frozen"; (iv) the state of the controller (any integrative residues or saturations take time to dissipate).

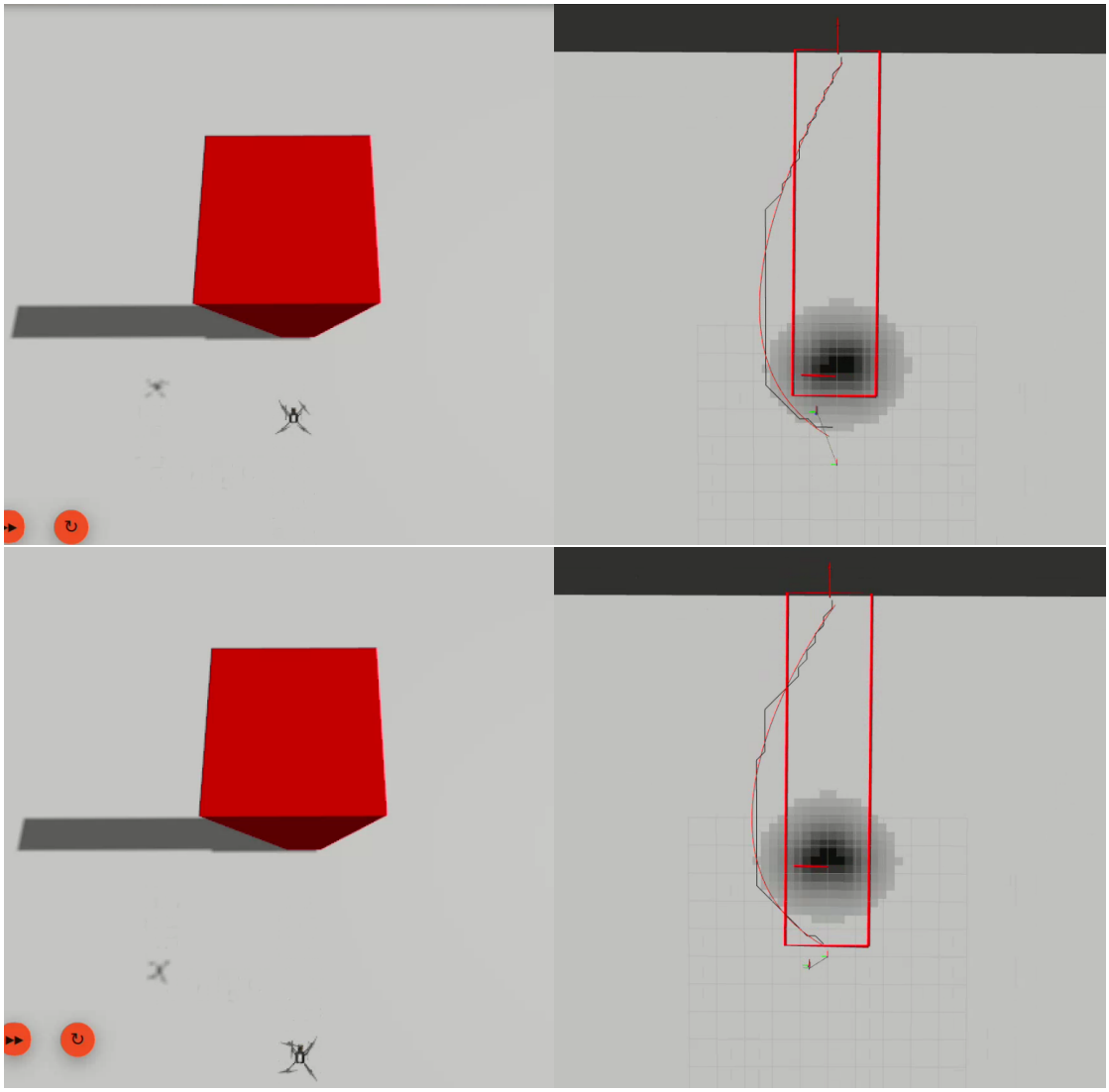

Figure 7.6: Drone simulation in World 1 - Gazebo view on the left, RViz view on the right - Collision Prevention in Mission

In reference to Figure 7.7, speeds are analyzed in relation to changes in vehicle status. The first image (up) shows the trend of the speed components; the second (down) shows the PX4 states read from `nav_state` (`/fmu/out/vehicle_status_v1`). The numbers shown correspond to:

- `NAVIGATION_STATE_POSCTL = 2`
- `NAVIGATION_STATE_AUTO_MISSION = 3`
- `NAVIGATION_STATE_AUTO_LOITER = 4`
- `NAVIGATION_STATE_OFFBOARD = 14`

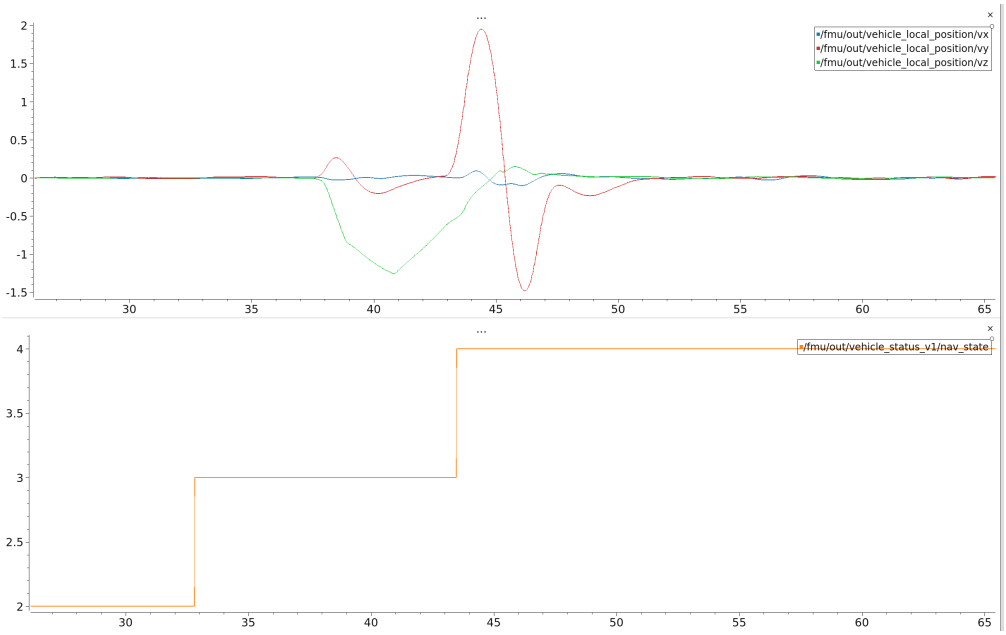

Figure 7.7: Velocity and State transition graphs of drone simulation in World 1 - Collision Prevention in Mission

- **COLLISION AVOIDANCE**

  On the other hand, when `collision_prevention_in_mission = False`, the drone does not enter `HOLD` after `TAKEOFF` but activates collision avoidance mode: it bypasses the obstacle by following the planned local path, Figure 7.8, and modulating the speed as a function of the risk, as shown in Figure 7.9. When free conditions are restored, the vehicle continues towards the target in `MISSION` mode without further interruptions.

  During the `OFFBOARD` phase the altitude is kept constant and equal to the altitude at which the aircraft was flying immediately before entering this mode

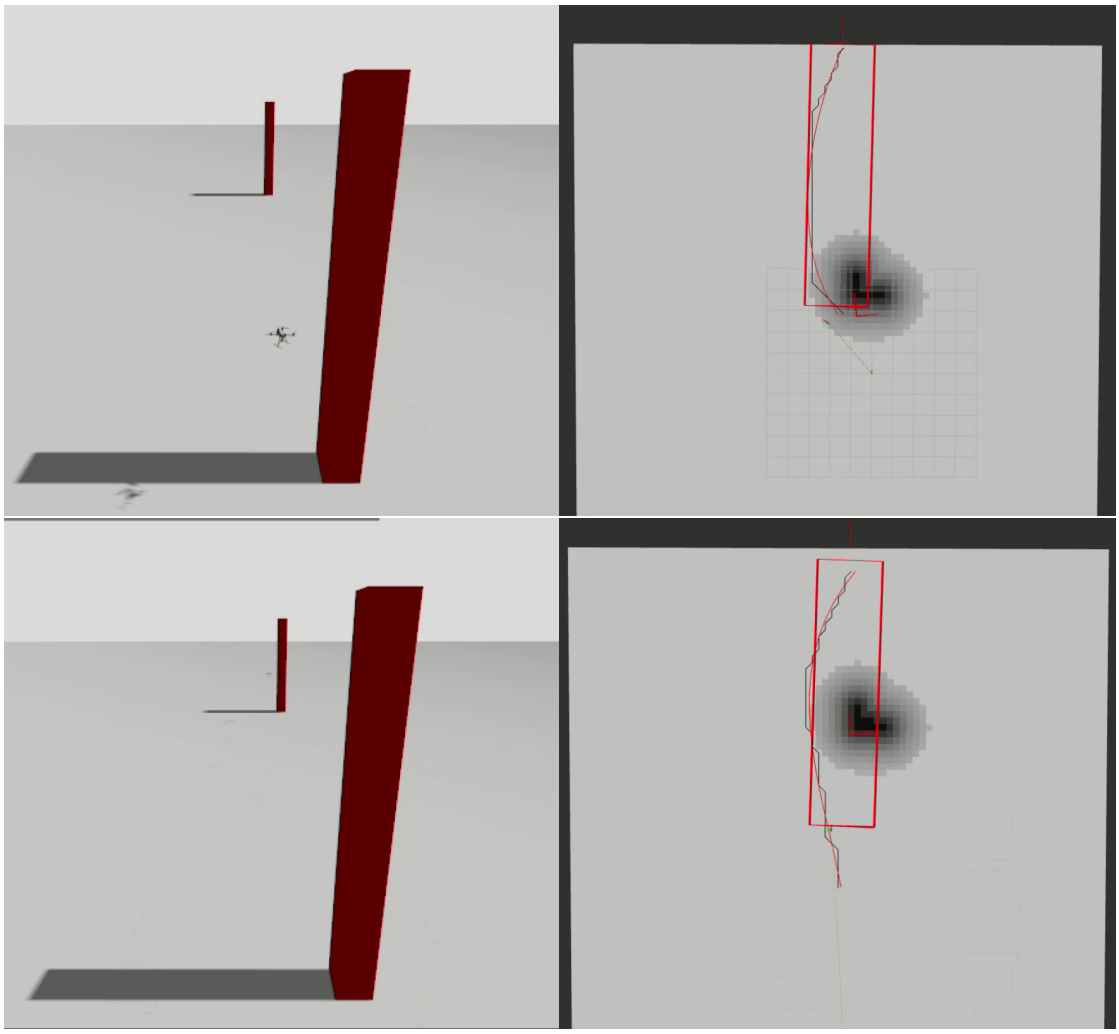

Figure 7.8: Drone simulation in World 1 - Gazebo view on the left, RViz view on the right - Collision Avoidance

(altitude stored at the time of handover). Since all the waypoints are set to the same altitude, this behavior is clearly evidenced in Figure 7.9, where the z-component of the velocity remains effectively zero—not only in `OFFBOARD`, but also throughout `MISSION`—confirming the maintenance of a constant flight altitude. This parameter, together with speed, distance travelled and mission time, can be easily monitored from the interface of QGroundControl, Figure 7.10. In the views, the red arrow indicates the instantaneous orientation of the drone along the trajectory; the track behind it represents the path actually taken.

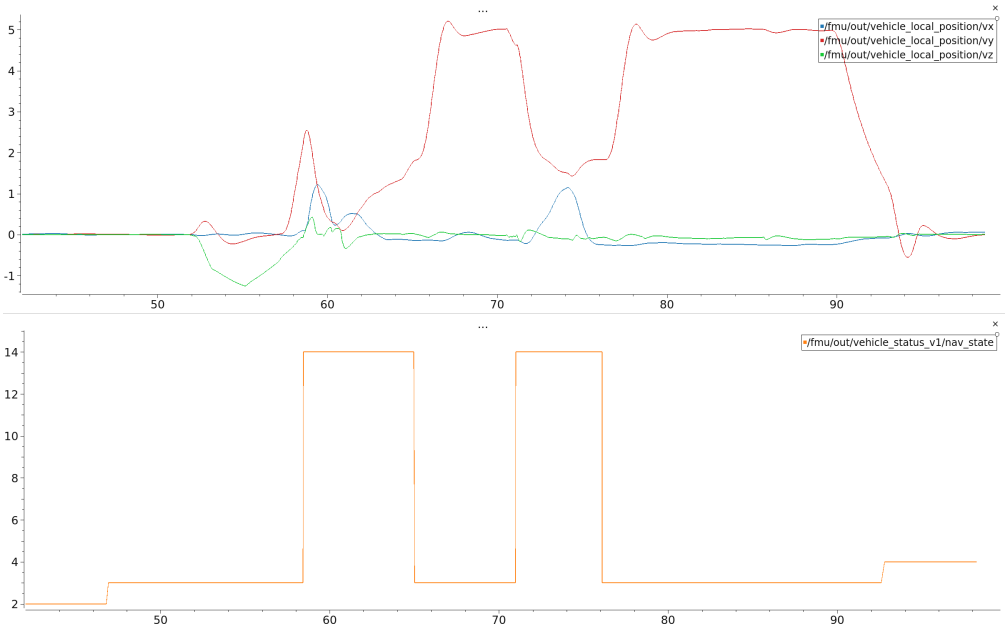

Figure 7.9: Velocity and State transition graphs of drone simulation in World 1 - Collision Avoidance

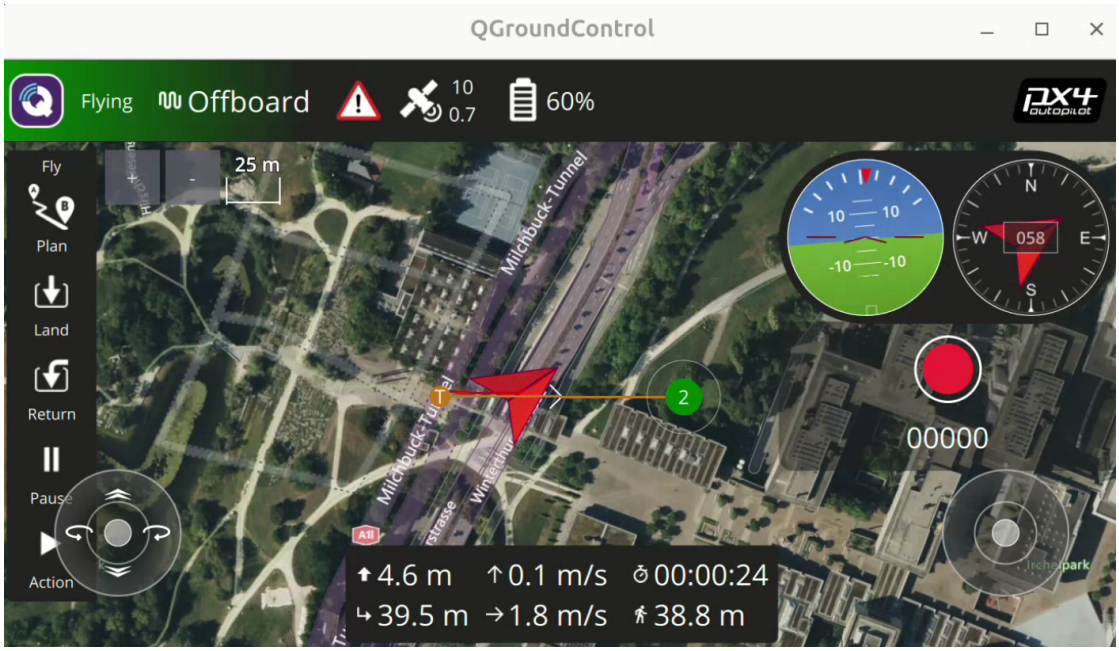

Figure 7.10: QGroundControl interface of drone simulation in World 1 - Collision Avoidance

World 2

The second scenario consists of a single volumetric obstacle: a cube $20 \times 20 \times 20m$ positioned approximately $20m$ from the drone's take-off point, along the forward direction, Figure 7.11.

- **COLLISION PREVENTION IN MISSION**

  In this case the UAV, upon first detection, passes through `HOLD` and stops in an orderly manner. Unlike the previous case (World 1), the phenomenon of "rebound" is not observed: after engaging the `HOLD` the drone does not continue beyond the activation point and then retreats, but reduces the speed until it stops without appreciable overshoot, Figure 7.12. The status and speed traces confirm a stable behavior without gear reversals, Figure 7.13.

- **COLLISION AVOIDANCE**

  World 2 is a critical case for collision avoidance mode. The obstacle occupies much of the costmap window, forcing a very abrupt change of direction on the UAV, Figure 7.14. Under these conditions the drone tends to get too close to the surface of the obstacle, remaining in a region of the map characterized by high costs, consequently, the A* is unable to identify an admissible route. The

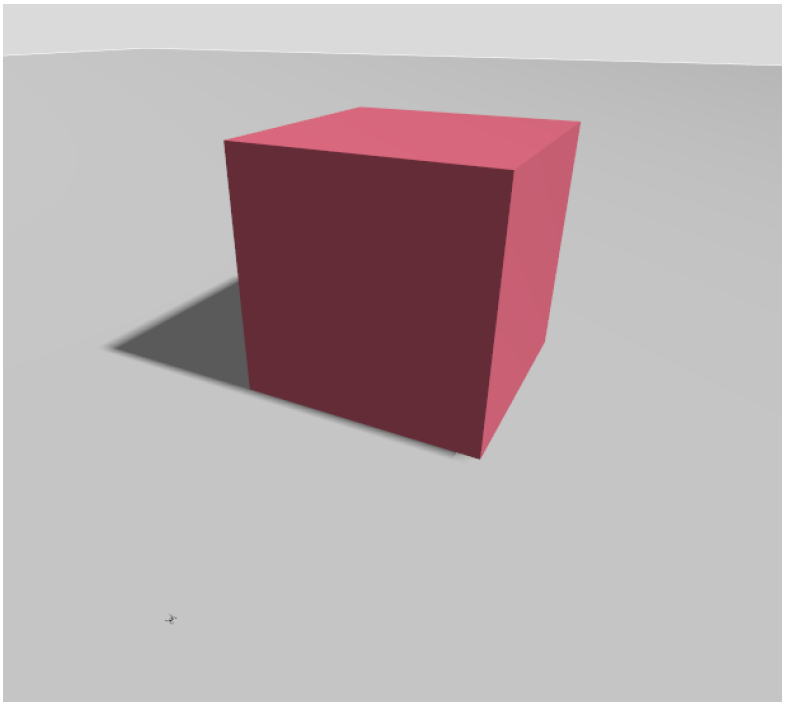

Figure 7.11: Drone World 2 in Gazebo

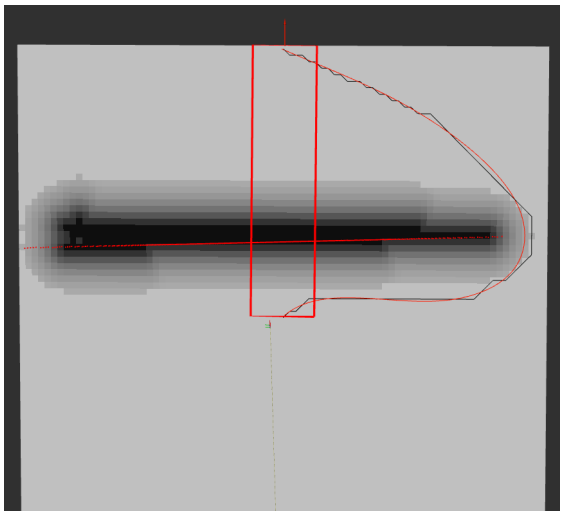

Figure 7.12: Drone simulation in World 2 - RViz view - Collision Prevention in Mission

stay in this zone exceeds the set time threshold (0.5 s), a fact that triggers the safety logic of the controller and determines the entry into `HOLD`. In the

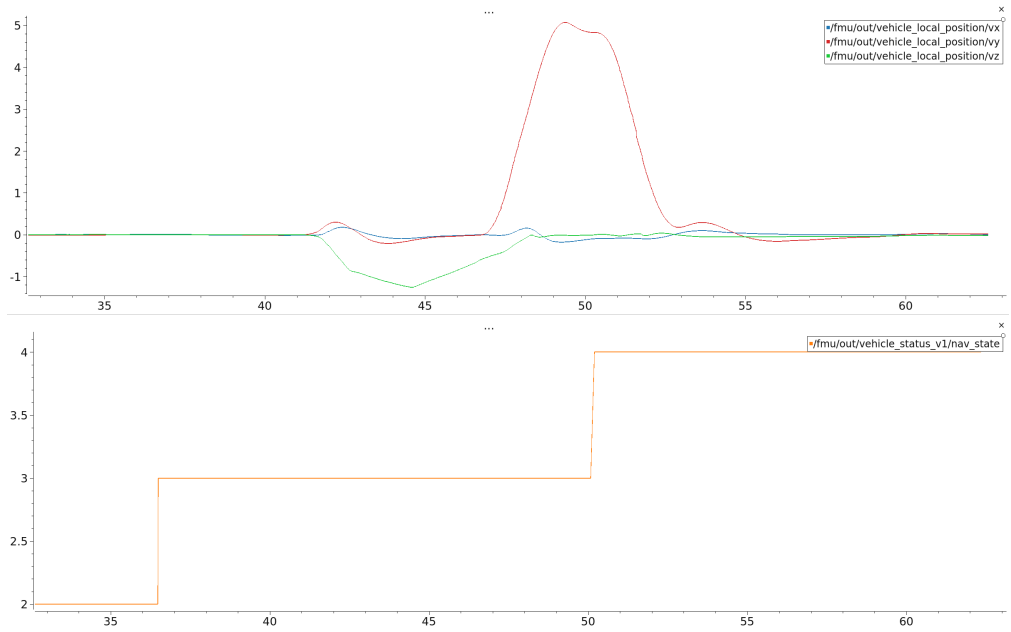

Figure 7.13: Velocity and State transition graphs of drone simulation in World 2 - Collision Prevention in Mission

third image, a slight residual displacement is observed: it is the effect of the delay between the `HOLD` request and the actual stop, during which the drone continues to follow the orientation of the last available path. Finally, note that the scheduling pipeline remains active: a path may still be visible as the A* continues to look for solutions; however, operator intervention is required to exit the `HOLD`.

To overcome the critical issues found in this case, a further simulation is performed by varying some parameters. Firstly, the `OFFBOARD` maximum speed is reduced from $2m/s$ to $1m/s$: the drone adheres better to the path and, thanks to the lower speed, handles the particularly angled initial section more robustly, approaching less the obstacle and continuing without entering `HOLD`. Having reached the corner of the obstacle, the UAV "sees" the side wall extending to the bottom of the costmap, so the drone–goal direction is not yet practicable.

At this point the issue shifts to the goal projection: with lateral angles of $\pm 15°$ and $\pm 30°$ no valid position is found within the local window, resulting in a return to `HOLD`. Widening the search angles to $\pm 45°$ solves the problem: as shown in Figure 7.16, the goal projection advances in small increments along the edge of the obstacle until its end; throughout the whole simulation

the projection remains locally attainable and the A* progressively plans the following sections, allowing the mission to continue without entering `HOLD`.

Figure 7.15 and Figure 7.17 show the speed and status variations related to the two system configuration, while Figure 7.18 shows the trajectory followed by the drone to avoid the obstacle, displayed in QGroundControl, in the second scenario.

World 3

The third scenario, Figure 7.19, represents a highly cluttered environment: 100 $2 \times 2 \times 20m$ pillars are arranged randomly starting from $10m$ away from the drone. This setup produces a texture of narrow passages and winding corridors, testing both goal projection and local planning on the costmap.

In COLLISION PREVENTION IN MISSION the behavior replicates that observed in World 2 while the COLLISION AVOIDANCE mode is analyzed below.

Figure 7.20 presents representative instants, while Figure 7.21 reports the QGroundControl interface with global multi–waypoint mission.

In the first image (Figure 7.20), the nominal operation in `OFFBOARD` is observed: A* plans on the OccupancyGrid (black track), the B-spline produces a regularized path (red track) and the drone follows its orientation while maintaining constant altitude. In the second image the red rectangle defining the obstacle verification area is shortened as the waypoint falls within the local costmap (blue arrow); consequently the control corridor is limited up to the target. In the absence of cells beyond the threshold inside the corridor, no alarms are generated and the vehicle continues in `MISSION` without having to activate `OFFBOARD`. The third image shows the system's ability to exploit narrow gaps when they are more favorable: A* selects the passage and the controller follows the smooth path orientation, which explains the transit of the drone in areas with a positive cost while respecting safety margins. In the last image, the waypoint, effectively positioned at an obstacle, is unreachable: the projection timeout is triggered, the log is issued *"Waypoint unreachable"* and the drone enters `HOLD`. In this state A* continues to search for solutions, but exiting the `HOLD` requires user intervention.

Figure 7.21 confirms the described dynamics: the drone indicator stops at waypoint 4 and does not reach waypoint 5.

In reference to Figure 7.22, the pattern of velocities is observed in correspondence with the changes of state during the mission. Since all waypoints are set at the

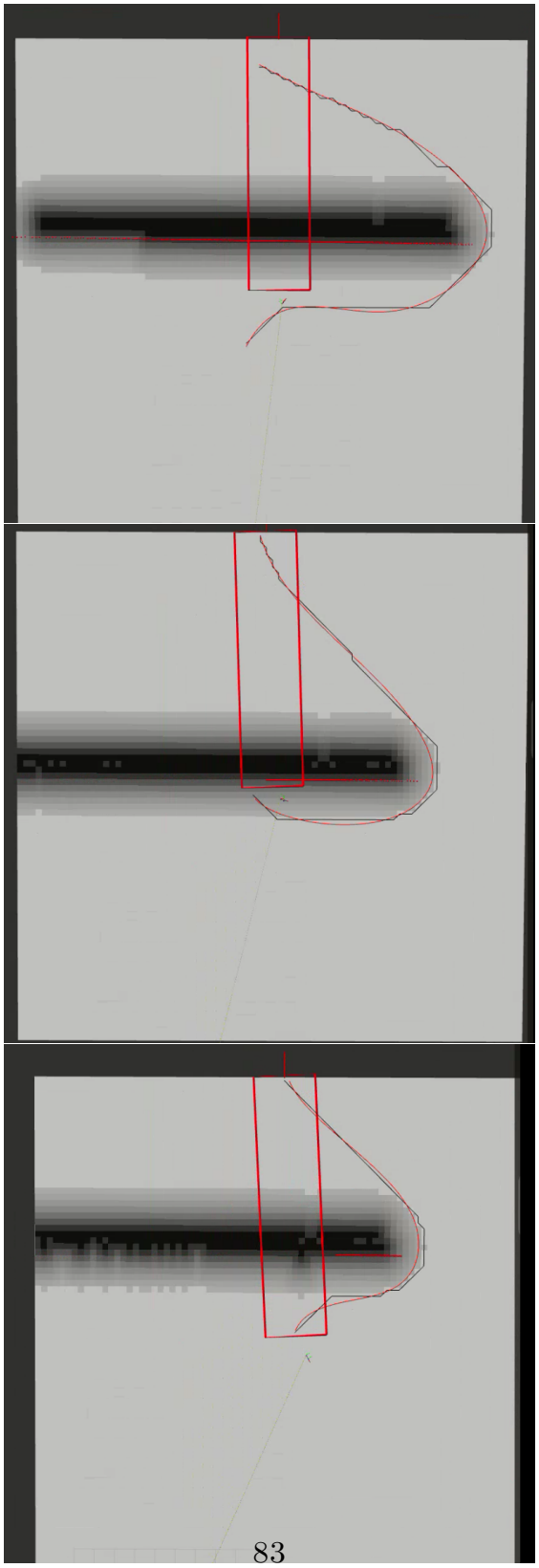

Figure 7.14: Drone simulation in World 2 - RViz view - Collision Avoidance

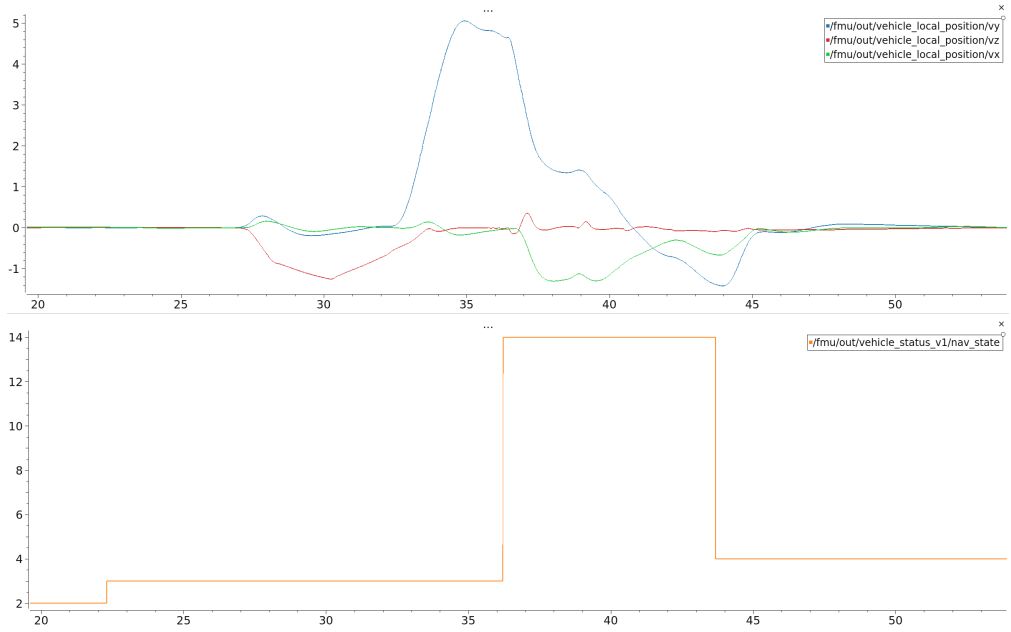

Figure 7.15: Velocity and State transition graphs of drone simulation in World 2 - Collision Avoidance

same altitude, the vertical component is almost zero: v_z does not show significant variations, confirming the maintenance of altitude. The components in the plane (v_x, v_y) instead highlight the modulations imposed both by the state transitions and by the distance from the obstacles: in the phases of approaching the constraints the controller progressively reduces the planar speed (until stopping, if necessary), while in the free sections there is a recovery towards the nominal speed. The short discontinuities coincide with the handovers between modes, followed by sections at a more stable speed when the corridor is devoid of penalized cells.

If the waypoints are assigned to different altitudes, the flight profile is mode–dependent. In `MISSION` the UAV advances towards the target along an oblique trajectory, simultaneously combining planar displacement and altimetric variation. In `OFFBOARD`, on the contrary, altitude is kept constant to prioritize bypassing obstacles in the horizontal plane. At the end of the avoidance maneuver and upon re-entry into `MISSION`, the system preliminarily performs a pure vertical transfer (null component on the xy plane) until alignment with the next waypoint altitude; only then does it resume planar motion towards the goal. This mode-dependent behavior yields a characteristic velocity pattern across state transitions, which is observable in simulation.

Figure 7.23 highlights the variation in speed as a function of state transitions

Figure 7.16: Drone simulation in World 2 with new parameters - RViz view - Collision Avoidance

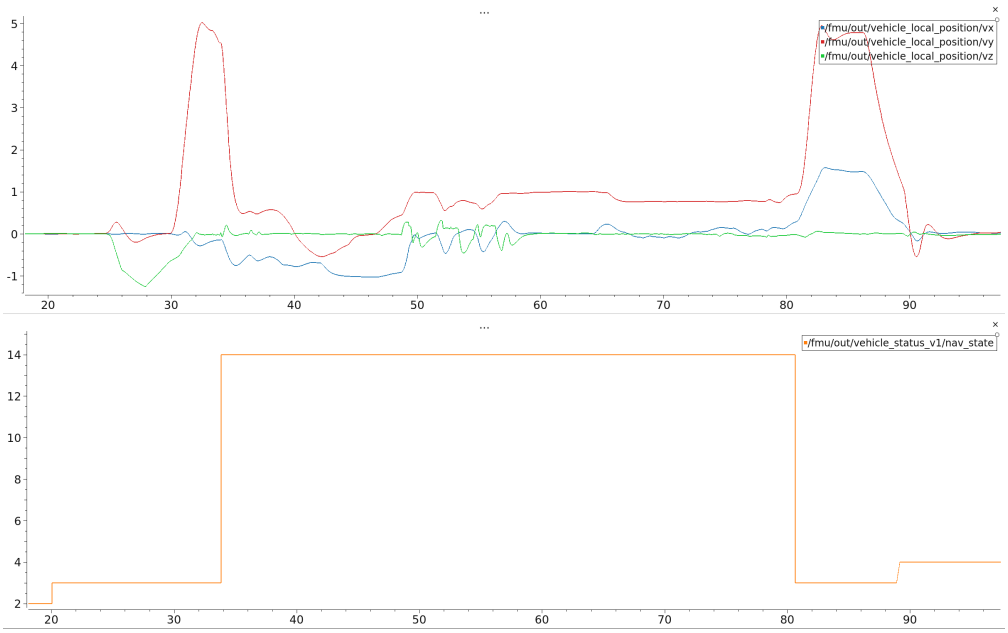

Figure 7.17: Velocity and State transition graphs of drone simulation in World 2 with new parameters - Collision Avoidance

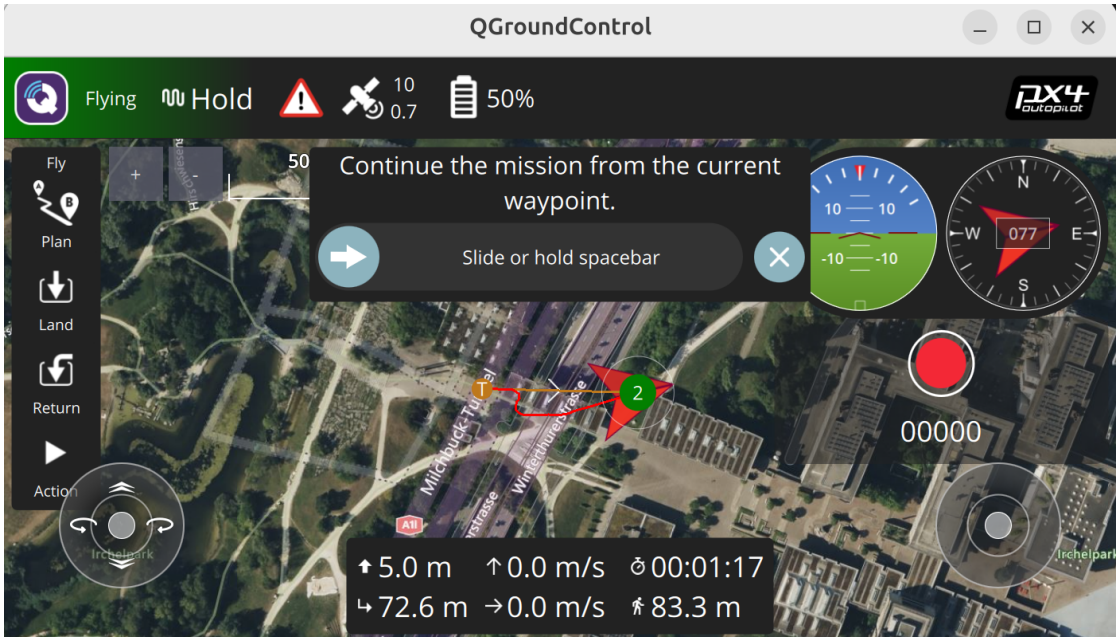

Figure 7.18: QGroundControl interface of drone simulation in World 2 with new parameters

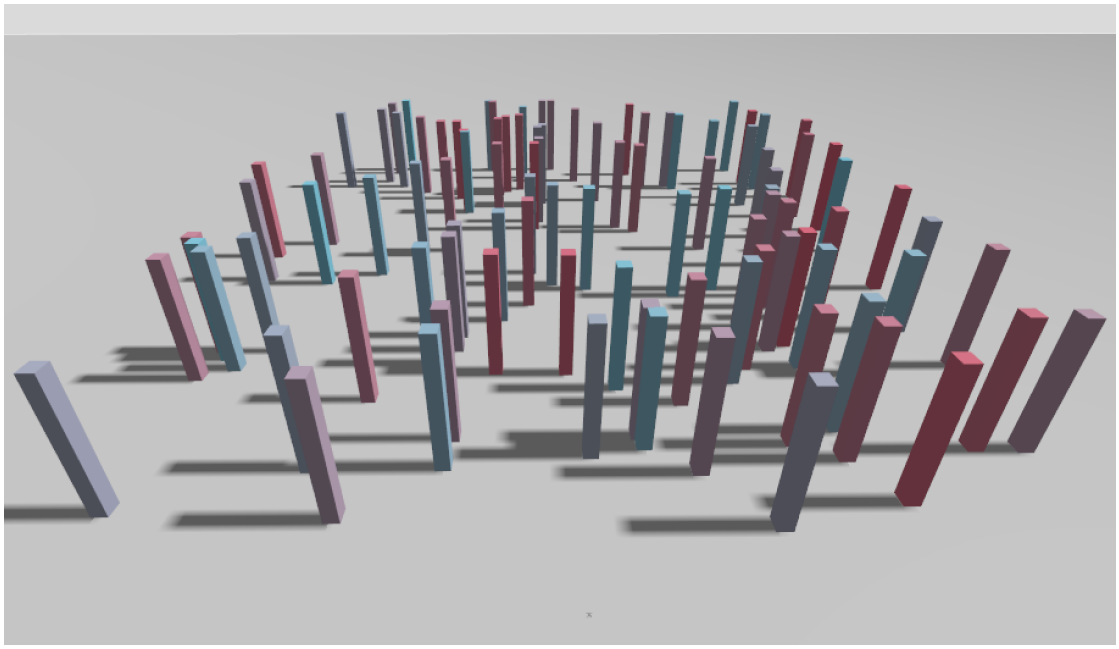

Figure 7.19: Drone World 3 in Gazebo

for a mission in World 3 that, after takeoff, includes three waypoints at different altitudes: $5m$, $10m$ and $15m$. A first, negligible entry into `OFFBOARD` is visible and can be ascribed to sensor noise. In the first `MISSION` segment the UAV reaches the first waypoint—at the same altitude as take off—then heads to the second while climbing; accordingly, in the second part of the `MISSION` state all three velocity components are present. After reaching the second waypoint, the vehicle proceeds toward the third one but encounters an obstacle and switches to `OFFBOARD`; the z-component of the velocity remains approximately constant until the return to `MISSION`. At that point, the v_x and v_y drop to (near) zero to allow a pure vertical ascent; once the altitude of the third waypoint is reached, the motion resumes in the xy plane.

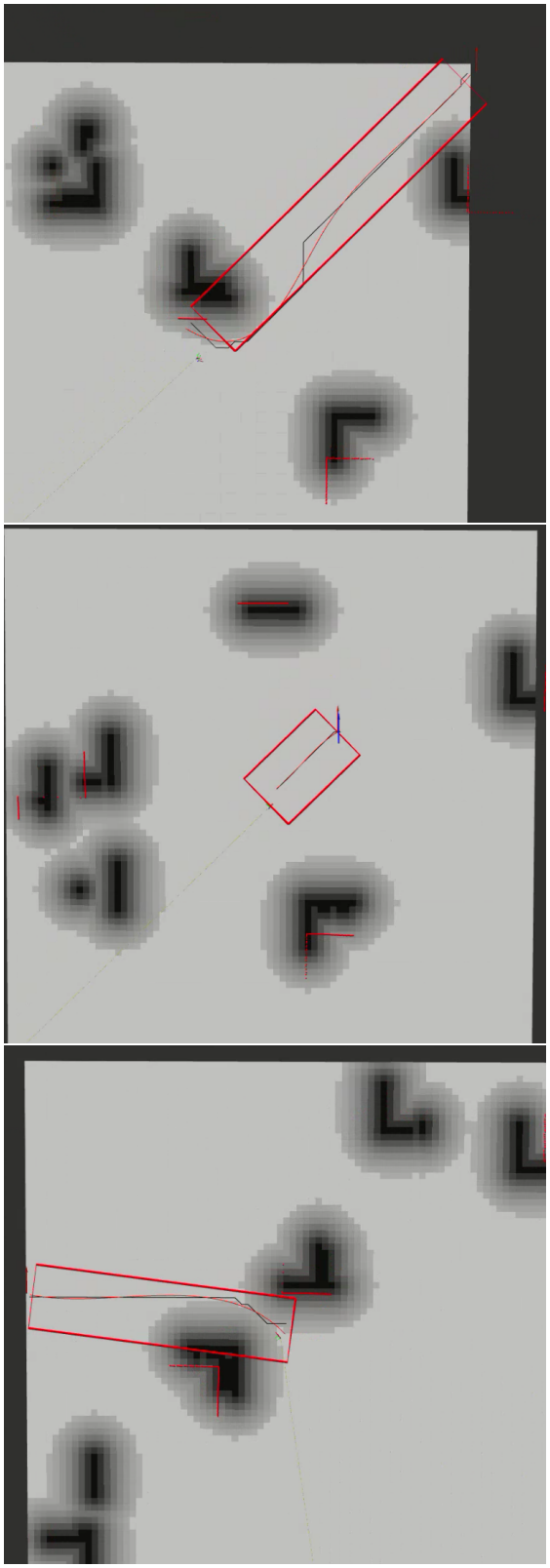

Figure 7.20: Drone simulation in World 3 - RViz view - Collision Avoidance

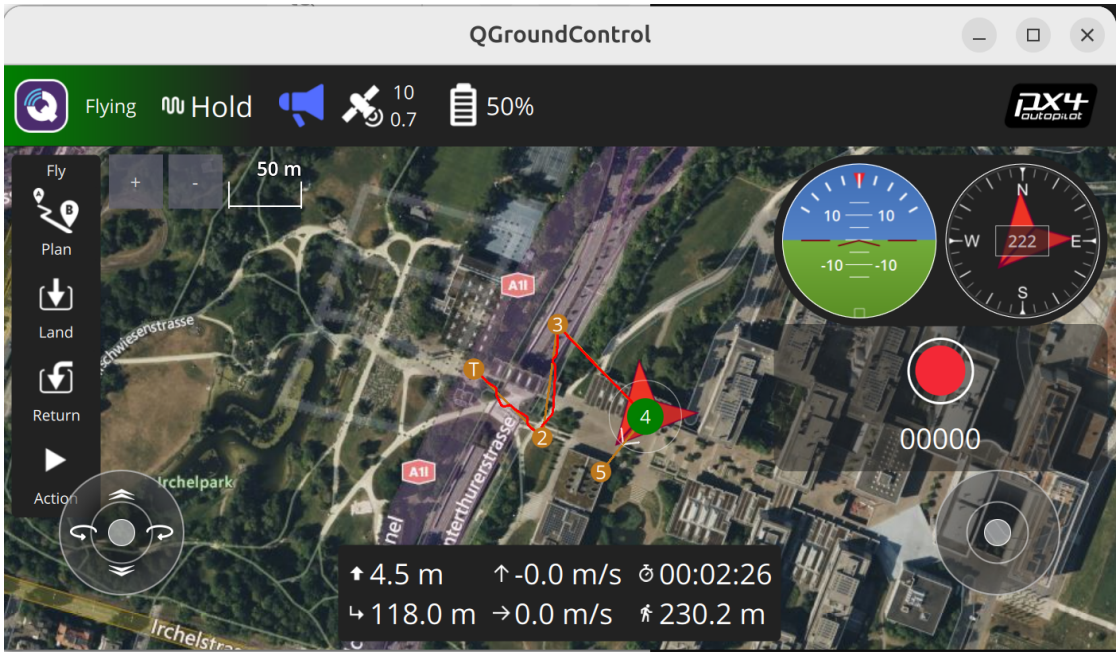

Figure 7.21: QGroundControl interface of drone simulation in World 3 - Collision Avoidance

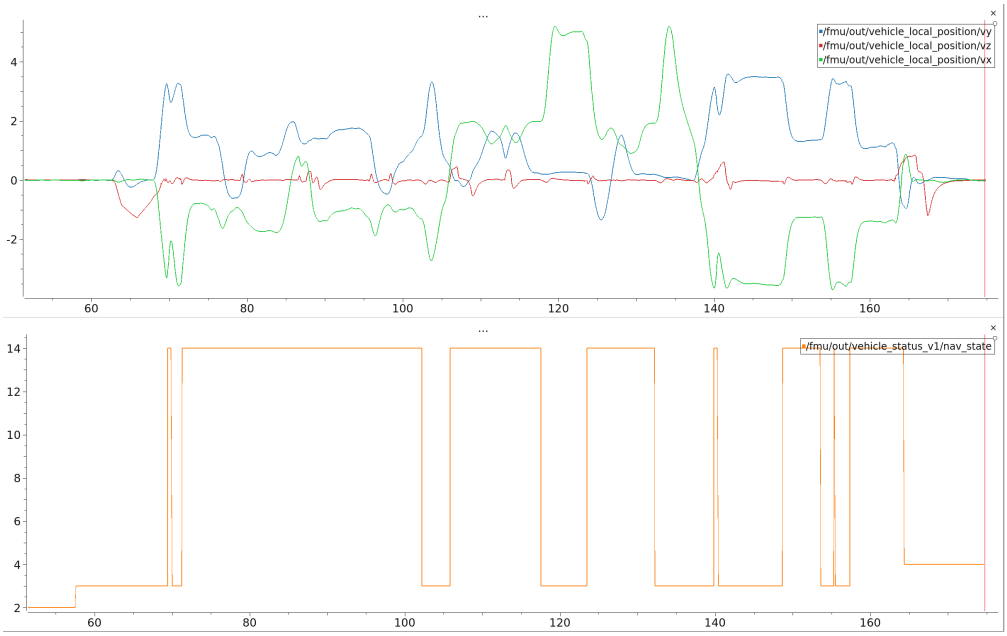

Figure 7.22: Velocity graph and State transition graph of drone simulation in World 3 - Collision Avoidance

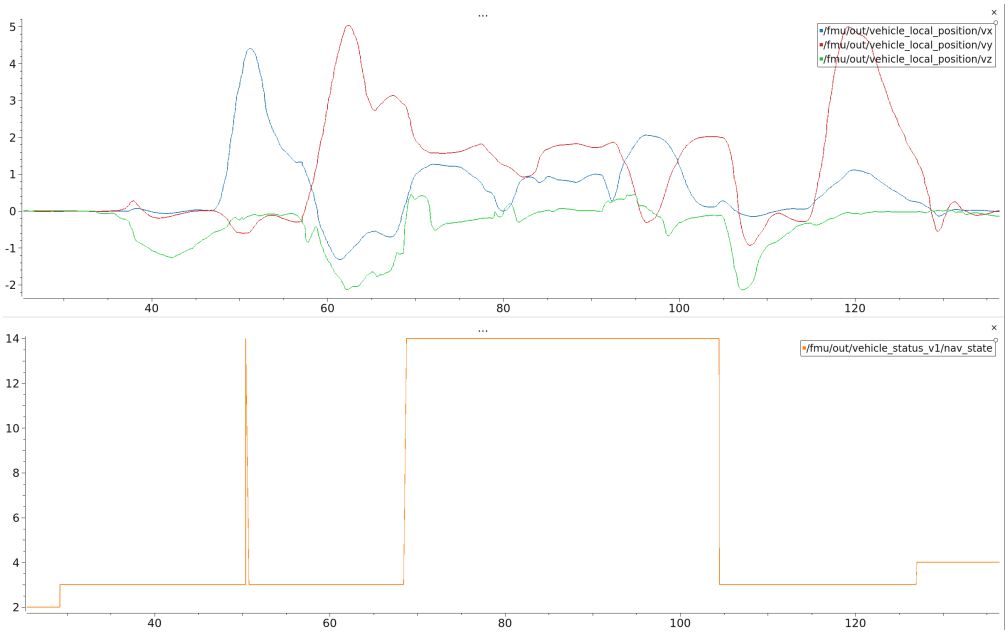

Figure 7.23: Velocity and State transition graphs of drone simulation in World 3 with waypoints at different altitudes - Collision Avoidance

Chapter 8

Conclusions

A UAV navigation pipeline integrated with PX4–ROS 2 was presented, converting mission waypoints into local goals projected into the costmap, planning with A* using a "soft" cost model (exponential penalty near obstacles), and tracking smoothed paths via B-splines, with collision prevention and safety-state management.

The simulations, conducted on both nominal and challenging scenarios, highlight reliable end-to-end operation: the combination of goal projection, exponential penalty and smoothing favors wider corridors and more regular trajectories, reducing exposure to high-cost regions and preventing stalls via projection and A* timeouts. A predictable sensitivity to operating parameters emerged (offboard speed, projection-angle span, costmap size); under conservative choices, tracking remains stable and planning proceeds without unnecessary entries into `HOLD`. The overall architecture is modular, allowing each component to be tuned or replaced independently.

Some results may have been partially influenced by the simulation's computational load, which can affect the UAV's behavior by increasing the command-to-state latency—i.e., the delay between issuing a request to enter `HOLD/OFFBOARD` and the actual transition. This is particularly evident in *Collision Prevention in Mission*, where two unexpected behaviors were observed: (i) a back-and-forth motion before entering `HOLD`, and (ii) when no "bounce" occurs, a stoppage only after traveling several additional meters. Under real conditions or on more capable platforms, the transition to `HOLD` is expected to be nearly instantaneous, without oscillations or significant overshoot.

Real-world tests were scheduled at the company but could not be executed due to time constraints; these trials will constitute the next step to consolidate the simulation evidence and, crucially, to perform systematic parameter tuning oriented to maximum safety (e.g., calibrated limits for offboard speed, adaptive projection-angle policies, and costmap window sizing matched to dynamics and sensor latency).

Future work also includes integrating the **Bug** fallback to navigate around obstacles whose extents exceed the costmap window, and **extending the pipeline to the 3D case** (altitude handling, volumetric obstacles, and planners/controllers consistent with full spatial dynamics).

Appendix A

Turttlebot Algorithms

A.1 Goal Projector

Algorithm 5: GoalProjector Node: project goal inside local costmap and publish

Subscribers:
`/goal_pose` → GOAL_CALLBACK (PoseStamped)
`/odom` → ODOM_CALLBACK (Odometry)
`/local_costmap/costmap` → COSTMAP_CALLBACK (OccupancyGrid)
`/goal_reached` → GOAL_REACHED_CALLBACK (Bool)
Publishers:
`/projected_goal` ← PoseStamped
Parameters:
`publish_rate`, `topic_goal`, `topic_odom`, `topic_costmap`, `topic_goal_reached`, `topic_projected_goal`
Initialization:
Create pubs/subs; start timer with period 1/`publish_rate`
State: `map_ready`←False, `goal_reached`←False
`current_pose`←None, `global_goal`←None, `local_costmap`←None
Callback COSTMAP_CALLBACK(OccupancyGrid msg):
`local_costmap` ← msg; `map_ready` ← True
Callback ODOM_CALLBACK(Odometry msg):
Build PoseStamped from header+pose → `current_pose`
Callback GOAL_REACHED_CALLBACK(Bool msg):
`goal_reached` ← msg.data
Callback GOAL_CALLBACK(PoseStamped goal_msg):
`global_goal` ← goal_msg
if not (`map_ready` **and** `current_pose`) **then**
  return
end if

$P \leftarrow$ PROJECT_GOAL_WITHIN_COSTMAP(`current_pose`, `global_goal`)
if P exists **then**
    publish P on `/projected_goal`
end if
Timer thread UPDATE_PROJECTED_GOAL():
if `goal_reached` **then**
    return
end if
if not (`map_ready` **and** `current_pose` **and** `global_goal`) **then**
    return
end if
$P \leftarrow$ PROJECT_GOAL_WITHIN_COSTMAP(`current_pose`, `global_goal`)
if P exists **then**
    publish P on `/projected_goal`
end if
Function PROJECT_GOAL_WITHIN_COSTMAP(start:PoseStamped, goal:PoseStamped):
    $res \leftarrow$ `local_costmap.info.resolution`; compute dx, dy, distance d
    $steps \leftarrow \max(1, \lfloor d/(0.5\,res) \rfloor)$
    Straight line (backward sampling):
for $i \leftarrow steps$ **to** 1 **do**
    Sample $p = (x, y)$ at fraction $i/steps$ along start→goal
    if IS_CELL_FREE(x, y) **then**
        return CREATE_POSE_STAMPED$(x, y, goal)$
    end if
end for
    Alternative directions: $\alpha \in \{-30°, -15°, 15°, 30°\}$
for each angle α **do**
    $\theta \leftarrow \text{atan2}(dy, dx)$; unit offset $(\cos(\theta + \alpha), \sin(\theta + \alpha))$
    for $i \leftarrow steps$ **to** 1 **do**
        $p = (x, y) = \text{start} + (i/steps) \cdot res \cdot (\cos(\theta + \alpha), \sin(\theta + \alpha))$
        if IS_CELL_FREE(x, y) **then**
            return CREATE_POSE_STAMPED$(x, y, goal)$
        end if
    end for
end for
return None
Function IS_CELL_FREE(x, y):
    From `local_costmap.info` get $res, width, height, origin$
    $mx = \lfloor (x - origin\_x)/res \rfloor$, $my = \lfloor (y - origin\_y)/res \rfloor$
if $0 \leq mx < width$ **and** $0 \leq my < height$ **then**
    $val \leftarrow$ `local_costmap.data`$[my \cdot width + mx]$; **return** $(val = 0)$ **or** $(val = -1)$
else
    return False

end if
Function CREATE_POSE_STAMPED(x, y, ref):
Make PoseStamped with ref header, position (x,y,0), orientation $w = 1.0$;
return pose
=0

A.2 Path Follower

Algorithm 6: PathFollower Node: send path to Nav2 `FollowPath` and handle result

Subscribers:
`/astar_path` → POSE_ARRAY_CALLBACK (PoseArray)
Publishers:
`/cmd_vel` ← Twist
`/controller_failed` ← Bool
Action Client:
`/follow_path` ↔ `nav2_msgs/FollowPath`
Parameters:
`topic_astar_path=/astar_path`, `topic_cmd_vel=/cmd_vel`, `topic_controller_failed=/controller_failed`
`controller_id=FollowPath`, `goal_checker_id=goal_checker`, `follow_path_action=/follow_path`
Initialization:
Create pubs/subs; create FollowPath action client
State: `latest_path`←None; `_current_goal_handle`←None
Optionally wait for action server to be available
Callback POSE_ARRAY_CALLBACK(PoseArray msg):
if msg.poses is empty **then**
warn and **return**
end if
Convert PoseArray → Path (copy header; wrap each pose into PoseStamped)
if Path.poses is empty **then**
publish Bool(True) on `/controller_failed`; **return**
end if
`latest_path` ← Path; SEND_PATH_TO_CONTROLLER(Path)
Procedure SEND_PATH_TO_CONTROLLER(Path):
if Path.poses is empty **then**
publish Bool(True) on `/controller_failed`; **return**
end if
Build `FollowPath.Goal`:

goal.path ← Path;
goal.controller_id = `FollowPath`; goal.goal_checker_id = `goal_checker`
Send with `send_goal_async(goal)` and register HANDLE_GOAL_RESPONSE
Procedure HANDLE_GOAL_RESPONSE(future):
`_current_goal_handle` ← future.result()
if goal not accepted **then**
    publish Bool(True) on `/controller_failed`; **return**
end if
Register `goal_handle.get_result_async()` with HANDLE_RESULT
Procedure HANDLE_RESULT(future):
status ← future.result().status; `_current_goal_handle` ← None
if status = `STATUS_ABORTED` **then**
    publish Bool(True) on `/controller_failed`
else
    publish Bool(False) on `/controller_failed`
end if
Procedure PUBLISH_STOP():
publish `Twist()` on `/cmd_vel`
Main:
    init rclpy; create node `PathFollower`; spin; on KeyboardInterrupt: stop, destroy, shutdown

A.3 Goal Checker

Algorithm 7: GoalChecker Node: goal reach monitoring and notification

Subscribers:
    `/odom` → ODOM_CALLBACK (Odometry)
    `/projected_goal` → GOAL_CALLBACK (PoseStamped)
Publishers:
    `/goal_reached` ← Bool
Parameters:
    `goal_tolerance`, `topic_odom`, `topic_projected_goal`, `topic_goal_reached`, `check_frequency`
Initialization:
    Create pubs/subs; start timer with period 1/`check_frequency`
    State: `current_pose`←None, `goal_pose`←None, `already_reached`←False
Callback ODOM_CALLBACK(Odometry msg):
    `current_pose` ← msg.pose.pose
Callback GOAL_CALLBACK(PoseStamped msg):

`goal_pose` ← msg.pose
Timer thread CHECK_GOAL_REACHED():
if `current_pose`=None **or** `goal_pose`=None **then**
return
end if
$dx \leftarrow$ `goal_pose.position.x` − `current_pose.position.x`; $dy \leftarrow$ `goal_pose.position.y` − `current_pose.position.y`
$d \leftarrow \sqrt{dx^2 + dy^2}$; *reached* ← ($d <$ `goal_tolerance`)
build Bool message m with $m.data \leftarrow$ *reached*
if *reached* **and not** `already_reached` **then**
publish m on `/goal_reached`; `already_reached` ← True
else if not *reached* **then**
publish m on `/goal_reached`; `already_reached` ← False
end if
Main:
init rclpy; create node `GoalChecker`; spin; on interrupt: destroy, shutdown

A.4 Manager

Algorithm 8: NavigationManager Node: supervise planner/controller and switch to Bug fallback

Subscribers:
`/goal_pose` → GOAL_CALLBACK (PoseStamped)
`/controller_failed` → CONTROLLER_FAIL_CALLBACK (Bool)
`/goal_reached` → GOAL_REACHED_CALLBACK (Bool)
`/bug0/abort` → BUG0_ABORT_CALLBACK (Bool)
`/astar_failed` → ASTAR_FAIL_CALLBACK (Bool)
Publishers:
`/fallback_to_bug` ← Bool
`/use_recovery` ← Bool
Parameters:
`topic_goal`, `topic_controller_fail`, `topic_goal_reached`, `topic_bug_abort`, `topic_astar_fail`, `topic_fallback_to_bug`, `topic_use_recovery`
Initialization:
Create pubs/subs
State: `controller_failures`← 0, `fallback_active`←False, `current_goal`←None, `goal_completed`←False
Callback GOAL_CALLBACK(PoseStamped msg):

```
    current_goal←msg; controller_failures← 0; fallback_active←False;
goal_completed←False
    publish Bool(False) on /fallback_to_bug ▷ prefer main controller (DWB)
Callback ASTAR_FAIL_CALLBACK(Bool msg):
if goal_completed then
    return
end if
if msg.data and not fallback_active then
    publish Bool(True) on /fallback_to_bug; fallback_active←True
else
    ▷ if msg.data=False, planner is OK; no immediate change
end if
Callback CONTROLLER_FAIL_CALLBACK(Bool msg):
if goal_completed then
    return
end if
if msg.data then ▷ controller failure
    controller_failures← controller_failures + 1
    if controller_failures ≥ 3 and not fallback_active then
        publish Bool(True) on /fallback_to_bug; fallback_active←True
    end if
else▷ controller success
    if controller_failures > 0 then
        ▷ reset failure counter
    end if
    controller_failures← 0
    if fallback_active then
        publish Bool(False) on /fallback_to_bug; fallback_active←False
    end if
end if
Callback BUG0_ABORT_CALLBACK(Bool msg):
if goal_completed then
    return
end if
if msg.data and fallback_active then
    publish Bool(False) on /fallback_to_bug; fallback_active←False;
controller_failures← 0
end if
Callback GOAL_REACHED_CALLBACK(Bool msg):
if msg.data and not goal_completed then
    publish Bool(False) on /fallback_to_bug
    fallback_active←False; controller_failures←    0;
goal_completed←True
end if
```

45 **Main:**
init rclpy; create node `NavigationManager`; spin; on interrupt: destroy, shutdown

Appendix B

Drone Algorithms

B.1 GPSToPose

Algorithm 9: GPSToPose Node

Subscribers:
`/fmu/out/home_position` → HomeCallback (HomePosition; QoS: BEST_EFFORT, TRANSIENT_LOCAL, KEEP_LAST(10))
`/fmu/out/position_setpoint_triplet` → WpCallback (PositionSetpointTriplet; same QoS)
Publishers:
`/goal_pose` ← PoseStamped (ENU, frame `odom`)
`/goal_ned` ← PoseStamped (NED, frame `odom`)
Parameters:
`topic_home`, `topic_wp`, `topic_goal_pose`, `topic_goal_ned`
Initialization:
Create pubs/subs with sensor QoS
State: `home_received`←False; `home_lat, home_lon, home_alt`← (0,0,0)
Callback HomeCallback(HomePosition msg):
`home_lat`←`msg.lat`; `home_lon`←`msg.lon`; `home_alt`←`msg.alt`
`home_received`←True
Callback WpCallback(PositionSetpointTriplet msg):
if `home_received`=**False then**
    return ▷ home not yet available
end if
`wp`←`msg.current`
if `wp.type` ≠ `POSITION`(0) **then**
    return ▷ ignore non-POSITION waypoints
end if
Build UTM CRS from home longitude

15 $zone \leftarrow 1 + \left\lfloor \frac{\texttt{home\_lon} + 180}{6} \right\rfloor$; $\texttt{epsg} \leftarrow 32600 + zone$
16 $\texttt{proj\_enu} \leftarrow$ PROJ($\texttt{epsg}, \texttt{always\_xy} = \texttt{true}$)
17 **Project** $(\texttt{home\_lon}, \texttt{home\_lat}) \mapsto (x_0, y_0)$; $(\texttt{wp.lon}, \texttt{wp.lat}) \mapsto (x_1, y_1)$
18 $dx \leftarrow x_1 - x_0$; $dy \leftarrow y_1 - y_0$; $dz \leftarrow \texttt{wp.alt} - \texttt{home\_alt}$
19 **Publish NED goal** (frame `odom`): $x \leftarrow dy$; $y \leftarrow dx$; $z \leftarrow dz$
`ned.header.stamp`←NOW(); `ned.header.frame_id`←"odom"
`ned.pose.position.x`← dy; `ned.pose.position.y`← dx; `ned.pose.position.z`← dz
PUBLISH(`/goal_ned`, `ned`)
20 **Publish ENU goal** (frame `odom`): $x \leftarrow dx$; $y \leftarrow dy$; $z \leftarrow dz$
`pose.header.stamp`←NOW(); `pose.header.frame_id`←"odom"
`pose.pose.position.x`← dx; `pose.pose.position.y`← dy; `pose.pose.position.z`← dz
`pose.pose.orientation.w`← 1.0
PUBLISH(`/goal_pose`, `pose`)
21 **Procedure** MAIN():
RCLPY.INIT(); `node`←NEW `GPSToPose`()
SPIN(`node`); DESTROY_NODE(`node`); RCLPY.SHUTDOWN()

B.2 Goal Projector

Algorithm 10: GoalProjector Node

1 **Subscribers:**
`/goal_pose` → GOAL_CALLBACK (PoseStamped)
`/local_costmap/costmap` → COSTMAP_CALLBACK (OccupancyGrid)
`/goal_reached` → GOAL_REACHED_CALLBACK (Bool)
2 **Publishers:**
`/projected_goal` ← PoseStamped
`/projection_timeout` ← Bool
3 **Parameters:**
`topic_goal`, `topic_costmap`, `topic_goal_reached`
`topic_projected_goal`, `topic_projection_timeout`
`publish_rate` (Hz), `projection_timeout_sec`
`side_angles_deg` (e.g., $[-30, -15, 15, 30]$)
4 **Initialization:**
Create pubs/subs (RELIABLE, TRANSIENT_LOCAL for costmap)
State: `current_pose`←None; `global_goal`←None; `local_costmap`←None
`map_ready`←False; `goal_reached`←False

`no_valid_since`←None; `timeout_sent`←False
`side_angles_rad`← RADIANS(`side_angles_deg`) *Starttimerwithperiod*1/`publish_rate`→
UPDATE_PROJECTED_GOAL
Init TF buffer+listener
Callback COSTMAP_CALLBACK(OccupancyGrid msg):
`local_costmap`←msg; `map_ready`←True
Function GET_CURRENT_POSE(`target`="odom", `source`="base_link") → PoseStamped|None:
try TF lookup (timeout 2s) ⇒ build PoseStamped from transform; else return None
Callback GOAL_CALLBACK(PoseStamped msg):
`global_goal`←msg; `no_valid_since`←None

if `timeout_sent` **then** publish Bool(False) on `/projection_timeout`; `timeout_sent`←False
end if
Function PROJECT_GOAL_WITHIN_COSTMAP(start, goal) → PoseStamped|None:
$res \leftarrow$ `local_costmap.info.resolution`
$dx \leftarrow goal.x - start.x$; $dy \leftarrow goal.y - start.y$; $d \leftarrow \sqrt{dx^2 + dy^2}$
$steps \leftarrow \max(30,\ \lfloor d/(0.25 \cdot res) \rfloor)$
(1) Straight line: for $i = steps \downarrow 1$ test $p = (start + i/steps \cdot [dx, dy])$ with IS_CELL_FREE; if free ⇒ MAKE_POSE(p,goal)
(2) Side directions: $h \leftarrow$ ATAN2(dy, dx); for $a \in$ `side_angles_rad` and $i = steps \downarrow 1$
$p \leftarrow start + (i/steps) \cdot res \cdot [\cos(h + a), \sin(h + a)]$; if IS_CELL_FREE$(p) \Rightarrow$ MAKE_POSE(p,goal)
else return None
Function IS_CELL_FREE$(x, y) \rightarrow$ bool:
From `local_costmap.info`: $res, width, height, origin$
$mx = \lfloor (x - origin.x)/res \rfloor$; $my = \lfloor (y - origin.y)/res \rfloor$
if in bounds: $val \leftarrow$ `local_costmap.data`$[my \cdot width + mx]$; **return** $(val \leq 5) \vee (val = -1)$; else **return** False
Function MAKE_POSE(p, ref) → PoseStamped:
Copy `header`←ref.header; set $(x, y, 0)$; orientation.w= 1.0; **return** PoseStamped
Timer UPDATE_PROJECTED_GOAL():
if `timeout_sent` **then**
return ▷ wait for new goal or goal reached
end if
`current_pose`←GET_CURRENT_POSE()
if not(`map_ready` **and** `current_pose` **and** `global_goal`) **then**
return
end if

proj ← PROJECT_GOAL_WITHIN_COSTMAP(`current_pose`, `global_goal`)
if *proj* ≠None **then**
    publish *proj* on `/projected_goal`; `no_valid_since`←None;
    return
end if
Timeout handling:
now ← CLOCK.NOW()
if `no_valid_since`=None **then**
    `no_valid_since`← *now*;
    return
end if
elapsed ← (*now* − `no_valid_since`) in seconds
if *elapsed* ≥ `projection_timeout_sec` **and not** `timeout_sent` **then**
    publish Bool(True) on `/projection_timeout`; `timeout_sent`←True
end if
Callback GOAL_REACHED_CALLBACK(Bool msg):
if msg.data **then**
    `no_valid_since`←None
    if `timeout_sent` **then** publish Bool(False) on `/projection_timeout`; `timeout_sent`←False
    end if
end if

B.3 Obstacle Detector

Algorithm 11: ObstacleDetector Node

Subscribers:
    `/local_costmap/costmap` → COSTMAP_CALLBACK (OccupancyGrid; QoS: RELIABLE, TRANSIENT_LOCAL, KEEP_LAST(1))
    `/goal_pose` → GOAL_POSE_CALLBACK (PoseStamped)
Publishers:
    `/obstacle_detected` ← Bool
    `/obstacle_area_marker` ← Marker (LINE_STRIP)
Parameters:
    `costmap_topic`, `goal_pose_topic`, `obstacle_detected_topic`, `obstacle_marker_topic`
    `half_width` (m), `check_interval` (s), `step_size` (m)
Initialization:
    Create pubs/subs with QoS above
    Init TF buffer + listener

Start timer every `check_interval` → CHECK_OBSTACLES
State: `latest_costmap`←None; `goal_x, goal_y`←None
Callback COSTMAP_CALLBACK(OccupancyGrid msg):
`latest_costmap`←msg
Callback GOAL_POSE_CALLBACK(PoseStamped msg):
`goal_x`←msg.pose.position.x; `goal_y`←msg.pose.position.y
Timer CHECK_OBSTACLES():
if `latest_costmap`=None **then**
return
end if
TF lookup: target=`latest_costmap.header.frame_id`, source=`base_link`, timeout 0.5 s
if TF fails **then**
return
end if
$(drone\_x, drone\_y) \leftarrow$ translation from TF
if `goal_x`=**None or** `goal_y`=**None then**
return
end if
$dx \leftarrow$ `goal_x` $- drone\_x$; $dy \leftarrow$ `goal_y` $- drone\_y$; $L \leftarrow \sqrt{dx^2 + dy^2}$
if $L < 0.01$ **then**
return
end if
Direction unit vectors: $(dir\_x, dir\_y) \leftarrow (dx/L, dy/L)$; $(perp\_x, perp\_y) \leftarrow (-dir\_y, dir\_x)$
Costmap bounds: from `info.resolution, width, height, origin`
Max length within map $L_{\max}$:
$L_{\max} \leftarrow L$; step $\leftarrow$ `step_size`
for $\ell = step, 2\,step, \ldots$ up to L **do**
$(x, y) \leftarrow (drone\_x, drone\_y) + \ell\,(dir\_x, dir\_y)$
if (x, y) outside costmap **then** $L_{\max} \leftarrow \ell$;
break
end if
end for
Scan occupied cells in rectangle (forward $[0, L_{\max}]$, lateral $[-$`half_width`$,$`half_width`$]$):
$found \leftarrow$False; iterate all indices idx with $cost > 0$
Map $(idx) \rightarrow (mx, my) \rightarrow (wx, wy)$ (cell center in world)
$(dx, dy) \leftarrow (wx - drone\_x,\ wy - drone\_y)$
$forward \leftarrow dx\,dir\_x + dy\,dir\_y$; $lateral \leftarrow dx\,perp\_x + dy\,perp\_y$
if $0 \le forward \le L_{\max}$ **and** $|lateral| \le$ `half_width` **then**
$found \leftarrow$True;
break
end if

PUBLISH(`/obstacle_detected`, Bool($found$))
PUBLISH_OBSTACLE_AREA_MARKER($drone\_x, drone\_y, dir\_x, dir\_y, perp\_x, perp\_y, L,$ `half_`
Procedure PUBLISH_OBSTACLE_AREA_MARKER($drone\_x, drone\_y, dir\_x, dir\_y, perp\_x, perp$
Build Marker (LINE_STRIP, red, lifetime 1s, frame=`costmap frame`)
Use $L_{\max}$ as length; corners:
$C_1 = (drone + -\frac{W}{2}\, perp)$; $C_2 = (drone + L_{\max}\, dir - \frac{W}{2}\, perp)$; $C_3 = (drone + L_{\max}\, dir + \frac{W}{2}\, perp)$; $C_4 = (drone + \frac{W}{2}\, perp)$; close C_1
Append corners as `Points`; PUBLISH(`/obstacle_area_marker`, Marker)

Bibliography

[1] Navaneetha Krishna Chandran, Mohammed Thariq Hameed Sultan, Andrzej Łukaszewicz, Farah Syazwani Shahar, Andriy Holovatyy, and Wojciech Giernacki. «Review on Type of Sensors and Detection Method of Anti-Collision System of Unmanned Aerial Vehicle». In: *Sensors* 23.15 (2023). ISSN: 1424-8220. DOI: 10.3390/s23156810. URL: https://www.mdpi.com/1424-8220/23/15/6810 (cit. on pp. 3–5, 8, 9).

[2] Min Jin, Jun Tao, Zhen Qiu, Jingpo Bai, Chong Liang, and Xiaoning Li. «Autonomous Obstacle Avoidance and Navigation Method for Unmanned Aerial Vehicles Based on Multi-Sensor Fusion Algorithm». In: *2024 3rd International Conference on Energy and Electrical Power Systems (ICEEPS)*. 2024, pp. 1329–1335. DOI: 10.1109/ICEEPS62542.2024.10693226 (cit. on pp. 4–6).

[3] António Raimundo, D. Peres, N. Santos, Pedro Sebastião, and Nuno Souto. «USING DISTANCE SENSORS TO PERFORM COLLISION AVOIDANCE MANEUVRES ON UAV APPLICATIONS». In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLII-2/W6 (Aug. 2017), pp. 303–309. DOI: 10.5194/isprs-archives-XLII-2-W6-303-2017 (cit. on pp. 4, 8).

[4] Priya Gupta, Fredrick Ambrose, Sahil Naik, Muiz Tanki, Aarti Kumbhar, and Jainam Shah. «STM32 Based Quadcopter For Obstacle Avoidance Using Ultrasonic Sensor». In: *2023 6th International Conference on Advances in Science and Technology (ICAST)*. 2023, pp. 122–126. DOI: 10.1109/ICAST59062.2023.10454993 (cit. on p. 4).

[5] Rahul H Kumar, Anil M Vanjare, and S N Omkar. «Autonomous Drone Navigation using Monocular Camera and Light Weight Embedded System». In: *2023 International Conference for Advancement in Technology (ICONAT)*. 2023, pp. 1–6. DOI: 10.1109/ICONAT57137.2023.10080483 (cit. on pp. 5, 8, 11, 13, 19–21).

[6] Aadi Nath Mishra, Stephanos Papakonstantinou, and Volker Gollnick. «A Soft Actor-Critic Based Reinforcement Learning Approach for Motion Planning of UAVs Using Depth Images». In: *2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC)*. 2024, pp. 1–10. DOI: 10.1109/DASC62030.2024.10748743 (cit. on pp. 5, 12).

[7] Kuijun Zuo, Xuan Cheng, and Heng Zhang. «Overview of Obstacle Avoidance Algorithms for UAV Environment Awareness». In: *Journal of Physics: Conference Series* 1865.4 (Apr. 2021), p. 042002. DOI: 10.1088/1742-6596/1865/4/042002. URL: https://dx.doi.org/10.1088/1742-6596/1865/4/042002 (cit. on p. 5).

[8] Danyang Li, Jingao Xu, Zheng Yang, Yishujie Zhao, Hao Cao, Yunhao Liu, and Longfei Shangguan. «Taming Event Cameras With Bio-Inspired Architecture and Algorithm: A Case for Drone Obstacle Avoidance». In: *IEEE Transactions on Mobile Computing* 24.5 (2025), pp. 4202–4216. DOI: 10.1109/TMC.2024.3521044 (cit. on pp. 6, 12).

[9] Zhihao Li, Mingqiu Li, Pengnian Wu, and Yixuan Li. «Dynamic Obstacle Detection for Quadrotors with Event Cameras». In: *2024 4th International Conference on Intelligent Communications and Computing (ICICC)*. 2024, pp. 126–130. DOI: 10.1109/ICICC63565.2024.10780721 (cit. on pp. 6, 11).

[10] Hashim Hashim. «Advances in UAV Avionics Systems Architecture, Classification and Integration: A Comprehensive Review and Future Perspectives». In: *Results in Engineering* (Apr. 2025), p. 103786. DOI: 10.1016/j.rineng.2024.103786 (cit. on pp. 6, 10, 11).

[11] Hang Yu, Fan Zhang, Panfeng Huang, Chen Wang, and Li Yuanhao. «Autonomous Obstacle Avoidance for UAV based on Fusion of Radar and Monocular Camera». In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 5954–5961. DOI: 10.1109/IROS45743.2020.9341432 (cit. on p. 6).

[12] Q. Liang, Z. Wang, Y. Yin, W. Xiong, J. Zhang, and Z. Yang. «Autonomous aerial obstacle avoidance using LiDAR sensor fusion». In: *PLoS ONE* 18.6 (2023), e0287177. DOI: 10.1371/journal.pone.0287177. URL: https://doi.org/10.1371/journal.pone.0287177 (cit. on p. 8).

[13] B. Cybulski, A. Wegierska, and G. Granosik. «Accuracy comparison of navigation local planners on ROS-based mobile robot». In: *2019 12th International Workshop on Robot Motion and Control (RoMoCo)*. 2019, pp. 104–111. DOI: 10.1109/RoMoCo.2019.8787346 (cit. on p. 9).

[14] D. M. Huynh, A. D. Nguyen, H. N. Nguyen, H. D. Tran, D. A. Ngo, J. Pestana, and A. Q. Nguyen. «Implementation of a HITL-Enabled High Autonomy Drone Architecture on a Photo-Realistic Simulator». In: *2022 11th International Conference on Control, Automation and Information Sciences (ICCAIS)*. 2022, pp. 430–435. DOI: 10.1109/ICCAIS56082.2022.9990214 (cit. on p. 9).

[15] Luis Felipe Muñoz Mendoza, Guillermo García-Torales, Cuauhtémoc Acosta Lúa, Stefano Di Gennaro, and José Trinidad Guillen Bonilla. «Trajectories Generation for Unmanned Aerial Vehicles Based on Obstacle Avoidance Located by a Visual Sensing System». In: *Mathematics* 11.6 (2023). ISSN: 2227-7390. DOI: 10.3390/math11061413. URL: https://www.mdpi.com/2227-7390/11/6/1413 (cit. on p. 9).

[16] Skylar X. Wei, Anushri Dixit, Shashank Tomar, and Joel W. Burdick. «Moving Obstacle Avoidance: A Data-Driven Risk-Aware Approach». In: *IEEE Control Systems Letters* 7 (2023), pp. 289–294. DOI: 10.1109/LCSYS.2022.3181191 (cit. on p. 11).

[17] Paula Fraga-Lamas, Lucía Ramos, Víctor M. Mondéjar-Guerra, and Tiago Fernández-Caramés. «A Review on IoT Deep Learning UAV Systems for Autonomous Obstacle Detection and Collision Avoidance». In: *Remote Sensing* 11 (Sept. 2019), p. 2144. DOI: 10.3390/rs11182144 (cit. on p. 12).

[18] Euihyeon Cho, Hyeongjin Kim, Pyojin Kim, and Hyeonbeom Lee. «Obstacle Avoidance of a UAV Using Fast Monocular Depth Estimation for a Wide Stereo Camera». In: *IEEE Transactions on Industrial Electronics* 72.2 (2025), pp. 1763–1773. DOI: 10.1109/TIE.2024.3429611 (cit. on p. 12).

[19] A.P. Kalidas, C.J. Joshua, A.Q. Md, S. Basheer, S. Mohan, and S. Sakri. *Deep Reinforcement Learning for Vision-Based Navigation of UAVs*. Encyclopedia. Accessed on 26 August 2025. 2025. URL: https://encyclopedia.pub/entry/52646 (cit. on p. 12).

[20] Ioannis Daramouskas, Isidoros Perikos, Ioannis Hatzilygeroudis, Vaios J. Lappas, and Vasilios Kostopoulos. «A Methodology For Drones to Learn How to Navigate And Avoid Obstacles Using Decision Trees». In: *2020 11th International Conference on Information, Intelligence, Systems and Applications (IISA*. 2020, pp. 1–4. DOI: 10.1109/IISA50023.2020.9284337 (cit. on p. 13).

[21] Hongqian Huang, Yanzhou Li, and Qing Bai. «An improved A star algorithm for wheeled robots path planning with jump points search and pruning method». In: *Complex Engineering Systems* 2 (July 2022), p. 11. DOI: 10.20517/ces.2022.12 (cit. on p. 17).

[22] Scott D. Lai. *A* Pathfinding Visualization*. https://scottdlai.github.io/a-star-pathfinding/. 2017 (cit. on p. 18).

[23] *ROS 2 Documentation: Humble*. https://docs.ros.org/en/humble/index.html. Documentazione online della distribuzione Humble Hawksbill. 2022 (cit. on p. 23).

[24] Mauro Martini and Marcello Chiaberge. *ROS 2: Introduction & Tutorial*. Lecture slides, Politecnico di Torino, M.Sc. Mechatronic Engineering. Course material. 2023 (cit. on pp. 24–26).

[25] *Gazebo Sim*. https://gazebosim.org/. Sito web ufficiale. 2025 (cit. on p. 26).

[26] *RViz2 — ROS 2 Documentation (Rolling)*. https://docs.ros.org/en/rolling/p/rviz2/index.html. Documentazione online — distribuzione Rolling. 2025 (cit. on p. 27).

[27] *PX4 User Guide (v1.12)*. https://docs.px4.io/main/en/. Documentazione online, versione stabile v1.12. 2021 (cit. on pp. 28–30).

[28] *Nav2 — Nav2 1.0.0 documentation*. https://docs.nav2.org/. Documentazione online — guida principale di Navigation2. 2025 (cit. on p. 34).

[29] *DWB Controller — Nav2 Documentation*. https://docs.nav2.org/configuration/packages/configuring-dwb-controller.html. Accessed: 2025-09-05 (cit. on p. 41).