POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Simultaneous exploration and mapping for fully autonomous vehicles: a mixed graph-mesh approach

Tesi di Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: **Pietro Tenani**

Student ID: 945753
Advisor: Prof. Lorenzo Mario Fagiano
Co-advisors: Danilo Saccani, Michele Bolognini
Academic Year: 2020-21

Abstract

The focus of this thesis is the development of a novel controller for mobile robots used in exploration and mapping tasks, in outdoor and non GPS-denied environments. Specifically, the controller is run on a multicopter equipped with a RGB-D camera and localisation sensor.

The proposed controller solves the problem of real-time autonomous exploration and mapping, with real-time obstacles reconstruction. The controller uses a navigation graph for collecting information about the obstacle-free area and computing paths, and a triangular mesh for representing the obstacles and selecting the best target.

The controller is composed of two loosely interacting data flows. The Mapping part merges the images generated by the camera into a global pointcloud, that is converted into a mesh. Frontiers are computed and the best one is selected by weighting the distance and the expected information gain factors. The Exploration part generates a local convex polytope that represents free space, integrates new nodes into the global navigation graph, and plans the path that reaches the node closer to the received target. Additional measures on the safety of the path and edge cases are implemented as well.

The thesis introduces novelties in a variety of domains, namely the study of the use of mesh in exploration and mapping duty, a novel approach for the generation of a convex polytope, a passive obstacle avoidance technique, and a rarely studied frontier definition.

The proposed controller is successfully tested on a simulated environment in Gazebo, and compared with a state of the art exploration and mapping framework.

Keywords: drone, autonomous vehicle, exploration, mapping, mesh based mapping

Abstract in lingua italiana

Questa tesi si pone l'obiettivo di presentare un controllore per robot mobili, con l'obiettivo di esplorare e mappare ambienti esterni in cui è presente un segnale di localizzazione. In particolare, il controllore verrà eseguito su un multicottero fornito di telecamera RGB-D e di un sensore di localizzazione.

Il controllore presentato affronta il problema dell'esplorazione e mappatura autonoma, con la visualizzazione in tempo reale degli ostacoli individuati. Il controllore costruisce un grafo di navigazione per raccogliere informazioni sulle zone senza ostacoli e per calcolare i percorsi da seguire; e una mesh triangolare come rappresentazione degli ostacoli e per selezionare il migliore obiettivo da raggiungere.

Il controllore è costituito da due flussi di dati solo debolmente interconnessi. Per la Mappatura viene costruita una nuvola di punti globale generata dall'unione delle immagini fornite dalla camera. Da questa viene creata la mesh, e poi determinati i punti di frontiera e selezionato il migliore, utilizzando una metrica che unisce la distanza del punto e l'informazione acquisita stimata. Per l'Esplorazione si genera un politopo convesso che contiene esclusivamente spazio senza ostacoli, vengono estratti dei nodi da integrare nel grafo di navigazione, e viene calcolato il percorso che più si avvicina all'obiettivo ricevuto. Sono poi eseguiti dei controlli aggiuntivi per garantire la sicurezza dei percorsi calcolati, e vengono gestiti alcuni importanti casi limite.

Gli aspetti innovativi introdotti nella tesi sono molteplici, in particolare lo studio sull'uso di una mesh per svolgere operazioni di esplorazione e mappatura, un approccio innovativo per la generazione del politopo convesso, un sistema di controllo di collisioni passivo, e l'adozione di un metodo di definizione delle frontiere raramente studiato in letteratura.

Il controllore presentato è stato validato tramite l'utilizzo del software di simulazione "Gazebo", è stato inoltre confrontato con un software che attualmente rappresenta lo stato dell'arte tra gli approcci di esplorazione e mappatura.

Keywords: drone, veicolo autonomo, esplorazione, mappatura, mappatura basata su mesh

Contents

Introduction

In recent years mobile robots usage has been increasing significantly and steadily. Many different fields can benefit from the use of mobile robots. Some examples include industrial robots, material handling in warehouses, operations in hazardous environments, inspection of structures, and search and rescue operations (figure 1).

The huge interest gained by mobile robots led to many different research areas, different solutions can focus on different aspects:

- the environment where the robot should live in: terrestrial, aerial, aquatic, or multiple;
- environment characteristics: the space the robot is in can be indoor or outdoor, explored or unknown, static or dynamic, safe or hazardous;
- movement freedom: the robot can have two degrees of freedom (i.e. it moves on a plane), threes degree of freedom (it moves along all three axis) or more (the robot can move and rotate); moreover the robot can be holonomic (the controllable degrees of freedom is equal to total degrees of freedom) or not;
- sensor type: the robot can be equipped with different sensors: inertial measurement unit (IMU), wheel sensor, global positioning system (GPS), LiDAR, vision camera, radio transmitter and receiver, and many other
- human or autonomous-guided: a human-guided robot is directly guided by an opera-

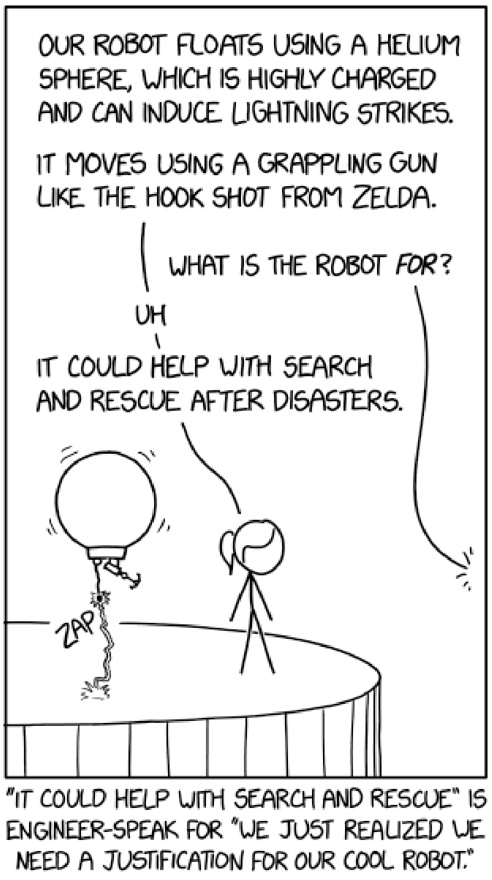

OUR ROBOT FLOATS USING A HELIUM SPHERE, WHICH IS HIGHLY CHARGED AND CAN INDUCE LIGHTNING STRIKES.
IT MOVES USING A GRAPPLING GUN LIKE THE HOOK SHOT FROM ZELDA.
WHAT IS THE ROBOT FOR?
UH
IT COULD HELP WITH SEARCH AND RESCUE AFTER DISASTERS.
ZAP
"IT COULD HELP WITH SEARCH AND RESCUE" IS ENGINEER-SPEAK FOR "WE JUST REALIZED WE NEED A JUSTIFICATION FOR OUR COOL ROBOT."

Figure 1: Humorous analysis of the research interest toward mobile robots [31]

tor, while an autonomous-guided robot must
autonomously reach its objective.

This thesis focuses on the development of a control system for an autonomous-guided aerial holonomic robot, that operates along all three axis in an unknown outdoor and static environment. The robot is equipped with localisation sensors and a stereoscopic RGB-D camera[1]. The objective is to autonomously map the area and provide an online reliable representation of the obstacles, while keeping an eye on computational efficiency.

Thesis outline

In this section the structure of the thesis is briefly explained.

- *Introduction* (current chapter) contains a broad introduction of the topic presented in the thesis, an analysis of the State of the Art on some of the main topics covered in the thesis, and a short description of the hardware and software used during the development.
- *Problem analysis and Proposed solution* (chapter 1) contains the problem formulation, with an analysis of the goal, requirements and limitation of the problem. Then it describes the proposed approach, by explaining the data structures used in the development and the structure of the proposed controller.
- *Obstacles management and Target selection* (chapter 2) describes how the obstacles are managed, from the local raw pointcloud to the final mesh through the global obstacle pointcloud. Then it analyses the target selection procedure, that extracts the next point to be reached.
- *Graph management and Navigation* (chapter 3) describes how the navigation graph is built and maintained, how the graph is used for navigating through the environment, and some techniques adopted to ensure a safe navigation.
- *Test and Results* (chapter 4) describes how the proposed controller has been tested. It explains the environment and parameters, the qualitative and quantitative results, and a comparison between the controller and a State of the Art algorithm.
- *Conclusion* (chapter 5) summarise the thesis development and the obtained results, highlighting the innovations brought by the work, and proposing some interesting

[1]RGB-D stands for "Red Green Blue Depth", an RGB-D camera therefore produces both a coloured video stream and a depth value for each pixel. In other words, the camera provides a stream of images, each pixel being characterized by its colour and its distance from the camera

future developments.

State of the Art

The problem tackled in the thesis is composed of various elements. A brief analysis to some of the most important advancements in Mapping, Path planning, Exploration is presented, together with a description of the SLAM approach and a comparison with some existing projects that share similarities with the proposed controller.

Mapping

The mapping task consists on building and maintaining a structure of the environment. The data structure used for the representation must be efficient, flexible and robust. A mapping framework should be able to separate obstacles from free space, or at least it should be efficient in storing information about one of the two elements. A few mapping approaches are shown.

Graph A graph is a common data structure for storing information about free space in an environment [35]. The use of graph-based maps allows to efficiently perform path planning, moreover, it is easy to integrate additional data. However the use of graphs is not efficient for storing information about obstacles.

Occupancy grid Occupancy grids are a simple representation of an environment. An occupancy grid subdivides the space into square (in two-dimensional environments) or cubic (in three-dimensional environments) cells, each cell being classified as free or occupied according to a probability value. This value indicates the probability that a cell is occupied, and is computed from sensor readings. The status of the cell (Free, Occupied and Unknown) is derived by comparing the probability value of that cell with one or more thresholds.

The introduction of occupancy grid is due to [30] and [36].

An evolution of occupancy grids are Quadtrees [13] (for two-dimensional environments) and Octrees [28] (for three-dimensional environments). These are hierarchical data structures, structured in a tree-like fashion, that allow the optimisation of time and space requirements for storing sparse data. See figure 2 for a visual comparison between an occupancy grid and the quadtree.

Some frameworks have been developed in order to optimize the workflow, a notable ex-

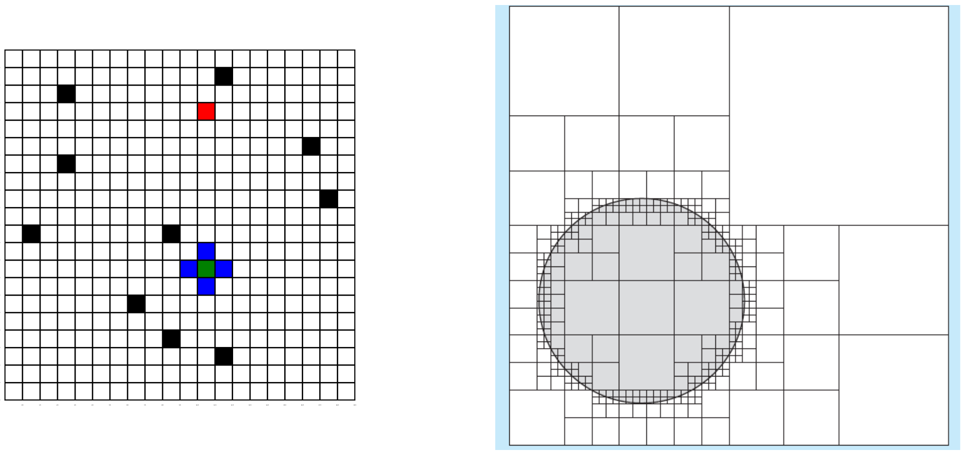

(a) Occupancy grid (b) Quadtree

Figure 2: Occupancy grid and Quadtree

ample is Octomap [16], a C++ multi platform library that guarantees high performance with complex environments.

Occupancy grids are quite efficient in representing both free space and obstacles, and also the navigation task can be easily handled by using this data structure.

Mesh A mesh is a data structure that can conveniently be used for representing solid objects, in this case, three-dimensional obstacles.

Common meshing algorithms usually construct triangular-faced meshes, however meshes with a higher number of edges-per-face have been studied as well [5]. The research interest toward meshes produced a great number of meshing algorithms [3], each one specialised on a different challenge (speed, accuracy, dataset size, etc.). Some broadly used algorithms are Poisson reconstruction [21] and Ball Pivoting Algorithm [4].

An interesting subfield of mesh reconstruction adds the real time constraint: i.e. data acquisition and mesh reconstruction happens simultaneously. Some examples of real time mesh reconstruction are [33] and [27].

Meshes are an excellent representation for the obstacles in an environment, while their use in navigation tasks is limited (a description of navigation mesh is in [41] and [45]).

Other Other data structures can be used for storing information about obstacles or free space. Two notable examples are Truncated Signed Distance Function (TSDF), introduced in [9], while an example of its use is kintinuous [46]; and surfels, used in known surface reconstruction frameworks, like ElasticFusion [47].

Path planning

The path planning task consists in finding the best path that connects the initial position to the target position through the map. Several approaches are available.

Classic graph search algorithms can be used if a graph representing the map is available. Some well known examples of such algorithms are the Dijkstra algorithm [11] and A* path planning algorithm [10].

Graph search algorithms can be easily adapted to work in a grid environment [49, 50].

Exploration

Exploration is the act of discovering new places and expanding the known region. Various techniques can be used to explore an environment, the most common are next best view and frontier exploration. Next best view [8] tries to find the best safe position that the robot can reach that helps the exploration. On the other hand, frontier exploration [48] consists on building a frontier set of unexplored regions and trying to visit them all, until the frontier set is emptied and the exploration is therefore terminated.

The most common definition of frontier using occupancy grids consists on finding the Unknown cells that are near Free cells [48]. These cells are then grouped together and a frontier set is built. This approach has the advantage of providing a homogeneous exploration of the space, and it maximise the explored region.

This approach is however not suitable if the main goal is to map an object in an unknown (and uninteresting) environment. Under this settings, a more appropriate definition of frontier is Unknown cell near Occupied cell. An example of this approach is [38], which presents an algorithm specialized in mapping walls. The solution presented in the paper only considers Unknown and Occupied cells, finds the "direction of growing" of a wall and determine the best view position that avoids obstacles and follows the direction found (see figure 3 for a visual analysis of the various steps).

SLAM

Simultaneous Localisation and Mapping (SLAM) solves the difficult problem of using visual sensor readings (either camera, range measurements, or lidar) both for building a map of an unknown environment and deriving the robot location. This results in a more robust system that requires fewer sensors, at the cost of a higher computational load. One of the main advantages of SLAM is the concept of loop closure. By computing the

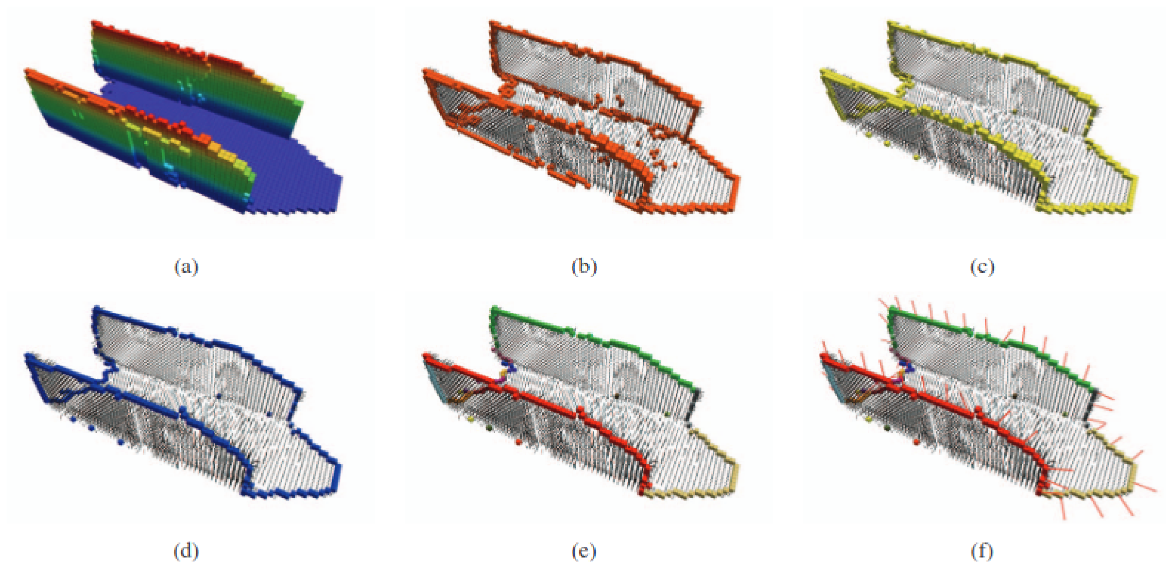


Figure 3: Phases of the surface frontier approach described in [38]

"similarity" between two frames, with different estimated positioning, it is possible to determine whether they represent the same place and estimate the drift of the sensors. Once a loop is detected it is possible to close it by "correcting" all the relevant nodes' position.

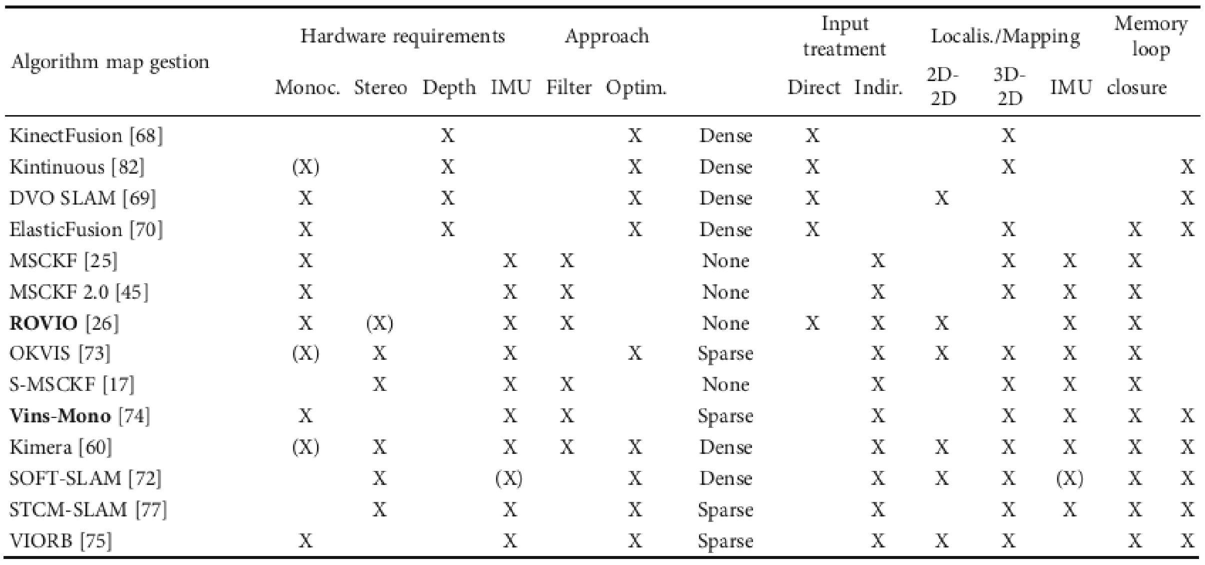
There are a plethora of SLAM algorithms [39], often directly provided as stand-alone ready-to-use framework. They differ in camera type (lidar, mono camera, RGB-D camera etc.), internal representation and output (occupancy grid, TSDF etc), in figure 4 a few well known algorithms are compared to each other. Three notable examples of SLAM algorithms are RTAB-Map [25], ORB-SLAM3 [6] and Kintinuous [46].

Similar approaches

A brief description of three works partially similar to the proposed controller follows. The similarities and differences with each one of them are highlighted.

G-BEAM

Graph-Based Exploration and Mapping (G-BEAM) [7] is a controller developed for an Unmanned Aerial Vehicle that moves at constant height equipped with a planar lidar and a localisation sensor. The controller is required to autonomously build a map of the environment, producing a navigation graph and a human readable representation of the obstacles.



Algorithm map gestion	Hardware requirements				Approach			Input treatment		Localis./Mapping			Memory loop	
	Monoc.	Stereo	Depth	IMU	Filter	Optim.		Direct	Indir.	2D-2D	3D-2D	IMU	closure	
KinectFusion [68]			X			X	Dense	X			X			
Kintinuous [82]	(X)		X			X	Dense	X			X			X
DVO SLAM [69]	X		X			X	Dense	X		X				X
ElasticFusion [70]	X		X			X	Dense	X			X		X	X
MSCKF [25]	X			X	X		None		X		X	X	X	
MSCKF 2.0 [45]	X			X	X		None		X		X	X	X	
ROVIO [26]	X	(X)		X	X		None	X	X	X		X	X	
OKVIS [73]	(X)	X		X		X	Sparse		X	X	X	X	X	
S-MSCKF [17]		X		X	X		None		X		X	X	X	
Vins-Mono [74]	X			X	X		Sparse		X		X	X	X	X
Kimera [60]	(X)	X		X	X	X	Dense		X	X	X	X	X	X
SOFT-SLAM [72]		X		(X)		X	Dense		X	X	X	(X)	X	X
STCM-SLAM [77]		X		X		X	Sparse		X		X	X	X	X
VIORB [75]	X			X		X	Sparse		X	X	X		X	X

Figure 4: State of the art review of SLAM approaches [39]

The adopted solution is based on a graph structure, that contains information both on free space and obstacles. Exploration is performed by computing the "exploration gain" (how much information can be retrieved by visiting the node) of each free space node, and then following the node with the highest gain/distance ratio.

The algorithm is composed of three main sections: polytope generation, graph update and exploration. The Polytope Generation node takes as input the lidar measurement and computes the greatest convex polygon that does not contains obstacles. The Graph Update node combines all the polygons into a single global graph, adds all the necessary nodes and edges, and computes the exploration gain of the newly added nodes. Finally, the Exploration Node searches for the node that maximise the gain function and finds the best path that connects current and target nodes.

Another element developed in the project is the Obstacle Avoidance node, which takes the lidar measurement and the location of the drone (position, direction) and modifies the direction of movements in order to avoid any obstacle on its path.

Similarities and differences The G-BEAM controller shares many similarities with the controller proposed in this thesis. Both controllers run on an Unmanned Aerial Vehicle, and a localisation module needs to be installed (no SLAM is performed). G-BEAM however never changes altitude (thus the approach is two dimensional) and is equipped with a planar lidar instead of a RGB camera.

For the development of the thesis the G-BEAM controller was used as a starting point, the data flow and the subdivision of the work load between nodes has been kept quite similar.

Espeleo planner

The Espeleo planner controller [1] is a controller developed for a Terrestrial Unmanned Vehicle, equipped with an imu and a 3D-lidar. The controller is required to autonomously map a confined and complex environment with traversability constraints and no global localisation.

The controller consists of a SLAM algorithm that localises the robot in the currently known map. The environment is mapped into an occupancy grid structure; a mesh representing obstacles is then built from the occupancy grid, and a traversability graph is generated from the mesh. The frontier set is extracted and the information gain is computed. An obstacle avoidance algorithm guides the robot to the best location.

Similarities and differences The Expeleo planner controller's main similarity with the proposed controller is the internal representation of obstacles and free space, and the process that leads to the optimal frontier. Both controllers use a mesh structure for storing the obstacles and computing frontiers, and a traversability graph for the navigation. However, Espeleo planner is developed for an Unmanned Ground Vehicle (instead of an UAV[2]) in a closed and GPS-denied environment (while the proposed controller should work on an open field with built-in a localisation module). Moreover, Espeleo planner derives the traversability graph from the mesh of the obstacles, while the proposed controller builds the mesh and graph in parallel.

Multi-Resolution Frontier-Based Planner

The planner proposed in the paper [2] is a controller developed for an Unmanned Aerial Vehicle, equipped with a 3D lidar. The controller is required to autonomously build a map of an open environment with obstacles.

The localisation part of the controller is performed by an external module that runs a well known SLAM algorithm. The SLAM algorithm generates a pointcloud that is then used to build a hierarchical three-dimensional occupancy grid. The occupancy grid is analysed in order to determine the frontiers, using the "free near unknown" paradigm. The frontiers are built at various levels on the octomap hierarchy, trying to optimise the computational load and accuracy of the result. The best frontier is selected by computing

[2]UAV stands for Unmanned Aerial Vehicle

the information gain integrated with visibility information, and is recomputed multiple time at different voxel size during the path.

Similarities and differences The similarity between the planner and the proposed approach consists on the execution status: both the works are planned for UAVs that move on an open space with obstacles, are equipped with 3D depth sensors (respectively 3D lidar and stereoscopic camera) and do not tackle the localisation issue (the planner implements a well known SLAM algorithm, the proposed controller mounts a GPS). Moreover, both the approaches relies on frontier based exploration and computes the target frontier multiple times on a single path, trying to optimise the planned path. The main differences between the two approaches are the data structures used in the mapping (occupancy grid versus mesh and traversability graph) and the exploration setting (the planner uses the "plan the free space" approach, while the proposed controller uses the "explore the obstacles" approach.

Technologies description

In the following section a list of the software and hardware used for the project is presented. A short description of each of them follows.

	version	**reference**
drone	Tarot 680 pro	-
camera	RealSense D435	[18]
ROS	ROS Melodic	[42]
gazebo	9.18	[22]
px4	1.12	[14]
PCL	1.8	[37]

Table 1: Main hardware and software technologies used in the thesis

(a) Gazebo logo. (b) ROS Melodic logo.

(c) px4 logo (d) Point cloud library logo

Figure 5: Some software used in the development

Drone

The drone adopted during the development of the thesis is the Tarot 680 Pro, a foldable hexacopter frame with carbon fiber components, with a motor-to-motor distance of 695 mm, ground-to-frame distance of 180 mm, weight of 800 g and up to 2 kg payload capacity. It mounts 13 inches carbon fiber blades, each powered by a 420W brushless motor.

The drone central unit is a ODROID-XU4 [20], that runs Ubuntu 18 and the flight controller. Moreover, it is equipped with an IMU and RGBD camera.

During the thesis development we used a simulated drone in Gazebo.

RealSense D435

Intel® RealSense™ D435 is a depth camera that uses infrared and RGB images to provide a coloured depth stream. The camera is equipped with an infrared projector, two infrared cameras and an RGB camera. The infrared projector projects a pattern that improves depth precision. The infrared camera's streams are combined in a single depth stream with a field of view of 87°x58°, 1280x720 resolution and 90 frames per second. The RGB camera provides a 1920x1080 30fps video. All these data are internally combined into a coloured pointcloud.

The camera can be read through ROS, and both raw camera streams and resulting point-cloud are available to the user.

In order to preserve the mechanical integrity of the camera, during the thesis development we used a simulated camera in Gazebo, imported as an external plugin [19, 32].

ROS

The Robot Operating System (ROS) is an open source software development kit for robotics applications. It is available on Unix-based platform, its major distributions are synchronised with Ubuntu's major releases.

It allows to create and use packages that cooperate seamlessly and efficiently in a multi-threaded environment. A **package** is a self-containing high-level structure. It contains multiple nodes and other elements. A **node** is a basic process that performs computation. Nodes can be programmed using C++ or Python. Communication between nodes consists of messages, a data structure with typed fields, similar to C struct. Messages can be sent through topics and services. A **topic** is a "named bus" that allows unidirectional one-to-many communication. A message published on a topic is received by all the nodes that subscribed to the node. On the other hand, **services** are used for request-response one-to-one message passing.

ROS has gained a wide user support, with many open source packages and an active community support.

The Distribution used in the thesis is ROS Melodic, associated with Ubuntu 18.04.

Flight controller

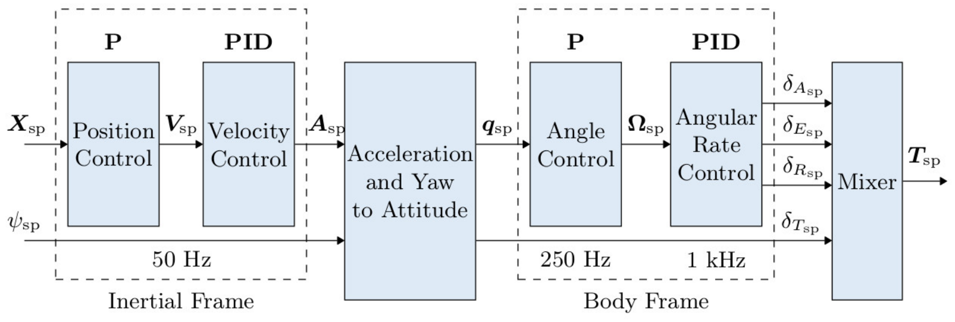


Figure 6: Data flow of px4

The flight controller used is px4, available as an open source project. It allows to control the drone both manually and by setting a desired position. It is composed by a number of smaller controller (either P or PID controllers), each responsible for controlling a specific variable (see figure 6).

The px4 controller is connected with the sensors mounted on the drone (in our case, the IMU) through MAVLink, a lightweight messaging protocol [23] specialized in drone communication. On the other hand, the communication between the px4 and the ROS nodes are bridged through MAVROS, a ROS package that behaves as a proxy between ROS and MAVLink messages.

Gazebo

Gazebo is an open source 3D simulation environment specialized in indoor and outdoor robot simulation. The physics engine is able to efficiently simulate complex environments seamlessly. Many robot and sensor models are available off the shelf, while a complex script and plugin system allows to add many other. Gazebo can interact with ROS, allowing to test the packages on a safe and simulated environment.

The version used in the thesis is Gazebo 9.19.0.

PCL

The Point Cloud Library (PCL) is a cross platform open source library for point cloud processing. It is well supported, optimized and widely used. PCL contains different modules that can be loaded independently, each deals with one specific task (for example filtering, segmentation, features extraction). It is shipped as a stand alone library for C++ with limited Python support. A few functions and ROS packages are devoted to converting ROS messages to PCL.

The version used in the thesis is PCL 1.8.

1 | Problem analysis and Proposed solution

The chapter contains an overview of the proposed controller. In section 1.1 an analysis of the problem is presented, listing goals, requirements and limitations. In section 1.2 the coordinate frames used by the controller are shown. Section 1.3 reveals the data structures adopted for the different tasks and internal processing. Finally, in section 1.4 the structure of the proposed controller is explained, highlighting the nodes that compose the structure and their role.

1.1. Problem analysis

1.1.1. Goal

Mapping The controller must be able to autonomously produce a real time, complete, human and machine readable representation of the structure of a building

1.1.2. Requirements

Complete map The representation produced by the controller must be complete. This means that the difference between the real building and the resulting map must be as small as possible.

Real time All the incoming data must be consumed as soon as they are acquired with no offline operations, and the exploration must not be slowed by the controller's internal computations.

Human and machine readable The representation produced by the controller must be easily understandable by humans and efficiently readable by machines.

Efficient mapping The controller must be efficient, minimizing the time required to complete the mapping.

Efficient operations The time and space performance must be efficient enough to run on limited hardware or high latency scenario.

Autonomous exploration The controller must be as autonomous as possible. This means that the human interventions should be limited and no emergency interventions are allowed

Safety The controller must be able to safely navigate through the environment, and must not damage the drone nor the environment.

1.1.3. Domain limitations

Robot equipment The robot controlled by the controller is a drone. The drone has access to its position and to a stream of images of the front view.

Environment The robot navigates in an outdoor, static and three-dimensional environment.

Limited hardware The controller runs entirely or mostly onboard. Onboard the hardware capacity is very limited, while offboard there is a great latency.

1.2. Coordinate conventions

In order to unambiguously refer to the position of the drone, the position of the obstacles, and the output provided by the sensors, it is necessary to define a coordinate convention.

The convention adopted follows the REP 103 [44] and the REP 105 [29] guidelines. The two REPs are ROS Enhancement Proposals that focus on the definition of "Standard Units of Measure and Coordinate Conventions" and "Coordinate Frames for Mobile Platforms" respectively. These guidelines are broadly accepted (especially within the ROS community) as they help to build a simple, robust and stable coordinate structure.

A number of different coordinate frames are adopted:

odom frame

The odom frame is world-fixed. The position of the drone within the odom frame changes continuously and is computed by an odometry module (inertial, visual, or other). The position of static obstacles within the odom frame does not change with time. Both the position and the orientation of the frame can be arbitrarily determined. For the sake of simplicity, in the proposed controller the drone's starting position and orientation match with the odom frame (i.e. the drone is located at the origin of the odom frame and its forward direction is aligned with the x-axis).

base link frame

The base link frame is drone-fixed. The origin of the coordinate frame is rigidly attached to the mobile robot base, in any arbitrary position or orientation. For the proposed controller, the frame is attached to the drone body, with an (x-forward, y-left, z-up) orientation.

camera frame

The camera frame is drone-fixed. The origin of the frame corresponds to the housing of the camera, with an (z-forward, x-right, y-down) orientation.

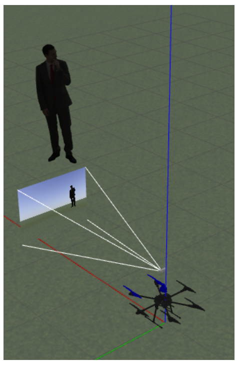
In figure 1.1 is shown a representation of the frames used in the controller. At the beginning of the mapping the odom frame is initialized at the robot position (hence the odom frame and the base link frame coincide). Then the drone starts flying, and is shown that the odom frame stays still, while the base link frame follows the drone movements and rotation. The camera frame is always rigidly linked to the base link frame.

1.3. Data structures

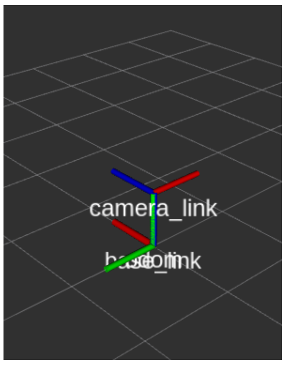
In order to maintain the Exploration and Mapping goals independent, we decided to tackle the two problems in parallel, with the minimum amount of information exchange.

Having little communication between the two sections makes it easy to keep the controller modular, and that allows to solve each problem independently of the other and searching for efficiency. Moreover, it would even be possible to easily exchange one or more components, having only a few interfaces to maintain.

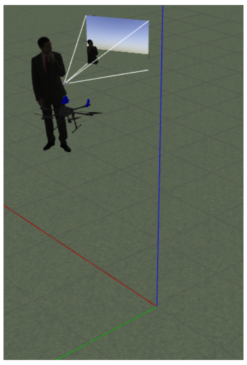
In the following sections, an overview of the structures used is presented.



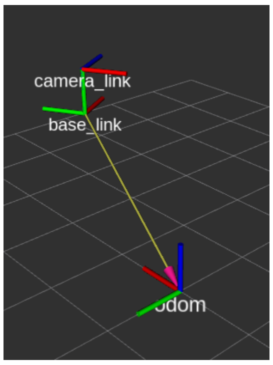
(a) Starting position (simulation)



(b) Starting position (frames)



(c) Flying (simulation)



(d) Flying (frames)

Figure 1.1: Coordinate frames

1.3.1. Mapping

Triangular Mesh

The data structure used to represent the obstacles is a Triangular Mesh. A mesh is an efficient structure that allows representing surfaces, it is easy to visually understand and makes it easy to perform simple geometric operations (e.g. finding the distance between a point and the mesh).

Pointcloud

Another structure used to internally represent obstacles is the Pointcloud. A Pointcloud can store a large number of points, and although being less informative than a Mesh, it allows performing many useful operations very efficiently.

Formal definition of pointcloud and mesh

Mathematically, a point p in space is defined by three coordinates: (p_x,p_y,p_z). Other parameters can be associated with the point, for example the colour (p_R,p_G,p_B), the normal direction of the surface the point is in (p_{Nx},p_{Ny},p_{Nz}) and many others.

A Pointcloud $\mathcal{P}$ is defined as a set of points $\mathcal{P} = \{p_1, p_2, p_n\}$. Each point inside the Pointcloud obviously maintains all the additional information they stored as single points.

A Mesh $\mathcal{M}$ is defined as a set of vertices, edges and faces that defines a shape. The vertices set V is a Pointcloud, each point being a vertex of the shape. The edges set E is composed of a set of pair of points $(E = (p_x, p_y) = (V(x), V(y)))$. The face set F is a set of tuples of edges, with the constraint that the edges must be contiguous and closed. A

Triangular Mesh only has faces composed of three edges.

In equation (1.1) the formal definitions of point, Pointcloud and Mesh. Note that in a Triangular Mesh the number of elements in each face is 3 ($F = \{e_1, e_2, e_3\}$).

$$\begin{cases} p = [x, y, z]^T \\ \\ \mathcal{P} = \{p_1 \cup p_2 \cup \cdots \cup p_n\} \\ \\ \mathcal{M} = \begin{cases} V = \mathcal{P} \\ E = \{(v_1, v_2) \mid v_1, v_2 \in V\} \\ F = \{e_1 \cup e_2 \cup \cdots \cup e_n \mid & e_i \in E \quad \forall i \in [1, n] \quad \wedge \\ & e_i.v_2 = e_{i+1}.v_1 \quad \forall i \in [1, n-1] \quad \wedge \\ & e_n.v_2 = e_1.v_1 \quad \} \end{cases} \end{cases} \tag{1.1}$$

1.3.2. Exploration

Navigation Graph

In order to navigate through the environment, a Navigation Graph is used. The use of a Graph allows benefiting from the well known path planning algorithms based on graphs.

A Navigation Graph $\mathcal{G}$ is composed of Vertices and Edges ($\mathcal{G} = (V, E)$). Each vertex $v \in V$ represents a point in space, is defined by its Cartesian coordinate and can have additional properties, such as being reachable, being visited, etc. The Edges set E is composed of a set of pairs of vertices ($E = \{(v_1, v_2 \mid v_i \in V)\}$), and represent a link between the two vertices. Some properties of edges are the euclidean length of the segment and the traversability of the edge. A Navigation Graph is undirected, so the Edge pair is unordered (i.e. $e_1 = (v_1, v_2) = (v_2, v_1) = e_2$, so e_1 and e_2 represents the same edge).

1.4. Proposed approach

The controller is composed by a number of different nodes, each one devoted to a few operations, such that the controller results being fairly well optimized and the nodes are independent from each other.

In figure 1.2 is shown how the nodes are organized and how they interact with each other.

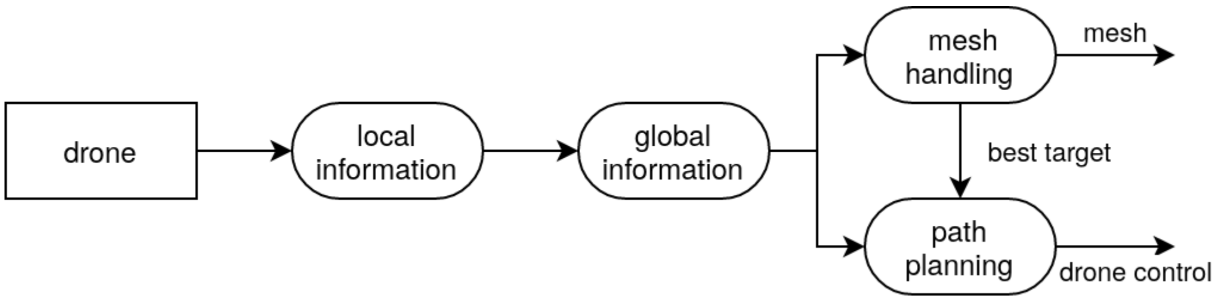


Figure 1.2: High level structure of the controller

1.4.1. Input

The controller expects as input a stream of pointclouds, allegedly generated by an RGB-D camera, and a stream of position transformations, generated by the localisation module. The pointcloud can be coloured or not, organized or not. The transformation tree must be able to convert from the origin of the map to the location of the camera.

1.4.2. Local information

The Local information node is required to parse the raw pointcloud input and extract local data that will be integrated into the global data structures (global Pointcloud and global Graph). It runs at low speed (controlled by the parameter T_{poly}). Its main features are:

Drone moving check (section 2.4) The input pointcloud is not reliable when the drone is moving too fast. This section, therefore, computes the instantaneous speed of rotation and translation, and if one of the two is too high, the obstacle pointcloud is not updated.

Local pointcloud generation (section 2.1) The raw pointcloud provided by the camera mounted on the drone (the Intel 435i [18]) contains a huge amount of points, over 400 thousand, on a relatively small projection plane. The point density is much higher than the requirements, so the raw pointcloud is heavily downsampled into the local obstacle pointcloud.

Polytope generation (section 3.1) The polytope generation module takes as input the obstacle pointcloud and retrieves a convex "safe area" between the drone and the obstacles in which the drone can safely navigate. The polytope must be as big as possible, while keeping the computational load limited.

Graph node extraction (section 3.2) In order to build a robust network the nodes need to be in a useful position and at a safe distance from the obstacles. From the convex polytope generated in the previous module, the graph node extraction module identifies some points inside the polytope and prepares them to be inserted into the global Graph. This is the first step that guarantees the quality and safety of the Exploration.

1.4.3. Global information

The Global information node is required to merge all the local information gathered through the run, build and maintain the global data structures (a global Pointcloud representing obstacles and a global Graph representing free space). Its main features are:

Obstacle pointcloud integration (section 2.1) The obstacle pointcloud integration module merges the global and local cloud, and then performs a filtering in order to maintain the desired pointcloud density.

Graph integration (section 3.3) The graph integration module takes as input the candidate new vertices, inserts them into the global graph if they satisfy some condition on the vertices density, and complete the update by inserting all the needed edges between visible nodes.

Path check (section 3.5) The path check module is used to ensure that every location traversed by the drone is safe and walkable. Upon receiving a path between nodes, it checks whether all the vertices and edges are far enough from the obstacle pointcloud.

1.4.4. Mesh handling

The Mesh handling node is required to generate the Mesh of the obstacles and provide it to the user, together with a wireframe model. Moreover, it processes the Mesh in order to find the most suitable next best view. Its main features are:

Vertices extraction (section 2.2) The vertices extraction module takes as input the global obstacle pointcloud, and processes it in order to extract the points that will become the vertices of the mesh.

Mesh creation (section 2.2) The mesh creation module starts from the vertices and builds the mesh objects.

Target selection (section 2.3) The target selection module uses the newly created mesh in order to compute some relevant metrics and find the most interesting location that should be viewed on camera.

1.4.5. Path planning

The Path planning node uses all the information acquired in the previous nodes and computes the optimal path to the target. Its main features are:

Path creation (section 3.4) The path creation module is devoted to creating the path between the drone's current location and the best target position. It determines the global Graph's vertex closest to the best target computed from the Mesh and computes the path that connects these vertices, taking care of traversing only reachable vertices and edges.

Path check (section 3.5) The path check module ensures that the path computed is completely walkable. It communicates with the Path check module on the Global information node and continues triggering the Path creation module until a safe path is found.

No path available (section 3.4) The no path available module is devoted to handle the edge case in which the newly computed best target is the current vertex in the Graph. In order to avoid a deadlock, a special procedure is followed. First a complete rotation is performed, then the second optimal vertex is set as target, and the current vertex is avoided.

In figure 1.3 the single nodes are expanded, for each node the modules it contains are visible. It is possible to notice the separation between the obstacle and navigation management. Each path is pretty linear and straightforward, with the exception of the Path planning node, which takes data from both the Mesh handling node and the Global information node, and communicates with the Global information node.

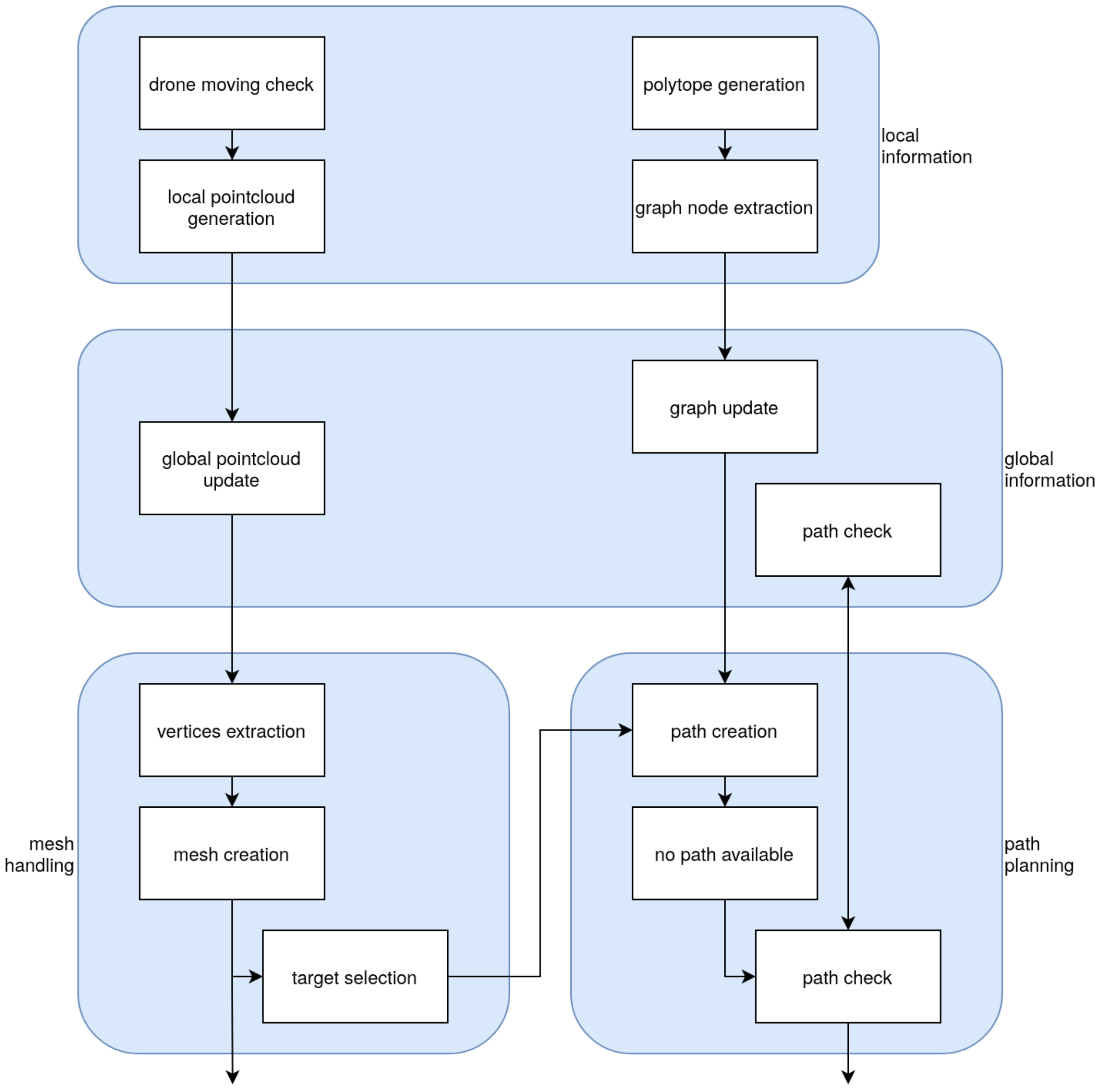


Figure 1.3: Low level structure and data flow

2 | Obstacles management and Target selection

The Obstacles management and Target selection module's task is to store and update a map representing the obstacles recorded by the drone, and to find the area of the map that is the most preferable to map, i.e. establish the best target that the drone should film next.

Referring to the global controller structure shown in figure 1.3, the current chapter covers the modules on the right, as in figure 2.1. The correspondence between the sections of this chapter and the referred module is quite straightforward, except for the "drone moving check" module that is explained in the Pointcloud drift section (section 2.4).

In section 2.1 is explained how the controller elaborates the raw pointclouds $\mathcal{P}_{raw}$ received by the camera, and converts them into local pointclouds $\mathcal{P}_{loc}$ that are merged into a global obstacle pointcloud $\mathcal{P}$. Then section 2.2 describes the conversion between this helper pointcloud structure and the main data structure used for representing obstacles, the triangular mesh $\mathcal{M}$. Section 2.3 explains how the next best target is derived from the mesh. Finally, in section 2.4 is described an unexpected drift error encountered during the testing and how it is tackled and overcome.

Figure 2.2 shows the various steps that compose the obstacle representation. The local pointclouds $\mathcal{P}_{loc}$ are merged into a single global obstacle pointcloud $\mathcal{P}$ (figure 2.2a), then from the global obstacle pointcloud are extracted the vertices of the triangles that will compose the mesh $\mathcal{M}$ (figure 2.2b), finally, the mesh is reconstructed (figure 2.2c shows the edges and figure 2.2b shows the coloured mesh) and the next best target is selected among the candidate targets (figure 2.2c).

2.1. Pointcloud

One of the two data structures used to represent obstacles is a Pointcloud. The Pointcloud is a simple data structure that allows for compactly storing large amounts of data

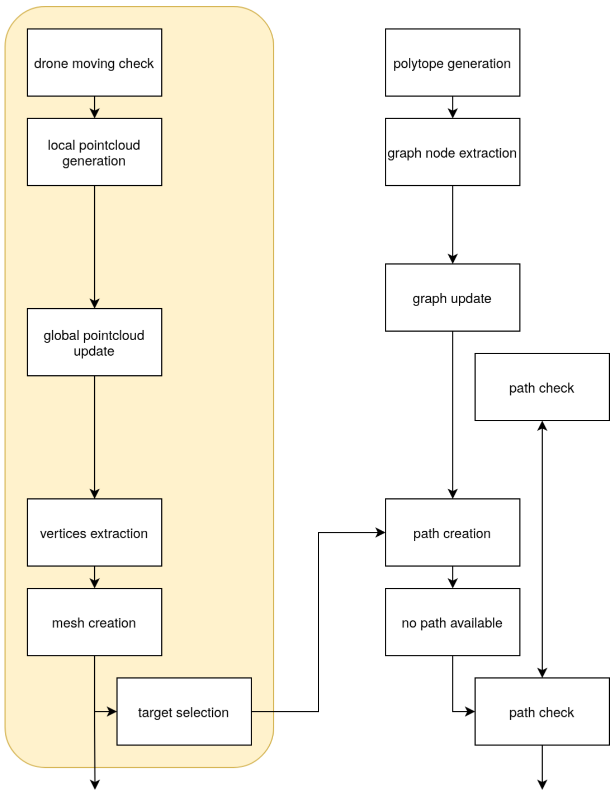


Figure 2.1: Obstacles management and Target selection modules, part of figure 1.3

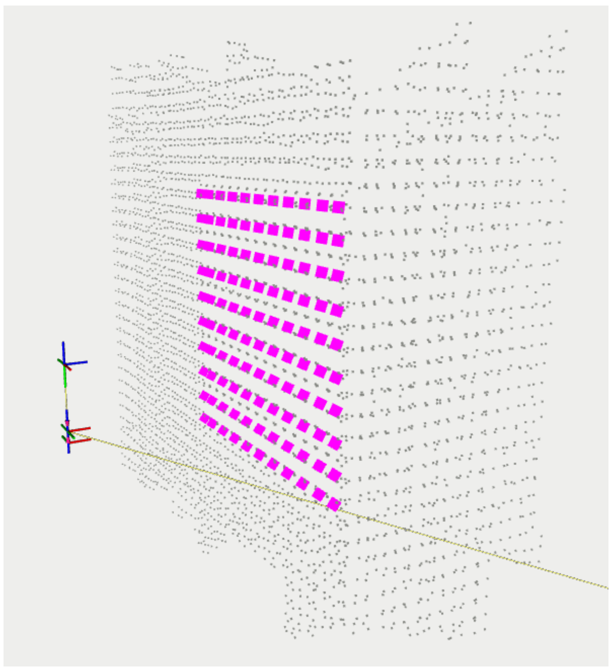
points and to efficiently perform a large number of operations (for example many search operations).

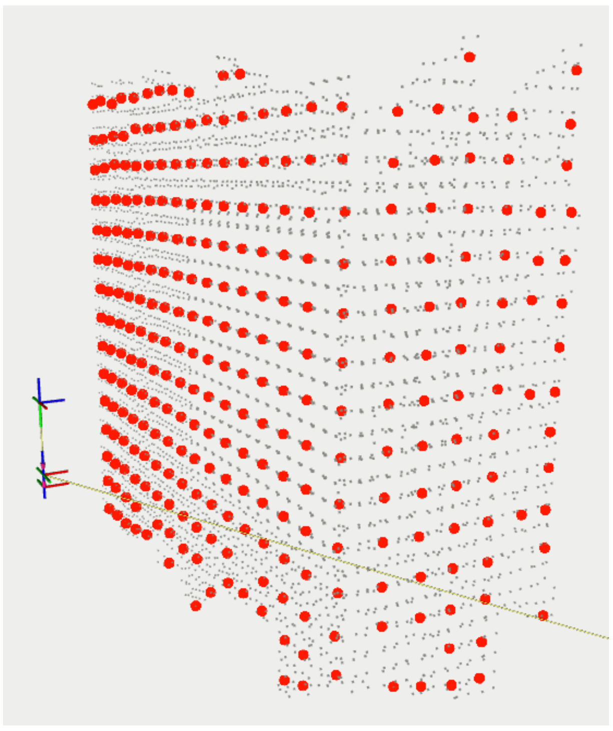
In order to build a reliable global obstacle cloud, some steps are required. From the raw obstacle pointcloud $\mathcal{P}_{raw}$ the local obstacle pointcloud $\mathcal{P}_{loc}$ is generated, and then integrated into a global obstacle pointcloud $\mathcal{P}$.

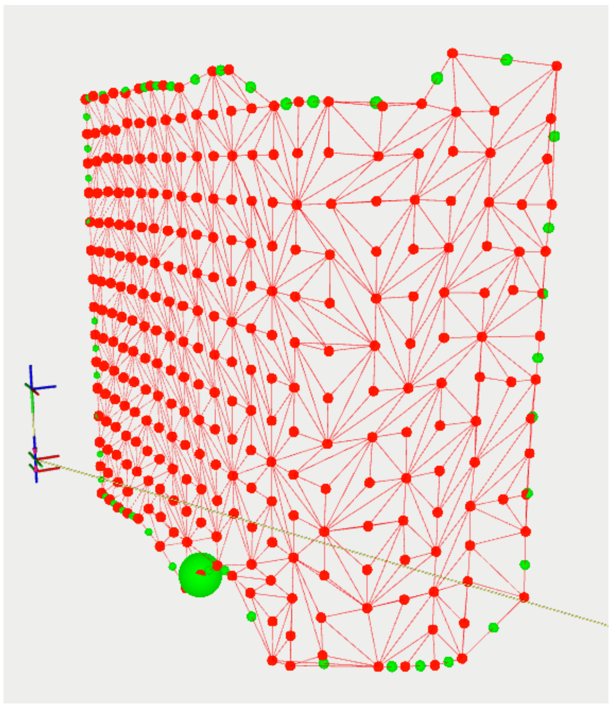
2.1.1. Raw obstacle pointcloud

The raw obstacle pointcloud $\mathcal{P}_{raw}$ is directly generated by the Intel RealSense d435 camera, that provides a stream of 640x480 pixel at 30 frames per second, for a total of 9216000 data point per second.

The raw obstacle pointcloud follows the camera frame (see section 1.2 for an explanation of the coordinate frames), thus is rigidly linked to the drone and aligned with a z-forward, x-right, y-down axis ordering. The maximum depth is 5 meters and the field of view is 87°x58°.


(a) $\mathcal{P}_{loc}$ to $\mathcal{P}$


(b) $\mathcal{P}$ to $\mathcal{M}$ vertices


(c) $\mathcal{M}$ vertices to wireframe, frontier and best target

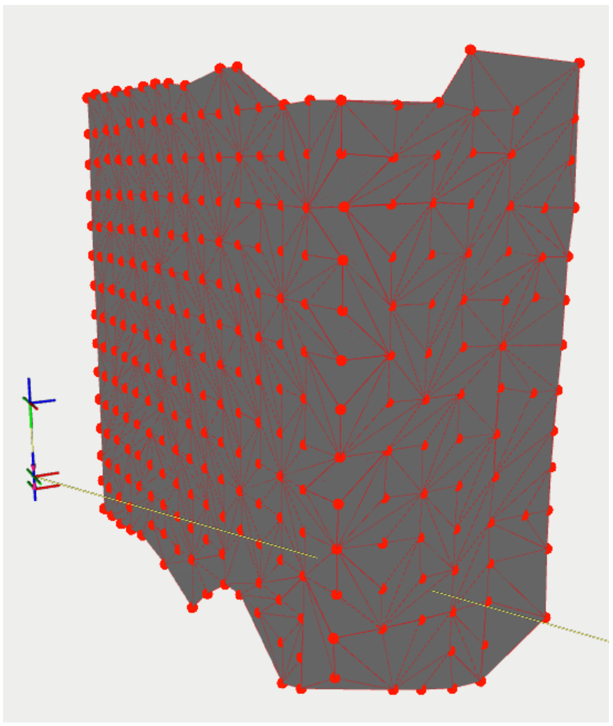

(d) final mesh $\mathcal{M}$

Figure 2.2: Overview of the Obstacle data management

2.1.2. Local obstacle pointcloud

The local obstacle pointcloud $\mathcal{P}_{loc}$ represents the obstacle in front of the camera at the moment the frame is acquired. The pointcloud is aligned with the base link frame (see section 1.2). In order to retrieve the local obstacle pointcloud $\mathcal{P}_{loc}$ from the raw obstacle pointcloud $\mathcal{P}_{raw}$, a few operations are required.

Reduce rate

Due to the relatively slow drone movement, it is not necessary to acquire new raw cloud $\mathcal{P}_{raw}$ too often. A parameter T_{poly} controls how often a raw pointcloud is acquired.

Reduce density

The raw pointcloud density is quite high. This ensures precise measurements even at high distances. However the point density is not useful if the measurement is noisy, and too many data points make the computation much harder. The raw pointcloud $\mathcal{P}_{raw}$ is therefore downsampled to a sparser cloud. A parameter d_{cloud} controls how the cloud is downsampled.

Cloud downsampling The algorithm that reduces a raw pointcloud to a sparse pointcloud works as follow.

A voxel grid of size d_{cloud} is built over the raw pointcloud. For each voxel, all the points present will be approximated with their centroid (see equation (2.1)). Although being slower than downsampling to the voxel centre, this approach is more sensible to the surface structure.

$$\begin{cases} v_i = \dfrac{1}{|V_i|} \displaystyle\sum_{p \in V_i} p \\ v_i = \mathbb{R}^3 \\ p = \mathbb{R}^3 \\ V_i = \{p_1, p_2, \dots, p_n | p_i \in \mathbb{R}^3 \wedge p_i \in \text{voxel i}\} \end{cases} \tag{2.1}$$

Align frames

As explained before, the camera frame does not follow the base link frame axis alignment (x-forward, y-left, z-up), therefore a transformation between the two coordinate frame is required. The transformation is straightforward and shown in equation (2.2).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.2}$$

Local obstacle pointcloud

The local obstacle pointcloud $\mathcal{P}_{local}$ is therefore computed from the raw obstacle pointcloud $\mathcal{P}_{raw}$ by performing a downsampling and frames alignment. In figure 2.3 it is possible to see how the cloud is transformed: the coloured pointcloud is the raw $\mathcal{P}_{raw}$, while the pink pointcloud is the local $\mathcal{P}_{local}$. Note how $\mathcal{P}_{local}$ is correctly aligned and is much sparser than $\mathcal{P}_{raw}$, while keeping the quality loss negligible.

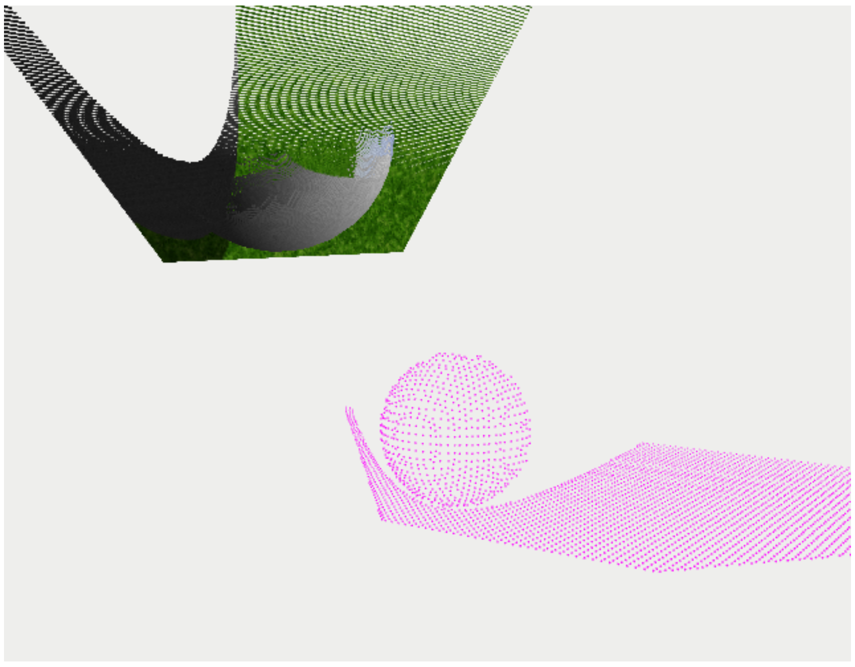


Figure 2.3: Transformations from $\mathcal{P}_{raw}$ to $\mathcal{P}_{local}$

2.1.3. Global obstacle pointcloud

The global obstacle pointcloud $\mathcal{P}$ contains all the obstacles acquired during the flight, and is used for performing all the operations that involve obstacles (aside from those that require a Mesh). The global obstacle pointcloud is centred on the map origin and the axis are oriented as usual.

The pointcloud $\mathcal{P}$ is built by merging multiple instances of the local obstacle pointcloud $\mathcal{P}_{loc}$, and the integration of each one is straightforward.

Map alignment

As described before, the local pointcloud $\mathcal{P}_{loc}$ is centred on the drone's current position, while the global pointcloud $\mathcal{P}$ is centred on the origin of the axis. In order to merge a new local cloud is therefore necessary to transform the points from drone-centred to map-centred. The transformation is a rototranlsation and is provided in equation (2.3), where (p_x, p_y, p_z) is the position of the drone's camera, and α is the yaw angle (i.e. the rotation along the z-axis). Note that the equation provided is simplified, as it does not take into account the roll and pitch. Those additional rotations are however considered during the development of the controller.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{translation} \cdot \underbrace{\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{rotation} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.3}$$

Cloud integration

The construction of the global obstacle pointcloud $\mathcal{P}$ from a series of realigned local obstacle pointclouds $\mathcal{P}_{loc}$ is straightforward (see equation (2.4)): at each iteration the current global obstacle pointcloud is merged with the newly received local obstacle pointcloud, and then the resulting cloud is filtered with the same algorithm explained in equation (2.1). This cloud is the new current global obstacle pointcloud.

$$\begin{cases} \mathcal{P} = filter(\mathcal{P} \cup \mathcal{P}_{loc}) \\ \mathcal{P} \cup \mathcal{P}_{loc} = \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m | a_i \in \mathcal{P} \wedge b_j \in \mathcal{P}_{loc}\} \end{cases} \tag{2.4}$$

Double filtering A legitimate question would be why it was necessary to perform two filtering processes, one between $\mathcal{P}_{raw}$ and $\mathcal{P}_{loc}$, and one from $\mathcal{P}_{loc}$ and $\mathcal{P}$. The reason behind the decision is that the computational effort required must be as small as possible. Accurate tests revealed that performing a single filtering (therefore avoiding $\mathcal{P}_{loc}$, performing all the preprocessing on $\mathcal{P}_{raw}$, and executing equation (2.4) on $\mathcal{P}$ and $\mathcal{P}_{raw}$) is much more demanding than the proposed solution. Moreover, in figure 1.3 it is possible to see that the data acquisition is performed on the Local information node, while the global pointcloud is built on the Global information node. Having a smaller message passing between the nodes is more efficient than sending a huge message, so the use of the local obstacle pointcloud $\mathcal{P}_{loc}$ is again more useful than the use of the raw obstacle pointcloud $\mathcal{P}_{raw}$ only. Finally, the discussed cloud helps manitain an elegant parallel data flow, as visible in figure 1.3.

2.2. Meshing

The Mesh is one of the two data structures used for representing obstacles. A Mesh is a (relatively) complex data structure that contains information about the surface points of the obstacle, the edges connecting the points and the faces of the object (see equation (1.1) for a mathematical definition of a mesh). Although being less efficient than a simple Pointcloud, it is lightweight, easy to handle and visually pleasant.

In order to build a Mesh $\mathcal{M}$ of the obstacles in the map, a few steps are required. From the global obstacle pointcloud $\mathcal{P}$ it is necessary to extract the vertices of the mesh V, compute some parameters that are used for building the mesh, and finally the mesh can be computed.

The mesh construction does not need to be executed too often, as it only produces a visual representation of the obstacles and computes the next target. The parameter that controls how often the mesh is recomputed is T_{mesh}. This parameter should be higher than the raw pointcloud acquisition time ($T_{mesh} > T_{poly}$).

2.2.1. Vertices extraction

The vertices V of the mesh are extracted from the global pointcloud $\mathcal{P}$.

Vertices extraction rationale

In order to successfully create a mesh, it is necessary to take into consideration three concepts: firstly, the sensors' readings could be noisy, and a noisy vertex set could worsen the mesh (or even completely ruin it); secondly, the mesh creation is a computational intensive task [40] that quickly scales up as the number of vertices grows; thirdly, the computed mesh is more similar to the mapped obstacle as the number of vertices increases.

The first two considerations would lead to a sparser vertices density (thus filtering out the noise and reducing the computational demand of the meshing creation), while the third one would lead to a denser vertices density (thus improving the mesh's similarity to the ground truth).

Vertices extraction methodology

The vertices extraction is performed by filtering the global obstacle pointcloud $\mathcal{P}$ with the algorithm already described in equation (2.1) and a voxel size controlled by the parameter $d_{vertices}$ (resulted as a tradeoff between the considerations exposed before).

An additional filtering is performed, that ensures that the vertices are not unnecessarily close. Compressing the global obstacle pointcloud to the centroid of the voxels could lead to a number of vertices too close to each other, thus increasing the mesh computation load without benefit. The algorithm used for filtering the cloud is presented in algorithm 2.1. From an already filtered cloud, for each point, all the points too close are marked and deleted. The parameter that controls the filtering range is derived from $d_{vertices}$, and ensures that no point-to-point distance is smaller than $d_{vertices} \cdot 3/4$.

Algorithm 2.1 Second filtering

```
 1: { $\mathcal{P} \leftarrow$ input pointcloud
    $\mathcal{D} \leftarrow$ points to delete
    $\mathcal{K} \leftarrow$ points to keep
    $radius = d_{vertices} * 3/4 \leftarrow$ minimum distance between points }

 2: for $p \in \mathcal{P}$ do
 3:     if $p \in \mathcal{D}$ then
 4:         continue
 5:     end if
 6:     $\mathcal{K} = \mathcal{K} \cup p$
 7:     $\mathcal{C} = \text{find close}(\mathcal{P}, p, radius)$
 8:     for $c \in \mathcal{C}$ do
 9:         if $c \notin \mathcal{K}$ then
10:             $\mathcal{D} = \mathcal{D} \cup c$
11:         end if
12:     end for
13: end for
14: for $d \in \mathcal{D}$ do
15:     delete$(P, d)$
16: end for

17: return $\mathcal{K}$
```

2.2.2. Mesh creation

The Mesh $\mathcal{M}$ is built from the vertices V computed as shown in the previous section, which are in turn extracted from the global obstacle pointcloud $\mathcal{P}$. The research on meshing algorithm is well developed, various solutions exist, and many well known algorithms provide a robust and reliable reconstruction [3, 5, 33]. Therefore we decided not to develop a new meshing algorithm, and take advantage of the already existing techniques, with a focus on fast real time reconstruction algorithms. The chosen Mesh reconstruction technique is the Greedy Projection Triangulation [27].

The Greedy Projection Triangulation algorithm (GP3) is a fast surface reconstruction that can parse large noisy datasets. The main features of the GP3 algorithm are: near real time performance, noise robustness and memory efficiency. The algorithm maintains a list of points from which the mesh can be grown (“fringe” points) and extends it until all possible points are connected. In algorithm 2.2 is shown how the Greedy Projection Triangulation technique works. Starting from the vertices pointcloud, select a point, then search for the k-NN and project them into an approximately tangential plane. After filtering points by visibility connect all the visible points to the current.

The meshing algorithm returns a list of triangles that compose the faces of the mesh.

The reconstructed Mesh accurately reflects the features of the global obstacle pointcloud. The results are optimal if the mapped object is thick enough and is regular enough, otherwise, the algorithm may fail in the k-NN and plane detection phases, and this leads to a wrong reconstructed mesh. This problem is clearly visible in figure 4.3, where short and slim columns, a thin rooftop and steep angle variations produce an imprecise reconstruction.

2.3. Target selection

In order to be able to completely map the scene, it is necessary to find a way to understand where the information gained is sufficient and where it is necessary to continue the mapping.

An important phase in the controller flow is therefore the computation of the next target to be reached. The exploration time should be as short as possible, so the next target must be designed as a tradeoff between the information acquired and the space travelled.

As explained in the State of the art review, two main mapping philosophies exist: the mapping process could try to map the entire environment or focus on the obstacles found.

Algorithm 2.2 Greedy Projection Triangulation

```
1: { $\mathcal{P} \leftarrow$ input pointcloud representing vertices
   $\mathcal{N} \leftarrow$ neighbour points of a given point
   $\mathcal{T} \leftarrow$ triangles set, each triangle being a triplet of points }

2: for $p \in \mathcal{P}$ do
3:     $\mathcal{N}$ = k-NN$(\mathcal{P}, p, radius)$
4:     $plane$ = tangent plane$(p, \mathcal{N})$
5:     for $n \in \mathcal{N}$ do
6:         if not is visible$(n, plane)$ then
7:             $\mathcal{N} = \mathcal{N} \setminus n$
8:         end if
9:     end for
10:    order neighbour around $p$
11:    for $n \in \mathcal{N}$ do
12:        $t = (n, n+1, p)$
13:        if satisfy condition$(t)$ then
14:            $\mathcal{T} = \mathcal{T} \cup t$
15:        end if
16:    end for
17: end for
18: return $\mathcal{T}$
```

Since the controller's objective is to map a given building, the adopted idea is the second one. Given these premises, the selected data structure for the next target computation is the Mesh $\mathcal{M}$ built over the global obstacle pointcloud. The idea behind the process is to extract the frontiers $\mathcal{F}$ from the Mesh, parse the frontiers and extract the holes $\mathcal{H}$, and then find the next target by analysing the holes and finding the area where the density of points is higher. A mapping session is considered complete once there are no holes.

2.3.1. Frontier detection and holes computation

Frontier detection

The frontier set $\mathcal{F}$ is defined as a set of Mesh's edges, with the requirement that an edge is in the frontier set if and only if only one and only one triangle face contains that edge (in equation (2.5) the frontier set $\mathcal{F}$ is built from the mesh $\mathcal{M}$, and the symbol $\exists_{=1}$ means "one and only one").

$$\begin{aligned}\mathcal{F} = \{e_1, e_2, \ldots, e_n \quad | \quad & e_i \in \mathcal{M}.E \\ & \wedge \exists_{=1}\ f \in \mathcal{M}.F \mid e_i \in f\}\end{aligned} \tag{2.5}$$

Determining the frontier set allows to find the locations where the Mesh is incomplete, and therefore requires further inspection. Note that in order to reduce the size of the frontier set to zero it is necessary for the building to be completely reachable (i.e. some weird geometries may lead to endlessly trying to reach some unreachable frontiers.

Holes computation

The hole set $\mathcal{H}$ is defined as a set of lists of edges, with the requirement that the edges are frontier elements and that each edge list is closed (in equation (2.6) the hole set $\mathcal{H}$ is computed from the frontier set $\mathcal{F}$). The resulting structure highlights the contiguous structure of frontiers and helps to understand how the frontiers are distributed.

Additionally, it is possible to use the information about holes by filtering out those holes that do not require further mapping. This could be useful for example if a real time mesh is not required, and the controller only needs to ensure that the pointcloud is dense enough. An example of a filtering procedure (tested and later on discarded) is shown in algorithm 2.3.

$$\begin{aligned}\mathcal{H} = \{e_1, e_2, \ldots, e_n \quad | \quad & e_i \in \mathcal{F} \\ & \wedge \quad e_i.v_2 = e_{i+1}.v_1 \quad \forall i \in [1, n-1] \\ & \wedge \quad e_n.v_2 = e_1.v_1\}\end{aligned} \tag{2.6}$$

2.3.2. Next target

The Next target module takes care of computing the optimal location that the drone should film. The selection of the next target is crucial in minimising the total flight time, as picking up the wrong location would lead to a longer exploration.

The procedure that computes the next target is composed of two phases: target evaluation and target selection. The first phase takes care of selecting all the possible candidate best targets, while the second phase analyses the candidates and selects the most suitable one.

Target evaluation

The target evaluation phase takes as input the holes $\mathcal{H}$ (optionally filtered) and produces a set of candidate targets. Each candidate target has associated a metric that evaluates the "information gain" of the target. The "information gain" basically defines the reduction of uncertainty that comes after a measurement [24].

Algorithm 2.3 Holes filtering

```
1: { $\mathcal{H}$ ← input holes set, composed of a closed list of edges
   $\mathcal{H}'$ ← filtered holes }

2: $\mathcal{H}' = \{\}$
3: for $h \in \mathcal{H}$ do
4:    $keep =$ true
5:    {ignore a hole if it is composed of a single triangle}
6:    $keep = \text{len}(h) < 4$
7:    $centroid = \text{centroid}(h)$
8:    {ignore a hole if all the edges are close enough to the hole's centroid}
9:    for $f \in h$ do
10:       if $\text{dist}(f, centorid) > max\ dist$ then
11:          $keep =$ false
12:       end if
13:    end for
14:    if $keep =$ true then
15:       $\mathcal{H}' = \mathcal{H}' \cup h$
16:    end if
17: end for

18: return $\mathcal{H}'$
```

The candidate target points are the vertices of the edges that compose the holes, while the information gain metric is the number of holes' edges near the candidate target. In algorithm 2.4 is shown the procedure that computes the information gain for each candidate target. The *radius* parameter in the algorithm is used to denote the limited field of view, in the sense that once the camera points to a candidate target, a circle of *radius* meters radius is surely visible, and therefore any edge inside that circle will be mapped. The parameter *radius* is referred in the controller by the parameter d_{target}.

Target selection

Given a list of candidate targets, and an information gain for each one, the target selection phase must decide which candidate target should be filmed next.

There are two main approaches for selecting the target.

Exploration The "exploration" approach selects the target only based on the information gain of the candidate targets. This causes the drone to choose always the most informative candidate target, regardless of the distance it has to travel. The exploration approach, therefore, tries to maximise the information acquired, without taking into consideration the time spent travelling through the map.

Exploitation The "exploitation" approach, on the other hand, selects the target only based on the distance between the candidate targets and the drone position, therefore choosing always the closest candidate target, regardless of its information gain. The exploitation approach thus tries to minimise the travelled distance.

A more interesting and robust approach is a mixture of the two. The target is selected by considering both the Exploration and Exploitation factors, and mixing them in order to compute a reasonable measurement for each candidate target. The target with the best score is selected as the next target.

A simple Exploration-Exploitation tradeoff is adopted in the proposed controller: the score of each candidate target is computed by multiplying the information gain with the reciprocal of the distance, elevated to the nth power (equation (2.7) shows the computation of the score for a target as an exploration/exploitation tradeoff). The candidate target with the highest score is then chosen as the next target. The distance adopted is the euclidean distance, which is less precise but much easier and faster than graph distances. The parameter that controls the balance between information gain and distance is *exp expl*.

$$score(t) = \frac{information\ gain(t)}{dist(t, drone)^{exp\ expl}} \tag{2.7}$$

It is optionally possible to modify the score computation in a way that takes into account the altitude difference between the drone and the candidate target. This is useful in tight environments, where staying at the current height is easier than changing altitude, so choosing a target at the same height as the drone is preferred. The modified score has an additional element, that is the multiplication with a "discouraging factor" proportional to the altitude difference. In equation (2.8) can be seen two examples of discouraging factors, a simpler one polynomially proportional to the height difference, and a more complex one that uses the indicator function $\mathbb{1}$.

$$\begin{cases} score = score \cdot \dfrac{1}{(|target_z - drone_z|)^k} \\ score = score \cdot \dfrac{1}{k_1 \cdot \mathbb{1}_{|target\ z - drone\ z| > k_2}} \end{cases} \tag{2.8}$$

Algorithm 2.4 Information gain computation

```
1: { $\mathcal{H} \leftarrow$ input holes set, composed of a closed list of edges }

2: $\mathcal{H}' = \{\}$
3: for $h \in \mathcal{H}$ do
4:     for $f \in h$ do
5:         $gain =$ num of neighbour$(f, \mathcal{H}, radius)$
6:     end for
7: end for
```

2.4. Pointcloud drift

The obstacle drift is a problem that was encountered during the testing of the controller. After a detailed investigation, the cause of the error is still not entirely known, and the problem has been identified to be mainly present when there are quick movements of translation and rotation.

The obstacle drift error causes the local obstacle pointcloud $\mathcal{P}_{loc}$ to be not exactly aligned with the ground truth model. Once multiple misaligned local pointclouds are merged into the global obstacle pointcloud $\mathcal{P}$, the obstacle map becomes too noisy and useless. In figure 2.4 it is visible how the obstacle drift ruins the global obstacle pointcloud and how the proposed solution completely solves the drift.

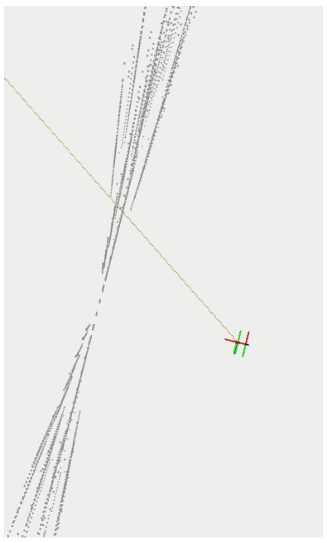
The problem is tackled by estimating the speed of the drone, and discarding the local pointcloud $\mathcal{P}_{loc}$ if the movements are too intense. A drawback of this approach is that it is necessary to ensure that periodically the drone stops moving, in order to accept the local pointcloud and update the global structure.

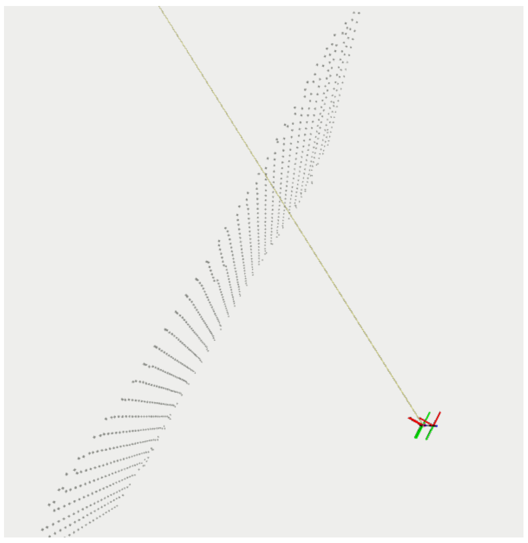
Due to the modularity of the controller, a two-phase solution has been adopted: the speed check part is addressed here in section 2.4, while the part that tackles the periodic stop is addressed in section 3.4.5. As for the scheme shown in figure 1.3, the speed check composes the "drone moving check" module of the local information node, and the periodic stop is contained in the trigger events of the "path creation" module in the path planning node.

2.4.1. Speed check

In order to avoid the obstacle pointcloud update while the drone is moving, the instantaneous linear and rotational speed are computed at each iteration the $\mathcal{P}_{loc}$ is expected to be read (i.e. every T_{poly} seconds).

The linear and rotational speed are computed by averaging the position and yaw variation


(a) Obstacle drift


(b) Obstacle drift solved

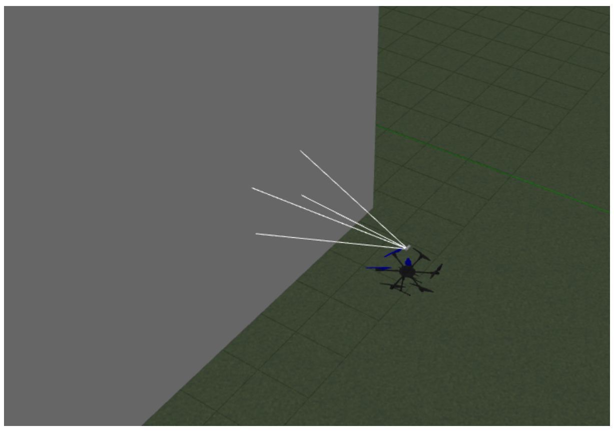

(c) Ground truth

Figure 2.4: Obstacle pointcloud drift

over a small time interval $dt = 0.1s$ (the computation is shown in equation (2.9)).

$$\begin{cases} v_x = \dfrac{pos_x - pos_{old\ x}}{dt} \\ v_y = \dfrac{pos_y - pos_{old\ y}}{dt} \\ v_z = \dfrac{pos_z - pos_{old\ z}}{dt} \\ linear\ speed = \sqrt{v_x^2 + v_y^2 + v_z^2} \\ \\ rotation\ speed = \dfrac{yaw - yaw_{old}}{dt} \end{cases} \tag{2.9}$$

If any of the two speeds is higher than the maximum allowed value the pointcloud is discarded and no further operations are performed. The maximum linear speed parameter is controlled by $v_{lin\ max}$, and the maximum rotational speed parameter is controlled by $v_{rot\ max}$.

2.4.2. Triggers for reading the pointcloud

An additional element is useful (although not necessary) for speeding up the exploration. Since the local obstacle pointcloud $\mathcal{P}_{loc}$ is updated only when the drone is not moving, and the reading of a new $\mathcal{P}_{loc}$ is a slow process (T_{poly} can be as long as some seconds) it is useful to setup a mechanism that quickly reacts to the drone's change of speed. This module should rapidly trigger a new local pointcloud reading once the drone stops, while keeping a low computational load on any other situation.

The solution adopted is shown in algorithm 2.5. Two timers are set up: a slow one that runs every T_{poly} seconds, and triggers the pointcloud processing if the drone is not moving; and a quicker one that triggers the pointcloud processing if the drone has just stopped moving.

Algorithm 2.5 Accepting new cloud process

```
 1: { standard timer ← timer that runs every $T_{poly}$ seconds
    fast timer ← timer that runs at higher rate }

 2: upon receiving a new $\mathcal{P}_{raw}$
 3: if not standard timer and not fast timer then
 4:     skip iteration
 5: end if
 6: $moving\ drone$ = check moving()
 7: if $standard\ timer \land \neg moving\ drone$ or
    $fast\ timer \land was\ moving \land \neg moving\ drone$ then
 8:     parse cloud
 9:     ...
10: else
11:     skip iteration
12: end if
```

2.4.3. Graph management and pointcloud drift

The pointcloud drift problem presented in this section is tackled only for the update of the global obstacle pointcloud $\mathcal{P}$, and not for the graph update. This means that the graph is updated even if the drone is moving, so even if the pointcloud could be misaligned with respect to the ground truth. The reason for this choice is that this allows to speed up the exploration, as the graph is constantly updated and new nodes are added while the drone is moving. Moreover, while the drift is big enough to cause errors in the obstacle representation, it does not cause any trouble in the structure of the graph, as errors of a few dozens of centimetres can be sustained without worries.

3 | Graph management and Navigation

The Graph management and Navigation module takes care of building and maintaining a data structure that represents the free space of the environment, and planning a safe path in order to reach the position target produced by the Obstacle module (as described in section 2.3). Together with the Obstacles management and the Target selection functionalities, this module completes the controller.

Referring to the global controller structure shown in figure 1.3, the current chapter covers the modules on the right, as in figure 3.1. The correspondence between the sections of this chapter and the referred module is quite straightforward, except for the pointcloud drift section (section 3.4.5) that is addressed in the enabling conditions of the "path creation" module, and the "path check" modules that are explained in the obstacle avoidance section (section 3.5).

In section 3.1 it is explained how the obstacle-free subspace (used for selecting candidate nodes to be inserted into the navigation graph) is computed. In section 3.2 is shown the extraction of the candidate nodes, and in section 3.3 is described the update of the navigation graph. The navigation approach is described in section 3.4, and a technique for building an obstacle-free path is shown in section 3.5.

3.1. Polytope generation

In order to construct a reliable graph, it is necessary to ensure that all the nodes and edges are in a safe position. The use of a convex polytope ensures not only that all the graph nodes that are generated at each iteration are far enough from the obstacles, but also that all the graph edges that connect the new nodes are far enough from the obstacles.

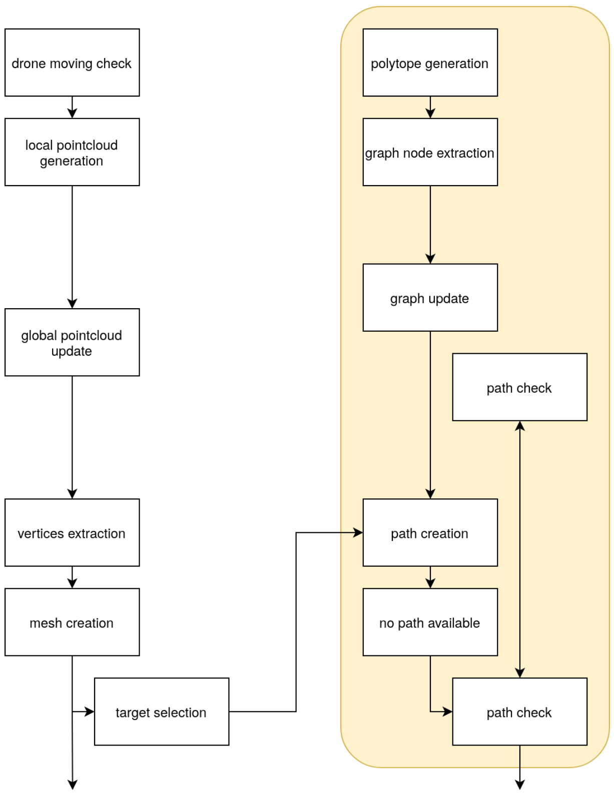


Figure 3.1: Graph management and Navigation modules, part of figure 1.3

3.1.1. Convex polytope

A **polytope** is defined as a geometric object with flat faces. The term refers to a generic n-dimensional object, while a three-dimensional polytope is called a polyhedron.

More formally, a **three-dimensional polytope** can be defined as a solid whose boundary is composed of a number of planes.

A **convex polytope** is a polytope defined as a solid whose boundary is composed of a number of *half-planes*. An alternative definition of convex polytope relies on the fact that all and only the internal points of the polytope can be expressed as a linear combination of the vertices of the solid [43]. Equation (3.1) and equation (3.2) show the two equivalent definitions of convex polytopes.

$$\begin{bmatrix} a_{1x} & a_{1y} & a_{1z} \\ a_{2x} & a_{2y} & a_{2z} \\ & \vdots & \\ a_{nx} & a_{ny} & a_{nz} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \leq \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{3.1}$$

$$\begin{cases} \begin{bmatrix} v1_x & v2_x & \cdots & vN_x \\ v1_y & v2_y & \cdots & vN_y \\ v1_z & v2_z & \cdots & vN_z \end{bmatrix} \cdot \begin{bmatrix} k1 \\ k2 \\ \vdots \\ kN \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \\ k1 + k2 + \cdots + kN = 1 \\ k1, k2, \ldots, kN \geq 0 \end{cases} \tag{3.2}$$

An interesting propriety of convex polytopes is that, for every pair of points inside the polytope, the entire segment connecting the two points lies inside the solid (see equation (3.3)). This fact helps understand why convex polytopes are useful in the navigation process: by forcing the drone to always navigate inside a convex polytope (i.e. all the nodes of the graph are extracted from inside the polytope, and only the nodes inside the polytope are connected by edges), it is ensured that the drone will remain inside the safe area and thus will not interact with obstacles.

$$\forall p_1, p_2 \in CH, \ \forall k \in [0,1] \ , \ k \cdot p_1 + (1-k) \cdot p_2 \in CH \tag{3.3}$$

Point inside convex polytope

A crucial operation that involves the convex polytope is the check if a point is inside the polytope.

two-dimensional convex polytope For a two-dimensional convex polytope (polygon) one example of simple and efficient approach is the "cross product rule" (the approach adopted in the G-BEAM controller [7]), that states that a point is inside the polygon if and only if the cross products between the point and each edge have the same sign. An example of this approach can be seen in figure 3.2. Given the black polytope, the edges are ordered and the vectors are found. For the point inside the polytope, all the cross products between the vectors connecting to a vertex and the edge starting from the vertex have the same sign (all blue), therefore the point is inside. For the point outside the polytope, the cross products do not have the same direction (two blue, two orange), therefore the point is outside the hull.

three-dimensional convex polytope However in a three-dimensional convex polytope it is more difficult to check whether a point is inside the polytope. Two main approaches exist: by using the inequality definition of the polytope equation (3.1) it is

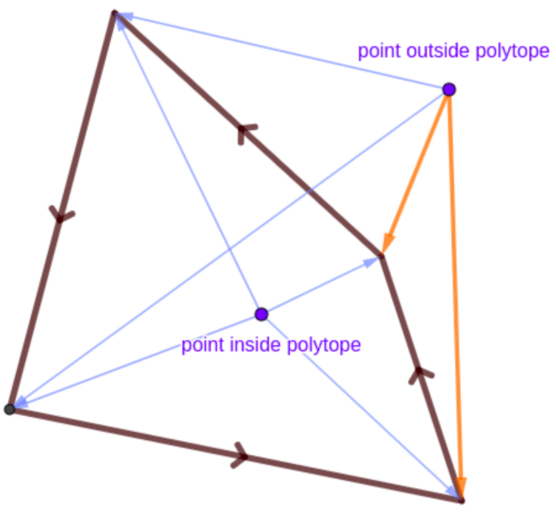


Figure 3.2: Cross product rule for point inside hull

necessary to verify that the given point satisfies all the inequalities; while by using the vertices definition equation (3.2) it is necessary to verify that the given point can be expressed as a linear combination of the polytope vertices.

Since it is easier to generate the polytope by storing its vertices, and converting the polytope from the vertices definition and inequalities definition is not trivial [43], the second approach has been adopted.

3.1.2. Convex polytope generation

The proposed controller requires finding an "obstacle free" area in front of the camera, in order to generate some potential nodes that will be integrated into the graph. The area should be as big as possible, far enough from the obstacles, and all the paths inside the area should be in a safe position. Therefore a convex polytope is required.

The selected shape is a "quasi pyramid" generated iteratively. The maximum right pyramid is constructed, and then shrunk until all the obstacles points are outside the polytope. The algorithm is shown in algorithm 3.1, and here follows a detailed explanation of its main characteristics.

The polytope generation algorithm takes as input the local obstacle pointcloud $\mathcal{P}_{loc}$ (described in section 2.1). Moreover, it requires some information on the camera range and field of view: *max depth* measures the maximum depth required for the polytope (note that *max depth* can be smaller than the maximum range of the camera), dx and dy are the half field of view on the horizontal and vertical axis. Finally, *sector size* denotes the required depth of each sector, and *delta* indicates the shrinking amount at each iteration. The controllable parameters are *poly_max_depth* (that controls *max depth*) and *poly_delta* (that controls *delta*).

Algorithm description and analysis

L 2 a standard right pyramid is created, with the maximum edge size

L 4 - 16 preprocess of the input cloud

- L 5 - 7 the sector of the point, the quadrant and the distance to the polytope edge is computed
- L 8 if the sector is greater than the maximum number of sectors (meaning that the point is too far), skip the point
- L 9 if the point is farther than the current quadrant max distance, skip the point
- L 11 - 14 if the point is close enough to the edge, trigger the optimisation, and set the point's sector as the maximum distance for the quadrant
- L 15 assign the point to its sector

L 17 parse each sector, from the farthest, starting from the farthest sector possible

L 18 - 19 parse all the points, from the closer to the edge to the last

L 20 - 21 continue shrinking the polytope until the point is outside the solid. In order to check if the point is inside the polytope the technique exposed in section 3.1.1 is adopted.

The pre-shrinkage of the polytope (lines 11 to 14 of the algorithm) states that if an obstacle is close to an edge, then the polytope will surely exclude be closer on that quadrant. This allows saving time, since some points get automatically discarded (those farther than the current closest).

By ordering the points from the closest to the edge (line 18) we ensure that the polytope gets downsized as little as possible: the closest points will necessarily need the close quadrant to shrink, while farther points cloud be excluded by other points (see for example figure 3.3e, where the last white point gets excluded while the red one is selected).

The algorithm can be fine-tuned by modifying the sector size and the delta factor. A higher sector size leads to a smaller volume loss (due to the pre-shrinkage optimization), while a small value creates lesser bucks and reduces the computational overhead. A higher delta size speeds up the computation since fewer iterations are required, at the cost of a higher volume loss; while a lower delta size returns a more precise output at the cost of more iteration and cost.

Advantages and disadvantages of the proposed convex polytope generation

The proposed convex polytope generation approach has a number of advantages with respect to the basic right pyramid, and a few disadvantages.

Advantages The algorithm is quite efficient, it guarantees an upper limit to the number of iterations (since the number of points inside the polytope strictly decrease), and returns an excellent volume if the obstacles are not perpendicular to the camera view.

Disadvantages The greatest disadvantage of the proposed algorithm is that its performance is much lower than the right pyramid if the camera view is perpendicular to the obstacles, while the computed volume is very similar.

Considering that the camera view is tilted with respect to the obstacles most of the time, the proposed algorithm is clearly superior and more efficient.

3.1.3. Polytope generation example

A simple example of the algorithm execution in a two-dimensional example is shown.

Figure 3.3a shows the drone camera (yellow), the obstacle pointcloud (white dots), the frontier separation (white full line), maximum range (blue full line), and the quadrant separation (white dashed line)

Figure 3.3b and figure 3.3c show the pre-shrinkage phase: a point is detected to be close enough to the edge and the polytope is reduced on the corresponding quadrant.

Figure 3.3d and figure 3.3e show the process of point extraction. From the last sector, ordering the points from the closest to the edge, the polytope is shrunk. Some points (for example the white one in figure 3.3e) are not parsed.

Figure 3.3f and figure 3.3g show the final result for the proposed approach and for the right pyramid simple case. The proposed approach covers a higher volume and is therefore preferable.

Figure 3.4 shows a point distribution in which the proposed approach and the right pyramid approach have similar results. However, this situation is very rare during the

Algorithm 3.1 Convex polytope generation

```
 1: { $\mathcal{P}_loc$ ← input local obstacle pointcloud
    $sector\ size$ ← length of each sector
    $delta$ ← shrink factor }

 2: generate right pyramid, vertex on origin, base at angle $dx, dy$ with length $max\ depth$
 3: $num\ sectors$ = (int)$max\ depth/sector\ size$
 4: for $p \in \mathcal{P}_{loc}$ do
 5:   $sector$ = (int)$p_z/sector\ size$
 6:   $quadrant$ = find closer edge($p$)
 7:   $distance\ edge$ = find distance to edge($p, quadrant$)
 8:   if $sector \geq num\ sectors$ or
      $sector > far\ quadrant[quadrant]$ then
 9:     skip point
10:   end if
11:   if $distance\ edge < min\ distance\ edge$ then
12:     $far\ quadrant[quadrant] = sector$
13:     shrink polytope such that edge reach $sector$
14:   end if
15:   insert point $p$ in the corresponding $sector$ point list
16: end for
17: for $sector \in sectors.reverse$ from sector num max $quadrant$ do
18:   sort $sector.points$ by distance wrt $distance\ edge$
19:   for $p \in sector.points$ do
20:     while inside polytope($p$) do
21:       shrink polytope on edge $quadrant$ by distance $delta$
22:     end while
23:   end for
24: end for

25: return polytope
```

flight. It is possible to notice that the pre-shrinkage optimization causes a lot of points to be excluded from the parsing, with a resulting higher performance.

3.1.4. Polytope generation timing

Similarly to the local obstacle pointcloud $\mathcal{P}_{loc}$ (as explained in section 2.1), the polytope generation is executed any time a new local obstacle pointcloud $\mathcal{P}_{loc}$ is available, so either following the standard timer that runs every T_{poly} seconds, or following the fast timer that runs at a higher rate and activates once the drone stops moving (as explained in section 2.4).

3.2. Graph node extraction

After the generation of the convex polytope, it is necessary to extract some points that will be considered as candidate nodes and inserted into the global graph.

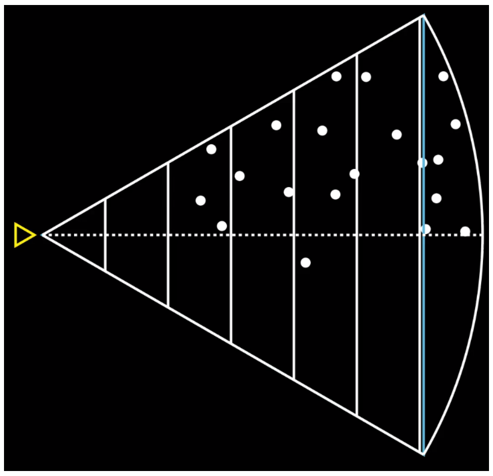
The number and position of the extracted candidate points have to be carefully evaluated: too many nodes would lead to a needlessly dense graph, with a high rejection rate and complex path planning; while too few nodes would lead to a too sparse graph, and consequent difficulty to plan a path that reaches far points.

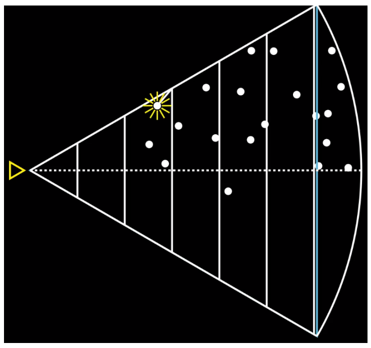
Another issue that needs to be considered is the fact that the space is not "isotropic", in the sense that moving along the xy plane is easier than changing height. This is due both to the camera location and orientation (in front of the drone, facing forward) and to the dynamic of the drone. Therefore it is necessary to compress the points on the z-axis, in such a way that the vertical navigation is simplified.

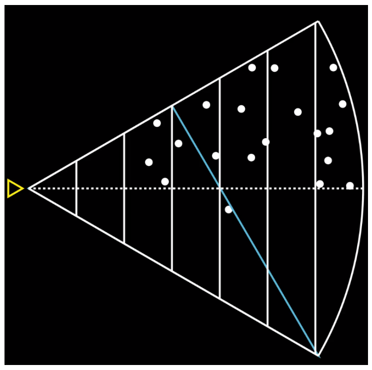
The simplest node extraction solution, selecting the vertices of the convex polytope, does not satisfy either of the issues just presented: the number of points is too low and the distribution does not take into consideration the z-axis requirement.

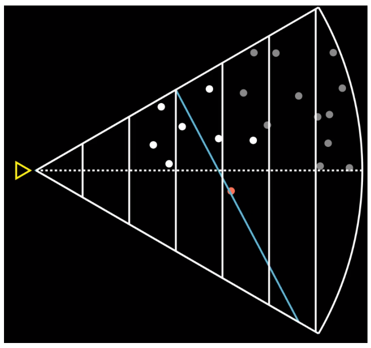
A safer and better solution consists of selecting both the polytope vertices and a few points from inside the polytope. The additional points are selected both from the polytope edges and from the polytope interior. Those points are extracted from a grid with a forward distance of $poly\_dist\_front$, and a vertical distance of $poly\_dist\_vertical$ (with an isotropic drone $poly\_dist\_front$ should be equal to $poly\_dist\_vertical$, for the proposed controller it is more reasonable if $poly\_dist\_front > poly\_dist\_vertical$).

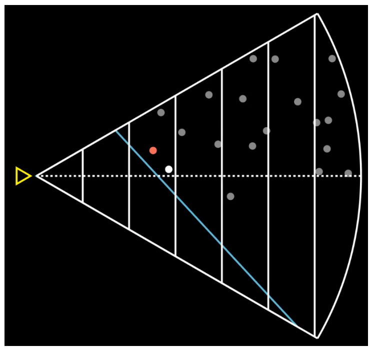
In figure 3.5 it is possible to see the convex polytope generated and the candidate nodes extracted from the polytope. Without obstacles the maximum number of points are

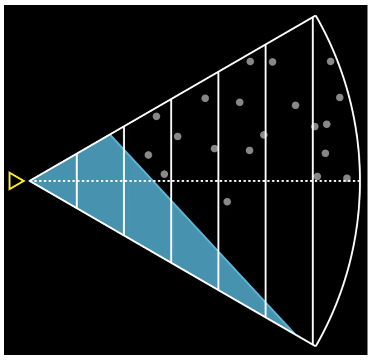

(a) Algorithm startup


(b) Pre-shrinkage optimization


(c) After pre-shrinkage


(d) First shrinkage


(e) Last shrinkage


(f) Final polytope

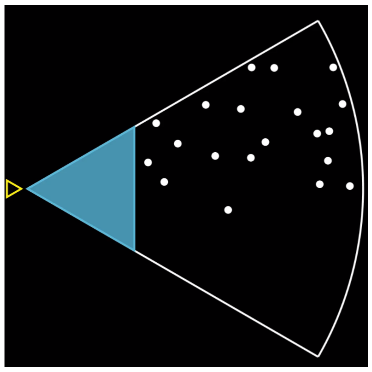

(g) Right pyramid polytope

Figure 3.3: Convex polytope generation example

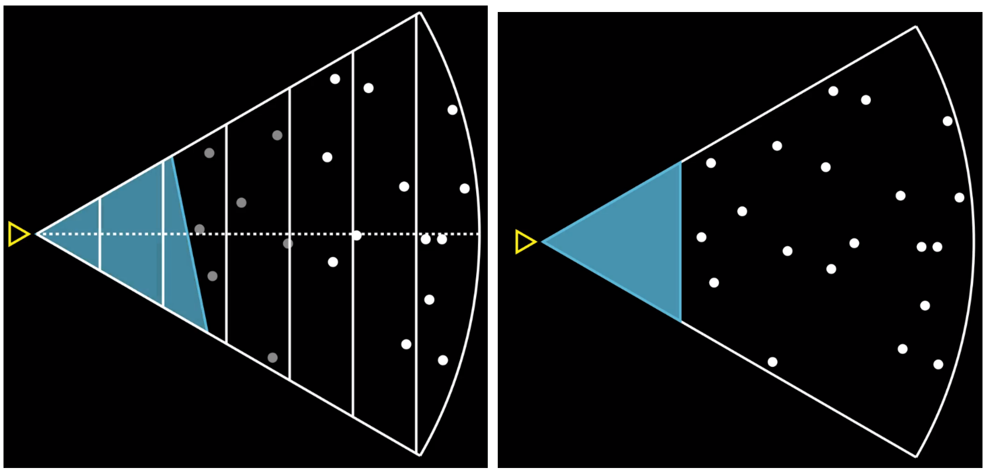

(a) Final polytope (b) Right pyramid polytope

Figure 3.4: Polytope generation result on another point distribution

extracted, guaranteeing the maximum distance between the nodes and an optimal vertical spacing. If there are obstacles, the convex polytope is smaller and the number of points is lower.

In the second image, it is clear the advantage of the proposed convex polytope generation (presented in section 3.1), as the right pyramid would be as small as the shortest edge, and therefore the number of candidate nodes would be much lower, awfully slowing down the exploration.

3.3. Graph integration

The main structure that holds data for navigation is the Navigation Graph $\mathcal{G}$. The Graph contains information about visitable nodes and walkable edges and is in charge of guaranteeing that any path that traverses the graph is valid and safe.

Similarly to the global obstacle pointcloud $\mathcal{P}$ generation, the navigation graph $\mathcal{G}$ is built iteratively every time a new set of candidate nodes is received. In order to merge a set of candidate nodes into the graph $\mathcal{G}$ some operations are required.

3.3.1. Graph node insertion

Candidate nodes alignment

In order to evaluate the candidate nodes, it is necessary to align the reference frame of the candidate nodes (centred at the drone position, with the x-axis aligned with the camera)

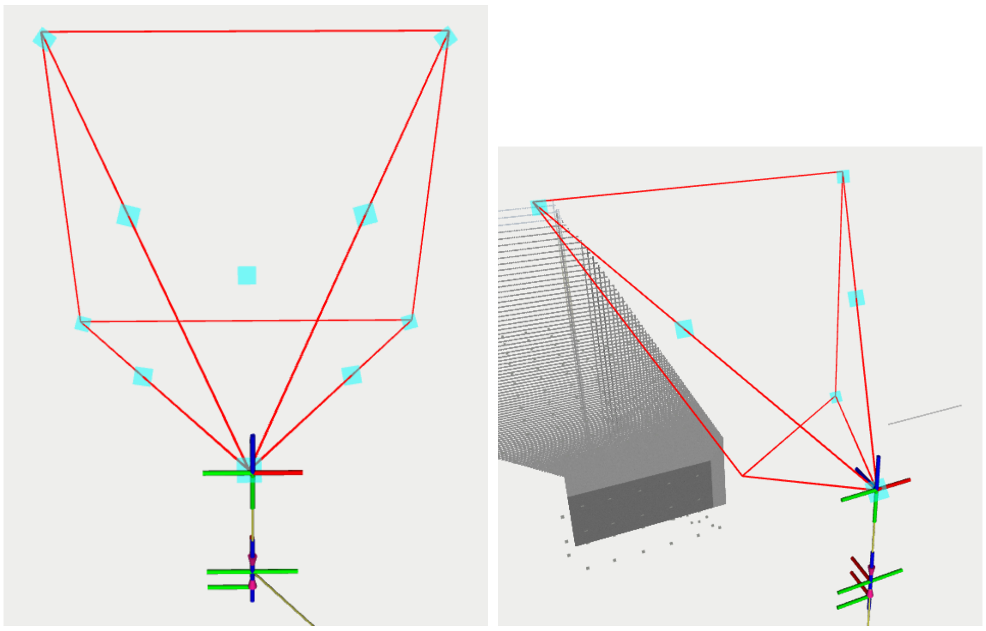

(a) Node extraction without obstacles (b) Node extraction with obstacles

Figure 3.5: Point extraction from convex polytope

to the global reference frame (centred at the origin, with the usual axis alignment).

The transformation that converts from the local frame to the global frame is the rototranslation shown in equation (2.3), where x, y, z indicates the drone position and α indicates the yaw, so the rotation along the vertical axis.

Candidate node filtering

In order to avoid getting a too dense graph, it is necessary to filter the candidate nodes and insert in the graph only the candidate nodes that are distant enough from the closest existing node. However, as stated in section 3.3, the z-axis needs to be "compressed" in order to simplify the vertical navigation.

Therefore, each candidate node will be excluded if the distance to the closest node is closer than *d min*, *but* the height difference must not be farther away than d_z *min*. The parameters that control the values are *graph d min* (for *d min*) and *graph* d_z *min* (d_z *min*). Figure 3.6 is a graphical representation of the filtering process, where the coloured area is the zone that rejects candidate vertices, while all the nodes that are in the white area are inserted into the graph.

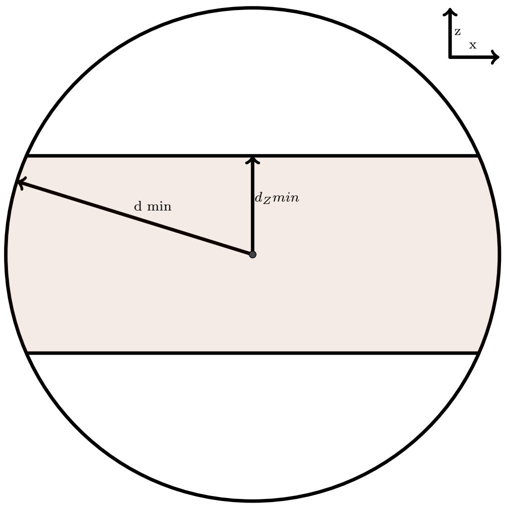


Figure 3.6: Candidate nodes filtering

Candidate node insertion

The remaining candidate nodes can be inserted into the graph, by simply appending the candidate nodes to the node list ($\mathcal{G}.N = \mathcal{G}.N \cup CN$).

3.3.2. Graph edge insertion

After the insertion of the filtered candidate nodes, it is necessary to update the graph $\mathcal{G}$ by adding the edges that correspond to newly discovered connections between nodes.

All the nodes inside the convex polytope, both already existing and newly added, are extracted from the graph (by using the technique already explained in section 3.1.1). Then for each pair of nodes, it is checked whether ad edge already exists and if the edge is set as walkable. If no edge links the two nodes, then the edge is added and set as walkable.

Algorithm 3.2 shows the entire algorithm of graph integration. It contains the candidate node alignment and filtering, and the edge insertion.

3.4. Exploration

The path planning phase is the final component of the controller. It takes as input the navigation graph $\mathcal{G}$ (described in section 3.3) and the next target (described in section 2.3) and is required to produce a path that minimizes the distance between the drone position and the target, while keeping the robot safe and without travelling too much.

Algorithm 3.2 Candidate nodes insertion

```
 1: { $CN \leftarrow$ input candidate graph nodes
     $\mathcal{G} \leftarrow$ navigation graph
     polytope $\leftarrow$ convex polytope }

 2: for $n \in CN$ do
 3:     align node to global frame
 4:     $closest =$ closest node$(n, \mathcal{G})$
 5:     if dist xyz$(n, closest) > d\ min$ or dist z$(n, closest) > d_z\ min$ then
 6:         $\mathcal{G}.N = \mathcal{G}.N \cup n$
 7:     end if
 8: end for
 9: $N = \{n | n \in \text{polytope}\}$
10: $E = \{(n_i, n_j) | n_i, n_j \in N\}$
11: for $e \in E$ do
12:     if $e \notin \mathcal{G}.E$ then
13:         $\mathcal{G}.E = \mathcal{G}.E \cup e$
14:     end if
15: end for
16: return $\mathcal{G}$
```

In order to obtain such a path a few steps are required.

3.4.1. Endpoint selection

As explained in section 2.3 the target is selected among the obstacle points, choosing the location that optimises the exploration/exploitation tradeoff. Therefore the point is not part of the navigation graph, and it is not a safe location for the drone to be (as it is adjacent to an obstacle).

Moreover, due to the non-ideal drone behaviour and to the path planning asynchronous timing, it is possible that a new path needs to be computed while the drone is not located exactly in a node (the drone positioning may not be perfectly accurate and the path computation may trigger while the drone is already navigating).

Starting node

The starting node is simply computed as the node with minimum distance to the drone position. While this is an approximation, accurate tests showed that this approximated approach causes no drawbacks, except for a few (rare) turnarounds required to reach just abandoned nodes.

Target node

The target node is instead computed by selecting the reachable node closest to the target received.

Shortest path First the shortest paths between the starting node and all the graph nodes are computed. The shortest path computation excludes from the paths the nodes that are marked as not reachable and the edges that are marked as not walkable (as explained in section 3.5).

Visibility factor Then for each reachable node, the "visibility factor" is computed. The visibility factor defines how easily the drone would be able to film the target from the node and is simply computed as the distance between the node and the target, with an additional term that discourages nodes with a great height difference. In equation (3.4) is shown the formula for computing the visibility factor, k_1 being a big enough factor and k_2 the maximum height difference acceptable. More complex definitions of visibility can be defined, however this approach is simple and robust.

$$visibility\ f(node, target) = \text{dist}(node, target) + k_1 \cdot \mathbb{1}_{|node\ z - target\ z| > k_2} \tag{3.4}$$

3.4.2. Path creation

Given the starting and target node, it is possible to plan a path that traverses the graph. The algorithm used for the path planning is a slight modification of the well known Dijkstra algorithm [11], in which the weight of the edges represents the Euclidean distance between the two connected nodes. The modification consists on the fact that each edge stores a value called walkability, a boolean value that indicates whether the edge can be safely traversed. Therefore the shortest path algorithm must compute the optimal path between the walkable edges, excluding the other.

In figure 3.7 is shown how the graph structure influences the path planning process. Blue node and black edges represent reachable and walkable parts of the graph, while red nodes and red dotted lines are non reachable and non walkable. Although the shortest path connecting the bottom and the top nodes only traverse three edges, one of them is marked as non walkable due to being too close to the obstacle, therefore the algorithm must bypass the infamous edge and the optimal feasible path is four edges long.

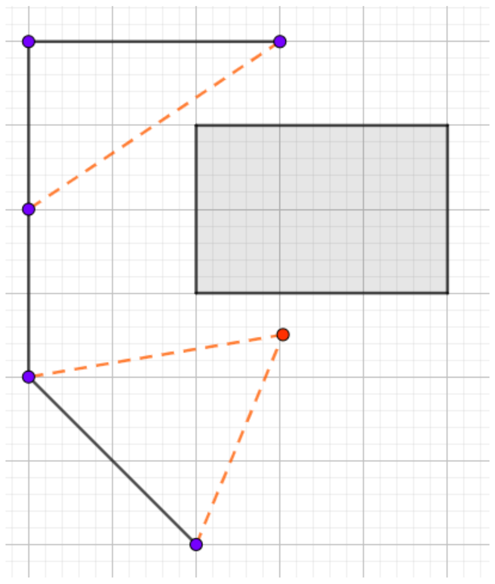


Figure 3.7: Example of a graph

3.4.3. No path available

A specific procedure is needed for handling the edge case that happens if no new paths are available. This is especially important at the beginning of the exploration and in local optimum points: at the beginning of the exploration the node distribution is often too compact and prevents the exploration advancement; while in local optimum the drone reaches a node that does not allow to film the target, while no closer nodes are available.

In figure 3.8 it is possible to see why the procedure is necessary. Given two obstacles (black), the navigation graph (straight lines and full points) and the target position, the drone reached the optimal position that does not allow to see the target. It must therefore continue the exploration, expanding the graph (empty points and dotted lines), without returning to the previously "optimal" node.

In order to solve this issue, a special flight procedure is performed any time the newly computed path is not different from the last, or if the target node is the same as the starting node.

The problem is handled as follows: first of all, the drone rotates in such a way that the camera faces the target. Then a full rotation is performed. Finally, the current node is set as "visited", so that until the received target changes, the current node will not be chosen again as the target node. The algorithm that describes this behaviour is shown in algorithm 3.3.

The first phase, the alignment to the obstacle, is necessary for ensuring that the obstacle

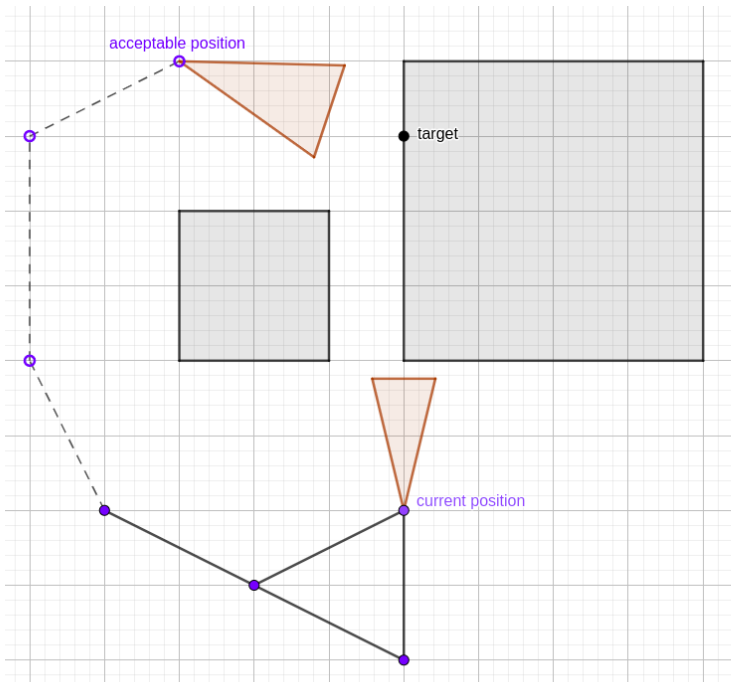


Figure 3.8: No path available

direction is clearly filmed, and possibly manage to map the area and either receive a new target or find a better path. The complete rotation around the axis is used in order to retrieve any information available from the current node. Again, this is done in order to hopefully find a new target or path. If the current node is clearly useless, after the node is completely explored it is inserted in a forbidden list, so that next paths will avoid setting as target node the current node. Of course, the forbidden list is only meaningful while reaching the current target, so once it is updated the list gets cancelled.

This approach, although quite simple, is very effective and helps avoid dangerous deadlocks. A problem could arise when a particularly tricky target is received, the best target node is reached and then discarded, the second-best target node is reached and then discarded, and so on, while the target does not change. At the end of the exploration, all the nodes have been analysed and discarded, so the forbidden list will contain all the nodes of the graph. This condition is theoretically possible, however in practice, it is irrelevant and is not handled during the development of the controller.

3.4.4. Target acceptance

An important element of the process of path planning is the criterion that guides the acceptance of a new target and the conditions that trigger the path planning process.

Algorithm 3.3 No path available procedure

```
1: { $forbidden\ list \leftarrow$ list of nodes that is avoided by the path planning algorithm
   $\backslash \leftarrow$ current node }
2: {phase 1: point to target}
3: if not $pointing\ toward\ target$ then
4:     rotate toward target
5: end if
6: {phase 2: complete rotation}
7: $rot = 0$
8: while $rot < 360$ do
9:     rotation of $rot$ angle
10:    $rot = rot + 45$
11: end while
12: {phase 3: add current node to forbidden list}
13: $forbidden\ list = forbidden\ list \cup n$
14: return $path$
```

Target acceptance

A new target is computed (as explained in section 2.3) every T_{mesh} seconds. However, not all the targets are used for the exploration process. Changing the current target requires a new path computation and the deletion of the forbidden list, and therefore a frequent update of the target would lead to a slower process.

The approach used to determine whether a newly received target should be used in the path planning checks if the new target has a distance from the current target greater than a threshold (controlled by the parameter *graph plan dist*). If this is the case, the received target is set as current (or next) target, otherwise it is discarded and no further action is taken. A high value of *graph plan dist* leads to a more stable navigation system, increasing the chances that the drone reaches the target node in a shorter amount of time at the cost of a possible suboptimal drone positioning. A low value of the parameter, instead, allows small correction of the target to be performed, optimizing the information collected at the cost of a possible longer travel time.

Path planning trigger

The path planning process triggers in certain specific moments.

First of all, it activates any time the drone reaches a node in the path. This ensures that the entire path is as optimized as possible, and also that the entire path is validated with the latest obstacle representation. Once the drone reaches the last node in the path and the last node in the path is the target node, before computing a new path the drone is oriented toward the target point, in order to maximise the area of obstacle filmed. Moreover, the process activates once a new target is received, if the previous path is empty, and when a newly computed path is rejected as unsafe (as described in section 3.4).

3.4.5. Pointcloud drift

Continued from section 2.4.

As described in the Obstacles handling chapter, a major issue encountered during the development of the controller is the pointcloud drift. If left untouched, the local obstacle pointclouds $\mathcal{P}_{loc}$ recorded while the drone moves are misaligned, causing mapping errors.

The solution adopted only saves a new pointcloud if the drone is not moving or rotating, in such a way that any time a pointcloud is recorded it is surely aligned with the global map. It is however necessary to ensure that any map is saved, as the drone never stops during its normal workflow.

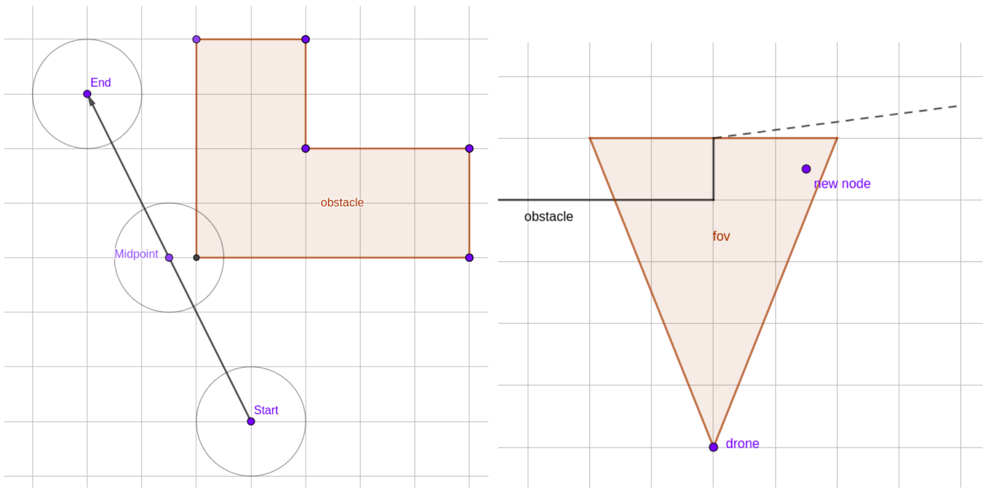
Therefore this second part enforces this requirement, blocking the drone for an amount of time equal to two times the fast drone timer $T\ timer\ fast$ whenever a waypoint is reached. This ensures that once the drone reaches a node the fast timer triggers at least once, letting a new local obstacle pointcloud to be integrated into the global graph.

3.5. Obstacle avoidance

An important requirement of the navigation system is the necessity of ensuring that any path computed is safe and traverses the graph without risk of hitting obstacles.

In figure 3.9 it is possible to understand some of the reasons that lead to the requirement of an obstacle avoidance system. On the left, both the starting and target node are at a safe distance wrt the obstacle, while the edge connecting the two nodes is dangerously close to the building. Therefore it is necessary to check if both newly created edges and already existing edges are safely traversable. On the right, the drone computes the convex polytope (as in section 3.1) and generates a point that seems to be far away from the obstacles. However later on a new wall is discovered, thus yielding the old node as non-reachable.

A controller can tackle the obstacle avoidance problem in two main directions: reactive



(a) Dangerous edge (b) Hidden obstacle

Figure 3.9: Path check issue

obstacle avoidance and passive obstacle avoidance.

3.5.1. Reactive obstacle avoidance

A reactive obstacle avoidance system is a low-level controller that actively interacts with the drone navigation system. It uses the robot's sensors and internal representation in order to verify whether the current trajectory may lead the robot to crash into obstacles. If a hit is predicted, the controller directly modifies the trajectory in order to avoid the barrier, ignoring the instructions generated by the path planning module.

An example of reactive obstacle avoidance is the G-BEAM controller [7]. This approach uses lidar measurements and acts on the drone's speed trying to discourage the movements toward the obstacle.

A reactive obstacle avoidance approach is able to guarantee that, no matter how the path the planning algorithm works, a safe distance between the drone and the obstacle is ensured to be kept. However this approach does not guarantee that the trajectory planned is actually followed, and this can be a problem if it is necessary to strictly follow the nodes of a graph.

3.5.2. Passive obstacle avoidance

A passive obstacle avoidance system does not rely on a low-level controller in order to avoid the obstacles. Instead, it ensures that any path computed by the path planning algorithm safely travels through the environment. It relies on the internal data structure to verify if a given path is safe enough.

A passive obstacle avoidance approach guarantees that the real trajectory strictly follows the planned path. The drawback is that the security control only uses the internal model of the environment, so the obstacle avoidance is secure only if the internal model is reliable: a wrong or unreliable obstacle representation can easily lead to a crash. Therefore a reactive obstacle avoidance approach can optionally support a passive obstacle avoidance system, especially if the internal obstacle representation is known to be noisy.

3.5.3. Proposed approach

The proposed obstacle avoidance process is a passive one, that checks whether a proposed path is far enough from the global obstacle pointcloud $\mathcal{P}$.

There are a number of possible solutions: it is possible to check the newly added nodes in the graph, or select only the nodes/edges that get updated each iteration, or scan the entire graph according to a timer. Each listed solution has some drawbacks, and none of them is particularly efficient.

Algorithm description

The proposed solution consists of a message passing between the graph and the path planning services. The algorithm is shown in algorithm 3.4: once a new path is computed, the list of traversed nodes is passed to the graph manager (located in the node Global information), which verifies whether all the nodes and all the edges are at a safe distance to the obstacles (for measuring the distance an octree built on the global obstacle pointcloud $\mathcal{P}$ is used, allowing to efficiently perform multiple queries). If any node or edge is not safe, the information is added to the global graph $\mathcal{G}$ and an updated graph is sent back to the Path planning node. The path planning node continues trying to plan a safe path (each new iteration excludes the nodes/edges invalidated in the previous) until a proposed path is accepted.

Once a path is validated, the controller selects the first node in the path and communicates with the low-level driver that steers the drone toward it.

Reactive obstacle avoidance

In order to take into account moving obstacles and measurement errors, that would lead to map drift, a reactive obstacle avoidance system based on a planar lidar is also adopted. The algorithm works similarly to the one described in the G-BEAM controller [7].

Algorithm 3.4 Path checking

```
1: { $CN \leftarrow$ input candidate graph nodes
   $\mathcal{G} \leftarrow$ current navigation graph
   $\mathcal{G}_{new} \leftarrow$ updated navigation graph provided by the verify path
   function if the path is not valid }

2: repeat
3:    $path$ = compute path()
4:    $(verified, \mathcal{G}_{new})$ = verify path()
5:    if not $verified$ then
6:       $forbidden = forbidden \cup n$
7:       $\mathcal{G} = \mathcal{G}_{new}$
8:    end if
9: until $verified$ = true

10: return $path$
```

`verify path`

```
1: { $\mathcal{G} \leftarrow$ navigation graph
   $path \leftarrow$ path proposal received by the other module,
   composed by as set of nodes and edges }

2: $path\ ok$ = true
3: for $n \in path.nodes$ do
4:    $dist$ = closest obstacle distance($\mathcal{G}, n$)
5:    if $dist < min\ dist$ then
6:       $\mathcal{G}.N[n].reachable$ = false
7:       $path\ ok$ = false
8:    end if
9: end for
10: for $e \in path.edges$ do
11:    for $p \in e.midpoint$ do
12:       $dist$ = closest obstacle distance($\mathcal{G}, p$)
13:       if $dist < min\ dist$ then
14:          $\mathcal{G}.E[e].walkable$ = false
15:          $path\ ok$ = false
16:       end if
17:    end for
18: end for

19: if $path\ ok$ then
20:    return true
21: else
22:    return false and $\mathcal{G}$
23: end if
```

4 | Test and Results

This chapter reports how the controller performed in a simulated test. Section 4.1 describes the testing setup, the testing environment and which evaluation metrics have been considered. Section 4.2 reports the results of the simulation, both quantitative and qualitative. Section 4.3 analyses the results and provides an evaluation of the main components of the controller. Finally, section 4.4 compares the proposed approach with a well known SLAM algorithm, RTAB-Map.

4.1. Testing setup

4.1.1. Simulation hardware and software

All the simulations and elaborations have been performed on a notebook, equipped with an Intel i7-6500 processor running at 2.5GHz, 12 GB RAM, and an Intel HD Graphics 520 graphic card. The software used for the testing are those described in the technology description section, while some handcrafted python scripts were used for elaborating the results.

4.1.2. Environment description

The controller has been tested in a simulated environment using gazebo. The testing area reproduces the Spino d'Adda satellite station [34], which contains various buildings, antennas and some vegetation. The objective of the flight is to map the main building, a big construction (it covers an area of 26x13 m, with a maximum height of 11 m) composed of two elements: a tall structure and a smaller one, separated by a hallway. The taller structure (width: 8m, depth: 9m, height: 11m) is mostly planar, with a few recesses of just a few centimetres in depth, and some windows, that are represented in gazebo as bigger holes. The smaller structure (width: 11m, depth: 13, height: 4 to 6m) is more complex, it has some windows, an overhanging roof, and a shelter supported by four small columns.

4.1.3. Testing parameters and metrics

Testing and noise parameters

A description of the parameters involved in the controller is listed in table 4.2, while the values adopted in the testing are shown in table 4.3.

The controller has been tested in two different operating conditions: without measurement errors (ideal conditions) and with measurement errors (real world conditions). The noise values used in the noisy simulation are listed in table 4.1. A brief explanation of the role of each error can be retrieved in [12].

Evaluation metrics

The metrics used for evaluating the performances are shown in table 4.4.

T_{tot} is the total running time, from the beginning to the end of exploration. T_{poly}, T_{update} and T_{mesh} indicate the maximum running time respectively for the Local information node, Graph information node, Mesh Handling node. Minimum running times are discussed in section 4.3, while the running time of the Path planning node is not reported, since it is negligible.

The Navigation Graph is evaluated by counting the number of nodes (N_{nodes}), the number of edges (N_{edges}), and the total distance traversed by the drone (d_{travel}).

The mesh is evaluated both quantitatively and qualitatively. The quantitative measures include the size of the pointcloud that represents the obstacles (N_{obs}), the number of vertices used in the mesh reconstruction (N_{vert}), the number of faces in the reconstructed mesh (N_{faces}), and some statistics computed from the mesh. As explained in [15], from the reconstructed mesh the centroids of each polygon are extracted (the formula for centroid extraction is shown in equation (4.1)), the distances between those points and the ground truth surface are computed, and the maximum distance (d_{max}), median distance (d_{med}), mean distance (d_{μ}) and standard deviation (d_{σ}) are reported. The qualitative evaluation of the mesh consists of the visual analysis of some details in the reconstructed mesh, highlighting the qualities and limitations of the reconstruction.

$$\begin{cases} x_c = \dfrac{x_1 + x_2 + x_3}{3}, \\ y_c = \dfrac{y_1 + y_2 + y_3}{3}, \\ z_c = \dfrac{z_1 + z_2 + z_3}{3}, \end{cases} \tag{4.1}$$

GPS	*random walk*$_{xy}$	1.0
	random walk$_z$	2.0
	noise density$_{xy}$	1.0e-4
	noise density$_z$	1.0e-4
	noise density v_{xy}	1.0e-1
	noise density v_z	2.0e-1
Gyroscope	*noise density*	1.9e-3
	random walk	3.9e-5
	bias correlation time	1.0e3
	turn on bias sigma	8.7e-3
Accelerometer	*noise density*	1.9e-3
	random walk	6.0e-4
	bias correlation time	3.0e2
	turn on bias sigma	2.0e-2
Magnetometer	*noise density*	4.0e-4
	random walk	6.4e-7
	bias correlation time	6.0e2
RGBD-Camera	*noise RGB*	5.0e-2
	noise depth	2.0e-2

random walk	$\left[\frac{rad}{s^2}\frac{1}{\sqrt{Hz}}\right]$	$\left[\frac{m}{s^3}\frac{1}{\sqrt{Hz}}\right]$
noise density	$\left[\frac{rad}{s}\frac{1}{\sqrt{Hz}}\right]$	$\left[\frac{m}{s^2}\frac{1}{\sqrt{Hz}}\right]$

Table 4.1: Noise values used in the simulation

Field	parameter name	measure	reference	description
Timing parameters	*T poly*	[s]	section 2.1	how often a new local point-cloud is read and processed
	T fast timer	[s]	section 2.4 section 3.4	timer for checking if drone just stopped
	T mesh	[s]	section 2.2	how often a new mesh is computed, is related with the delay of visualization, computational load and target update frequency

Continued on next page

Table 4.2 – continued from previous page

Field	parameter name	measure	reference	description
Obstacle density	d_{cloud}	[m]	section 2.1	local obstacle pointcloud $\mathcal{P}_{local}$ downsampling distance; average distance between points
	$d_{vertices}$	[m]	section 2.2	mesh $\mathcal{M}$ vertices minimum distance, is related with mesh quality and computational load
Target selection	d_{target}	[m]	section 2.3	radius for clustering together the candidate targets, is related with the field of view of the camera
	exp_{expl}	[-]	section 2.3	exploration/exploitation weighting factor
	graph plan dist	[m]	section 3.4	minimum distance between current target and new candidate target required for accepting the new target
Graph construction	*poly_max_depth*	[m]	section 3.1	maximum polytope height, is related with the reliability of the camera pointcloud and affects the proximity of nodes of the graph
	poly_delta	[m]	section 3.1	shrinking factor for polytope generation, is related with the quality and efficiency of the convex polytope
	graph d min	[m]	section 3.3	minimum distance between nodes at the same height
	graph d_z min	[m]	section 3.3	minimum vertical distance between nodes

Continued on next page

Table 4.2 – continued from previous page

Field	parameter name	measure	reference	description
Pointcloud drift	$v_{lin\ max}$	[m/s]	section 2.4	maximum linear speed that allows obstacles to be updated
	$v_{rot\ max}$	[deg/s]	section 2.4	maximum rotational speed that allows obstacles to be updated

Table 4.2: Simulation parameters description

Field	parameter name	value
Timing parameters	$T\ poly$	2 s
	$T\ fast\ timer$	0.5 s
	$T\ mesh$	5 s
Obstacle density	d_{cloud}	0.1 m
	$d_{vertices}$	0.3 m
Target selection	d_{target}	0.5 m
	exp_{expl}	3
	$graph\ plan\ dist$	0.3 m
Graph construction	$poly\_max\_depth$	3 m
	$poly\_delta$	0.5 m
	$graph\ d\ min$	1.5 m
	$graph\ d_z\ min$	0.5 m
Pointcloud drift	$v_{lin\ max}$	0.2 m/s
	$v_{rot\ max}$	10 deg/s

Table 4.3: Simulation parameters value

4.2. Testing results

Numerical results

The numerical results obtained from the tests are reported in table 4.4.

			without noise	with noise
Time	T_{tot}	[s]	4300	4500
	T_{poly}	[s]	0.3	0.5
	T_{update}	[s]	0.07	0.2
	T_{mesh}	[s]	0.58	1.03
Graph	N_{nodes}		2318	2245
	N_{edges}		6444	5385
	d_{travel}	[m]	1070	1110
Mesh	N_{obs}		118273	285917
	N_{vert}		4936	4956
	N_{faces}		9586	9625
	d_{max}	[m]	0.342	0.507
	$\mathbf{d}_{\mu}$	[m]	**0.048**	**0.101**
	d_{σ}	[m]	0.043	0.062
	$\mathbf{d}_{\mathbf{med}}$	[m]	**0.039**	**0.083**

Table 4.4: Simulation results

Ground truth model

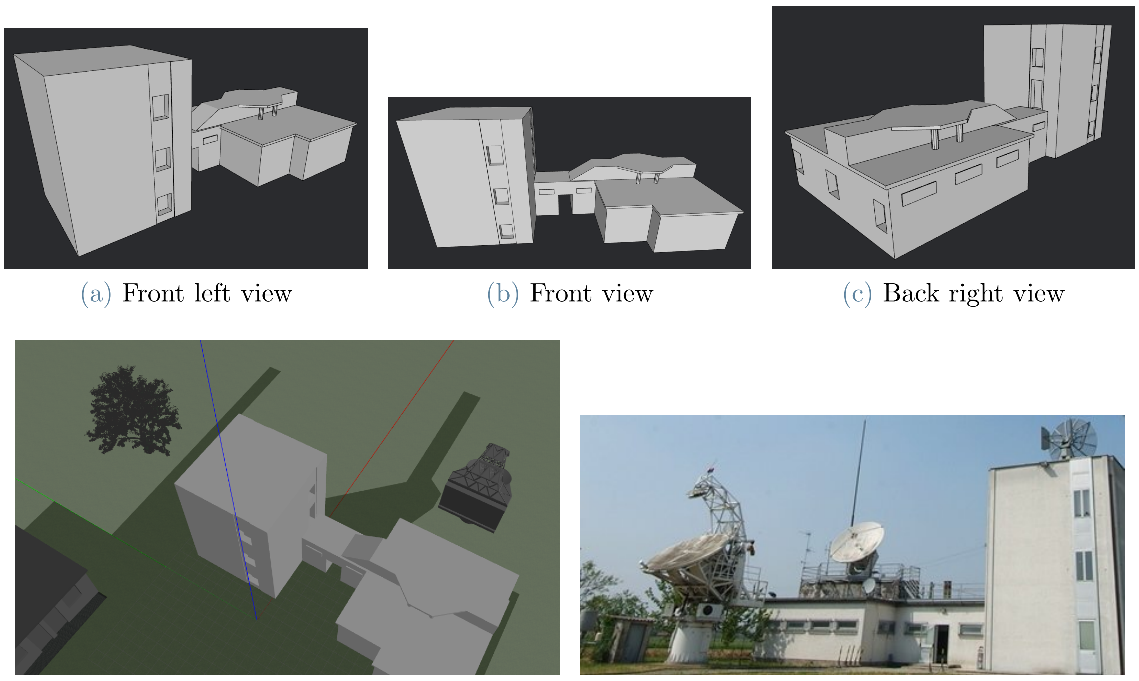
In figure 4.1 is shown the model of the building mapped during the tests. In figure 4.1a the tall building is in the foreground, and it is possible to see the windows and the small recess. In figure 4.1b the hallway is clearly visible, while in figure 4.1c the overhanging roof, the shelter and the columns can be seen. Finally, in figure 4.1d the whole gazebo model is shown. The obstacles present are a tree (on the left) and an antenna (on the top right); the small building on the bottom left is far enough and does not interfere with the mapping.

4.2.1. Reconstructed overview

In figure 4.2 is shown the reconstructed model. It is possible to see the vertices used for reconstructing the mesh, the wireframe of the reconstructed mesh, and the mesh itself. Due to the lack of textures and the low quality light reflection, it is difficult to easily discern the individual elements from a static image.

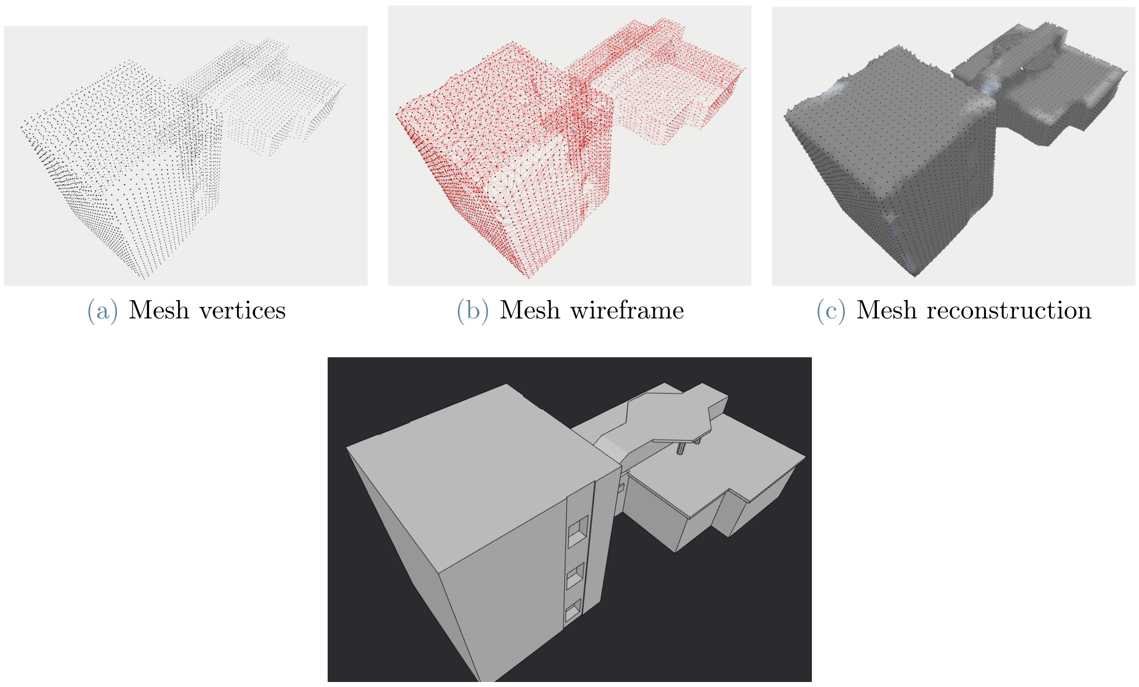
4.2.2. Detail: shelter

In figure 4.3 is shown a detail of the small building. The front shelter with columns is visible. In figure 4.3b and figure 4.3c it is possible to see the obstacle pointcloud (black dots), the vertices of the mesh (yellow squares), the wireframe and the final mesh. The reconstruction is not optimal in both cases. In the noisy case the obstacle pointcloud is


(a) Front left view (b) Front view (c) Back right view

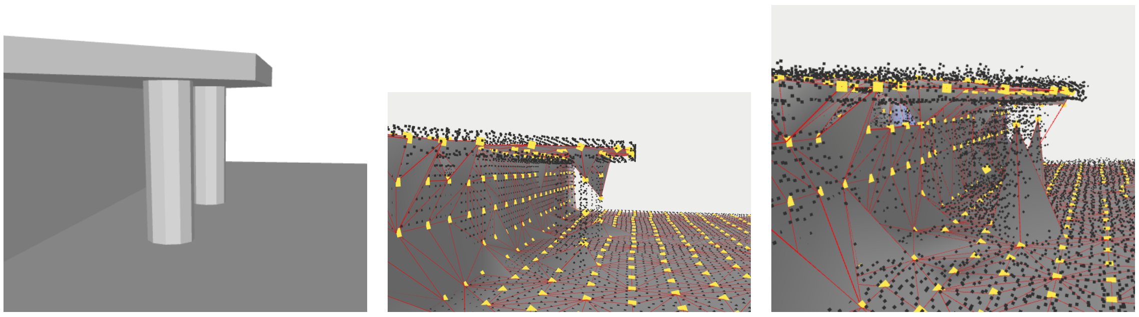
(d) Spino d'Adda complete model (e) Spino d'Adda site

Figure 4.1: Overview of the test environment


(a) Mesh vertices (b) Mesh wireframe (c) Mesh reconstruction

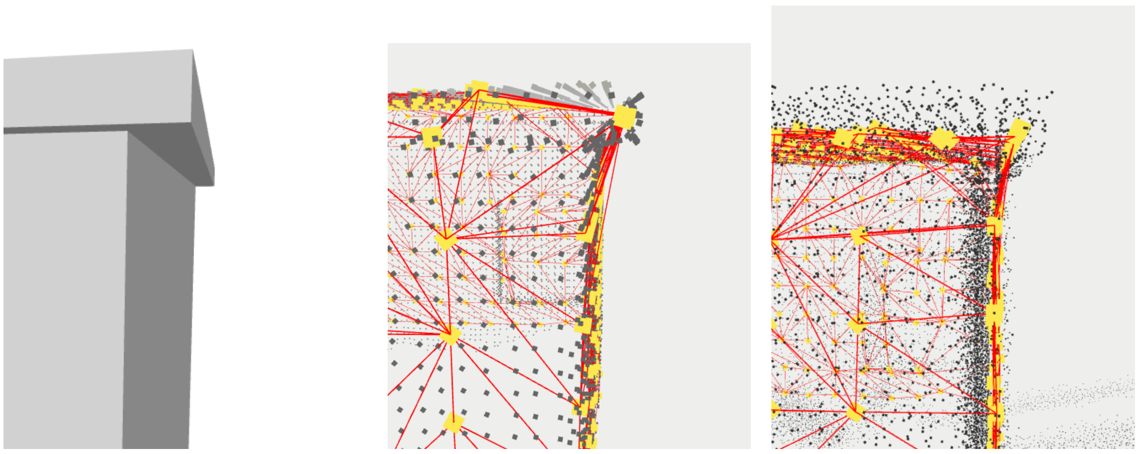
(d) Ground truth

Figure 4.2: Noiseless reconstruction



(a) Ground truth (b) Without noise (c) With noise

Figure 4.3: Shelter and column details



(a) Ground truth (b) Without noise (c) With noise

Figure 4.4: Overhanging roof detail

more shattered, making it even more difficult to reconstruct correctly the mesh.

4.2.3. Detail: roof

Similarly, in figure 4.4 is shown a detail of the overhanging roof in the small building. Here the final mesh is not shown, and it is easier to detect the noisy pointcloud, that however gets correctly filtered during the vertex detection. It is also visible how the vertices distribution ignores some detail in the ground truth model, so the overhanging roof is approximated as a sloping wall.

4.3. Discussion

The simulation shows that the addition of the noise does not drastically change the results, and in both the tests the reconstructed mesh is quite precise, slightly better in the noiseless simulation.

4.3.1. Odometry

The use of Visual Odometry (i.e. deriving the robot position from the image stream) is not suitable in a textureless simulated environment. Due to the lack of details in the surfaces, it is difficult for the algorithm to correctly detect movements, thus leading to unexpected jumps and localisation errors. The problem could partially be solved by integrating Visual and Inertial Odometries, however, the result would still be unnecessarily noisy and unreliable, so we decided to keep only the Inertial Odometry (i.e. GPS and IMU reading).

4.3.2. Running time

Total running time

The total exploration time T_{tot} is quite high, this is mainly due to the pointcloud drift problem described in section 2.4, that requires the drone to stay still for some moments at every waypoint reached. The high time can be partially explained also by the low maximum speed, imposed for safety reasons, that could be increased without concern.

The noisy running time, as well as the noisy distance travelled, is slightly higher than the noiseless case. This is however not caused by the presence of the noise. Instead, the small difference in travel distance and total time is due to different decisions taken by the target selection and path planning algorithm, which caused the drone to travel a bit longer.

Minimum running times

In table 4.4 the minimum running times of the nodes are not reported, as not useful in the analysis. The polytope generation running time does not depend on the mapping progress, its small variations are only due to the variable raw pointcloud size. Its minimum value $T_{poly\ min}$ is approximately 0.1 s for both the noiseless and noisy case. The graph update running time does not depend on the mapping status, and the minimum values $T_{update\ min}$ changes only slightly with respect to the raw obstacle size. The mesh handling node depends heavily on the size of the global obstacle pointcloud. At the beginning of the mapping the minimum values $T_{mesh\ min}$ are negligible, and then linearly grow with respect to the mesh size.

Partial times

The polytope generation time T_{poly} is constant over the whole flight, since it does not depend on the graph and obstacle size. The graph update time T_{update} greatly increases during the exploration: this can be explained by the big and complex data structures that must be accessed at each iteration. The difference in T_{update} between the noiseless and noisy simulations is due to the difference in size of the partial cloud that needs to be integrated into the global pointcloud: the noisy cloud can not be efficiently compressed, so it is much bigger and requires more time to be integrated. Similarly, the meshing time T_{mesh} doubles between the (relatively) small pointcloud digested in the noiseless case and the bigger one analysed in the noisy case.

Noise contribution and parameter tradeoff

The main difference between the running times of the noiseless and noisy cases is caused by the number of points inside the obstacle pointcloud, which in turn are the result of the aliases generated by the presence of noise. This problem can be reduced by reducing the pointcloud density (i.e. increasing the distance between points), at the cost of a less precise reconstruction.

Gazebo contribution

As a side note, Gazebo itself consumes a lot of computational resources, so using the controller in a real world scenario could greatly reduce the time spent on the various modules. For example, running the meshing node without gazebo (on a precomputed obstacle pointcloud) reduces T_{mesh} from over half a second to about 0.30 s in the noiseless case, and from one second for the noisy to 0.50 in the noisy one.

4.3.3. Graph management

Graph management works almost flawlessly. The iterative polytope generation (as described in section 3.1) speeds up the exploration during the whole run and helps avoiding some deadlocks that could happen at the beginning of the process. The addition of noise to the sensors do not decrease the performances: the measurements errors do cause a small shift of the nodes, which is however irrelevant during the flight. All the nodes are kept at a safe distance from the obstacles, the path planning algorithm allows the drone to plan a safe path between any nodes, and the maximum number of consecutive path checks (see section 3.4) before acceptance is always quite low, usually three to five.

Obstacle avoidance

As explained in section 3.5, the proposed passive obstacle avoidance system is associated with a pre-existing reactive obstacle avoidance approach. This is for ensuring that the drone is always safely distant from the obstacles.

However, during the test, it has been noticed that the reactive obstacle avoidance controller never modified the trajectory planned by the path planning node. This validates the proposed passive obstacle avoidance algorithm, and the removal of the reactive obstacle avoidance controller should not cause issues in the safety of the navigation.

4.3.4. Mesh

The meshing algorithm works quite well in most the situations, but it is not very precise if the obstacle pointcloud is not clearly defined (for example thin objects, as in figure 4.3). The reconstruction of the tall building is complete and almost perfect both in the noiseless and noisy cases, while the smaller building has higher errors and is more affected by the noise.

Quantitative analysis

Even if the obstacle pointcloud size N_{obs} is very different between the noiseless and noisy case, the number of vertices N_{vert} and faces N_{faces} are almost identical. This emphasizes the quality of the meshing workflow, which is able to filter out the noise.

In table 4.4 it is shown that the noisy case has worse results than the noiseless in all the distance metrics. Both the average distance d_{μ} and the median distance d_{median} double from one run to the other. However, the median distance d_{median} shows that most of the points are closer to the actual building than the average distance, and the average distance itself is quite small in both cases.

Qualitative analysis

The error is distributed mainly near the edges of the building. This is due to the fact that the obstacle pointcloud is not exactly aligned with the building structure, so edges often penetrate the building (see for example figure 4.4). In the tall building, the main sources of error are the windows, that although being completely covered by the mesh they are not exactly mapped. In the smaller building, the errors are distributed along with the smaller details, in particular near the windows and the overhanging roof.

Regarding the next target computation, the adopted approach seems to be able to compute efficiently the best position to be scanned, with an acceptable tradeoff between distance and information gain.

Parameter tradeoff

Most of the problems related to the meshing errors could be solved by increasing the pointcloud density, thus reducing the size of the edges. This solution has two main drawbacks. Firstly, it makes the computation much slower: just by halving the distance between points, the computation time required for the meshing in the noiseless case spikes from half a second to about five seconds. Secondly, the robustness to the noise rapidly decreases: all the noise that is not "filtered" by downsampling the pointcloud will appear in the final mesh, so increasing the point nearness must be performed with caution.

4.4. RTAM-Map comparison

A comparison between the controller and a well known mapping framework has been performed. This allows checking whether the controller, aside from producing excellent results on its own, can be compared with a State of the Art technique. The chosen framework is RTAB-Map [25], a popular SLAM algorithm.

In order to evaluate the two approaches, we decided to compare the generated obstacle pointclouds and verify whether they are similar or not. For the controller the cloud is the global obstacle pointcloud, while for RTAB-Map the cloud is the one published on the topic `rtabmap/cloud_map`. Due to the noise implementation in the camera, it is not possible to feed the noise to RTAB-Map, so only the noiseless case is analysed.

Upon the various metrics proposed for comparing pointclouds (an analysis of some broadly used metrics is available in [26]), a popular one is the Hausdorff distance [17], computed as the maximum distance between any point in a cloud and the closest point in the other cloud (see equation (4.2)). A more informative metric is instead the L2 norm between each point of a class and the closest point of the other class. From the L2 distances, the maximum, minimum, mean, median and standard deviation are extracted.

$$d_H(X,Y) = \max\left\{\sup_{x\in X} d(x,Y), \sup_{y\in Y} d(X,y)\right\} \tag{4.2}$$

The results are available in table 4.5 and table 4.6 (note that in table 4.5 the "proposed controller" column is the same as table 4.4).

		RTAB-Map	proposed controller
N_{obs}	[-]	447568	118273
d_{max}	[m]	0.548	0.342
d_{μ}	[m]	0.042	0.048
d_{σ}	[m]	0.033	0.043
d_{med}	[m]	0.035	0.039

Table 4.5: RTAB-Map and proposed controller distances wrt ground truth

d_{min}	[m]	0.0005
d_{max}	[m]	0.438
d_{μ}	[m]	0.042
d_{σ}	[m]	0.020
d_{med}	[m]	0.041

Table 4.6: RTAB-Map and proposed controller distance with respect to each other

RTAB-Map and controller vs ground truth

From table 4.5 it is interesting to notice how the pointcloud density influences the distance errors. RTAB-Map stores four times the number of points that the proposed controlled handles, with no quality improvement. All the measured parameters, the average distance d_{μ}, the variance d_{σ} and the median distance d_{median} are very similar, with RTAB-Map having just slightly better results.

In table 4.6 is shown the distances between the two pointclouds. The average d_{μ} and median d_{med} distances show that the two pointclouds are almost perfectly overlapping. The huge maximum distance d_{max} is due to the fact that the meshing algorithm does not require the whole surface to be scanned, as some small areas can be left unmapped without errors in the reconstruction. Inside those zones the obstacle pointcloud and the RTAB-Map's one could differ.

In conclusion, the two approaches produce comparable results. The two pointclouds are very similar, and both are close to the ground truth. Moreover, the proposed approach is simpler than RTAB-Map, and therefore much faster to execute.

RTAB-Map advantage

It is important to remind that RTAB-Map is a powerful SLAM algorithm, while the proposed approach only performs Mapping. Moreover, RTAB-Map is able to perform loop closures and handle higher levels of noise. Therefore, in a real noise environment, the proposed controller would easily be outperformed by RTAB-Map.

5 | Conclusion

This thesis focused on the development of a controller that solves the task of simultaneous exploration and mapping for fully autonomous vehicles, adopting a mixed graph-mesh approach.

The proposed solution tackles the problems of Exploration and Mapping individually, keeping the two computation flows as independent as possible, thus allowing to optimise each step without worries of interdependencies (figure 1.3 shows how each data flow is quite independent from the other).

For the Mapping task, a Mesh is constructed from a pointcloud, that is built by merging the stream published by the camera. Using the mesh, the next best target to be filmed is extracted.

The Exploration task is solved by constructing a Navigation Graph from a series of convex polytopes that represents free space. The graph is then used for computing the optimal path that drives the drone to the desired location. An important edge case is tackled with care: namely when the optimal node is not suitable for mapping the target.

Additionally, a passive obstacle avoidance system has been developed, that takes care of ensuring that any planned path is safe and obstacle free.

The proposed controller has been validated in a simulated environment of an existing testing area. The results are surprisingly good, the controller managed to drive the drone and map a large building with excellent precision and efficiency. Some qualitative and quantitative measurements are taken, validating the controller result. Moreover, although the development focused mostly on an error-free scenario, some noisy trials revealed that the performance in presence of noise is still quite good.

Finally, the Mapping section of the proposed controller has been compared with a state of the art SLAM approach, RTAB-Map, and the results showed that the two obstacle representation are very similar, while the performances are all in favour of the proposed controller.

In conclusion, the proposed controller performs autonomous exploration and mapping, with good results in a noiseless environment and satisfactory result in a noisy environment. Notably, with a low amount of noise, the proposed controller produces similar artefacts as a State of the art approach, with the advantage of lower computational load.

5.1. Novel elements

The proposed controller introduces novelties in different fields.

Convex polytope generation A novel algorithm for generating a three-dimensional convex polytope has been developed. The proposed approach guarantees a good tradeoff between computational cost and covered volume, allowing to optimise the extraction of candidate graph's nodes,

Passive obstacle avoidance Compared to the unpredictable result of a reactive obstacle avoidance system, the proposed passive obstacle avoidance approach allows planning a reliable path that is known to be safely distant from obstacles and that can be exactly followed. The proposed approach, compared to other possible solutions, manages to reduce the computational requirements for the check. Note that the use of a reactive obstacle avoidance system is still advised if moving objects or measurement errors (and thus map drift) may be present.

Mesh based exploration The mesh based exploration is a novel idea, quite unexplored in the literature, that has been tackled and successfully solved with remarkable results. The use of the mesh allowed to seamlessly generate the frontiers of the unknown area and evaluate the potential targets.

Obstacle based exploration Differently from the most studied approach, the proposed controller does not focus on exploring the free space (i.e. frontier defined as free space near unknown space), but instead, analyse the frontier between occupied space and the unknown area. This approach allows the drone to only explore connected and visible obstacles, thus leading to much faster exploration that is suitable in certain conditions (for example, if the objective is to map a specific building).

Mixed mesh-graph approach Most approaches to the Exploration and Mapping task only rely on a single data structure, or maintain a main Mapping structure and then derive some information about the Exploration. The proposed controller instead builds and maintains two different data structures, each one specialised, and thus optimised, on a single task. While there exists data structures that perform fairly well both on Exploration and Mapping (for example occupancy grids), having the possibility of optimising the two tasks separately is a plus.

5.2. Future improvements

Although the proposed controller produced excellent results, a number of improvement directions could be analysed.

Real world test The controller has been tested exclusively in a simulated environment. It would be interesting to evaluate the performances outside the software. This would allow testing some features that could not be used in gazebo (for example, the textureless environment makes it impossible to use visual odometry).

Multidrone The proposed controller is developed for a single drone mapping, thus no cooperation is possible. An interesting development direction would integrate the measurements of different robots into a single map, thus allowing for faster, collaborative and optimised reconstruction.

Prior knowledge integration in non-static environment Although the proposed controller allows performing a multisession exploration and mapping, it is not possible to correct an already created map. Adding the possibility of invalidating old sensor readings would allow tracking the changes on the map structure while keeping the map updated.

SLAM The proposed approach requires an external localisation module to provide information about the drone position. The most common sensors have some well known drawbacks: the GPS signal is not always available and not always reliable, and a low quality IMU may quickly build up errors. An interesting evolution of the controller would be the use of the local pointcloud and the global Mapping structure to perform localisation, thus making the controller a proper Simultaneous Localisation and Mapping approach. Moreover, the localisation with the camera would allow detecting loop closures, thus improving even more the quality of the reconstruction in noisy environments.

Bibliography

[1] H. Azpúrua, M. F. M. Campos, and D. G. Macharet. Three-dimensional terrain aware autonomous exploration for subterranean and confined spaces. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2443–2449, 2021. doi: 10.1109/ICRA48506.2021.9561099. (page 8)

[2] A. Batinovic, T. Petrovic, A. Ivanovic, F. Petric, and S. Bogdan. A multi-resolution frontier-based planner for autonomous 3d exploration. *IEEE Robotics and Automation Letters*, 6(3):4528–4535, 2021. (page 8)

[3] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, J. A. Levine, A. Sharf, and C. Silva. State of the art in surface reconstruction from point clouds. *Eurographics STAR (Proc. of EG'14)*, 2014. (page 4, 31)

[4] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, 1999. doi: 10.1109/2945.817351. (page 4)

[5] D. Bommes, B. Lévy, N. Pietroni, C. Silva, M. Tarini, and D. Zorin. State of the art in quad meshing, 2012. (page 4, 31)

[6] C. Campos, R. Elvira, J. J. Gomez, J. M. M. Montiel, and J. D. Tardos. ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021. (page 6)

[7] L. Cecchin, D. Saccani, and L. Fagiano. G-beam: Graph-based exploration and mapping for autonomous vehicles. In *2021 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1011–1016. IEEE, 2021. (page 6, 43, 59, 60)

[8] C. Connolly. The determination of next best views. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, volume 2, pages 432–435, 1985. doi: 10.1109/ROBOT.1985.1087372. (page 5)

[9] B. Curless and M. Levoy. A volumetric method for building complex models from

range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996. (page 4)

[10] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32(3):505–536, jul 1985. ISSN 0004-5411. doi: 10.1145/3828.3830. URL `https://doi.org/10.1145/3828.3830`. (page 5)

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. (page 5, 54)

[12] ETHZ ASL. Imu noise model, 2021. URL `https://github.com/ethz-asl/kalibr/wiki/IMU-Noise-Model`. (page 64)

[13] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 03 1974. doi: 10.1007/BF00288933. (page 3)

[14] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart. *RotorS - A Modular Gazebo MAV Simulator Framework*, volume 625, pages 595–625. , 01 2016. ISBN 978-3-319-26054-9. doi: 10.1007/978-3-319-26054-9_23. (page 9)

[15] A. Handa, T. Whelan, J. McDonald, and A. Davison. A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM. In *IEEE Intl. Conf. on Robotics and Automation, ICRA*, Hong Kong, China, May 2014. (page 64)

[16] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. doi: 10.1007/s10514-012-9321-0. URL `https://octomap.github.io`. Software available at `https://octomap.github.io`. (page 4)

[17] D. Huttenlocher, G. Klanderman, and W. Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, 1993. doi: 10.1109/34.232073. (page 74)

[18] Intel RealSense. Depth camera d435, 2021. URL `https://www.intelrealsense.com/depth-camera-d435/`. (page 9, 18)

[19] Intel RealSense. Ros wrapper for intel realsense devices, 2021. URL `https://github.com/IntelRealSense/realsense-ros`. (page 11)

[20] J. Ivković, A. Veljović, B. Randjelovic, and V. Veljović. Odroid-xu4 as a desktop pc and microcontroller development boards alternative. In , 05 2016. (page 10)

[21] M. M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson surface reconstruction. In *SGP 06*, 2006. (page 4)

[22] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004. (page 9)

[23] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui. Micro air vehicle link (mavlink) in a nutshell: A survey. *IEEE Access*, 7:87658–87680, 2019. doi: 10.1109/ACCESS.2019.2924410. (page 12)

[24] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86, 1951. doi: 10.1214/aoms/1177729694. URL `https://doi.org/10.1214/aoms/1177729694`. (page 33)

[25] M. Labbé and F. Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, 2019. doi: https://doi.org/10.1002/rob.21831. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21831`. (page 6, 74)

[26] V. Lehtola, H. Kaartinen, A. Nuchter, R. Kaijaluoto, A. Kukko, P. Litkey, E. Honkavaara, T. Rosnell, M. Vaaja, J.-P. Virtanen, M. Kurkela, A. Issaoui, L. Zhu, A. Jaakkola, and J. Hyyppä. Comparison of the selected state-of-the-art 3d indoor scanning and point cloud generation methods. *Remote Sensing*, 9:796, 08 2017. doi: 10.3390/rs9080796. (page 74)

[27] Z. C. Marton, R. B. Rusu, and M. Beetz. On Fast Surface Reconstruction Methods for Large and Noisy Datasets. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 12-17 2009. (page 4, 31)

[28] D. Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer, 10 1980. (page 3)

[29] W. Meeussen. Coordinate frames for mobile platforms, 2010. URL `https://www.ros.org/reps/rep-0105.html`. (page 14)

[30] H. Moravec and A. Elfes. High resolution maps from wide angle sonar. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, volume 2, pages 116–121, 1985. doi: 10.1109/ROBOT.1985.1087316. (page 3)

[31] R. Munroe. New robot, 2019. URL `https://xkcd.com/2128/`. (page 1)

[32] pal-robotics. Intel realsense gazebo ros plugin, 2021. URL `https://github.com/pal-robotics/realsense_gazebo_plugin`. (page 11)

[33] E. Piazza, A. Romanoni, and M. Matteucci. Real-time cpu-based large-scale three-dimensional mesh reconstruction. *IEEE Robotics and Automation Letters*, 3(3):1584–1591, 2018. doi: 10.1109/LRA.2018.2800104. (page 4, 31)

[34] Politecnico di Milano. Spino d'adda satellite station, 2021. URL `https://www.deib.polimi.it/eng/deib-labs/details/45`. (page 63)

[35] I. M. Rekleitis, G. Dudek, and E. E. Milios. *Graph-Based Exploration using Multiple Robots*, pages 241–250. Springer Japan, Tokyo, 2000. doi: 10.1007/978-4-431-67919-6_23. URL `https://doi.org/10.1007/978-4-431-67919-6_23`. (page 3)

[36] Y. Roth-Tabak and R. Jain. Building an environment model using depth information. *Computer*, 22(6):85–90, 1989. doi: 10.1109/2.30724. (page 3)

[37] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011. IEEE. (page 9)

[38] P. Senarathne and D. Wang. Towards autonomous 3d exploration using surface frontiers. In *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 34–41. IEEE, 2016. (page 5, 6)

[39] M. Servières, V. Renaudin, A. Dupuis, and N. Antigny. Visual and visual-inertial slam: State of the art, classification, and experimental benchmarking. *Journal of Sensors*, 2021, 2021. (page 6, 7)

[40] M. Skotniczny. Computational complexity of hierarchically adapted meshes. In V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, editors, *Computational Science – ICCS 2020*, pages 226–239, Cham, 2020. Springer International Publishing. ISBN 978-3-030-50420-5. (page 29)

[41] G. Snook. Simplified 3d movement and pathfinding using navigation meshes. *Game programming gems*, 1(1):288–304, 2000. (page 4)

[42] Stanford Artificial Intelligence Laboratory et al. Robotic operating system, 2018. URL `https://www.ros.org`. (page 9)

[43] C. Toth, J. O'Rourke, and J. Goodman. Basic proprieties of convex polytopes. In *Handbook of Discrete and Computational Geometry*, chapter 15. CRC Press, 2017. ISBN 9781498711425. doi: 10.1201/9781420035315. URL `https://www.csun.edu/~ctoth/Handbook/HDCG3.html`. (page 42, 44)

[44] M. P. Tully Foote. Standard units of measure and coordinate conventions, 2015. URL `https://www.ros.org/reps/rep-0103.html`. (page 14)

[45] W. G. Van Toll, A. F. Cook IV, and R. Geraerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23(6):535–546, 2012. (page 4)

[46] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, T. Whelan, J. Mcdonald, M. Kaess, M. Fallon, H. Johannsson, and J. J. Leonard. Kintinuous: Spatially extended kinectfusion, 2012. (page 4, 6)

[47] T. Whelan, S. Leutenegger, R. Salas Moreno, B. Glocker, and A. Davison. Elasticfusion: Dense slam without a pose graph. *Robotics: Science and Systems XI*, 2015. doi: 10.15607/rss.2015.xi.001. (page 4)

[48] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA 97. Towards New Computational Principles for Robotics and Automation*, pages 146–151, 1997. doi: 10.1109/CIRA.1997.613851. (page 5)

[49] P. Yap. Grid-based path-finding. In *Conference of the canadian society for computational studies of intelligence*, pages 44–55. Springer, 2002. (page 5)

[50] I. Zarembo and S. Kodors. Pathfinding algorithm efficiency analysis in 2d grid. In *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings of the International Scientific and Practical Conference*, volume 2, pages 46–50, 2013. (page 5)

List of Figures

List of Tables

List of Algorithms