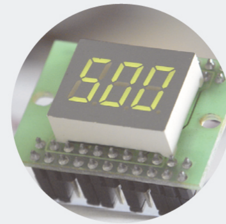
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

isep



Behavior Tree UAV Mission Control in Warehouse Logistics

ANDRÉ FILIPE OLIVEIRA MOURA

novembro de 2021

POLITÉCNICO
DO PORTO

Politécnico do Porto
Instituto Superior de Engenharia do Porto

Behavior Tree UAV Mission Control in Warehouse Logistics

André Filipe Oliveira Moura

Master in Electrical and Computer Engineering
Specialization Area of Autonomous Systems

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

November, 2021

This dissertation partially satisfies the requirements of the Thesis/Dissertation course of the program Master in Electrical and Computer Engineering, Specialization Area of Autonomous Systems.

Candidate: André Filipe Oliveira Moura, No. 1161150, `1161150@isep.ipp.pt`

Scientific Guidance: André Miguel Pinheiro Dias, `apd@isep.ipp.pt`

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

November, 2021

Acknowledgements

Throughout the development and writing of this thesis, I have received a great deal of support and assistance.

First of all, I am sincerely thankful to my thesis advisor Dr. André Dias for guiding my research work and for the many opportunities that he has presented to me that contributed to extending my knowledge and experience in the fields of autonomous systems and robotics.

I would also like to thank my colleague José Antunes that worked with me throughout my master's degree and with whom I shared most of my research in the subject of this thesis. His knowledge and practicality were always essential to our multiple combined projects.

Moreover, I would like to thank my colleagues José Oliveira and David Carvalho, with whom I worked at ISEP's robotics group (NRISEP) and throughout my master's degree. The experiences we shared while working for and competing in robotics competitions were really important to solidify my interest in this field and are some of my best memories from my time at this school.

Finally, my sincere thanks to my family, especially my mom, for supporting me and allowing me to pursue this degree, and to Mariana, for giving me advice and motivation to pursue my objectives.

Abstract

The Industry 4.0 revolution requires the automation of more industrial processes. Particularly in warehouse logistics, Unmanned Aerial Vehicles (UAVs) are a key technology for the digital transformation of warehouse management tasks, providing the ability to perform automatic real-time inventory counting, localize hard-to-find items, and reach narrow storage areas. The use of this type of robot poses new challenges, such as UAV indoor localization, navigation, collision avoidance, and fleet management. This thesis proposes an indoor navigation system for a small commercial UAV, with a visual-inertial Graph-SLAM approach, and a simple way for executing user-defined behavior tree missions through the Aerostack framework, in order to achieve an easy, fast, and cost-efficient implementation. A system architecture was presented, with the development of Robot Operating System (ROS) interfaces for the multiple hardware and software elements, and the implementation of new behaviors related to inventory management tasks. The system was tested in an indoor environment, where the executed mission allowed the UAV to take off, navigate to two marked locations, take photos of the markers, and return to the take-off location. The system performance was evaluated by comparing sensor data and the vehicle behavior during the mission with the expected behaviors, according to the mission plan.

Keywords: UAV, indoor navigation, industry 4.0, behavior trees, mission execution, warehouse, logistics, inventory management, ROS, Aerostack.

Resumo

A revolução da Indústria 4.0 requer a automação dos processos industriais. Particularmente na logística de armazéns, *Unmanned Aerial Vehicles* (UAVs) são uma tecnologia chave para a transformação digital das tarefas de gestão de armazéns, possibilitando a realização de contagem automática de inventário em tempo real, a localização de itens de difícil acesso e a navegação em áreas de armazenamento estreitas. O uso deste tipo de robô apresenta novos desafios, como a localização e navegação indoor de UAVs, prevenção de colisões e gestão de frotas. Esta tese propõe um sistema de navegação visual e inercial em ambientes *indoor* para um UAV comercial de pequenas dimensões, utilizando Graph-SLAM, e através de uma maneira simples de executar missões definidas em *behavior trees* pelo utilizador através da *framework* Aerostack, de forma a alcançar uma implementação fácil, rápida e acessível. Foi apresentada uma arquitetura de sistema, com o desenvolvimento de interfaces Robot Operating System (ROS) para os vários elementos de *hardware* e *software*, e a implementação de novos comportamentos relacionados com tarefas de gestão de inventário. O sistema foi testado num ambiente *indoor*, tendo a missão executada permitido ao UAV descolar, navegar até dois locais marcados, tirar fotos dos marcadores e retornar ao local de descolagem. O sistema foi avaliado comparando os dados dos sensores e o comportamento do veículo durante a missão com os comportamentos esperados, de acordo com os *behaviors* definidos no plano da missão.

Palavras-Chave: UAV, navegação indoor, indústria 4.0, behavior trees, execução de missões, armazéns, logística, gestão de inventário, ROS, Aerostack.

Contents

List of Figures **vii**

List of Tables **ix**

Listings **xi**

List of Acronyms **xiii**

1 Introduction **1**

1.1 Problem Statement 2

1.1.1 Objectives 3

1.2 Thesis Outline 4

2 Related Work **5**

2.1 UAV Navigation 5

2.2 Autonomous UAVs in Inventory Management 6

2.3 Mission and Task Planning 7

2.3.1 The State Machine Approach 8

2.3.2 The Behaviour Tree Approach 8

3 Fundamentals **11**

3.1 ROS 11

3.2 Behavior Trees 13

3.3 Aerostack 14

3.3.1 Architecture 14

3.3.2 Behaviors 15

3.3.3 Mission Planning and Execution 16

3.4 PID Controllers 17

4 Project **19**

4.1 UAV Platform and Interface 19

4.2 Localization System 20

4.3 Mission Planner and Navigation 21

4.4 Proposed Architecture 22

5 Implementation **27**
5.1 ROS Interface with DJI Mobile SDK 27
5.2 Navigation System 34
5.2.1 Image Tranformation 34
Image Decompression 34
Image Rectification 35
5.2.2 Hardware Interfaces 35
State Interface 35
Command Interface 39
5.2.3 Aerostack Framework Implementation for Mission Planning and Execution 43
Keyboard Teleoperation 43
Behavior Catalog 45
Behavior Tree Mission Editor 47
5.3 Test Conditions and Mission Plan 48

6 Results **51**

7 Conclusion **61**
7.1 Future Work 62

References **63**

List of Figures

1.1 High-level architecture of the system. 3

3.1 Representation of a typical behavior tree model. 13
3.2 The Aerostack reference architecture (Version 4.0). 15
3.3 Block diagram of a PID controller. 18

4.1 Structure of the localization system. 21
4.2 Proposed architecture of the navigation system. 25

5.1 First page of the ROS interface Android app. 28
5.2 Second page of the ROS interface Android app. 28
5.3 Third page of the ROS interface Android app. 29
5.4 Comparison between the user interfaces for the original version of the keyboard teleoperation package and the modified version used in the new implementation. 45
5.5 Usage of the two developed behaviors in the Aerostack behavior tree editor. 47
5.6 3D point cloud of the test environment in RViz. 48
5.7 Complete behavior tree of the mission plan. 49
5.8 Trajectory defined in the behavior tree mission plan. 50

6.1 UAV at the beginning of mission execution. 52
6.2 UAV after the take-off behavior execution. 53
6.3 UAV capturing the photo of the first board. 54
6.4 UAV capturing the photo of the second board. 55
6.5 UAV beginning the landing behavior execution. 56
6.6 UAV at the end of mission execution. 57
6.7 Comparison between the localization algorithm's optimized trajectory and the defined mission path. 58
6.8 Main camera gimbal pitch during mission execution. 59
6.9 Vehicle yaw during mission execution. 59
6.10 Photos captured during mission execution for the two boards representing inventory racks. 60

List of Tables

3.1 Aerostack behaviors used in the UAV indoor warehouse navigation system missions. 16

5.1 ROS topics published by the interface. 29
5.2 Mavic_Commands and correspondent Response messages. 32
5.3 Vehicle flight states. 36

Listings

5.1 Code for publishing vehicle velocity data using threads. 30
5.2 onSurfaceTextureUpdated function, responsible for receiving the main camera image in the SurfaceTexture variable type and converting it into a byte array. 31
5.3 Thread responsible for generating and publishing the messages in the Video_Feed ROS topic. 31
5.4 Vehicle commands subscriber and corresponding callback function. . 33
5.5 Parse_Command function associated with translating the string command messages present in the Mavic_Command ROS topic to vehicle commands, using DJI Mobile SDK functions. 33
5.6 Code associated with sending three-dimensional linear velocity and yaw angular velocity commands to the UAV and handling of the command responses sent back to the ground-station computer. 34
5.7 Function responsible for publishing the camera/ image_raw and camera/camera_info ROS topics. 35
5.8 Setup function of the mavic_state interface. 37
5.9 Code associated with determining the vehicle flight state in the mavic_state interface. 37
5.10 ROS publishers and subscribers in the mavic_command interface. . 40
5.11 Main function of the mavic_command interface. 41
5.12 PID controller for yaw angle control in the mavic_command interface. 42
5.13 Function responsible for publishing the velocity commands in the mavic_command interface. 44

List of Acronyms

AGV	Automated Guided Vehicle
BT	Behavior Tree
CPS	Cyber-Physical System
EKF	Extended Kalman Filter
FSM	Finite-State Machine
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GTSAM	Georgia Tech Smoothing and Mapping
GUI	Graphical User Interface
HSM	Hierarchical State Machine
IMU	Inertial Measurement Unit
IoT	Internet of Things
LiDAR	Light Detection and Ranging
P2P	Peer-to-Peer
PID	Proportional-Integral-Derivative
QR	Quick Response
RFID	Radio-frequency Identification
ROS	Robot Operating System
RPC	Remote Procedure Call
SBC	Single-Board Computer
SDK	Software Development Kit
SLAM	Simultaneous Localization and Mapping
UAS	Unmanned Aerial System
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
UI	User Interface
USB	Universal Serial Bus
UWB	Ultra-wideband

Chapter 1

Introduction

Unmanned Aerial Vehicles (UAVs) are an increasingly more popular technology that has its utility associated with various types of applications, such as search and rescue, surveillance, and remote inspection. This type of vehicle stands out for its ability to access and operate in dangerous and isolated environments, with technology that has been evolving at a high rate, with reduced costs, and capability for partial or even full autonomy.

Additionally, the transition to smarter factories, associated with the industry 4.0 perspective and the undergoing change in industrial automation based on concepts as Internet of Things (IoT), Cyber-Physical Systems (CPSs), and Big Data, introduces integration opportunities for emerging technologies. In this particular context, UAVs have the potential to carry out dangerous and repetitive factory tasks with little human interaction, which favors both work safety and efficiency [1].

One of the activities that can benefit from UAV capabilities is warehouse management. Warehouses and distribution centers usually rely on storing inventory in pallet racks along aisles that workers and warehouse machines can access. In this scenario, shallow racks and narrow aisles favor cost efficiency and practicality, prioritizing space utilization and quick inventory access.

This type of environment is compatible with the characteristics and capabilities of modern UAVs, which can perform tasks in inventory management, logistics, inspection, and surveillance. In warehouse environments, UAVs are a useful tool to efficiently complete these tasks, and their remote operation with partial or full autonomy can prevent the danger and repetitiveness associated with manual tasks,

overall improving workers' conditions. In particular, inventory management applications have gained a lot of interest from the industry for their potential and there are some industrial implementations that passed testing phase [2].

This thesis addresses the application of commercial UAVs in inventory management tasks for an easy, fast, and cost-efficient implementation across multiple warehouses.

1.1 Problem Statement

Due to the typical disposition of stock in tall inventory racks along aisles in warehouses and distribution centers, some inventory management tasks can be both repetitive and dangerous to industry workers. UAVs can be a useful tool in this context, as they can navigate through the aisles and have sensors that can be used to inspect inventory racks.

In the last few years, this has been the motivation for integrating UAVs into warehouse inventory management tasks. However, much of the work done with UAVs in this area is not scientifically documented and is based on proprietary solutions that are not engineered for widespread implementation. For these reasons, there is interest and opportunity in researching, documenting, and developing a UAV system suited for inventory management in an industrial warehouse environment.

Along with this opportunity, some requirements are necessary for ensuring an easy, fast, and cost-efficient implementation across multiple warehouses. These requirements are:

- The navigation system must be compatible with commercial-grade UAVs widely available on the market;
- The system cannot rely on a external map of the warehouse, because the layout of the warehouse can be changed;
- The navigation algorithms should run on a ground-station computer, also used for monitoring vehicle states, defining mission plans, and taking manual control of the UAVs if needed.

The navigation system must also include a localization system based on the implementation previously documented in [3], that was adapted and reimplemented in the context of another master's thesis [4]. Furthermore, this decision implies that the minimum requisites for the selected UAV commercial platforms are associated with the capabilities of providing a monocular image and inertial data, more specifically the vehicle velocity and attitude data.

The system high-level architecture is represented in Fig. 1.1. There are two main elements to this architecture: the UAV and the ground station computer.

The UAV system sends sensor inertial data and a monocular image to the ground station computer and in return receives commands from the ground station computer that allow to control its actuators. The ground station computer runs Ubuntu with Robot Operating System (ROS) and is responsible for the navigation system. This navigation system counts with Graph-SLAM algorithms based on the approach previously implemented in [3] that allow to estimate the vehicle pose. Additionally, the navigation system counts with a mission execution module that receives the optimized pose from the Graph-SLAM module and reads a mission plan, that is defined by a user and describes the mission in various distinct parts. The mission execution module is also responsible for all the commands not directly associated with the UAV navigation. Another module, the path planner, is responsible for defining the path for the vehicle during mission execution and manage the movement along that path by sending commands to the vehicle through a command interface, which should result in the desired trajectory for successfully completing the defined mission plan.

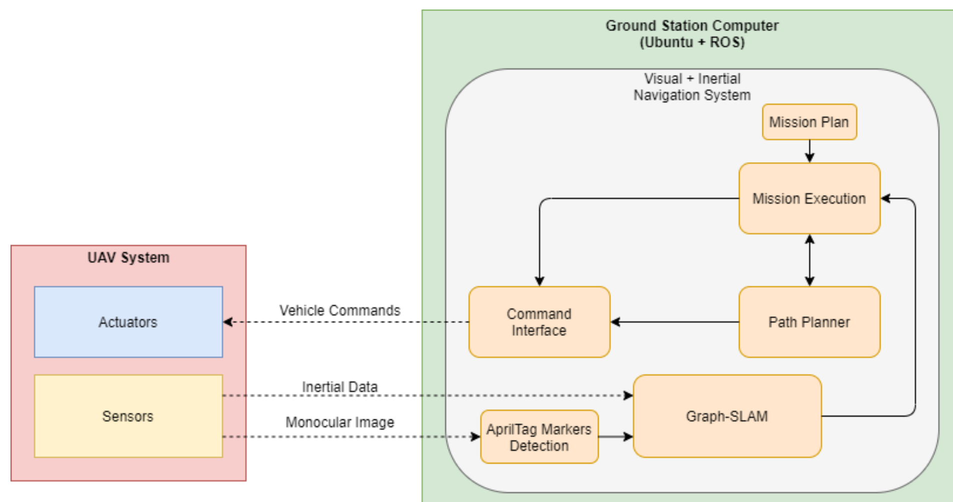


Figure 1.1: High-level architecture of the system.

1.1.1 Objectives

This thesis aims to introduce scientific contributions regarding the application of UAVs in warehouse inventory management tasks by developing a navigation system based on commercial-grade UAVs, suitable for autonomously performing inventory counting tasks, and compatible with the requirements for ensuring an easy, fast, and cost-efficient implementation across multiple warehouses.

This main objective can be divided into multiple smaller intermediate objectives that need to be fulfilled:

- Research and selection of algorithms for mission planning and execution suited for indoor navigation;
- Integration of the mission planning, mission execution, and Graph-SLAM algorithms in a navigation system;
- Development of interfaces between the navigation system and a commercial UAV system;
- Definition of a mission plan for testing the navigation system in the context of warehouse inventory management.

1.2 Thesis Outline

Chapter 2 of this thesis addresses the state of the art and work related with UAV navigation in indoor environments, inventory management, and mission and task planning.

Chapter 3 presents the fundamentals behind the work done in this thesis.

Following this, Chapter 4 describes the most important aspects of the project and the navigation system proposed architecture. Then, in Chapter 5 the details of the system implementation are described.

Chapter 6 presents the results obtained form testing the implemented system in a real-world scenario.

Finally, Chapter 7 addresses the conclusions obtained from testing the system and future research lines to improve on the work done.

Chapter 2

Related Work

The implementation of a UAV navigation system for inventory management tasks in indoor warehouses imposed research on the various types of UAV navigation, the application of these robots in inventory management, and mission and task planning approaches. This chapter gives an overview on some of the methods and developments associated with these topics.

2.1 UAV Navigation

UAV navigation refers to the process of planning a trajectory that allows the aerial vehicle to reach a certain target location, relying mostly on environment and vehicle state data. The three most common types of UAV navigation are satellite navigation, inertial navigation, and vision-based navigation [5].

Despite Global Navigation Satellite System (GNSS) sensors being available in various commercial UAVs, satellite navigation is not well suited for indoor navigation as it relies on communications that have low to no signal in closed environments, making them unreliable in this type of situation. Conversely, due to their wide availability with low costs and small size, Inertial Measurement Units (IMUs) are present in most commercial grade platforms and are well suited for indoor operation. However, these sensors accumulate errors over time, providing inaccurate measurements when used alone [6]. Regarding vision-based navigation, visual sensors and monocular cameras, in particular, are very common in UAV platforms because they

provide a large quantity of information about the surrounding environment at relatively low cost. Using monocular cameras as the main sensors for navigation is becoming widely researched and shows satisfactory and promising results, as Weiss *et al.* demonstrated in one of the first documented implementations in this category [7, 8].

These navigation methods are often combined to compensate for their individual disadvantages and for achieving better accuracy, being the combination of visual and inertial navigation well-suited for indoor environments, as industrial warehouses. For this combination, while IMUs provide inertial data at a higher sampling rate but accumulate drift error over time, positioning data obtained from vision sensors through image processing has lower sampling rates but generally provides measurements with no drift over time. These characteristics make the fusion of vision and inertial sensors' measurements capable of achieving accurate pose estimations [9].

2.2 Autonomous UAVs in Inventory Management

Currently, there are a few solutions on the market that rely on UAVs for autonomous inventory management. Many of these solutions are associated with systems where the autonomous UAV is tethered to an Automated Guided Vehicle (AGV) on the warehouse floor, as is the case with the doks. inventAIRy XL [10] and Dexion Infinium Scan Drone [11] systems. The connection between the two vehicles is beneficial in terms of battery autonomy, safety, and the amount of data that the two systems can collect together. However, the usage of these two vehicles can have some drawbacks in terms of speed, agility, and cost, making it unideal in some scenarios.

Regarding systems based only on autonomous UAVs, there are also some commercially available solutions. FlytWare [12], Eyesee [13], and Ware [14] are three examples of systems with vision-based navigation and obstacle avoidance capabilities. These systems are currently available on the market and are capable of performing missions planned by operators, with tasks such as inventory counting. Additionally, a partnership between SEAT S.A. and Eurecat has started work with an autonomous UAV with vision and Ultra-wideband (UWB) sensors for performing logistics tasks in warehouse environments [15]. Despite these solutions being compatible with this thesis' objectives, they are proprietary and not scientifically well documented or described.

As for published work, Kalinov et al. [16] presented a robotic system for real-time barcode detection and scanning in the context of warehouse stocktaking, where an autonomous UAV equipped with a laser scanner and a camera for barcode detection works in cooperation with an Unmanned Ground Vehicle (UGV). The UAV localization was implemented using a system with sensors on the UGV that estimate a relative pose between both robots, while the UAV also uses barcodes for

improved localization and position adjustment. The UGV navigates on the ground below the aerial robot during the inventory record tasks, being used for navigation in the warehouse and for global localization of both vehicles.

Additionally, in [17], Harik et al. implemented a similar autonomous warehouse inventory system with an UAV and an UGV. In this case, the ground vehicle is used for localization reference, for carrying the UAV between racks, and for charging the aerial vehicle.

In the case of systems only based on UAV autonomous navigation, Kwon et al. [18] proposed a system capable of autonomous navigation between warehouse inventory racks. The system used an Extended Kalman Filter (EKF) to fuse data from three vision systems, associated with visual odometry, Apriltag recognition, and lane recognition algorithms, one range sensor, used to measure the vehicle altitude, one 2D laser scanner, and one IMU.

There is also some research on using Radio-Frequency Identification (RFID) technologies for stock identification and localization, as it is shown in both [19] and [1].

Moreover, a system proposed by Beul et al. [20] used both RFID and Apriltag markers to identify, locate, and count stock. The navigation of this system was based on Simultaneous Localization and Mapping (SLAM) algorithms and data from a 3D Light Detection and Ranging (LiDAR) sensor, having allowed the UAV to successfully execute inventory counting tasks with accurate positioning, navigation, and obstacle avoidance.

However, both RFID and LiDAR technologies are not available or easily integratable in most commercially available UAV platforms.

2.3 Mission and Task Planning

In the context of warehouse management, UAVs have to perform missions that can be clearly partitioned into distinct phases. For instance, in an inventory counting routine, the UAV is supposed to take off from its base, move to the warehouse location where the inventory is, move along racks while scanning and registering the entirety of the inventory codes, and go back to its base at the end of the routine. It should also account for critical events, as low levels of battery or navigation impediments.

Executing behavior control and planning this kind of missions can be made mainly through two different kinds of approaches: Finite-State Machines (FSMs) and Behavior Trees (BTs).

2.3.1 The State Machine Approach

A FSM is a way of modeling a system, in which a direct graph defines the system with nodes and edges that represent the system states and system transitions, respectively. In the case of mission planning, the states represent the various tasks of a mission and the links between states are associated with events of task ending or inability to end a task. Since the tasks of a mission can have multiple steps, the state machines used may need to have states that are other FSMs themselves [21]. This type of state machine is called a Hierarchical State Machine (HSM) and their concept originates from Harel's statecharts [22].

SMACH is a Python library that allows the definition of HSM with task-level architecture for complex robots [23]. Additionally to being natively compatible with ROS, the package has been extended to offer increased support with ROS-specific states, with `smach_ros` introducing possibilities as calling ROS services and listening to ROS topics [24, 25].

This library is not specific for UAV systems, but it has been used for this type of system in many occasions. In [26], the library was used to implement a framework for cooperative missions with UAVs, offering useful testing tools and a virtual world for visual understanding of the mission progress. The library was also used in [27], together with the Actionlib client-server architecture for specifying the pretended results, monitoring progress and reporting tasks completion, developing a simple task scheduler that makes use of both libraries' advantages.

SMACC is another similar purpose library, allowing the definition of event-driven, asynchronous, and behavioral state machines for complex and multi-component robotic systems in real-time ROS. The library is written in C++, and was inspired both by SMACH and Harel's statecharts. It supports ROS topics, services and actions [28].

2.3.2 The Behaviour Tree Approach

BTs are another common way to approach mission planning and behavior control, through directed graphs with a tree structure. This method originates from the video game industry, where it was first used for modeling the behavior of non-player characters. BTs have one of their main advantages related to modularity, which allows for reusing code, incremental design of functionality, and efficient testing of that functionality [29, 30].

The efficiency in creating complex modular and reactive systems make BTs crucial in many applications with autonomous agents, namely in robotics, where they have been gaining importance for controlling robots in complex missions [31].

One implementation of BTs in the field of robotics is the behavior_tree ROS library which allows users to define leaf nodes with both Python and C++ code.

The library includes two packages, one with the source code for the BT core and one containing action and condition specifications for BT leaf nodes running as external ROS nodes. The leaf nodes can be of selector, sequence, parallel, decorator, action or condition types [32].

Similarly, BehaviorTree.CPP is a ROS compatible C++ library that allows specification of BTs for robot missions. The BTs are defined in XML files, that can be directly edited or edited using the Groot Graphical User Interface (GUI) [33]. The library was designed to be flexible, easy to use, reactive and fast and support multiple types of sequence, fallback and decorator nodes [34].

Additionally, Aerostack is a software framework that supports ROS and provides reusable components specialized in functional tasks of aerial robotics and Unmanned Aerial Systems (UASs), including an user-friendly editor and an interpreter for BTs that can be used for planning and executing UAV missions specified through the editor or with python programming. The BT interpreter supports nodes for adding or removing beliefs, query nodes that verify the state of certain belief, sequence nodes, selector nodes, parallel nodes, inverter nodes, nodes that repeat the execution of child nodes a number of times or until a child node fails, and succeeder nodes that succeed independently of the child nodes return results [35].

Chapter 3

Fundamentals

This chapter addresses some of the principles that help to understand the work done in the context of this thesis, namely the main characteristics and communication structure of the ROS framework, the logic and structure of a behavior tree mission plan, the architecture and functionalities of the Aerostack framework, and the base fundamentals of Proportional-Integral-Derivative (PID) control.

3.1 ROS

ROS is a widely used, well-documented, open-source, modular, and flexible framework for developing robot applications software. It is a collection of tools, libraries, and conventions that simplify the creation of complex and robust robotic systems for a wide variety of applications, providing operating system services, such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management [36, 37].

Some of this framework's main characteristics are:

- Distributed computation: ROS supports Peer-to-Peer (P2P) communication between multiple processes and across multiple computers, being very useful in modern systems with robots that use multiple computers for controlling subsets of sensors and actuators, in robots with complex software divided into small, stand-alone parts, in multi-robot cooperation, and in human interfaces running in external computers;

- Software reuse: being widely used in a variety of research, industry, and hobbyist projects, ROS has a growing collection of algorithms for a variety of tasks, in the form of ROS packages that can be reused in new projects;

- Efficient testing: well-designed ROS systems have a clear distinction between low-level direct control of the hardware and high-level processing. Additionally, ROS allows to record and play-back sensor data and other kinds of messages and supports multiple robot simulators, providing interfaces identical or similar to the real robot interface. These two characteristics make the process of testing high-level algorithms very easy and time-efficient, as no changes to the software are required when testing the algorithms away from the physical robot;

- Support for multiple programming languages: ROS supports multiple client libraries that provide compatibility with many programming languages, such as C++, Python, Lisp, and Java;

- Variety of tools: in addition to the client libraries, ROS includes a central server, a set of command-line tools, a set of graphical tools, and a build system [38].

The ROS communication structure is associated with four fundamental concepts: nodes, messages, topics, and services.

Nodes are software modules, or processes that perform computation. A ROS system is normally composed of many nodes, with the denomination "node" being associated with the graph representation of the P2P communications at runtime, with processes as graph nodes and P2P links as arcs.

ROS nodes communicate with each other by passing messages. These messages are data structures that can be the standard primitive message types (integer, floating-point, boolean, etc.) or more complex structures composed of multiple primitive messages or arrays of primitive messages, as is the case for communicating a vehicle pose with a three-dimensional position and an orientation represented by a quaternion. These more complex messages can be part of the many message libraries available or they can be custom, defined by the developer to better suit some not contemplated specific purposes.

Nodes send messages by publishing them in topics represented by simple strings that name the type or purpose of the data that is being published. As for the receiving end of the communications, subscriber nodes subscribe to the topics with the data they are interested in. Multiple nodes can publish or subscribe to the same topic, as well as a single node can publish or subscribe to many topics, and, in general, nodes are not aware of the other nodes on the system.

Even though topic-based publish-subscribe model offers great flexibility, it is not appropriate for synchronous communications and Remote Procedure Call (RPC), often required in a distributed system. For this particular type of communications, ROS services are used for request/response communication. Services are defined by two messages, one for the request and another for the response, being that the service is provided by a certain providing node under a string name, and a client node calls the service by sending the request message and awaiting the corresponding response. In this context, only one node is allowed to advertise a service of any particular node, but the service is available to all the nodes on the communication network [39].

3.2 Behavior Trees

A BT's structure, as exemplified by the typical behavior tree model in Fig. 3.1, is defined by a upside-down tree consisting of various nodes and edges. In this representation, nodes are associated with actions and, when the system size increases, the behavior tree is expanded with new behavior nodes or new behavior subtrees. The root node, which corresponds to the node at the beginning of the tree with no parent nodes, sends enabling signals to all the tree's child composite nodes and receives the corresponding return results. These composite nodes can have many types, such as selection nodes, sequential nodes, modifier nodes, and parallel nodes. They are the parents to condition nodes, action nodes, and other composite nodes. Condition nodes are logical nodes that return true or false results to a certain condition and action nodes are associated with the execution of specific behaviours. During execution, the system constantly reads the BT's return results, going from top to bottom and left to right until the currently activated node is reached and the behavior tree is refreshed [40].

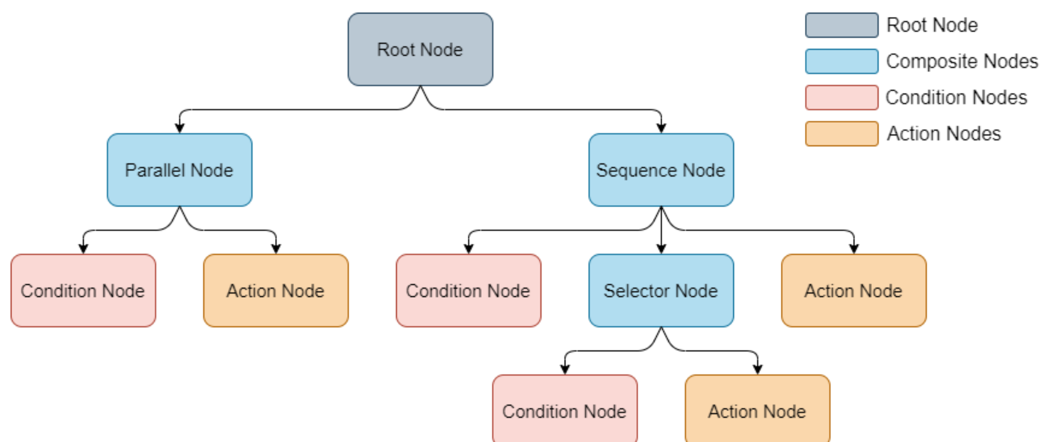


Figure 3.1: Representation of a typical behavior tree model.

3.3 Aerostack

Aerostack is an open-source software framework focused on the development of aerial robotic systems, providing a collection of reusable components specialized in common tasks of aerial robotics and an integration method in a multi-layered cognitive architecture [35].

Among the many framework functionalities, Aerostack has software tools programmed in C++ and Python for mission planning and execution with behavior trees, some interfaces for hardware and simulators, SLAM processes, motion controllers and planners, and support for multiple robotic behaviors, as self localization, attention, navigation, and social interaction with other robots and human interfaces.

Some of the Aerostack framework's main characteristics are:

- Compositive structure: Aerostack's software architecture has different layers with multiple software blocks that can be configured, adapted, or added to the stack to develop many different systems;
- Distributed processing: each software block is independent and can be executed in onboard or ground station computers in a local network, communicating using ROS;
- Open-source code: the framework's software is available to researchers and developers under a BSD-3 license and has periodic software launches;
- Hardware independence: Aerostack works in most types of computers, including Single-Board Computers (SBCs), and can be used with multiple different UAV platforms;
- Support for teleoperation and autonomous flight: Aerostack has tools for both manual control in teleoperated flights and for executing autonomous UAV missions;
- Support for multiple robots: the framework supports multi-robot aerial missions [41].

3.3.1 Architecture

The Aerostack reference architecture, as seen in Fig. 3.2, shows various blocks that correspond to the framework's multiple software modules implemented in the form of ROS nodes. These modules are organized in the following categories:

- Interfaces: these modules allow to receive data from sensors, send commands to actuators, and communicate with human operators and other robots;

- Communication channel: this channel contains the shared dynamic information between processes, being implemented with ROS topics and messages;
- Robot behaviors: these modules are associated with the functional abilities of the system, such as state estimation, motion control, and motion planning.
- Behavior management: the execution of the system behaviors is controlled by a behavior management system;
- Belief memory: the belief memory system saves relevant states of the world;
- Mission control: the mission control module is responsible for executing mission plans in the form of a behavior tree or a Python program. Additionally, this module includes processes for safety monitoring and recovery to provide fault tolerance [42].

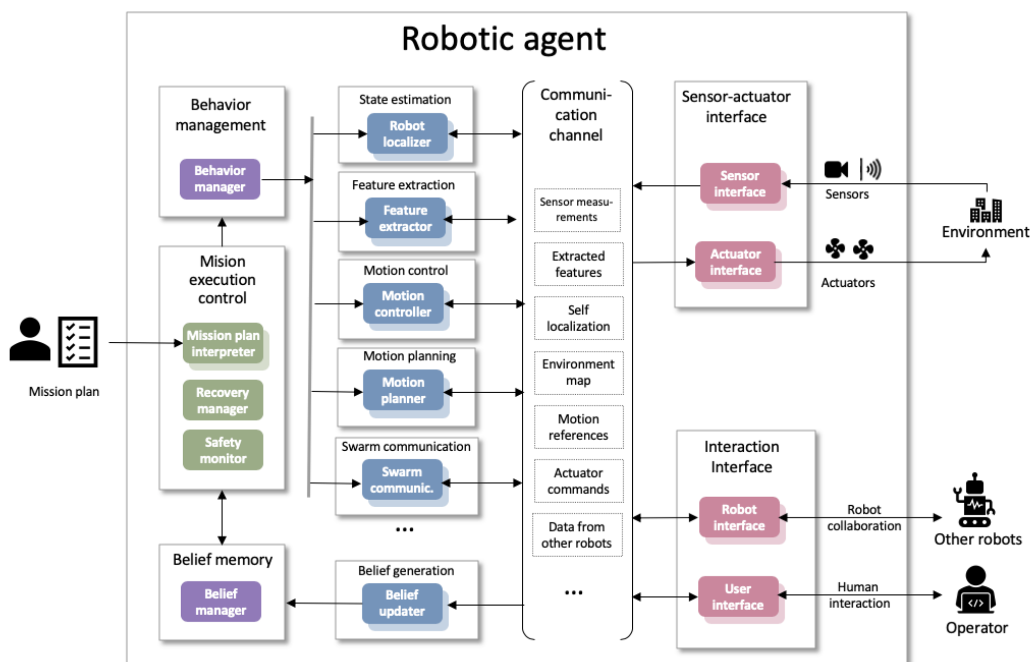


Figure 3.2: The Aerostack reference architecture (Version 4.0) [42].

3.3.2 Behaviors

Aerostack has a vast collection of behaviors available for simple usage in missions for the UAV systems developed with the framework. The name and description of the Aerostack Version 4.0 behaviors that were used in the system implemented in the context of this thesis are listed in Table 3.1.

Behavior	Description
TAKE_OFF	The UAV takes off vertically from the ground, reaching the default takeoff height.
LAND	The UAV lands on the ground below its current position.
WAIT	The UAV waits a certain amount of time, no running tasks are stopped.
HOVER	The UAV keeps hovering in its current position.
ROTATE	The UAV rotates left or right a certain number of degrees or until it reaches a certain number of degrees in heading, depending on the configuration.
FOLLOW_PATH	The UAV tries to follow a path, defined by a sequence of 3D points.

Table 3.1: Aerostack behaviors used in the UAV indoor warehouse navigation system missions [42].

3.3.3 Mission Planning and Execution

The framework's mission plan interpreter supports both the usage of mission plans with behaviors in a Python program or in a behavior tree structure.

Regarding the usage of behavior trees, in addition to the behavior operation nodes and nodes associated with interactions with the belief memory system, the behavior tree hierarchy includes intermediate nodes that establish the control regimes. The types of intermediate nodes available in the Aerostack framework are:

- Sequence node: this node's child nodes are executed in sequence, succeeding only if all child nodes succeed;
- Selector node: similarly to the sequence node, all child nodes are executed in sequence, although the node succeeds if any of the child nodes succeed;
- Parallel node: child nodes are executed at the same time, in parallel, and the node succeeds if the number of successful child nodes is equal or greater than a user defined threshold number;
- Repeat node: this node can only have one child node and returns success when the execution of the child node is repeated a determined number of times;
- Repeat until fail node: similarly to the repeat node, this intermediate node can only have one child node and always succeeds, although the child node execution is repeated until it fails;
- Inverter node: this node returns the opposite execution result of its child node, returning success if the child node fails and fail if the child node succeeds;
- Succeeder node: this node always returns success, regardless of the result of its child node execution [43].

Aerostack has a behavior tree editing tool that allows the users to make mission plans in a visually intuitive manner, by placing nodes in a graphic representation of the mission behavior tree.

3.4 PID Controllers

PID controllers are three-term controllers with proportional, integral, and derivative terms, being very common and one of the most widely used control loop mechanisms. The input to the controller is an error signal based on the difference between a desired reference signal and the actual measured system output, and the output of the controller applies a correction to the system that is intended to allow the system to achieve a desired output.

As a three-term controller, a PID controller is based on three types of control:

- Proportional control: associated with the PID controller's P-term, the proportional control is used for a controller action proportional to the error $e(t) = r(t) - y(t)$, the difference between the desired reference signal $r(t)$ and the actual measured system output $y(t)$. The proportional control representation, in the time domain, is given by the control signal $u_c(t)$, with

$$u_c(t) = k_P e(t) \tag{3.1}$$

  , where k_P is the controller proportional gain;

- Integral control: associated with the PID controller's I-term, the integral control is used for correcting any output offset from the reference signal. Unlike proportional control, integral control eliminates offset without using excessively large controller gains. As for the integral control time domain representation, the control signal equation is represented by

$$u_c(t) = k_I \int^t e(\tau) d\tau \tag{3.2}$$

  , with k_I being the integral controller gain;

- Derivative control: associated with the PID controller's D-term, the derivative control uses the rate of change of the $e(t)$ error to introduce an element of prediction to the controller. Representing the derivative control in the time domain, the control signal is given as

$$u_c(t) = k_D \frac{de}{dt} \tag{3.3}$$

  , where k_D is the derivative controller gain.

When combining the three types of control, the proportional control affects the speed of the system response, the integral control can be used to eliminate offset in the reference response, and the derivative control can be used to tune response

damping [44]. The typical structure of the parallel PID controller is represented in the time domain in Fig. 3.3.

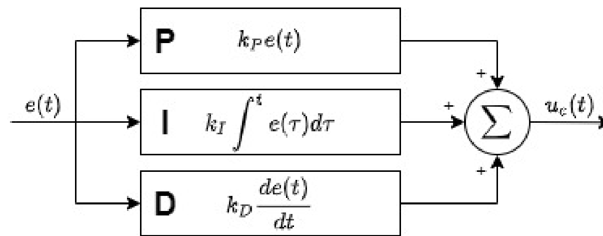


Figure 3.3: Block diagram of a PID controller.

Chapter 4

Project

This chapter presents some choices that were made in the development of the project, regarding the commercial UAV platform and the corresponding interface, the localization algorithms, the framework used for mission planning and navigation, and the proposed system architecture.

4.1 UAV Platform and Interface

The robot platform choice was based on the need for straightforward integration of a small commercially available UAV on the warehouse inventory management processes, without the need for implementation of additional sensors. Following the work previously done in [3], where a platform was selected and the interface and localization systems for that platform were developed, this project used the DJI Mavic 2 drone. This decision allowed to adapt the localization algorithms for integration in the developed navigation system.

The DJI Mavic 2 is equipped with a monocular camera placed on a 3-axis gimbal, an IMU, altimeter, Global Positioning System (GPS), and eight vision sensors (without access to these cameras' images), being all eight for obstacle detection and two with additional visual odometry functionality. When obstacles are detected by the collision avoidance vision sensors, the aircraft has the ability to stop its movement and hover.

The controller for the manual operation of the drone functions both as a transmitter of vehicle commands and as a receiver for the vehicle's sensor data. Additionally,

this platform has a Software Development Kit (SDK) that allows the creation of mobile apps capable of using the controller interface functionalities, when a smartphone is connected to the controller via a Universal Serial Bus (USB) cable. These were the reasons that justified the usage of an Android smartphone as the interface between the vehicle system and the ground-station computer used for processing localization and navigation algorithms.

Regarding the communication between the mobile device and the ground-station computer, the obvious choice for the system was the usage of ROS, because of it being open-source and language-agnostic, having an extensive user base, being widely used in academic research, and having a vast collection of compatible software, that can be integrated into new systems. ROS was also used in the development of the localization algorithms, prior to this project, which makes the localization system compatible and easy to implement in the navigation system.

The usage of ROS for communication requires the creation of a network that both the mobile device associated with the interface and the ground-station computer can connect to. Most recent distributions of Android support WiFi mobile hotspot functionality, which was used in this project because it showed to be an effective and simple way of implementing a wireless network for both devices. In this context, the ground-station computer was used to run the ROS master and the other roscore nodes, being the computer responsible for the majority of processes and leaving the mobile device with only interface-related processes.

4.2 Localization System

The localization algorithms used for the navigation system implemented in this thesis were adapted from the work done in [3], being the optimized version of these algorithms that was used in this project implemented on a different master's thesis. The decision to use this localization approach was justified because the algorithms were implemented with similar objectives and showed to be reliable in estimating the pose of the DJI Mavic 2 in indoor unmapped environments, as required for this application.

The localization system uses the vehicle inertial data and data acquired from the visual observation of artificial landmarks as the inputs for a Graph-SLAM algorithm that is able to estimate a vehicle pose, an optimized trajectory, and the artificial landmarks' poses.

Regarding the inertial data obtained from the UAV through the mobile app interface, it consists of the vehicle attitude, represented with roll ϕ_B, pitch θ_B, and yaw angles ψ_B, and the three-dimensional velocity vector represented by (V_x, V_y, V_z). This data is the input for a pose estimator that obtains a vehicle pose estimation, the motion input α_i of the Graph-SLAM algorithm.

As for the visual processing, the image from the UAV's main camera I_n is processed through visual perception algorithms, where the apriltag_ros package [45] is used to identify Apriltag visual markers, that should be placed throughout the warehouse without the need for mapping their location, and determines the relative pose between the vehicle camera and the markers. The output of the visual perception algorithms is the measurement input ζ_n, with the relative poses between the vehicle and the Apriltag markers.

The Graph-SLAM algorithm, which is based on Georgia Tech Smoothing and Mapping (GTSAM) [46], creates a factor graph that establishes relations between consecutive vehicle states, based on the motion inputs, and between vehicle states and landmark states, based on the measurement inputs. This method allows estimating vehicle poses at the inertial data rate while using the visual markers observations to optimize those poses and the vehicle trajectory, preventing the drift errors associated with inertial measurements.

The structure of the localization system that was used in this project is represented in Fig. 4.1.

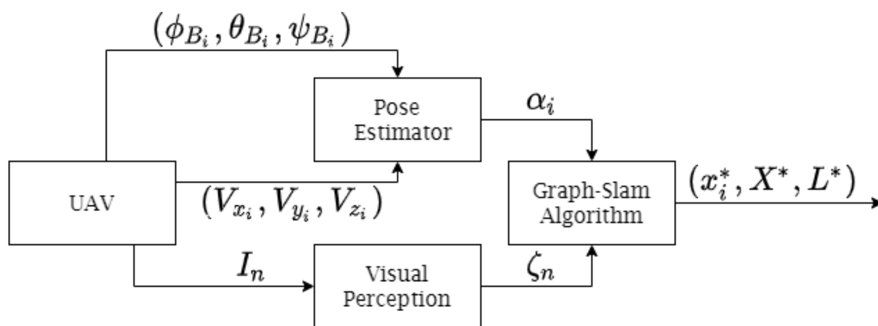


Figure 4.1: Structure of the localization system.

4.3 Mission Planner and Navigation

When considering the mission planner for the navigation system, it was necessary to choose between a state machine or a behavior tree approach for the definition of the UAV missions in the context of warehouse inventory management. We opted for the behavior tree approach, because of the modularity of behavior trees and their intuitive structure, which facilitate the definition of missions by the users.

Following this decision, it was necessary to choose a method to implement the behavior tree missions in a navigation framework. For this purpose, and after considering various ROS-compatible packages and libraries, we opted for using the Aerostack software framework's modules for mission planning and execution [42]. This choice was motivated by the vast amount of UAV behaviors already available

in the framework that can be used in inventory management missions, the simplicity of the Aerostack behavior tree editor, and the ability to add support for new hardware and UAV platforms through the development of simple interfaces.

In addition to these mission planner modules and behaviors, there are some other Aerostack modules that were useful in this project, being used in the implementation, such as the path tracker algorithms and the keyboard teleoperation.

4.4 Proposed Architecture

Considering all the hardware, communication interfaces, and software modules, the architecture proposed for the navigation system is represented in Fig. 4.2.

This architecture is composed of four hardware elements: the DJI Mavic 2 UAV, the DJI remote controller, an Android smartphone, and the ground-station computer.

The communication between the aircraft and its remote controller is established with a proprietary protocol that alternates between 2.4 GHz and 5 GHz Wi-Fi to ensure a stable connection. This connection is used to transmit vehicle sensor data from the vehicle to the controller and commands from the controller to the vehicle.

As for the Android smartphone, this device is used to run the interface between the UAV system and the ROS-compatible navigation algorithms running in the ground-station computer. This interface was implemented in an Android app that uses the DJI Mobile SDK to access the communications with the vehicle, transforming the vehicle sensor data to be compatible with ROS, and, in the opposite direction, translate ROS command messages that are sent by the navigation system to be compatible with the UAV operation. The smartphone is connected to the DJI remote controller via a USB connection, which replicates the functionality of the connection between the aircraft and its controller, giving the Android app interface the ability to control the exchange of the vehicle sensor data and the commands between the vehicle and the navigation system.

Finally, the communication between the smartphone and the ground-station computer is done through a Wi-Fi 5 GHz network, using the smartphone's Wi-Fi hotspot functionalities. This connection allows the computer to send commands to the vehicle and receive vehicle sensor data, through the use of ROS for communication.

In particular, regarding the navigation algorithms that run in the ground-station computer, these can be divided in five categories: image transformation, coordinate frames manager, localization, hardware interfaces, and Aerostack framework modules.

As for the image transformation, it is composed of two parts that respectively decompress the image received from the UAV through the ROS interface and rectify that decompressed image according to the intrinsic camera parameters.

The coordinate frames manager refers to using the ROS tf package to transform points and vectors between different coordinate frames. In this particular navigation system, the package is used to convert vehicle attitude, vehicle velocity, and visual markers' poses to the same world map-fixed frame.

The localization algorithms were the continuation of the work done in [3] and was implemented in the context of another master's thesis, not being documented in detail as part of this one. The inputs of this localization algorithms are the vehicle attitude, the vehicle linear velocity, and its main camera's image. The attitude and velocity data are used to estimate the vehicle pose and the main camera image is used in a visual markers detector to identify and estimate the pose of Apriltag visual markers. These two estimation results are then used in a Graph-SLAM algorithm that estimates the optimized pose of the vehicle.

Regarding the hardware interfaces, the navigation system has two software modules that are meant to create compatibility between the Aerostack framework's software, the UAV system, and the localization system. The first module, "mavic_state interface", receives the vehicle velocity, the vehicle attitude, and flight actions, and provides the flight state variable, essential to the Aerostack modules operation. The second module, "mavic_command interface", manages the commands sent to the vehicle through the ROS interface, using as inputs the optimized pose of the vehicle, variables provided by the Aerostack modules for reference poses, reference speed values, and flight actions, and the vehicle's responses to the commands sent.

The Aerostack modules are integrated in this navigation system for planning, execution, and management of missions. These missions are declared by the user in the form of a mission plan, which in our implementation takes the form of a behavior tree, and can be edited using Aerostack's behavior tree editor. The missions are controlled by the mission execution controller, that communicates with the behavior manager for execution of different behaviors and with the Aerostack belief memory system. The behavior manager is responsible for managing the behavior execution, which involves many behaviors with different functions, some already existent in the framework and some newly implemented or adapted. These behaviors are associated with the creation of flight actions, reference speed values, and reference poses, used for the creation of vehicle commands.

Some of the behaviors that were used in the navigation system were part of the "basic_quadrotor_behaviors" package, associated with tasks such as take-off and landing. Additionally, the package "quadrotor_motion_with_pid_control" allowed the execution of tasks related to following paths, hovering, and rotating the UAV. The "keyboard_teleoperation_with_pid" was also used as an fail-safe alternative to

the autonomous control, after being adapted to work with this particular UAV, and new gimbal and photo behaviors were created to allow sending commands to the vehicle, respectively for controlling the attitude of the main camera gimbal and for taking photos with the main camera during the mission.

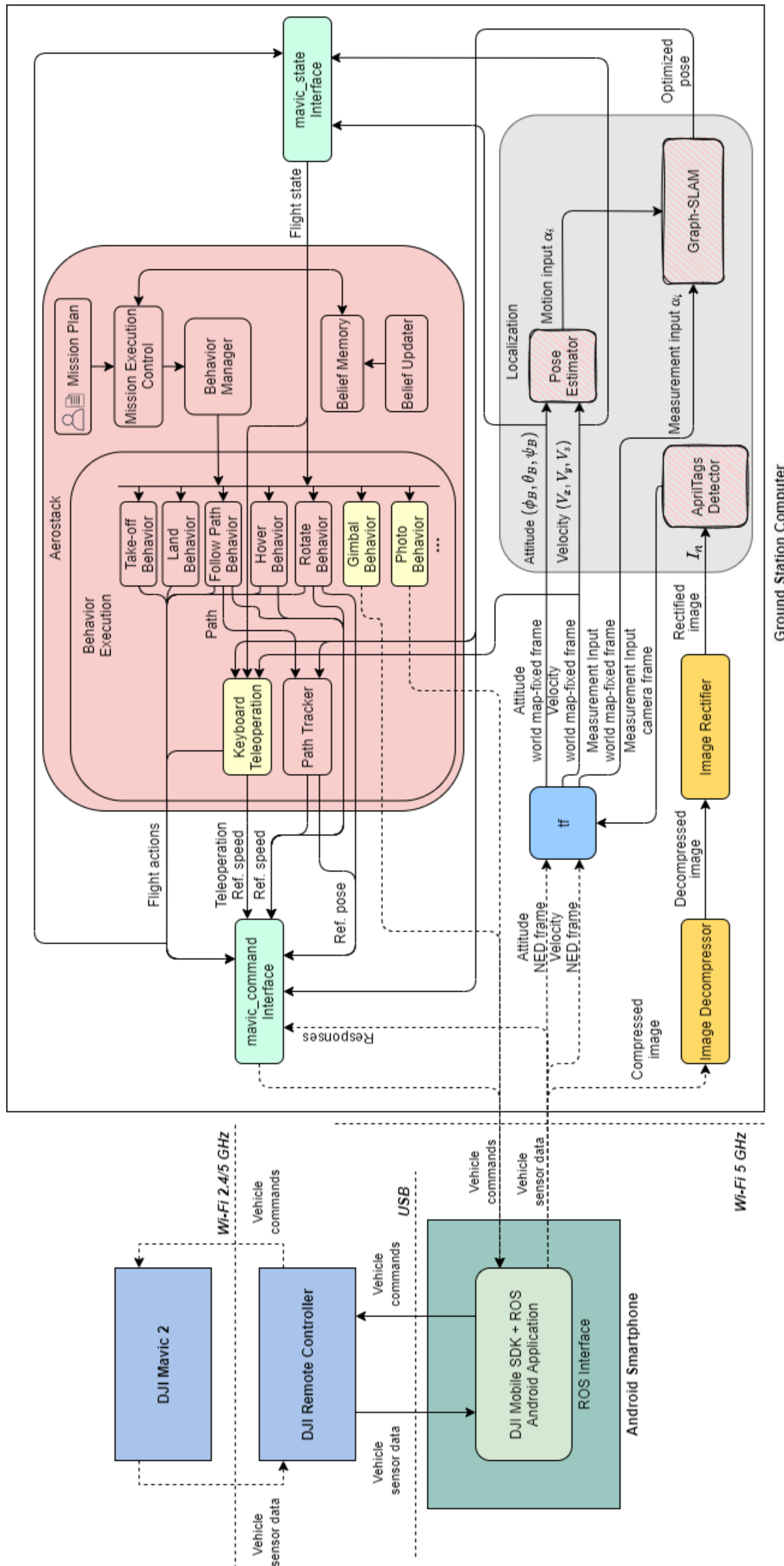


Figure 4.2: Proposed architecture of the navigation system.

Chapter 5

Implementation

The implementation of the navigation system required the development of an interface for creating compatibility between the UAV system and ROS, packages for image transformation, hardware interfaces, and new behaviors in the Aerostack framework. This chapter describes the details of these software modules, while also presenting the conditions in which the navigation system tests were conducted and the mission plan that was used for these tests.

5.1 ROS Interface with DJI Mobile SDK

Considering the choice of the DJI Mavic 2 for the system implementation, it was necessary to develop an interface capable of exchanging sensor data from the UAV to the ground station computer and action commands from the computer to the vehicle. This interface was implemented in an Android app, coded with Java and XML programming languages, built using the DJI Mobile SDK available tools and functions, and using ROS topics for the communications, through the use of rosjava [47, 48].

Regarding its operation and functionality, in terms of the User Interface (UI), the developed Android app is composed by three distinct pages, respectively associated with the connection to the UAV remote controller, the detection of a running ROS master, and the normal running operation with interface functionalities.

At startup, the first page of the app, presented in Fig. 5.1, shows an "Open" button that is unlocked after the connection with the controller is established and

the vehicle is connected to the controller. It is also in this page that in its first time starting up, the app is registered with an access token, which requires an internet connection. After this initial page, clicking the button takes the user to the second page.

The second page, as seen in Fig. 5.2, has a text input that allows the user to define the value associated with the ROS_MASTER_URI setting, which locates and identifies the ROS Master in the network, in this case being composed of the ground-station computer IP address and the port associated with the ROS Master. There is also a button on this page that allows to define this setting by reading a Quick Response (QR) code with the required information. Additionally, there are two other buttons on this page, a "Connect" button that applies the changes associated with the ROS_MASTER_URI setting, establishes the communications, and takes the user to the third app page, and a "Cancel" button to exit the page.

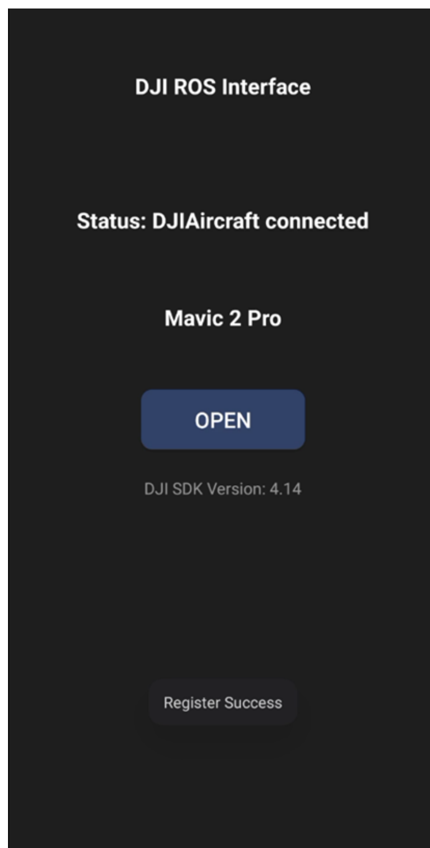


Figure 5.1: First page of the ROS interface Android app.

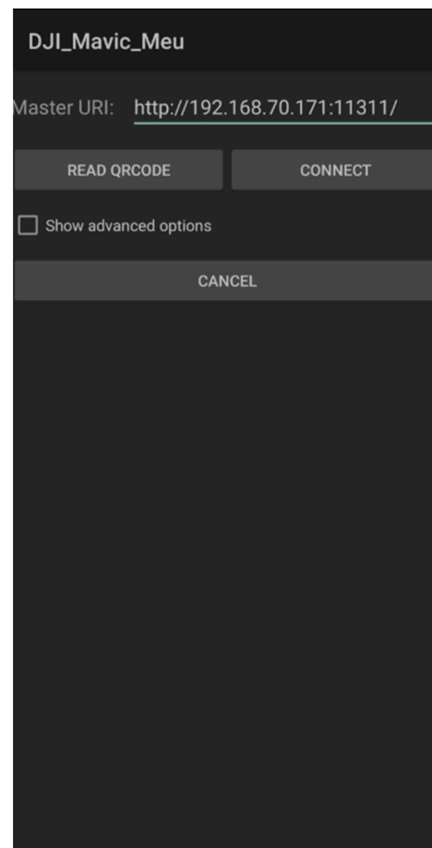


Figure 5.2: Second page of the ROS interface Android app.

Finally, in the third page of the app, the interface is set up and actively running with interface functionalities. The layout of this page, represented in Fig. 5.3, has a video stream from the UAV main camera and buttons that allow to capture photos and record video to the aircraft's internal memory.

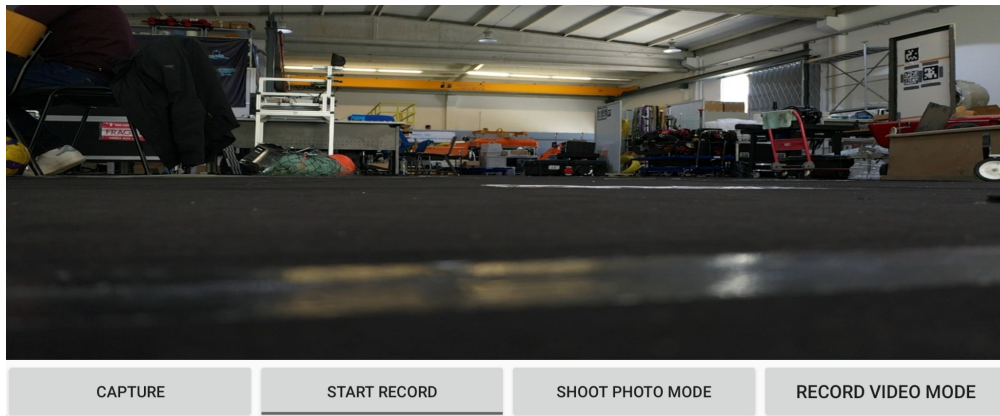


Figure 5.3: Third page of the ROS interface Android app.

As for the operation of the interface, after the setup in the two initial pages, the app is running various ROS publishers, responsible for sending data and request responses from the vehicle to the ground-station computer, and one ROS subscriber, for receiving commands and requests sent from the ground-station navigation system.

The ROS publishers associated with vehicle data are coded with individual threads assigned, as exemplified by the code in Listing 5.1, associated with the periodic publication of velocity data in a ROS topic. This method is used because it is an efficient way to manage the various processes of asking the aircraft for a new sensor data message, translating it, and publishing it to a ROS topic, which is individually done for every data type provided by the multiple vehicle sensors.

The ROS topics that are published through the interface app are listed in Table 5.1. The topics published using individual threads are vehicle data related, being associated with vehicle attitude, vehicle velocity, main camera gimbal attitude, battery status and main camera video feed. Additionally, the topic "Responses" occasionally publishes responses to requests and commands from the ground-station computer that indicate if these requests and commands were interpreted and answered by the UAV.

Table 5.1: ROS topics published by the interface.

ROS Topic	**Message Type**	**Frequency**
Attitude_RPY	geometry_msgs/PointStamped	20 Hz
Velocity	geometry_msgs/QuaternionStamped	20 Hz
Gimbal_Attitude	geometry_msgs/PointStamped	20 Hz
Battery	sensor_msgs/BatteryState	1 Hz
Video_Feed	sensor_msgs/CompressedImage	16 Hz
Responses	std_msgs/Int32	——

```
final Publisher<geometry_msgs.QuaternionStamped> Velocity =
    connectedNode.newPublisher("drone0/Velocity",
"geometry_msgs/QuaternionStamped");

connectedNode.executeCancellableLoop(new CancellableLoop() {
  protected void loop() throws InterruptedException {
    Get_Velocity_Altitude();
    geometry_msgs.QuaternionStamped Value =
        (geometry_msgs.QuaternionStamped) Velocity.newMessage();
    geometry_msgs.Quaternion Value_Point = Value.getQuaternion();
    std_msgs.Header Head = Value.getHeader();
    Head.setStamp(connectedNode.getCurrentTime());
    Value_Point.setX(Velocity_X_Drone);
    Value_Point.setY(Velocity_Y_Drone);
    Value_Point.setZ(Velocity_Z_Drone);
    Value_Point.setW(Altitude_Drone);
    Value.setQuaternion(Value_Point);
    Value.setHeader(Head);
    Velocity.publish(Value);
    Thread.sleep(50L);
  }
});
```

Listing 5.1: Code for publishing vehicle velocity data using threads.

In the particular case of the ROS topic "Video_Feed" and the publication of the main camera image messages, additional processing was needed to create a message compatible with rosjava. Using the SDK, it is possible to obtain uncompressed images of the SurfaceTexture variable type, which need to be converted to a ChannelBufferOutputStream variable type before being published as a "sensor_msgs/CompressedImage" message. In order to complete this conversion, a callback function is used to receive each SurfaceTexture video frame, converting it into a 1280x720 bitmap, and then into a JPEG byte array, with 90% compression. Then, the conversion from the byte array to the ChannelBuffer, used in the ChannelBuffer-OutputStream variable, is done in the thread associated with generating the image message, before publishing it to the "Video_Feed" ROS topic as a compressed image. Listing 5.2 shows the callback function responsible for receiving the image in the SurfaceTexture variable type and converting it into a byte array. Additionally, Listing 5.3 shows the code for the thread responsible for generating and publishing the final compressed image message to the "Video_Feed" ROS topic.

On the other direction, the ROS topic "Mavic_Commands", that is published by the navigation system running in the ground-station computer and subscribed by the interface, contains request and command messages of type "std_msgs/String". The interface subscribes these messages, interprets them, and sends the appropriate correspondent commands to the vehicle, through the functions available in the DJI Mobile SDK. The messages associated with the "Mavic_Commands", related to

```
public void onSurfaceTextureUpdated(SurfaceTexture surface) {
    Bitmap image;
    image = mVideoSurface.getBitmap();
    Bitmap Image = getResizedBitmap(image, 1280, 720);
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    Image.compress(Bitmap.CompressFormat.JPEG, 90, stream);
    byteArray = stream.toByteArray();
    flag_done = true;
}
```

Listing 5.2: onSurfaceTextureUpdated function, responsible for receiving the main camera image in the SurfaceTexture variable type and converting it into a byte array.

```
Video = connectedNode.newPublisher("drone0/Video_Feed", "sensor_msgs/
CompressedImage");
connectedNode.executeCancellableLoop(new CancellableLoop() {
    protected void loop() throws InterruptedException {
        if(Camera_Node.flag_done==true){
            ChannelBuffer buffer = ChannelBuffers.copiedBuffer(
                ByteOrder.LITTLE_ENDIAN, Camera_Node.byteArray);
            ChannelBufferOutputStream stream_1 = new
                ChannelBufferOutputStream(buffer);
            sensor_msgs.CompressedImage Value = (CompressedImage)
                Video.newMessage();
            Value.setData(stream_1.buffer().copy());
            stream_1.buffer().clear();
            Video.publish(Value);
            Camera_Node.flag_done=false;
        }
        Thread.sleep(60);
    }
});
```

Listing 5.3: Thread responsible for generating and publishing the messages in the Video_Feed ROS topic.

commands for turning on or off the vehicle motors, taking off, landing, switching on or off the virtual joystick remote control of the aircraft, setting up the control modes and flight coordinate system of the aircraft, executing main camera gimbal rotation movements, controlling vehicle linear velocity and yaw angular speed, and taking photos with the main camera, are listed in Table 5.2.

Table 5.2: Mavic_Commands and correspondent Response messages.

Mavic_Commands message	**Arguments**	**Response possible messages**
"Motor (state)"	state = {"On", "Off"}	0 = failure 1 = success
"Takeoff"	——	0 = failure 1 = success
"Land"	——	0 = failure 1 = success
"Joystick (state)"	state = {"On", "Off"}	0 = failure 1 = success
"SetUp (x) (y) (z) (w)"	x = {0, 1} (vertical control mode: 0 = velocity, 1 = position) y = {0, 1} (roll pitch control mode: 0 = angle, 1 = velocity) z = {0, 1} (yaw control mode: 0 = angle, 1 = angular velocity) w = {0, 1} (flight coordinate system: 0 = ground, 1 = body)	0 = failure 1 = success
"Gimbal (x) (y) (z) (w)"	x = roll angle in degrees y = pitch angle in degrees z = yaw angle in degrees w = gimbal rotation movement duration in seconds	0 = failure 1 = success
"Velocity (x) (y) (z) (w)"	x = x velocity in m/s y = y velocity in m/s z = z velocity in m/s w = yaw angular velocity in deg/s	0 = failure 1 = success
"Take Photo"	——	0 = failure 1 = success

The command messages in the "Mavic_Command" topic are received by the interface through a ROS subscriber, declared as shown in Listing 5.4, and are interpreted in a callback function that translates these commands for the robot, using different functions for every command, each one using the tools available through the SDK to complete the corresponding action, as seen in Listing 5.5.

These individual functions are not only responsible for sending the commands to the vehicle, but also manage the feedback given to the ground-station computer about the result of those commands, through the ROS topic "Responses". Listing 5.6 is an excerpt from the Send_Flight_Control_Data_To_Aircraft() function, that shows the way the commands are sent to the vehicle and the way eventual errors from the vehicle in interpreting those commands are reflected in the published "Response" messages, which is handled through the Send_Command_Result() function. This particular example relates to three-dimensional linear velocity and yaw angular velocity commands.

```
Receive_Command = connectedNode.newSubscriber("drone0/Mavic_Commands",
    "std_msgs/String");

Receive_Command.addMessageListener(message -> {
  String Command = message.getData();
  Parse_Command(Command);
});
```

Listing 5.4: Vehicle commands subscriber and corresponding callback function.

```
public void Parse_Command(java.lang.String Command) {
  if (Command.equals("Motor Off")) {
    Motors(false);
  }
  else if (Command.equals("Motor On")) {
    Motors(true);
  }
  else if (Command.equals("Takeoff")) {
    TakeOff_Landing(false);
  }
  else if (Command.equals("Land")) {
    TakeOff_Landing(true);
  }
  else if (Command.equals("Joystick Off")) {
    Activate_Virtual_Joystick(false);
  }
  else if (Command.equals("Joystick On")) {
    Activate_Virtual_Joystick(true);
  }
  else if (Command.startsWith("SetUp")) {
    Set_Up(Command.charAt(6), Command.charAt(8), Command.charAt(10),
        Command.charAt(12));
  }
  else if (Command.startsWith("Gimbal")) {
    Rotate_Gimbal(Command);
  }
  else if (Command.startsWith("Velocity")) {
    Send_Flight_Control_Data_To_Aircraft(Command);
  }
  else if (Command.equals("Take Photo")){
    Take_Photo();
  }
  else {
    Send_Command_Result(0);
  }
}
```

Listing 5.5: Parse_Command function associated with translating the string command messages present in the Mavic_Command ROS topic to vehicle commands, using DJI Mobile SDK functions.

```
FlightControlData velocity_data;
velocity_data = new FlightControlData((float) Velocity_X,(float)
    Velocity_Y, (float) Velocity_Yaw, (float) Velocity_Z);

((Aircraft)
    product).getFlightController().sendVirtualStickFlightControlData(
velocity_data, djiError -> {
  if (djiError == null) {
    Send_Command_Result(1);
  } else {
    Send_Command_Result(0);
  }
});
```

Listing 5.6: Code associated with sending three-dimensional linear velocity and yaw angular velocity commands to the UAV and handling of the command responses sent back to the ground-station computer.

5.2 Navigation System

As described in Section 4.4, the navigation system is composed of software divided in five main categories: image transformation, coordinate frames manager, localization, hardware interfaces, and Aerostack framework modules.

The image transformation and vehicle interface software modules were fully developed specifically for this implementation. As for the coordinate frames manager, a standard implementation of the ROS tf package was used. The localization algorithms were not developed in this thesis, but follow the principles described in Section 4.2. Regarding the implementation of the Aerostack framework that was done in this project, some modules were adapted to be compatible with the system as well as some new behaviors were developed. Most of the software used in the navigation system, and in particular all the software that was developed or adapted, was written in the C++ programming language.

5.2.1 Image Tranformation

The images published through the ROS interface to the "Video_Feed" topic are compressed and, in order to be used in computer vision algorithms, these images need to be decompressed and then rectified, according to the UAV's main camera intrinsic parameters.

Image Decompression

Decompressing the image that is sent to the ground-station computer was done using the imdecode() function from the OpenCV library [49] and the CVBridge library [50], used to convert between OpenCV images and ROS image messages. The code

in Listing 5.7 refers to the usage of these tools to decompress the image, publishing the decompressed image, and publishing the CameraInfo message associated with the UAV's main camera meta information. The CameraInfo message contains lens distortion and intrinsic parameters that were obtained through calibration of the camera.

```
void mavic_receive_video::uncompress_frame_and_pub()
{
    camera_info_manager::CameraInfoManager caminfo(nh, camera_name,
        calibration_file_path);
    sensor_msgs::Image raw_video_msg;
    sensor_msgs::CameraInfo camera_info_msg;
    Mat image =
        cv::imdecode(cv::Mat(image_compressed.data),IMREAD_ANYCOLOR);

    std_msgs::Header header=image_compressed.header;
    cv_bridge::CvImage img_bridge = cv_bridge::CvImage(header,
        sensor_msgs::image_encodings::BGR8, image);
    img_bridge.toImageMsg(raw_video_msg);

  camera_info_msg.header = header;
    camera_info_msg=caminfo.getCameraInfo();
    camera_info_pub.publish(camera_info_msg);
    video_raw_pub.publish(raw_video_msg);
}
```

Listing 5.7: Function responsible for publishing the camera/image_raw and camera/camera_info ROS topics.

Image Rectification

As for the rectification of the image, it was done using the image_proc package nodelets [51], which subscribe the image in camera/image_raw and the meta information in camera/camera_info and publishes the rectified image in the camera/image_rect ROS topic.

5.2.2 Hardware Interfaces

State Interface

The "mavic_state" interface was created to provide the Aerostack framework modules information relative to the vehicle flight state. The flight states supported by the framework are described in Table 5.3, they allow the framework modules to make decisions, such as what behaviors can be run and if a behavior was completed, based on the limitations imposed by the current vehicle flight state.

Table 5.3: Vehicle flight states.

Flight state	Description
LANDED	The UAV is considered landed at system start-up and after being in the LANDING state, when the vehicle altitude and the velocity in the z-axis are null
FLYING	The flying state is achieved after the TAKING_OFF state, when the velocity in the z-axis is null and the vehicle reached a predetermined altitude, or after a MOVE action, when the vehicle is above take-off altitude and doesn't have null linear and angular velocities
HOVERING	The hovering state can be achieved after a MOVE or HOVER command, when the vehicle linear and angular velocities are null
TAKING_OFF	The UAV achieves the taking off state after the TAKE_OFF command, if the current flight state is either LANDED or UNKNOWN
LANDING	The UAV achieves the landing state after the LAND command, if the current flight state is either HOVERING or LANDING
UNKNOWN	This vehicle state indicates that the current vehicle flight state is unknown

The state interface has three ROS subscribers, for subscribing to velocity and altitude, attitude, and the flight actions generated by the behaviors executed through the Aerostack framework, and four ROS publishers, that are responsible for publishing messages relating, not only to the flight state variable, but also to other topics that can be used by the framework, relating to raw linear speed, IMU sensor measurements, and the vehicle altitude. The declaration of these subscribers and publishers and their corresponding topic names can be seen in Listing 5.8.

Regarding the vehicle flight states, the interface uses the current flight action, the previous state, the linear velocity, the angular velocity, and the altitude to determine the current vehicle flight state, through the connections described in Table 5.3. These connections for determination of the current flight state are implemented in the code in Listing 5.9, where a switch structure is used to infer the current flight action command and various if conditions are used to evaluate previous state, linear velocity, angular velocity, and altitude, and ultimately determine the current vehicle state. The flight actions that can be received from the Aerostack behavior execution are TAKE_OFF, HOVER, LAND, MOVE, and UNKNOWN, referring respectively to the commands for the vehicle to initiate vertical flight, to fly with null linear and angular velocities, to land on the ground below its current pose, to execute some movement command, or to an unknown command.

```
void StateInterface::ownSetUp() {
    cout<<"[ROSNODE] Mavic State Setup"<<endl;
    velocity_sub = nh.subscribe("Velocity_World", 1,
        &StateInterface::velocityCallback, this);
    attitude_sub = nh.subscribe("Attitude_World", 1,
        &StateInterface::imuCallback, this);
    flight_action_sub=nh.subscribe("actuator_command/flight_action", 1,
        &StateInterface::flightActionCallback, this);
    flight_state_pub=nh.advertise<aerostack_msgs::FlightState>(
        "self_localization/flight_state", 1, true);
    linear_speed_rad_pub=nh.advertise<geometry_msgs::TwistStamped>(
        "raw_localization/linear_speed", 1, true);
    imu_rad_pub =
        nh.advertise<sensor_msgs::Imu>("sensor_measurement/imu", 1,
        true);
    altitude_pub = nh.advertise<geometry_msgs::PointStamped>(
        "sensor_measurement/altitude", 1, true);
    aircraft_state.state = aerostack_msgs::FlightState::LANDED;
}
```

Listing 5.8: Setup function of the mavic_state interface.

```
void StateInterface::send_state_aerostack(){
    switch(command_aerostack.action){
        case aerostack_msgs::FlightActionCommand::TAKE_OFF:{
            if (aircraft_state.state ==
                aerostack_msgs::FlightState::LANDED ||
                aircraft_state.state ==
                aerostack_msgs::FlightState::UNKNOWN){
                aircraft_state.state =
                    aerostack_msgs::FlightState::TAKING_OFF;
                time_takeoff = ros::Time::now();
            }else{
                if (aircraft_state.state ==
                    aerostack_msgs::FlightState::TAKING_OFF){
                    ros::Duration diff = ros::Time::now() - time_takeoff;
                    if (std::abs(speed_msg.twist.linear.z) < 0.1 &&
                        std::abs(drone_altitude_msg.point.z) > 0.2 &&
                        diff.toSec() >= 5){
                        ROS_INFO("Flying");
                        aircraft_state.state =
                            aerostack_msgs::FlightState::FLYING;
        }}}}
        break;
        case aerostack_msgs::FlightActionCommand::HOVER:{
```

```
    if(std::abs(drone_altitude_msg.point.z) > 0.1 &&
        std::abs(speed_msg.twist.linear.x) < 0.1 &&
        std::abs(speed_msg.twist.linear.y) < 0.1 &&
        std::abs(speed_msg.twist.linear.z) < 0.1 &&
        std::abs(speed_msg.twist.angular.x) < 0.1 &&
        std::abs(speed_msg.twist.angular.y) < 0.1 &&
        std::abs(speed_msg.twist.angular.z) < 0.1){
        aircraft_state.state =
            aerostack_msgs::FlightState::HOVERING;
}}
break;
case aerostack_msgs::FlightActionCommand::LAND:{
    if (aircraft_state.state ==
        aerostack_msgs::FlightState::HOVERING ||
        aircraft_state.state ==
        aerostack_msgs::FlightState::FLYING){
        aircraft_state.state =
            aerostack_msgs::FlightState::LANDING;
    }else{
        if (aircraft_state.state ==
            aerostack_msgs::FlightState::LANDING){
            if (std::abs(drone_altitude_msg.point.z) == 0 &&
                std::abs(speed_msg.twist.linear.z < 0.1)){
                aircraft_state.state =
                    aerostack_msgs::FlightState::LANDED;
}}}}
break;
case aerostack_msgs::FlightActionCommand::MOVE:{
    ROS_INFO("%f %f %f %f %f", drone_altitude_msg.point.z,
        speed_msg.twist.linear.x, speed_msg.twist.linear.y,
        speed_msg.twist.linear.z, speed_msg.twist.angular.z);
    if(std::abs(drone_altitude_msg.point.z) > 0.2 &&
        (std::abs(speed_msg.twist.linear.x) > 0.05 ||
        std::abs(speed_msg.twist.linear.y) > 0.05 ||
        std::abs(speed_msg.twist.linear.z) > 0.05 ||
        std::abs(speed_msg.twist.angular.z) > 0.25)){
        aircraft_state.state =
            aerostack_msgs::FlightState::FLYING;
    } else if(std::abs(drone_altitude_msg.point.z) > 0.2 &&
        std::abs(speed_msg.twist.linear.x) < 0.05 &&
        std::abs(speed_msg.twist.linear.y) < 0.05 &&
        std::abs(speed_msg.twist.linear.z) < 0.05 &&
        std::abs(speed_msg.twist.angular.z) < 0.05){
        aircraft_state.state =
            aerostack_msgs::FlightState::HOVERING;
```

```
        }}
        break;
        case aerostack_msgs::FlightActionCommand::UNKNOWN: break;
        default:{
            if(std::abs(drone_altitude_msg.point.z) < 0.1 &&
                std::abs(speed_msg.twist.linear.x) < 0.1 &&
                std::abs(speed_msg.twist.linear.y) < 0.1 &&
                std::abs(speed_msg.twist.linear.z) < 0.1 &&
                std::abs(speed_msg.twist.angular.x) < 0.1 &&
                std::abs(speed_msg.twist.angular.y) < 0.1 &&
                std::abs(speed_msg.twist.angular.z) < 0.1){
                aircraft_state.state =
                    aerostack_msgs::FlightState::LANDED;
        }}
        break;
    }
    flight_state_pub.publish(aircraft_state);
}
```

Listing 5.9: Code associated with determining the vehicle flight state in the mavic_state interface.

Command Interface

The "mavic_command" interface is responsible for setting up the UAV for control through the navigation system and managing all the commands that are sent to the vehicle through the ROS interface running in the Android smartphone.

This interface counts with one ROS publisher, responsible for publishing the UAV command messages to the "Mavic_Commands" topic that is subscribed by the ROS interface, and six ROS subscribers, that are used to obtain the responses to the commands sent, motion reference speeds from both the autonomous flight software modules and the keyboard teleoperation, flight actions, the current vehicle pose, and motion reference poses, as seen in Listing 5.10.

The setup done by the interface allows to send a command message to the aircraft that defines the necessary control parameters, associated with how vertical control values, manual roll and pitch values, and manual yaw values are interpreted by the vehicle, as well as the flight control coordinate system that is used. This setup is necessary for operating the DJI Mavic 2 with the Mobile SDK tools and, in this implementation, the values used for these parameters established velocity mode for vertical control and roll-pitch control, angular velocity mode for yaw control, and a body coordinate system. The control parameters decisions were mostly based on compatibility with the default Aerostack control modes, but in the case of the yaw

```
resp_sub = n.subscribe("Responses", 1,
    &CommandInterface::Responses_Callback, this);
ref_speed_sub=n.subscribe("motion_reference/speed",1,
    &CommandInterface::ref_speed_callback,this);
ref_teleop_speed_sub=n.subscribe("motion_reference/speed_teleop",1,
    &CommandInterface::ref_teleop_speed_callback,this);
flight_action_sub = n.subscribe("actuator_command/flight_action",
    1, &CommandInterface::flightActionCallback, this);
current_pose_sub = n.subscribe("self_localization/pose", 1,
    &CommandInterface::self_localization_callback, this);
ref_pose_sub = n.subscribe("motion_reference/pose", 1,
    &CommandInterface::ref_pose_callback, this);
comm_pub=n.advertise<std_msgs::String>("Mavic_Commands",1, true);
```

Listing 5.10: ROS publishers and subscribers in the mavic_command interface.

control mode and the coordinate system, these options were used to minimize the effect of inertial drift errors by the vehicle sensor system.

Additionally, at startup, and before any action command is sent to the vehicle, it is necessary to send a command to activate the aircraft's joystick mode. This mode is necessary for controlling the vehicle, but when activated suspends the commands sent via the DJI remote controller. The joystick mode can be manually deactivated with the remote controller or using the SDK functions.

After the setup process, the declaration of necessary variables, and the activation of the joystick mode, the flight actions received are interpreted and the interface generates the command messages that correspond to those actions, either publishing the string messages associated with landing and taking off or publishing the string messages with the yaw angular velocity and linear velocities, in case of the hover or move flight actions. When the hover flight action is received, these velocities are set to null values.

Additionally, in order to easily terminate the joystick mode from the keyboard teleoperation user interface, it was defined that after more than two consecutive hover commands the joystick mode is deactivated, making the UAV hover in its current position until the vehicle controller is used or the joystick mode is reactivated, sending velocity commands through the keyboard teleoperation interface.

Regarding the move flight actions, this interface can operate in two distinct ways, sending keyboard teleoperation commands if there are any keyboard teleoperation motion reference speed messages being published or sending the other commands associated with the autonomous navigation if there are no keyboard teleoperation motion reference speed messages and there are new motion reference speed and pose messages published from Aerostack's behaviors. This ensures that the keyboard teleoperation has priority over the autonomous operation, which is done for safety

reasons, giving an operator the control and capability to stop and move away the UAV in case of unexpected, inadequate, or dangerous movements. The main function of the interface in Listing 5.11 shows the initial setup functions, the activation of the joystick mode, and the handling of these two operation modes.

```
int main(int argc,char **argv){
    ros::init(argc, argv, "CommandInterface");
    CommandInterface command_interface;
    command_interface.setUp();
    command_interface.start();
    ros::Rate loop_rate(30);

    while(ros::ok()){
        command_interface.run();
        //activate joystick mode on first velocity received
        if(command_interface.flag_first_velocity==1){
            //activate joystick mode
            command_interface.activate_joystick_mode(true);
            command_interface.flag_first_velocity=-1;
            ros::Duration(0.5).sleep();
        }
        //send velocities
        if(command_interface.teleop_flag_new_velocity){
            command_interface.teleop_send_velocity();
        }
        else if(command_interface.flag_new_velocity ||
            command_interface.flag_new_pose){
            command_interface.send_velocity();
        }
        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

Listing 5.11: Main function of the mavic_command interface.

The original keyboard teleoperation available through the Aerostack framework was adapted to output only linear and angular velocities, as the defined control modes of the vehicle require. However, these changes were not implemented to the behaviors already available. This is the reason why both speed and pose motion reference messages are received for autonomous operation. These speed messages relate to the commands for speed along the three axis and the pose messages relate to reference yaw values.

Regarding the motion reference pose messages that are used by the autonomous navigation behaviors to express the reference angle for aircraft yaw rotations, the interface uses a PID controller to control yaw angular velocity and achieve that reference yaw angle. The implementation of this controller is a straightforward standard implementation of a parallel PID controller, as seen in Listing 5.12.

```
void CommandInterface::PosePIDController(){
    if (flag_first_self_loc) { //if there is already self localization
        data
        if (r_first == true) { //first data
            r_previous_time = current_time;
            r_first = false;
        } else {
            r_time_diff = (current_time - r_previous_time).nsec / 1E9;
            //Rotation PID Control
            r_error = r_ref_yaw - current_yaw;
            //pick lower heading diff
            if (r_error > 180){
                r_error -= 360;
            }
            else if(r_error < -180){
                r_error += 360;
            };
            if (abs(r_error)>0.5 && flag_new_pose){
                double integral = r_integral_prior + r_error *
                    r_time_diff;
                double derivative = (r_error - r_error_prior);
                double output = r_Kp * r_error + r_Ki * integral + r_Kd
                    * derivative;
                r_error_prior = r_error;
                r_integral_prior = integral;
                r_previous_time = current_time;
                command_vel.twist.angular.z = output;
                ROS_INFO("YAW PID CONTROLLER\n-------------------\nGoal:
                    %f Error: %f Output: %f", r_ref_yaw, r_error,
                    output);
            }
            else{
                flag_new_pose=false;
            }
        }
    }
}
```

Listing 5.12: PID controller for yaw angle control in the mavic_command interface.

The angular velocity that is obtained from the PID control algorithm is published in the same message as the linear velocity obtained from the autonomous control behaviors. The commands that contain the angular and linear velocities from the autonomous behaviors and from the keyboard teleoperation are published in a similar way, differing the coordinate frame transform that is applied to each. This difference is justified by the fact that the velocity commands generated by the keyboard teleoperation and the autonomous behaviors are in two different coordinate frames and both need to be transformed to the coordinate frame in which the UAV interprets their values. For increasing the safety of the navigation system tests, the velocity command values were also limited. The process of selecting between keyboard teleoperation and autonomous navigation commands, applying the correspondent necessary transform, limiting the commands, and publishing them to the "Mavic_Commands" ROS topic is demonstrated in Listing 5.13.

5.2.3 Aerostack Framework Implementation for Mission Planning and Execution

Keyboard Teleoperation

In order to have a method to manually control the UAV through the ground-station computer, a keyboard teleoperation user interface was used. This interface was based on the package "keyboard_teleoperation_with_pid_control" from the Aerostack framework, which was modified to be compatible with the UAV's control modes and coordinate system.

The keyboard teleoperation package modifications were made so that the z-axis control affects the vertical speed directly, instead of the reference altitude, the yaw control affects yaw angular speed, instead of yaw reference angle, and in order to correctly affect the speed in the x and y axes, in accordance with the vehicle coordinate frame. Fig. 5.4 shows the difference between the user interfaces in the original keyboard teleoperation and the modified version used in this implementation.

The modified user interface allows an operator to manually command take-off, landing, and hovering actions, execute emergency stops, control the vehicle linear velocity, and control the vehicle yaw angular speed.

```cpp
void CommandInterface::doTransform_andSend(){
    geometry_msgs::TransformStamped velocity_transform;
    try{
        geometry_msgs::TwistStamped command;
        if (teleop_mode){
            velocity_transform =
                tf_buffer.lookupTransform("drone0_velocity",
                "drone0_optimized", ros::Time(0));
            command = teleop_command_vel; teleop_mode = false;
        }else{
            velocity_transform =
                tf_buffer.lookupTransform("drone0_velocity",
                "drone0_ned", ros::Time(0));
            command = command_vel;
        }
        geometry_msgs::Point xyz_velocity_trasform;
        xyz_velocity_trasform.x=command.twist.linear.x;
        xyz_velocity_trasform.y=command.twist.linear.y;
        xyz_velocity_trasform.z=command.twist.linear.z;
        double angular_yaw=command.twist.angular.z;
        tf2::doTransform(xyz_velocity_trasform, xyz_velocity_trasform,
            velocity_transform);

        if(xyz_velocity_trasform.x>0.4) xyz_velocity_trasform.x=0.4;
        else if(xyz_velocity_trasform.x<-0.4)
            xyz_velocity_trasform.x=-0.4;
        if(xyz_velocity_trasform.y>0.4) xyz_velocity_trasform.y=0.4;
        else if(xyz_velocity_trasform.y<-0.4)
            xyz_velocity_trasform.y=-0.4;
        if(xyz_velocity_trasform.z<-0.4) xyz_velocity_trasform.z=-0.4;
        else if(xyz_velocity_trasform.z>0.4)
            xyz_velocity_trasform.z=0.4;
        if(angular_yaw>60) angular_yaw=60;
        else if(angular_yaw<-60) angular_yaw=-60;

        geometry_msgs::Quaternion velocity_send;
        velocity_send.x=xyz_velocity_trasform.x;
        velocity_send.y=xyz_velocity_trasform.y;
        velocity_send.z=xyz_velocity_trasform.z;
        velocity_send.w=angular_yaw;
        std_msgs::String message;
        message.data = std::string ("Velocity ") +
            std::to_string(velocity_send.x) + std::string(" ") +
            std::to_string(velocity_send.y) + std::string(" ") +
            std::to_string(velocity_send.z) + std::string(" ") +
            std::to_string(velocity_send.w);
        comm_pub.publish(message);
    }
    catch (tf2::TransformException &ex) ros::Duration(0.01).sleep();
}
```

Listing 5.13: Function responsible for publishing the velocity commands in the mavic_command interface.

(a) Original version

(b) Modified version

Figure 5.4: Comparison between the user interfaces for the original version of the keyboard teleoperation package (a) and the modified version used in the new implementation (b).

Behavior Catalog

The Aerostack framework already has most of the behaviors necessary for a standard UAV navigation system implementation. In our navigation system, various behaviors from the Aerostack library were used to implement most UAV movement actions. Additionally, it was also necessary to develop two new behaviors. Considering this, the catalog of behaviors that was used in the navigation system was based on the following packages:

- **basic_quadrotor_behaviors**

  This Aerostack package was used for executing the most basic tasks in the UAV missions, through the behaviors:

  - TAKE_OFF - goal-based behavior that publishes a homonymous flight action command responsible for making the vehicle lift-off vertically and makes the vehicle hover when this goal is achieved;
  - LAND - goal-based behavior that publishes a homonymous flight action command responsible for landing the vehicle vertically on the ground, below its current position;
  - HOVER - recurrent behavior that publishes a homonymous flight action command that locks the vehicle in its current pose, with null horizontal velocity, constant heading, and constant altitude;
  - WAIT - goal-based behavior that achieves its goal after a certain duration, previously defined in the mission behavior tree.

- **quadrotor_motion_with_pid_control**

  Being also part of the original Aerostack framework, this package contains more advanced behaviors, from which the following two were used in this project's missions:

  - ROTATE - goal-based behavior that publishes a motion reference pose that allows the vehicle to change its heading to a predetermined yaw angle, while hovering;
  - FOLLOW_PATH - goal-based behavior that allows the vehicle to move between a predetermined group of three-dimensional points, using a path tracker subpackage.

  In the case of the FOLLOW_PATH behavior, the path tracker subpackage is used to publish the motion reference velocities that allow the vehicle to move between the various points in the path.

  Although this package normally uses another subpackage for PID control of the UAV velocity based on the motion references published by the behaviors, in our implementation this was not used, due to this system's UAV receiving velocity commands directly and controlling its velocity efficiently.

- **mavic_behaviors**

  This package is not part of the original Aerostack framework and was implemented to take advantage of particular features of the DJI Mavic 2. The two behaviors developed for this package were:

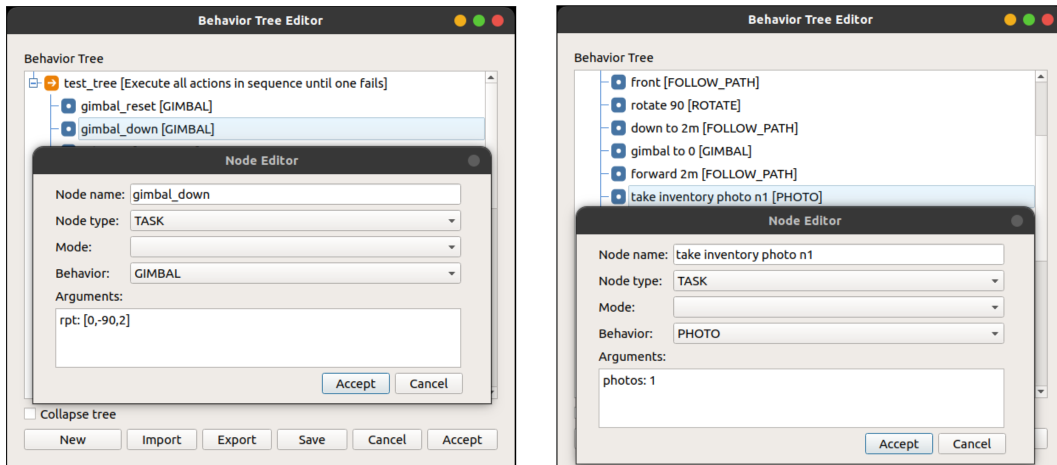
  - GIMBAL - goal-based behavior implemented in order to control the UAV's main camera gimbal during missions. The behavior receives a three-dimensional point as a parameter named rpt, with coordinates that respectively indicate a relative rotation angle in roll, a relative rotation angle in pitch, and the duration for the rotation movement. The command for this gimbal rotation is directly published as a message in the Mavic_Command ROS topic.

    To confirm the achievement of the behavior goal, a ROS subscriber aquires the current gimbal pose, which allows to compare the desired pose with the current one and determine if the action was completed. If the action exceeds a certain timeout duration and the goal was not achieved, the behavior fails.
  - PHOTO - goal-based behavior implemented in order to allow taking a predetermined number of photos with the main camera during the UAV's missions. The command message is directly published in the Mavic_Command topic, the photos are stored in the UAV's internal memory, and the behavior always succeeds.

Behavior Tree Mission Editor

The implementation of this project relied on the usage of the Aerostack's behavior tree editor, which offers an intuitive way for the user to define the missions' behavior trees. In our tests, this tool was used to define the mission plan, represented by a behavior tree.

The editor allowed to use the original Aerostack behaviors, as well as the ones developed specifically for our system. In the case of the two new behaviors, an example of their usage in the editor is shown in Fig. 5.5.



(a) GIMBAL behavior (b) PHOTO behavior

Figure 5.5: Usage of the two developed behaviors in the Aerostack behavior tree editor.

5.3 Test Conditions and Mission Plan

Regarding the tests made to evaluate the navigation system, these were conducted in our laboratory, using printed AprilTag markers taped on the floor and on two boards, that represent inventory racks in our tests.

As for the ground-station computer and mobile device that were used during the tests, these were respectively a laptop with an Intel Core i7-1065G7 CPU @ 1.30 GHz, 16 GB of 3733 MHz RAM, an NVIDIA GeForce MX250 GPU, and running Ubuntu 18.04 LTS with ROS Melodic and a smartphone with a Samsung Exynos 990 CPU, 8GB of LPDDR5 RAM, and running Android 11.

The test environment at the laboratory was captured with millimeter accuracy in a three-dimensional point cloud obtained with a FARO Focus 3D S120 laser scanner. This point cloud, which can be seen represented in a RViz [52] environment in Fig. 5.6, allowed to compare the motion of the UAV in the real world with the motion in a virtual world, which facilitates the visualization of mission and behaviors execution.

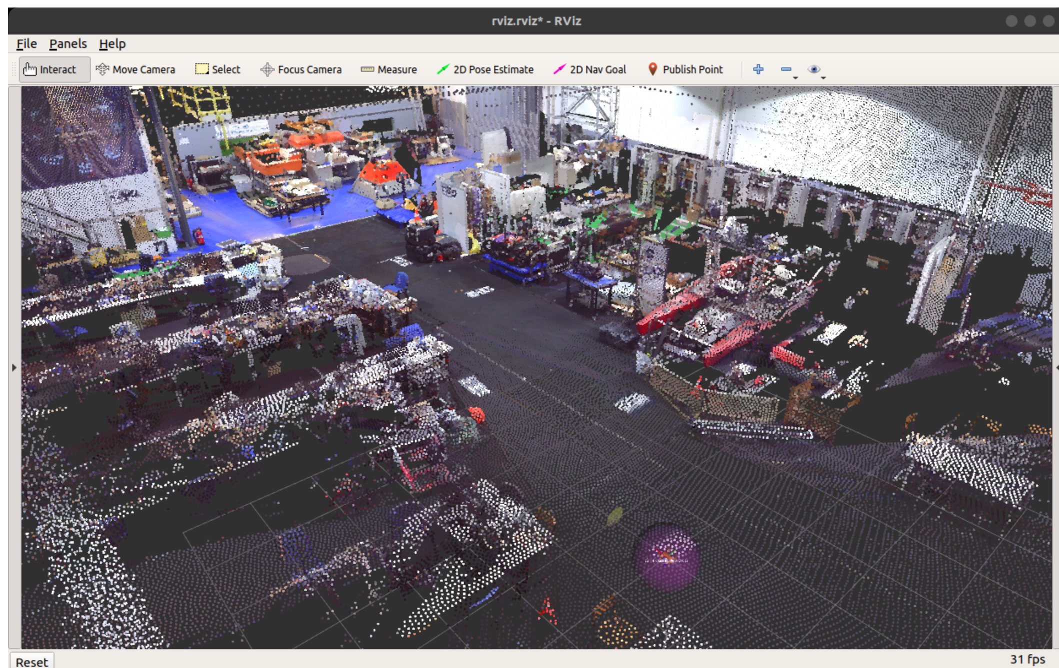


Figure 5.6: 3D point cloud of the test environment in RViz.

Considering the test conditions described, the implemented mission defined a path that allowed the vehicle to position itself in front of the two boards that represent inventory racks and take a photo of each board, going back to its take-off spot after finishing the mission. The various behaviors that allowed to implement this mission are present in the mission plan in Fig. 5.7.

As for the path that was defined in the mission plan, its representation on the three-dimensional map of the environment is shown in Fig. 5.8, divided in two parts,

respectively the path for taking the two photos and the path for going back to the take-off location.

In addition to the behaviors that allowed to follow this path, the mission plan has the take-off and land behaviors, rotation behaviors, that ensure that the vehicle is facing forward in the path direction, gimbal behaviors, that rotate the gimbal along the path so that the visual markers are seen by the camera in an optimized way, photo behaviors, used to take the photos of the two boards, and wait behaviors, to stop the vehicle for a short time in front of the boards, allowing for better photos and to see this step of the mission more clearly in video.

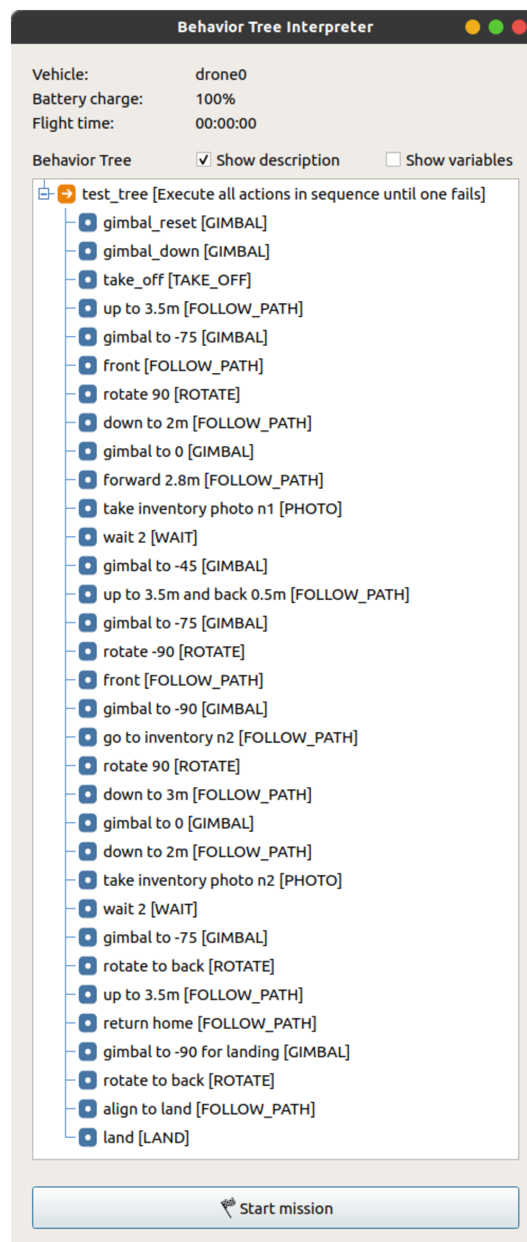


Figure 5.7: Complete behavior tree of the mission plan.

Figure 5.8: Trajectory defined in the behavior tree mission plan.

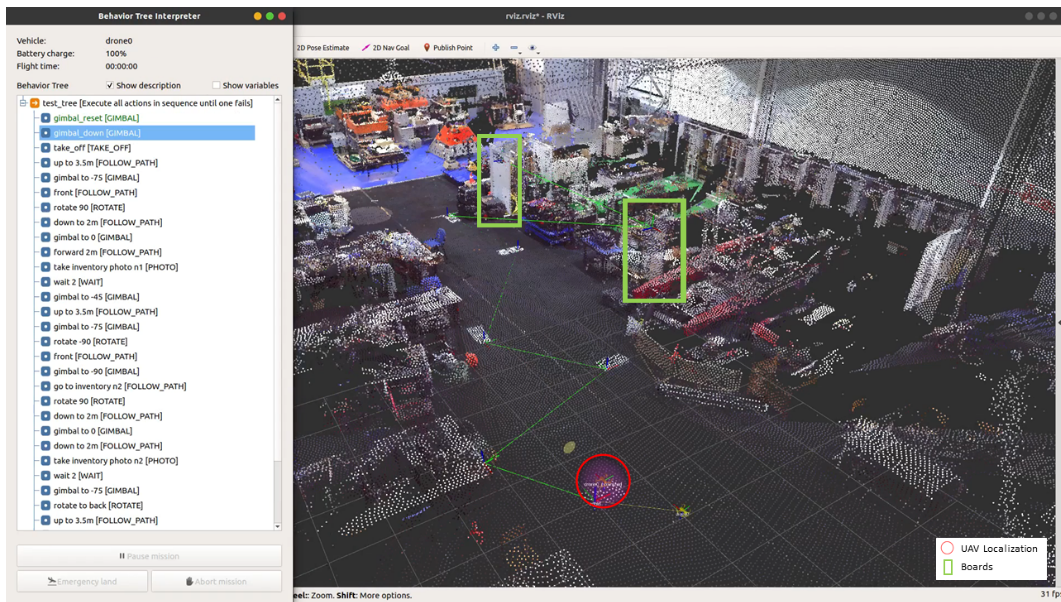
Chapter 6

Results

Considering the test conditions and the mission plan disclosed in Section 5.2.3, the implemented navigation system was evaluated by executing the mission in our laboratory. The results shown in this chapter were obtained during mission execution, from video recordings of the mission and the ground station computer screen and from rosbag [53] logs of the vehicle sensor data and the navigation algorithms data.

Before mission execution, since no map of the visual markers is used in the localization, the drone was manually flown across the test environment, in order to detect the AprilTag markers and initialize the factor graph with relative location estimations of their poses. This procedure is important for obtaining a good localization performance, specially in short missions, such as the one that was tested.

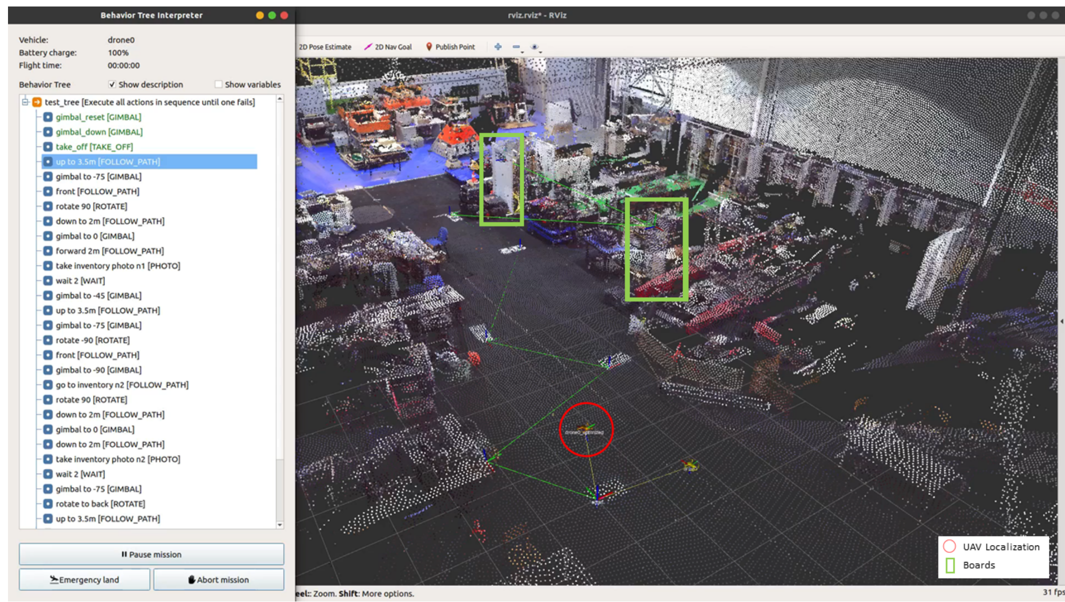
Fig. 6.1–6.6 show frames obtained from the recordings of both the ground station computer screen and the external video of the test environment in various moments of the mission that was executed according to the plan in 5.7, namely the beginning and ending of the mission and the instants after take-off, after sending the commands for the two photos, and after landing. In these frames, where the red circle identifies the UAV and the green rectangles identify the two boards used for representing inventory racks, it is noticeable that the real-world location of the vehicle is consistent with the behaviors being executed and, to some extent, that the UAV optimized pose shown in Rviz is representative of the real pose of the vehicle in these key moments of the mission.



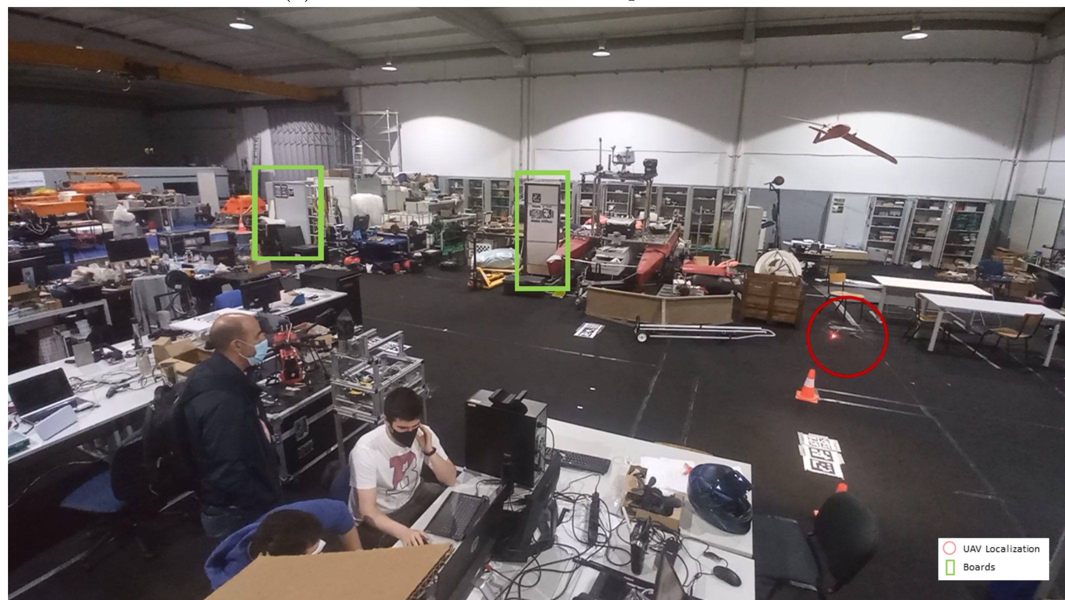
(a) RViz and behavior tree interpreter interface

(b) Frame from mission test video

Figure 6.1: UAV at the beginning of mission execution.

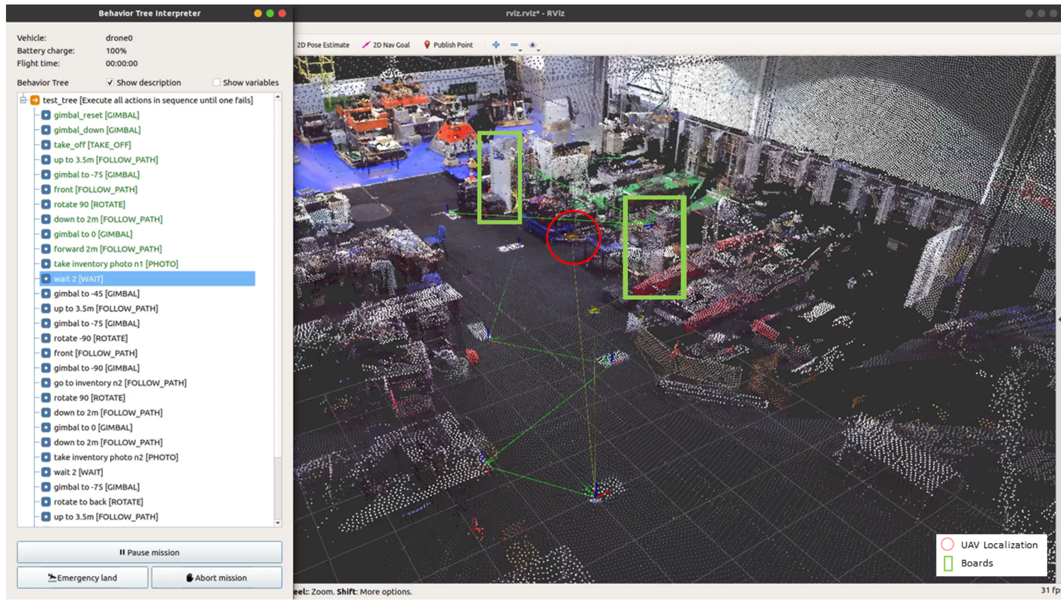


(a) RViz and behavior tree interpreter interface



(b) Frame from mission test video

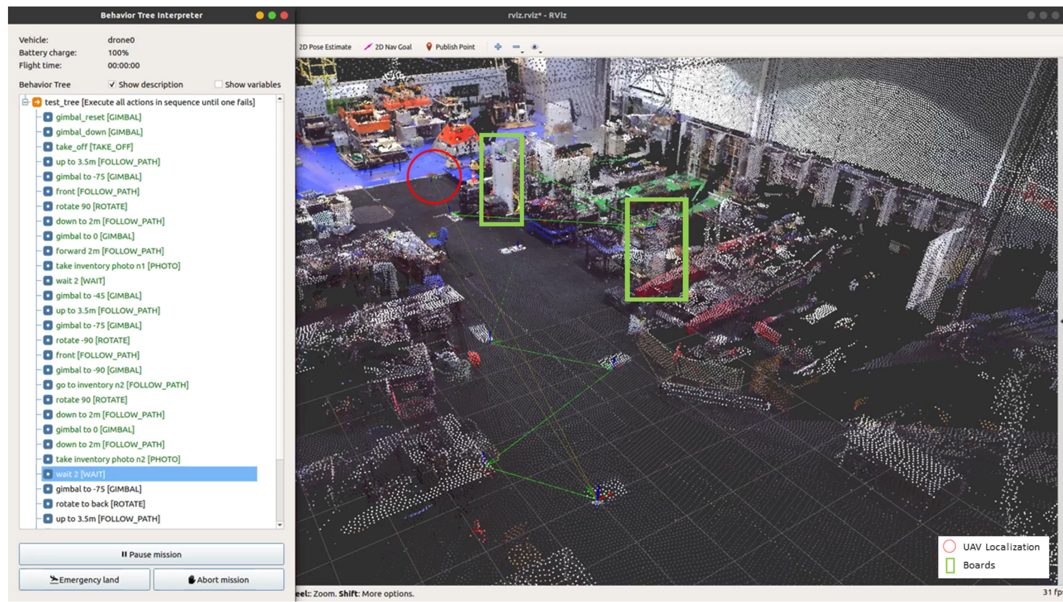
Figure 6.2: UAV after the take-off behavior execution.



(a) RViz and behavior tree interpreter interface

(b) Frame from mission test video

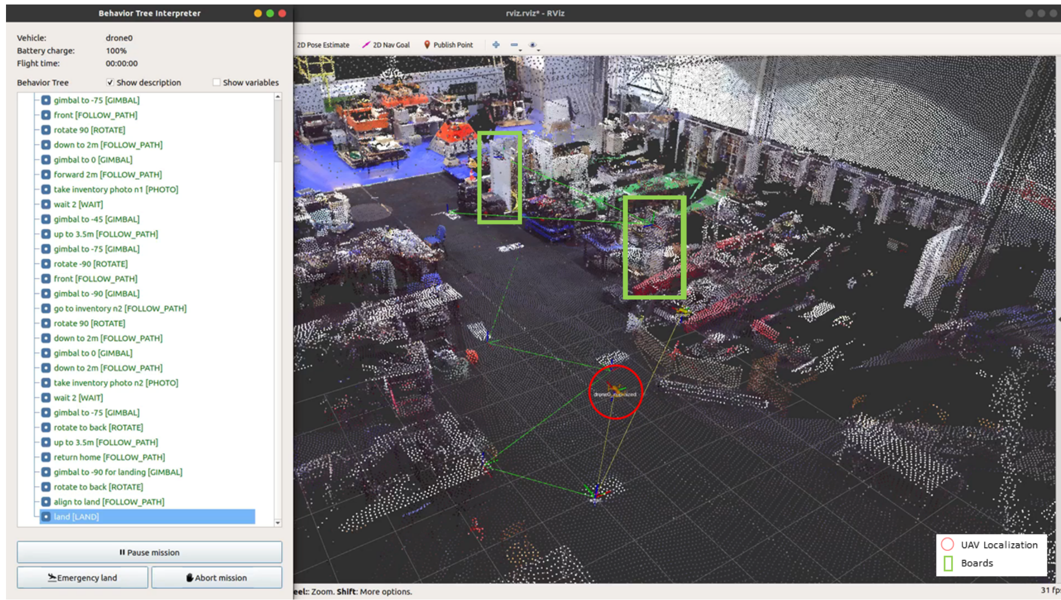
Figure 6.3: UAV capturing the photo of the first board.



(a) RViz and behavior tree interpreter interface

(b) Frame from mission test video

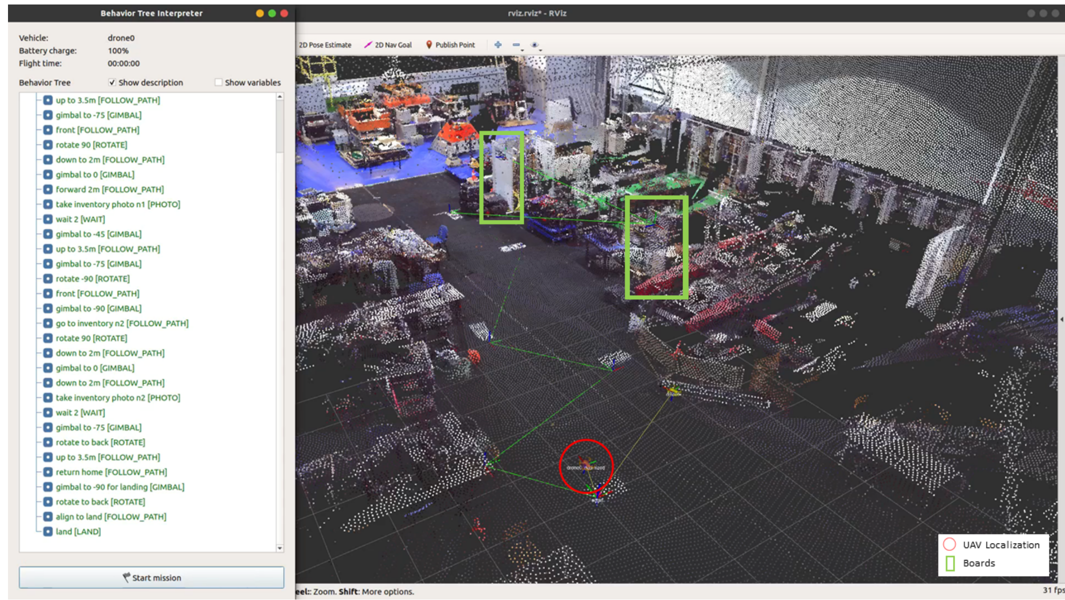
Figure 6.4: UAV capturing the photo of the second board.



(a) RViz and behavior tree interpreter interface

(b) Frame from mission test video

Figure 6.5: UAV beginning the landing behavior execution.



(a) RViz and behavior tree interpreter interface

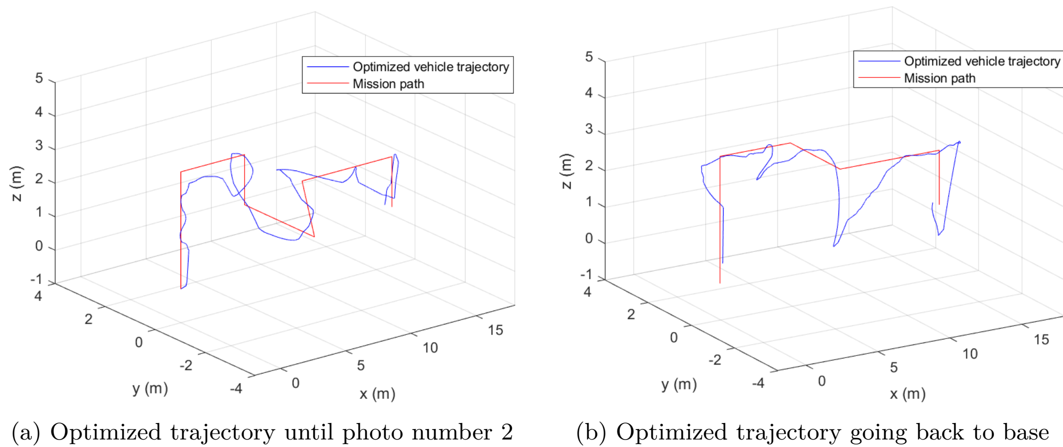
(b) Frame from mission test video

Figure 6.6: UAV at the end of mission execution.

Looking in more detail at the vehicle trajectory during the mission, and using the straight lines path between the 3D points specified in the mission plan as reference, the localization algorithm's optimized trajectory at the end of the mission, as seen in Fig. 6.7, shows that the main navigation objectives of the mission were achieved. However, it is also noticeable that at certain times the vehicle deviates significantly from the defined path, having abrupt changes to recover from those deviations.

The deviations from the defined path happen due to the nature of the localization algorithm, that relies on inertial sensor data for localization when no visual marker is detected, and due to the low resolution of the UAV velocity data of only 0.1 m/s, that limits the performance of the localization algorithm when no markers are being detected.

After taking the second photo, and before going back to base, there are two noticeable descents, one at the beginning of the trajectory going back to base and one on the first turn after. In the first case, the path following behavior was commanding a small velocity in the z-axis which was not detected due to the fact that no visual marker was being seen in the camera image and the resolution of the inertial measurements wasn't enough to detect this change but a short time after the visual marker was seen and the vehicle increased its altitude to continue following the right path. In the second case, the UAV executed a long straight line relying only on inertial measurements and with a small downwards velocity that wasn't being detected, which resulted in a decrease of altitude during the straight line that was then fixed when the visual marker was detected at the end of that line.



(a) Optimized trajectory until photo number 2 (b) Optimized trajectory going back to base

Figure 6.7: Comparison between the localization algorithm's optimized trajectory and the defined mission path.

As for the control of the gimbal during the mission, the graph in Fig. 6.8 shows the gimbal pitch angle during mission execution and the time of each GIMBAL behavior that commands a rotation for changing this angle. The results in this graph show that the commands that were sent to change the gimbal orientation

quickly affected the gimbal pitch and the angles achieved were in accordance with the values defined in the mission plan.

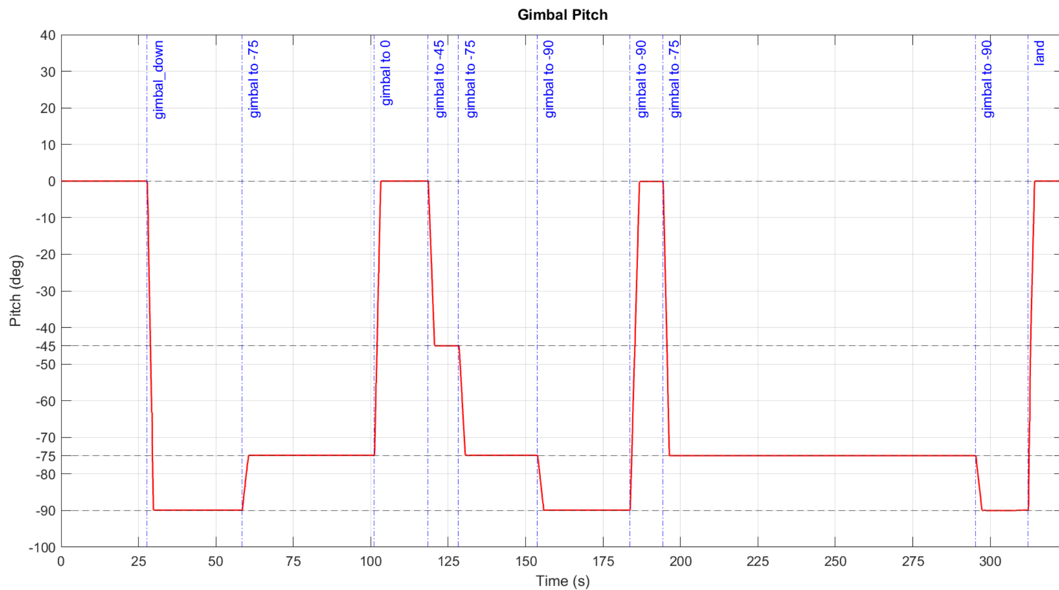


Figure 6.8: Main camera gimbal pitch during mission execution.

The graph in Fig. 6.9 shows the curve of the vehicle heading during mission execution, where it can be seen the effect of the various 90° rotations and the final 180° degree rotation before landing, that were commanded using rotation behaviors in the mission plan. The yaw angle curve shows that these behaviors achieved the desired rotations and the PID controller used for controlling the rotation allowed for quick and effective changes in vehicle heading.

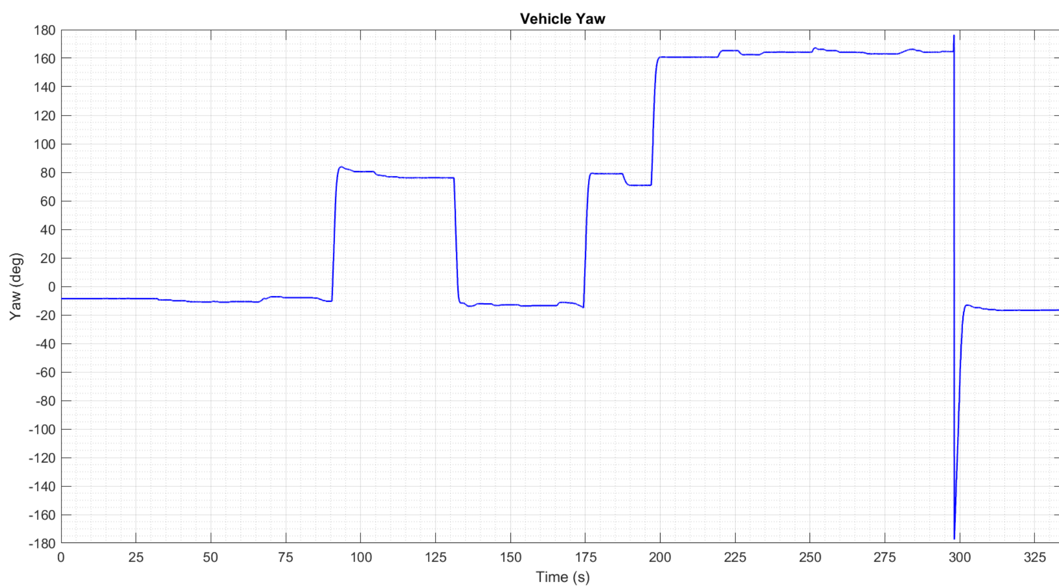
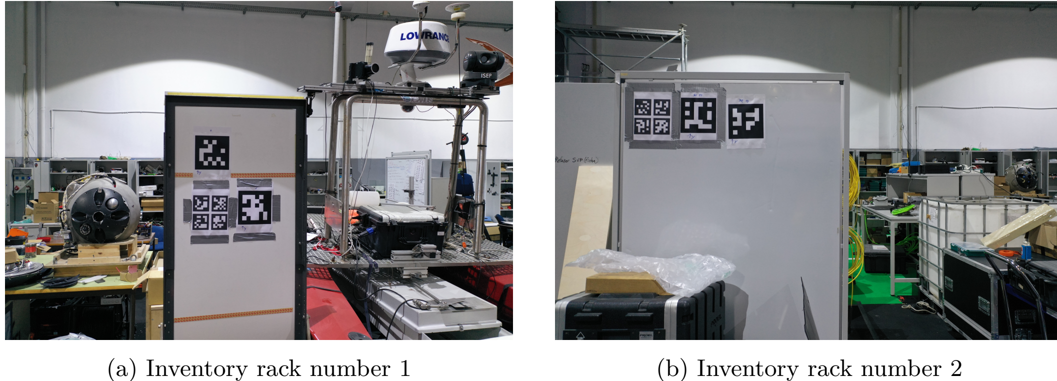


Figure 6.9: Vehicle yaw during mission execution.

Regarding the photos of the two boards, that were taken using the PHOTO behaviors during the mission, these are shown in Fig. 6.10 and demonstrate that the two boards were captured in their respective photos, as intended. These results show that in real warehouse scenarios, the inventory in the inventory racks could be photographed using the behavior in this way.



(a) Inventory rack number 1

(b) Inventory rack number 2

Figure 6.10: Photos captured during mission execution for the two boards representing inventory racks.

As for the duration of the mission, the total execution of the behavior tree and the mission completion were achieved after a little less than 5 minutes. However, this time could be greatly improved in the future, as the vehicle's velocity was heavily limited in order to test the system in safe conditions.

Considering all these results, it was determined that the system that was implemented was capable of taking off, follow predetermined paths, rotate in place a certain number of degrees, rotate the camera gimbal, take photos, waiting in place, and landing, with all of the mission plan behaviors being successfully executed and all its objectives being achieved.

Chapter 7

Conclusion

This thesis aims to support the usage of UAVs in indoor warehouses inventory management tasks, through the development of a navigation system compatible with commercial-grade platforms, in order to ensure an easy, fast, and cost-efficient implementation across multiple warehouses.

After the selection of a commercial platform, the development of a dedicated ROS interface, and the implementation of all software modules in the ground-station computer, namely the localization, mission execution, and path planning modules, the work done resulted in a system that follows the requirements presented and is compatible with this thesis' objectives.

Considering the results obtained in the test and evaluation of this navigation system, it was concluded that the implemented system is capable of navigating in indoor environments, following mission plans declared in the form of behavior trees involving take-off and landing actions, path following, heading rotations, control of the main camera gimbal, and taking photos of inventory.

However, the test results also demonstrated some problems with the localization approach, that relies on inertial sensor data and the detection of visual markers. The low resolution of the UAV velocity data outputs restricted the path following capabilities of the system, as the absence of visual marker detections in some parts of the specified path were responsible for significant deviations that were only corrected after detection of new visual markers.

Overall, this implementation supported the possibility of using UAVs for remotely accessing inventory in indoor warehouse environments and performing inventory management tasks in a safe and efficient way.

This thesis' contributions are associated with the presentation of a system that uses an inexpensive UAV platform, is able to safely and efficiently navigate indoor environments such as warehouses, can be easily implemented in different environments as no map is required, offers an intuitive interface for declaring mission plans in the form of behavior trees, and is able accomplish those missions successfully, executing a vast group of behaviors that can form very complex missions.

7.1 Future Work

Regarding future work, the localization approach is one of the topics that can be improved in the future. Replacing or adapting the method for obtaining inertial data, in particular velocity data, can greatly improve the overall system performance for following paths. Additionally, increasing the quantity of visual markers available in the environment could also improve the performance of the localization estimations, also affecting positively the path following performance of the system.

It is also possible to further optimize the path following algorithms and increase the speed limits of the system in a safe way, which would result in more efficient mission execution.

Finally, the addition of new behaviors could improve the capabilities of the system. In particular, implementing behaviors for counting inventory or avoiding obstacles in path following during missions would significantly affect the applicability of the system, and possibly making it ready for implementation in real-world scenarios.

References

[1] T. M. Fernández-Caramés, O. Blanco-Novoa, I. Froiz-Míguez, and P. Fraga-Lamas, "Towards an Autonomous Industry 4.0 Warehouse: A UAV and Blockchain-Based System for Inventory and Traceability Applications in Big Data-Driven Supply Chain Management," *Sensors*, vol. 19, May 2019. [Cited on pages 1 and 7]

[2] L. Wawrla, O. Maghazei, and T. Netland, "Applications of drones in warehouse operations," tech. rep., ETH Zurich, Zurich, Switzerland, Aug. 2019. [Cited on page 2]

[3] A. Moura, J. Antunes, A. Dias, A. Martins, and J. Almeida, "Graph-SLAM Approach for Indoor UAV Localization in Warehouse Logistics Applications," in *2021 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pp. 4–11, 2021. [Cited on pages 2, 3, 19, 20, and 23]

[4] J. Antunes, "Graph-SLAM Approach for Indoor UAV Localization in Warehouse Logistics Applications," Master's thesis, Polytechnic Inst. of Porto, 2021. [Cited on page 2]

[5] Y. Lu, Z. Xue, G.-S. Xia, and L. Zhang, "A survey on vision-based uav navigation," *Geo-spatial Information Science*, vol. 21, no. 1, pp. 21–32, 2018. [Cited on page 5]

[6] J. Duo and L. Zhao, "UAV autonomous navigation system for GNSS invalidation," in *2017 36th Chinese Control Conference (CCC)*, pp. 5777–5782, 2017. [Cited on page 5]

[7] S. Weiss, D. Scaramuzza, and R. Siegwart, "Monocular-SLAM-Based Navigation for Autonomous Micro Helicopters in GPS-Denied Environments," *J. Field Robotics*, vol. 28, pp. 854–874, 11 2011. [Cited on page 6]

[8] M. W. Achtelik, S. Lynen, S. Weiss, L. Kneip, M. Chli, and R. Siegwart, "Visual-inertial slam for a small helicopter in large outdoor environments," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2651–2652, 2012. [Cited on page 6]

[9] G. Balamurugan, J. Valarmathi, and V. P. S. Naidu, "Survey on uav navigation in gps denied environments," in *2016 International Conference on Signal*

Processing, Communication, Power and Embedded System (SCOPES), pp. 198–204, 2016. [Cited on page 6]

[10] doks., “inventAIRy XL: an innovative technology.” Available at `https://doks-innovation.com/download-area`. (Last accessed in 30/10/2021). [Cited on page 6]

[11] Dexion, “Dexion | Infinium Scan Drone.” Available at `https://www.dexion.co.nz/wp-content/uploads/2018/04/Dexion-Infinium-Scan-Drone-Spec-Sheet-FINAL.pdf`. (Last accessed in 30/10/2021). [Cited on page 6]

[12] FlytWare, “FlytWare – Autonomous Drones for Warehouse Inventory.” Available at `https://flytware.com/`. (Last accessed in 30/10/2021). [Cited on page 6]

[13] Eyesee, “Eyesee, the inventory drone solution.” Available at `https://eyesee-drone.com/eyesee-the-inventory-drone-solution`. (Last accessed in 30/10/2021). [Cited on page 6]

[14] Ware, “Ware | Drones For Warehouse Automation.” Available at `https://www.ware.ai/`. (Last accessed in 30/10/2021). [Cited on page 6]

[15] E. Anglada, R. Soria, and G. Solà, “Drones en la fábrica del futuro.” Available at `https://www.seat-mediacenter.es/smc/seat-sa/seat-sa-storiespage/Drones-en-la-fabrica-del-futuro.html#`, 2021. (Last accessed in 27/10/2021). [Cited on page 6]

[16] I. Kalinov, A. Petrovsky, V. Ilin, E. Pristanskiy, M. Kurenkov, V. Ramzhaev, I. Idrisov, and D. Tsetserukou, “WareVision: CNN Barcode Detection-Based UAV Trajectory Optimization for Autonomous Warehouse Stocktaking,” *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6647–6653, 2020. [Cited on page 6]

[17] E. H. C. Harik, F. Guérin, F. Guinand, J.-F. Brethé, and H. Pelvillain, “Towards an autonomous warehouse inventory scheme,” in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–8, 2016. [Cited on page 7]

[18] W. Kwon, J. H. Park, M. Lee, J. Her, S.-H. Kim, and J.-W. Seo, “Robust Autonomous Navigation of Unmanned Aerial Vehicles (UAVs) for Warehouses’ Inventory Application,” *IEEE Robotics and Automation Letters*, vol. 5, no. 1, pp. 243–249, 2020. [Cited on page 7]

[19] C. Li, E. Tanghe, P. Suanet, D. Plets, J. Hoebeke, E. De Poorter, and W. Joseph, “ReLoc 2.0: UHF-RFID Relative Localization for Drone-Based

Inventory Management," *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–13, 2021. [Cited on page 7]

[20] M. Beul, D. Droeschel, M. Nieuwenhuisen, J. Quenzel, S. Houben, and S. Behnke, "Fast Autonomous Flight in Warehouses for Inventory Applications," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3121–3128, 2018. [Cited on page 7]

[21] M. Yannakakis, "Hierarchical State Machines," in *IFIP International Conference on Theoretical Computer Science*, vol. 1872, pp. 315–330, Jan. 2000. [Cited on page 8]

[22] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, June 1987. [Cited on page 8]

[23] J. Bohren and S. Cousins, "The SMACH High-Level Executive [ROS News]," *Robotics & Automation Magazine, IEEE*, vol. 17, p. 17, Dec. 2010. [Cited on page 8]

[24] F. Kolbe, *Goal Oriented Task Planning for Autonomous Service Robots*. PhD thesis, Hamburg University of Applied Sciences, Hamburg, Germany, Nov. 2013. [Cited on page 8]

[25] J. Bohren and W. Meeussen, "executive_smach - ROS Wiki." Available at `http://wiki.ros.org/executive_smach`. (Last accessed in 30/10/2021). [Cited on page 8]

[26] J. Millan-Romera, H. Perez-Leon, A. Castillejo-Calle, I. Maza, and A. Ollero, "ROS-MAGNA, a ROS-based framework for the definition and management of multi-UAS cooperative missions," in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, (Atlanta, USA), pp. 1477–1486, June 2019. [Cited on page 8]

[27] C. Pradalier, "A task scheduler for ROS," research report, UMI 2958 GeorgiaTech-CNRS, Jan. 2017. ROBOTICS. [Cited on page 8]

[28] ReelRobotix, "SMACC [Source code]." Available at `https://github.com/reelrbtx/SMACC`, 2019. (Last accessed in 30/10/2021). [Cited on page 8]

[29] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," *IEEE Transactions on Robotics*, vol. 33, pp. 372–389, April 2017. [Cited on page 8]

[30] J. A. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, M. Pivtoraiko, J.-S. Valois,

and R. Zhu, "An integrated system for autonomous robotics manipulation," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2955–2962, 2012. [Cited on page 8]

[31] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. Chapman & Hall/CRC Artificial Intelligence and Robotics Series, CRC Press, 2018. [Cited on page 8]

[32] M. Colledanchise, "ROS-Behavior-Tree [Source code]." Available at `https://github.com/miccol/ROS-Behavior-Tree`. (Last accessed in 30/10/2021). [Cited on page 9]

[33] D. Faconti, "Groot [Source code]." Available at `https://github.com/BehaviorTree/Groot`. (Last accessed in 17/08/2021). [Cited on page 9]

[34] D. Faconti, "BehaviorTree.CPP." Available at `https://www.behaviortree.dev/`. (Last accessed in 30/10/2021). [Cited on page 9]

[35] J. L. Sanchez-Lopez, M. Molina, H. Bavle, C. Sampedro, R. A. S. Fernández, and P. Campoy, "A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems: The Aerostack Framework," *Journal of Intelligent & Robotic Systems*, vol. 88, no. 2-4, pp. 683–709, 2017. [Cited on pages 9 and 14]

[36] A. Hentout, A. Maoudj, and B. Bouzouia, "A survey of development frameworks for robotics," in *2016 8th International Conference on Modelling, Identification and Control (ICMIC)*, pp. 67–72, 2016. [Cited on page 11]

[37] Open Robotics, "ROS/Introduction -ROS Wiki." Available at `http://wiki.ros.org/ROS/Introduction`. (Last accessed in 30/10/2021). [Cited on page 11]

[38] J. M. O'Kane, *A Gentle Introduction to ROS*. Independently published, Oct. 2013. [Cited on page 12]

[39] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," vol. 3, 01 2009. [Cited on page 13]

[40] S. Feng, J. Xi, C. Gong, J. Gong, S. Hu, and Y. Ma, "A Collaborative Decision Making Approach for Multi-Unmanned Combat Vehicles based on the Behaviour Tree," in *2020 3rd International Conference on Unmanned Systems (ICUS)*, pp. 395–400, 2020. [Cited on page 13]

[41] A. R. Perales, "Arquitectura de Componentes Interoperables para Control de Movimiento de Robots Aéreos," Master's thesis, Universidad Politécnica de Madrid, 2020. [Cited on page 14]

[42] Computer Vision and Aerial Robotics - Universidad Politécnica de Madrid, "Aerostack Wiki." Available at https://github.com/cvar-upm/aerostack/wiki, 2020. (Last accessed in 30/10/2021). [Cited on pages 15, 16, and 21]

[43] M. Molina, A. Carrera, A. Camporredondo, H. Bavle, A. Rodriguez-Ramos, and P. Campoy, "Building the executive system of autonomous aerial robots using the Aerostack open-source framework," *International Journal of Advanced Robotic Systems*, vol. 17, 2020. [Cited on page 16]

[44] M. A. Johnson and M. H. Moradi, *PID control: New identification and design methods*. Springer, 2005. [Cited on page 18]

[45] D. Malyuta, "apriltag_ros - ROS Wiki." Available at http://wiki.ros.org/apriltag_ros. (Last accessed in 30/10/2021). [Cited on page 21]

[46] F. Dellaert, "Factor Graphs and GTSAM: A Hands-on Introduction," sep 2012. [Cited on page 21]

[47] "rosjava - ROS Wiki." Available at http://wiki.ros.org/rosjava. (Last accessed in 30/10/2021). [Cited on page 27]

[48] "android - ROS Wiki." Available at http://wiki.ros.org/android. (Last accessed in 30/10/2021). [Cited on page 27]

[49] OpenCV, "OpenCV: Open Source Computer Vision Library [Source code]." Available at https://github.com/opencv/opencv, 2019. (Last accessed in 30/10/2021). [Cited on page 34]

[50] P. Mihelich and J. Bowman, "cv_bridge - ROS Wiki." Available at http://wiki.ros.org/cv_bridge. (Last accessed in 30/10/2021). [Cited on page 34]

[51] P. Mihelich, K. Konolige, and J. Leibs, "image_proc - ROS Wiki." Available at http://wiki.ros.org/image_proc. (Last accessed in 30/10/2021). [Cited on page 35]

[52] D. Hershberger, D. Gossow, J. Faust, and W. Woodall, "rviz - ROS Wiki." Available at http://wiki.ros.org/rviz. (Last accessed in 30/10/2021). [Cited on page 48]

[53] T. Field, J. Leibs, J. Bowman, and D. Thomas, "rosbag - ROS Wiki." Available at http://wiki.ros.org/rosbag. (Last accessed in 30/10/2021). [Cited on page 51]