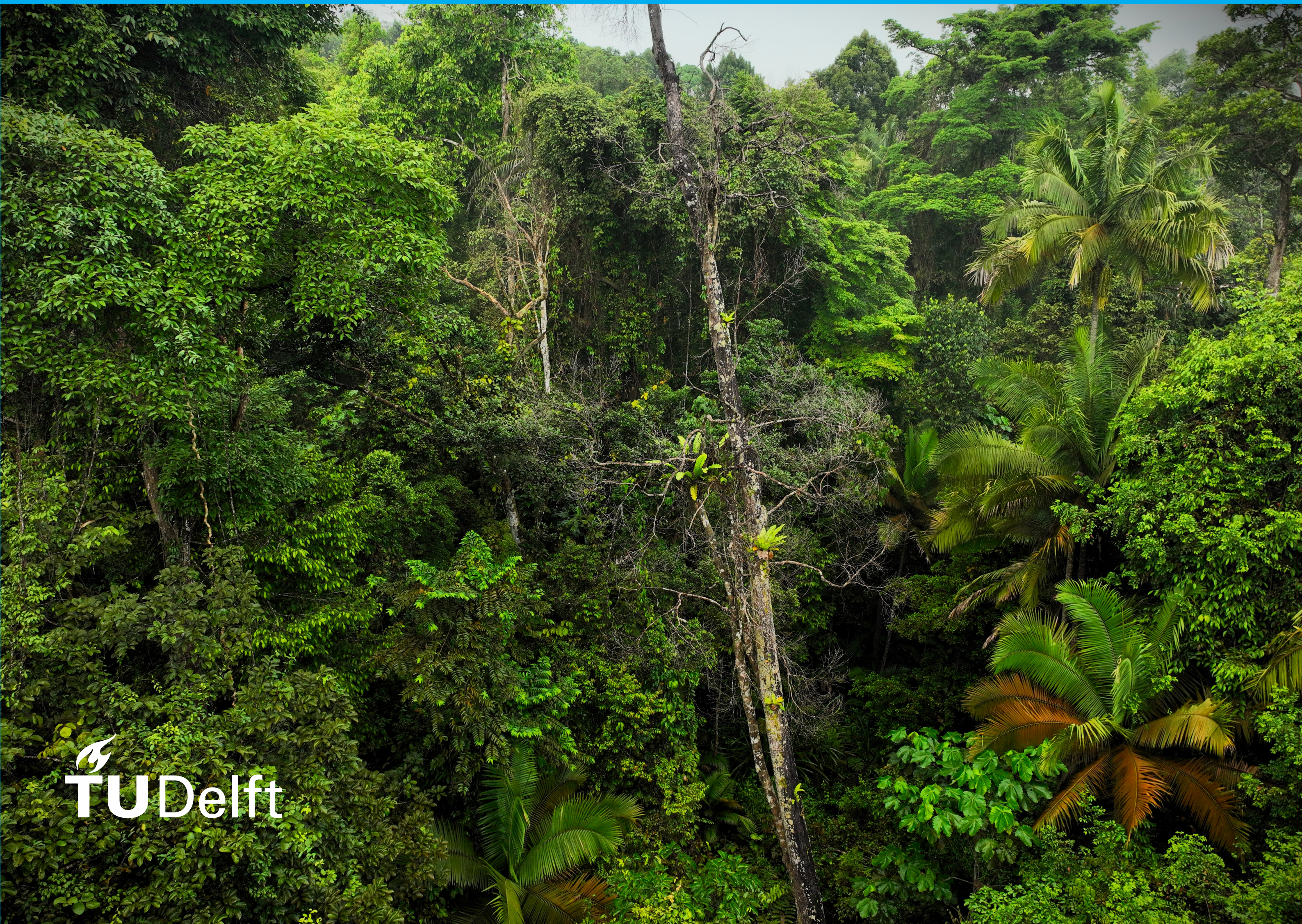
A lightweight quadrotor autonomy system

To navigate in densely cluttered forest environments

A. Zwanenburg

A solution to safe planning-based navigation
Full autonomy on a 400grams quadrotor
Without the use of GPS, SLAM, or motion-capture systems



A lightweight quadrotor autonomy system

To navigate in densely cluttered forest environments

by

A. Zwanenburg

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday January 22, 2024 at 2:00 PM.

Student number:	5413494	
Project duration:	Januari 5, 2023 – January 22, 2024	
Thesis committee:	Prof. dr. ir. M. Wisse,	TU Delft, Biorobotics, supervisor
	Dr. ir. S. Hamanza,	TU Delft, BioMorphic Intelligence Lab
	Ir. D. Benders,	TU Delft, Cognitive Robotics, drone specialist

TU Delft

Preface

Welcome to my thesis project about flying small drones autonomously in the rainforest of Singapore. I was fortunate enough to be asked to assist team Biodivix, a team from TU Delft and multiple other universities, in building an autonomy system for a competition in which the rainforest's biodiversity has to be measured.

Looking back, creating single-handed an autonomous drone capable of executing missions in the rainforest was an ambiguous task. Although the developed drone did not function properly in Singapore, the results of this project are still impressive. Moreover, I learned many things, not only about drone autonomy, but also about testing and debugging real-world missions in the wild. I took the cover photo of this report bu myself by another drone in Singapore. This photo emphasizes how wild and diverse the Singapore Rainforest is.

I am grateful for the opportunity to have gone to Singapore for this competition. In this competition setup, I was able to see and learn many things about data collection and visualization within a multidisciplinary team of roboticists, biologists, and data analysts.

I want to thank Martijn, my supervisor from Robotics, for his support and feedback in the structure and reports during the project. I would also like to thank Salua, my supervisor at the MAVlab, for her inspiration and guidance on this topic, and for arranging the competition in Singapore. Also, I would like to thank Liming and Seamus, fellow students with whom I worked on the competition, for their support and pleasant company in the lab and in Singapore.

Lastly, I would like to express my sincere gratitude to Erik, the drone specialist at MAVlab, for his unwavering support throughout the project. He assisted me with materials, software issues, supervising outdoor testing, and provided valuable insights for the drone competition. Erik is also a source of inspiration for me in terms of my future aspirations.

I have written this report to help you understand the methods used to obtain the results. My intention is to present technical details of the methods and results, while providing enough background information to make it understandable for those who are technically proficient but not necessarily experts in robotics. I acknowledge the use of AI in this work. But only to refine text to be clear and coherent, not to create content.

A. Zwanenburg
Delft, January 2024

Abstract

Tropical rainforests, facing imminent threats from deforestation, species extinction, and climate change, demand effective monitoring solutions. Automated robots, particularly drones, hold promise for efficiently covering expansive and remote rainforest terrain. This study addresses that challenge by developing a lightweight autonomy system for rainforest exploration, with a focus on the Rainforest Competition by the Xprize Foundation.

A comprehensive review of competition rules, challenges, and existing literature on autonomous drone navigation creates the selection of a dead-reckoning planning-based navigation algorithm. The report shows the subsequent design, construction, and testing of the drone, emphasizing lightweight engineering for prolonged flight times. Observations from the rainforest competition provide context for an in-depth investigation into downscaling autonomous navigation and evaluating performance and safety through simulations based on real forest flights.

The study contributes to autonomous drone navigation in dense and cluttered environments, such as rainforests, filling a gap in existing literature. The implemented dead-reckoning method showcases operational efficiency, offering insights for future missions. Research shows the impact of downscaled autonomy on performance, revealing a tradeoff between processor utilization, obstacle map resolution, and the resulting navigable gap size. Advancements in downsizing enhance exploration effectiveness, but limitations like reduced flight speed are acknowledged.

The research highlights the potential for cost-effective drones and extended flight times, emphasizing the need for future refinement in practical applications for rainforest environments. This work lays the groundwork for continued improvements in downscaling autonomous drone systems, contributing to both technological capabilities for autonomous flight and environmental conservation efforts for rainforests.

Key findings indicate that safe navigation is achievable through dead-reckoning navigation for downscaled autonomous drone navigation with path planning. Increasing obstacle mapping resolution most effectively reduces processor and memory loads but compromises the drone's performance in navigating through narrow gaps.

Contents

Nomenclature

BVLOS	Beyond Visual Line of Sight
CAAS	Civil Aviation Authority of Singapore
CAD	Computer Aided Design
CPU	Central Processing Unit / processor
CTR	Control (air)Traffic Zone
DNN	Deep Neural Network
eDNA	environmental DNA
EMC	Electromagnetic Compliance
ENU	East, North, Up coordinate system
GPS	Global Position System
IMU	Internal Measuring Unit
LLA	Latitude, Longitude, Altitude coordinate system
MAV	Micro Air Vehicle
NED	North, East, Down coordinate system
NN	Neural Network
QP	Quadratic program
RC	Radio Controller
ROS	Robot Operating System
RRT	Rapidly-exploring Random Tree
RVIZ	ROS Visualiser
SLAM	Simultaneous Localisation And Mapping
SSH	Secure Shell, communicate protocol
VINS	Visual Inertial location estimator
VPN	Virtual Private Network

List of Figures

List of Tables

1 Introduction

Tropical rainforests are vital ecosystems that face numerous threats, including deforestation, species extinction, and the consequences of climate change. Protecting these indispensable ecosystems is crucial for global well-being. Measuring biodiversity in these environments is essential for understanding and preserving their delicate balance. However, monitoring rainforests' expansive and rugged terrain is a challenging task. The dense vegetation and complex topography present significant obstacles to researchers and conservationists seeking to gather comprehensive data on the diverse flora and fauna within these ecosystems.

Robots, specifically drones, have emerged as a promising solution to cover extensive and remote areas efficiently. While aerial imagery captures valuable insights from above, the dense canopy layer conceals the majority of biodiversity. Navigating inside or beneath this layer poses a significant challenge due to the dense vegetation. Drones equipped to explore and measure the diverse environment under the canopy must navigate adeptly without mishaps, given the challenging recovery options in these remote areas.

This report details a thesis project that aimed to develop a lightweight quadrotor autonomy system for safely navigating densely cluttered forest environments. The project involved creating a drone for measuring biodiversity in the rainforest of Singapore, where researchers face numerous challenges when trying to collect data. The necessity for these robots is underscored by the Xprize Foundation, which created the Rainforest Competition in Singapore to assess biodiversity in remote rainforest areas. Using small autonomous systems for rainforest navigation is crucial for improving exploration efficiency. Previous studies have shown adequate results in autonomous flights in forest conditions. However, limited research exists on autonomous flight in rainforests, which has its unique complexities.

The primary focus of this research is to develop a lightweight autonomy system capable of safe navigation in rainforest environments with complete onboard processing. A more lightweight system can either work on the same drone and prolong the flight time or be built on a smaller drone system to be safer and more cost-effective. The project involved in-depth research on downscaling the autonomous navigation system on a drone, specifically in rainforest conditions.

In the following chapters, this report will delve into the background and related work, the challenges encountered in rainforest environments, the methods used in designing and testing the autonomous drone, downscaling the autonomous navigation, and the conclusions drawn from the project. Through this comprehensive exploration, the report aims to provide valuable insights for future researchers and engineers seeking to deploy robotics in challenging natural settings, such as tropical rainforests. By providing more context on the importance of measuring biodiversity in the rainforest and the challenges researchers face, this report aims to help readers understand the significance of the work done in this project.

2
Background & related work

To set the stage for this research, a thorough literature review was conducted, aiming to clarify the context of the research question. This involved examining publications related to the research topic and summarizing the key findings. Delving into the background of the research question was particularly crucial to meet the specific requirements of the rainforest competition. Understanding the competition environment, requirements, and team objectives was essential in shaping the research question.

First, the objectives and requirements are elaborated upon. Then the following sections give more background information on these objectives and requirements. The background and related work research present the best localisation and navigation methods and what challenges to expect in the rainforest. The primary goal of the literature review is to ensure that the project builds upon and exploits state-of-the-art technologies, preventing duplication of existing work. The chapter concludes with a brief summary, highlighting the most promising methods for this project's work. For an in-depth exploration of the literature reviewed, please refer to the complete and comprehensive literature report [1].

2.1. Objectives and requirements

To elaborate on the objectives and requirements mentioned in the introduction, this section details the objectives, hypothesis, requirements and constraints for this project.

Objective and hypothesis

The objective is to create an autonomous system which is lightweight, can navigate safely, and fits the Xprize Rainforest challenge requirements. This system must be computationally efficient as it will have limited computational resources. The goal is to find relations between flight performance and system downscaling to make the system lightweight and computationally efficient. For this downscaling it is taken into account the system must navigate safely through dense, cluttered environments, finding paths through narrow gaps.

Based on previous work, the hypothesis is for the drone to fly with an average speed of 1m/s through dense vegetation. Also, the drone is expected to find a route through the canopy where the gap is 70cm or wider. Examples from previous work show that a flight time of approximately 15 minutes is the maximum one can get with a drone containing all resources required to perform the required task.

Requirements and constraints with planning-based navigation

To create a solution for autonomously guiding a drone through a rainforest, some basic assumptions and limitations are defined to address the challenge effectively.

A crucial requirement is to ensure the drone's avoidance of crashes. A crash significantly negatively impacts the team's score and entails retrieving the drone only after 24 hours of competition. Crashing potentially leads to a complete system loss due to exposure to rain, dirt, or a significant fall. Concerning trajectory planning, all obstacles within the rainforest are assumed to be stationary. This simplification

reduces the complexity of planning the drone's path, consequently lowering the computational workload.
Another requirement is that the drone is capable of running all computations onboard. Wireless communication with the operator is only for supervising the flight. The drone should be able to continue operations even when wireless communication drops out.

2.2. Background Xprize Rainforest Competition

This section addresses the complexities of autonomous navigation for drones within the context of the Xprize competition [2]. It provides a comprehensive overview of the specific constraints imposed by the Xprize challenge, delineates the tasks expected of the drone, and elucidates the key challenges associated with drone navigation in this unique scenario.

The primary objective of the Xprize challenge is to evaluate biodiversity in a rainforest environment, necessitating the detection of various species across different forest layers—ground, understory, canopy, and emergent layers. These layers are illustrated in figure 2.1. Teams must deploy robots, including drones or ground robots, to collect data remotely, preferably autonomously, due to restricted access to the forest area [3]. The competition covers a 100-hectare area, launching robots from a small jungle hut, and spans 24 hours for exploration and data collection, followed by 48 hours for data processing and result publication. Because of the fixed competition date, teams have to cope with all weather conditions.

The drone developed in this project's final goal is exploring the rainforest and collecting image data via its cameras to identify plant species. In the competition, dense vegetation compromises GPS reliability under the canopy; this necessitates the integration of an alternative localization method. This chapter underscores the critical issue of determining a suitable localization method in a rainforest environment lacking GPS functionality.

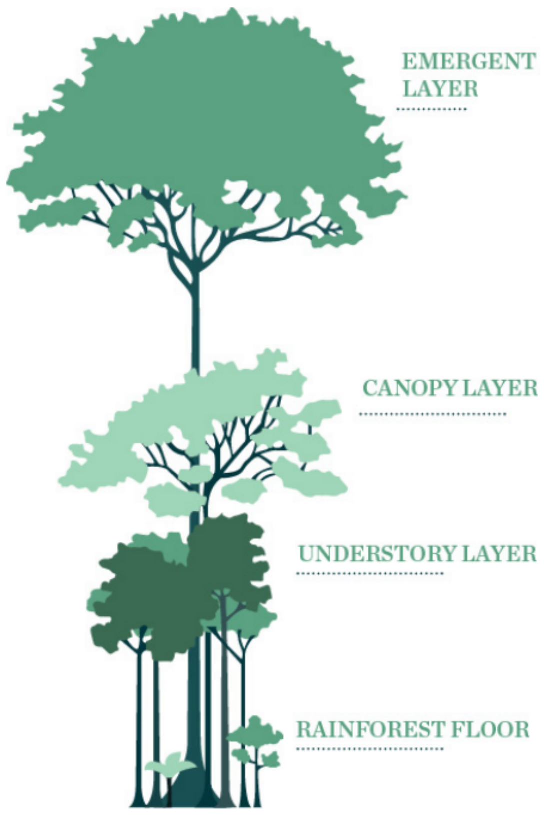


Figure 2.1: Rainforest vegetation layer categories, from competition guidelines [3]

Given the weak and interrupted signals while flying through the forest's understory layer, and the impracticality of remote human piloting, the drone is designed to navigate autonomously. Three primary navigation methods found in the literature research for autonomous navigation are reactive-based, neural-network-based, and planning-based navigation.

As reasoned in the literature research, planning-based navigation emerges as the most reliable solution for navigating dense vegetation. It demonstrates a high success rate in finding a path solution in complex environments and a safe method to avoid collisions—a crucial aspect in the Xprize competition. However, this method demands significantly higher computation compared to neural-network-trained and reactive-based navigation.

While neural-network-based navigation can facilitate agile and fast drone movement through complex environments [4] [5], its efficacy relies on substantial training data from the specific environment, which is unavailable in this project. Moreover, its success rate of avoiding collisions is around 80%, falling short of the competition standards.

Reactive-based navigation, like planning-based navigation, exhibits reliable obstacle avoidance [6]. Related work indicates its efficacy in spacious environments with relatively lightweight sensor and processor hardware solutions. However, as environments become denser and more cluttered, this method faces challenges in finding solutions to traverse further into the forest.

To navigate the understory and canopy layers of the forest with planning-based navigation, specific system hardware and software components are required. Most related work structures their algorithms around two main components: mapping of obstacles and path planning. Obstacles are detected using lidar or depth cameras, and an onboard processor computes collision-free paths. The calculated desired path is then sent to the drone flight controller.

In the subsequent sections, related work on mapping and trajectory planning is presented and compared. To comprehend the most relevant features, the specific requirements and constraints for the drone in the competition are summarized first.

Conclusion
In conclusion, for the competition, the drone is required to fly under the canopy layer of the rainforest. Flying under the canopy brings its challenges, such as a dense and cluttered environment in which navigation needs to find small openings to traverse. Also, under the canopy, the GPS signal is blocked; thus, another localisation method has to be found. Based on previous works, localisation and navigation will most likely be done with Simultaneous Localisation And Mapping, and with trajectory planning. The objective of this project is to create an autonomous system that navigates safely through dense, cluttered environments, finding paths through narrow gaps. Due to unreliable signals in forests, the drone has to compute all autonomy onboard, and for simplicity, all obstacles are assumed to be static.

2.3. Known challenges for navigation in the rainforest

From related work found in the literature study, some challenging conditions are worth noticing because these might influence mission performances during testing and the rainforest competition. This section addresses the relevant challenges described in the literature.

The main challenges highlighted by found literature are poor signal qualities and the cluttered environment [7] [8] [9]. Poor signal qualities are caused by rapidly degrading radio signals due to humid air and plants with high water content. The water and moisture adsorb, scatter, and reflect radio signals. Therefore, GPS signals are scattered and damped, resulting in poor or no reception. This phenomenon also negatively impacts the signals between the GCS and the drone to send control inputs or receive telemetry and video data from the drone.

The rainforest environment is can be extremely cluttered, especially the ground and the area up to 5 meters in height. When attempting to navigate a path to reach a destination, the drone must locate narrow openings to pass through. There is also a possibility that the drone may encounter consecutive gaps, only to realize it has reached a dead end. In such a cluttered environment, obstacles can also be difficult to detect since they can be of any color, size, or shape. Tiny objects or objects with low contrast might not be detected by the depth camera, leading the drone to follow an inefficient path. Moreover, if the object is detected too late or not at all, the drone might hit the object and crash. The two primary challenges are blocked GPS signals and the presence of cluttered objects.

2.4. Localisation

Effective navigation relies on trajectory planners utilizing obstacle information to compute collision-free paths in unmapped forests, crucial for drones [10]. Mapping involves accurately placing 3D scans (lidar or a 3D camera) in an accumulating map, particularly challenging in areas with limited GPS reception [11, 12, 13]. What localisation methods are suitable for the drone in this project, and what would be the best method?

Simultaneous Localization and Mapping
Simultaneous Localization and Mapping (SLAM) integrates mapping and localization, estimating the drone's position by comparing obstacle maps with recent observations, contributing to continuous mapping [14]. In SLAM research, three primary methods are identified: RGB-D data compares depth observations with the map; Visual SLAM tracks landmarks in images, estimating the camera's trajectory; Visual-inertial-SLAM combines camera and IMU data for robust tracking, adaptable to various cameras [15].

The optimal combination for this project remains undetermined. While color/stereo cameras may enhance tracking precision, the risk of processor overload leading to errors must be considered. Incorporating IMU data reduces workload, aiding feature tracking without significant impact. RGB-D data significantly increases computational demands, involving an extensive search to align depth observations with the 3D map through numerous iterations and minimization of Euclidean distances.

In conclusion, in the rainforest competition, visual-inertial-SLAM appears promising. However, every 3D SLAM method appears to have a high computational demand on the processor. Whether the processor can handle visual-inertial-SLAM has to be tested. The potential consequences of processor overload, which leads to position estimation errors, highlight this challenge's significance.

Dead reckoning localisation
As an alternative localisation method, dead reckoning might be an option that requires less computational power. In navigation, dead reckoning involves determining the current location of a moving object based on a previously known position (fix) and considering estimates of speed, direction, and time elapsed [16]. Is this method suitable for autonomous navigation in rainforest conditions on a processor with limited resources?

This method is susceptible to accumulating errors. Modern advancements in navigation, such as accurate position information from systems like the Global Positioning System (GPS), have largely rendered traditional human dead reckoning obsolete for most practical purposes. Nevertheless, inertial navigation systems, leveraging precise directional data, continue to utilize dead reckoning and find extensive applications.

2.5. Trajectory planning

Based on the found literature, three categories of trajectory planning methods are compared. These are random-tree search, gradient-based navigation, and trajectory sampling. These are based on the work of FASTER-planner [17], EGO-planner [18], SWARM-WILD [19], and REAL-planner [20]. A one-page description of each algorithm from the literature studies is added in Appendix A The comparison considers computational efficiency, robustness, and safety in forest conditions based on information from the literature. This section outlines which method is best for navigating rainforest environments where safe navigation by small processors and sensors is required.

Computational Efficiency
To provide context for the relative computational efficiency, let's examine the operational mechanisms of these trajectory planning methods.

REAL-planner excels in computational efficiency by employing a hybrid approach combining sampling-based and optimization-based techniques. This minimizes the computational load, producing smooth trajectories without additional optimization steps.
In contrast, both EGO-planner and FASTER-planner demonstrate slightly lower computational efficiency. EGO-planner uses Bayesian optimization for global optimization, while FASTER-planner introduces a dependency on roadmap connectivity. The latter may require more iterations in low-connectivity scenarios, impacting computational demands.

Robustness in the rainforest
Robustness in a rainforest environment depends on factors like complexity and density. Therefore, the distinctive characteristics of each method's approach are investigated.
EGO-planner uses Bayesian optimization, potentially showcasing robustness in rainforest environments. FASTER-planner employs a receding horizon planning method, offering potential robust performance. REAL-planner, while highly efficient, may face challenges in rainforest settings due to its sampling-based optimization approach.

Safety in Rainforest Conditions
Examining the unique safety considerations of each approach, EGO-planner's Bayesian optimization offers robust safety. FASTER-planner combines global and local optimization, holding promise for safety. REAL-planner's safety performance may face challenges due to its reliance on sampling in cluttered environments.

Conclusions
In conclusion, among the trajectory planning methods assessed, EGO-planner appears strong, emphasizing safety through Bayesian optimization, making it particularly suitable for rainforest environments. While REAL-planner excels in efficiency, its reliance on sampling may pose challenges in cluttered environments. FASTER-planner, though promising, depends on factors like roadmap quality. The emphasis on safety through Bayesian optimization makes EGO-planner the preferred option for navigating challenging and densely vegetated environments. Further empirical testing in such conditions is essential to confirm its performance.

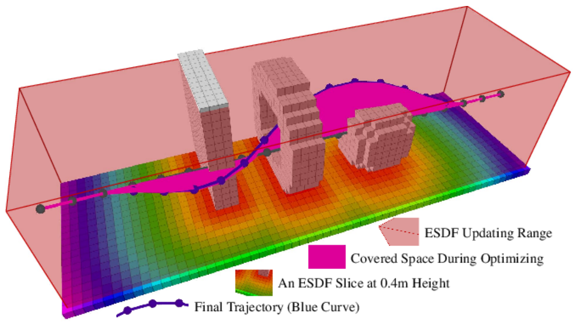

ESDF Updating Range
Covered Space During Optimizing
An ESDF Slice at 0.4m Height
Final Trajectory (Blue Curve)
Figure 2.2: Illustration of EGO-planner functionality [18].

Figure 2.2 shows an illustration of the EGO-planner. In this illustration by [18], the gradient fields of voxel map obstacles are visualised, and how the planner re-routes the path to find a jerk-optimised obstacle-free trajectory.

2.6. Conclusions from literature review

After reviewing the available literature, some conclusions can be drawn that support further research on autonomous quadrotor navigation in the rainforest.
To begin with, various algorithms exist for SLAM (simultaneous localisation and mapping). This method is mostly applied in GPS-denied environments. While drones commonly use this method to estimate pose and position in 3D space, it requires a powerful processor board and cannot easily run on an embedded chip in real-time.

Furthermore, multiple methods exist for trajectory planning that plans acceleration or jerk-optimised trajectories in 3D for quadrotors. Among the found path planners, the EGO planner appears to be the most suitable algorithm for generating safe trajectories in dense and cluttered environments on a small processor, based on its paper description.
For estimating pose and position, the accumulative estimate of the IMU sensor could be used, also known as dead-reckoning navigation. As every trajectory planning method relies on a position estimate, and SLAM would be too computationally intensive, this might offer a solution. Although this method is known for its drift over time, testing can determine whether this causes significant problems or not.

Finally, little literature describes the challenges of a rainforest environment, but we can summarize two main takeaways. The forest is a dense, unstructured space where obstacles can occur at any location and can also be any size and shape. Additionally, the GPS position cannot be used as the primary position estimate because the GPS signal is poor or unavailable.

3
System integration

An autonomous quadcopter drone has to be built and programmed to navigate through complex tropical rainforest environments. From examples and suggestions in the literature and the competition requirements, a perception is created of what modules are required for such a system and how this should work efficiently as a whole. From this perception, together with some try-and-error iterations, a system that fulfils the autonomous obstacle avoidance task in foresty environments is created. A note to make is that in literature, the state-of-the-art solutions used multiple expensive parts like high-precision flight controllers and processor boards that fell outside the budget for this project. Therefore, for the realisation of this drone, lower-performing parts are chosen that were either available in the lab, or inside the budget to purchase.

What does the created design look like? In this chapter, the details on component choice, CAD design, and assembly are given on how the hardware is built as lightweight as possible. After that, the software for autonomous navigation is discussed, which the drone uses to traverse autonomously through unknown rainforest environments. Finally, there is a list of ten extra flight systems that are built for outdoor missions in the competition. These are all flight systems not directly related to obstacle avoidance but with connectivity, control, mission planning, safety features, and debugging.

3.1. Hardware component selection

In this section, the selection of hardware components is explained. Every part is carefully chosen based on its weight and energy consumption to maximise the flight time on a single battery charge. First, the drone platform and processor are discussed on which all hardware and software is built in this project. Thereafter, the choice of sensors used for autonomous navigation is highlighted. In the sensor choices, the additional hardware required to fly missions in the rainforest is explained.

3.1.1. Drone platform

The initial goal of this project was to implement and test the autonomous navigation on a drone also used in another project. In the other project, the drone autonomously has to detect and perche on a randomly suitable branch in the rainforest. The combined autonomous navigation and perching would result in a fully autonomous system that explores the rainforest, chooses a branch, perches on the branch, records data, and returns home. Due to time and complexity constraints, autonomous navigation and perching have been developed separately. As the drone platform for the perching was not ideal for early-stage testing and had low availability, another drone platform was chosen to implement autonomous navigation. This section answers the question of which platform is the best choice within a set of available platform options and for a set of requirements.

The aspects of safety, weight, robustness, control ability, and flight time must be considered when choosing a suitable drone platform for implementing and testing autonomous navigation. Safety, or in inherent safety, is important to consider as testing operators include many close encounters, possibly with some collisions. Preferably, the collision damage with obstacles is minimal. Also, in some testing conditions, the human pilot/supervisor is not in a protected area. Therefore, if the drone platform is inherently safe, injuries to the pilot are minimal. The drone's weight has a big influence on the collision impact as propellers are stronger, bigger, and spin faster on a more heavy drone. Therefore, safety increases when the weight of the drone is minimised. Also, to follow the same objective as the autonomous perching project regarding drone size, the weight should stay below 800 grams.

A robust drone platform is preferred when collisions are likely to occur during testing. Higher impact resistance reduces downtime during testing. For example, when a drone breaks something on average after 10 crashes, instead of after 2 crashes, testing can continue 5 times longer before having to leave to the workshop. This increased robustness also increases progress on the project as, otherwise, repairs generally take a lot of time and interrupt the testing sessions. Even greater so when the testing session is at a remote place, for example, in a forest, where travel time and costs are significantly large compared to the tests.

Additionally, it's essential to have the ability to control the drone using software that's either provided through a Software Development Kit (SDK) or open-source. This is necessary to allow the external autonomy system to guide the drone's movements. Control commands and feedback to external systems can vary per system. How agile and accurate the system is can, in part, be influenced by the available control choices, the feedback it receives, and how quickly it communicates. Agility and accuracy contribute to higher flight speeds in complex terrain. Therefore, the drone can cover more distance in the same amount of time.

Lastly, maximising flight time will improve the distance that can be covered during exploration, which is primarily important for the Rainforest competition. The drone has to fly with the additional payload of the autonomous system and its energy consumption.

Three options are considered when choosing a drone platform. These three options were available in the lab, can carry the payload, and weigh below 800 grams. These are a PX4-based 5" quadcopter and the Parrot Bebop 2 with factory or Paparazzi UAV software.

Speedybee drone
The considered PX4-based quadcopter includes a 5-inch Speedybee frame and a Pixracer pro flight controller. This is the drone setup used in the perching project. This drone has more powerful engines, designed for drone racing or freestyle flight. The drone size is 24x24x7cm, and without accu, it weighs 400 grams. The accu size is free to choose, but would typically weigh 200 grams. Flight time without extra payload is 8 to 13 minutes. This Pixracer flight controller has the ability to communicate and be controlled by an external processor via MAVlink protocol. The drone can be configured with many different parts as this is a do-it-yourself design. The configuration as is available in the lab with the frame, motors, flight controller, propellers, and 1 battery is around 500 euros. Spare parts are available online. In the lab, not many spare parts are available, except for spare propellors.

Parrot Bebop 2 drone
The Bebop 2 drone, released in 2015, has been used in the MAVlab since 2017 and is known for its robustness, stability, and decent 20 minutes of flight time. In the lab, every spare part of the drone is available in case repairs are required, and all parts are accessible for replacement. Also, the drone has a nifty body design where flexion of the body absorbs the impact of a crash without breaking. The only parts that breaks frequently are the plastic one-euro propellers, absorbing the impact of a crash.

Further, the drone is not only well-stabilised by mechanical and control-loop design, but it can also hover still without drifting. By the use of a low-resolution bottom camera, the drone can hover without drifting by using optic flow [21]. Two conditions that have to be met are a ground surface with enough colour texture and sufficient light. There are not many off-the-shelf drones available comparable to the size (30x30x10cm), weight (400gram), flight time (20-25 minutes), and have at the same time some onboard Software-Development-Kit (SDK) options. And neither of the drones that come close to these specifications can compete with the 300 euro price (only available on the second-hand market nowadays).

The two platform comparisons are summarized as scores from 1 (worst) to 5 (best) in table 3.1 on the right. This comparison shows that the Bebop 2 platform would be a better choice compared to the Speedybee based on the aspects compared. The only aspect where the Speedybee outperforms the Bebop 2 is control availability because the Bebop components cannot be modified or reconfigured, and the commands to the flight controller are more restricted.

Table 3.1: Comparison Speedybee and Bebop 2 drone platform. score 1 (worst) to 5(best)

Aspect	**Speedybee**	**Parrot Bebop 2**
Safety	1	4
Weight	2	3
Robustness	3	4
Control Ability	4	2
Flight Time	2	5
Availability and Price	4	4

Thus, the Bebop 2 is the better choice as drone platform. It has a significantly longer flight time, better off-the-shelf stability, is safer, and is available in the lab with many spare parts.

3.1.2. Processor board

To read all sensor inputs, map the environment, generate the trajectory, and send commands to the drone, an onboard processor board is required. As the autonomy systems have to be run completely onboard the drone, the processor board has to be able to process all data and send outputs at sufficient loop rates.

Figure 3.1: Odroid XU4 [22]

In literature, most often, a processor board of the Nvidia Jetson series is chosen as these have the most processing capability on an embedded system that fits on a small drone. The processor itself is 24 grams and consumes 7.5 to 15 watts. The downside is that it requires cooling and a carrier board, making the complete set heavy (130 grams) and voluminous (100*80*29 mm) for a small drone. The Nvidia Jetson board (TX2, as [18]) with cooling and carrier board cost about 800 euros, which does not fit the budget.

For these reasons, a less powerful processor board was chosen, the Odroid XU4, shown in figure 3.1. This board has results comparable to the Raspberry Pi 4B (Rpi4B), and multiple were available in the lab for the project, unlike the Rpi4B due to the chip shortage. This board is half the volume of the Nvidia Jetson board (83*58*20 mm), and also less than half the weight (60 grams). Thereby, weight is significantly reduced, and the drone has increased flight time. This board costs 80 euros, which is 10% of the Jetson board. Multiple of these boards were available in the lab, which is helpful for the challenge as spare parts for every part were not a must but highly recommended.

3.1.3. Sensors and communication

Examples from the literature are used to select the appropriate sensors and communications for autonomous navigation. Typically, GPS, IMU, and a depth camera sensors are used to achieve robust autonomous navigation on a drone platform. Since the drone already contains several sensors like IMU, barometer, and GPS, these can be utilized for autonomous navigation.
This section highlights the considerations for different depth cameras. In addition, to support flight missions in the wild, extra modules are required onboard the drone for receiving and transmitting signals. These modules are an RC receiver, a 4G dongle, and a secondary GPS module. In the lightweight module for the drone, these components are carefully chosen regarding size, weight, energy consumption, and signal strength.

Depth camera

Further, a depth camera is required for the chosen path planner, which will be discussed in section 3.3.3. By the authors of the used path-planner [18], and the rest of the top three literature sources mentioned in chapter 2, the Realsense D435i depth camera [23] is used. This camera is popular in research projects as it has compatible drivers in ROS, the robot software that will be used. The 72-gram camera is heavy for a small drone, but as smaller cameras with the same accuracy, robustness, and range are not available, the D435i is still the preferred choice.

In this project, the D435i camera will be used. But, as the authors of EGO-planner did, the camera will be stripped of all heavy, unnecessary components to reduce the weight to about 30 grams. Heavy components removed from the camera are the metal housing and frame that functions as a heatsink. A 3D-printed frame replaces these components to keep all camera parts together and give some protection. As the initial frame also functioned as a heatsink for electronic components, smaller heatsinks are placed on the specific components (chips) that generate heat during operation.

The Arducam TOF depth camera [24] was also considered. This camera has a weight of only 8 grams and a compact size of 40x40x5mm, which is 9 times lighter than the Realsense Camera. However, its maximum range is only 4 meters, which is half of the range of the Realsense camera. Despite the limited range, this camera is an attractive option due to its lightweight design. Unfortunately, the camera drivers for Linux have compatibility issues and are only available for RaspberryPi-OS Bullseye. ROS-1 cannot be installed on this operating system, making it impossible to use one of the ROS packages for autonomous navigation. Due to limited time and uncertainty about how well this camera would perform for drone navigation, we did not attempt to solve the driver compatibility issues and instead chose the Realsense camera.

GPS
In Chapter 2, it was found that the forest hinders GPS reception, and the drone's 2015 internal M8 GPS is outdated. To address this, a lightweight HGLRC M100 GPS (weighing 2.4 grams) with a UBlox-M10 chip was chosen.

The M100 GPS outperforms the previous version in several ways. Its 3dB higher antenna gain allows operation in weaker signal conditions. It tracks four satellite constellations (GPS, Galileo, GLONASS, and BeiDou), automatically selecting the best in view. The M10 chip consumes five times less energy, and improves true position error confidence by 95% compared to the M8 series.

To optimise reception, the additional GPS antenna is placed atop the hardware, ensuring good reception from all sides. This is crucial since the internal GPS, positioned in the nose above propeller height, may face signal blockage from added autonomy hardware.

RC receiver
The RC receiver reads signals from the pilot's remote controller, which is used to fly the drone manually and switch to autonomous mode. This RC receiver has to be lightweight and compact as it has to be carried onboard the drone. Also, the receiver has to receive weak signals when the RC is further away, preferably as far as possible.

The FrSky R-XSR is chosen as the receiver, only weighing 1.5 grams. This component is the smallest 16-channel receiver compatible with the SBUS protocol used on the available RC, the Radiomaster TX12. This module also supports telemetry being sent back to the RC from the drone. But, this is not used as telemetry is monitored via 4G.

4G module
To establish a 4G connection, a module is essential to link the onboard processor with cell towers. Three options were considered to connect the onboard processor to cell towers: a 4G USB dongle, an LTE modem, and a PCB 4G module.
The preferred option is to use a 4G USB dongle, connecting the processor board to the internet via the USB port. Despite being larger and heavier than the serial PCB, the USB dongle is more compact and lightweight than the ethernet modem. Additionally, the bandwidth over the USB protocol is significantly superior to that over a serial port by the PCB module.

The Huawei E3372-325 4G LTE USB dongle is selected for 4G communication on the drone. This 58-gram module, though relatively heavy, proves to be the most robust solution, combining compactness, high signal strength, and compatibility with the onboard system. The dongle, twice the size of a USB drive, comes with an internal antenna and accommodates a standard-size sim card. Optionally, the dongle can be equipped with two external 3DBi 3G/4G LTE antennas for enhanced signal strength, each weighing an additional 6 grams.

For debugging, indoor, or with close-range testing, the 4G dongle can be substituted with a WiFi dongle. This 20-gram Wi-Fi-module is significantly lighter than the 4G dongle but only gives a stable connection upto about 20 meters.

3.2. Hardware Autonomy backpack design

This section discusses the hardware design that facilitates autonomy and mission processing. The design integrates all components from the previous section into one device, named the 'autonomy backpack', which snaps on top of the Bebop drone. Multiple 3D-printed parts are designed to integrate all components.

When selecting the hardware and designing the 3D-printed parts, each component is placed on a weigh scale to optimize for low weight. A weight budget table is created, which lists the weight of every part added to the system. You can see the weight budget table in 3.2 below. The total weight of the autonomy backpack is 202 grams. As an indication, the new-price of the used components is also listed. Most components were used from the lab, but the components' total price would be around 432 euros. For cheap components, like nuts, bolts, and zipties, a minimum price of 2 euros is calculated in the table.

Although the table also mentions the price and weight of the Bebop drone and its battery, these are not included in the calculations for total price and weight because these are considered not to be part of the autonomy backpack. The table indicates which components were fixed and which had design freedom, providing insights into the system's design flexibility.
The upcoming sections will discuss CAD design, 3D printing, and electrical design.
The following section presents how all these components are integrated into one module that can easily be mounted and unmounted from the drone. Thereafter, the wiring of all electronics within the module is discussed. Also, a table with an overview of all components, weights, and costs is given. Then, the software design for the autonomy system is discussed for navigation and, additionally, the software required for outdoor missions in the competition.

3.2.1. CAD design & 3D printing

All the chosen hardware components must be put together in the drone. So, a 'backpack design' is created to neatly attach them to the drone. This design strategically places components to optimize their performance, considering dimensions and weight distribution. The main structure of this design is 3D printed, ensuring a lightweight and secure housing for all components on the drone.

The backpack module is designed to let the drone fly backwards. The drone's built-in sensors, like the camera, IMU, compass, and GPS, are located at the front. To minimize interference with the compass and GPS, the backpack is on the drone's back on top of the battery. The camera faces backwards to avoid blocking its line of sight by parts of the drone body. Consequently, the autonomous system is programmed to navigate backwards. Beyond aerodynamics, which isn't crucial at expected flight speeds, a quadrotor's flight direction is assumed to not notably affect stability or efficiency.

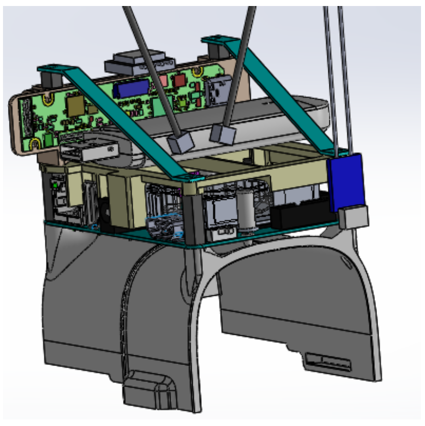


Figure 3.2: 3D model of autonomy backpack

Components are placed close to the drone's centre to maintain stability and energy efficiency. Reducing the moment of inertia makes the drone respond quicker and use less

energy in roll, pitch, and yaw. A lightweight design reduces
inertial factors and lowers the thrust needed for the drone to hover, significantly increasing flight time. According to literature sources, implementing autonomy on a quadrotor that weighs less than 1 kg could potentially reduce flight time by half or one-third. This suggests that even a small weight reduction of 10% could result in a significant increase in flight time, up to 20-30%.

The 3D-printed structure is lightweight due to specific design choices that were made. The walls are thin, and ridges and profiles add stiffness and durability. Components are mounted with minimal supporting structure, minimizing the need for fasteners. The used fasteners are plastic. All components except the antennas are enclosed by the frame, which absorbs impacts during crashes, reducing repair time and costs. Antennas are placed partly outside the frame for clear transmission. They are flexible and assumed not to pose significant risks in a crash.

Considering the possibility of rain during the rainforest competition, the drone might encounter water droplets from trees. While it is not designed to fly in the rain, waterproofing is necessary to protect electronics in the backpack from water above. To keep the waterproof cover lightweight and compact, the backpack design's polygon shape must be compact. USB cables connecting components are shown to be the limiting factor for compactness, and a solution is presented in the following section.
In Figure 3.2 above, the autonomy backpack's 3D CAD model is shown. Apart from the cables, this mechanical design includes all components that will be mounted on top of the drone.

To give a more detailed view of how and where all components are mounted in the design, Figure 3.3 shows an exploded view. The components are given unique colours for the exploded view to make the parts distinguishable. The relevant components are annotated in the exploded view as:

1. Mounting for backpack on drone - 3D printed
2. Processor board - Odroid XU4
3. Frame for electronics - 3D printed
4. Frame for Depth camera - 3D printed
5. Stereo Camera - Depth module Realsense D430
6. Depth processor - Realsense vision processor D4
7. GPS module - HGLRC M100 (Ublox M10)
8. USB 4G dongle - Huawei E3372
9. 4G antennas - 5dBi CRC9
10. RC-reciever - FrSky R-XSR
11. Electronics protection - 3D printed

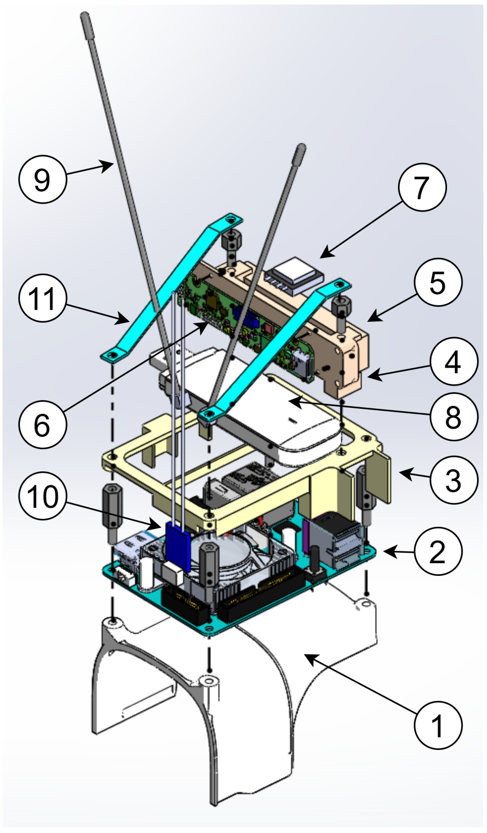


Figure 3.3: Exploded view of mechanical design Autonomy Backpack. With component annotations

Enlarged images with annotations of the mechanical design are included in appendix B.1. Table 3.2 below provides details on the type, weight, and costs of components integrated into the autonomy backpack. The autonomy backpack, including a 4G module for the rainforest mission, weighs 202 grams. The Bebop drone itself, with a battery, weighs 500 grams, and its stable payload capacity is 250 grams, which was tested. Above 250 grams of payload, the bebop can still deliver the total thrust to compensate for gravity, but motors start to saturate, resulting in loss of yaw/pitch/roll authority for position control. The 202 grams of the autonomy backpack remain within this limit. Without the 4G module, the backpack weighs 167 grams. The total weight, including the drone, is 702 grams with the 4G module and 667 grams without. The estimated cost for components and 3D prints is approximately 432 euros, rising to around 932 euros when the drone is included.

In comparison, the drone used in the EGO-planner paper [18] is estimated to cost 2000 to 3000 euros and estimated to weigh 800 to 1000 grams. This means the cost is factor 2-3 lower, and the weight is roughly reduced by 1/3rd in this first design step. In chapter 6, the weight and costs will be optimised to reduce these more.

Table 3.2: Weight and cost overview for autonomy backpack

Part	Type	Weight (grams)	Price (Euro)	Design variable	Note
Drone platform	Parrot Bebop 2	(259)	(500,-)	No	Not included in calculation of 'autonomy backpack'
Drone battery	Parrot Bebop 2	(240)	(60,-)	No	Not included in calculation of 'autonomy backpack'
Depth camera	Realsense D430	16	120,-	Yes	
Depth processor	Realsense vision processor D4	5	60,-	Yes	
Depth camera frame and cooling	3D printed and 10x10x5mm alu heatsinks	9	6,-	Yes	
Processor board	Odroid XU4	60	55,-	No	
Backpack mounting	3D printed	20	6,-	Yes	
Backpack frame electronics	3D printed	7	2,-	Yes	Combination of multiple parts
Backpack protections	3D printed	2	2,-		
Backpack bolts and nuts	plastic	4	2,-	Yes	
Backpack zipties	Plastic, 4.8mm	2	2,-	Yes	
4G module	Huawei E3372	35	60,-	Yes	Only used in Singapore
Wifi usb dongle	TL-WN725N	2	12,-	Yes	
4G antenna 2x	5dBi CRC9	10	16,-	Yes	
RC receiver	FrSky R-XSR	2	23,-	Yes	
Power converter 12v to 5v	BlueSky UBEC 5V 3A	6	7,-	Yes	
External GPS receiver	HGLRC M100	3	30,-	Yes	
Cable drone to processor	Custom soldered cable	8	8,-	Yes	
Cable camera to processor	8cm flatcable with 90deg usb plugs	5	15,-	Yes	
Cable 4G receiver to processor	Custom soldered cable	4	6,-	Yes	
	Total	**202**	**432**		

3.2.2. Electrical wiring

To facilitate power distribution to the drone's modules, a system of wiring is required. The modules have different connectors such as USB-A, micro-USB, Molex-JST, and more. Every connection requires a cable with the correct connector at both ends. Rare connector types or rare connector combinations within a cable limit the choice of standard cables. To minimise weight and make the backpack most compact, cables are preferably exactly the required length. Longer cables add extra weight, make the backpack more bulky, and increase interference with radio signals.

As standard commercially available cables with the correct combination of connectors on both ends are not found in the required lengths, longer cables are bought. These longer cables are cut to the required length and soldered back together. On the compact system with multiple wireless communications, EMC shielding is important to prevent signal interference. Therefore, all bought cables are shielded, and the shielding is restored when soldering to the correct lengths.

There are some exceptions where no standard cable is used. For connecting the 4G module, it was found that the connector of the standard cable did not fit well in the backpack design, making the complete backpack more voluminous and, therefore, harder to cover with waterproofing. So, a custom cable is soldered with pcb-type USB connectors and a shielded USB cable. This minimalist approach, often referred to as "bare-bone," is highly effective in achieving significant weight reduction. Additionally, this custom soldering process affords the flexibility to create angled connectors, ensuring that the cables do not protrude sideways, thereby allowing for a more compact assembly that is better suited for applying a waterproofing cover.
Figure 3.4 provides a compact overview of the wiring diagram and connection scheme. In appendix B.2, the full-scale diagram and scheme are shown in more detail. In the wiring scheme, all modules connected to the processor board are shown with the proper colour coding. The camera, 4G module,

and drone connect to the board by USB standards. These modules also receive their power supply via USB and have no additional connections. For simplicity, the USB modules are not shown in the diagram. The diagram illustrates the distribution of power to the various modules via the board, using black and red cables. The board powers itself via the drone battery through a step-down regulator, which converts the 11.1-volt DC input to a stabilized 5-volt DC output.

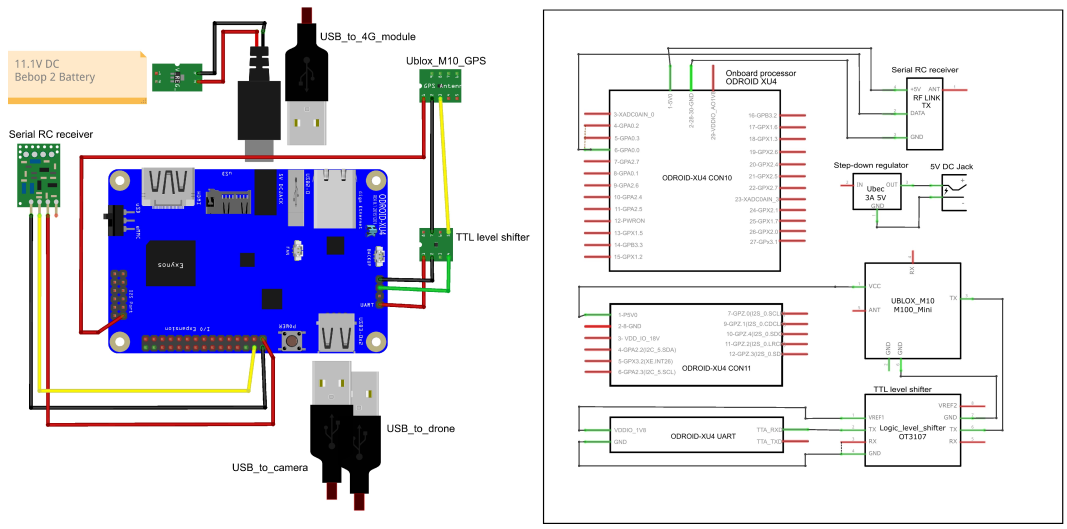


Figure 3.4: Compact version wiring diagram and connection scheme Odroid XU4

The radio-controller (RC) receiver connects to a serial port at pin 6 (on the CON10 header). The GPS receiver, on the other hand, requires a TTL level-shifter module to convert its 5V serial output to the 3.3V serial input on the board's serial console port. Both the RC receiver and GPS receiver only connect with their serial transmit (TX) port to the board and not with the serial receive (RX) port, as bi-directional communication is not required. The connection scheme shows the same connections between the modules and the board as shown in the wiring diagram. The connection scheme elaborates on which pins with specific name coding are connected.

3.3. Software for autonomous navigation

The drone is now equipped with all the necessary systems to communicate and fly in the wild. However, in order for the drone to fly autonomously, navigation and control software is required. This is the most challenging and essential part of the project. In this section, the software created specifically for collision-free navigation is discussed. Additionally, there is an outline of all other flight support systems for remote communications, mission planning, safety, and more in the following section.

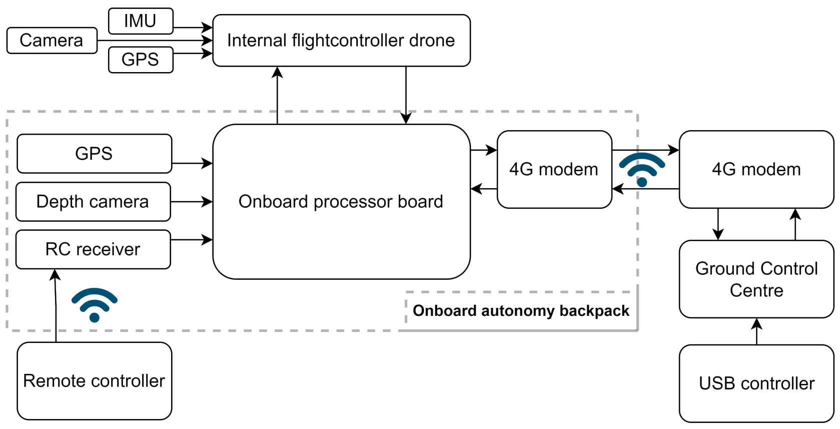


Figure 3.5: Drone system architecture for autonomous navigation in the rainforest

Figure 3.5 provides an overview of the system's different modules and how they communicate with one another. The diagram highlights whether the communication between modules is unidirectional or bidirectional, as well as whether it is a wired or wireless connection. The dotted square represents the systems that are integrated inside the autonomy backpack. The drone's internal flight controller and sensors, in addition to the autonomy backpack, form all the onboard systems on the drone. In this overview, the communication from the ground control centre to the drone is assumed to be 4G. In the case of a Wi-Fi connection, instead of 4G, the "4G modem" is replaced by the "Wi-Fi modem". Appendix B.3 shows an enlarged version of this system overview, together with the automatically generated ROS RQT-graph.

The upcoming subsections will explain the process of building the system software. To start with, there will be a brief introduction to ROS and how the drone's SDK functions with ROS. Following that, the generation and monitoring of collision-free trajectories using the EGO-planner will be explained.

3.3.1. Robot Operating System

At the heart of the extra onboard processor, ROS (Robot Operating System) is used to run read all sensors, process all data, and produce navigation commands to navigate autonomously through rainforest. ROS is an open-source framework that facilitates the development of robotic systems. It provides tools, libraries, and conventions for creating and managing distributed and modular software components. ROS enables module communication with tasks like sensor data processing, control algorithms, and simulations.

Multiple open-source software packages, such as drivers and algorithms, are available in ROS. Therefore, ROS can be used to quickly integrate a new type of sensor or test/simulate someone else's algorithm as long as a package is available. As ROS is the most used robotic operating system worldwide, popular sensor drivers and algorithms also have community support for bug fixes, improvement suggestions, and compatibility updates.

3.3.2. Parrot Software Development Kit & Bebop Autonomy

As a drone platform is used with SDK-enabled functionalities, some communication interface is required to link the drone flight controller and the extra onboard computer that processes the autonomous navigation. Bebop Autonomy is an open-source ROS-package that links a ROS network to the Parrot Bebop SDK. The SDK only supports high-level controls at low rates (max 10 Hz).
The Parrot SDK combined with Bebop Autonomy supports the following functions:

- Takeoff, land, kill-motors.
- Roll, pitch, yaw, and vertical speed. (as controlled from a RC)
- Goto GPS waypoint.
- Flip, go home, start/stop recording, take photo.
- sensor data: GPS position, accelerometer, and camera stream.

During early tests, it was discovered that indoor position control is inaccurate because the Bebop autonomy driver only estimates the (cartesian) X,Y, and Z position of the drone by integrating the accelerometer readings at 10 Hz. In comparison, most flight controllers which use this method integrate measurements at 200 Hz. Therefore, due to the low rate, the integration for the position estimate is less accurate and drift quickly occurs.
The Parrot SDk supported a GoTo (relative) X,Y,Z,Yaw position, which was not integrated into the Bebop autonomy driver. For this project, the open-source driver is extended to enable this functionality. Now, the drone can move to a relative position and hover still at that position as long as lighting conditions are sufficient for the drone's optic flow camera.

The other option to link the Bebop drone flight controller would be to load Paparazzi control software from the MAVlab into the flight controller. Paparazzi is a flight system software that does low-level flight control and is also capable of some high-level control within the processor capacity of the drone itself. When Paparazzi software is loaded into the drone, it replaces the default control software provided by the factory. Paparazzi is developed by drone labs, including the TU Delft MAVlab, to test agile low-level control at high rates at a component level.

The main reasons Parrot SDK combined with Bebop Autonomy is chosen over Paparazzi are implementation time constraints and the reliability of the system. The chosen option was already compatible with ROS and was, therefore, less work to integrate. If Paparazzi had been used, a driver for ROS had to be programmed from scratch, which would have cost engineering time that was not available. Also, the default software is well-developed and bug-fixed by the factory, while the Paparazzi software can have many factors of instability or unreliability. Thereby, the default software has a reliable optic flow to hover still in a situation without GPS, while Paparazzi has not yet a proven optic flow in the position controller. Thus, with Paparazzi, the drone slowly drifts away, possibly into an obstacle, when it assumes to hover at a fixed position.

The use of Paparazzi would also have had some benefits. The biggest benefit would be the integrated mission planning with safety sequences and a user interface to supervise the drone. These features were initially lacking in the ROS system and were programmed minimalistically for the purpose of the Rainforest competition, described in section 3.4.6.

3.3.3. Collision-free trajectory planning

As was concluded from the related work research, the EGO-planner is used for trajectory planning. This planner is a ROS package that also includes obstacle detection and mapping. The underlying methods for the trajectory planner are already discussed in chapter 2.

Input

The planner relies on the drone's odometry and depth image from the camera as its inputs. The odometry, which is obtained from the drone driver at a rate of 5 to 10Hz, uses the dead-reckoning method for localization. It consists of the position and velocities in 6 dimensions, which are the x,y,z directions and rotation around the x,y,z axis. The depth image input has a resolution of 720x480 and provides a depth value for each pixel. The depth image is captured at a rate of 15Hz.

Mapping

The EGO-planner detects and maps obstacles. The mapping and planning are based on voxel-grid maps, which define a real-world map with specific dimensions in x, y, and z, and a size per voxel cube. Figure 3.6 shows an example of such a voxel-grid map. In this example, two scenarios are shown: one mapping of cluttered objects, such as vegetation, and one mapping with structured objects, such as walls.

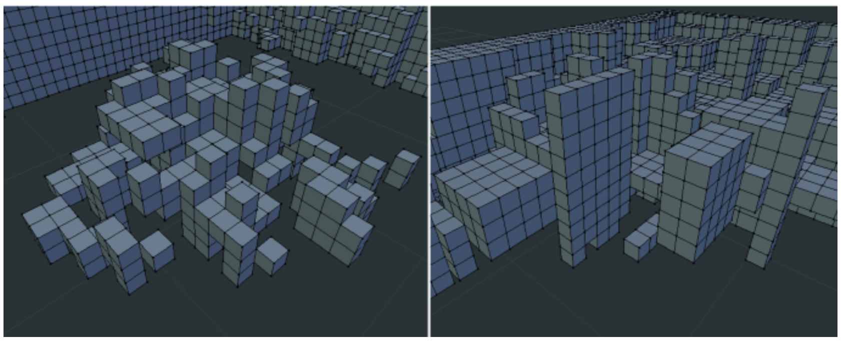


Figure 3.6: Voxel-grid map examples with cluttered environment on the left and structured environment on the right

For each depth image received, every pixel is projected onto a 3D space and translated to the real-world position using the drone's pose and position estimate. For the voxel block where the 3D detection is assigned to, the likelihood of an obstacle in that block is increased. The probability of obstacles is simultaneously reduced in the voxel blocks between the drone and detection. Since the camera cannot see through obstacles, it assumes that there are no obstacles between the camera and a detected point. This process is called raycasting. This report explains these projection and raycasting processes in more detail in section 6.1.

When the probability of a voxel block exceeds a certain threshold, it is marked as an obstacle on the 3D map. In the trajectory planning algorithm, the drone is assumed to be infinitely small. To avoid collision with the drone's body and plan a path with sufficient clearance around obstacles, all obstacles in the map must be inflated.

Inflation means adding a certain number of blocks around each obstacle block in all directions. The number of blocks added is determined by the inflation distance, and the resulting map is called the inflated obstacle map. The planner uses this map to create trajectories that avoid colliding with any of the inflated obstacle blocks.
An important feature of the EGO-planner is the local updating and usage of the map within a specific range around the drone. By updating and using only a local map, the obstacle detection and trajectory generation processes consume fewer processor resources. This feature enables the drone to operate more efficiently and conserves processing power.

Path generation
As described in Chapter 2, the EGO-planner follows a three-step process to generate a path. First, it samples potential paths using an A* algorithm [25] on an inflated obstacle map. Second, it chooses the path with the lowest cost using a specific cost function. Finally, it optimizes this path using the expected improvement (EI) criterion. The optimization process involves sampling the path N times and returning the B-spline trajectory with the lowest cost. However, each sampled path is checked for collision, and discarded if a collision is detected. Figure 3.7 shows an example where obstacles are mapped, a path is generated, and the drone follows the path.

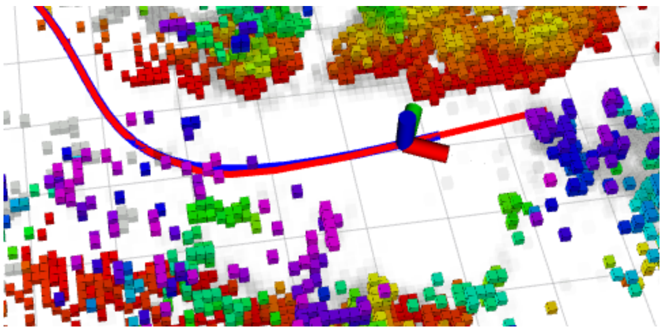


Figure 3.7: Visualisation of mapping, path generation, and path tracking

The following section will explain how the drone follows the generated trajectory.
The maximum number of iterations, cost functions, and maximum planning time/distance configurations are tested and tuned during indoor and outdoor tests on the built drone platform in forest conditions. These tests will be discussed in the next chapter.

3.3.4. Execute trajectories

When a navigation system determines the path for a drone to follow, a control system ensures that the drone accurately follows the computed trajectory. Without a control system, the drone could only move along a straight path to a single waypoint. State-of-the-art planners generate a trajectory that includes a list of timestamps with desired positions, velocities, and accelerations for each timestamp.

When a system needs to follow computed trajectories, it has two options: it can either follow a trajectory relative to the drone's position or follow a trajectory relative to the map/world frame. If the first option is chosen, accumulating drift by position control errors will cause the end position to be inaccurate. The flown track will correspond the shape of the calculated trajectory, but the drone will not be following the position of the trajectory. On the other hand, with trajectory following based on the world frame, the system aims to compensate for the position error. This means that the accuracy of the end position is likely to be better.

The Bebop SDk cannot take the desired position, velocity, and acceleration as input to track a path such as PX4 autopilots do, as in [18]. The Bebop SDK can only take in a desired x,y,z,yaw position relative to the drone's current position and heading. To work by this limitation, a ROS node publishes the desired position on the trajectory for a second in advance every second. This approach may suffice for basic trajectory requirements, but it increases the risk of collisions as the drone may take shortcuts and collide with obstacles.

3.4. Additional software for outdoor missions in competition

For the Xprize competition's missions, additional critical flight operation modules complement obstacle avoidance. This section elaborates on these modules.
Firstly, strategic 4G cellular connectivity extends the operational range and ensures signal robustness in dense vegetation. Next, the explanation covers live drone telemetry streaming, which is essential for compliance with Singapore's air traffic control regulations. The third part introduces redundant RC control, integrating both USB-wired and wireless radio controllers.

The redundancy extends to duplicate GPS localization, integrating a second GPS module for enhanced accuracy, detailing GPS switching logic, HDOP values, and ROS packages. Subsequently, redundant geofencing is discussed, emphasizing compliance with safety requirements. For ground-control-station mission planning and monitoring, ROS modules cover telemetry, geofencing, waypoint mission planning, video streaming, and live map monitoring.

To enable autonomous missions, waypoint goals are integrated into the EGO-planner, providing insights into GPS goal translation and handling sporadic GPS positions. Then, the section highlights the logging strategy to meet competition requirements and analyze drone performance, encompassing both flight controller data and ROS logging. This section concludes by discussing the automated startup of all software modules.

3.4.1. 4G cellular connectivity via VPN

To achieve extended range and overcome potential signal loss issues in dense vegetation, 4G connectivity was chosen for supervising the drone in the rainforest. The cellular coverage in Singapore's rainforest made 4G seem to be the most reliable option for the competition. Extensive research identified the ”M1” sim card provider as having the best coverage, further confirmed by the Xprize organisation. As described in section 3.1.3, the Huawei e3372 LTE USB dongle was chosen to provide 4G connectivity to the laptop and drone.

To connect the laptop and drone's ROS network while utilizing 4G, a VPN was established using Tailscale. Tailscale creates a mesh-VPN network, eliminating the need for a designated host or server and enhancing system robustness. Alternative solutions like fixed IP sim cards were considered but were less practical due to encryption, costs, and ROS integration challenges.

Installing Tailscale VPN software on all devices created a virtual connection to the same local network. This allowed SSH, ping, and ROS networking features between devices, including laptops, smartphones, and tablets. In emergencies, terminal commands can be sent to the drone from any device connected to the same VPN network. Examples of created terminal commands are ”bebop_home” (auto-return-home), ”kill_drone”, ”land”, and ”takeoff”.

3.4.2. Drone telemetry streaming to web server

To ensure the safety of all air traffic in Singapore, every drone was required to stream live telemetry data to a server of the competition host, Garuda Robotics. Garuda offered two solutions: streaming the data via a 4G network to the server via HTTP, or using a GPS tracker device from Garuda that will send the data automatically.

As the built drone was already connected via 4G, and the external tracker would add another 80 grams, the data was streamed to the server via API. To implement this, data had to be sent in specific formats, specific decimal values, and correct timestamps in an HTTP request to their server. To annotate which drone was sending the data, every registered drone for the competition had its own 20-digit hexadecimal code, which was shared with the server on initial contact. Garuda provided documentation on the required order and format of the data sent to the server.

To stream the data from ROS, the "rosbridge-server" package was used to create the HTTP request. This package is adapted to stream from the right sources, convert the data into the right formats, and fill it in the correct fields for the HTTP request. Figure 3.8 shows the airspace visualisation of the server to which the data is sent.

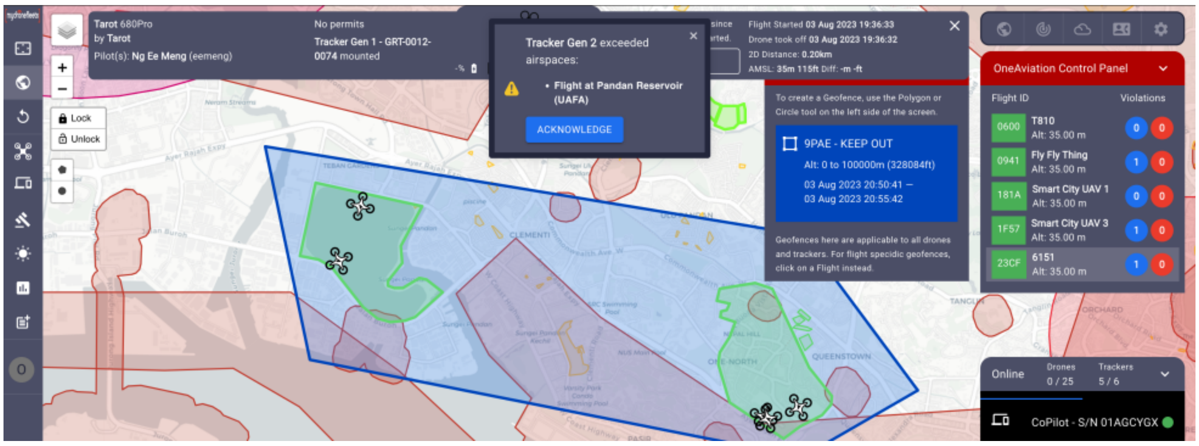

Tarot 680Pro
by Tarot
Pilot(s): Ng Ee Meng (eemeng)
No permits
Tracker Gen 1 - GRT-0012-0074 mounted
Tracker Gen 2 exceeded airspaces:
Flight at Pandan Reservoir (UAFA)
ACKNOWLEDGE
Flight Started 03 Aug 2023 19:36:33
Drone took off 03 Aug 2023 19:36:32
2D Distance: 0.20km
AMSL: 35m 115ft Diff: -m -ft
To create a Geofence, use the Polygon or Circle tool on the left side of the screen.
9PAE - KEEP OUT
Alt: 0 to 100000m (328084ft)
03 Aug 2023 20:50:41 –
03 Aug 2023 20:55:42
Geofences here are applicable to all drones and trackers. For flight specidic geofences, click on a Flight instead.
OneAviation Control Panel
Flight ID
Violations
0600 T810 Alt: 35.00 m
0941 Fly Fly Thing Alt: 35.00 m
181A Smart City UAV 1 Alt: 35.00 m
1F57 Smart City UAV 3 Alt: 35.00 m
23CF 6151 Alt: 35.00 m
Online
Drones 0 / 25
Trackers 5 / 6
CoPilot - S/N 01AGCYGX
Lock
Unlock
CLEMENTI
QUEENSTOWN
ORCHARD

Figure 3.8: Singapore Airspace visualizer by Garuda Robotics [26]

3.4.3. Redundant RC control

Initially, the drone was operated using a USB-wired universal game controller connected to the laptop. Joystick inputs were translated into desired attitude commands (roll, pitch, yaw, and vertical speed). These commands were then sent to the drone through a ROS topic via a WiFi connection.
This direct communication with the drone bypasses the laptop and Wi-Fi connection.

The ROS package Sbus-serial was utilized to read the receiver's serial signal on the drone's processor. As the Sbus protocol works by inverted serial data, some soldering work on the RC receiver was required to retrieve standard serial data. The ROS package maps the RC channels to appropriate commands for the ROS Bebop driver such as roll, pitch, yaw, thrust, takeoff, land, kill, autopilot on/off, and set new goal.

Considering the dual-controller setup, priority settings were implemented. Each ROS node that reads controller topics received inputs from both controllers. The wireless RC has priority during normal operations, while the wired RC kicks in when the other wireless RC fails or goes out of range. A switch on the wired controller can also force the system to listen to that controller.

If one controller becomes unavailable, the system automatically switches to the other. The added benefit is that the second RC isn't tethered to the laptop by a USB cable, providing the pilot with freedom of movement and better visibility of the surroundings for safer operations. And, when flying beyond visual line of sight (BVLOS), the wired controller is used via the 4G connection while the pilot supervises the drone from the laptop.

3.4.4. Redundant GPS localisation

As described in section 3.1.3, the system uses a second GPS receiver for higher accuracy and more reliable reception in poor signal conditions. The ROS nodes that use GPS listen to both this and the drone's internal module.

Priority switching makes sure the best GPS location estimate is used. If both modules give a GPS reading, the GPS value of the module with a valid fix and the lowest estimated error is chosen. For the estimated error, the GPS fix's horizontal dilution of precision (HDOP) is used. Also, a hysteresis value of 0.2m HDOP is used to prevent switching too often between the GPS sources if the accuracy is about the same. Due to time constraints, the GPS input only switches as described. A Kalman filter with the input of both GPS modules and IMU values would give a more precise location estimate.

The second GPS module is read over a serial port on the processor board. The raw NMEA strings going over the serial port are interpreted by a ROS module named nmea-navsat-driver. This package processes the raw data and publishes ROS topics with the GPS position, speed, and time reference.

3.4.5. Redundant Geofencing with kill-function

To fly a drone in the Netherlands under the TU/Delft MAVlab, extra safety measures are required beyond what the Dutch government requires. The drone must have a geofencing system that makes it land or shut down in case of a fly-away, regardless of pilot input or autonomy system commands.

Most consumer drones, like the Bebop 2, have geofencing that prevents them from flying outside a maximum-distance-from-home circle but doesn't land or shut down in case it breaches this distance. To meet MAVlab's specific requirements, the drone requires land or killing functionality, even if most modules fail. The geofencing should work independently of the remote control, ROS, processor board, and the connection between the processor and drone.

Fortunately, the drone operates on a Linux-based system, allowing one to create custom scripts for added functionality. However, the flight controller code is precompiled, limiting script accessibility. As a result, the geofencing had to be implemented in a separate bash script, making it harder to access system data like GPS and send commands like landing or shutdown.
In the drone's Linux system, a bash script is created that checks the drone's GPS position and distance at 4Hz. During startup, the script establishes the average of the first 100 GPS fixes as the origin. The loop calculates the difference in latitude and longitude from this origin and translates it to meters. If the drone's distance from the origin exceeds the configured maximum distance, the script triggers a reboot, shutting down all motors. Figure 3.9 below shows an example of this situation with the geofencing overlaid over a satellite image from Google Earth.

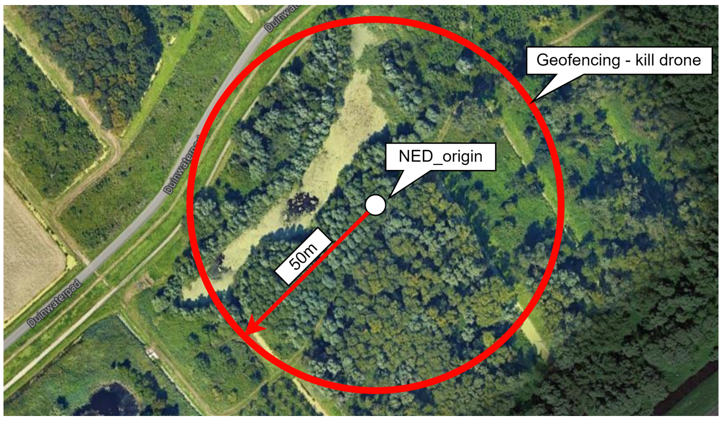


Figure 3.9: Geofencing kill function for redundancy. Example overlay on an image from Google Earth.

The script also filters out invalid checksums, non-valid fixes, outliers, and Dilution of Precision from the raw GPS data, preventing the drone from shutting down due to inaccurate readings. Although implementing these calculations and checks in a Bash script presented challenges, it proved to be the most suitable solution for the drone's minimalistic processor.

The bash script is triggered by a double button press on the drone's power button instead of running automatically at startup. This prevents potential issues like the drone getting stuck in an infinite reboot loop when the script fails. In that example, the drone's operating system could never be accessed again to fix the bug, rendering the drone useless. An alternative approach was considered, such as introducing a one- or two-minute delay before activating the geofencing automatically. However, this posed a risk of the drone taking off during that time, potentially resulting in a fall if the geofencing activates and exceeds the limits.

For the competition, a polygon-shaped geofencing would have been preferable to cover all parts of the competition region. Although this feature is standard in PX4 flight controllers, implementing this in a bash script was deemed too complex, and even in ROS, it would have presented significant challenges.

3.4.6. Ground-Control-Station mission planning and monitoring

A standard ground control station (GCS), like the commonly used Qgroundcontrol with PX4, provides features for configuring waypoint missions, handling system failures, and implementing geofencing. It displays telemetry, live video, error messages, waypoints, and geofencing details.

Due to hardware choices, many functionalities had to be developed in ROS from scratch as no compatible GCS was available in this project. The GCS features are crucial for safely monitoring autonomous operations and will be checked during the airworthiness test by the Garuda Robotics company. This section covers telemetry, geofencing, waypoint mission planning, video feed streaming, and live satellite-view map monitoring.

Telemetry
Telemetry broadcasts real-time drone diagnostics, including speed, battery level, signal quality, and warnings. The list of required telemetry values includes:

- Speed & heading
- Battery level & expected remaining flight time
- Signal quality to GCS.
- Low battery warning
- Signal lost warning
- Nearby/at/outside geofencing warning
- Sensor failure warning
- switched autonomous/manual flight-mode warning

These values are displayed in a command-line box where a telemetry node pulls the necessary data from the ROS network. Error messages are triggered by a script checking for specific conditions, meeting competition requirements inspired by various GCS software packages.
Apart from the values directly read from ROS in the drone, the drone's 4G connectivity for telemetry and video streaming measures ping time between the GCS and drone, providing insights into signal quality as well.

Waypoint mission planning
Autonomous missions require detailed planning, including mission-specific parameters and emergency protocols. The competition requires that at least these planning and emergency protocols are visible on the GCS:

- Waypoint list
- Waypoint route for mission
- Maximum altitude
- Maximum distance from home
- Geofenicing configuration
- Minimum Battery level before auto-return-home
- Return home altitude

- Maximum speeds (return home, waypoint mission, manual flight)
- Auto landing at sensor failure
- Protocol for signal loss

Ideally, these configurations would be in a user-friendly menu on the GCS. But, due to time constraints, these are hard-coded in the script or a configuration file.

Video stream and live map
The competition requires a live video stream and live drone position on the map during beyond visual line of sight (BVLOS) flights.
ROS incorporates video streaming within the same VPN network as the GCS. The 4G network connection enables this video stream, with a delay of 500 to 1000 milliseconds.
Displaying the live drone location on a map is not possible due to compatibility issues. During the competition, the Singapore government's website was used to track all drones' positions on a map. Geofencing settings were displayed on a satellite image during the pre-flight check, as implementing them elsewhere on a live map on short notice was not feasible. This solution was acceptable to the competition organization.

Geofencing
Geofencing ensures the drone stays within the operation area, activating specific procedures when breached. Our drone, supporting only circular geofencing, uses a hardcoded solution in the flight controller to stay within defined boundaries.

Appendix B.4 shows a screenshot of the created ground control centre interface and created geofencing. In conclusion, the mission planning and monitoring section highlights the adaptation of standard ground control station functionalities, like Qgroundcontrol with PX4, to our project's unique needs. With no compatible GCS, the development of essential features in ROS became imperative. The telemetry system ensures real-time monitoring of critical drone parameters, while waypoint mission planning involves configuring various parameters for mission execution and safety. Geofencing safeguards the drone's operational boundaries, and the use of 4G connectivity for telemetry and video streaming introduces novel considerations. Despite challenges, the section underscores the effective implementation of these elements, meeting competition requirements and ensuring the successful monitoring and control of the drone during autonomous operations.

3.4.7. GPS waypoints to planner goals in EGO-planner

To explore the rainforest, the drone must follow waypoints, which can be a single destination or a route. But, the EGO-planner can only use 3D cartesian waypoints instead of GPS coordinates, therefore translating GPS locations to cartesian is required.

At each new GPS fix, the planner recalculates the cartesian goal relative to the drone. It calculates NED (North, East, Down) coordinates from the drone's GPS location to the desired waypoint. The new cartesian goal is then added to the drone's location estimated by IMU, with the NED coordinate added. By this approach, the relative position from the drone to the waypoint is correct at that moment, no matter what the drift was in the dead-reckoning localisation. This concept is illustrated in Figure 3.10 on the right.

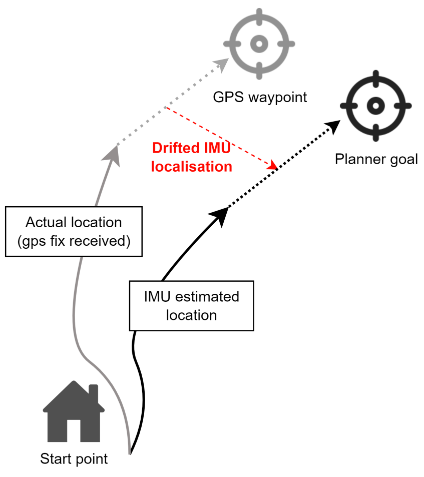


Figure 3.10: Illustration of GPS waypoint to Cartesian goal conversion

When autonomously navigating with obstacle avoidance, the drone's low speed limits its range, travelling at around 1 m/s compared to its maximum speed of 16 m/s. When flying at an altitude safely above all canopy, obstacle avoidance is not necessarily required. Therefore, flying without obstacle avoidance to a waypoint is also incorporated.

Flying straight to a waypoint without obstacle avoidance allows the drone to fly at 10 m/s, having a more extended reach on a single battery charge.

3.4.8. Logging

To track the performance of the flight tests, fulfil the Xprize requirements, and debug test flights, system data is logged. In ROS at the processor board, the Rosbag functionality records data passing through predefined topics. In the drone itself, a black-box logger records all flightcontroller data.

By using ROS Launch file, the topics that need to be logged are defined and started logging. Logging all topics is unnecessary, generating larger log files and slowing down the system. The log files can be used for analysis later. They can be plotted in graphs, printed in tables, and replayed as datasets as if the system is live running.

For debugging and getting flight performance insights, the ROSbag files can be used to playback data. This data can be visualised, such as plotting tracking error, rates of trajectory planning and localisation drift. This data can be used to measure the performance of the system. Also, telemetry data must be recorded for the competition. This requirement can be met by logging with ROSbag files.

Inside the Linux system of the drone itself, a black-box recording can be enabled. In this recording, most low-level control values, such as attitude, body rates, individual motor thrust, etc, are logged in a CSV file format. This log can write up to a rate of 200 Hz. The CSV file can be extracted from the drone after flight to inspect the low-level data if required.

3.4.9. Automated startup of all flight systems

In the field, ensuring swift and straightforward battery replacements is crucial. To streamline this process, the startup and initialization of all software systems in ROS is automated. This automation eliminates the need to manually launch individual scripts in a specific order, making it more efficient and robust, especially in challenging field-testing conditions without a dedicated workspace, power supply for the laptop, and potential exposure to adverse weather conditions.

The user can activate this automation process via a ROS Launchfile, which executes a set of scripts with a single shell command. Via the WiFi or 4G+VPN connection, from any device with an ssh terminal, this Launchfile can be started.
Dependencies are carefully managed to ensure that each script is launched only when the preceding one is successfully running. Furthermore, the automated startup sequence includes a validation step, ensuring the proper functioning of every required process within the ROS network. If any abnormalities are detected, such as a malfunctioning camera connection, the system promptly halts the automated startup process.

The automated startup defines how and when a failing process needs to be restarted to restore the process in case a module failure occurs during flight. During recovery, the drone receives no new input and stays in a stable hovering state. When the drone does not receive any input from the RC or autonomy system for a period of 30 seconds, the drone can activate auto-return-home, depending on the mission. These comprehensive automation and fail-safe mechanisms ensure the robustness and reliability of the drone's software systems in dynamic field conditions.

3.5. General overview

To summarise what the mechanical, electrical, and software design of the autonomy backpack looks like, this section gives a general overview.
First, the drone platform, processor board, and sensors are chosen. These are:

- Parrot Bebop 2 drone
- Odroid XU4 processor
- Realsense D435i depth camera
- external GPS receiver

- USB 4G dongle
- Receiver for wireless manual controller

Thereafter, multiple frames are modelled to integrate all components compactly, lightweight, and protected on the drone. To show how all systems should be wired, an electrical design is shown with diagrams.

A software architecture is created in ROS to make the drone fly autonomously. The software components share information interactively with each other to process sensor data, map the environment, plan an obstacle-free route, and command the drone to follow this route. To map the environment and plan routes, functionalities of the EGO-planner are utilised. The drone's software development kit is used to read onboard sensors and control the drone.

In addition, to not only avoid obstacles but also perform a complete outdoor mission in the rainforest the following modules are implemented in the software:

- 4G cellular connection via VPN
- Drone telemetry streaming to webserver
- Redundant RC control
- Redundant GPS localisation
- Redundant Geofencing
- Ground control monitoring and mission planning
- GPS waypoints missions for planner
- Logging
- Automated system startup

4 Testing and calibration

In order to ensure the robustness of the system integration discussed in the previous chapter, several tests were conducted. These tests, which strived towards a robust system for the competition, are discussed in this chapter. Additionally, the demo and test flights that took place in Singapore will also be discussed at the end of this chapter.

4.1. Testing system integration

To validate all systems work properly, several tests are conducted, and calibrations are applied where necessary. Within this project, testing and calibration can be split into two categories: flight operation support systems and autonomous navigation with obstacle avoidance.
As various flight operation support systems are required and mandatory to operate the drone in the wild, these systems have to be tested to see if they work individually and also integrate correctly with the other systems onboard the drone. In this chapter, the hardware and software for these systems are addressed. Testing methods are shown, and results and/or complications are addressed.
The autonomous navigation with obstacle avoidance is tested to integrate correctly with all systems onboard the drone. And optimised to work as efficiently as possible for the competition within the given time to create the system. This section shows the test methods and results for this system.

4.1.1. Localisation, obstacle detection, and mapping

Testing the drone's mapping system for accuracy and reliability involves testing its localization and obstacle detections. The drone's localization was tested by monitoring its IMU position estimate over time. The drone was flown manually in the Cyberzoo for different durations, while being kept in motion as much as possible with a speed of roughly 1.5 meters per second.

Dead-reckoning localisation

During these tests, the drone took off from the exact centre of the Cyberzoo and landed manually at the same location after each test. After landing, the drone's x, y, and z position estimates were then documented. Ideally, the position estimate should have been close to zero. However, the IMU position estimate of the Bebop driver drifted over time, resulting in a position estimate not close to zero. The observed drift is shown in table 4.1 on the right.

Table 4.1: Observed position estimate drift over different lengths of time

Flight duration	**X**	**Y**	**Z**
60 seconds	-0.56	0.26	0.43
120 seconds	0.95	-0.67	0.94
180 seconds	1.39	-1.43	1.82

The table does not show a consistent relation between flight time and drift, which was also not expected. What the table does show is that the drift accumulates over time, thus creating a bigger drift if the flight time is longer, which follows the hypothesis. It is important to mention that the Cyberzoo environment significantly interferes with magnetic fields due to many devices, transmitters and metal parts, which may affect the drone's heading estimate and potentially worsen the X and Y position drift.

Obstacle detection and mapping
During testing of obstacle detection and mapping, several objects, including trees, were placed in the Cyberzoo. The mapping behaviour was observed in RVIZ during manual test flights.

The system was tested to see if it would map all branches of the fake trees. To test this, the point cloud from the depth camera was compared to the actual shape of the tree. The point cloud displayed in RVIZ was then compared to the voxel mapping created by the planner's algorithm. Based on human assessment, the voxel map was deemed accurate in terms of the shape and size of the obstacle.

Next, the mapping was tested by flying the drone around in the Cyberzoo and observing obstacles from various angles while moving back and forth from/to it. It was found that the voxel map would sometimes shift the blocks representing an obstacle. During further investigation, a pattern was detected, and it was discovered that the transformation matrix between the camera's and the drone's centre did not include the mounted offset of the camera. The shift of obstacles was solved when the camera mounting offset of 70mm was included in the matrix. Over time, some shifting in obstacles occurred, which is inevitable when localisation drifts in dead-reckoning.

4.1.2. Path planning and tracking

To test the tracking capabilities of generated paths, the first focus lies on testing the drone's position control functionality. This involves evaluating the GoTo functionality within the Parrot Software Development Kit (SDK). Subsequently, the tracking of the path, guided by this position control, is tested. In this test, the trajectory server must publish GoTo commands to the drone to follow the generated trajectory as accurately as possible.

Bebop GoTo by Parrot SDK
The drone is programmed to follow a specific path by receiving cartesian goals relative to its position and orientation. In ROS, the trajectory-server divides the calculated path into multiple points and passes them with the correct timing to the drone's GoTo function. First, the functionality of the GoTo function is tested to ensure its accuracy and whether the trajectory server passes the coordinates correctly in the Bebop's coordinate frame. The Bebop's coordinate frame is X=forward, Y=left, and Z=down.

In this experiment, the drone was commanded to GoTo positions [-8, -8, 0] and [+8, +8, 0] repeatedly. The drone uses its IMU and the optic flow of the bottom camera for positioning in this case. Diagonal flights through the Cyberzoo were repeated ten times in one minute using a looping bash script. After ten iterations, the drone drifted about one meter, which was better than expected.

When given a GoTo command, the drone quickly accelerates to reach cruising speed (10m/s) within the first 2 meters, decelerates slowly when arriving at the goal, and then makes a quick stop at the goal without overshooting. Although it would have been useful to add to the report, the positioning and acceleration of this test were not measured by Optitrack.

Tracking paths with the trajectory server
After validating the functionality of the GoTo function, the trajectory server needs to be tested to ensure that it provides the correct goals at the right time. To achieve this, the EGO-planner is used to test the trajectory server. A simple object is placed in the middle of the Cyberzoo, and the planner calculates a path, which is followed by the drone.

The EGO-planner is not yet well-calibrated for this test as it is the first time using this planner. The default settings are used, except for the maximum flight speed and acceleration, which are set to 0.5 m/s and 0.5 m/s^2. Additionally, the planner is set to generate only one path and execute it instead of continuously updating the path. This helps to compare the flown path with the generated path.

During the test, it was observed that the drone lags behind significantly to the desired position on the trajectory. This is due to the GoTo command incorporating acceleration and deceleration to the goal, which creates a delay of 1 to 2 seconds since the distance is short. Due to the delay, it cuts corners of the trajectory, which could result in a collision with the obstacle.

To overcome this issue, the GoTo command is multiplied by two, which means that it only accelerates to the goal and does not plan a deceleration to a full stop. This reduced the bebop's lagging behind the desired position to about half a second delay. And, by following the path's curves instead of cutting corners, the drone improved its performance. The multiplication of the GoTo command is made into a system variable that can be fine-tuned in future testing and calibration.

Creating feed-forward control or implementing a PID position controller was considered to increase the tracking performance. However, due to limited time, it was not incorporated. Moreover, the drone's IMU is only published at 5 to 10hz in ROS, which is problematically low to create a good position controller. Meanwhile, the GoTo command is processed in the drone's flight controller, which provides stable and reliable results. The GoTo command also uses the drone's optic flow camera, from which the data cannot be accessed in ROS for position control.

4.1.3. Telemetry streaming via Garuda server

As described in section 3.4.2, streaming telemetry data to air traffic control is required for the Xprize competition. This streaming requires a test in advance of the competition. The test would check if all required data is sent, with exactly the required decimal places for every value. For the test, the organisation gave the following instructions:

- Start sending telemetry
- Fly a 200m*200m square in four minutes
- Do some other movements to make a flight of 10 minutes in total
- Land the drone
- Repeat one more time this flight
- Send the organisation an email with the specific date and time the test flight was conducted.

This test is executed a month before the competition. The test took place outdoors at a big empty parking lot which had enough space to fly the big square. After the test, the Xprize organisation was emailed to validate the results on their server. The organisation then validated the data and replied to the email that the telemetry was correctly sent.

4.2. Outdoor testing and calibration in Dutch forests

Multiple tests were carried out in Dutch forest environments to assess the effectiveness of obstacle avoidance, refine obstacle mapping settings, and determine the optimal configurations for computational efficiency and flight speed. Various locations were utilized to simulate diverse forest scenarios. Due to restricted airspace in Delft and the absence of a forest in the MAVlab's outdoor test field, most tests took place near Breskens in Zeeland.

Conducting tests in forests provided valuable insights into flight operations in natural environments. As previously mentioned, forests lack facilities for repairing the drone or charging equipment. With no tables or chairs available, ground operations had to be conducted directly on the forest floor. Due to the limited number of functional batteries (only four), each trip to the forest yielded approximately four flights lasting around 10 minutes each, after which the batteries needed recharging.

Figure 4.1 illustrates a test flight in Breskens, where the 4G dongle's latency and bandwidth were evaluated. The test also involved observing the visualization of autonomous navigation, latency in the camera live stream, and control via the 4G network.

From the Dutch forest tests, three key findings emerged. Firstly, the drone easily exceeded the defined map size, leading to unresponsive autonomy. Careful consideration of map size within the processor's memory limit is crucial to cope with this issue. A challenge encountered was the drift in height estimate, causing the drone to breach the virtually defined map's border. As shown in table 4.1 in section 4.1, the drone has significant drift over time. The map size in the x and y directions is a couple of hundred meters, but in the z-direction, only a couple of meters lower the map's memory usage. In theory, the

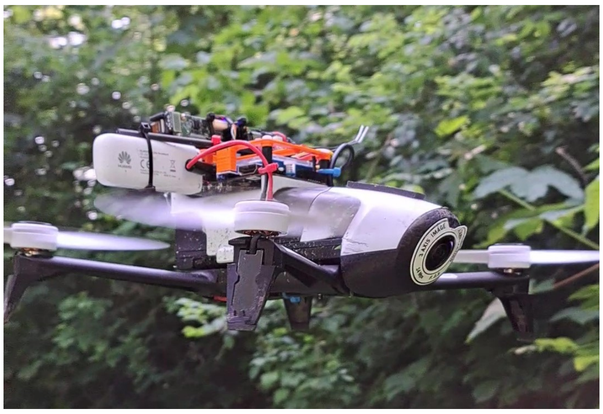


Figure 4.1: Testing and calibration of autonomy system in Dutch forest

drone needs only a couple meters above ground level to explore the forest, but when it drifts about 1m every 2 minutes, the map might need 15m in the z-direction not to exceed the map's borders with the localisation drift. Appendix D.3 shows how rapidly the map size, for example, in the z direction, increases the use of RAM memory of the processor board.

The second finding revealed that flying under a dense canopy with low light intensity caused disturbances in the drone's internal optic flow camera, affecting stable position control. Consequently, the drone showed unexpected instability in these scenarios, posing a risk of crashing. It's important to note that low light conditions did not notably affect the depth camera used for obstacle detection.

The third finding highlighted that additional weight distribution on the drone and imu localization drift leads to excessive drift when doing many yaw rotations. In dense environments, when the drone struggled to navigate and generated paths with significant yaw rotations, it tended to drift into obstacles. Although the depth camera prevented collisions with visible obstacles, the 270-degree region not covered by the camera was susceptible to crashing.

4.3. Airworthyness test Xprize Competition Singapore

To assess the airworthiness of a custom-made drone, the Singapore flight authorities required an examination of the drone. The requirements were extra strict for Beyond Visual Line of Sight (BVLOS) operations, which would be most missions in the forest. The evaluation included pre-flight, in-flight, and post-flight procedures, overseen by the organization and two representatives from the Civil Aviation Authority of Singapore (CAAS).

Originally, the tests were scheduled for Saturday morning but were postponed to Sunday due to persistent rain. However, Sunday had rain and thunder warnings all day, which caused further delays. The final opportunity for the airworthiness tests was granted on Monday morning, just before the start of the competition. Fortunately, the weather on Monday was favourable, allowing the flight tests to proceed.

On Monday morning, the weather conditions were sunny, with a temperature of approximately 35 degrees Celsius. The test flight field was soaked and damp, resulting in 95% to 100% humidity—typical for rainforest areas post-showers but untested conditions for the drone. The pre-flight checks involved verifying telemetry data, live video streaming, drone mapping, geofencing configurations, and other flight settings on the Ground Control Station (GCS). Despite challenging conditions, including wet grass and a lack of power sources, these pre-flight checks were successfully completed. Battery conservation of the laptop and drone was challenging but crucial due to the unavailability of charging options.

The in-flight checks included a flight mission of the following aspects:

- Takeoff
- Hover still for a minute in the same place without user input

- Fly a 20 by 20 meters square for 4 minutes as a waypoint mission
- Fly the drone to the (closely set) geofencing and try to breach the geofencing manually
- Activate the auto-return-home function

During takeoff, the drone showed instability and difficulty maintaining a hover. When manually piloted toward the geofencing, the drone initially appeared to halt, but then unexpectedly breached the geofencing and continued outside the designated area. Manual piloting it back to the take-off position required significant effort. Due to these safety concerns, the drone was landed. The planned square waypoint-mission and auto-return-home functions were omitted. The post-flight check, which would have included validating correct flight log recordings, did not occur. Because of a drained laptop battery and limiting time to the start of the competition, it was decided not to debug the problem and not to redo the airworthiness test. Therefore, the drone did not pass the test on Monday morning and was not used in the actual competition.

Analysis of the flight logs revealed unstable and likely incorrect readings from multiple sensors. The magnetometer and GPS, crucial for attitude calculation and positioning, showed abnormal fluctuations and inaccuracies. Exposure to intense sunlight, high temperature, and maximum humidity during the tests likely contributed to these failures. Unfortunately, there was no opportunity for private testing in these conditions in Singapore before the CAAS evaluation. The airworthiness test was the drone's first and last flight in Singapore.

Despite enabling black-box data logging on the Bebop drone, attempts to retrieve and analyze this data from the test flight failed. The black-box file had been overwritten in the drone's internal system, resulting in the loss of all sensor data from the flight.
Appendix E.1 lists some thoughts about future work to improve the EGO planner to be used in the wild.

5 Difficulties in rainforest environments

This chapter highlights the complexities of deploying autonomous robots in unpredictable natural settings. The chapter particularly focuses on challenges experienced in the Xprize rainforest competition discussed in the preceding sections. This chapter carefully examines the challenges encountered in the competition in Singapore's rainforest. It provides valuable insights for future researchers and engineers to deploy robotics in these scenarios.
Rainforests, known for their dense vegetation, high humidity, and variable climates, present unique challenges for autonomous systems. This chapter explores the impact of temperature and humidity on electronics, the difficulties of navigating cluttered terrain, connectivity constraints within thick foliage, and the limitations of lightweight sensors on drones. This chapter also delves into mission planning complexities, including slow progress, uncertain distances, and the nuances of regulatory requirements, notably Beyond Visual Line Of Sight (BVLOS) operations in forested environments.
This comprehensive examination serves as a foundational resource for engineers and researchers aiming to deploy autonomous systems in the diverse and challenging ecosystems of our natural world.

5.1. Environment

Robots in industrial environments often have mostly repetitive tasks, with nearly identical products and few variable environmental conditions. When deployed in nature, however, the working conditions are unknown when the robot is designed and often also unknown when deployed. When the robot is teleoperated, the design of the robot should be able to withstand all environmental conditions and obstacles. With autonomous operations, there is an even more complex task for engineers to design the autonomous system such that it finds solutions in all possible cases, and can also detect when there is no solution available. Unknown challenging conditions can be found in any form of nature, like deserts, oceans, mountains, snowland, or forests. This article focuses on rainforest conditions as these challenges are encountered by the author.

In the rainforest, dense vegetation can be found in a tropical, humid, and warm environment. Of all places, rainforests have the most varying biodiversity and are, therefore, a place of interest for researchers. Rainforests are also threatened by deforestation and climate change. Therefore, researchers would like to map and keep track of the biodiversity over time. As the rainforest often is a big area that is hard to access for humans, robots like drones and rovers can be deployed to scan the area remotely. These robots have to deal with the environmental conditions, but also connectivity, sensor, mission, and requirement issues that come along with these environmental conditions.

The environmental conditions are for now grouped into temperature, humidity and the dense cluttered environment. To start with temperature, this varies from 20 degrees at night to 30 degrees Celsius average air temperature during the day. When exposed to direct sunlight, objects can get much warmer depending on their colour and material. When exposed to the sun, electronics can easily get up to 60 degrees Celsius. Electronics have trouble working in these temperatures, mainly because the processor chips cannot be cooled anymore inside the electronic devices, causing failure, incorrect data, or blackouts.

Humidity is another factor that compromises the functionality of electronic devices. In rainforests worldwide, the average humidity is 77 to 88 per cent, often reaching 100 per cent after rain showers. In environments where the humidity is higher than 50%, electronics are susceptible to damage. Firstly, at high humidity, water droplets might condense onto components, causing short circuits. Secondly, humidity causes sensors to give wrong readings due to changes in material properties. Lastly, the humidity causes strong interference in wireless signals, this will be covered later in this article.

Where temperature and humidity are mostly related to the functionality of the electronic components of the robotics, the dense, cluttered environment is more related to the teleoperated or autonomously operated mission of the robot. The dense environment makes it hard for the robot to move through the forest in or under the canopy. As obstacles are densely populated and often only narrow open places to move through the forest exist, the openings might be hard to find for an operator or autonomous system. Also, the margins in which the robot has to operate not to hit and crash into an obstacle are narrow.

How wide the openings in the vegetation are to navigate through depends on the region, altitude, and vegetation type, and is a big uncertainty in the robot design and mission deployment. Where a rover is limited to the openings at ground level, the drone can vary altitude to find the openings. Our experience from the rainforest is that the varying altitude is a must for a drone to find openings. Navigating at a fixed altitude is nearly impossible, even if the jungle is less dense. The cluttered environments cause the openings to appear in rather unpredictable places. Also, the robot might have to move to some opening first to find out if the space behind this vegetation gives solutions to move further. This common occlusion makes mapping and navigating through the dense cluttered forest a slow process. For a drone, which has limited battery life, this slow process with big uncertainty of finding a solution to the goal and, even more important, finding a way back home, the slow process is challenging.

5.2. Connectivity

As mentioned in the previous paragraph, the humidity of the air causes interference in wireless signals. In addition, the dense vegetation with many leaves causes even more interference, mainly because the leaves contain much water. The water in the vegetation and air causes the signals to block/absorb, scatter and reflect the signals. As most robots rely on radio signals for control and/or supervision by the operator, this interference is a significant problem.

The more vegetation between the operator and the robot, the more the signal degrades. Therefore, there is a severe limitation on how far the robot can get into the forest. More powerful transmitters and directional antennas improve this range, but no more than a factor of two. On small robots like drones, this option might not be feasible due to weight and power limitations. Adding cellular connectivity to a robot is a common approach to support an unlimited range, but the coverage of cellular towers in rainforests is poor or sometimes non-existent. One factor that makes all wireless signals even a lot worse is rain. When it is raining, or when it just stopped raining, all vegetation in the forest is wet, causing even more absorption and scattering of the signals.
GPS signals from satellites are relatively weak signals. When the air is humid, the vegetation is dense and possibly wet, the signals are interfered so badly that a GPS fix is not possible or otherwise far from accurate. Inaccuracies of 50 meters are no exception in these conditions. Therefore, robotic solutions cannot fully rely on GPS localization when operating under the dense canopy layer. Other solutions for localization are challenging and make the robot often more complex.

5.3. Sensors

When the goal is autonomous navigation, multiple sensor difficulties have to be taken into consideration. Most probably, the robots engineered to find their way through the dense vegetation are compact to fit through the narrow openings. Therefore, the choice of sensors is limited as these need to be compact and lightweight to fit on a small-sized robot. The lighter weight sensors often have a more limited measuring range, accuracy, and signal-to-noise ratio. These factors make accurate mapping for navigation more challenging.

In addition, the available processing resources onboard the drone are limited because of the same reasons. When more noisy signals have to be pre-processed on a smaller processor, the response time will be longer and the update frequency lower. At the same time, the measuring range is shorter, which causes the system to look less far ahead. The result is less efficient path planning as new obstacles occur later in the system. Lastly, when the sensor accuracy is lower, the planner might have to re-adjust the trajectory more often while the drone is closing in on obstacles. For all these reasons, the drone has to move slower through the environment, compared to a system with higher quality sensors, to process detected obstacles and generate a safe collision-free trajectory reliably.

5.4. Mission

When performing a mission under the canopy or in the canopy layer, some difficulties arise that are worth mentioning. Firstly, most drone systems have a return-to-home sequence as a standard solution for error handling, such as signal loss or low battery. When this sequence triggers, the drone will (1) vertically ascend to a predefined altitude, normally safe above all obstacles, (2) fly horizontally straight home to the takeoff location, (3) descend and land. If there occurs any obstacle above the drone during the ascend, like tree vegetation or a branch, the drone might crash. If the drone has omnidirectional obstacle avoidance, as higher-segment DJI drones have, it might bypass the obstacle and find a safe route home. With omnidirectional obstacle avoidance, this problem can be tackled when flying in the canopy layer where there are openings to escape the forest. But, when flying under the canopy layer, there is a probability no gaps exist or can be found. In this case, a normal return-to-home sequence is not feasible.

As mentioned before, the progress of movement by robots through the dense forest is slow. Where, for example, most drones fly 15 meters per second from waypoint to waypoint in the open sky, a highly advanced drone in a dense forest might only fly 1 to 4 meters per second when avoiding obstacles. As the flight speed is depended on the density of the forest and the ease of finding openings to fly through, the distance that can be covered in a certain time is highly uncertain. Therefore, when the drone flies under the canopy, big margins in mission planning have to be planned on battery life when to return home. As the battery life for drones that perform difficult autonomous flight tasks in the forest is already limited to 5 to 10 minutes, the maximum distance from takeoff the drone can reach limits to some tens or hundreds of meters. Pushing the design of the drone to be most agile, fast computing and manoeuvring as possible will lower the flight time performance. The same holds visa-versa. Therefore, exploration in a big forest area is not very efficient. Flying over a part of the forest to explore an area under the canopy further away from the takeoff location is an option, but it is still bound to the limited flight time.

Trapping environments is another reason why the distance that can be covered under the canopy is highly uncertain. Under the canopy, different types of plants and other vegetation occur. In some places, gaps might be relatively wide and easy to spot, while in other places, gaps are hard to detect, too small, or non-existent. Also, the robot might follow a sequence of feasible gaps that are narrowing towards the end, ending up in a place with no gap or a too-narrow gap as the only option leading to the goal. For an autonomous system, these situations might be hard to escape from. In any way, manually piloted or autonomous flight, the drone loses crucial time flying into a narrow space and back out of this trapping space without making progress in flying to the goal or back home.

5.5. Requirements

Lastly, the robot design and operator might be subjected to requirements from governments or landowners. For ground robots, like rovers, there are often fewer regulations as a risk to the environment by a rover is assumed to be minimal. For drones, on the other hand, there are by default already tens of requirements, but also some more strict ones introduced by the type of operation expected in foresty environments.

For drones, flying Beyond Visual Line Of Sight (BVLOS) has, for nearly any country worldwide, an extra set of requirements. This BVLOS operation occurs as soon as the drone is out of the pilot's sight behind some tree or other vegetation. Therefore, BVLOS requirements must be considered for operations in foresty conditions.

Some examples of requirements BVLOS might add to the operation are more advanced geofencing, additional pilot training, transponder for aviation, and real-time video streaming. BVLOS often has to be requested a couple of days in advance to be approved by the flight authorities. These will check the pilot's licences and the risk assessment of the drone in combination with the location of operations. BVLOS might not only be more costly due to the requests and extra pilot training but also, the required hardware on the drone might add extra weight. This additional weight can have a significant impact on small drones with only a couple hundred grams of payload, greatly reducing the flight time. Every region has its own set of requirements and limitations that should be checked.

5.6. Challenges summarised

To wrap up the highlighted challenges in this chapter, the rainforest presents unique challenges for autonomous systems with its dense vegetation, high temperature and high humidity. The temperature and humidity significantly impact the functionality of electronics. Sensor readings can be affected to give wrong values, causing the system to function incorrectly or even unstable. Also, connectivity is significantly impacted by the humidity. Signals scatter and degrade through the humid air and even more by thick and wet foliage. This potentially causes signal loss earlier than expected.

Missions under the canopy in this complex terrain have their challenges as well. Slow traverse, poor or no GPS localisation, and returning home before the battery dies are examples thereof. In addition, flying under the canopy will commonly be Beyond Visual Line Of Sight (BVLOS) missions. BVLOS missions have higher restrictions on supervision, while the signal to stream video and control the drone is worse under the canopy. Therefore, meeting these requirements when flying under the canopy while monitoring the drone remotely is hard.
Appendix E.1 lists some thoughts about future work to improve the EGO planner to be used in the rainforest as required by the Xprize competition.

6 Downscaling: the tradeoff between performance and processor load

Multiple research gaps are found at this stage of the research. From those gaps, the one that shows the strongest relation to the topic and objectives of this research was chosen to study. This research gap is downscaling hardware and algorithms towards small sub-250-gram drones while optimising for a prolonged flight time, and, at the same time, having a robust system which can safely traverse to its goal in environments with narrow gaps and cluttered obstacles. The study aims to provide guidelines and insights for researchers and engineers on how autonomous navigation performs on smaller processors. The feasibility of applying a state-of-the-art path-planning algorithm to smaller systems is assessed through systematic testing with various configurations.

Downscaling a drone improves many factors of autonomous rainforest exploration. First, a more lightweight autonomy system extends the maximum flight time, which helps to explore more terrain. Additionally, the weight reduction potentially reduces environmental impact as it does less harm on impact when crashing, and it also produces less noise as it uses less thrust.
Secondly, the smaller form factor enables the drone to navigate through smaller gaps and denser environments, increasing accessibility and improving the drone's ability to work in more versatile terrain. And lastly, downscaling components down to a certain level reduces costs. Reducing costs makes the system more accessible to deploy on a large scale.

While adjusting algorithm settings may reduce processor load, potential performance trade-offs are acknowledged. To quantify and compare performance, dedicated metrics are established to assess configurations' effectiveness in robustly navigating a rainforest environment. These metrics offer insights into environmental density, sensitivity to obstacles, resolution of trapping conditions, and response time to emerging obstacles, establishing a direct link between processor load and performance changes.

Simulations on one specific dataset from outdoor flights are used for benchmarking and evaluating the planner's performance across various configurations. The uniform scenario ensures consistent testing conditions for diverse planner setups. Then, data is processed into Python format, and analysis and visualization are executed through a Python script. The script generates plots of CPU and memory load per simulation, performance metrics, and combined CPU load against performance metrics, accompanied by explanations of observed trends.

6.1. Performance metrics

In order to determine the optimal settings for path planning on a processor with limited resources, benchmark tests and measurements are conducted. During the tests, the CPU and memory load of the processors can be measured. However, to accurately describe the performance of navigation and obstacle avoidance, certain metrics are required to make the performance measurable and comparable. This section outlines the metrics that have been designed for this purpose.

As this research focuses on safe navigation in dense and cluttered environments, the metrics have been designed with these objectives in mind. Four metrics have been developed and presented, namely 'reactiveness time', 'solvable obstruction width', 'minimum detectable object size', and 'minimum solvable gap size'. In this section, the differences between perception and planning are highlighted to provide a better understanding of the reasoning behind the metrics. The metrics are then explained in detail, with background information, illustrations, and formulas. Finally, this section describes how the metrics will be tested on the benchmark simulation.

6.1.1. Perception vs Planning in Relation to Performance Metrics

Autonomous planning for robotics involves two essential components: perception and planning. The perception step processes the robot's environment perception and the generation of a map for subsequent planning. The planning step utilizes this map to navigate around obstacles and lead the robot toward its goal. In this research, the EGO-planner integrates both perception and planning steps.

Understanding the influence of different configurations on the path planner is crucial. Notably, the study's initial findings revealed that processing load is more significantly impacted by perception configurations than planning configurations. Moreover, the performance metrics designed to achieve reliable flight in the rainforest are closely linked to perception configurations rather than planner configurations.

Perception configurations, such as depth image rate and resolution, influence the processing load significantly. Additionally, certain configurations are used in both the perception and planning modules, such as voxel grid resolution and local map size. On the other hand, planner configurations encompass parameters related to the cost function, maximum iterations, and replanning requirements. These configurations are important for performance but are assumed not to significantly change processor load.

By aligning the discussion of perception and planning with the performance metrics, this section provides valuable insights into the differences, similarities, and relations between these components and their impact on the overall performance of autonomous flight in challenging environments.

6.1.2. Depth image projection

In depth-image projection, the 2D depth image, provided by the camera, is translated to 3D detections. This is done by using the camera's intrinsic and extrinsic matrices [20] to determine the position of each depth pixel in relation to the camera coordinate frame.

To translate the detection to the coordinate frame of the obstacle map, we use the position and pose estimation of the drone coordinate frame, as well as the transformation from the drone's coordinate frame to the camera's coordinate frame. After this translation, each detection point is assigned to the appropriate grid block, and the obstacle probability of this specific block is increased.
Figure 6.1 provides an illustration of how image pixels are projected into 3D detection points in the world coordinate frame.

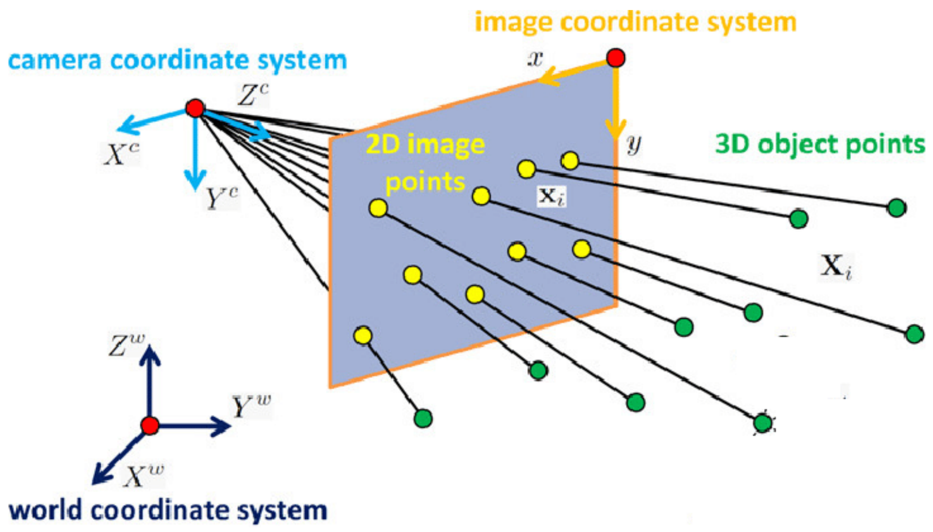


Figure 6.1: Illustration of depth image projection [27]

As part of the simulation, we will be lowering the resolution of the depth camera to understand how it impacts the drone's perception performance. This is particularly interesting because using a high-resolution camera, but only using a fraction of the pixels, may impact CPU and memory load. In addition, this also simulates the effect of using lower-quality hardware that, by default, may output lower-resolution depth images. Therefore, we want to explore how the CPU and memory load are related to the perception performance with different resolution configurations.
However, it is important to note that when skipping pixels, small obstacles may be missed if they only appear in the skipped pixels. To illustrate this point, figure 6.2 below provides a side-view 2D illustration highlighting the "blind zones" caused by skipping pixels.

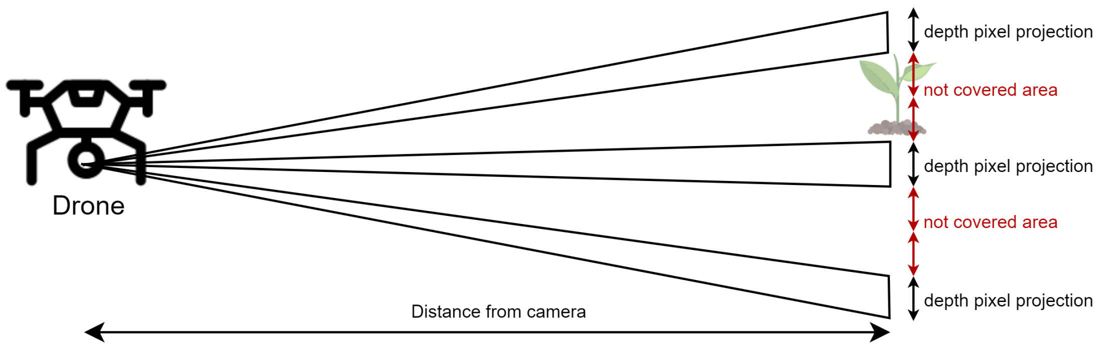


Figure 6.2: Illustration of skipping 2 out of 3 pixels in the projection step

To calculate this minimal size, first, the original pixel size has to be calculated. This represents the projection of a single pixel at a certain distance from the camera. By using either the camera's horizontal resolution and horizontal field-of-view, or the camera's vertical resolution and vertical field-of-view, and the camera range, the pixel projection size at maximum distance can be determined.

To estimate the size of a potentially missed object at maximum camera distance, formula 6.1 below can be used as a rule of thumb. In this formula, PP_{max} is the maximum pixel projection, FOV the camera's field of view, CR_{max} the camera range, OR the original resolution, SP the number of pixels that is skipped, and SM_{max} the maximum object size that can be missed. To determine the minimum size, first calculate the original pixel projection using either horizontal or vertical resolution and field-of-view, along with camera range. For small objects, multiply the projection size by the number of skipped pixels.

Note: To be safer, assume the object is at the edge of pixels and add two extra pixel sizes. The diagonal distance between pixels should be calculated for geometrically correct dimensions.

$$\begin{aligned} PP_{max} &= \frac{\tan\left(\frac{\text{FOV}}{2}\right) \cdot CR_{max} \cdot 2}{OR} \\ SM_{max} &= SP \cdot PP_{max} \end{aligned} \tag{6.1}$$

The pixel projection scales linear with the range from the camera. Therefore, at half the maximum camera range, objects twice as small can be detected.

Hypothesis
The hypothesis is that reducing the number of pixels will decrease the CPU load as there will be fewer points to project into 3D space and raycast. Consequently, if we need to detect smaller obstacles at a certain distance, we may experience a higher CPU load.

Depth image frequency and reactiveness
Besides skipping pixels in an image, depth image frequency can be varied to potentially reduce the processor load. By only processing one out of every N-received depth image, the ego planner can simulate a lower depth image frequency. This technique is particularly useful in simulating lower-performance camera hardware, which might have a lower depth image frequency. Therefore, it is important to test the system's performance on this variation. For instance, the standard depth image output frequency of the used Realsense Depth camera is 15Hz. By skipping one out of every two images, a rate of 7.5Hz can be tested. Similarly, by skipping two out of every three images, a rate of 5Hz can be tested.

Reducing the frequency of depth images can have a downside as it may affect the planner's reactiveness. Reactiveness refers to the time it takes for the planner to detect an obstacle on the generated path and realize that the current path will collide with it. When the algorithm receives fewer images per second, the time interval between each image is longer, which may cause delays in processing new inputs. Additionally, the probability function for mapping obstacles in the voxel grid requires N iterations before reaching the threshold that considers the obstacle.

Assuming that the loop times for perception are much smaller than the time period between two depth images, the reactiveness time is only added once. Perception loop time includes projection, raycast, and update loop times. To calculate the total reactiveness time, we need to calculate the time taken by the probability function (which is the number of images multiplied by the camera delay) and add the time taken by the perception and collision check loop. This can be written as a rule of thumb as in formula 6.2 below. In this formula, PI is the number of iterations until the threshold is reached, P_t and P_i are respectively the probability threshold and probability increase, R_{cam} the camera rate, and T_{proj}, T_{ray}, T_{upd}, T_{col} the loop times for projection, raycasting, updating, and collision check. The square brackets, also known as the ceiling brackets, indicate that the decimal number inside should be rounded up to the nearest whole number.

$$\begin{aligned} PI &= \lceil \frac{P_t}{P_i} \rceil \\ T_{react} &= PI \cdot \frac{1}{R_{cam}} + T_{proj} + T_{ray} + T_{upd} + T_{col} \end{aligned} \tag{6.2}$$

As described in the later section 6.4, during testing, it was found that the loop times for perception and collision checks were 10 to 100 times lower than the delay caused by the depth image rate. Therefore, the reactiveness time can be simplified by removing the perception and collision check loop times from the equation. This simplified formula provides an approximate value for the reactiveness time, as shown in formula 6.3 below.

$$T_{react} \approx \lceil \frac{P_t}{P_i} \rceil \cdot \frac{1}{R_{cam}} \tag{6.3}$$

Hypothesis
It is expected that when the reactive performance is plotted against CPU load, the CPU load will be

highest for the highest depth image rate, resulting in the best reactiveness. As the image rate decreases, the CPU load will also decrease, resulting in longer reaction times. It is uncertain whether the relationship between CPU load and reactiveness will be linear or a higher degree polynomial. There is no direct relationship expected between memory load and the probability function or camera rate, as these do not influence matrix dimensions in the code.

6.1.3. Raycasting process

When creating a 3D map from a depth image, the raycasting algorithm checks if previously detected obstacles should be removed from the map. The algorithm makes the assumption that if the drone detects an obstacle at point B from point A, there can't be any other obstacle on the line between point A and point B. Therefore, every block on that line (also called ray) that was previously marked as an obstacle will now be updated towards free space. The algorithm does this by lowering the obstacle probability.

For every grid block within the drone's range, there is a value between 0.0 and 1.0. This value is updated using a probability function that increases or decreases it. A threshold value determines whether a block is marked as free space or an obstacle. Figure 6.3 below illustrates the raycasting step. In this example, both the solid and dotted blocks were previously marked as obstacles. Now, the drone raycasts through the dotted blocks, causing the blocks to have a lower probability of being obstacles and potentially being removed from the map.

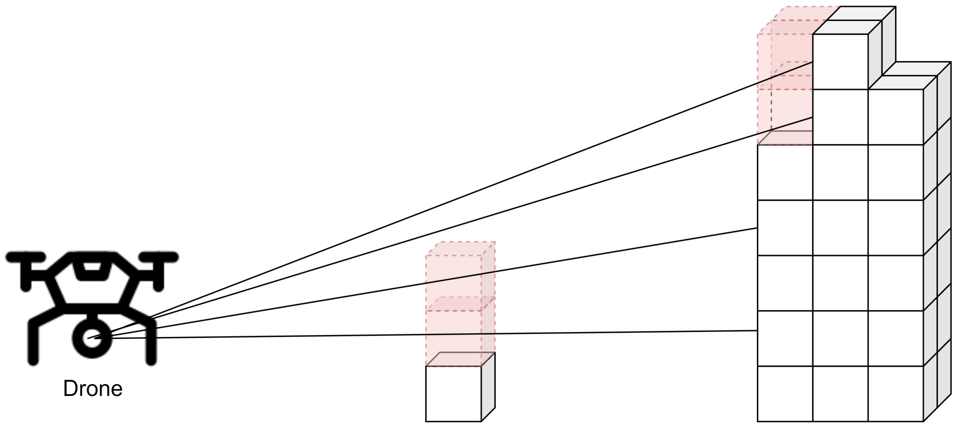


Figure 6.3: Illustration of raycasting process. After being raycasted, the red blocks are removed from the obstacle map.

In a perfect world, where the drone knows its exact position and pose, and obstacles are static, there would be no need for raycasting as every obstacle can be accurately mapped. Unfortunately, the position and pose estimation with dead-reckoning drift are not very accurate, which makes raycasting an essential process to keep updating the map.

For example, when the drone is in front of a narrow gap, and it drifts to the right, the left edge of the gap becomes an obstacle, making the gap smaller. In such a situation, the raycasting step detects that the first part of the right edge has disappeared and updates the obstacle map. Consequently, the gap has roughly the same width as before the drift. Therefore, raycasting plays a crucial role in navigation while flying dead-reckoning with drift in the position estimate.

6.1.4. Local planner range

The ego planner is a local planner that considers only a limited set of obstacles for navigation. In contrast, a global planner maps and considers all obstacles and can plan a route through them all, thereby using more memory and processor. The ego planner's obstacle avoidance algorithm stores and considers obstacles up to a certain distance from the drone's location. Additionally, all camera detections within the maximum camera range are included in the obstacle avoidance algorithm. The maximum camera range can also be configured in the ego planner.
A smaller local planner range means fewer obstacles to consider, resulting in potentially lower CPU load

and memory usage. However, as the planner takes fewer obstacles into account, it may not efficiently plan for obstacles ahead, resulting in less feasible or less efficient routes.
Choosing a local planner range smaller than half the obstacle width can result in the drone being unable to find a solution. This issue is illustrated in an example in figure 6.4 below.

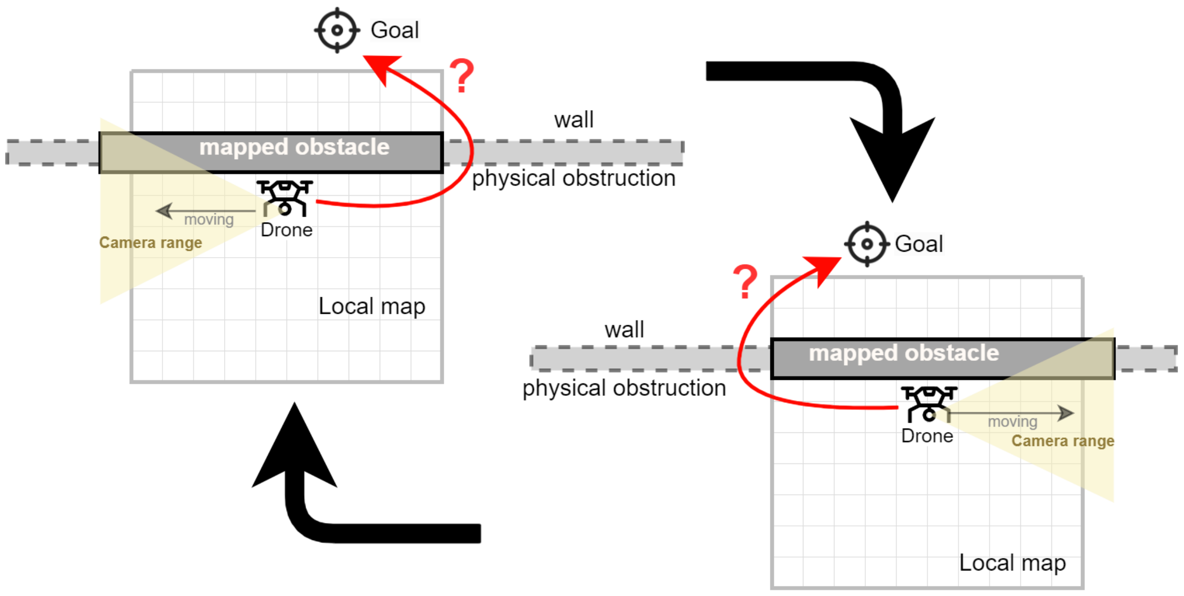


Figure 6.4: Illustration of local map range problem with wide obstructions where navigation goes into infinite loop

As mentioned earlier, all obstacles from the local map and within the camera range are considered in the trajectory planner. In the illustrated example, when the goal is behind a wall having a certain width, the drone might never reach the goal. When the camera range is bigger than the local planner range, the drone sees a longer wall ahead compared to the area behind the drone. Now the planner thinks the best path is turning around and flying to the goal. After turning around, the camera now detects a new piece of the wall. Also, everything behind the drone, outside the local map, is again forgotten. The planner now thinks turning around and trying the other way is better. From experience by testing, it can be recalled the navigation can stay forever in this loop, never reaching the goal.

When the local planner range is equal to or bigger than the camera range, this problem might still occur. When the obstruction, which can be a set of obstacles, in between the drone and the goal is wider than the local planner range, the same situation can exist, as shown in figure 6.4.
As a rule of thumb, formula 6.4 can be used to determine what obstruction width can be solved by the algorithm. In the formula, D_{infl} is the inflation distance, D_{min} the minimum inflation distance, GR the voxel grid resolution, R_{loc} the local range, and SOW the solvable obstruction width. The square brackets, also known as the ceiling brackets, indicate that the decimal number inside should be rounded up to the nearest whole number.

$$\begin{aligned} D_{infl} &= \lceil \frac{D_{min}}{GR} \rceil \cdot GR \\ SOW &= R_{loc} - 2 \cdot D_{infl} \end{aligned} \tag{6.4}$$

Hypothesis
The hypothesis for solvable obstruction width performance is an increase of CPU load by an increase of obstruction width. Both the perception and planner algorithms have to process more voxel blocks when the local map is bigger. Also, when the local map is bigger, the voxel map contains more blocks and, therefore, increases the processor memory load. Whether the relations for CPU and memory load with the solvable obstruction width are linear or a higher polynomial is unsure.

6.1.5. Minimum solvable gap size

To guarantee a path-planning solution from a perception perspective, the drone requires a minimum gap size which ensures at least one grid block of free space in the middle of the gap. This minimum gap size is used as a performance metric. In most cases, depending on the voxel grid's initialization, it is possible to find a more narrow gap than this minimum requirement is based on. But as this does not hold for all cases, these more narrow gaps cannot be guaranteed. This section assumes the worst alignment of pixels and grid blocks that can occur.

When obstacles are mapped into the local map, the centre of the depth image pixel is projected to 3D space and assigned to one of the voxel blocks in the grid. This projection represents a cross-section of the ray from the camera that corresponds to a specific pixel. The size of the projection varies depending on the distance from the camera; it is relatively small up close and relatively large when far away. This is illustrated in section 6.1.2, figure 6.2.
If the projection of a pixel is mostly outside of an obstacle, but not completely, the pixel can still represent the obstacle by giving it the depth of the obstacle. In cases where the pixel projection is located on the edge of an obstacle, half the pixel-projection-width narrows the gap at both edges.

After projecting the centre of the pixel to the 3D space and assigning it to a voxel block, it might be on the edge of a voxel that mostly lies towards the gap's centre. At worst, this would narrow the gap by one grid size on both sides in the obstacle map.

Up until now, we have only talked about the grid blocks present in the obstacle map. However, inflating the obstacles is a crucial step towards ensuring a safe flight. The inflation of obstacles defines the margin that the trajectory planner maintains between the centre of the drone and the mapped obstacles. For the inflated obstacle map that the planner uses, each voxel in the obstacle map is inflated in all three directions (X, Y, and Z), both negative and positive, by a value known as the **minimum inflation - parameter**. The inflation value is always a multiple of the voxel grid size. For instance, if the grid size is 15cm, then the inflation value could be 30cm or 45cm. If the inflation value is not a multiple of the voxel grid size, the planner will automatically choose the next bigger multiple.

In order for the planner to find a solution, there must be at least one obstacle-free grid block in the middle of the gap on the inflated obstacle map. To calculate the width of the gap, we can take into account the following factors: half a pixel projection width from each edge towards the middle of the gap, plus one grid block width, and finally adding the inflation width. In the middle of the gap, there should be one grid block of free space as well. This is illustrated in figure 6.5 below.

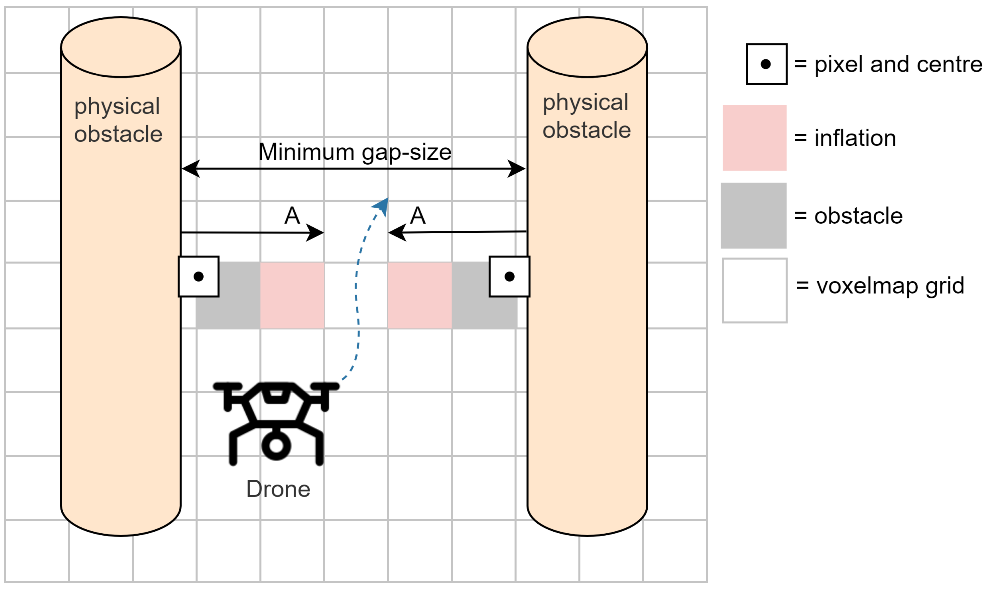


Figure 6.5: Illustration of minimum gap size for a set of configurations

The above reasoning applies only when the drone is facing a gap that is perpendicular to the x, y, or z axis. However, in cases where the gap is rotated relative to the axis, the reasoning is no longer valid. When the gap is diagonal to the axis, the obstacle mapping and inflation remain the same, but the span of the grid block becomes wider. In such cases, the width of the grid block is equal to the diagonal span of the voxel block, instead of the resolution of the voxel grid map. Figure 6.6 below provides an illustration of a diagonally oriented gap with respect to the grid axis.

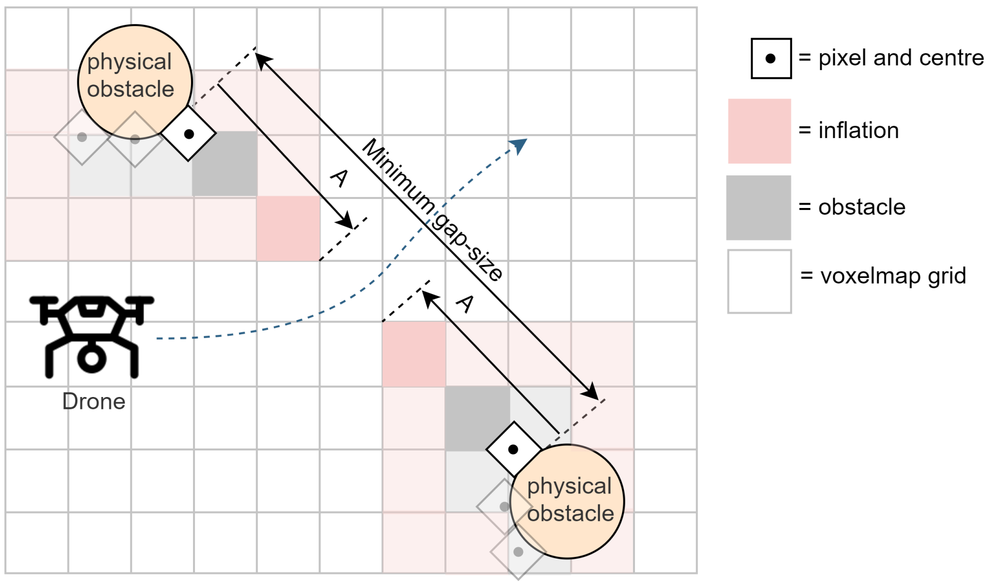


Figure 6.6: Illustration of minimum gap size for a gap diagonal to the grid axis

From these conclusions, we can make a rule of thumb for the minimum gap size that a drone can navigate through with numerical certainty. This rule of thumb is formula 6.5 below. In the formula, PS is the pixel size, FOV the camera's field of view, R_{cam} the camera range, C_{res} the camera resolution, D_{infl} the inflation distance, D_{min} the minimum inflation distance, GR the voxel grid resolution, and GS_{min} the minimum solvable gap size. A refers to the dimensioning illustrated in figure 6.6. The square brackets, also known as the ceiling brackets, indicate that the decimal number inside should be rounded up to the nearest whole number.

$$\begin{aligned}
PS &= \frac{\tan\left(\frac{FOV}{2}\right)\cdot R_{cam}\cdot 2}{C_{res}} \\
D_{infl} &= \left\lceil\frac{D_{min}}{GR}\right\rceil\cdot GR \\
A &= \frac{PS}{2} + \sqrt{2\cdot(GR)^2} + \sqrt{2\cdot(D_{infl})^2} \\
GS_{min} &= A\cdot 2 + \sqrt{2\cdot(GR)^2} \\
GS_{min} &= PS + 3\cdot\sqrt{2\cdot(GR)^2} + 2\cdot\sqrt{2\cdot(D_{infl})^2}
\end{aligned} \tag{6.5}$$

Hypothesis
The hypothesis for the minimum gap size that can be guaranteed to be solvable is that the CPU load increases with decreasing gap sizes. This is because a smaller gap requires the planner to process a higher-resolution voxel grid, which then means more voxel blocks have to be processed. Additionally, a higher image resolution for a smaller gap may also increase the CPU load since more pixels have to be processed.

For the processor memory load, the hypothesis is that a higher voxel grid resolution will require more memory to store the local obstacle and inflation map. No significant memory difference is expected for variations in image resolution as this does not affect the 3D matrix of the obstacle map.

6.1.6. Configuration variations

In order to compare the performance of the planner with the processor and memory load, configurations of the planner will be varied over a set of simulations. This section explains which configurations can be varied and why they are chosen or not. The number of parameters to vary is limited to six, as testing more would take too much time to process and analyze the data. To make the final results more readable, we will eliminate variations that show no significant difference. The six parameters to vary are voxel grid resolution, local map range, depth camera range, skip pixels, and image rate. These are described below.
This section also covers briefly why certain parameters are not chosen to vary. At the end of this section, a table is given with an overview of the planned variations.

The voxel grid resolution refers to the block size in meters used to cluster the detections of obstacles in a map. This parameter relates to the performance of minimal gap size and solvable obstruction width. This parameter significantly affects the processor and memory loads, as a finer grid requires more iterations to map the obstacles, generate trajectories and check for collisions. In addition, a finer grid results in a larger matrix representing the same volume, thus increasing the memory load.
Intuitively, for minimal gap size and solvable obstruction width, the variation is chosen to be [0.05m, 0.10m, 0.15m, 0.20m].

The local map range is the area around the drone that the local planner uses to create new trajectories. If the range is larger, there are more grid blocks to consider as obstacles, which increases the number of iterations and processor load. The local update range also limits the maximum width of an obstruction that the drone can navigate without getting stuck in a loop, which impacts its performance. For solvable obstruction width, the variations on the local map range are chosen to be 1 meter up to 35 meters, with steps of 1 meter.

The depth camera range refers to the farthest distance that the camera can detect obstacles or the distance limit set in the code. This affects the minimal gap size, maximum obstruction width, and minimal object size.
One of the goals is to use a smaller, lower-performance depth camera that can detect obstacles up to 3 to 4 meters instead of 6 to 8 meters. Therefore, it is crucial to measure the difference in performance when using a camera with a smaller range because the drone will not be able to detect obstacles as far in advance.
The depth camera range is chosen to be [4m, 6m, 8m] to simulate both the performance by using the current camera, as well as the performance by using the lower-performance camera.

Depth image skip pixels is the number of pixels skipped by the mapping algorithm. For example, a value of 1 means it moves to the next pixel without skipping any. A value of 4 means it skips three and moves to the fourth pixel. If the skip value is set to 4 on a 720x480 pixel image, the algorithm divides the rows and columns by 4, resulting in an image size of 180x140 pixels. Reducing the number of depth pixels reduces the points to project and raycast in the obstacle map, which affects the algorithm's performance of the minimum obstacle size it can detect.
To analyze the parameter's effect, skip-pixels will be varied with [1, 2, 3, 4] pixels, which represent a full, half, one-third, and quarter-sized depth image.
Keep in mind that in a half-sized image, half of the rows and half of the columns remain. In this example, therefore, there is only a quarter of the pixels left to process.

Skip depth image refers to the number of depth images that are skipped before the next one is processed. The planner receives these depth images from the camera. By skipping a certain number of images, such as 2, only 1/3rd of the original rate remains.
For instance, if the camera captures images at 15Hz and skips two images, the rate will drop to 5Hz. This rate has an impact on the planner's responsiveness performance.
Initially, this parameter was not present in the EGO-planner code. However, it was later introduced to simulate a low-performance camera that captures images at a lower rate.
For the reactiveness test, there are four variations of skip-pixel [0, 1, 2, 3, 4]. These values correspond to image rates of 15Hz, 7.5Hz, 5Hz, 3.8Hz, and 3Hz, respectively.

A maximum of 2 parameters is varied per test. If more than two parameters influence a specific performance, the two most influential variations are chosen for simulation. Table 6.1 below summarizes the variation settings for each performance test for the five parameters that will be changed during simulations.

Table 6.1: Per performance test, the settings and variations for the simulations

Perfromance test	**Voxelgrid resolution [m]**	**Local map range [m]**	**Camera resolution [-]**	**Camera rate [Hz]**
Gap size	[.05 .10 .15 .20]	10	[1 1/2 1/3 1/4]	15
Obstacle size	0.10	10	[1 1/2 1/3 1/4]	15
Obstacle width	[.05 .10 .15 .20]	[1 2 ... 34 35]	1	15
Reactiveness	0.10	10	1	[15 7.5 5 3.8 3]

The EGO-planner has other parameters that can be modified to manipulate its behaviour. Below are the parameters that are not varied, with a brief explanation of why they are not chosen for variation.

- The total size of the map in the x, y, and z directions is a fixed parameter. This parameter is determined by the available memory on the processor board, which sets the maximum size that can be used. While it does limit the operation range of the drone at some point, it does not directly affect the performance of safe and robust flight in the forest.
- The maximum velocity and acceleration, as well as the cost function for the trajectory planner, will not be varied. The parameters specific to the planner have not been chosen, as they have a smaller impact on safe and robust flight in a forested area. However, these planner parameters might have delivered some interesting results in terms of flight performance for metrics other than the four designed metrics in this research.
- The minimal inflation is not varied as this is determined by the size of the drone platform, plus an extra margin for obstacle avoidance defined by the user.
- The parameters for mapping probability are kept static. However, this can affect the system's responsiveness by changing the number of iterations required for an obstacle to be marked.

6.2. Dataset recording

Creating a dataset for simulations involves several steps. Firstly, it is necessary to modify ROS nodes to produce real-time data that can be used to analyze the status of specific processes. In addition, extra measures are taken to gather ground-truth data during the dataset collection process. Afterwards, the dataset collection has to be planned. This includes deciding on the location of obstacles and the drone's movement during the recording. This section will discuss the steps taken to create the outdoor dataset in detail. The recording of the indoor dataset can be read in appendix C.1 but is not described in this section as this dataset is not used in further steps. Appendix C gives more details on the creation of the datasets, with some enlarged figures.

6.2.1. Prepare ROS to output measured performance

To prepare for data processing after the dataset collection, several nodes and topics are added to the system. These are necessary to calculate and measure performance. This section focuses on the changes made to the planner algorithm and explains how processor and memory load are measured.

Extract planner information

In a robotic system using ROS, multiple nodes process specific information and communicate by passing data through a network. Once the system is designed, debugged, and ready to use, the nodes can be combined into one. This makes the system work smoothly and eliminates the need for the ROS network. However, in the ego-planner code, the merging process has already been done, making it difficult to monitor data traffic between different functions without modifying the code. To measure and

keep track of algorithm performance, ROS publishers are added to the algorithm, allowing specific data from inside the algorithm to be shared. Reverting the ego-planner to a setup with multiple nodes and topics would require a lot of work, and the outcome would be similar to the current solution, with the addition of extra publishers.
Publishers added to the ego-planner algorithm to measure performance per function:

- Trajectory planner
  - Initialisation step. Loop time (seconds)
  - Optimisation step. Loop time (seconds)
  - Refining step. Loop time (seconds)
  - Replanning average time. Loop time (seconds)
  - Replanning iterations before finding a valid trajectory. (-)
  - Total time to generate the valid trajectory. (seconds)
  - Collisioncheck of current trajectory. Loop time (seconds)
- Environment planner (mapping)
  - Projection of depth image to voxel grid. Loop time (seconds)
  - Raycasting of new voxel grid on total voxel grid. Loop time (seconds)
  - Local map updating step. Loop time (seconds)

Measure processor and memory load
In order to measure the processor load and active memory usage, a ROS package called 'cpu_monitor' [28] is used. This package contains a ROS node that reads the CPU and memory load for each ROS-related process running on the hosting system and publishes the values to corresponding topics. The node is programmed to read and publish these values at a pre-set interval of 200 milliseconds.

6.2.2. Creating ground truth position data

As one of the research objectives is to measure the performance of a path-planning drone system using dead-reckoning, it is useful to collect ground-truth data to monitor the drift. However, different methods need to be used for indoor and outdoor scenarios as indoor ground-truth methods cannot be used outdoors and vice versa. In this section, the collection of position data using motion-capture systems, GPS, and Visual-inertial tracking are discussed.
For indoor scenarios, motion-capture data is the most logical choice due to its high precision and consistency. On the other hand, for outdoor ground-truth position data, both GPS and visual-inertial methods are applied side by side since it is unclear which method gives the best result. By using both methods simultaneously, the best method can be chosen during data analysis.

Motion-capture data
During indoor tests, ground-truth position data is obtained an Optitrack system. This motion-capture system estimates the position and pose of the drone by analyzing markers on the drone through cameras at different angles. The position estimate provided by this system has an accuracy that ranges between 5mm to 1mm.
To input this data onto the ROS network, the laptop connected to the drone is also connected to the Optitrack network via Ethernet. The data is then sent onto the ROS network using the ROS package 'Mocap_Optitrack' and can be read by the drone .

GPS position data
The first method to record the drone's location outdoors involves using an accurate GPS, which publishes LLA and NED (in XYZ) coordinates on the ROS network. For this dataset recording, a UBLOX F9P GPS with a helix antenna is used. The used helix antenna has a high sensitivity to receive poor or scattered satellite signals in the forest, as a result of which the GPS module is able to get a better location estimate. The ROS node used to read the GPS module is a part of the 'gps_navsat_driver' package. The package was adapted to make the processor work with an I2C GPS module. This adaptation to the I2c protocol was necessary due to the shortage of free serial ports on the drone's onboard

processor board.

During implementation and testing it was found that mounting the the GPS atop of the electronics gave a poor signal and bad accuracy. Upon further research, it was found that the used helix antenna was sensitive to EMC-radiation of the electronics and would therefore give no accurate reading. A spectrum analysis was performed to give insights in the signal strength and noise. Figure 6.7 shows the analysis results of mounting the GPS antenna atop the electronics and 30cm away from the electronics. During the spectrum analysis, it was observed that the GPS signal strength was 2-3 times stronger (blue line) when the antenna was placed away from the electronics than when it was mounted atop the electronics (black line). The difference between the blue and black lines is assumed to be signal noise at the GPS frequency generated by EMC radiation.

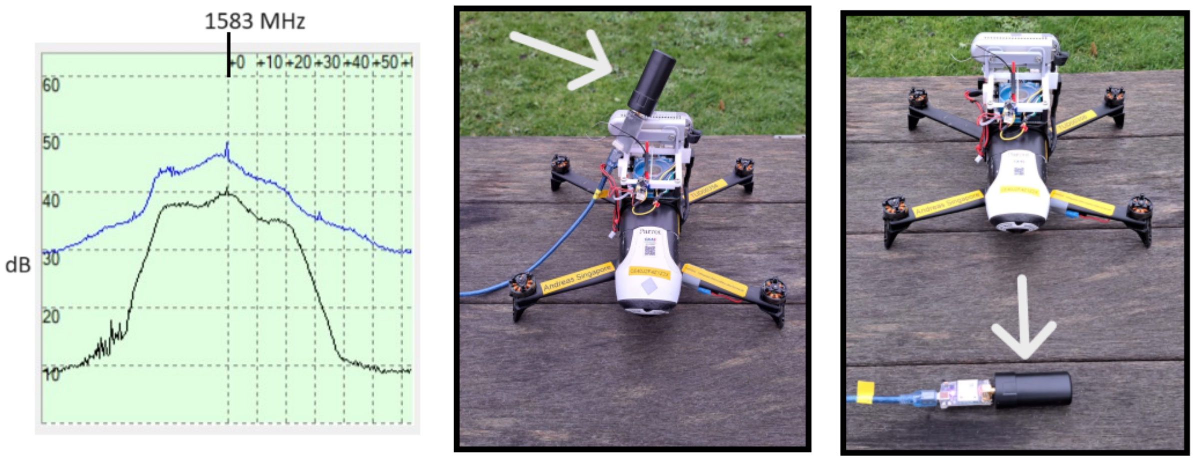


Figure 6.7: GPS spectrum analysis for antenna atop electronics (blue line) and antenna outside drone frame (black line)

To record the best possible GPS location on this drone, the GPS antenna is mounted on a carbon rod, at least 30cm away from these electronics, to reduce the influence of EMC radiation. Figure 6.10 in section 6.2.3 shows this GPS and carbon rod mounted on the drone. Now that the GPS is mounted with an offset to the centre of the drone, a transformation matrix is required to translate the GPS fix and drone heading into the actual GPS position of the drone.

Visual-inertial position data
A second method for obtaining outdoor ground truth position data is through visual-inertial odometry by the ROS 'VINS-fusion' package [21], as was used by the authors of EGO-planner. Due to its CPU-intensive nature, this method could not be utilized for real-time position input for the planner's onboard computing. Nonetheless, it can be run on a laptop with the best settings to obtain ground-truth information in post-processing.
To process the visual-inertial odometry, the recorded ROSbag file can be played back on the laptop. During the playback, by using the ROS 'VINS-fusion' package, the position can be estimated using feature tracking in the infrared images of the camera, combined with the camera's IMU. To achieve this, the correct ROS topics must be published and recorded. As a result, the depth camera is programmed to publish its IMU, infrared-left and right camera. These topics are also recorded in the dataset.

The figures below demonstrate how the images are post-processed to estimate the position of the object. Figure 6.8 shows the VINS-fusion algorithm's visualization of tracking the landmarks in the image. On the left, the optic flow between two consecutive images being processed is visualised with the red and blue dots. On the right, the landmark points are projected into 3D space. By keeping track of the landmarks in 3D space, the algorithm can estimate the positional and orientational changes between consecutive images.

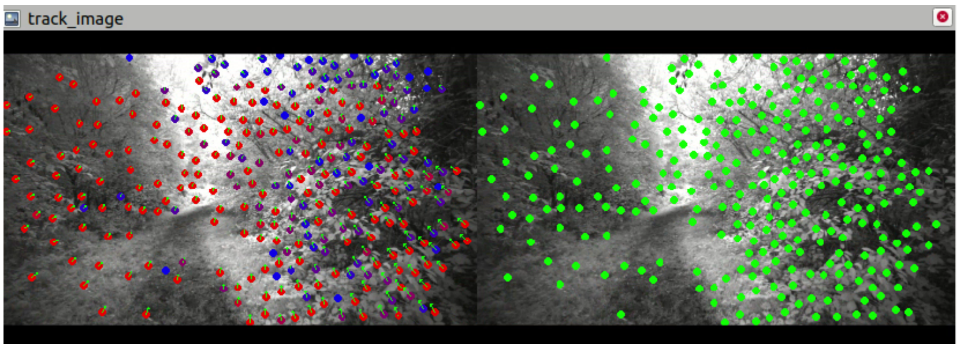


Figure 6.8: VINS feature tracking in the infrared image

In figure 6.9, the landmarks tracked in 3D space are represented as white dots. The VINS-fusion algorithm uses optic flow and landmark tracking to estimate the trajectory, which is shown as a green line. The red-green-blue icons in the figure represent the position estimates by the drone's IMU in real-time. In section 6.4.7, the estimated locations of VINS-fusion, GPS, and IMU are compared.

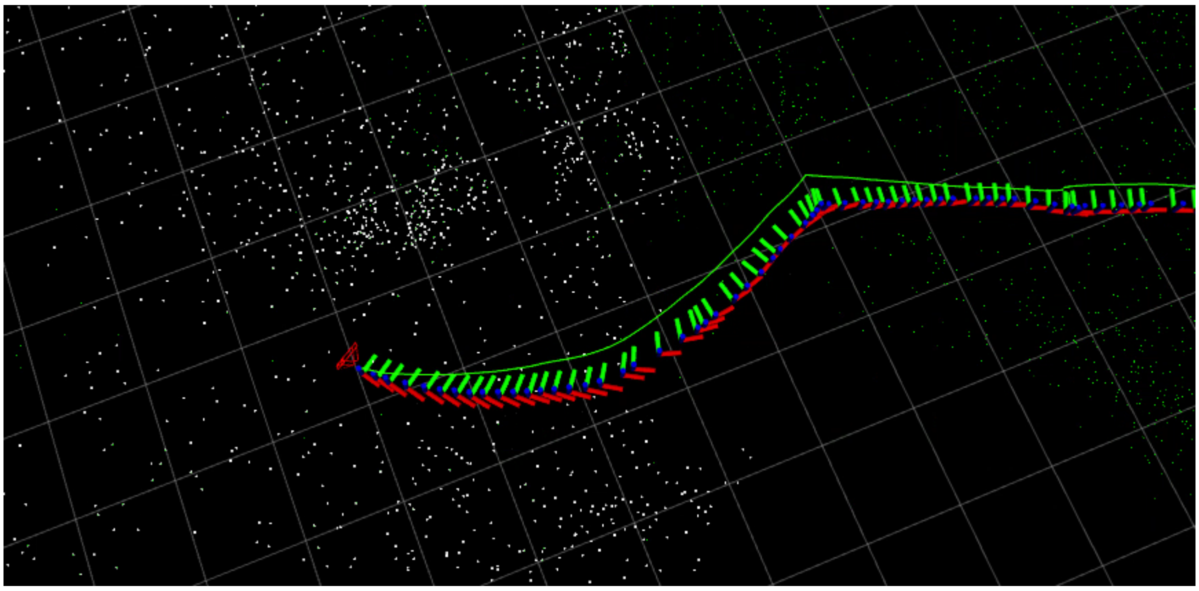


Figure 6.9: The drone's 3D position estimation of VINS-fusion and IMU visualized alongside the tracked features.

6.2.3. Record dataset outdoors

For the outdoor dataset, a forest is chosen to fly in and record sensor data. The forest was selected due to the relatively low height of its trees, which allows for GPS ground-truth position data. Additionally, a video of the test can be recorded from above with another off-the-shelf drone. To navigate through the forest, a walking trail with cluttered branches and multiple gaps of 1 meter was chosen. The drone has to navigate through a zigzag in the trail, which is approximately 20 meters long in total.
To record this dataset, first, a manual flight over the trail has been recorded without running the ego-planner. Afterwards, the drone navigated autonomously over the same trail using the planner. Since all autonomous flight tests over the trail were successful, the data from one of the autonomous runs was chosen to create the forest dataset.
The logged flight for the forest dataset had the ego-planner set to a maximum speed of 1m/s, maximum acceleration of 0.3m/s, and inflation of 0.3m.
In figure 6.10, an image is shown of the entrance to the zigzag trail to give an impression. In figure 6.11, a screenshot shows a top-view of the mapped trail to give an idea of its shape.

Figure 6.10: Image of drone flying autonomously in the forest for the forest dataset recording

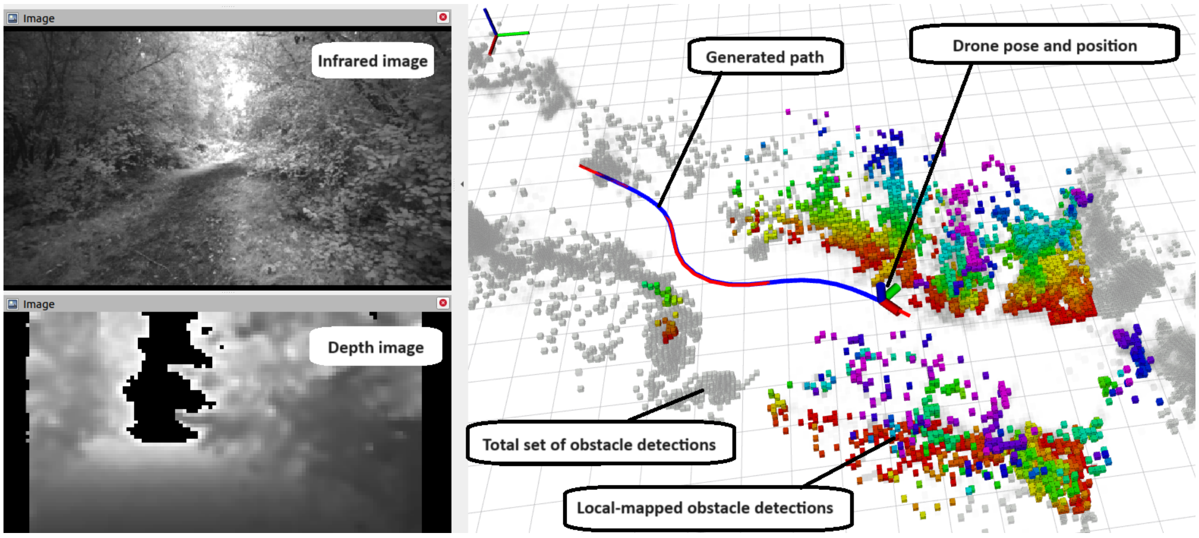


Figure 6.11: Screenshot RVIZ - ego-planner visualisation for forest dataset

6.3. Running simulations
To simulate a drone flight, the planner algorithm can now be run with different configurations using the recorded datasets. The ROS network receives sensor input from the dataset instead of the actual sensors, allowing processing to be done at a desk without the drone actually flying. This section describes the necessary steps to prepare the datasets, run the simulations, and extract the data for analysis.

6.3.1. Create a dataset for simulation
When using the logged data from a real flight as a dataset, it is essential to filter it to keep only the sensor input data. Additionally, the dataset needs to be trimmed to keep only the best usable 90-second fragment from the total time of the log. In this example, we will use the forest dataset to explain the process.

To find the best fragment, the ROSbag from the entire test flight is replayed and visualised in RVIZ. A fragment of 90 seconds is selected. For the beginning and end of this fragment, the time-of-week is noted. We then filter this fragment within these timestamps by using:

```
$ rosbag filter bos_6.bag bos_6_r4.bag ”t.secs>=1531425960 and t.secs<=1531426050”
```

The resulting ROSbag is now 90 seconds long but still needs to be filtered to keep only the sensor data. To accomplish this, the topics have to be specified that need to be included. These are depth-image and drone odometry. In addition, for ground truth, these are also included: GPS-position, camera-imu, camera-infrared-left, and camera-infrared-right. The following commands are used to filter the ROSbag to only keep the specified data:

```
$ FilterArgs=
”topic == ’/bebop/odom’
or topic == ’/camera/depth/image_rect_raw’
or topic == ’/camera/depth/camera_info’
or topic == ’/camera/infra1/image_rect_raw’
or topic == ’/camera/infra2/image_rect_raw’
or topic == ’/camera/imu’
or topic == ’/gps/gps_position_NWU’ ”

$ rosbag filter bos_6_r4.bag bos_6_r4_filtered.bag FilterArgs
```

A dataset named ”bos_6_r4_filtered.bag” has now been created to simulate the ego-planner. This dataset provides the advantage of repeatability and eliminates the need for field tests. It also offers a realistic representation of the planner’s performance in real forest environments.

6.3.2. Automated simulations and data generation

A bash script is created to run the planner on a dataset multiple times while varying the configurations. This script automatically loops over the variations and directly processes the results into Python data for analysis. The script enables the drone’s processor to run all desired simulations in sequence without requiring user input. It can be run via any device connected to the processor through WiFi, which can open an SSH terminal. This automated process improves the efficiency of simulation and analysis. For example, it enables batch execution of simulations overnight, ensuring timely availability of analysis data in the morning. This example is shown in figure 6.12 below.

Figure 6.12: Example of running automated simulations on the drone’s processor by a tablet with SSH terminal

First, in the bash script, the user specifies the dataset to be used and the parameter variations required. Thereafter, the user specifies the number of times the entire set of variations must be repeated. This feature is helpful in verifying the consistency of the outcomes.

The script utilizes nested for-loops to loop over the variations. During the simulations, the bash script prints progress updates in the terminal. Each simulation generates an output file in ’nohup’ format that can be used to trace bugs or issues. Furthermore, every simulation is logged into a ROSbag file, which is, after the simulation, filtered by required topics for data analysis. The bash script produces a folder containing ’nohup’ logs, a ROSbag file for each simulation and a ROSbag file with filtered data. Finally, the filtered ROSbag files are processed into a single ’pickle’ file. A pickle file is a file format compatible with Python data analysis to store data structures. This pickle file is named by the date-time stamp when the simulation was started, and by the name of the used dataset.

Due to the limited 1.5Gb RAM available on the processor board, it is unable to handle excessively large data files. Therefore, when the total number of variants exceeds 15, multiple pickle files are created. These pickle files have the same name but are numbered sequentially.

6.4. Data analysis

The generated Python data from the simulated variations can now be analyzed to draw conclusions. This section covers the steps involved in getting results and insights from the data. First, the data is imported and processed. Next, methods and plots for data visualization are presented. Subsequently, the reasoning behind the results and conclusions is explained in detail. Lastly, the localization accuracy of dead-reckoning is compared, as it was an important factor in the chosen navigational method. Appendix D shows additional results not shown in this section but used as useful insights on the complete downscaling research.

6.4.1. Data post-processing & analysis

When using Python to process the data off-board, for instance, on a Windows PC, the data produced by the simulations needs to be imported. During the simulation step, all data is prepared into a single large pickle file. During the processing step, this data can be easily imported by extracting this pickle file.

For the analysis, it is also possible to import multiple pickle files from different simulation sets. These sets can be merged into a single Panda's Dictionary, which contains the data from all imported pickle files. Inside this dictionary, each simulated run has its own designated Panda Dataframe, which contains all of the logged data from one specific simulation.
Each Dataframe has a timestamp index and various columns with every logged ROS topic. The structure of the data can be visualized as a tree structure, as shown below.

For the analysis, importing multiple pickle files from different simulation sets is also possible. These sets can be merged into a single Panda's Dictionary, which contains the data from all imported pickle files. Inside this dictionary, each simulated run has its own designated Panda Dataframe, which contains all of the logged data from one specific simulation.

Each Dataframe has a timestamp index and various columns with every logged ROS topic. The structure of the data is shown in here in figure 6.13

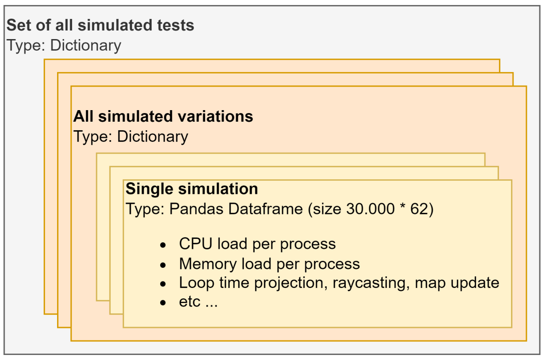


Figure 6.13: Data structure of simulation results for analysis in Python

When running a simulation, each Dataframe with logged data consists of 62 columns with about 30,000 time-stamped data points for a 90-second dataset. So, each Dataframe contains 1.86 million data points. If multiple variations per dataset are loaded into the analysis script, the total data becomes a multiple of the data within one Dataframe. Therefore, loading multiple simulation logs into the analysis script requires a reasonably powerful PC with sufficient RAM memory. If this would be an issue, the logs can be analyzed individually to extract necessary information and delete the rest.

The analysis of all data is handled in a single Python script. To calculate the performance and analyze it in comparison to the processor load, function definitions are written with the formulas for performance as described in section 6.1. These formulas include pixel size, minimum gap size, maximum obstruction width, minimum obstacle size, and reactiveness.

To analyze the CPU load of a simulated run, it is important to understand the difference between the scanning and planning phases when measuring the CPU load. During the scanning phase, the algorithm only processes depth images to update the local map. On the other hand, during the planning phase, the load peaks while the algorithm samples, optimizes, and refines a path. Once the first trigger is activated, a path is known, and the system checks continuously if the path collides with updated versions of the obstacle map. At every detected collision or trigger for a new path, a peak in the load is expected. Figure 6.14 provides an example of how CPU load is distributed during a simulation run.

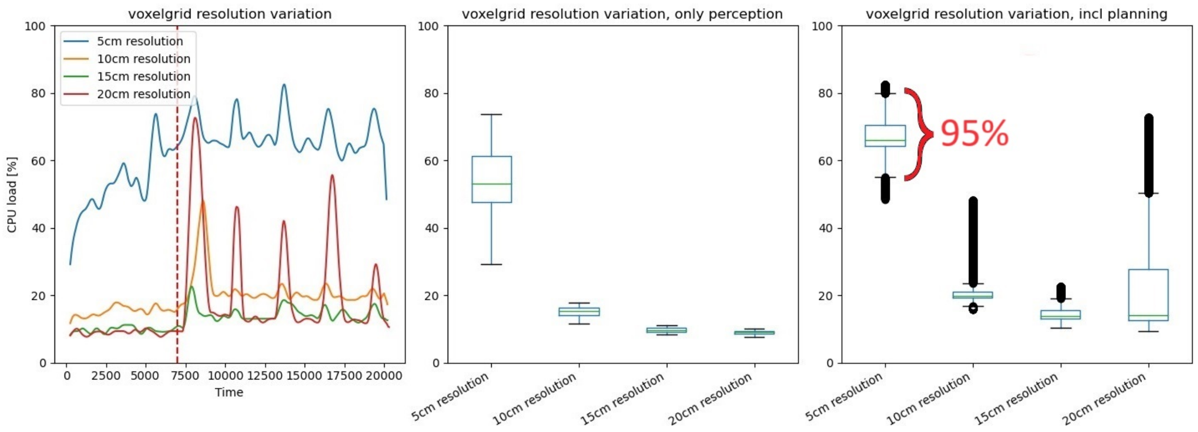


Figure 6.14: CPU load distribution example. Left: time graph for CPU in simulation set. Middle: CPU load distribution per simulation, with perception only. Right: CPU load distribution per simulation, with planning

In the example figure, a red dotted line is shown in the left graph, which separates the time graph into two parts - running perception-only and running perception, planning, and collision check. For visualisation purposes, this graph is smoothed by a running average filter of 1 second. For the middle and right plot, raw data is used. The middle boxplot indicates the distribution of CPU load from the start of the simulation up to the red dotted line. Since the load remains relatively steady during this state, the distribution also has a relatively narrow spread. On the other hand, the boxplot on the right represents the CPU load distribution from the red dotted line up to the end of the simulation. Due to the load's peak during path generation, the distribution is more spread out than the middle boxplot.

Peak loads pose a greater risk of system interruption and overload; therefore, these define the CPU load metrics over a simulation. Since CPU load logs also exhibit extreme outliers, the upper sigma-2 of the standard deviation is specifically chosen as load over a complete simulation. This sigma-2 encompasses 95% of the load measurements closest to the average load, filtering out the top 2.5% of the highest and lowest loads. To determine the maximum load within the sigma-2, the function *cpu_load.quantile([0.9544])* is used in Python.
The following sections compare this CPU peak load with the designed performance metrics. To generate the necessary data points for the comparisons, the simulated variations are analysed per performance metric. The parameters that are not varied are carefully selected so that the processor does not easily saturate at 100%. If the processor saturates, there will be no variation in load to measure, and the results will not show by what factor the load increases or decreases compared to other variations.

6.4.2. Results gap size performance

For the gap size performance test, the CPU load is measured for a varying voxel grid size of 0.05 to 0.20 meters per grid block. To get a better visualization of the trend, and possibly detect outliers, more variations are tested than initially designed. 40 variations in total.
The grid is varied width [.05 .06 .07 .08 .09 .10 .12 .15 .17 .20].
And the image resolution is varied by [full half 1/3rd 1/4th].

In figure 6.15 below, the CPU load on the vertical axis is plotted against the minimum solvable gap size on the horizontal axis. To interpret this figure: a smaller minimum gap size is a better gap size performance.
The plot indicates a non-linear relationship between gap size performance and processor load. When the planner is set to find smaller gaps, the CPU load increases, which aligns with the hypothesis that

an increase in gap size performance results in an increase in CPU load.

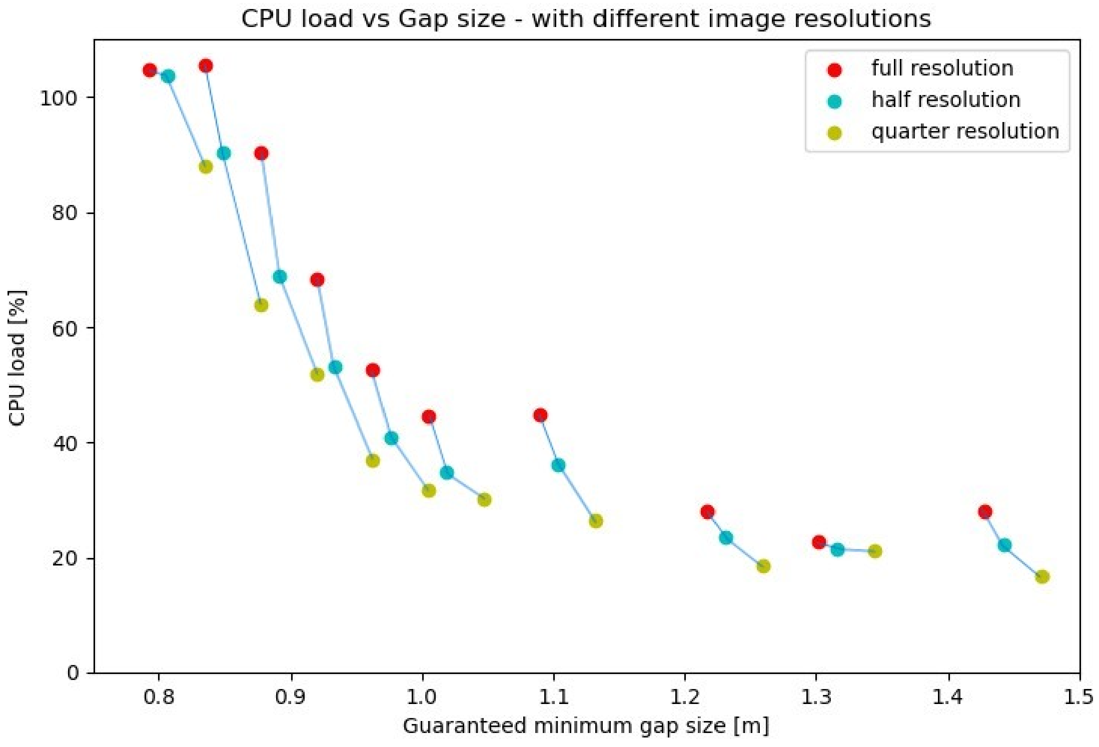


Figure 6.15: CPU load vs minimum solvable gap size for path planning [varying voxelgrid 0.05m-0.20m]

The plot displays the full, half, and quarter image resolutions. However, since there was no significant difference, the 1/3rd resolution was excluded to simplify the plot. To emphasize which image resolutions belong to the same voxel grid variation, dots are connected. These connected groups show a relatively small difference in CPU load for a large gap size, but a significant difference for a smaller minimum gap size.

The plot shows that for a fixed image resolution, there is a non-linear relationship between an 80cm gap and a one-meter gap. Even a small change in voxel-grid resolution can lead to a significant drop in CPU load. This happens because the obstacle grid map has three dimensions. When the grid resolution doubles, the number of grid blocks increases eightfold as 2^3 equals 8. For gap sizes of one meter and larger, the plot indicates a mostly linear trend. The CPU load remains relatively stable compared to gap sizes below one meter. This linear trend is due to the grid's significance in relation to other parameters. When the grid size becomes more coarse, other parameters such as image rate and resolution start playing a more significant role in CPU load.

Memory load

The resolution of the voxel grid significantly affects processor memory usage. Therefore, the gap size against memory usage is also plotted, shown in 6.16 below. As in the plot for CPU load, the sets of measurements with the same voxel grid resolution are connected.

Not so surprisingly, the memory usage follows the same trend shape as the CPU load. The same reasoning can explain this. Specifically, the memory usage for the voxel map increases by a factor of 8 when the grid resolution scales up by a factor of 2. Once the gap size reaches a particular threshold, the map size becomes less significant compared to other variables and processes that also use memory. At this point, the trend appears to be linear.

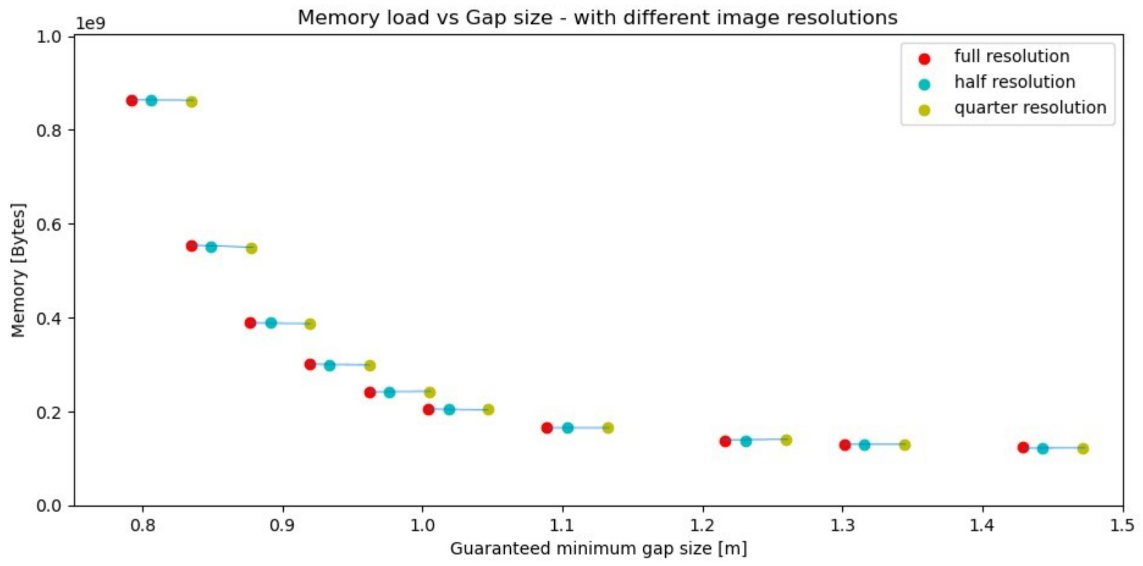


Figure 6.16: Memory load vs minimum solvable gap size for path planning [varying voxel grid 0.05m-0.20m]

6.4.3. Results obstacle size performance

Simulations were performed to test the CPU load for the minimum detectable obstacle size at the maximum camera range. In the simulation, the depth image resolution setting is varied by full, half, 1/3rd and 1/4th resolution.

Initially, with 10cm voxel grid blocks, the CPU load remained almost unchanged with different image resolutions. To validate this observation, the voxel grid resolution was also varied as this is the most CPU-load influencing parameter.

Subsequently, with three different voxel grid resolutions, the plot showed an almost linear relationship between the CPU load and the minimum detectable object size. Furthermore, for higher grid resolutions, there was a significant drop in CPU load when the depth image resolution was lowered.

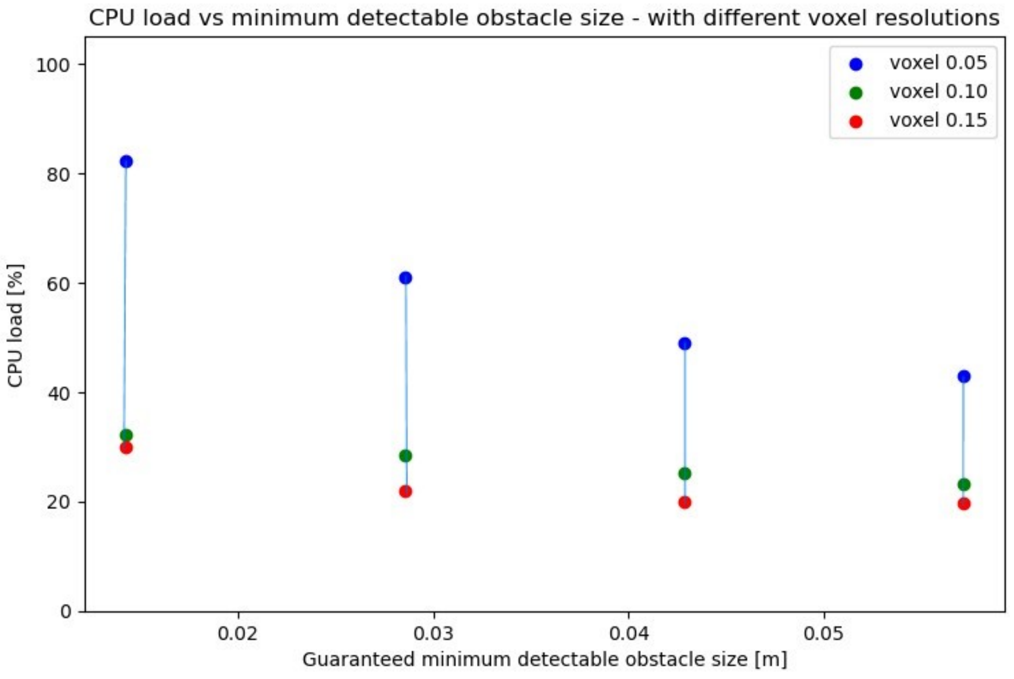


Figure 6.17: CPU load vs minimum guaranteed detectable object size

To interpret the plot in figure 6.17, a smaller minimum detectable object size represents better performance. In the plot, the measurements related to the same image resolution are connected to emphasize their relationship. As the varied voxel grid resolution does not influence the minimum detectable obstacle size, the measurements per image resolution do vertically align.

Reducing the image resolution was assumed to result in a lower CPU load. However, the drop in CPU load is significant only with a high-resolution voxel grid while it is minimal with a lower-resolution voxel grid.

An explanation for this significant impact with a high-resolution voxel grid is the number of voxels that need to be processed per new image. When the camera detects numerous obstacles, many pixels have to be processed. Regardless of the voxel grid resolution, the 3D projection takes the same amount of time for each loop. However, the raycasting and local map update step processes scale exponentially by the power of 3 based on the voxel grid resolution. This means that increasing the image resolution will significantly increase the CPU load at a faster rate with a higher voxel grid resolution.

6.4.4. Results obstruction width performance

The plot below, Figure 6.18, displays the relationship between the CPU load and the maximum solvable obstruction width. The local map size was varied from 1 to 35 meters in steps of 1 meter, while the voxel-grid resolution was varied for values of 0.10, 0.13, 0.15, and 0.20.
To understand the plot, note that a larger maximum solvable obstruction width indicates better performance. The results where local maps were similar in size are connected to highlight their relationship.

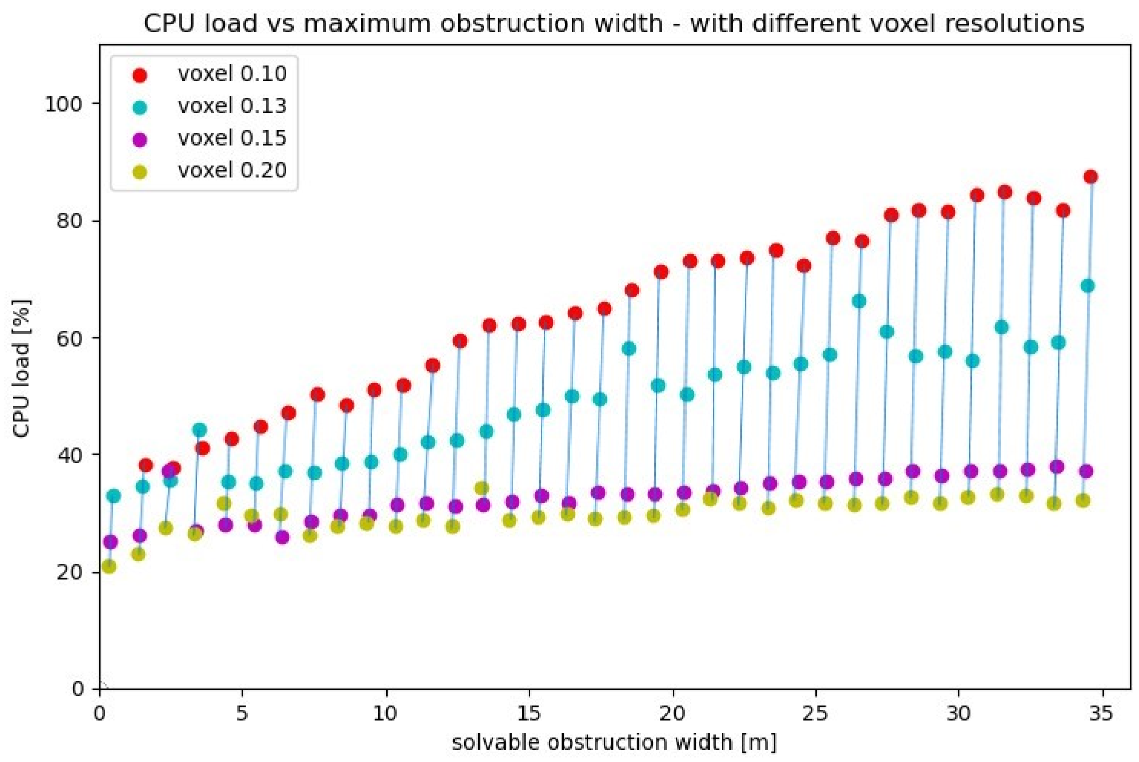


Figure 6.18: CPU load vs maximum solvable obstruction width

The plot shows a clear linear trend between increasing CPU load and increasing solvable obstruction width, with the exception of a few outliers. This trend supports the hypothesis. When the local map is made bigger, a higher voxel grid resolution leads to a stronger increase in CPU load.

However, when the voxel grid is made coarser, the CPU load does not significantly increase beyond a certain resolution, even for larger solvable obstruction widths. Similarly, as with the minimum gap size performance, the influence of other parameters might become more significant beyond a certain point. Since those other parameters were not varied in this test, the variance of CPU load flattens out for this parameter at coarser voxel grid resolutions.

6.4.5. Results reactiveness perfromance

The last result is the reactiveness performance. In other words, the time it takes for the planner to detect a new occurring obstacle and detect a collision. To measure reactiveness, the depth-image rate was varied between 15Hz, 7.5Hz, 5Hz, 3.8Hz, and 3Hz. Additionally, voxel grid resolution was varied between 0.05, 0.10, and 0.15 meters to test different system loads.
Figure 6.19 shows the CPU load on the vertical axis and reactiveness on the horizontal axis. A lower reactiveness time indicates better performance. The measurements with the same image rate are connected in the plot to show their relationships. Also, the image rate is displayed for every connected group.

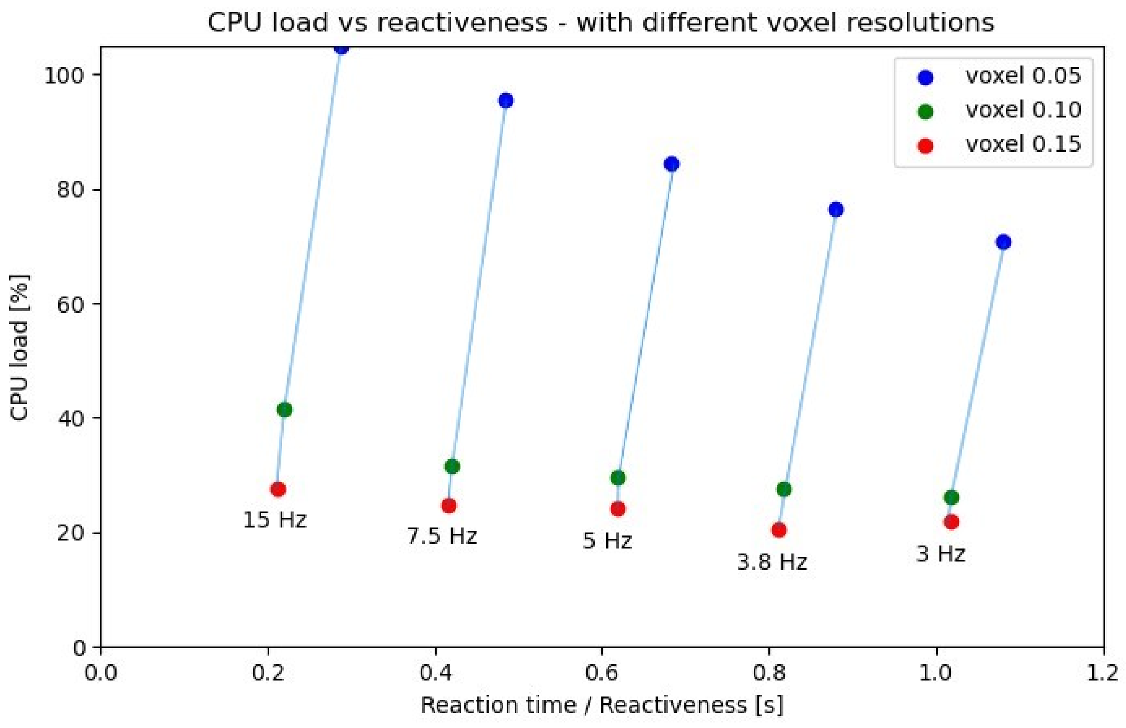


Figure 6.19: CPU load vs maximum time delay for reactiveness collision detection

According to the plot, reducing the image rate only leads to a slight decrease in CPU load, which results in a longer reaction time. Only when the system is under high load conditions, as with a voxel resolution of 0.05, does lowering the image rate seem to have a significant impact on the CPU load.
While the trend shown in the plot aligns with the hypothesis, its significance falls short of expectations. One plausible explanation for the limited variation in CPU load is that the image rate solely impacts the perception, not the path planner. A higher image rate increases the perception's constant CPU load. Nonetheless, the path planner, responsible for peak loads, operates with the same voxel grid and map, thereby maintaining a consistent additional peak load.

6.4.6. Tradeoff safety-vs-performance

The previous sections have indicated the CPU load for various configurations of 4 metrics. Now is explored how safety and performance relate to each other when the CPU is utilized either half or fully. For performance, from minimum gap size and solvable obstacle width, the minimum gap size is chosen as the most important metric to fly in a rainforest. And, for safety, from reactiveness and minimum detectable obstacle size, reactiveness is chosen as the most important to fly in a rainforest.

Seventy-two simulations were conducted with twelve variants of voxel resolution (ranging from 0.03 to 0.20) and six variants of depth-camera rate (ranging from 3hz to 15hz). The peak loads were plotted in figure 6.20 by the corresponding reactiveness and gap size. In the figure, for every result, the peak load is annotated.

For practical reasons, a CPU load of 90% was assumed to represent full utilization. Simulations with higher loads showed unclear behaviour, possibly due to CPU resource saturation. The results are grouped by colours. The yellow group shows the simulations where the peak load was between 50% and 90% CPU utilization. The purple group shows the simulations where the peak load was below 50% CPU utilization. For both groups, a Pareto-front is drawn through the results that show the best performance-safety relation.

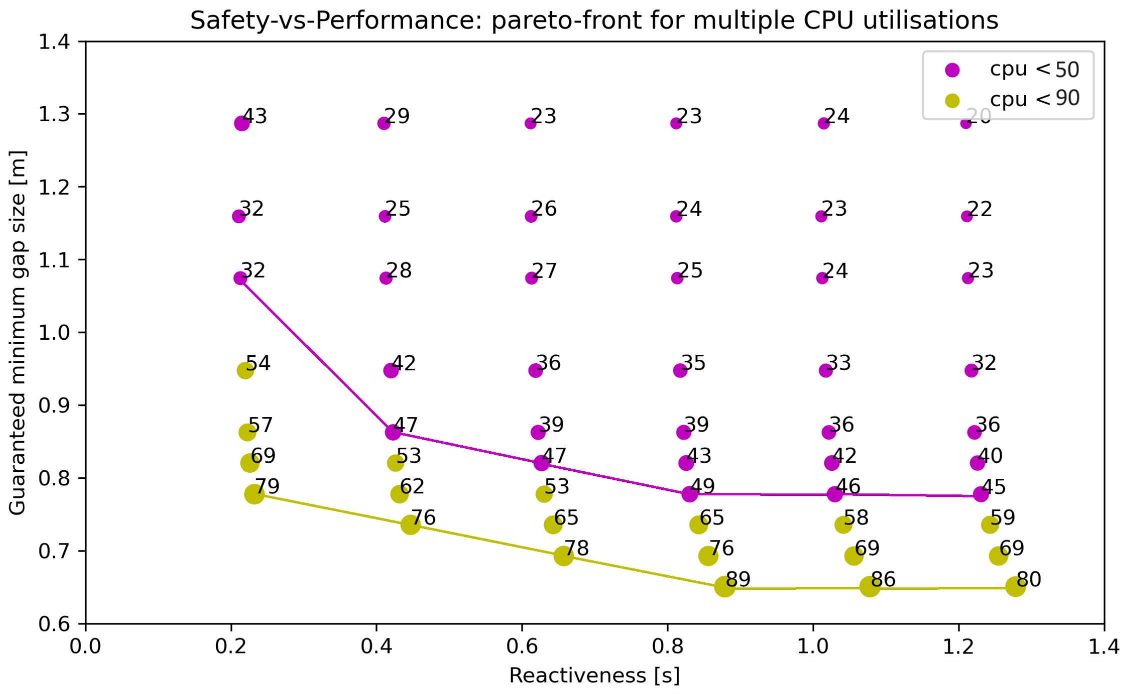


Figure 6.20: Tradeoff Safety-vs-Performance, with the Pareto-fronts for utilisation of different percentages of the processor

The Pareto-front shows the maximum performance for a limited CPU utilization. Within that maximum performance, there is a tradeoff between either performance or safety at the cost of the other. What was already known from the previous sections is that the gap size, mainly influenced by the voxel grid resolution, had the most significant variation. Again, this plot shows that the gap size has a significantly higher impact on the CPU load compared to the reactiveness.

The two main conclusions from these results are the following:

- **Downscaling does negatively impact the safety and performance**
  The Pareto-front for full utilisation lays closer to a small gap and lower reaction time than compared to half utilisation.
- **There is a clear tradeoff between safety and performance within half or full CPU utilisation**
  For full utilisation, there is a tradeoff in the range of combinations of [230ms reaction time & 0.78m gap size] to [880ms reaction time & 0.65m gap size].
  For half utilisation, the tradeoff shows a range of combinations of [210ms reaction time & 1.08m gap size] to [830ms reaction time & 0.78m gap size].

- **No significant gap size improvement for a depth image rate lower than 3.75hz**
  Above 900ms reaction time, corresponding to 3.75 Hz depth-image rate, the Pareto-front shows no significant difference in gap size.

For example, at half utilisation, the drone could react four times quicker if it is acceptable to increase the minimum solvable gap size from 0.78m to 1.08m.
Or, visa-versa; for example, for full utilisation, the drone can find a gap of 0.65m instead of 0.78m if it is acceptable the drone's reaction time is 880ms instead of 230ms. These examples show the extremes. For product deployment, a healthy balance should be found within this range.

6.4.7. Analysis 'dead-reckoning' position estimate

In this project, dead-reckoning position estimation replaces the visual-inertial position estimate used in the ego-planner paper. This significantly reduces CPU load (more than every other adaptation in this research) and thereby enables running ego-planner on the smaller processors. But this adaptation negatively impacts the quality of perception and planning, as described in 2.4. Therefore, now the results of the simulations are known, the ground-truth position data is extracted from the dataset to compare with the dead-reckoning position estimate.

Figure 6.21 below shows a top-view of the GPS, Visual-inertial, and IMU-based localisation estimates plotted over the flown path in the forest dataset. In the plot, the red dots represent the estimated location by the IMU, which is used for dead-reckoning navigation. The green line indicates the location estimated by the visual-inertial method, generated in post-processing as discussed in section 6.2.2. The purple circles represent GPS fixes, estimated by the added GPS module, also discussed in section 6.2.2. The grey contours in the figure signify the obstacles detected by the camera for the height ranging from 0.3m to 2.5m. These contours can help in better understanding the flown path and also illustrate the obstacle density within the mapped range, which might explain the GPS inaccuracy halfway through the visualised track. Lastly, the grid shown in the background is 1x1m, providing a reference of scale in this overview.

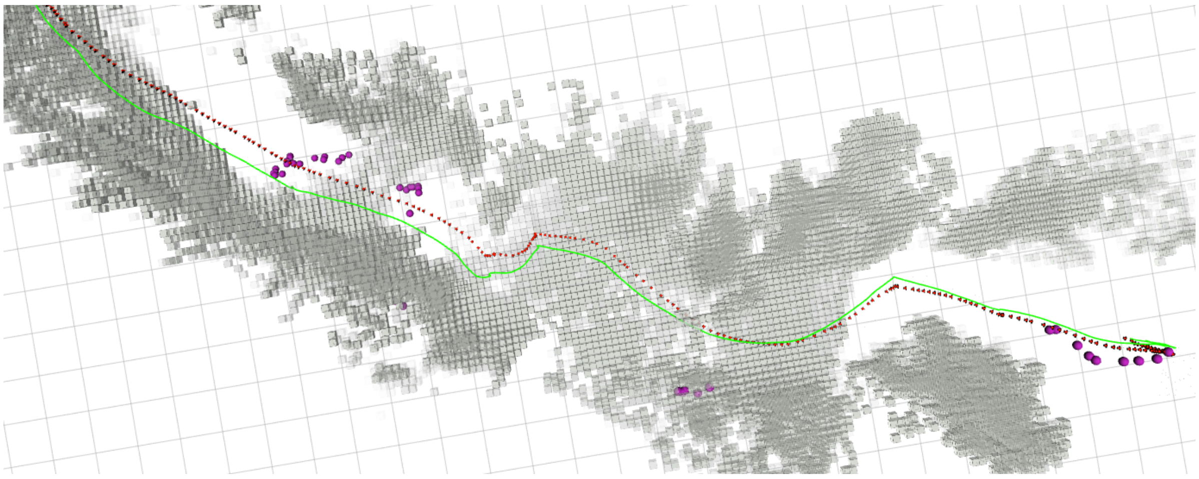


Figure 6.21: Obstacles in forest dataset (grey), with overlays of localisation by IMU (red dots), VINS (green line), and GPS (purple dots)

During the 90-second flight recorded in the dataset, the IMU and visual-inertial path show similar patterns and do not diverge significantly. Adding GPS to the data did not provide any useful insights into which path drifted the least. This highlights the limitations of GPS for localization in forest environments, even in low and open forests.
Based on these observations, it can be concluded that the IMU localization is suitable for a local planner as it did not show any significant drift during the test. However, it is still unclear whether visual-inertial localization is better than dead-reckoning for precise localization.

6.5. Downscaling applied

In appendix F, an example shows how this downscaling can be applied to a small processor on a small drone. In this example, a 100-gram drone can fly autonomously, similar to the Bebop used in this project, but with reduced performance. The autonomy of this system uses a 10-gram Raspberry Pi Zero 2w processor board, which costs 15 euros. In addition, another type of camera is used, which only weighs 8 grams but has half the range of the camera used on the Bebop drone. Figure 6.22 shows the integrated design of the system in Solidworks, this figure is shown enlarged in the appendix.

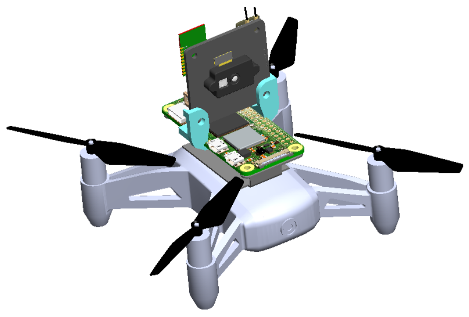


Figure 6.22: Solidworks 3D model of Tello drone with autonomy system

6.6. Conclusions on downscaling

In this chapter, the research is focused on the tradeoff between autonomous flight performance in the rainforest and processor load for small processors. The challenges of navigation in the forest are highlighted, leading to the design of four performance metrics to measure the autonomous quadrotor navigation system's performance in the forest. These performance metrics include the minimum detection size of obstacles, the reactiveness of the system to newly occurring obstacles, the maximum solvable width of obstructions, and the minimum solvable gap size.

The study involves testing various planner configurations to understand how four metrics relate to CPU load. Preparing data and simulating variations on a drone's processor in a real-world forest scenario can be done without the need for actual flights. The logged data from simulations is analyzed, and plotted against CPU load to reveal trends in performance metrics. These trends are analyzed in relation to the hypothesis and discussed in terms of their plausibility.

The tests analyzed in this chapter show that for narrower gap configurations, the CPU load and memory increase exponentially. However, for smaller detectable obstacles, the CPU load increase is negligible, except when the system is already under high load. The CPU load to solve wider obstructions increases gradually and linearly. Lastly, reactiveness only shows a significant increase in CPU load for the highest range of tested voxelgrid resolutions.

By the known influences of the metrics on CPU load, a final simulation test is run to show the maximum safety and performance on a half-and full utilisation of the processor load. This results of this test show firstly the negative impact on safety and performance by downscaling the processor capacity, and secondly, the tradeoff between better safety or better performance. This tradeoff shows how the reaction time of the system can be four times quicker when increasing the minimum navigable gap size by 0.15m to 0.30m.

In addition, the accuracy of the position estimate in dead-reckoning has been analyzed for the forest dataset created, and it has been compared to GPS and visual-inertial position estimates. The analysis shows that there is no significant divergence between the dead-reckoning (IMU) estimate and the visual-inertial estimate. GPS localization, however, was found to be significantly less accurate. Although a sporadic GPS fix may be useful in the rainforest to find back the home location, this analysis proves that it is not (solely) suitable as a localisation estimate for navigation.
To showcase the downscaling, this method is applied to fly a 100-gram drone successfully autonomously with all path planning computed onboard. Appendix E.2 lists some thoughts for future work on how downscaling can be improved further.

7 Conclusion

While examining the impact of downscaling the autonomy system on performance, a critical factor emerges: the resolution of the obstacle map. This significantly impacts the drone's ability to navigate safely through densely grown forest areas, and is critical for the choice of hardware. The choice of hardware consequently impacts the system weight and, thus, the maximum flight time. Tests reveal a noteworthy relationship between processor utilization and the drone's minimum navigable gap size. Fully utilizing the processor used in this project (Odroid XU4) enables successful navigation through gaps as small as 70cm, whereas with only utilising a quarter of the processor, the smallest navigable gap is limited to 120cm or larger. This underscores the tradeoff between downscaling the processor, the obstacle map resolution, and influencing the drone's performance in navigating challenging rainforest terrains.

This research worked towards a lightweight autonomy system for prolonged and safe flight under the rainforest's canopy layer. The results contribute to autonomous drone navigation, particularly in the challenging context of rainforest environments, advancing both technological capabilities for autonomous flight and environmental conservation efforts for rainforests. Literature on autonomous drone flight in dense and cluttered terrains, such as rainforests, is limited. Real-world tests conducted in this project revealed challenges not covered in previous research.

The implemented navigation method for dead-reckoning flight showcases an efficient operational framework. This strengthens the reliability of future fully autonomous drone missions in rainforest-like environments. Applying the insights of the downscaling research to this system integration not only minimizes the system size but also has the potential to yield more cost-effective drones and extended flight times. This method increases exploration effectiveness and safety and makes a valuable contribution to the broader field of autonomous drone technology.

While this research demonstrates significant improvements in reducing system size, it is important to acknowledge certain limitations. Limitations include a notably lower flight speed due to the limited update speed of the algorithm as well as potential inaccuracies in geo-tagging collected data and home-return positioning under the canopy. These challenges, inherent to downscaled autonomy, point to areas for refinement in future research.

In conclusion, this research advanced the development of a lightweight autonomy system for safe and prolonged rainforest drone flights. The downscaling process, while introducing trade-offs in flight speed and finding smaller gaps, delivers insights for enhancing the drone's cost-effectiveness and flight duration. Recognizing the limitations, particularly in dead-reckoning and sporadic GPS fixes, underscores the challenges inherent in downscaled autonomy. Future research should address the identified limitations, focusing on refining the system's practical applications in rainforest environments.

References

[1] Andreas Zwanenburg. *Literature Review - Autonomous MAV navigation through a dense tropical rainforest*. Tech. rep. Delft: TU/Delft, July 2023.

[2] XPRIZE. *XPRIZE Rainforest | XPRIZE Foundation*. 2023. URL: https://www.xprize.org/prizes/rainforest.

[3] XPRIZE. *XPRIZE Rainforest Competition Guidelines*. Tech. rep. 2023.

[4] Antonio Loquercio et al. “Learning high-speed flight in the wild”. In: *Science Robotics* 6.59 (Oct. 2021), p. 5810. URL: https://www-science-org.tudelft.idm.oclc.org/doi/10.1126/scirobotics.abg5810.

[5] M Kisantal. *Deep Reinforcement Learning for Goal-directed Visual Navigation*. Tech. rep. 2018. URL: http://resolver.tudelft.nl/uuid:07bc64ba-42e3-4aa7-ba9b-ac0ac4e0e7a1.

[6] Javier Antich Tobaruela et al. “Reactive navigation in extremely dense and highly intricate environments”. In: (2017). DOI: 10.1371/journal.pone.0189008. URL: https://doi.org/10.1371/journal.pone.0189008.

[7] Eric Hyyppä et al. “Under-canopy UAV laser scanning for accurate forest field measurements”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 164 (June 2020), pp. 41–60. DOI: 10.1016/J.ISPRSJPRS.2020.03.021.

[8] A. Pirti. “Using GPS near the forest and quality control”. In: *Survey Review* 38.298 (Feb. 2012), pp. 286–298. DOI: 10.1179/003962605790586572.

[9] Tiberiu Paul Banu et al. “The Use of Drones in Forestry”. In: *Journal of Environmental Science and Engineering B* 5.11 (Nov. 2016). DOI: 10.17265/2162-5263/2016.11.007.

[10] Takafumi Taketomi et al. “Visual SLAM algorithms: a survey from 2010 to 2016”. In: *IPSJ Transactions on Computer Vision and Applications* 9 (2017), p. 16. DOI: 10.1186/s41074-017-0027-2.

[11] *3D Tree Segmentation using Deep Learning and Lidar*. URL: https://interpine.nz/improvements-in-3d-tree-segmentation-using-deep-learning/.

[12] Manuel Rucci. *A General Purpose Control Design For Vision Based Autonomous Quadrotor Navigation*. Tech. rep. 2017.

[13] Julio A Reyes-Munoz et al. “A MAV Platform for Indoors and Outdoors Autonomous Navigation in GPS-denied Environments”. In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)* (2021). DOI: 10.1109/CASE49439.2021.9551409.

[14] Andréa Macario Barros et al. *A Comprehensive Survey of Visual SLAM Algorithms*. Feb. 2022. DOI: 10.3390/robotics11010024.

[15] Chi Zhang et al. “A Lightweight and Drift-Free Fusion Strategy for Drone Autonomous and Safe Navigation”. In: *Drones* 7.1 (Jan. 2023), p. 34. DOI: 10.3390/drones7010034.

[16] Artur Shurin et al. “QDR: A quadrotor dead reckoning framework”. In: *IEEE Access* 8 (2020), pp. 204433–204440. DOI: 10.1109/ACCESS.2020.3037468.

[17] Jonghoek Kim. “Fast Path Planning of Autonomous Vehicles in 3D Environments”. In: *Applied Sciences 2022, Vol. 12, Page 4014* 12.8 (Apr. 2022), p. 4014. DOI: 10.3390/APP12084014. URL: https://www.mdpi.com/2076-3417/12/8/4014/htm%20https://www.mdpi.com/2076-3417/12/8/4014.

[18] Xin Zhou et al. “EGO-Planner: An ESDF-free Gradient-based Local Planner for Quadrotors”. In: (Aug. 2020). URL: http://arxiv.org/abs/2008.08835.

[19] Xin Zhou et al. *Swarm of micro flying robots in the wild*. Tech. rep. 2022, p. 5954. URL: https://www.science.org.

[20] Eungchang Mason Lee et al. “REAL: Rapid Exploration with Active Loop-Closing toward Large-Scale 3D Mapping using UAVs”. In: (Aug. 2021). URL: http://arxiv.org/abs/2108.02590.

[21] *GitHub - HKUST-Aerial-Robotics/VINS-Fusion: An optimization-based multi-sensor state estimator.* URL: https://github.com/HKUST-Aerial-Robotics/VINS-Fusion.

[22] *Odroid XU4.* URL: https://botland.store/odroid-computers-modules/6314-odroid-xu4-samsung-exynos5422-octa-core-20ghz14ghz-2gb-ram-5904422365707.html.

[23] *Depth Camera D435i – Intel® RealSense™ Depth and Tracking Cameras.* URL: https://www.intelrealsense.com/depth-camera-d435i/.

[24] *TOF Camera - Arducam Wiki.* URL: https://docs.arducam.com/Raspberry-Pi-Camera/Tof-camera/TOF-Camera/.

[25] Standford. *Introduction to A*.* 2020. URL: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[26] *Airspace Visualizer | MyDroneFleets.* URL: https://mydronefleets.com/airspace-visualizer/.

[27] Ma Weinmann et al. “Fast and automatic image-based registration of TLS data”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 66.6 SUPPL. (Dec. 2011). DOI: 10.1016/j.isprsjprs.2011.09.010.

[28] *GitHub - alspitz/cpu_monitor: ROS node that publishes all nodes’ CPU and memory usage.* URL: https://github.com/alspitz/cpu_monitor.

[29] Jesus Tordesillas et al. “MIT: Fast and Safe Trajectory Planner for Flights in Unknown Environments”. In: (). URL: https://github.com/jtorde.

[30] *Raspberry Pi Zero 2 W – Raspberry Pi.* URL: https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/.

[31] *Time of Flight (ToF) Camera - Arducam.* URL: https://www.arducam.com/time-of-flight-camera-raspberry-pi/.

[32] *DJI Tello.* URL: https://store.dji.com/nl/product/tello?vid=38421.

[33] *Step-up Boost Converter - 2A - 5V.* URL: https://www.tinytronics.nl/shop/nl/power/spanningsconverters/boost-(step-up)-converters/dc-dc-step-up-boost-converter-2a-5v-output.

A

Literature research - relevant path planning methods

This appendix provides more background details on the topics that are highly relevant for deploying autonomous navigation in the rainforest. It is a recapitulation of some of the topics that were discussed in chapter 2, but with more in-depth information. The background information presented in this appendix has been sourced from the Literature Research [1], which was conducted prior to this research. Some of the information has been directly copied from the literature research into this appendix to make it easier for readers to access the most significant background information in detail.

In the literature research, four papers were found with relevant methods that relate to the research question what method or algorithm would be best for autonomous flight in the wild. In this selection, all requirements, such as onboard computation, are considered. For example, methods that are not feasible to compute onboard on a small drone are not filtered out.
The four found methods all relate to a research paper. In this appendix section, the four methods are explained below in more detail than in the report itself. Also, illustrations of the methods are shown to give a better insight into the method.

A.1. FASTER-Planner

FASTER is a trajectory planner, published in 2019 by MIT [29]. The trajectory planner is focused on safe navigation through unknown environments. The planner combines rapidly exploring random tree (RRT) with a safety module that searches for reachability to ensure that the planned trajectory is collision-free.

The RRT explores the environment by sampling random points and connects these to the tree structure. The safety module continuously looks for a backup path with the highest probability of being collision-free. This way, the drone always has a backup trajectory to follow, considering acceleration constraints to break in time before hitting an obstacle. If the drone cannot calculate a new trajectory for some reason, for example, when the system is overloaded, the safe trajectory always ensures safe deceleration for the drone from forward flight into hovering.

Compared to previous state-of-the-art algorithms, the efficiency improvement of this algorithm is the dynamic adjustment of the RRT sampling region based on the drone's current position and orientation. This makes the algorithm focus on the areas most likely to contain a path to the goal while not wasting computation time in unlikely areas.

Figure A.1 below illustrates the functionality of the FASTER-planner. The planning horizon is shown as a black circle. First, with RRT, the planner finds a feasible path (yellow line). The planner then calculates polygons around the feasible path in the mapped environment that are completely in the explored free space (green area) and polygons in the unexplored possibly free space (red area). The optimised trajectory is generated within the red, possibly obstacle-free polygon (red line). Also, a safe trajectory is calculated (blue line) in the completely explored free space as a backup in case an obstacle might occur in the not yet explored space, such as the orange obstacle with the dotted line in this example. For visualisation purposes, the illustration shows the functionality of the planner in 2D, but all calculations and optimisations in the FASTER planner are in 3D.

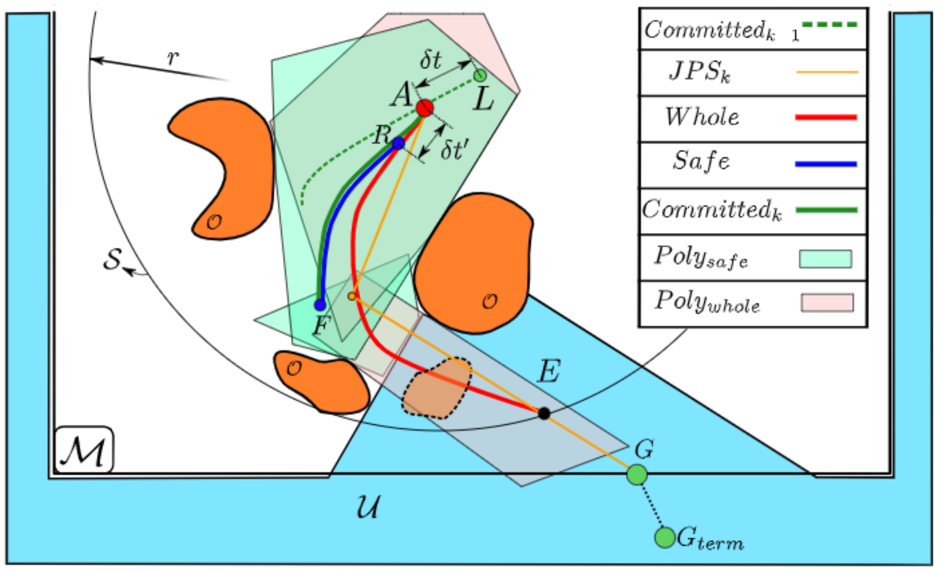


Figure A.1: FASTER-planner method illustrated [29]. Unexplored space is light blue, obstacles are orange. One obstacle is unknown indicated by a dotted line.

The FASTER planner is tested in simulation and real-world environments. Code is open-source and available on GitHub, and Youtube videos are included with proof that the drone can safely manoeuvre through complex environments. A note to make is that the conducted tests shown in the paper and videos are only in a dense but structured environment. For example, a simulation in a dense forest, only with perfectly cylinder-shaped objects. And a real-world test with perfectly square boxes, densely placed with narrow corridors to pass through.

A.2. EGO-Planner

EGO-planner is a trajectory planner, published in 2020 by ZJU-Fastlab [18]. This gradient-based planner uses the expected improvement criterion to generate safe and smooth drone trajectories.
The expected improvement (EI) criterion is Bayesian optimization, used in multiple state-of-the-art planners, that balances the tradeoff between exploration and exploitation. In exploration, the algorithm searches for promising regions, while in exploitation, it samples points around the current best solution. By balancing this tradeoff, the EI criterion efficiently identifies the optimal solution.

The planner first creates a probabilistic model of the environment and predicts the cost of potential trajectories. It then selects the trajectory with the highest expected improvement in cost and optimises this trajectory. This process is repeated to generate a sequence of trajectories to guide the drone towards its goal while minimizing cost.

Figure A.2 below illustrates the functionality of the EGO-planner. The gradients are only calculated for the region of the colliding obstacles. Then the smoothed b-spline trajectory deviates from the original straight path with the help of the gradient map until the trajectory is collision-free.

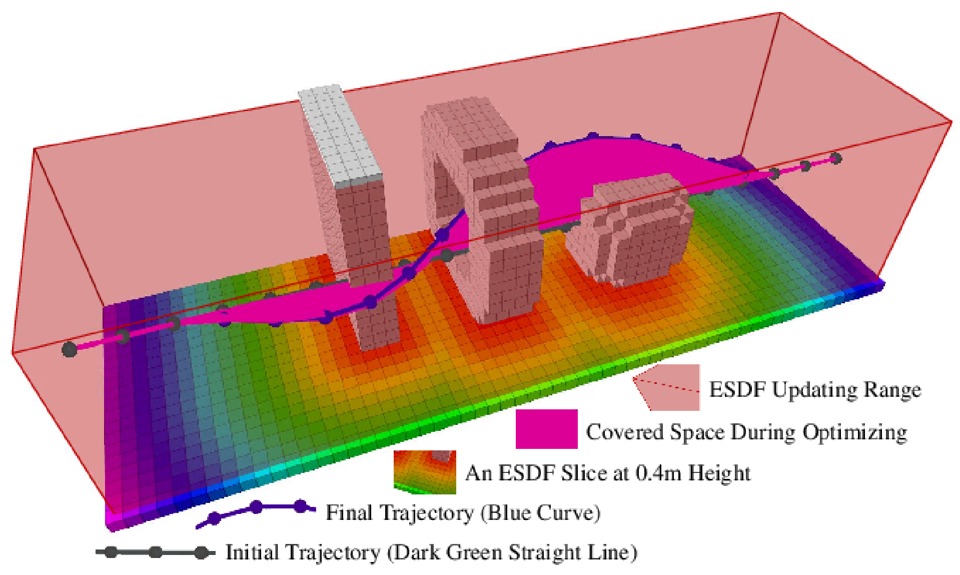

ESDF Updating Range
Covered Space During Optimizing
An ESDF Slice at 0.4m Height
Final Trajectory (Blue Curve)
Initial Trajectory (Dark Green Straight Line)

Figure A.2: EGO-planner functionality illustrated [18]. The trajectory is planned by utilizing the gradient fields of obstacles to navigate around them.

One of the key advantages of EGO-Planner is that it can handle complex and dynamic environments with uncertain and time-varying constraints. The algorithm can generate trajectories optimized for different objectives, such as speed, energy efficiency, or safety.
The EGO planner has been tested in simulation and real-world scenarios. Also, similar to the FASTER planner, source code is available on Github and videos are published on Youtube.

A.3. Swarm in the wild

Based on the EGO-planner, ZJU-Fastlab proposed in 2022 a method to fly a swarm of drones in a bamboo forest [19]. This swarm aims to efficiently explore, map, and plan through new environments by exploiting the most out of the total group effort and data. To elaborate on this, there are a couple of key features in this method, these are formation flight, joint mapping, and multi-agent trajectory planning.

The swarm of drones has a predefined formation that the group aims to preserve. The spread and shape of the formation helps to map the environment efficiently and get better localization. Every drone in the swarm has a radio beacon and a receiver. With this technology, the group knows their relative distances from each other. This is used to preserve the shape and distances of the formation and improve the location estimate.

In this swarm method, visual-inertial SLAM is used for localisation. This uses camera images onboard the drone to measure movements, thereby estimating the location. Every drone in the swarm runs this localisation, and every individual will have some estimate error. But, by using the relative distances in the swarm, the swarm can reduce each other errors and get a more robust position estimate.
The Swarm merges their mapping data and creates a more accurate map. Occlusion of obstacles is a general problem with mapping. But, merging the different viewpoints of the drone's depth sensors reduces occlusions. Therefore the obstacles will be mapped more accurately, which is beneficial for trajectory planning.

The trajectory planning is based on the EGO-planner, but now multiple agents (drones) must plan in the same space. By optimising the objective of the complete swarm, the drones communicate how to plan around obstacles or through a narrow gap. The objective of the swarm is to preserve the formation and minimise the group's energy effort to reach the goal. Figure A.3 shows a top-down view of the swarm flying in a bamboo forest with a specific formation shape. Also, the planned trajectories for every drone is plotted.

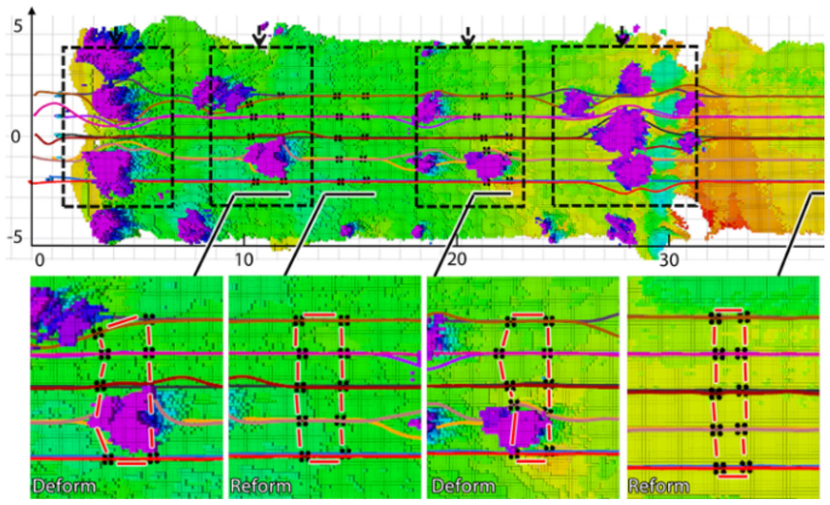


Figure A.3: Preserving and reforming drone swarm formation to avoid obstacles in a forest, as seen from a top-down view [19].

Although all benefits of using a swarm do not apply to the drone in the Xprize rainforest challenge, this paper has some valuable information. The deployment of the EGO-planner is better explained than in the original EGO-planner paper. And the effectiveness is shown in a forest environment. Also, the paper explains how their method can be used on a single drone.

The paper is, as one of the few, fully reproducible. The exact parts list and building scheme of the used drones are published. And the algorithm, including all options and variations, is published on their GitHub repository, including detailed documentation.

A.4. REAL-Planner

The Rapid Exploration with Active Loop-Closing (REAL) is a planner that focuses on large-scale 3D mapping and is published by the Urban drones Lab [20]. This planner does rapid exploration in a GPS-denied environment with a quadrotor drone. The planner is optimised to generate and exploit large scaled maps and uses loop-closure to refine the map and the drone its position estimate. The paper proposes a method to apply rapid exploration, loop-closing, and active optimization techniques to improve the efficiency and accuracy of 3D mapping using a drone. The paper describes the different components used in this system, including the drone, cameras, and software to process the data.

The Peacock Trajectory Planner used in this paper is based on the concept of a probabilistic roadmap. This is a graph structure that is constructed by sampling the configuration space of the drone. In this graph, nodes represent valid configurations of the drone and edges represent feasible paths between those configurations. The Peacock planner extends this concept by considering not only the configuration space of the drone, but also the space of trajectories that the drone can follow. Figure A.4 shows a screenshot of a real-world test where the drone samples multiple trajectories.

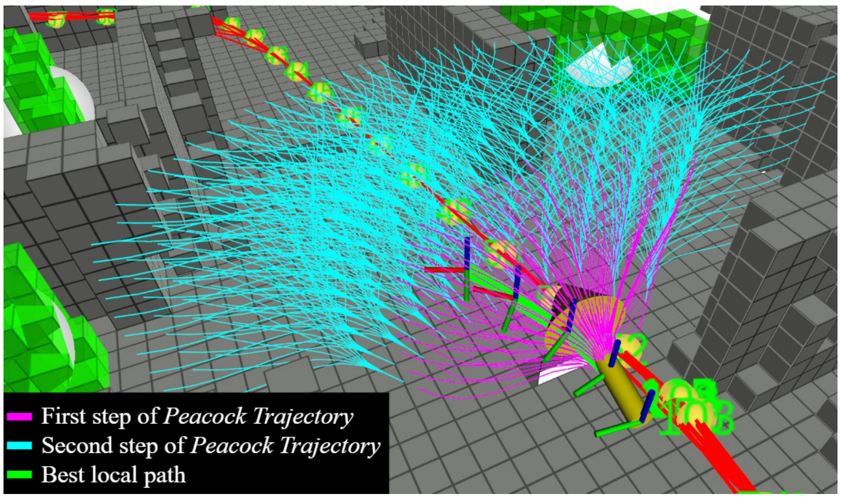

First step of *Peacock Trajectory*
Second step of *Peacock Trajectory*
Best local path

Figure A.4: REAL [20] sampling peacock trajectories (purple and blue) and selecting the one with the highest reward to the goal.

The Peacock Trajectory Planner uses a two-stage process to generate collision-free paths for the drone. In the first stage, it generates a roadmap of trajectories by sampling the space of feasible trajectories. The planner uses a probabilistic model to evaluate the likelihood that a given trajectory will be collision-free. In the second stage, the planner searches the roadmap for a collision-free path between the start and goal configurations of the drone. The planner uses an A* search algorithm [25] to find the shortest path between the start and goal configurations, taking into account the likelihood of collision along each trajectory.
Overall, the Peacock Trajectory Planner is a powerful and efficient planning algorithm that is, according to the authors, well-suited for quickly exploring complex environments and generating collision-free paths for drones.

The method is tested in simulation and real-world environments. The authors did not provide an open-source code repository. But as the paper describes how best to combine different open-source available ROS packages, the project still seems reproducible. The authors did also provide multiple youtube videos that demonstrated the functionality of the method and compare this to other related previous work. A note to make is that the proposed drone in this paper is a factor 2 larger and heavier than the drone that will be used for the rainforest challenge. The presented drone uses a Realsense D435i depth camera and a Realsense T265 pose-tracking camera for pose estimation, which is a significant benefit for localisation. But this camera is 60 grams which too heavy to add to the drone in the rainforest project.

B
Design Autonomy Backpack

This appendix shows more details on the autonomy-backpack design. The design of this module, that snap-fits on top of the Bebop 2 drone, is divided into three categories which are mechanical, electric, and software design. These designs are discussed in this appendix in that order.

The mechanical design section details where the chosen parts are located in the design and how these are mounted together with the help of 3D-printed frames. The electric design section gives enlarged diagrams of how components are wired to the processor board. Lastly, the software design shows the structure of the software in ROS in more detail.

B.1. Mechanical design

The mechanical design is modelled in Solidworks in 3D. By drawing all parts in 3D, standard comments and custom parts, the assembly is created. To optimize the design for a compact form factor and lightweight, this assembly is crucial. In the assembly, the frame is designed to mount all parts compactly and validate that all parts will fit. In the 3D assembly, it is also validated that electric cables and connectors have enough space. But, the cables and connectors are not included in Solidworks for simplicity.

In this section, the assembled 3D design is shown in Figure B.1. Thereafter, the same design is shown as an exploded view in Figure B.2 to show how all components fit together in the design, followed by a name list of all components annotated in the exploded view.

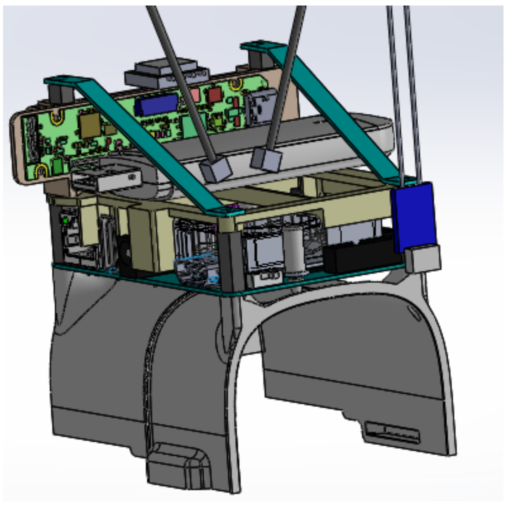


Figure B.1: 3D backpack design screenshot in Solidworks

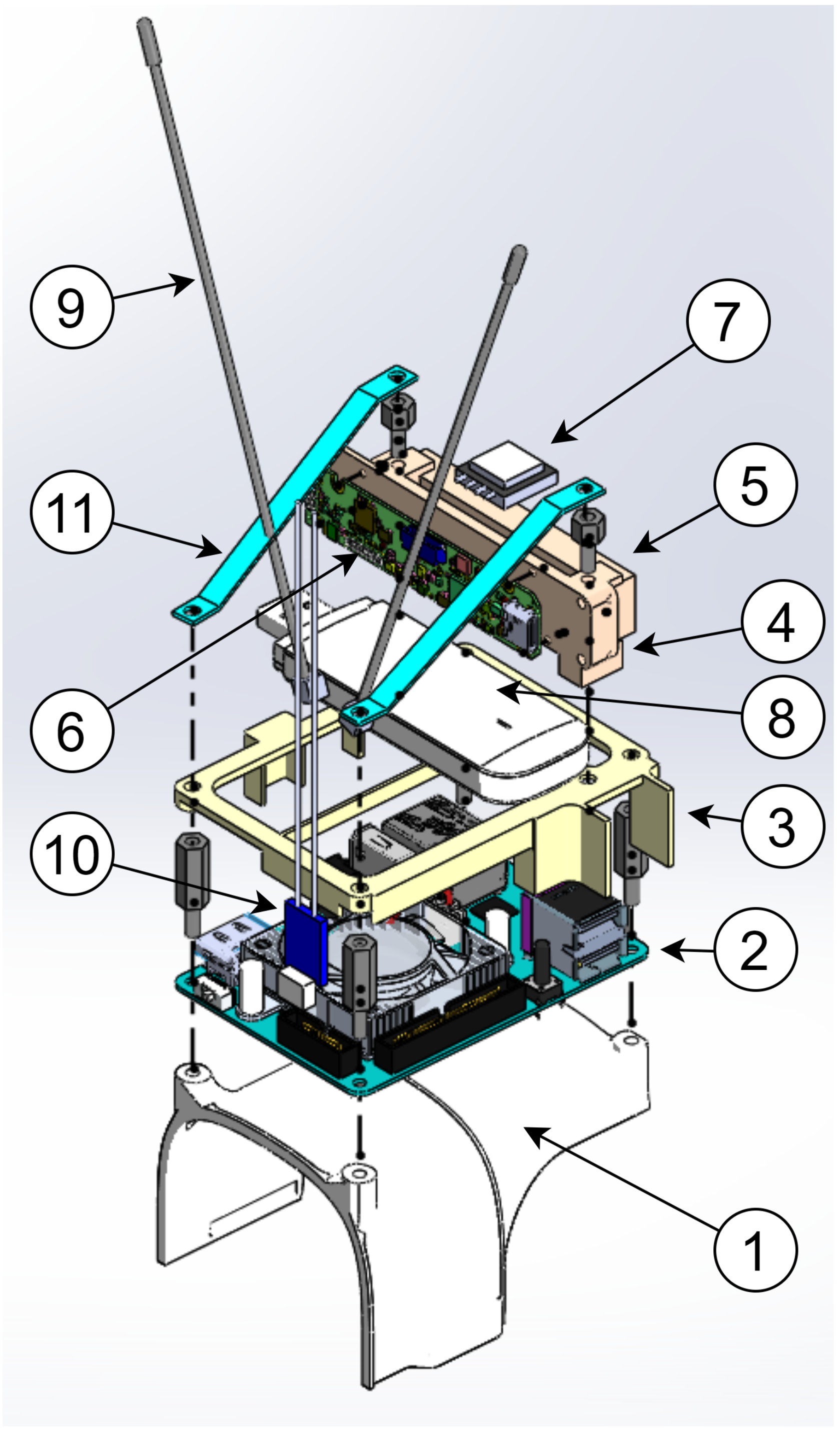


Figure B.2: Rendering of 3D backpack design in Solidworks

1. Mounting for backpack on drone - 3D printed
2. Processor board - Odroid XU4
3. Frame for electronics - 3D printed
4. Frame for Depth camera - 3D printed
5. Stereo Camera - Depth module Realsense D430
6. Depth processor - Realsense processor D4
7. GPS module - HGLRC M100 (Ublox M10)
8. USB 4G dongle - Huawei E3372
9. 4G antennas - 5dBi CRC9
10. RC-receiver - FrSky R-XSR
11. Electronics protection - 3D printed

B.2. Electric design

To give an overview of how all components are connected to the processor board, electric wiring diagrams are drawn in Fritzing software. This section gives enlarged diagrams of the illustrated wiring diagram in figure B.3 and the official wiring diagram in figure B.4 to which pins the components connect.

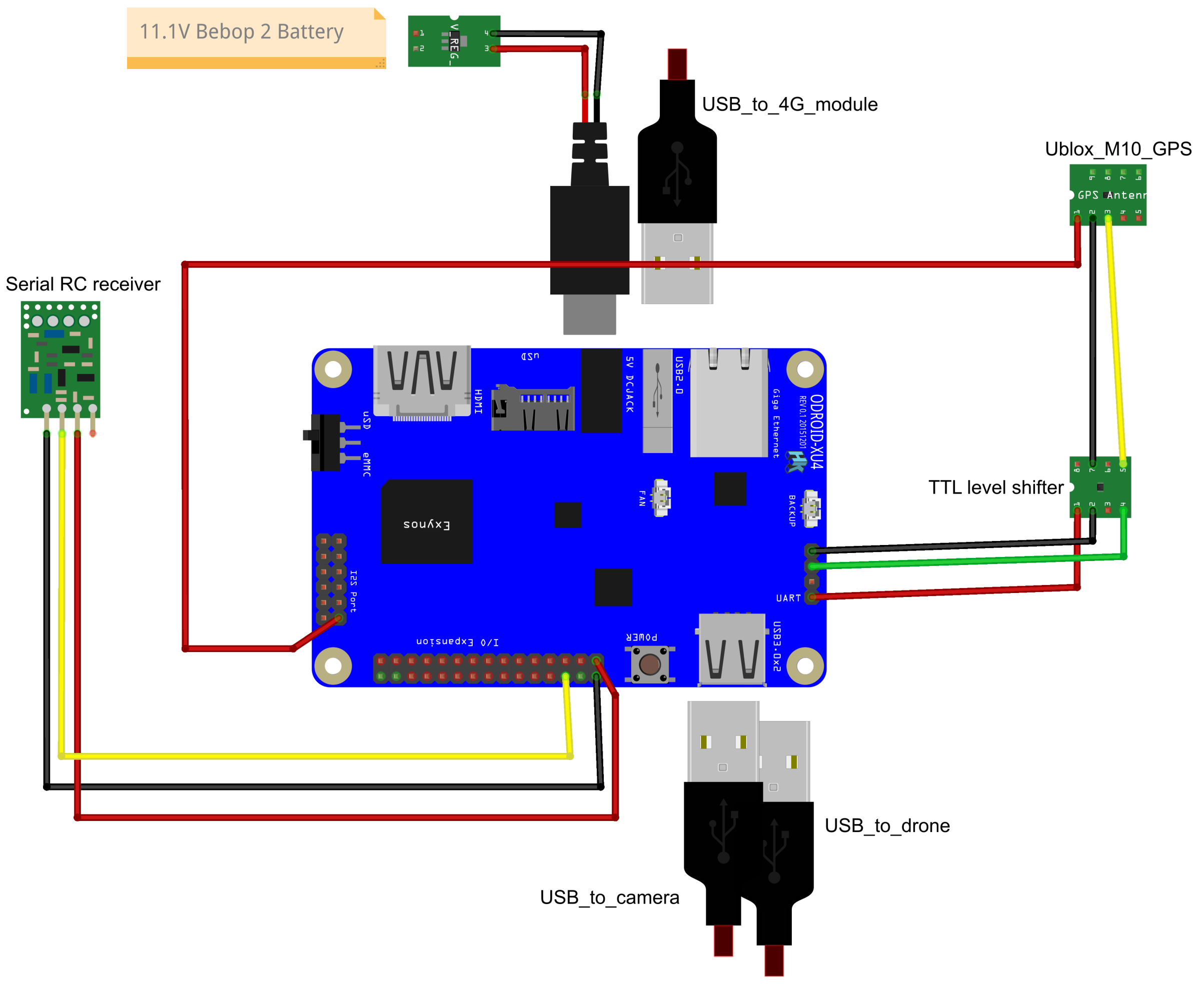


Figure B.3: Wiring scheme Odroid XU4

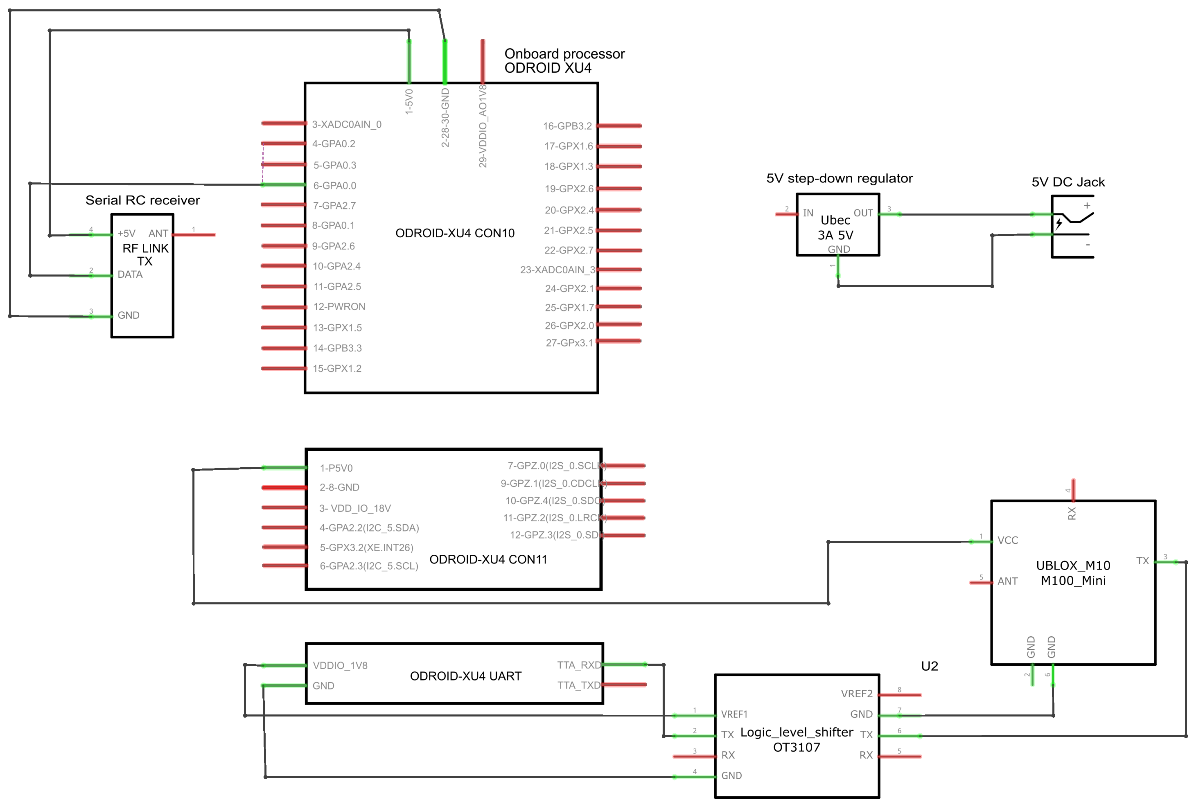


Figure B.4: Connection scheme Odroid XU4, exclusive USB modules

B.3. Software design

To highlight the software design in more detail, figure B.5 below shows an enlarged overview of the system architecture for communications between the modules of the system.

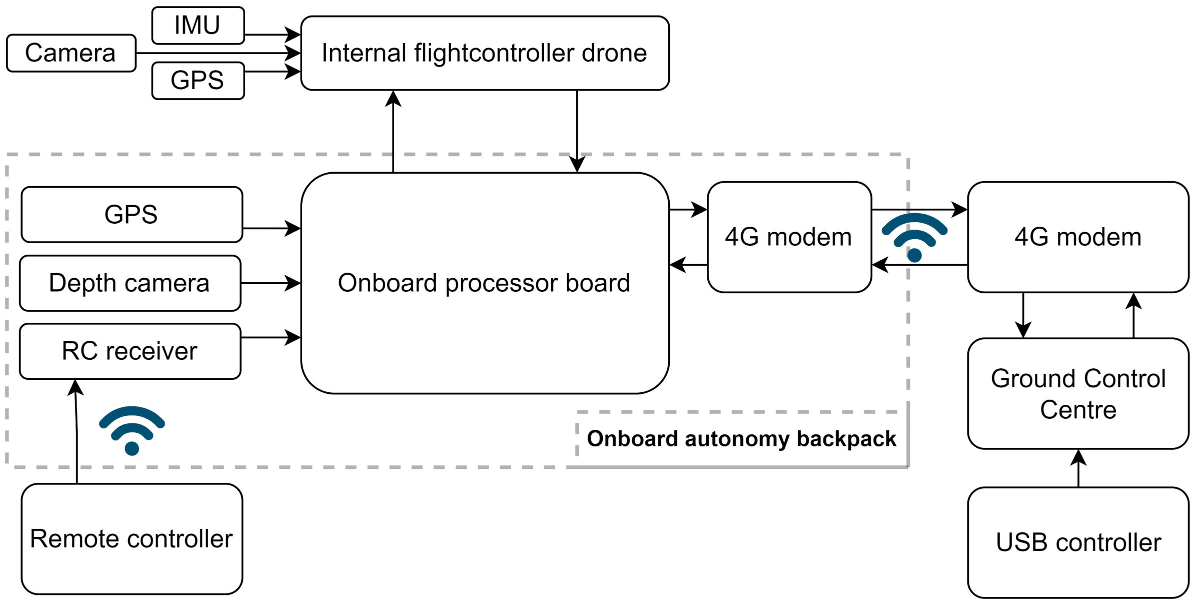


Figure B.5: Drone system architecture for autonomous navigation in the rainforest

Figure B.6 shows the ROS node and topic architecture. This automatically generated graph shows how data (square blocks) is shared between different processes (oval blocks).

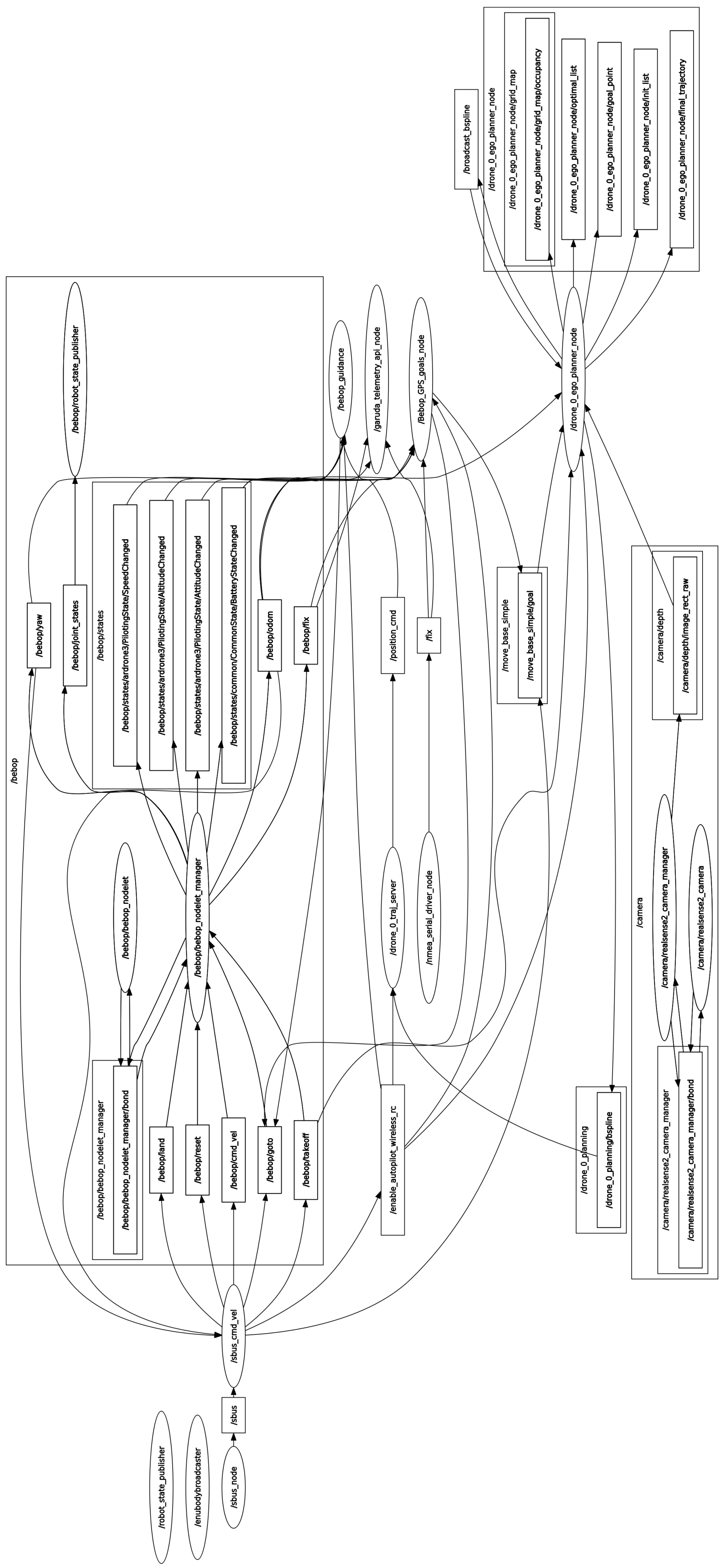
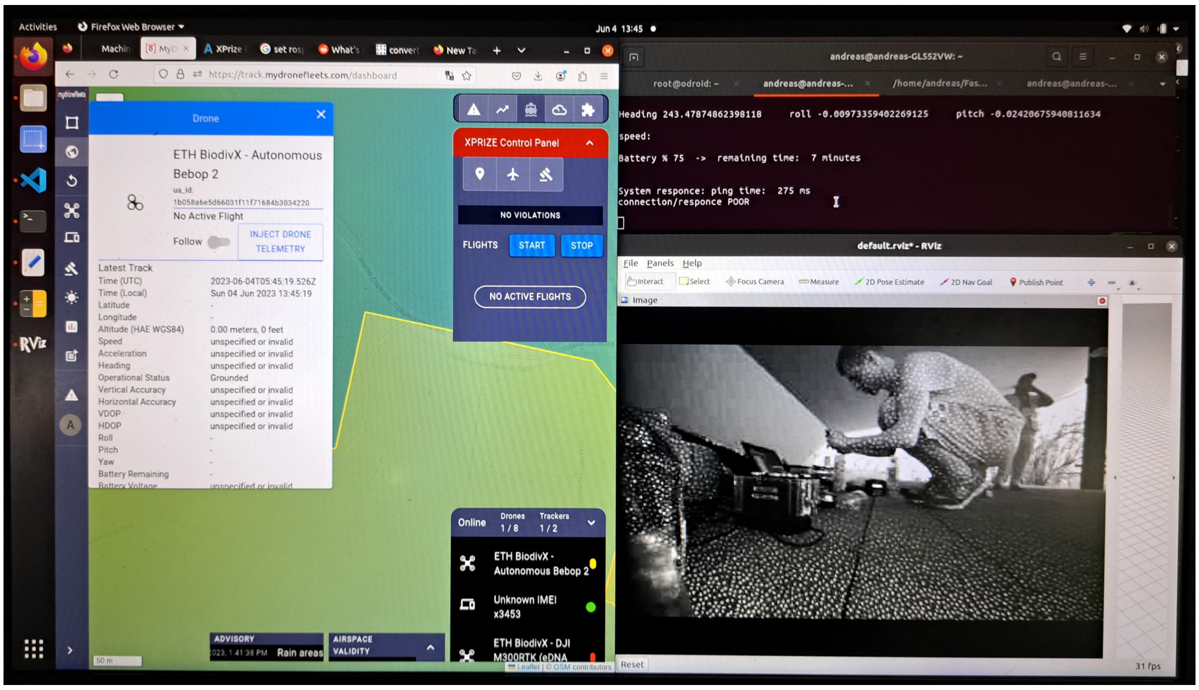

/bebop
/bebop/robot_state_publisher
/bebop/yaw
/bebop/joint_states
/bebop/states
/bebop/states/ardrone3/PilotingState/SpeedChanged
/bebop/states/ardrone3/PilotingState/AltitudeChanged
/bebop/states/ardrone3/PilotingState/AttitudeChanged
/bebop/states/common/CommonState/BatteryStateChanged
/bebop/odom
/bebop/fix
/bebop/bebop_nodelet
/bebop/bebop_nodelet_manager
/bebop/bebop_nodelet_manager
/bebop/bebop_nodelet_manager/bond
/bebop/land
/bebop/reset
/bebop/cmd_vel
/bebop/goto
/bebop/takeoff
/bebop_guidance
/garuda_telemetry_api_node
/Bebop_GPS_goals_node
/position_cmd
/fix
/drone_0_traj_server
/nmea_serial_driver_node
/enable_autopilot_wireless_rc
/move_base_simple
/move_base_simple/goal
/drone_0_ego_planner_node
/broadcast_bspline
/drone_0_ego_planner_node
/drone_0_ego_planner_node/grid_map
/drone_0_ego_planner_node/grid_map/occupancy
/drone_0_ego_planner_node/optimal_list
/drone_0_ego_planner_node/goal_point
/drone_0_ego_planner_node/init_list
/drone_0_ego_planner_node/final_trajectory
/camera
/camera/depth
/camera/depth/image_rect_raw
/camera/realsense2_camera_manager
/camera/realsense2_camera
/camera/realsense2_camera_manager
/camera/realsense2_camera_manager/bond
/drone_0_planning
/drone_0_planning/bspline
/sbus_cmd_vel
/sbus
/sbus_node
/robot_state_publisher
/enubodybroadcaster

Figure B.6: ROS RQT Graph, which shows the network structure for nodes and topics

B.4. Outdoor mission components

In the competition, the organisation assigned a company named Garuda Robotics to conduct airworthiness tests beforehand and supervise drone operations during the competition. Specifically, the self-made drones had to be examined extensively in this airworthiness test. The airworthiness check had a big checklist with requirements that were not shared with the competitors beforehand, although the team asked for these requirements. In Singapore, in the days of calibration and airworthiness checks, the checklist showed multiple requirements that were not incorporated in the drone. The system components described in this section were implemented on short notice in the 48-hour window of the airworthiness checks.

Some of the main points that were required but not yet implemented were a ground control station (laptop or RC controller with screen) on which the drone's camera live stream was shown, a live map with the drone's position, and basic drone telemetry data. To fix this before the competition, a ground station interface was improvised on the laptop with multiple windows showing the different information. An example is shown in figure B.7 below. In this image, taken at the competition's airworthiness check, on the left is the live map with the drone's position and the positions of other drones and aircraft. On the bottom right, the live stream of the drone, with a 1.0-second delay, was acceptable apparently. On the top-right, the telemetry of the drone with attitude, altitude, ground speed, heading, battery %, expected remaining flight time, and the signal speed/strength. This data for the ground control station was streamed via 4G.



Drone

ETH BiodivX - Autonomous Bebop 2
ua_id:
1b058a6e5d66031f11f71684b3034220
No Active Flight
Follow
INJECT DRONE TELEMETRY

Latest Track	
Time (UTC)	2023-06-04T05:45:19.526Z
Time (Local)	Sun 04 Jun 2023 13:45:19
Latitude	-
Longitude	-
Altitude (HAE WGS84)	0.00 meters, 0 feet
Speed	unspecified or invalid
Acceleration	unspecified or invalid
Heading	unspecified or invalid
Operational Status	Grounded
Vertical Accuracy	unspecified or invalid
Horizontal Accuracy	unspecified or invalid
VDOP	unspecified or invalid
HDOP	unspecified or invalid
Roll	-
Pitch	-
Yaw	-
Battery Remaining	-
Battery Voltage	unspecified or invalid

```
Heading 243.47874862398118    roll -0.00973359402269125    pitch -0.02420675940811634

speed:

Battery % 75  ->  remaining time:  7 minutes


System responce: ping time:  275 ms
connection/responce POOR
```

Figure B.7: Alternative created ground station for flight monitoring

In addition, before takeoff, the drone pilot needed to show a Garuda safety officer the applied geofencing overlaid on a satellite image map. As the ground station used the interactive map in the browser from a 3rd-party service, the geofencing lines could not be drawn on this map. Although not ideal, but acceptable, the applied geofencing of the drone was plotted over a satellite screenshot by a python script. The python script extracts the geofencing data from the drone to draw over the image. Figure B.8 below shows an image from this geofencing plot at the airworthiness check in Singapore. The blue square is where the drone will kill the motors, defined by north and east bounds. The green circle is the maximum distance from the takeoff position what the drone will not exceed in normal flight operations, either manually or on autopilot.

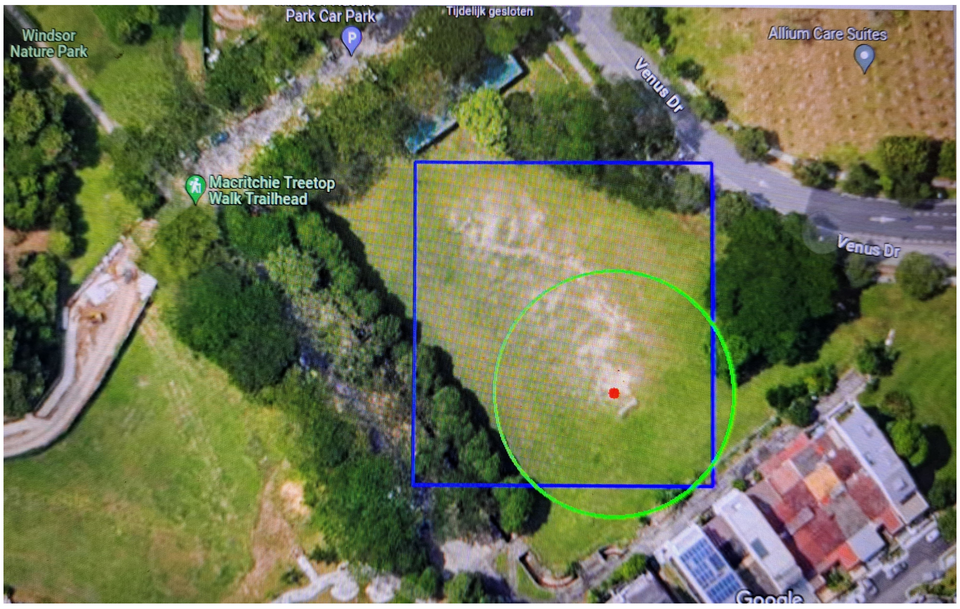


Figure B.8: Alternative created geofencing plotted on satellite image map

C
Dataset creation

To benchmark different configurations of the path planning algorithm on exactly the same sensor input, and thus the same scenario, datasets are created. These datasets are recorded in a real flight and collected the data spread over the ROS network. From this data, sensor data is filtered and injected into the system in a simulation to see how the path planner algorithm performs.

At first, a dataset indoors was created to see the performance of the algorithm in a specifically controlled scenario. Thereafter, an outdoor dataset was created to test the performance of the algorithm in real and varying foresty conditions. The creation of these datasets is discussed in this order in the following sections. Lastly, the content of the outdoor dataset is shown to give insight what data is recorded.

C.1. Record dataset indoors

To record the dataset indoors, the first step is to create a desired setting with obstacles. Then, multiple recording methods are initiated, in ROS, and externally with a top-view camera. Multiple runs are recorded to have multiple datasets to choose from. These steps are discussed in this section.

Two different tests need to be recorded to evaluate the planner's performance in an indoor environment. In the first test, the drone is trapped inside a non-convex cube-shaped obstacle. The planner has to escape the cube through a single narrow opening. The second test is the reactiveness test, which checks how long it takes for the planner to detect a collision in the generated path and replan accordingly. The reactiveness test covers the complete 8x8m space of the indoor test facility. Therefore, the two tests are split into two different datasets. In both recordings, all sensors required for the ego-planner are logged. The ego-planner itself is not executed. For data recording, the flights are manually flown since the planner will only be tested in simulation for this scenario.

During the cube-escape recording, the drone is placed inside a cube of 3x3x2 meters (l*w*h). Three of the walls are solid, while the fourth one has fake trees. The wall with the trees has an opening in the middle with a gap size of 100cm. At the beginning of the dataset, the drone takes off from the centre of the cube, facing one of the solid walls with the camera. Then, the drone performs a one-and-a-half rotation scan by yawing at a constant speed of 90 degrees per second while staying in the centre of the cube. The scan ends with the camera facing one of the solid walls again. After the scan, the drone continues to hover in the middle of the cube for 20 seconds before landing. These last 20 seconds are used to sample paths during the simulations.
During the recording of indoor datasets, top-view/iso-view cameras positioned at the ceiling are used to record videos for future validation or presentations. Figure C.1 shows the varying gap size test with a gap of 60cm.

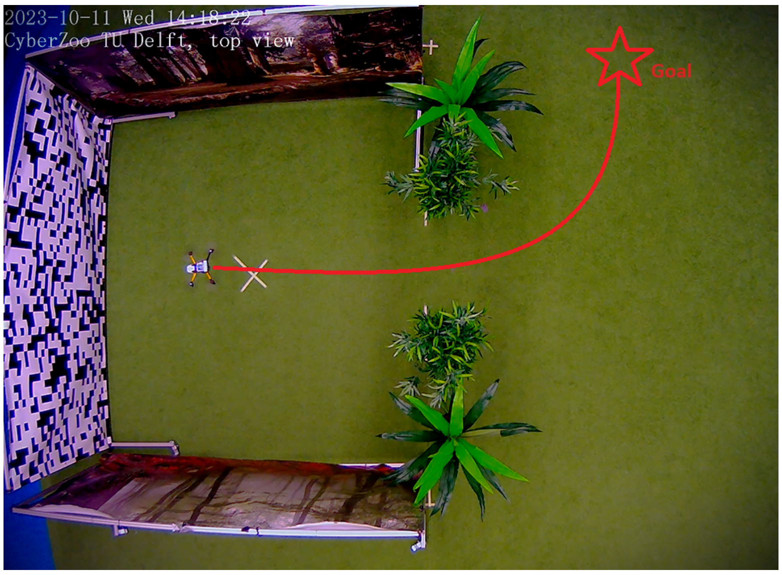


Figure C.1: Varying gap size test (60cm), top-view, in Cyberzoo for indoor dataset

During the reactive flight test, the drone is flown towards a set of fake trees in a straight line at constant speed. The test begins with the drone taking off from one corner of the test facility. It then flies at a constant speed towards the diagonally opposite corner. Just before the drone reaches the obstacles, it stops and hovers for five seconds before landing. The obstacles are placed 8 meters from the takeoff position, so they can only be detected when the drone is already in motion. This test is conducted at three different speeds, namely low speed (approximately 0.5m/s), medium speed (approximately 1.0m/s), and fast speed (to be determined by +-1.8m/s).

During simulation, the aim is that the planner detects the obstacles as soon as possible and replans from the initial path, which is straight to the goal, to a new collision-free path around the obstacles, towards the goal behind the obstacles. Figure C.2 shows the setup of this test with the drone flying and the desired trajectory from start to goal drawn in red.

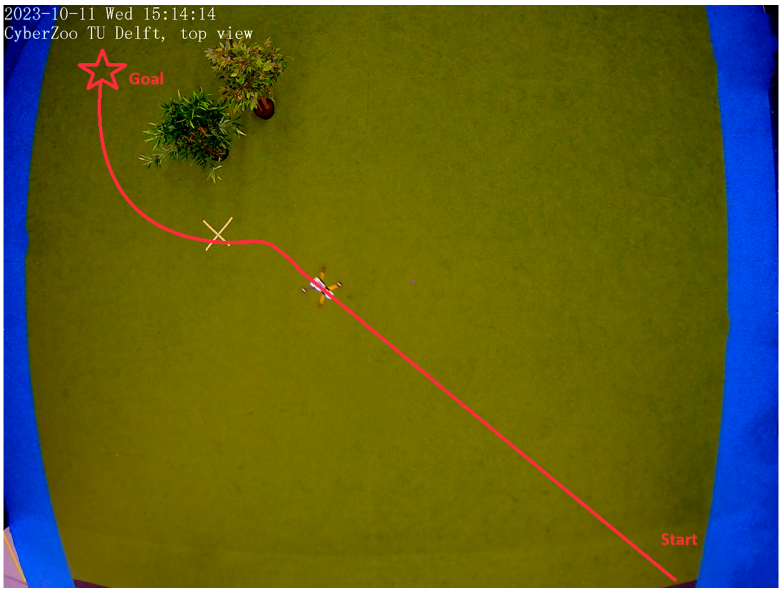


Figure C.2: Reactive flight test, slow (0.5 m/s), top-view, in Cyberzoo for indoor dataset

C.2. Record dataset outdoors

For the outdoor dataset, a forest is chosen to fly in and record sensor data. The forest was selected because of the relatively low height of its trees, which allows for GPS ground-truth position data. Additionally, a video of the test can be recorded from above with another off-the-shelf drone. To navigate through the forest, a walking trail with cluttered branches and multiple gaps of 1 meter was chosen. The drone has to navigate through a zigzag in the trail, which is approximately 20 meters long in total.

To record this dataset, first, a manual flight over the trail has been recorded without running the ego-planner. Afterwards, the drone navigated autonomously over the same trail using the planner. Since all autonomous flight tests over the trail were successful, one of the autonomous runs is chosen to create the forest dataset from.
The logged flight for the forest dataset had the ego-planner set to a maximum speed of 1m/s, maximum acceleration of 0.3m/s, and inflation of 0.3m.

In figure C.3, an image of the zigzag trail's entrance is shown to give an impression. In this image, also the setup of the drone is visible with the external GPS module on a carbon rod. As described in the report, this GPS mounted on a rod reduces EMC interference from the processor and camera onto the sensitive helix GPS antenna.

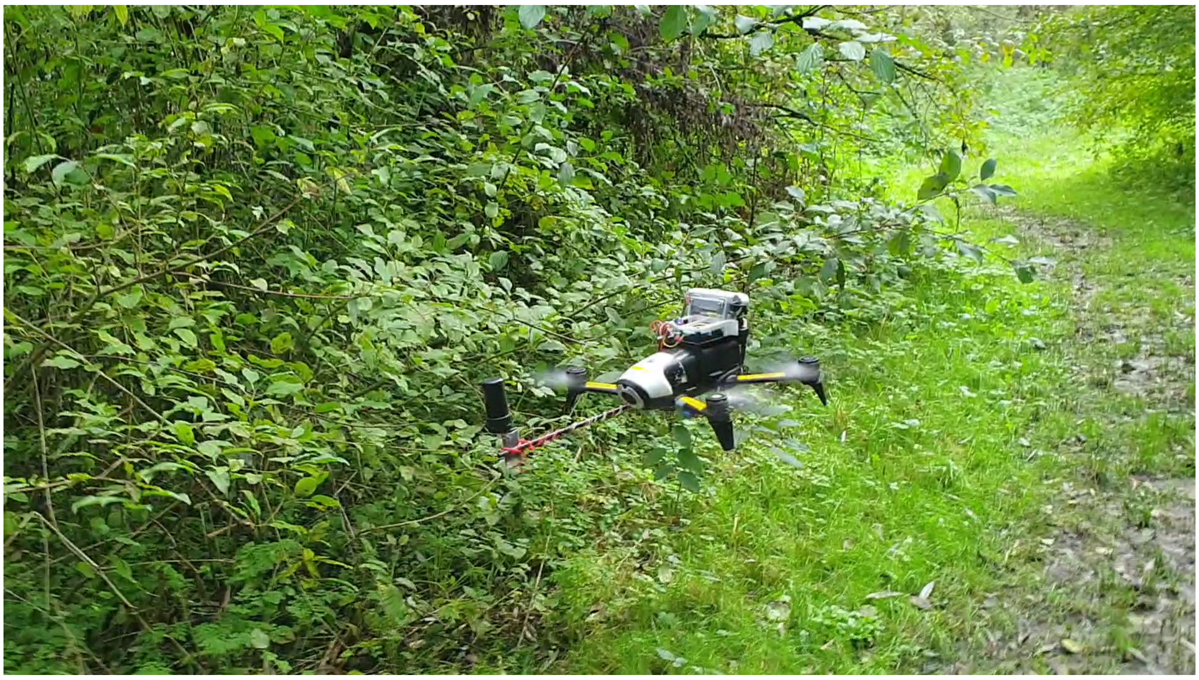


Figure C.3: Image of drone flying autonomously in the forest for the forest dataset recording

Figure C.4 shows the infrared and depth image recorded in the forest dataset. As the drone flew autonomously during the recording of this dataset, not only the sensor input is available, but also the obstacle detection, mapping, and trajectory planning. These are also shown in figure C.4. In the figure, annotations are made to get a better impression of what is shown in this screenshot from the RVIZ viewer.

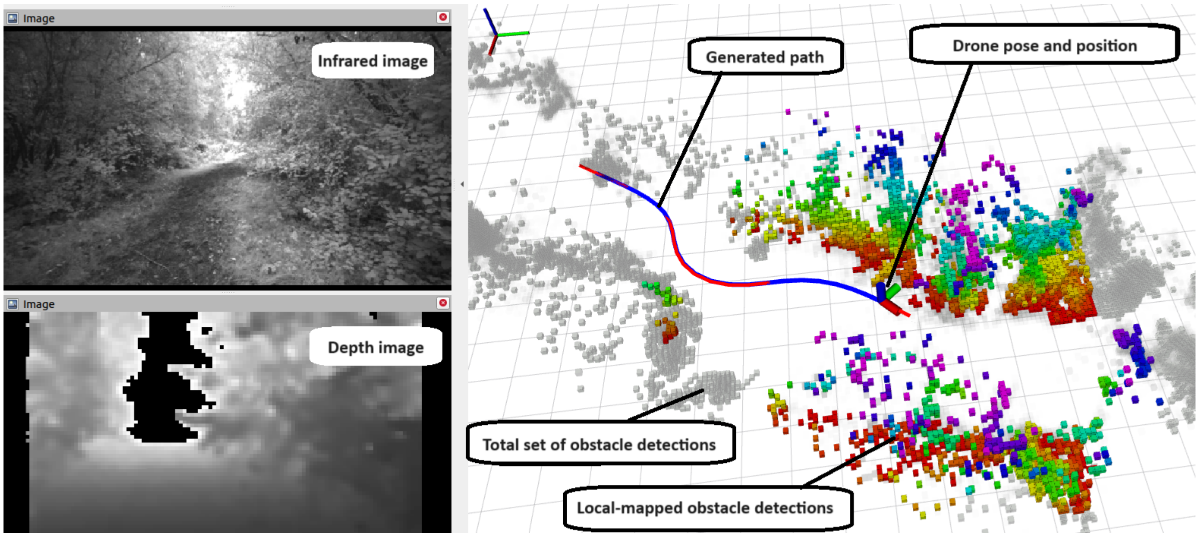


Figure C.4: Screenshot RVIZ - ego-planner visualisation for forest dataset

C.3. Content of dataset recordings

In the data recordings applied in the two above-described methods, all data communicated over the ROS network (topics) is saved onboard the drone at the SD card. To give insight in the recorded topics and the number of stored data points, the content of the outdoor dataset file is shown below in code listing C.1. This overview of the dataset content is created by using the bash command *rosbag_info_bos_6_r4_GPS_VINS.bag*. This dataset does not only contain the data from the outdoor flight, it also contains the post-processed position data of VINS-fusion.

Code Listing C.1: Content Rosbag dataset outdoors

```
path:          bos_6_r4_GPS_VINS.bag
version:       2.0
duration:      1:38s (98s)
start:         Nov 08 2023 10:53:43.22 (1699437223.22)
end:           Nov 08 2023 10:55:21.58 (1699437321.58)
size:          1.5 GB
messages:      106857
compression:   none [1412/1412 chunks]
topics:
    /bebop/bebop/nodelet/manager/bond                                    188 msgs
    /bebop/cmd/vel                                                       470 msgs
    /bebop/fix                                                            94 msgs
    /bebop/goto                                                           79 msgs
    /bebop/joint/states                                                  466 msgs
    /bebop/odom                                                          466 msgs
    /bebop/states/ardrone3/PilotingState/AltitudeChanged                 467 msgs
    /bebop/states/ardrone3/PilotingState/AttitudeChanged                 467 msgs
    /bebop/states/ardrone3/PilotingState/FlyingStateChanged               10 msgs
    /bebop/states/ardrone3/PilotingState/PositionChanged                  94 msgs
    /bebop/states/ardrone3/PilotingState/SpeedChanged                    467 msgs
    /bebop/states/common/CommonState/BatteryStateChanged                  12 msgs
    /bebop/yaw                                                           466 msgs
    /bebop/guidance/visualisation                                         79 msgs
    /broadcast/bspline                                                    87 msgs
    /camera/depth/camera/info                                           1412 msgs
    /camera/depth/color/points                                          1412 msgs
    /camera/depth/image/rect/raw                                        1412 msgs
```

```
/camera/depth/image/rect/raw/compressed                      1412 msgs
/camera/depth/metadata                                       1412 msgs
/camera/imu                                                 18863 msgs
/camera/infra1/camera/info                                   1411 msgs
/camera/infra1/image/rect/raw                                1411 msgs
/camera/infra1/image/rect/raw/compressed                     1411 msgs
/camera/infra1/metadata                                      1411 msgs
/camera/infra2/camera/info                                   1412 msgs
/camera/infra2/image/rect/raw                                1412 msgs
/camera/infra2/image/rect/raw/compressed                     1412 msgs
/camera/infra2/metadata                                      1411 msgs
/camera/realsense2/camera/manager/bond                        188 msgs
/collision/check/duration                                    1769 msgs
/cpu/monitor/bebop/bebop/nodelet/cpu                          161 msgs
/cpu/monitor/bebop/bebop/nodelet/mem                          161 msgs
/cpu/monitor/bebop/bebop/nodelet/manager/cpu                  163 msgs
/cpu/monitor/bebop/bebop/nodelet/manager/mem                  163 msgs
/cpu/monitor/bebop/robot/state/publisher/cpu                  163 msgs
/cpu/monitor/bebop/robot/state/publisher/mem                  163 msgs
/cpu/monitor/bebop/guidance/cpu                               163 msgs
/cpu/monitor/bebop/guidance/mem                               163 msgs
/cpu/monitor/camera/realsense2/camera/cpu                     163 msgs
/cpu/monitor/camera/realsense2/camera/mem                     163 msgs
/cpu/monitor/camera/realsense2/camera/manager/cpu             161 msgs
/cpu/monitor/camera/realsense2/camera/manager/mem             161 msgs
/cpu/monitor/drone/0/ego/planner/node/cpu                     163 msgs
/cpu/monitor/drone/0/ego/planner/node/mem                     163 msgs
/cpu/monitor/drone/0/traj/server/cpu                          163 msgs
/CPU/monitor/drone/0/traj/server/mem                          163 msgs
/cpu/monitor/gps/gps/to/pose/conversion/node/cpu              163 msgs
/cpu/monitor/gps/gps/to/pose/conversion/node/mem              163 msgs
/cpu/monitor/gps/i2c/gps/node/cpu                             163 msgs
/cpu/monitor/gps/i2c/gps/node/mem                             163 msgs
/cpu/monitor/gps/nmea/topic/driver/node/cpu                   163 msgs
/cpu/monitor/gps/nmea/topic/driver/node/mem                   163 msgs
/cpu/monitor/gps/set/gps/reference/node/cpu                   163 msgs
/cpu/monitor/gps/set/gps/reference/node/mem                   163 msgs
/cpu/monitor/record/1696598434779780845/cpu                   163 msgs
/cpu/monitor/record/1696598434779780845/mem                   163 msgs
/cpu/monitor/robot/state/publisher/cpu                        163 msgs
/CPU/monitor/robot/state/publisher/mem                        163 msgs
/CPU/monitor/rosout/cpu                                       163 msgs
/cpu/monitor/rosout/mem                                       163 msgs
/cpu/monitor/sbus/cmd/vel/cpu                                 163 msgs
/cpu/monitor/sbus/cmd/vel/mem                                 163 msgs
/cpu/monitor/sbus/node/cpu                                    163 msgs
/CPU/monitor/sbus/node/mem                                    163 msgs
/CPU/monitor/total/active/mem                                 163 msgs
/cpu/monitor/total/available/mem                              163 msgs
/cpu/monitor/total/buffers/mem                                163 msgs
/cpu/monitor/total/cached/mem                                 163 msgs
/CPU/monitor/total/CPU                                        163 msgs
/CPU/monitor/total/free/mem                                   163 msgs
/CPU/monitor/total/inactive/mem                               163 msgs
/cpu/monitor/total/shared/mem                                 163 msgs
/cpu/monitor/total/slab/mem                                   163 msgs
```

```
/cpu/monitor/total/used/mem                                          163 msgs
/diagnostics                                                         466 msgs
/drone/0/ego/planner/node/ego/result/average/time                     87 msgs
/drone/0/ego/planner/node/ego/result/init/time                        87 msgs
/drone/0/ego/planner/node/ego/result/optimise/time                    87 msgs
/drone/0/ego/planner/node/ego/result/refine/time                      87 msgs
/drone/0/ego/planner/node/final/trajectory                           100 msgs
/drone/0/ego/planner/node/global/list                                  2 msgs
/drone/0/ego/planner/node/grid/map/occupancy                         855 msgs
/drone/0/ego/planner/node/grid/map/occupancy/inflate                 855 msgs
/drone/0/ego/planner/node/grid/map/project/image/time               1411 msgs
/drone/0/ego/planner/node/grid/map/raycast/image/time               1411 msgs
/drone/0/ego/planner/node/grid/map/upd/local/map/time               1411 msgs
/drone/0/ego/planner/node/init/list                                  120 msgs
/drone/0/ego/planner/node/optimal/list                               107 msgs
/drone/0/planning/bspline                                             89 msgs
/drone/0/planning/data/display                                      8141 msgs
/enable/autopilot/wireless/rc                                        428 msgs
/gps/ecef                                                             43 msgs
/gps/fix                                                              43 msgs
/gps/gps/position/NED                                                 43 msgs
/gps/gps/position/NED/offset                                          43 msgs
/gps/gps/position/NWU                                                 43 msgs
/gps/gps/transform                                                    43 msgs
/gps/nmea/sentence                                                   138 msgs
/gps/time/reference                                                   43 msgs
/itterations/trajectory                                               87 msgs
/move/base/simple/goal                                                 2 msgs
/new/valid/trajectory                                                 87 msgs
/pos/setpoint                                                       8848 msgs
/position/cmd                                                       8848 msgs
/rosout                                                             1794 msgs
/rosout/agg                                                         2004 msgs
/sbus                                                                470 msgs
/tf                                                                  975 msgs
/trajectory/collision                                                 71 msgs
/vel/setpoint                                                       8848 msgs
/vins/fusion/camera/pose                                             702 msgs
/vins/fusion/camera/pose/visual                                      702 msgs
/vins/fusion/extrinsic                                               702 msgs
/vins/fusion/image/track                                            1410 msgs
/vins/fusion/key/poses                                               701 msgs
/vins/fusion/keyframe/point                                          518 msgs
/vins/fusion/keyframe/pose                                           518 msgs
/vins/fusion/margin/cloud                                            705 msgs
/vins/fusion/odometry                                                702 msgs
/vins/fusion/path                                                    702 msgs
/vins/fusion/point/cloud                                             705 msgs
```

To give a better understanding how the environment is perceived and recorded by the drone in the forest dataset, a series of images below shows images and the obstacle map.

Figure C.5 below first shows an infrared image taken from the drone at 18.1 seconds in the dataset. Thereafter, figure C.6 shows the depth image taken from the drone at the same location and at the same time.

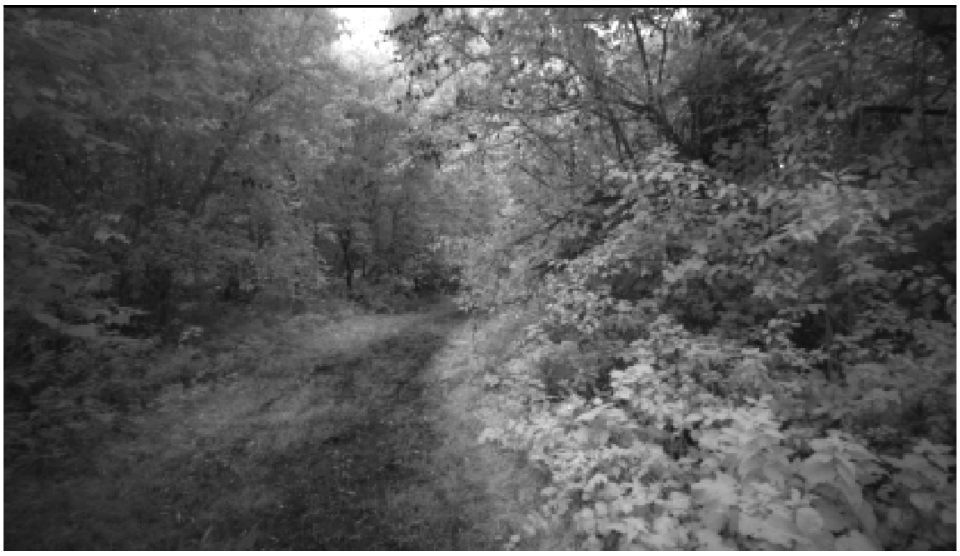


Figure C.5: Infrared image from dataset at 18.1 seconds

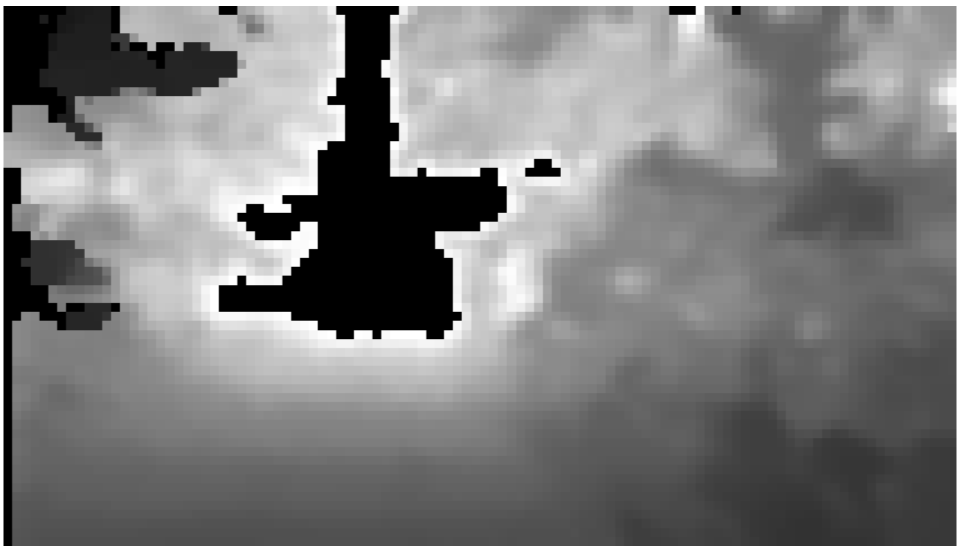


Figure C.6: Depth image from dataset at 18.1 seconds

The following three screenshots from Rviz show all mapped obstacles together from the start to the end of the simulation. In the Rviz visualisation, the obstacle voxels' colours change in the z-direction to make the height differences more visible. The green line in the screenshots shows the drone's flown path in the dataset. The grid at the bottom is the ground plane, with a 1x1m grid size as a reference for dimensions. Figure C.7 shows a top-down overview of the mapped environment. Figure C.8 shows a side view where the heights are better visible. Figure C.9 shows a front view from the location where the drone took off and entered the narrow passage in the forest.

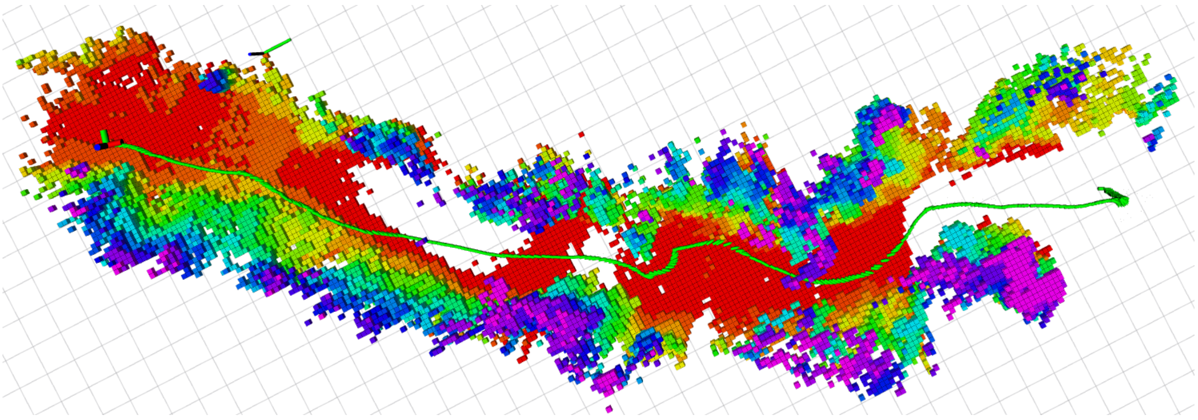


Figure C.7: Top-view of mapped 3D environment in dataset and the flown path

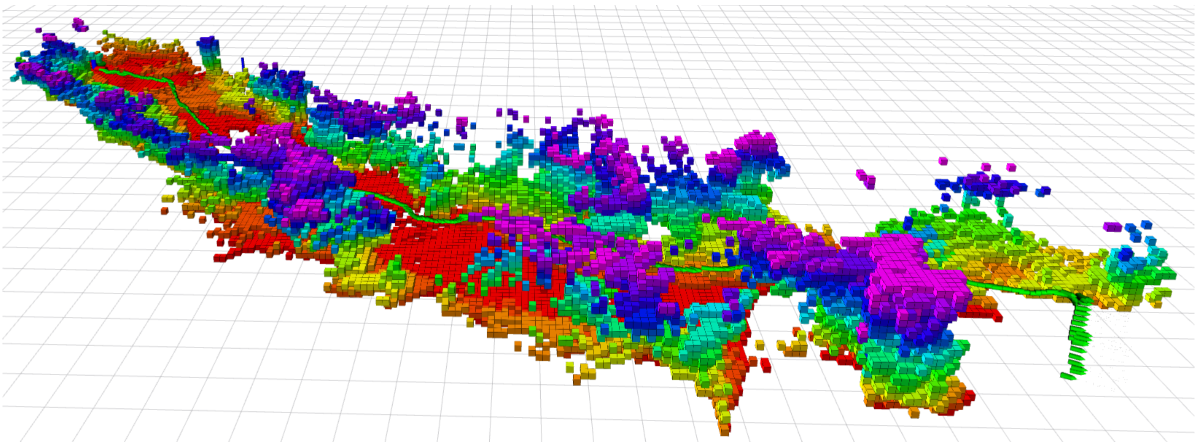


Figure C.8: Side-view of mapped 3D environment in dataset and the flown path

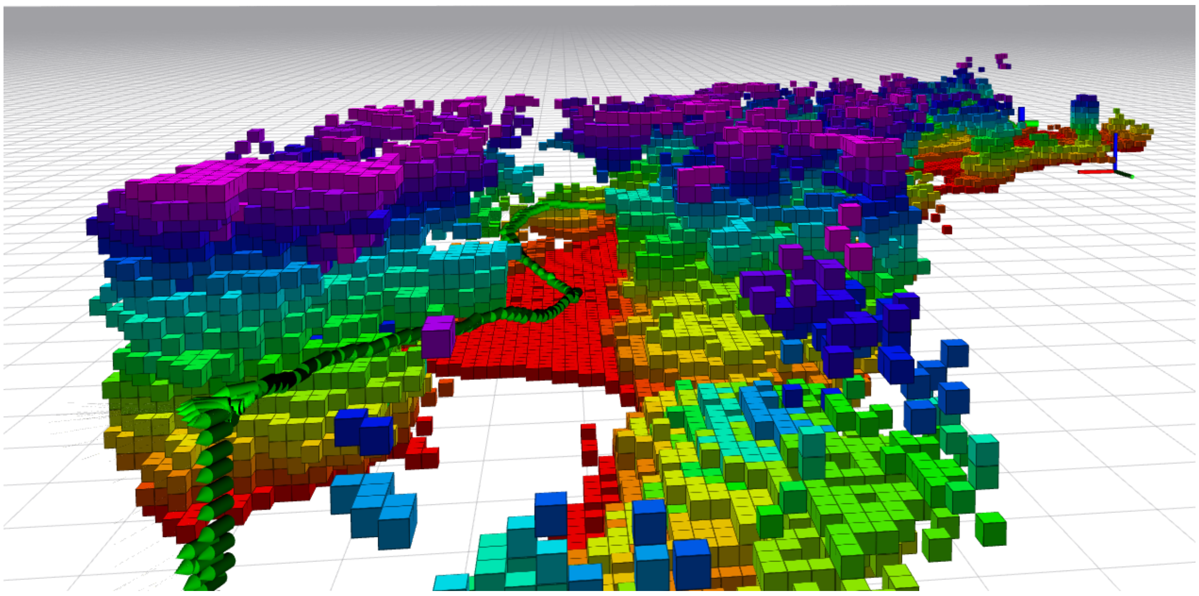


Figure C.9: Front-view of mapped 3D environment in dataset and the flown path

C.4. Localisation comparison

When recording the dataset, GPS and image data is additionally recorded to compare different localisation techniques. This is described in section 6.2.2. This section shows some enlarged figures from these localisation methods.

Figure C.10 shows VINS-fusion localisation with optic flow detection on the left and feature tracking on the right.

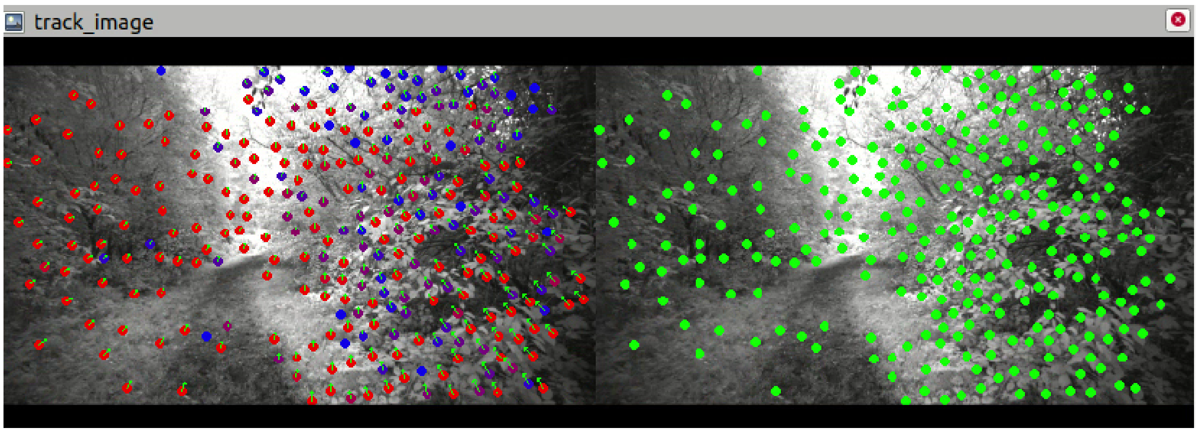


Figure C.10: Tracking features in image with VINS

C.11 shows the tracked features in a 3D map where the drone traverses the forest. In this figure, the green line shows the localisation estimate from VINS-fusion, while the red-green-blue axis represents the drone's position estimated by dead-reckoning.

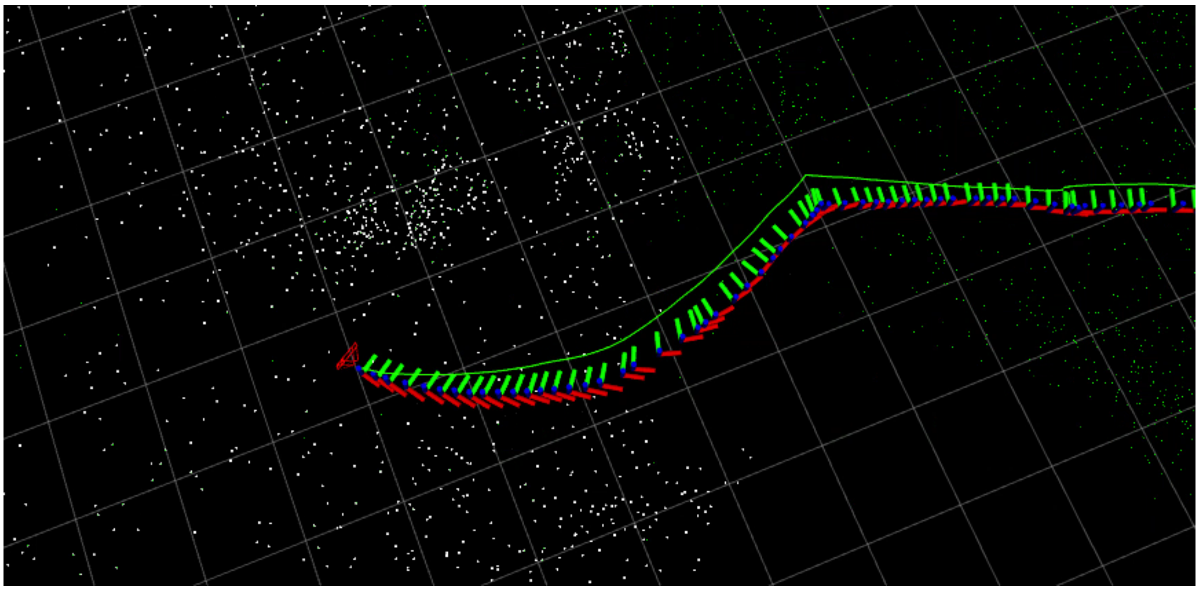


Figure C.11: Mapping 3D tracked points with VINS

C.12 shows the localisation of GPS (purple balls), VINS-fusion (green line), and dead-reckoning (red dots) overlaid in one figure with on the background the mapped obstacles of the forest where the drone flew.

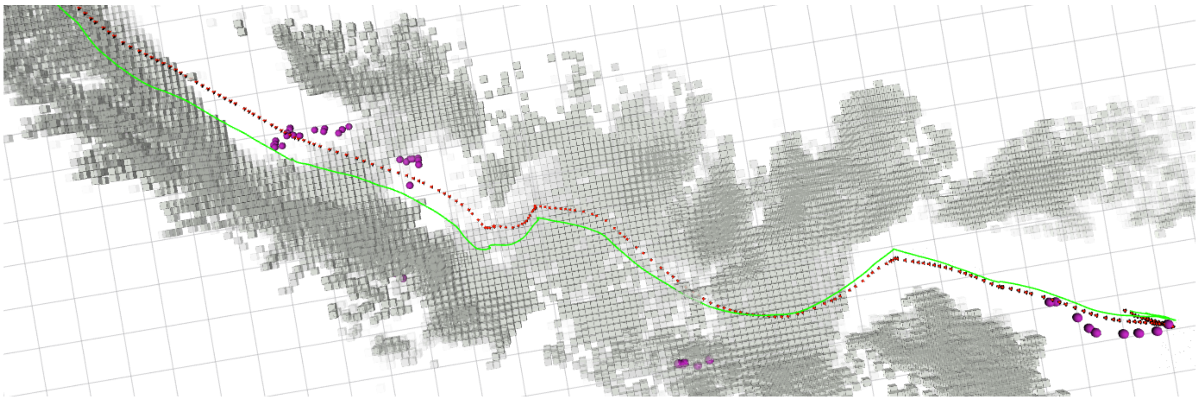


Figure C.12: Drone system architecture for autonomous navigation in the rainforest

D
Analysis EGO-planner processes

In this appendix, additional results of the downscaling research are shown. First, the process of the planning algorithm is shown to give a better inside in its functionality. Then, the CPU and memory load are shown for varying planner configurations. Thereafter, loop durations for specific processes inside the planning algorithm are visualised and compared for varying planner configurations. And, lastly, some visual comparison is shown on different behaviour of the trajectory planning when varying planner configurations.

D.1. Algorithm structure

To give insights in the functionality and structure of the EGO-planner, this section shows the pseudocode of the algorithm by [18]. Figure D.1 below shows the Psuedo code for checking which points of the path are inside mapped obstacles (algorithm 1) and the pseudocode to rebound the path outwards of the obstacle until it does not collide anymore (algorithm 2).

Algorithm 1 CheckAndAddObstacleInfo

```
1: Notation: Environment $\mathcal{E}$, Control Points Struct $\mathbf{Q}$, Anchor Points $\mathbf{p}$, Repulsive Direction Vector $\mathbf{v}$, Colliding Segments $\mathbf{S}$
2: Input: $\mathcal{E}$, $\mathbf{Q}$
3: for $\mathbf{Q}_i$ in $\mathbf{Q}$ do
4:     if FindConsecutiveCollidingSegment($\mathbf{Q}_i$) then
5:         $\mathbf{S}$.push_back(GetCollisionSegment())
6:     end if
7: end for
8: for $\mathbf{S}_i$ in $\mathbf{S}$ do
9:     $\mathbf{\Gamma} \leftarrow$ PathSearch($\mathcal{E}$, $\mathbf{S_i}$)
10:    for $\mathbf{S}_i$.begin $\leq j \leq$ $\mathbf{S}_i$.end do
11:        $\{\mathbf{p}, \mathbf{v}\} \leftarrow$ Find_p_v_Pairs($\mathbf{Q}_j, \mathbf{\Gamma}$)
12:        $\mathbf{Q}_j$.push_back($\{\mathbf{p}, \mathbf{v}\}$)
13:    end for
14: end for
```

Algorithm 2 Rebound Planning

```
1: Notation: Goal $\mathcal{G}$, Environment $\mathcal{E}$, Control Point Struct $\mathbf{Q}$, Penalty $J$, Gradient $\mathbf{G}$
2: Initialize: $\mathbf{Q} \leftarrow$ FindInit($\mathbf{Q}_{last}$, $\mathcal{G}$)
3: while $\neg$ IsCollisionFree($\mathcal{E}, \mathbf{Q}$) do
4:     CheckAndAddObstacleInfo($\mathcal{E}, \mathbf{Q}$)
5:     $(J, \mathbf{G}) \leftarrow$ EvaluatePenalty($\mathbf{Q}$)
6:     $\mathbf{Q} \leftarrow$ OneStepOptimize($J, \mathbf{G}$)
7: end while
8: if $\neg$ IsFeasible($\mathbf{Q}$) then
9:     $\mathbf{Q} \leftarrow$ ReAllocateTime($\mathbf{Q}$)
10:    $\mathbf{Q} \leftarrow$ CurveFittingOptimize($\mathbf{Q}$)
11: end if
12: return $\mathbf{Q}$
```

Figure D.1: Psuedo code for collision detection and path rebound by [18]

D.2. CPU load

To give insights into how different configurations of the EGO-planner load the processor board, the CPU load of the EGO-planner process in Linux is measured and stored in the Rosbag file. In the analysis, for every simulation, a time series plot is created with the processor load over the simulation and a boxplot with the statistics of this data. The variation of voxel grid resolution gives the most significant change in processor load. For this variation, the processor load plots are shown below in figure D.2. To generate this data, the planner is simulated with these settings: skip-pixels=2, voxel-grid=[0.05 0.10 0.15 0.20], local-map=6.0m, camera-rate=15Hz.

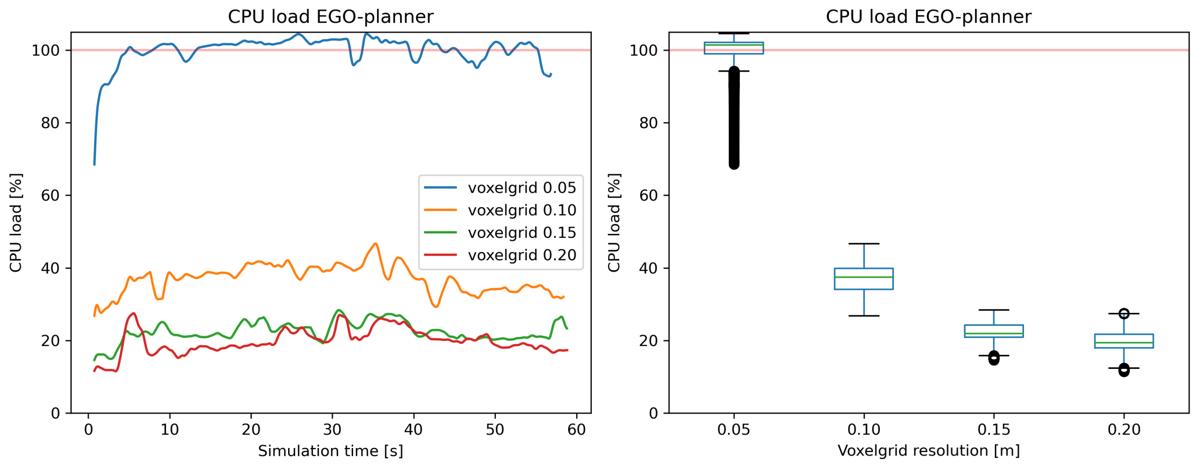


Figure D.2: CPU load EGO-planner for 1 core. Varying voxel grid resolution

To give contrast to how other variations affect the CPU load, figure D.3 below shows the load for a varying depth image resolution. The difference in CPU load is less significant compared to the voxel grid resolution variation but also shows a clear pattern of load reduction. Where the load varied from 20 to 100% in the first plot, this plot from 17 to 24% load. Note here the scale of this plot only goes from 0% to 40% CPU load, while the previous plot goes up to 105% CPU load. To generate this data, the planner was used with the following settings: skip-pixels=[1 2 3 4], voxel-grid=0.10, local-map=4.0m, and camera-rate=15Hz.

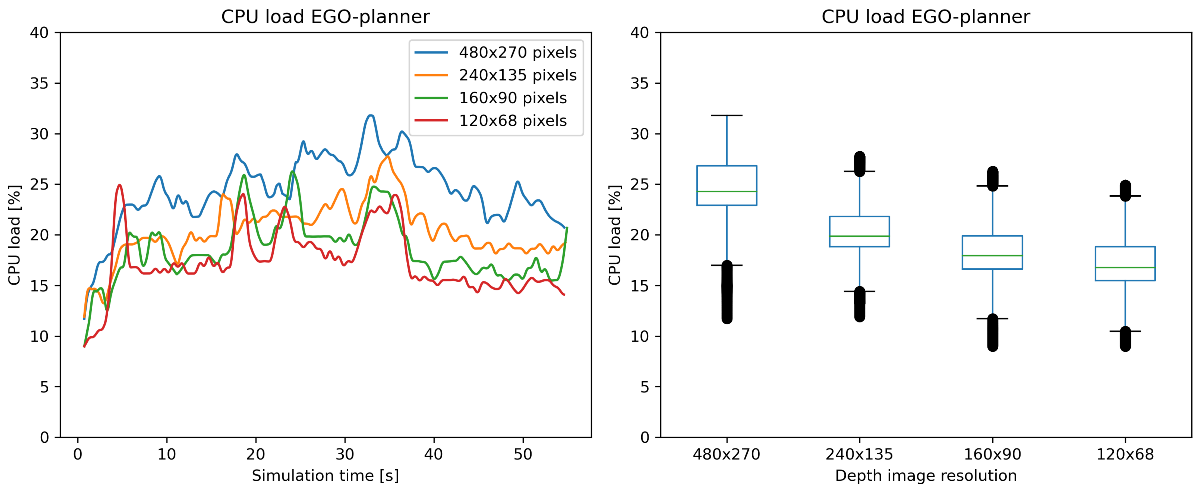


Figure D.3: CPU load EGO-planner for 1 core. Varying depth image resolution

D.3. Memory usage and map size

Table D.1 below shows a comparison of measured RAM memory usage. The table first shows how, for limited memory usage, the map size can increase when the voxel grid resolution is more coarse. Thereafter, the table shows how memory usage can change when the map size (in meters) is constant for different voxel grid resolutions.

Table D.1: Test memory usage for total map size

Variation voxel grid	**Total map size**	**Voxel blocks**	**Memory usage**
Constant number of voxel blocks			
SP=2, **VR=0.05**, LR=6, CR=0	50x50x2.5m	1000x1000x50	8.631*10e8 Bytes
SP=2, **VR=0.10**, LR=6, CR=0	100x100x5m	1000x1000x50	8.618*10e8 Bytes
SP=2, **VR=0.15**, LR=6, CR=0	150x150x7.5m	1000x1000x50	8.611*10e8 Bytes
SP=2, **VR=0.20**, LR=6, CR=0	200x200x10m	1000x1000x50	8.606*10e8 Bytes
Constant map size			
SP=2, **VR=0.05**, LR=6, CR=0	50x50x2.5m	1000x1000x50	8.631*10e8 Bytes
SP=2, **VR=0.10**, LR=6, CR=0	50x50x2.5m	500x500x25	2.042*10e8 Bytes
SP=2, **VR=0.15**, LR=6, CR=0	50x50x2.5m	333x333x16	1.378*10e8 Bytes
SP=2, **VR=0.20**, LR=6, CR=0	50x50x2.5m	250x250x12	1.224*10e8 Bytes

D.4. Loop duration for sub-processes

To give insights into the intermediate computing steps in the EGO planner, the loop durations are registered during test flights and simulations. This gives a better understanding of how the individual processes vary in computation time when varying planner parameters. This section shows some results of these loop durations, which were insightful in determining the parameters that impact performance and processor load the most.

D.4.1. Perception

Figure D.4 below shows loop durations for the drones perception with variations of voxel grid resolutions. In this figure, plots are shown for the three sub-processes (3D projection, ray-casting, and map update) within the drone's obstacle mapping. To generate this data, the planner is simulated with these settings: skip-pixels=2, voxel-grid=[0.05 0.10 0.15 0.20], local-map=6.0m, camera-rate=15Hz.

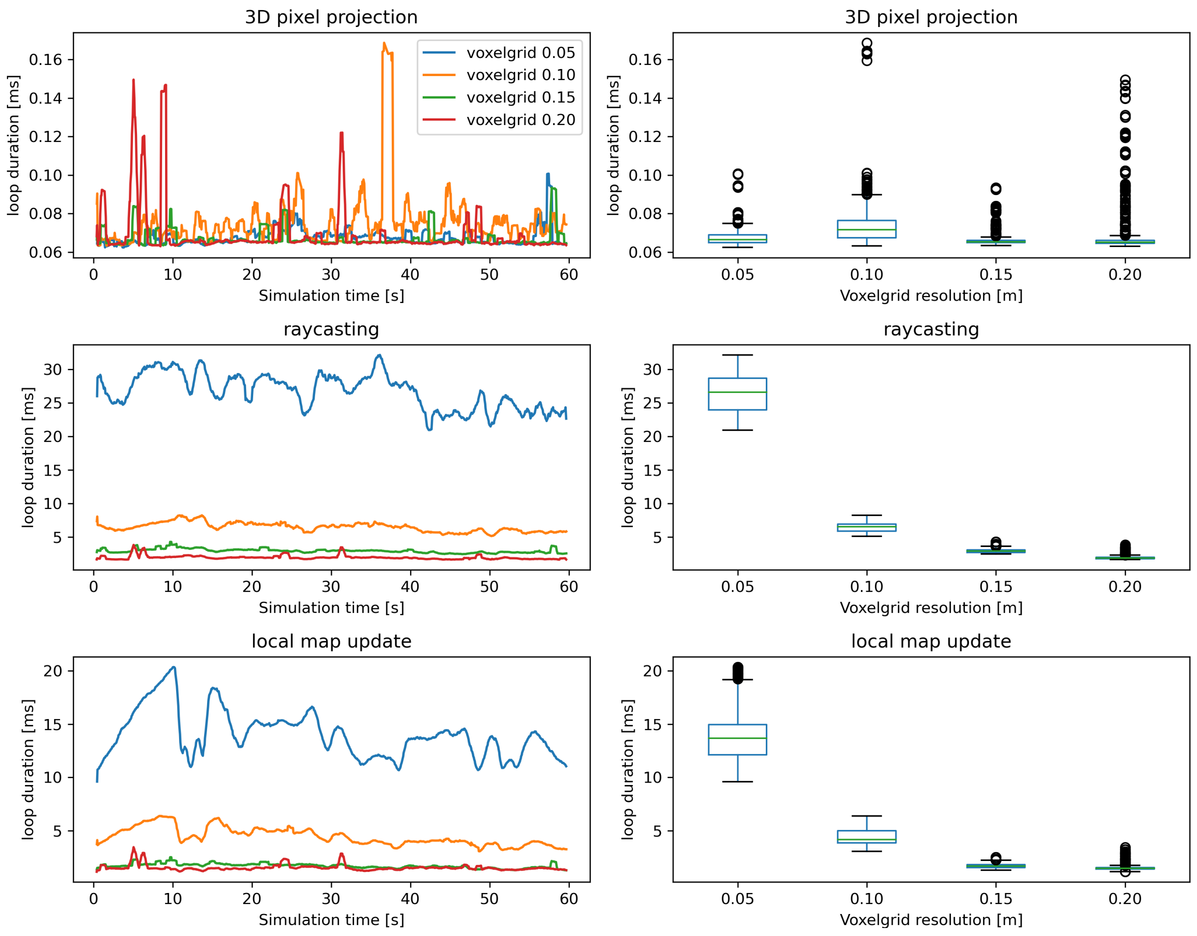


Figure D.4: Loop durations sub-processes EGO-planner perception. Varying voxel grid resolution

The loop duration boxplots show a pattern in the relation between voxel grid resolution and durations. Raycasting and local-map-update durations reduce exponentially when the resolution is reduced from 0.05m to 0.20m, with loops getting nearly 10 times shorter. Meanwhile, the 3D pixel projection boxplot shows no significant variation in loop duration.

The pattern of similar durations for 3D pixel projection is expected as for every new received depth image the same amount of pixels coordinates has to be transformed from 2D to 3D, no matter the voxel grids' resolution. It is unclear why the 3D pixel project shows high peaks. Every pixel from the depth image is transformed to a 3D coordinate by exactly the same matrix, so this gives no reason for

variation. But, given the significantly lower scale than the raycasting and local-map-update, peaks in the graph are still non-significant longer durations.
When using raycasting at a higher resolution, the computation time to update the voxels increases because more voxels are crossed. Similarly, for local-map-update, the computation time increases when more voxels have to be updated with a higher resolution. For instance, if the resolution doubles, the number of blocks in the x, y, and z directions is twice as much, resulting in eight times more voxels to update.

Figure D.5 below shows plots of loop times for perception, but this time for different camera resolutions instead of varying voxel grid resolution. To generate this data, the planner was used with the following settings: skip-pixels=[1 2 3 4], voxel-grid=0.10, local-map=4.0m, and camera-rate=15Hz. To clarify the number of skipped pixels, 1 out of every N (skipped pixels) is used for obstacle mapping. So with 1 skipped pixel, 1 out 1 pixel is used, thus all pixels. With 4 skipped pixels, 1 out of 4 columns and rows is skipped when iterating over all pixels in the image. Therefore, with 4 'skipped pixels', only 1/8th of all pixels remain. Thus, an image with 480x270 pixels is used as an image with 120x68 pixels.

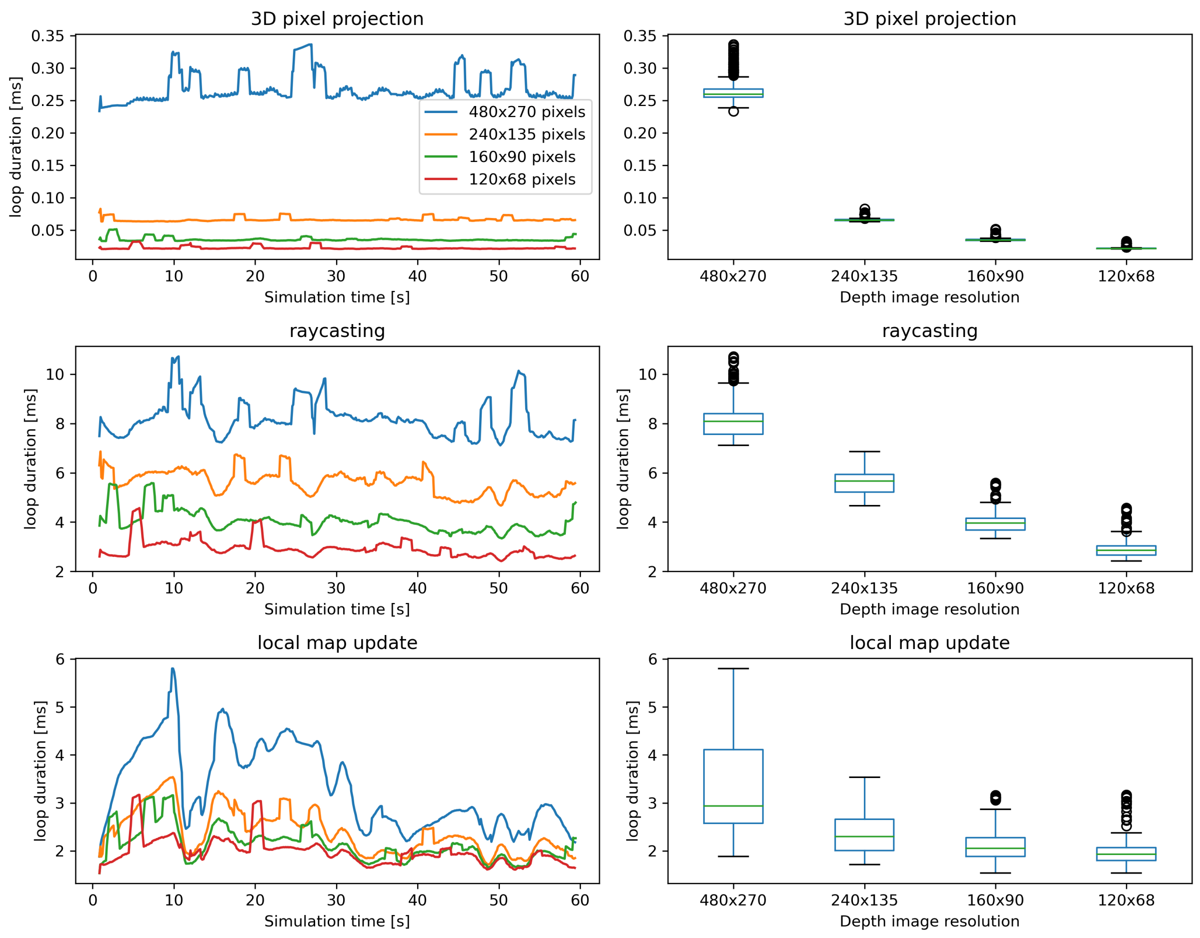


Figure D.5: Loop times sub-processes EGO-planner perception. Varying depth image resolution

In the shown plot, the 3D pixel projection boxplot shows an exponential reduction in loop duration when the number of pixels is reduced. As both rows and columns are skipped, the total number of pixels to project in 3D space reduces quadratically. Therefore, the curve in the boxplot is as expected.

The raycasting shows a near-linear reduction in loop duration when more pixels are skipped. This curve was expected to have an exponential shape as it has quadratically fewer pixels to process, as with the 3D projection. One potential reason the relation is near linear is that the 3D projected points are first clustered in the voxel grid. So, multiple detections that cluster in the same voxel cube will only be raycasted once, as the raycast function raycasts the voxel's centre. Therefore, with a high-resolution image, more detections will be clustered in the same voxel and thus there are fewer ray-casted points than the number of 3D detections.

The local-map-update loop was expected to show a near-constant duration for the different image resolutions. But in the boxplot, there is a slight reduction visible in duration for lower image resolutions. The most likely found explanation is that by the low-resolution image, fewer obstacles are mapped, and, therefore, there are fewer voxels to inflate for the inflated obstacle map.

D.4.2. Planning

As with the perception section, this section starts with simulation results for a varying voxel grid resolution and the other parameters fixed. The time series and boxplot are plotted below in figure D.6 for a varying voxel grid resolution. And the same plots are shown in figure D.7 for a varying depth image resolution.

For the varying voxel resolution, the initialisation and optimisation duration doubles. The initial hypothesis was that a more course voxel grid would give less computation and, therefore, a shorter duration. When inspecting the simulation, it was found that the mapping of 0.20m voxel size barely shows an opening on the forest's trail for the planner to find. Due to the coarse resolution, the gaps were hard or invisible to detect and plan a route through. Other than that, the plots show no significant variation in loop duration. Note the y-axis scale for the plots.

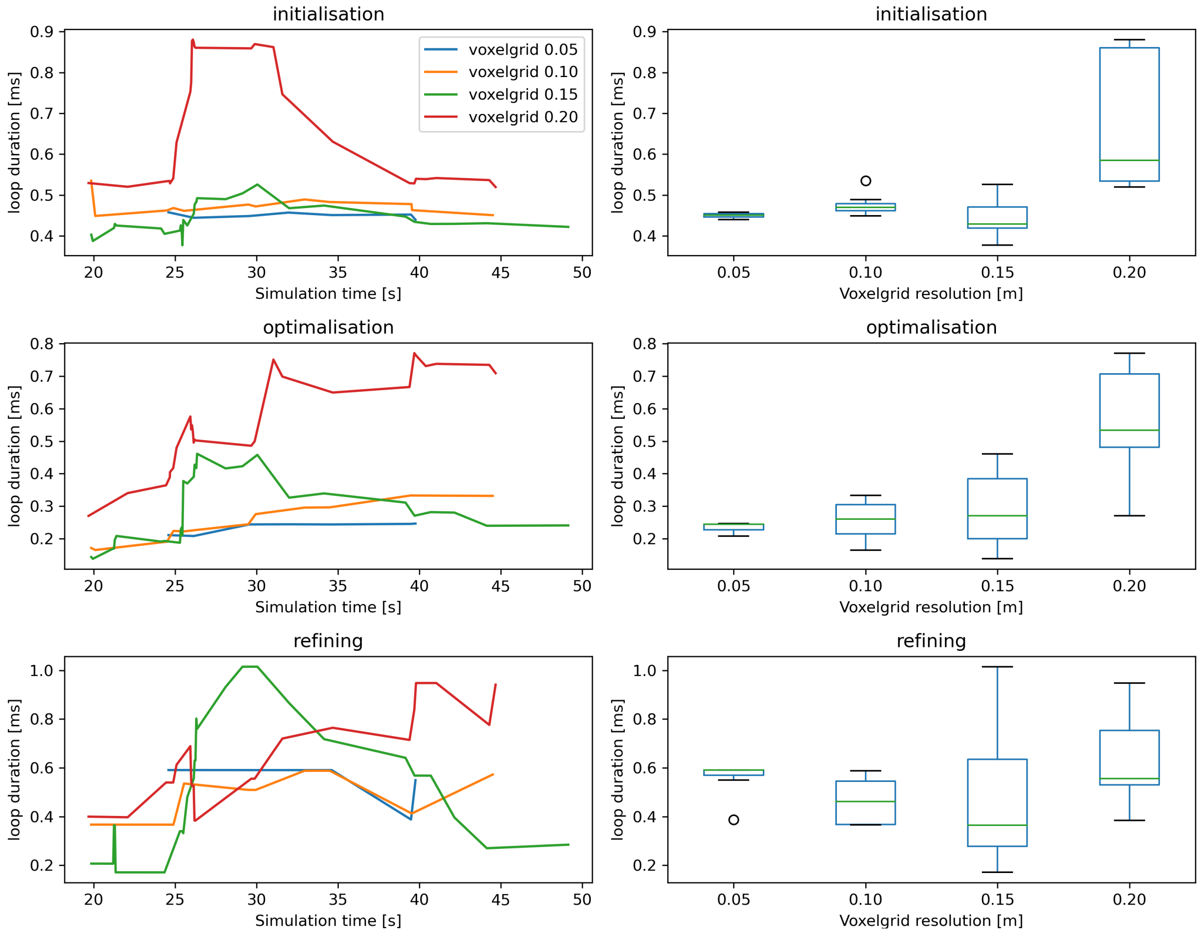


Figure D.6: Loop durations sub-processes EGO-planner planning. Varying voxel grid resolution

For the varying depth image resolution, a small, non-significant duration decrease is visible for a lower resolution. Note the scale of the y-axis. This research does not clearly answer why this decreases as image resolution has only direct effect on the mapping process.
One possible reason can be that some obstacles are missed, and therefore, the mapping contains fewer obstacles and lets the planner find a path more easily, although possibly with collisions.
Another reason could be that the lower resolution gives the mapping process a lower duration and the CPU a lower load. The parallel running threads on the processor might affect the duration of other processes, also when the CPU is not fully utilised.

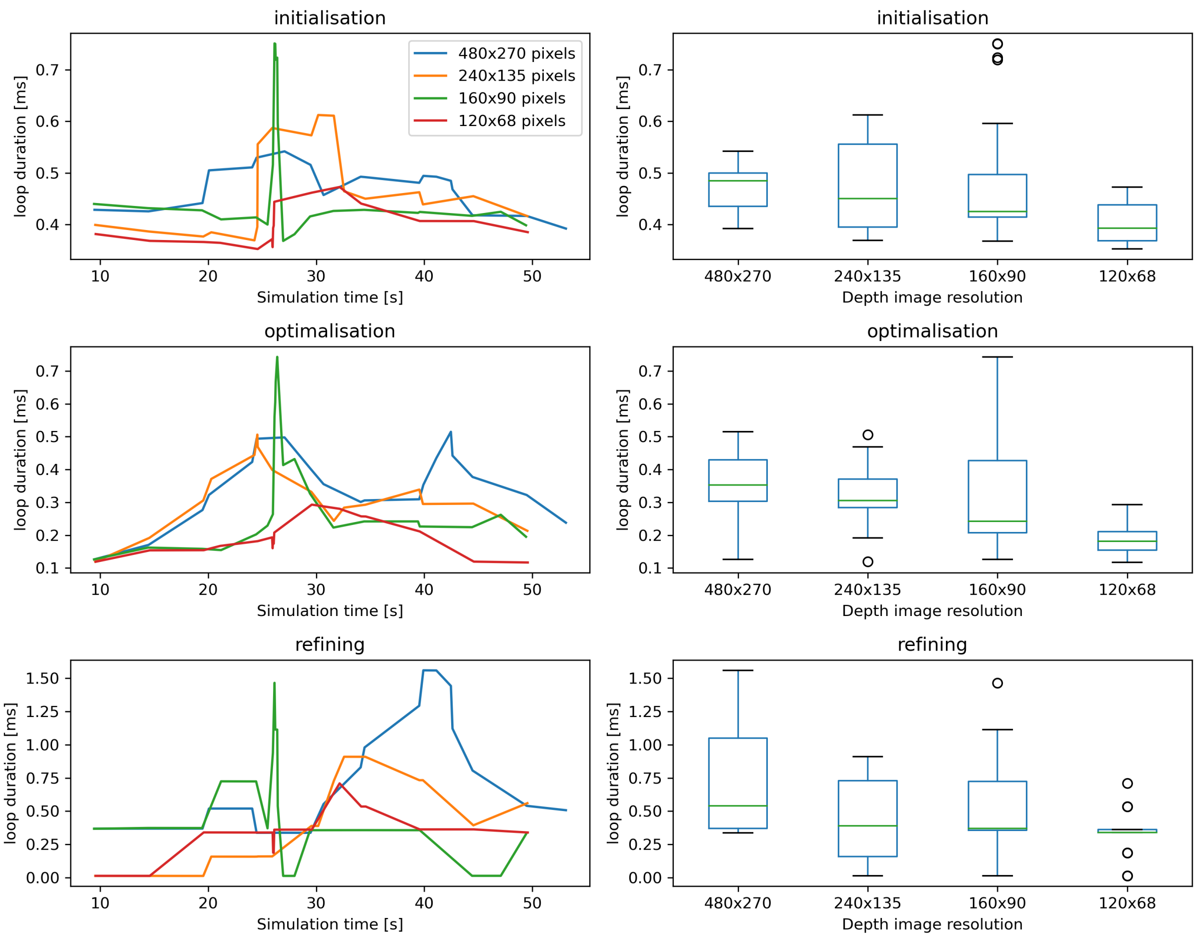


Figure D.7: Loop durations sub-processes EGO-planner planning. Varying depth image resolution

D.4.3. Collision check

To observe the loop duration for the collisioncheck, this is recorded in the ROSbag file during simulation and plotted in a time-series and boxplot graph. Figure D.8 shows the plots for a varying voxelgrid resolution. And figure D.9 shows the plots for a varying depth image resolution.

In the collision check loop, the active trajectory is checked with the obstacle map if there are no collisions with mapped obstacles. The planner tries to find a trajectory, and once one is found, only a new one is generated when this is triggered in the system, for example, when the collision check finds a collision in the future. The mapping runs continuously in a separate thread and updates at every new received depth image. The collision check function is called after every update of the obstacle mapping.

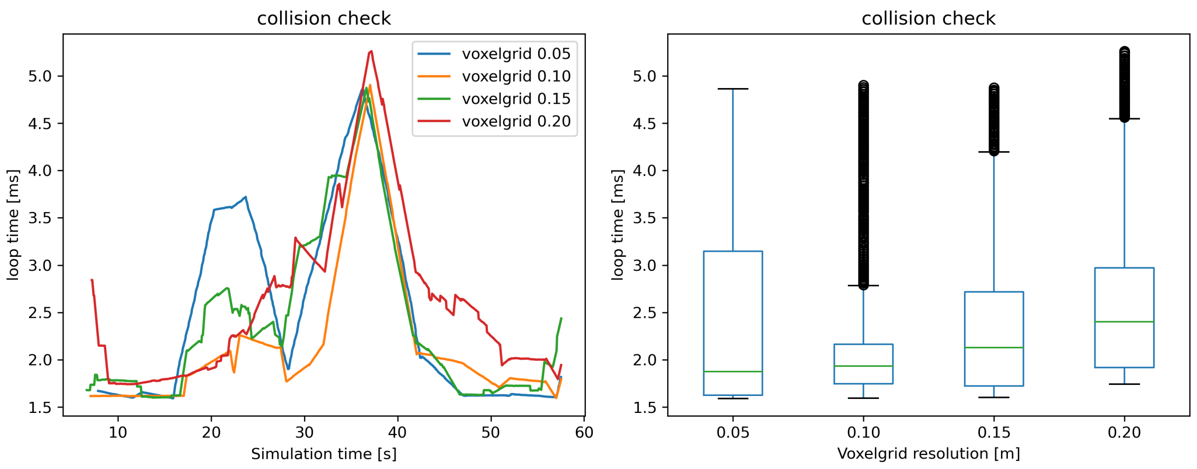


Figure D.8: Loop durations sub-processes EGO-planner collision check. Varying voxel grid resolution

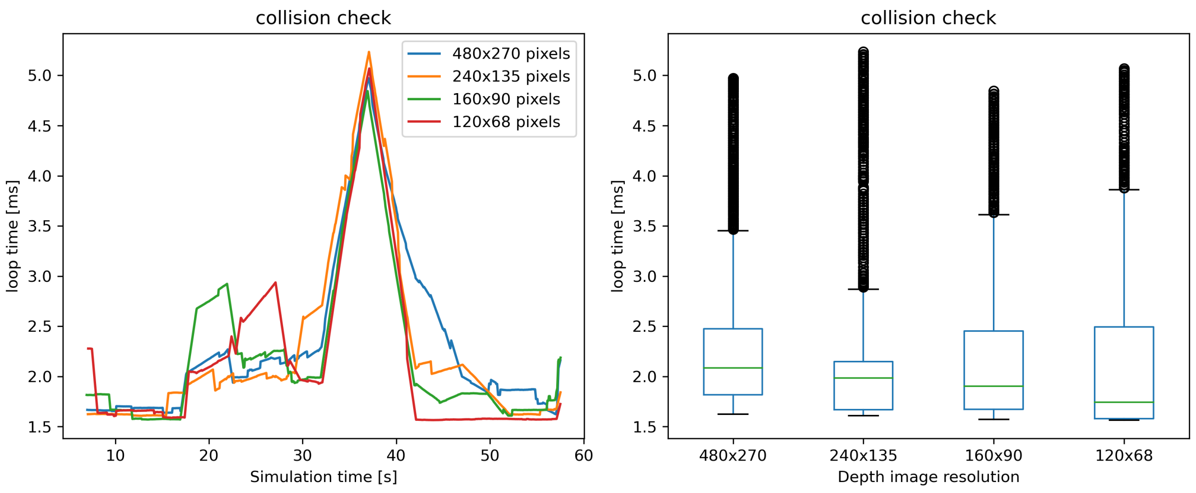


Figure D.9: Loop durations sub-processes EGO-planner collision check. Varying depth image resolution

No significant increase or decrease is visible in duration for the varied voxel grid resolution and depth image resolution. After some investigation of the code, the collision check was found to only check collisions for the control points, which are points spread over the line at a defined distance. Therefore, the hypothesis is that the collision check would only vary significantly when the number of control points is varied. This parameter is outside the scope of this research but might be interesting for future research.

D.5. Simulation visual comparison performance

Although the path and obstacles are prerecorded in the dataset, the behaviour of the mapping and planner are different over the different parameter variations. The setup with a dataset for simulation gives the opportunity to improve the desired responses of the path planner in certain pre-recorded situations.

In figure D.10 below, the RVIZ simulation is shown with the behaviour of two different configurations.
The green line shows the generated trajectory for a configuration in which the local planner range was 5 meters in the xy direction. This is marked in the figure by the outer dotted square.
The blue/red line shows the generated trajectory for a configuration in which the local planner range was 3 meters. This is marked in the figure by the inner dotted square.
The coloured blocks are the mapped obstacles for the simulation of the red/blue line with the smaller local planner range. The grey blocks are all mapped obstacles that exist in the simulation. The coloured and grey blocks within the outer dotted square are used for the simulation with the green line. That simulation might have mapped more obstacles than shown in the figure, as the blocks in the screenshot belong to the other simulation.

This screenshot displays two simulations overlaid on each other. It shows that in this particular example, the planner with the longer range plans around obstacles farther ahead, resulting in increased efficiency for the drone in terms of speed and energy consumption. The simulation with the blue and red lines would also successfully avoid the obstacles, but it requires the drone to make more deceleration and acceleration manoeuvres. It is worth noting that this image is only showing one frame, and in reality, the blue/red line simulation would evade the obstacles approximately two seconds ahead of what is displayed.

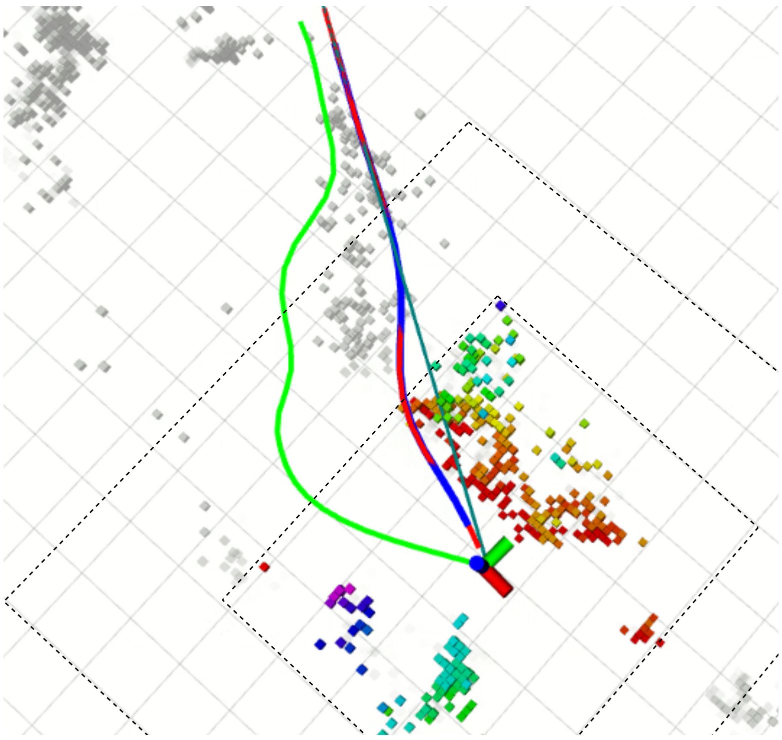


Figure D.10: Two simulations overlaid in Rviz with different local-planner-range configurations

The figure above was created by first running the simulation with the longer range and recording the output of the generated trajectories. Thereafter, the simulation with the shorter planner range was run, and the Rosbag file with the previously generated and recorded trajectories was replayed at the same time. When plotting the trajectories in different colours, the performance can be compared in Rviz.

E

Further considerations

During the project, some ideas and thoughts came up that were not followed up as they were outside the project's scope, mostly due to time constraints. This appendix briefly summarizes these ideas and thoughts for future inspiration.

E.1. EGO-planner and drone improvement for use in the wild

First, the ideas and thoughts on improving the drone and ego-planner for use in the wild are listed in this section. This section covers centralising the local planner, the ground control station, adaptive reactive and planning-based navigation, path backtracking, drone and sensor choice, the uncertainty of unseen areas, and SLAM by Raycasting.

E.1.1. Centeralise obstacle map and planner around the drone

When launching the EGO planner, a voxel grid world map with a defined size is created to map obstacles. From the world map, a small section around the drone's position is used to map local obstacles and plan trajectories around these local obstacles. The obstacles stored in the world map are not meant to be used by the local planner at a later moment when the drone returns to the same location because the ego planner then resets and remaps these obstacles. There is some global planner incorporated in the ego planner, which uses the complete world map, but this does not seem fully functional.

The world map is a big 3D matrix stored in RAM memory, like the other variables in C++. At a high-resolution voxel grid, a bigger matrix is required to define the same 3D volume. The available RAM memory on the processor board limits the voxel grid resolution and size of the world map in x,y, and z directions. Therefore, the ego-planner cannot freely explore because it will stop working when it reaches the edge of the map.

For a processor board like the Odroid XU4 (2gb RAM) and the PiZero2w (0.5gb RAM), at a 0.1m voxelgrid resolution, the map limits to 250x250x5m and respectively 125x125x5m. At startup, the drone initialises in the middle of this map and at zero altitude. So, after flying 125m, respecably 62m, in one direction, the drone reached the border. Increasing the RAM with Swap memory (using an SD card to dump memory overflow) does not work well. This is tested, but it seems to crash the ego-planner. Possibly because every variable in Swap first goes back into RAM before it is used by the system. If the single matrix does not fit in the entire RAM, it might, therefore, crash the system.

To solve this problem, the ego-planner code can possibly be rewritten to only use a local planner with the drone always centred in this local map. To achieve this, first, the local obstacle map has to shift the currently mapped obstacles regarding the drone's change of position. Secondly, the planner's trajectories must be transformed to the drone's position shift for the rebound and collision check functions.

When a world map is still required for some additional purpose, this can be stored as sectors in different smaller files stored on the SD card. Then, when the drone flies into another sector, the current sector of the world map is saved to the SD card and another sector is loaded.

A workaround solution for the limited map problem would be to re-initialise the ego-planner at the centre of the map every time the drone reaches the limit of the map. The drone can then either remap all obstacles from scratch or, more nicely, the latest local map of obstacles is copied towards the centre of the map where the drone is relocated.

E.1.2. Adaptive navigation able to switch between reactive navigation and planning-based navigation

Although most drones can fly at speeds over 10m/s, autonomous drones with planning-based navigation tend to fly at 1m/s to max 4m/s. The most important reason is the reaction time of the planner when an obstacle occurs within the sensor range. The sensing range of the mostly used depth camera (Realsense D435) is 6m. When a drone is flying at 10m/s, and an obstacle occurs in the camera frame 6m ahead, the drone has about 0.6 seconds to compute a collision, generate a response (break), and decelerate to a complete standstill. In this example, the reaction time would be too short, and the drone would most likely crash.

But, if the drone had a long-range sensor (10-20m), like a single-array TOF sensor or ultrasound sensor, the obstacle could be detected earlier. When flying in open space, the drone could switch to reactive-based navigation, enable the long-range sensor, and fly at higher speeds while still ensuring safe flight. In addition, the depth camera can be a smaller, more lightweight sensor with a shorter detection range. The drone would then still be able to solve complex scenarios with the ego-planner but at a bit lower speeds, and it can fly fast and effectively in open space with reactive-based navigation.

E.1.3. path backtracking and adaptive margins

To find a route back home in the rainforest, taking the takeoff position as a goal and trying to navigate back from the current position is a significant risk. When the drone cannot find a path before the battery dies, it will crash in the rainforest and likely be lost.

Following the same path back as it came seems to be a safe solution in which the drone knows exactly the distance to fly back home. Although, after considering many ideas to solve this during the project, this seems to be the most simple and robust solution, it comes with some challenges.

To apply this, firstly, the position estimate of the drone drifts over time, as is explained in the report. So, the drone thinks this path is elsewhere than where it has flown. For this problem, SLAM, preferably with loop closure, seems to be the most obvious solution. But this requires about 2 times the processing power as what the complete EGO-planner uses.

Secondly, as the drone still needs safe obstacle avoidance, the EGO-planner should keep running when it flies back. So the ego-planner has to be adapted to imitate the flown path. One possible solution can be to incorporate this in the cost function of the planner, where the weight is zero when the drone is exploring, and the weight is high when the drone has to return home.

E.1.4. unsafe to fly around corners into unseen areas

In contrast to the FASTER-planner [17], the EGO-planner [18] does not take into account whether free voxel-blocks in the mapping are free because there is no obstacle or free because the obstacle has not been mapped yet. When flying around a corner into new unexplored space, there is a risk that obstacles occur just around the corner, and the drone has little response time to stop before it crashes.

This functionality could be implemented into the EGO planner in 2 steps. The first step is initialising voxels as unknown, and marking voxels as free-space when they are crossed during the raycasting process. In the EGO planner, the probability of the voxels in the map being an obstacle is now a value between 0.00 and 1.00. The probabilities could be initialised at -1 and set to 0 or higher when they are first raycasted or marked for obstacles to store the information for unknown voxels in the same matrix. The second step is actively using this information in the path planner. For example, reducing the speed at the trajectory it plans in unknown voxels. The same function used for the collision check could be used to check if the generated trajectory is in an unknown space.

E.1.5. use raycasting to estimate/correct the position drift

In this project's research, SLAM was found to be too computationally intensive to run on the small processor implemented in the drone. Visual-inertial SLAM was found to be the most lightweight but still showed 2 times more processor load than the path planner itself. Using the obstacle map and the point cloud from the camera for SLAM is also a robust method, but it uses significantly more CPU resources. This method checks how all points from the camera's point cloud match best with the mapped obstacles by a function named iterative-closest-point.

During testing, it was found that when the voxel grid resolution is high enough, a significant drift in a short time is visible in the raycasting and mapping. A flow of obstacles is visible where some obstacles disappear on one side and grow on the other side. Instead of using the iterative-closest-point, the flow in the mapping might be detectable in the same way that visual SLAM uses optic flow to see how a scene changes. This thought is not extensively investigated, and it might already be applied by someone.

E.2. Downscaling research

During the downscaling research in chapter 6, some thoughts arose. This section shares thoughts on the possibilities of the simulation method, using different hardware, and the processor tasking management.

E.2.1. Possibilities with the used downscaling method and analysis

In the downscaling research, a method was created in which data is first recorded in a forest, filtered, and a dataset is created. This dataset is then processed by the ego-planner using a specific setup, and the planner's responses are recorded. The recorded data is then filtered and translated into Python data. The entire process of feeding the dataset into the ego-planner with different configurations and generating Python data is fully automated. With this automation, it is possible to run 100 simulations with varying configurations in just a few hours. All the data obtained can be plotted and compared in a Python script for analysis. Moreover, if a particular simulation yields remarkable results, the simulation's ROSbag file can be replayed to determine the cause of such results.//
This method works efficiently to test many variations of algorithm configurations on a specific scenario (the dataset). This method has the benefit of simulating a computer-simulated environment in that it represents the real world best, as the sensor input is literally from the real world. And, the same as with a computer-simulated environment, there is no damage by crashing if one configuration does not work well. Another benefit is that two simulations with different results can be overlaid, for example, in Rviz, to see the difference and learn the impacts of varying the planner's configurations. When flying the drone in the real world, comparing performance for small changes in the configuration is more difficult. For this method, the dataset can already be created before the implementation of the planner works. It only has to record the sensor data from the drone and depth camera, while the drone can be manually piloted.

E.2.2. Downscaling on different hardware

In the shown downscaling results, the performance and CPU load reductions are shown to apply the planning-based navigation on a smaller system. However, it's important to note that changing the hardware from the used Odroid XU4 processor to a different processor, such as the Raspberry Pi Zero2w, might result in different scaling of performance and load. This relationship may look slightly different when using different hardware configurations, such as 32-bit vs. 64-bit, different RAM, CPU cores, or Linux OS.

E.2.3. Processor tasking management

When the EGO-planner process is initiated according to the algorithm's design, the C++ script runs as a single task on a single core of the CPU. This could be problematic for a smaller system, as it may not be able to utilize all available computational resources. Although it may seem like a safe choice to reserve the remaining cores for other ROS and non-ROS processes, running the EGO planner on only one core limits its performance. The algorithm could benefit from running some computations in parallel, which would improve its efficiency.

F Downscaling applied

These days, people see more and more applications for drones, including monitoring rainforests to protect plant and animal species. However, drones face challenges when navigating through the dense and cluttered vegetation of the forest. These environments necessitate advanced autonomous detection and navigation to make the drone traverse robustly and fly safely. In addition, the forest brings extra challenges, such as blocked signals for GPS localisation, remote control, and remote supervising.

In this thesis project, a drone is designed, built, and programmed to navigate autonomously in the rainforest with complete onboard computing and no GPS localisation. This 500-gram drone is being extensively tested and optimized in real forest conditions, and a dataset is being created from its autonomous flights to simulate various configurations of the path-planning algorithm. The results of these simulations on this dataset are then used for thorough research on how the algorithm can downscale to smaller systems and how this affects performance.

By using the results of this research on downscaling, a 100-gram drone is built and programmed to fly in forest conditions with complete onboard computation. Challenging on this small-size drone is the use of low-quality lightweight sensors and processor. The processor only weighs 10 grams, and the depth camera weighs 8 grams. Unique on this small drone is the 3D path planning fully computed onboard and the implementation of a new type of depth camera.

This appendix covers the design for the system onboard the 100-gram drone, which enables it to fly autonomously. This design covers the hardware and software. Thereafter, the results show that this 100-gram drone is actually flying.

F.1. Module design

To make autonomy on a small drone, components have to be small. Different components are chosen for this design compared to the autonomous 500-gram drone. This section covers the chosen components and the 3D integrated design thereafter.

F.1.1. Components

This section covers the components chosen for the lightweight autonomy system, which are the processor board, depth camera, drone, power regulator and RC receiver.

PiZero2w

To calculate the required autonomy on a small-scale drone, the lightweight and cheap Raspberry Pi Zero 2w [30] is chosen. This board weighs 10 grams and costs 15 euros. The board has only 0.5 GB RAM and a *1GHz quad-core, 64-bit ARM Cortex-A53 CPU* processor. Therefore, this board can show the true potential of the downscaling applied as this is about 1/4 of the RAM and 1/2 of the processor power

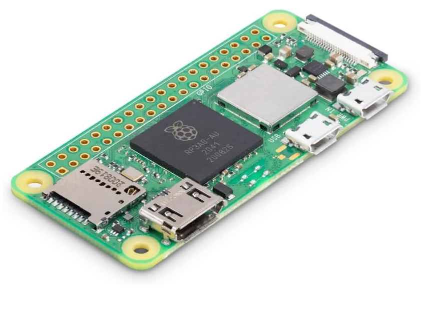


Figure F.1: Raspbery Pi Zero2w [30]

compared to the Odroid XU4 processor used on the 500-gram drone in this project. And an even bigger gap compared to the Jetson xavier nx processor used in the original EGO-planner paper [18]. Figure F.1 shows the Pi zero 2w.

TOF camera

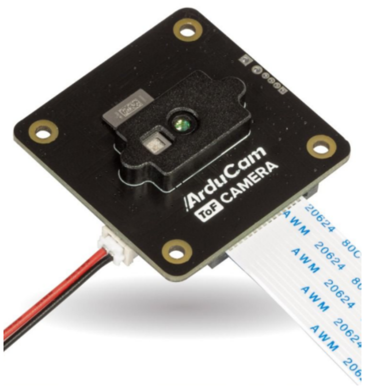


Figure F.2: Arducam TOF depth camera [31]

To detect obstacles, a lightweight sensor is required for the drone to be able to lift the payload of the autonomy system. As the the 60 grams of the Realsense D430 used on the 500gram drone would be too heavy, another approach is tested. A new camera model named Arducam TOF camera is tested. This camera weighs only 8 grams, has a small formfactor, and costs 60 euros. In contrast to the heavy Realsense camera, this TOF camera can only measure 2 to 4m depth.

After emitting an infrared flash, the camera measures for 240×180 pixels the time for the flash to return. From the time of flight (TOF), the camera calculates the depth per pixel. This method has been used in the industry before but has never been seen in this small form factor before with comparable image resolution and weight. No literature is found where this camera is used for navigation on a drone. Figure F.2 shows the TOF camera.

Tello drone

The DJI Tello drone is chosen to showcase the small and lightweight autonomy system on a small drone. This drone is 10x10x5cm, weighs 80 grams, and costs 100 euros. The main reason this drone was chosen was its availability and because it was controllable via an SDK. In addition, the drone has out-of-the-box stable position control by using its bottom and forward camera for optic flow.

The DJI Tello drone also powers the camera and the processor. To accommodate this, a positive and ground wire is soldered to the drone's battery connector inside. As Tello's single-cell lithium battery provides around 3.8 volts, a step-up power regulator is used to convert the battery voltage to the required 5V. Figure F.3 shows the small Tello drone.

Figure F.3: DJI Tello drone [32]

Figure F.4: DC step-up converter 5V 2A [33]

power regulator

As a step-up power regulator to supply 5V and max 2A is chosen to convert the variable +- 3.8V of the drones' battery to a stable 5V for the electronics. The regulator board is 20x10x5mm and weighs around 4 grams. Figure F.4 shows the regulator board.

FrSky receiver

To receive remote controller commands, the same receiver is used as on the 500-gram drone in this project. This receiver is the FrSky R-XSR model, which is small and weighs 2.4 grams.

F.1.2. 3D integrated design

To build and mount the autonomy system compactly and lightweight on the Tello drone, a 3D design is made. This model contains all the components mentioned above, a mount to attach the processor board to the drone, and a mount to attach the depth camera to the processor board. Figure F.5 below shows a screenshot from the Solidworks model in which all components are integrated.

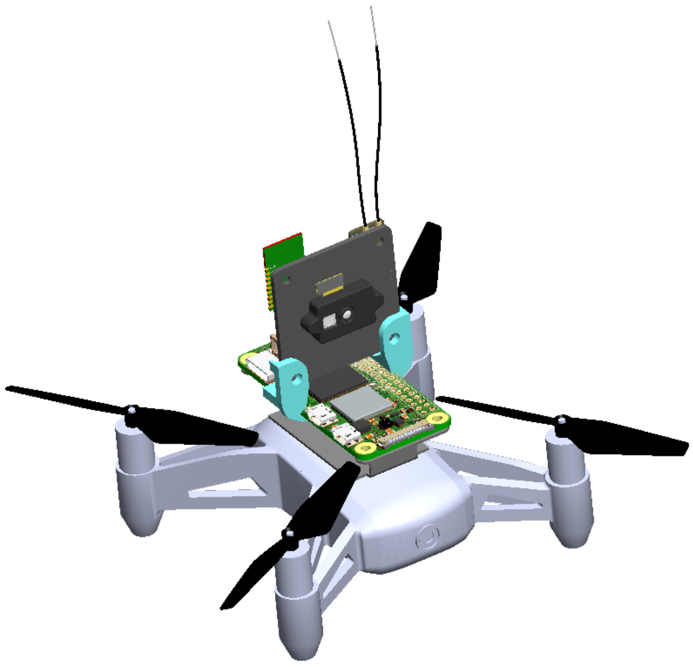


Figure F.5: Solidworks 3D model of Tello drone with autonomy system

In the integrated design, the mount for the processor board on the drone sticks to the drone's top cover with double sided tape. This 3D printed mount is printed as a flat sheet and bend into shape to achieve better strength with the 3D printed layer directions. Holes in this mount are tapped to M3 and the board is attached with plastic M3 bolts.

The depth camera is clamped to the processor board edges with 3D prints. These clamps, in light-blue in the figure, are designed such that they clamp tighter when the bolts on the camera are fastened.

F.2. Software adaptations and configuration

To make the software, initially designed for the 500-gram drone, work on the smaller processor board, some adaptations were required. In this section, these adaptations are briefly mentioned.

ROS driver for Tello drone

To control the Tello drone via the ROS architecture, a ROS node is required to communicate with the drone's SDK. This ROS node has to send control commands to the drone received from other nodes and share telemetry data with other nodes. Multiple Tello drivers (node package) were open-source available, but none had the exact set of features required for this project. Ideally, the Tello driver has input and output data identical to the Bebop drone used in this project (the 500-gram drone). Therefore, a Tello driver for ROS was made from scratch using the DJI-TelloPy Python library. This ROS node communicates with data similar to the Bebop driver.

Remapping the ROS network

To make the Tello drone work in the same ROS architecture structure as used with the Bebop drone, some connections between topics and nodes have to be remapped. All nodes connected to Bebop-related topics have to be connected to Tello-related topics. Further, some processes are removed from the system such as a GPS driver and a waypoint mission planning node.

Configurations for path-planning
To make the path planning run on the small processor board, configurations are changed based on the downscaling research. For the tests with the 100-gram drone, the configurations are set as follows:

- Voxelgrid resolution = 0.1m
- Local planner range = 4.0m
- Depth image resolution = 120x90 pixels
- Depth image rate = 10 Hz

F.3. Results

With some tests performed indoors and outdoors, the Tello drone appeared to perform decently with the added payload for the autonomy system. This section gives a comparison of flight times with this payload. The ROS system functioned as expected, similar to the system of the 500-gram drone. The camera used in this drone has a shorter detection range, and the configuration is downscaled, resulting in lower performance in obstacle detection and route planning compared to the 500-gram drone. This section shows the most relevant performance. Lastly, this section shows some difficulties with the high processor load during testing.

Flight times
Having lightweight autonomy on a 100-gram drone might have some applications, but the biggest advantage is expected to have a bigger autonomous drone with prolonged flight time. With this in mind, the flight time of the Tello drone and Bebop drone are compared with the original autonomy payload described in this report, the newly introduced autonomy payload described in this appendix, and no payload. Table F.1 shows the measured flight times for both drones. For the Tello drone, the original autonomy system could not be tested because the drone cannot lift this weight at all. The flight times shown are rounded to whole minutes, except for the duration below 5 minutes. Also, the flight times are measured with 100% down to 10% battery. The last 10% of battery is not used as lipo and lithium batteries can damage by deep discharge.
Although the flight time of the Tello drone reduced significantly to about 1/3rd with the new lightweight autonomy, the same payload attached to the Bebop reduced the flight time by about 15% compared to no payload.

Table F.1: Measured flight times for Bebop and Tello drone with different payloads for autonomy

	Payload	**Bebop drone**	**Tello drone**
With Autonomy backpack	202 gram	11 minutes	-
With downscaled autonomy	29 gram	21 minutes	3.5 minutes
Without payload	-	25 minutes	11 minutes

Autonomous navigation performance
The configurations applied to the EGO planner can be used to derive the performance as described by the performance metrics of the downscaling research.
By applying formula 6.1 from the downscaling research, the theoretical minimum detectable obstacle size can be determined. Filling in: field-of-view=65 (horizontal), camera-range=4.0m, original-resolution=240 (horizontal), skip-pixels=2, gives a minimum object size of 4.2cm that can theoretically not be missed.

By applying formula 6.4 from the downscaling research, the maximum solvable obstacle width can be determined. Filling in inflation=0.1m, voxel-grid-resulution=0.1m, and local-planner-range=4.0m gives a maximum solvable obstacle width of 3.8m.

By applying formula 6.5 from the downscaling research, the minimum (guaranteed) solvable gap width can be determined. Filling in the same values as mentioned above, the minimum solvable gap width is 0.75m.

The reaction time cannot be estimated at this point because logging loop rates was difficult with this setup. This is described below in this section.
Screenshots of the planner in action during a test flight in the Cyberzoo are shown at the end of this appendix.

High processor load
The processor board runs at a high load with the chosen configuration. The system tends to crash when running another process in parallel, such as data logging or data streaming over WiFi. Therefore, logging data or live-streaming the planner visualisation is difficult without further downscaling the system's performance.

Testflight screenshots
In figure F.6 below, the test environment used in the Cyberzoo for the results that follow is shown.

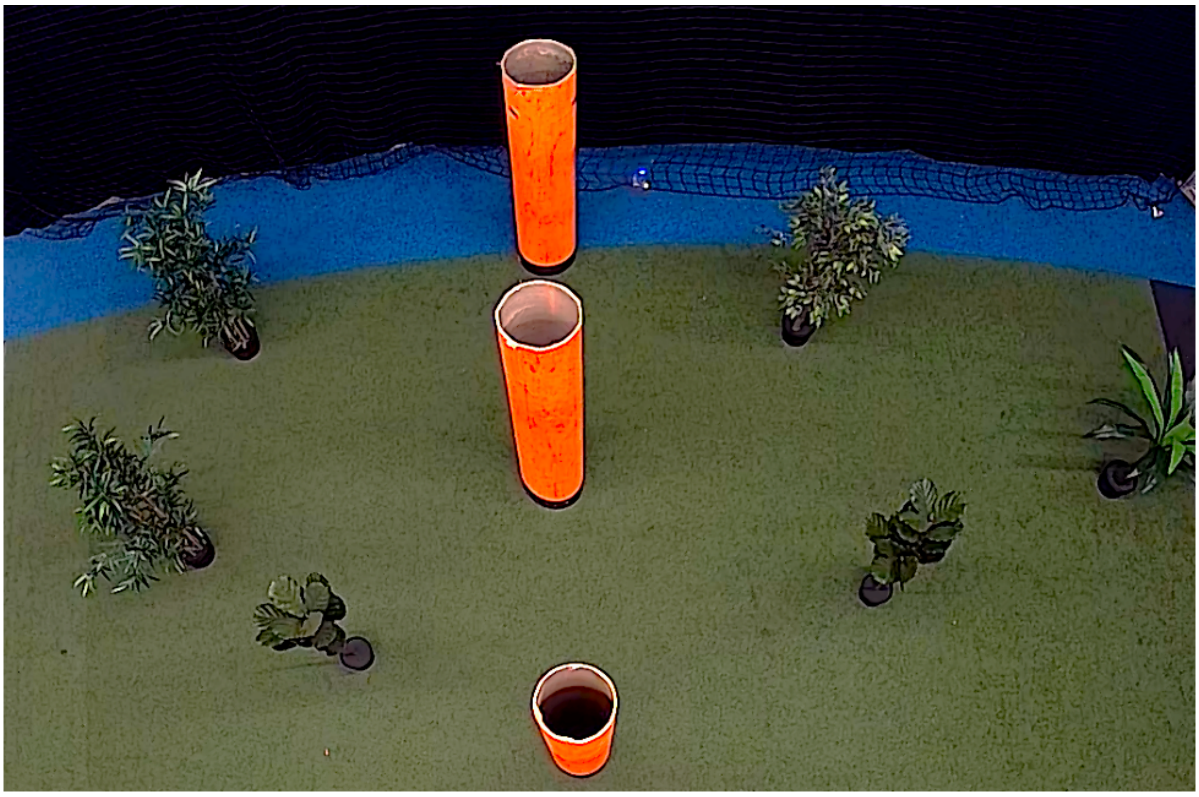


Figure F.6: Topview of test environment with Tello drone

Figure F.7 below shows a screenshot in Rviz. On the left, the depth image is shown for that time instance. In this depth image, a plant is visible nearby on the left, and two poles are visible more in the back. On the right in the figure, the mapped obstacles are shown and the trajectories the drone is planning. Blue and red are sampled paths that change often until a valid path is found. Blue is the initial path, the collision-free path, and red is the smoothed path. But at this time instance, the sampled path is not yet a final path. The green path is the last calculated valid path that the drone is following.

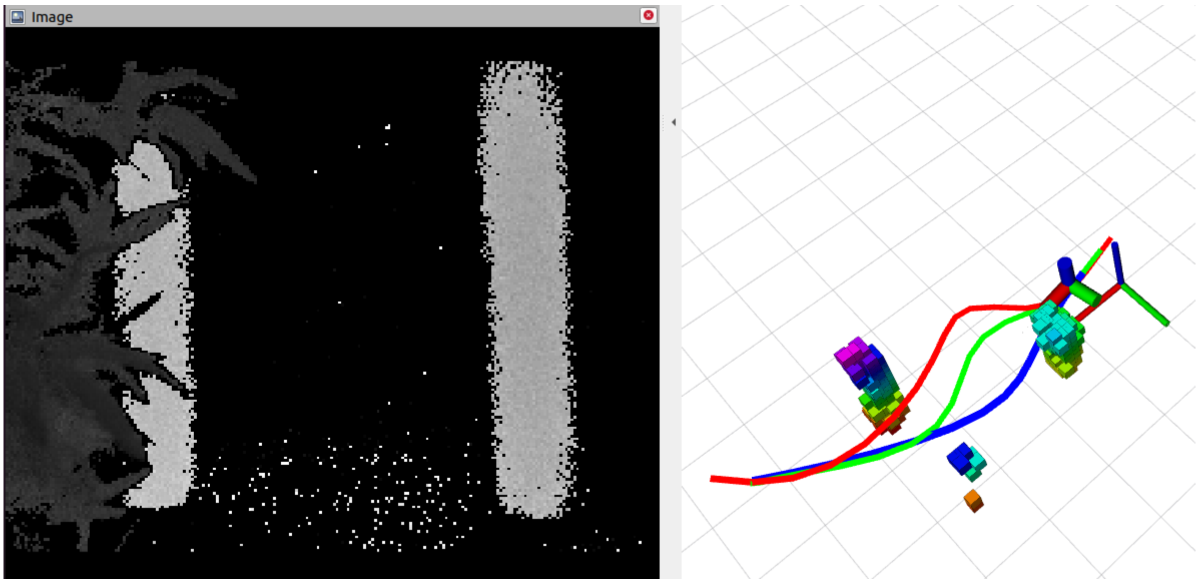


Figure F.7: Rviz with depth image, obstacle mapping, and planned trajectories for Tello drone

In figure F.8 below, a screenshot in Rviz is shown from the same flight but at another time instance. In this screenshot, a plant is visible in the depth image. And, the planner has planned a path over this plant, shown on the right. The mapped obstacles on the right also include two poles, which the drone just passed, no longer visible to the camera.

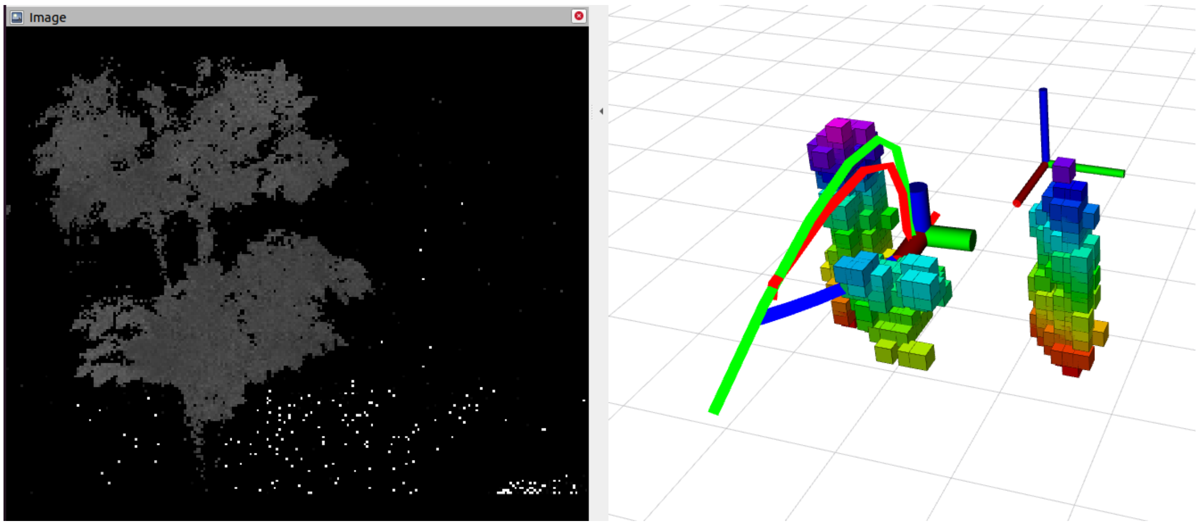


Figure F.8: Rviz screenshot. Tello drone plans a path over a plant.