INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Graph-SLAM Approach for Indoor UAV Localization in Warehouse Logistics Applications

JOSÉ FILIPE DA SILVA OLIVEIRA ANTUNES
novembro de 2021

POLITÉCNICO
DO PORTO

Graph-SLAM Approach for Indoor UAV Localization in Warehouse Logistics Applications

José Filipe da Silva Oliveira Antunes
Nº 1161055

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Sistemas Autónomos
Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

2021

Dissertação, para satisfação parcial dos requisitos do Mestrado em Engenharia Eletrotécnica e de Computadores

Candidato: José Filipe da Silva Oliveira Antunes

Nº 1161055

Orientador: André Miguel Pinheiro Dias

Mestrado em Engenharia Eletrotécnica e de Computadores

Área de Especialização de Sistemas Autónomos

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

November 19, 2021

Abstract

In the context of Industry 4.0, warehouses are transforming and new solutions are allowing more warehouse tasks to be executed in autonomous ways.

The use of robots is a great advantage and a great addition to the autonomy of such structures since they can help with the automatic cyclic counting of the inventory, the localization of hard-to-find items, and the access to narrow or high storage areas. In this context, the usage of an Unmanned Aerial Vehicle (UAV) becomes a requirement in this type of operations.

Nevertheless, using this type of robots inside a warehouse comes with great engineering challenges such as indoor autonomous localization and navigation, collision avoidance, and automated UAV fleet management.

Therefore, this thesis addresses the development of a localization framework based in Graph-SLAM that is capable of calculating the real-time position and orientation of a commercial UAV in warehouse scenarios.

Keywords: Indoor Localization, SLAM, Graph-SLAM, UAV, Industry 4.0, ROS, Tags, DJI.

Esta página foi intencionalmente deixada em branco.

Resumo

No contexto da Indústria 4.0, os armazéns estão a sofrer transformações e a criar soluções que permitem que cada vez mais tarefas relacionadas com estas infraestruturas sejam realizadas de forma autónoma.

A utilização de robôs traz grandes vantagens e grandes acréscimos à autonomia de tais estruturas pois estes podem ajudar na contagem cíclica e automática do inventário, na localização de *stock* difícil de encontrar e no acesso a áreas de armazenamento estreitas ou altas. Neste contexto, o uso de um UAV torna-se um requisito neste tipo de operações.

No entanto, o uso deste tipo de robôs dentro de um armazém apresenta grandes desafios de engenharia, como a localização e navegação autónoma, a prevenção e deteção de colisões e a gestão automatizada de frotas de veículos.

Neste contexto, esta tese aborda o desenvolvimento de um algoritmo de localização capaz de calcular a posição e orientação em tempo real de um UAV em cenário de armazém indoor.

Palavras-Chave: Localização Indoor, SLAM, Graph-SLAM, UAV, Indústria 4.0, ROS, Tags, DJI.

Esta página foi intencionalmente deixada em branco.

Agradecimentos

Para a conclusão do presente projeto foi necessário muito esforço e dedicação o que não seria possível sem o apoio e contributo de várias pessoas. A todas elas deixo os meus sinceros agradecimentos.

Em primeiro lugar, um agradecimento especial ao meu amigo e companheiro de curso, André Moura por me ter acompanhado não só durante esta dissertação mas também durante todos os projetos que fomos desenvolvendo. Sem dúvida que fazemos uma boa equipa e sem ti a conclusão deste mestrado não teria sido possível.

Ao Professor Doutor André Miguel Pinheiro Dias, por todas as oportunidades, pela possibilidade de participar no desenvolvimento deste projeto e por todo o apoio, acompanhamento e aconselhamento que disponibilizou ao longo do mestrado.

A todos aqueles com que desenvolvi laços de amizade no decorrer do curso, nomeadamente os colegas, José Oliveira, David Carvalho, Hélder Ribeiro, Pedro Martins e José Tavares.

Aos docentes e investigadores que fazem parte do LSA, estou grato pelos ensinamentos fornecidos no decorrer dos últimos dois anos. Um especial obrigado ao companheiro e investigador Pedro André Silva que se mostrou uma mais valia no desenvolvimento desta dissertação.

Por fim um agradecimento muito especial à minha tia Felisbela Silva, pelo carinho, incentivo, cuidado e educação que me proporcionou durante o decorrer de toda a minha vida, não só académica mas também pessoal.

Esta página foi intencionalmente deixada em branco.

Contents

Abstract iii

Agradecimentos v

List of Figures xi

List of Tables xv

Listing xvii

List of Acronyms xx

1 Introduction 1

1.1 Objectives 2

1.2 Structure 2

2 Problem Formulation 5

3 State of Art 7

3.1 Indoor Localization Technologies 7

3.1.1 Vision Systems 8

3.1.2 Inertial Sensors 9

3.1.3 Magnetic Sensors 9

3.1.4 Sound Sensors 10

3.1.5 LIDAR Sensors 10

3.1.6 RFID Sensors 10

3.1.7 UWB Sensors 11

3.2 Vision Based Localization Methods 11

3.2.1 Vision Methods Based In Natural Features . . . 11
3.2.2 Vision Methods Based In Artificial Features . . . 13
3.2.3 Vision Methods Based In Hybrid Features . . . 14
3.3 Artificial Markers . . . 14
3.3.1 Types of Artificial Markers . . . 15
3.3.2 Comparison Between Visual Markers . . . 17
3.3.3 AprilTag Detector . . . 18
3.4 State of Art Discussion . . . 19

4 Theoretical Concepts **21**
4.1 SLAM . . . 21
4.2 SAM - Smoothing and Mapping . . . 24

5 Project **27**
5.1 Robotic Platform . . . 27
5.2 SDK . . . 29
5.3 Drone and Ground Station Communication . . . 29
5.4 Localization Algorithm . . . 30
5.5 Proposed Architecture . . . 31

6 Implementation **33**
6.1 Mobile SDK + ROS Application . . . 33
6.2 Decompression Calibration and Rectification . . . 42
6.3 Frame Transformation and Pose Estimator . . . 43
6.4 AprilTags Detector . . . 49
6.5 Graph-Slam Algorithm . . . 56
6.6 UcoSlam . . . 60

7 Results **63**
7.1 Mobile SDK and ROS Application . . . 63
7.2 Decompression and Calibration . . . 68
7.3 Localization Algorithms . . . 69
7.3.1 Box of Tags Tests . . . 69
7.3.2 Evaluation of the Localization During a Warehouse Mission . . . 76
7.4 Results Discussion . . . 93

8 Conclusions **95**

Bibliography **97**

A Non Simplified Proposed Architecture **101**

Esta página foi intencionalmente deixada em branco.

List of Figures

1.1 Conceptual Approach for UAV Warehouse Logistic Inventory [4]. 2

2.1 Global Architecture of the Project. 5

3.1 Diagram of the most commonly used indoor localization technologies [5]. . 7
3.2 Quick Response Code (QR Code) [25]. 15
3.3 Aztec Code [25]. 15
3.4 ARToolKit, ARTag, AprilTag and ArUco markers [27]. 16
3.5 Vuforia Frame Marker [26]. 16

4.1 Example of a factor graph [4]. 22

5.1 Localization algorithm architecture. 30
5.2 Proposed Architecture. 32

6.1 Sample code USB connection layout [35]. 34
6.2 Sample code video feed layout [35]. 34
6.3 ArUco markers bundle used to perform the calibration. 42
6.4 Tool used the perform the camera calibration. 43
6.5 Coordinate system used for the localization algorithm. 44
6.6 Coordinate system used by DJI gimbal. 47
6.7 Difference between the coordinate systems of a DJI gimbal and an image [42]. 48
6.8 Standalone AprilTag. 53
6.9 Bundle of AprilTags. 53
6.10 AprilTag coordinate system. 54

7.1 Application entry layout. 64
7.2 ROS connection layout 65
7.3 Application main layout. 66
7.4 Rosgraph of the android application. 66
7.5 ROS topics shared between the android and the ground station. 67
7.6 Vehicle attitude topic. 67
7.7 Vehicle velocity and altitude topic. 67
7.8 Image frame obtained after the decompression. 68
7.9 Tower of AprilTags scenario. 69
7.10 Tower of ArUco tags scenario. 70
7.11 RViz view of the drone and tags estimated position from the Graph-Slam Algorithm. 70
7.12 Graph-Slam trajectory with markers during the initial tests. 71
7.13 Graph-Slam trajectory with markers during the initial tests (XZ). 71
7.14 Graph-Slam trajectory with markers during the initial tests (YZ). 72
7.15 Graph-Slam trajectory with markers during the initial tests (YX). 72
7.16 RViz view of the drone and tags estimated position from the UcoSlam Algorithm. 73
7.17 GUI available in the UcoSlam framework. 73
7.18 Uco-Slam trajectory with markers during the initial tests. 74
7.19 Uco-Slam trajectory with markers during the initial tests (XZ). 74
7.20 Uco-Slam trajectory with markers during the initial tests (YZ). 75
7.21 Uco-Slam trajectory with markers during the initial tests (YX). 75
7.22 Instrumented warehouse look-alike scenario. 76
7.23 Real-Time visualization of the Graph-Slam algorithm. 77
7.24 Real-Time visualization of the Graph-Slam algorithm from another perspective. 78
7.25 Graph-Slam processing time along the flight time. 78
7.26 Graph-Slam optimized trajectory with markers during the final tests. . . . 79
7.27 Graph-Slam optimized trajectory with markers during the final tests (XZ). 79
7.28 Graph-Slam optimized trajectory with markers during the final tests (YZ). 80
7.29 Graph-Slam optimized trajectory with markers during the final tests (YX). 80
7.30 Graph-Slam optimized trajectory vs raw data provide by DJI SDK. 81

7.31 Graph-Slam optimized trajectory vs raw data provide by DJI SDK (XZ). . 81
7.32 Graph-Slam optimized trajectory vs raw data provide by DJI SDK (YZ). . 82
7.33 Graph-Slam optimized trajectory vs raw data provide by DJI SDK (YX). 82
7.34 Real-time trajectory vs fully optimized trajectory. 83
7.35 Real-time trajectory vs fully optimized trajectory (XZ). 83
7.36 Real-time trajectory vs fully optimized trajectory (YZ). 84
7.37 Real-time trajectory vs fully optimized trajectory (YX). 84
7.38 UcoSlam processing time along flight time. 85
7.39 UcoSlam real-time trajectory with markers during the final tests. 86
7.40 UcoSlam real-time trajectory with markers during the final tests (XZ). . . 86
7.41 UcoSlam real-time trajectory with markers during the final tests (YZ). . . 87
7.42 UcoSlam real-time trajectory with markers during the final tests (YX). . . 87
7.43 Faro estimated tags compared to Graph-Slam estimated tags. 88
7.44 Faro estimated tags compared to Graph-Slam estimated tags (XZ). 89
7.45 Faro estimated tags compared to Graph-Slam estimated tags (YZ). 89
7.46 Faro estimated tags compared to Graph-Slam estimated tags (YX). 90
7.47 Faro estimated tags compared to UcoSlam estimated tags. 90
7.48 Faro estimated tags compared to UcoSlam estimated tags (XZ). 91
7.49 Faro estimated tags compared to UcoSlam estimated tags (YZ). 91
7.50 Faro estimated tags compared to UcoSlam estimated tags (YX). 92

A.1 Non Simplified Proposed Architecture. 102

Esta página foi intencionalmente deixada em branco.

List of Tables

5.1 Comparison table between the DJI's Mavic model platforms. 28

6.1 Presentation and explanation of all the developed ROS topics. 40
6.2 Presentation and explanation of all the implemented vehicle commands. . 41

7.1 Graph-Slam estimated tags errors. 92
7.2 UcoSlam estimated tags errors. 93

Esta página foi intencionalmente deixada em branco.

Listings

6.1 Establishing the connection to the ROS master. 35
6.2 Creating a publisher without an associated thread. 35
6.3 Creating a publisher with an associated thread. 35
6.4 Example of how to obtain the drone attitude. 36
6.5 Subscriber and callback responsible for receiving aircraft commands. . . . 37
6.6 Parse_Command function. 37
6.7 Example of how to send a velocity to the aircraft. 38
6.8 Conversion of a SurfaceTexture in a byte array. 39
6.9 Transmission of a video feed image frame. 40
6.10 Decompression of the video feed. 42
6.11 Creating of the **ned** frame regarding the **origin** frame. 44
6.12 Calculation of the raw pose of the vehicle. 45
6.13 Creation of the **drone0** frame. 47
6.14 Creation of the **drone0_gimbal** frame. 48
6.15 Creation of the **drone0_image** frame. 49
6.16 Creation of a frame for a detected marker. 51
6.17 Publication of the marker id along side the relative pose. 52
6.18 Example of the tags.yaml file. 55
6.19 Example of the settings.yaml file. 55
6.20 Definition of the motion and measurement noises. 57
6.21 Addition of the motion inputs to the factor graph. 57
6.22 Addition of the measurement inputs to the factor graph. 58
6.23 Optimization of the factor graph. 59
6.24 Obtainment of the image and calculation of the drone pose. 60

Esta página foi intencionalmente deixada em branco.

List of Acronyms

AGV Automated Guided Vehicle

AI Artificial Intelligence

API Application Programming Interface

BA Bundle Adjustment

CATEC Center for Advanced Aerospace Technologies

EKF Extended Kalman Filter

FPV Flying First Person View

GNSS Global Navigation Satellite System

GPU Graphical Processing Unit

GTSAM Georgia Tech Smoothing and Mapping

GUI Graphical User Interface

IDE Integrated Development Environment

IMU Inertial Measurement Unit

IoT Internet of Things

iSAM2 Incremental Smoothing and Mapping 2

LIDAR Light Detection and Ranging

MAP Maximum-A-Posteriori

NED North East Down

ORB Oriented FAST and Rotated BRIEF

OS Operating System

RAM Random Access Memory

RANSAC Random Sample Consensus

RF Radio Frequency

RFID Radio Frequency Identification

ROS Robotic Operating System

SAM Smoothing and Mapping

SDK Software Development Kit

SLAM Simultaneous Localization and Mapping

TDOA Time Difference of Arrival

TOA Time of Arrival

UAV Unmanned Aerial Vehicle

USB Universal Serial Bus

UWB Ultra-Wideband

Chapter 1

Introduction

The technological transformations of warehouses are happening at a global scale, and this is driven by safety, costs, and revenue advantages that are generated when compared to the manual ways. The application of Automated Guided Vehicles (AGVs) is not a solution to the problem, since the increase in the supply chain volatility, the demand uncertainty, and the operational complexity make these systems not compensate for the high capital investments.

In this context, the Warehouse 4.0 together with the next wave of technological innovations, such as Artificial Intelligence (AI), Internet of Things (IoT), and the use of commercial drones bring the promise of reducing the inventory counting times to hours and days instead of weeks with almost 100% accuracy and minimal downtime with cheap and predictive maintenance's [1].

Warehouses and distribution centers usually rely on storing inventory in racks along aisles that can be accessed by workers and warehouse machines. In this case, cost efficiency and practicality increase with the shallower and narrower the racks and aisles are. The ideal scenario would be narrow aisles with one pallet deep racks on both sides, prioritizing space utilization and quick inventory access.

Considering this context, UAV autonomous or semi-autonomous systems are starting to play the main role in this type of warehouses. Given their abilities to fly autonomously, carry payloads, avoid obstacles, and land precisely. These aerial vehicles also have the capabilities of operating autonomously in a swarm increasing their efficiency when used in a large number. Taking into account the UAV characteristics and the way a warehouse is disposed, these vehicles have the potential to be used in systems designed to autonomously perform tasks such as, inventory management, logistics, inspection, and surveillance [2] [3]. From these four main tasks, inventory management applications

have gained a lot of interest from the industry for their potential and there are even implementations that passed the testing phase in some companies [2].

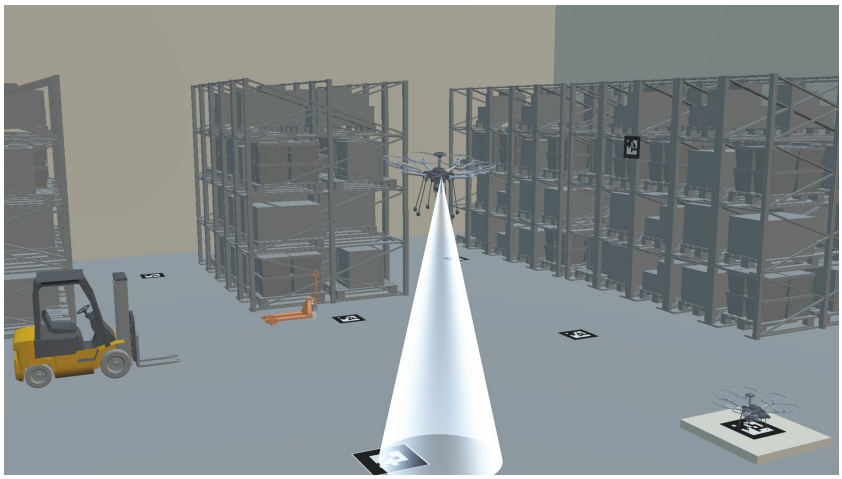

Figure 1.1: Conceptual Approach for UAV Warehouse Logistic Inventory [4].

1.1 Objectives

The main objective of this thesis is the implementation of an indoor localization algorithm to be used in a UAV in a warehouse inventory management scenario. This main objective can be divided into the following steps:

- Analyses of the State-of-Art for UAV localization methods in indoor scenarios;
- Development of an indoor localization approach able to be used in a commercial UAV (not require additional sensors);
- Benchmark the developed approach with an exogenous Ground-Truth;
- Comparison between the estimated 3D position and the state-of-art available methods.

1.2 Structure

This thesis is divided into eight chapters. In the second chapter, it is possible to find the problem formulation where a brief explanation of the challenges that generated this project are presented.

The state of art is introduced in chapter three, where some indoor localization technologies are presented, some vision-based localization methods are compared and a study about the best type of artificial markers is made.

In chapter four, the theoretical concepts regarding the developed localization algorithm are mathematically explained.

The project and all the decisions made during it are presented in chapter five. At the end of this chapter, a low-level architecture of the project is also shown.

In chapter six, all the implementations that were made during the development of this thesis are shown and explained. This chapter is divided into sections where each section explains every different thing that was implemented, that is, an android application, the decompression of the video feed, the calibration of the camera, and finally the localization algorithms.

The penultimate chapter exposes all the results associated with the implementations that were made during the project.

Finally, the conclusion in the last chapter expresses some commentaries about all the work that was developed while also explaining future developments.

Esta página foi intencionalmente deixada em branco.

Chapter 2

Problem Formulation

An UAV performing warehouse logistic applications is the typical problem of having a drone accomplishing a mission in an indoor scenario. Such type of scenario can be a challenge for most commercial platforms since almost every one uses some type of Global Navigation Satellite System (GNSS) signal to solve the localization problem. Another problem that arises in a warehouse scenario is that no commercial UAV is prepared to accomplish a mission related to logistic applications since most platforms are created with the main purpose of being used in photography-related missions. Alongside these problems, there is also an issue related with the possibility of having static and dynamic obstacles scattered through the scenario.

Given all the challenges that a warehouse logistic application can generate, the following global architecture was created.

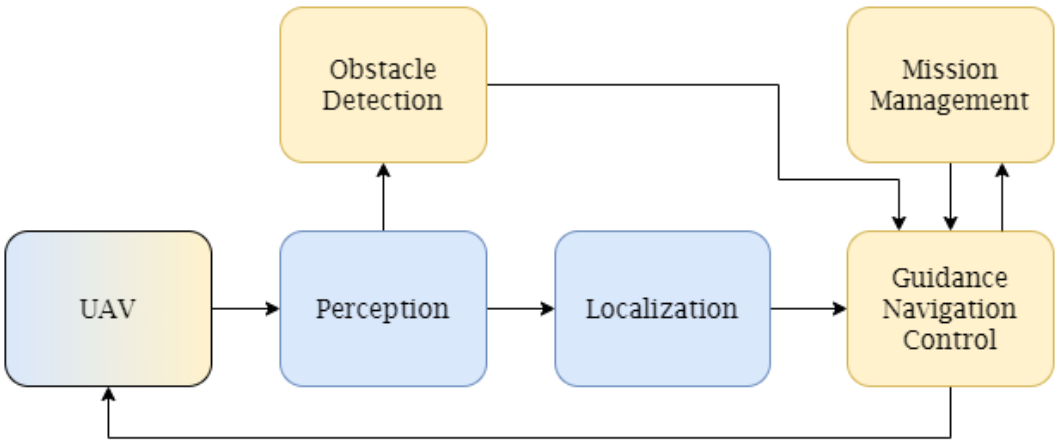

Figure 2.1: Global Architecture of the Project.

The proposed global architecture is divided in the following blocks:

- The UAV block, consisting of the robotic platform and all the communication protocols used to communicate with it;
- The perception block, responsible for receiving the sensory data coming from the UAV and prepare it to be used by the localization block and the obstacle detection block;
- The localization block receives as input the perception data and calculates the real-time localization of the drone inside the warehouse;
- The obstacle detection block, that must be capable of identifying all the detected obstacles using the sensor data coming from the perception block;
- The mission management block decides the next steps to be accomplished given the pre-planned mission and the real-time drone state;
- The guidance navigation control block is responsible by autonomously controlling the drone given the real time localization and the mission objectives.

Taking into account the complexity, the project was divided in two major problems: 3D indoor localization problem and the mission planning and navigation problem. The main focus of the presented thesis will be the development of a indoor localization framework capable of solving the indoor localization challenge.

Chapter 3

State of Art

This chapter will present the state of art related with some of the most used indoor localization technologies alongside some of the most well known algorithms that solve the localization problem using vision based systems. Since one of the main focus of this thesis is the warehouse 4.0 theme, this chapter will also introduce some of the state of art related with artificial markers doing a comparison between each one of markers in terms of applicability in indoor localization. It will also present one of the most promising algorithm regarding the detection of these markers. Finally a brief discussion about the state of art will also be made.

3.1 Indoor Localization Technologies

One of the main challenges for an UAV in indoor environments is the 3D position due to the the lack of GNSS signals. Therefore, the robotic community has already explored some approaches to mitigate this problem by combining different sensors, as displayed in Figure 3.1.

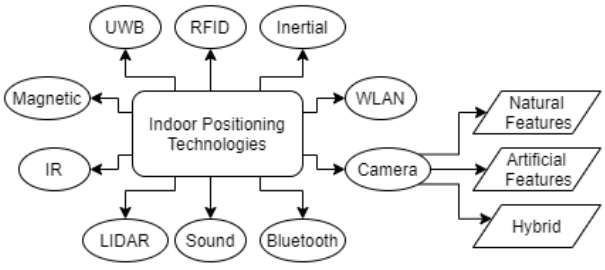

Figure 3.1: Diagram of the most commonly used indoor localization technologies [5].

As seen in [5] some technologies are more frequently found in an UAV, namely :

- Vision systems
- Inertial sensors
- Magnetic sensors
- Sound sensors
- Light Detection and Ranging (LIDAR) sensors
- Radio Frequency Identification (RFID) sensors
- Ultra-Wideband (UWB) sensors

3.1.1 Vision Systems

In the last decade vision systems have become extremely popular in localization problems due to the capability of obtaining accuracies from 0.1 mm to 1 dm with typical coverages between 1 and 10 m. This increase in popularity comes from the technological advancements that not only reduce the price of these sensors but also advance computational capabilities. Nevertheless, one of the biggest disadvantages of these systems when compared to other sensors still is the high computational demands [5] [6].

Vision systems have three main techniques widely used to calculate the position, one technique relies solely on the usage of natural features, the other uses artificial features, also called markers, and finally, a hybrid technique that uses both natural and artificial features for solving the localization problem.

3.1.1.1 Natural Features

Natural features are objects that already exist in the ambient. The camera can take advantage of this type of features to calculate the position taking into account movement and angles between camera frames. The big advantage of this technique is that there is no need to alter the environment.

The natural features can also be stored in a database, solving the localization problem with a big coverage while maintaining high accuracy, but the computation time will also increase. On the other hand, if there is no database, less computation time is required but the localization is only good in a small coverage [5]. It is important to denote that when the feature matching fails the localization accuracy becomes heavily degraded.

3.1.1.2 Artificial Features

Artificial features, also known as artificial markers, such as AprilTags, QR Codes, ArUco markers, etc, have the big disadvantage that the environment must be altered to add this type of feature. But as an advantage, these markers are very robust to changes in light conditions while also making the detection faster and more accurate. The problem of wrong feature matching is also solved since each marker has a unique identification code [5].

3.1.1.3 Hybrid Features

As the name implies, hybrid features are a junction of both types of features, this type of approach is common in places that already have some artificial features but not enough to solve the localization problem, in this cases the use of natural features when there are no artificial ones in sight solve the need of a high number of markers. The lack of robustness of the natural features is also partially solved by the use of artificial markers.

3.1.2 Inertial Sensors

Inertial sensors have typical accuracies close to 1% and typical coverages from 10 to 100 m and can be relatively inexpensive, allow dead reckoning positioning, where an initial position is known and the integration of all previous displacements of the vehicle allows to know the vehicle position at a given time. Among these types of sensors, the Inertial Measurement Unit (IMU) is one of the most popular. The most complete ones consist of a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer, that respectively measure acceleration, angular velocity, and magnetic field changes along the three axes. The main disadvantage of this kind of sensor is the error accumulated over time [5] [6].

3.1.3 Magnetic Sensors

Magnetic sensors as magnetometers and compasses obtain position information by analyzing the variation in magnetic fields, that can be associated with special transmitters or the earth's magnetic field. This type of sensors can have centimeter to millimeter accuracies, have typical coverages between 1 and 20 m, and their measurements are based on fingerprinting and raging methods [5] [6].

3.1.4 Sound Sensors

Sound sensors have typical accuracies of centimeters, typical coverages between 2 and 10 m, and work by measuring the time of travel of sound waves. Usually, this technology makes use of ultrasonic waves, as they are less obtrusive. Some of the sound sensors most commonly used in localization are sonars where, onboard the vehicle, the transmitter emits a sound signal to the environment and the echo of that signal is captured by a receiver, allowing to measure the distance traveled by the sound wave, that represents the distance to the ground, walls or obstacles. There are also systems with a passive device on the vehicle that captures sound signals from static emitters and through methods as Time of Arrival (TOA) or Time Difference of Arrival (TDOA) measurements can estimate a relative position and systems with an active device that determine a location based on multilateration from three or more ranges to fixed receivers deployed at known locations [5] [6].

3.1.5 LIDAR Sensors

LIDAR sensors provide dense distance measurements of the environment from the travel time of a pulsed laser. These measurements generally provide very accurate measurements, but these sensors can be very costly and are only used when high accuracy and dense environmental information is needed. The simplest LIDAR sensors provide only a horizontal line scan, but more advanced ones can also measure distances in the vertical plane [5] [6].

3.1.6 RFID Sensors

RFID sensors which have typical accuracies in the decimetre to meter range and typical coverages of 1 to 50 m. This technology is rapidly developing and uses wireless communication for automatic identification of tags, which are located via proximity, or in some cases by fingerprinting and lateration techniques. The tags used in RFID systems can be passive, using some of the energy in the Radio Frequency (RF) signal, making them inexpensive but also having a small range, or they can be active, which typically increases the range, but also makes them more expensive and means that batteries are limited and need replacements. This sensor technology is commonly used and interesting for inventory unit-level tracking, because of its price and the high reading rate that some readers provide [5] [6].

3.1.7 UWB Sensors

UWB sensors stand out because high bandwidths improve signal reliability since it is more likely for frequencies to go through or around obstacles, they also offer high resolutions and improved accuracies. UWB localization is usually based on body reflection and TOA methods having typical accuracies in the centimeter to meter range and typical coverages between 1 and 50 m [5] [6].

3.2 Vision Based Localization Methods

Vision systems and computer vision-based localization algorithms are one of the most commonly used for localization applications. There are two main kinds of approaches, one consisting of using static cameras in known locations that allow locating the robot, and another where the platform to be tracked carries the cameras used to monitor its surroundings while also locating himself [7].

Regarding the approach with static cameras, some examples demonstrate the real-time high precision capabilities of such systems. One example of this is the system used in the indoor testbench at Center for Advanced Aerospace Technologies (CATEC) facilities, which uses 20 Vicon motion capture cameras and passive markers placed on a UAV moving along the scenario. The system is capable of determining the vehicle's pose with millimeter precisions [8].

Nevertheless, this type of system requires a big monetary investment and takes a long time to calibrate and implement.

By contrast, implementations that use onboard cameras can be cheaper while also having good precision and allowing a rich representation of the environment. These methods can use multiple types of cameras and techniques, tracking both artificial and natural features from the captured images.

3.2.1 Vision Methods Based In Natural Features

The project in [9] is a good demonstration of the capabilities of a computer vision system with simple camera sensors. This system is applied to a UAV and the main objective is finding appropriate landing sites in emergencies and unknown environments. In this system, a single monocular camera allows capturing images that are used to locate the vehicle and map the environment through a Simultaneous Localization and Mapping (SLAM) algorithm. An Oriented FAST and Rotated BRIEF (ORB) [10] feature detection algorithm is used to find features and then the SLAM algorithm, with additional

data from an IMU, maps the features in a global reference frame. The resulting map is then transformed into a 2D grid map with a ground height that is used to find the optimal location for landing. The given results show the applicability of monocular cameras to construct reliable 3D maps of the environment, particularly using SLAM algorithms.

In [11], it is shown a more general usage of the optical flow technique, that uses this method measurements to make velocity estimations. Using an UAV downward-looking camera, the developed system recurs to a feature-based optical flow method to estimate the velocity of the drone. When the camera is looking at a high-textured surface this method uses an ORB feature detection algorithm to extract features while also using nearest neighbor to match these features and Random Sample Consensus (RANSAC) [12] for outlier rejection. When looking at low-textured surfaces, an image pyramids-based global optical flow method is used, based on the Lucas-Kanade method [13]. Finally, a classic Kalman Filter [14] fuses the optical flow measurements with the data coming from an IMU and a sonar, resulting in the final UAV velocity estimates.

On a different approach to optical flow in localization, in [15] optical flow measurements from an UAV onboard fisheye camera video feed are used to create a depth map that allows avoiding wall collision in indoor corridors navigation. For consecutive video frames, the optical flow is determined for pixels selected in the first frame using Shi and Tomasi's corner finder [16] with the Pyramidal Lucas-Kanade method. After the optical flow estimation, IMU data is used for compensation of rotation-based effects. The depth estimation is then calculated, assuming linear motion of the vehicle and considering that optical flow is a function of the robot's forward velocity, the distance to the obstacle, and the angle between the direction of travel and the obstacle. These depth values allow the construction of a depth map and the calculation of an error relative to the center of the corridor, used in a simulated PD controller that showed a centering behavior of the UAV.

One of the state-of-art algorithms in visual SLAM using only natural features is ORB-SLAM2 [17]. This technique relies on three main parallel threads responsible for all the processing. The tracking thread, responsible for localizing the camera with every frame by finding matches between features present in a local map. The local mapping thread, that manages the local map and optimizes it performing a local Bundle Adjustment (BA) [18]. And finally a loop closing thread, responsible for detecting large loops and correct the accumulated drift by performing pose-graph optimizations. The system uses ORB features for tracking and mapping and a place recognition module based on DBoW2 [19] for relocalization in case of tracking failure. The results show that the algorithm is capable of doing real-time accurate estimations of the position using either monocular,

stereo, or RGB-D cameras. Even though the position using a monocular camera is accurate the depth information is still missing, for this reason, the monocular results are not in a metric scale. Pure rotation movement also poses a negative impact on the results of monocular cameras.

3.2.2 Vision Methods Based In Artificial Features

When talking about methods that only use artificial features to solve the localization problem, the SPM-SLAM [20] is considered one of the state-of-art algorithms. This algorithm was created to be used with monocular cameras and relies heavily on the usage of ArUco markers to calculate the pose of the camera. Initially, the algorithm searches for several markers to start the tracking process. Once the initialization step is completed, for each frame obtained, the algorithm tries to detect markers, once the markers are detected the current pose is estimated by minimizing the reprojection error of the visible marker corners. When the new camera pose is estimated, the map is updated, if there is a new marker in the frame, this frame and the markers are added to the global map and the frame is transformed into a keyframe, if there are no new markers, the map is only updated and the frame transformed in a keyframe if the 3D distance to the previous frame is higher then a certain threshold. To avoid the unnecessary growth of the map, a keyframe culling process is run whenever a new keyframe is added to the map. The goal of this culling process is to keep a minimal set of keyframes that still allow a proper optimization of the reprojection error of the markers. After the culling process, a local optimization process is run. At the same time, the frame passes a loop closure check to detect a loop close and reduce possible drift and reprojection errors. The final step is adding the keyframe to the loop closure detection map. When there are no markers in the frame, the system moves to a state of relocalization, where it waits until any valid marker is observed. The results obtained by the algorithm are promising, showing high accuracy and a fast computing time. One of the biggest advantages of the algorithm is that it can estimate the real-world scale using the markers, and for this reason, all the poses are metric scaled. The biggest disadvantage is that if the system does not detect any markers during a big set of frames the localization gets compromised, for this reason, the ambient needs to have a big quantity of markers for the method to work.

Another algorithm that only uses artificial markers is the one presented in [21], similarly to SPM-SLAM, this method also relies on a high quantity of AprilTags distributed by the map. It was also created to be used with monocular cameras and it can also solve the problem of real-world scaled measurements using this kind of cameras. The algorithm

creates a factor graph [22] using the Georgia Tech Smoothing and Mapping (GTSAM) [23] framework to construct the factor graph and solve the localization and optimization problems. Even though the results from TagSLAM are not as good as the ones from the SPM-SLAM, the algorithm is still capable of solving the localization problem robustly recurring to an easy-to-use framework. The TagSLAM also faces heavy losses on the accuracy when no markers are detected in a large set of frames, having the same disadvantage as the SPM-SLAM.

3.2.3 Vision Methods Based In Hybrid Features

In real-world applications, the use of approaches that rely solely on artificial markers is sub-optimal, since in most cases it is impossible to distribute markers all over the working space. Taking this into account the developers of the SPM-SLAM created the UcoSLAM algorithm present in [24]. The developed algorithm solves the problem that emerges when no markers are in the frame using natural features to solve the localization problem. When no markers are found, the algorithm uses a changed version of the ORB-SLAM2 to calculate the pose of the camera. When there are markers in the frame, the method calculates the reprojection errors of the markers observation and decides if they can be used. If the reprojection error is below a certain threshold the markers are used for tracking and localization similarly to the ones from SPM-SLAM. The rest of the algorithm is identical to the SPM-SLAM, with the same keyframe culling, local optimizations, loop closing, and relocalization steps. The obtained results are even more positive than the ones from ORB-SLAM2 or SPM-SLAM, obtaining even better localization accuracies while maintaining real-time processing times. The system ceases to need a big set of markers to calculate the poses of the cameras since it can use natural features when no markers are in the frame. And the problem of missing depth information on monocular systems is completely solved since every time the algorithm sees a marker it scales all the previous positioning information while also reducing possible drifts associated with the previously calculated scales.

3.3 Artificial Markers

Since the artificial markers are such a good technology to be used in the context of warehouse 4.0 it is important to understand what is a marker, what type of markers exist and which ones are better for real-time applications. For these reasons, this section of the state of art will be focused on artificial markers.

An artificial marker, also known as a tag or visual marker, is a mark that can be detected by a camera since it is perceptible in the visible spectrum of light. When the tag is visible a finder pattern is normally used to detect it. The most common patterns are squares and circles since it is easier to detect such shapes in an image [25].

3.3.1 Types of Artificial Markers

From all the artificial markers available, the following list provides some general outlook about the most used ones in the market.

- The Quick Response Code, shown in Figure 3.2, is a type of squared marker identified with three finder patterns distributed in the top left and right sides and the bottom left corners. The QR Code is black and white, and the combination of these colors in its inside pattern represents the binary digits it can hold. This marker can hold up to 2953 bytes of data and for this reason, it does not have the real-time performance to be used in a localization system for a UAV [25][26].

Figure 3.2: Quick Response Code (QR Code) [25].

- The Aztec Code is another squared type of marker that also uses the black and white colors, where the white color represents the background of the marker and the black color represents the data. The finder's pattern is in the center of the tag with a "square bull's eye" form. This marker can store up to 1914 bytes of data, making it unusable in real-time tasks [25][26]. The Figure 3.3 shows an example of an Aztec Code.

Figure 3.3: Aztec Code [25].

- The ARToolKit markers, presented in Figure 3.4, are a type of markers that theoretically can have a detection pattern with any shape as long as this pattern is surrounded by a black square since the detection algorithm relies on a template-matching process. This type of markers can only store a simple binary code making the detection fast and usable in real-time applications. From this type of markers others were made, where the most used ones are the ARTag the AprilTag, and the ArUco. These markers were created with the purpose of the uniqueness of markers, avoiding confusion in the detection process. Their detection is also more robust if part of the marker is hidden. These tags can also be used in real-time applications [25][26].

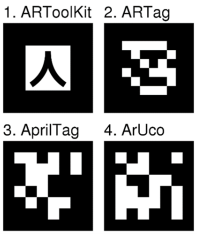

Figure 3.4: ARToolKit, ARTag, AprilTag and ArUco markers [27].

- The Vuforia Frame Marker is a type of marker developed by Qualcomm with a Software Development Kit (SDK) capable of detecting and estimating the pose of the marker concerning the camera, there are only 512 different frame markers with identification codes from 0 to 511. This type of marker is also a square type one with the advantage that only the border of the marker is needed to detect it and identify the respective ID. Since the internal part of the marker is not used in the detection algorithm it is possible to use an image or a logo inside, making it more esthetically good-looking than others. Unfortunately, the SDK is not open source, making it impossible to understand the detection algorithm behind these types of tags. The Vuforia markers are also usable in real-time scenarios [26]. An example of a Vuforia marker is present in Figure 3.5.

Figure 3.5: Vuforia Frame Marker [26].

Even though there are much more types of visual markers, the ones named above are the most used in warehouse 4.0 scenarios. For this reason, the other type of markers will not be referenced in this state-of-art.

For the markers to be used in the localization system they must respect certain characteristics, which are real-time detection, robustness to changes in light conditions, robustness when detected in blurred or out-of-focus images, and they must be detected while having the smallest size possible. The artificial markers that best fulfill these requirements are the AprilTag, the ArUco, and the Vuforia markers.

3.3.2 Comparison Between Visual Markers

In [26] a comparison between these three types of markers is made. The conducted tests consisted in comparing the performance in the detection of the three markers using three different sizes for each marker type, 6.5 cm x 6.5 cm, 5.0 cm x 5.0 cm, and 3.2 cm x 3.2 cm, for each size different distances between the marker and the camera were also tested, 80 cm, 100 cm, and 120 cm. Finally taking into account the previous sizes of markers and distances, different light conditions were also tested, good light conditions, average light conditions, and poor light conditions.

For the Vuforia markers, it is possible to conclude that the detection gives good real-time results for the two bigger tag sizes in good and average light conditions. The performance keeps getting worse for poor light conditions and if the size of the marker is the smallest one.

The tests show that the ArUco markers work very well for any light conditions for the two bigger marker sizes, but the performance is a little worse when the marker size is reduced and the distance between the camera and the marker is 120 cm on average and poor light conditions. In general, the results are better than the Vuforia markers.

Finally, the AprilTag marker had the best results in the conducted tests having no problem to be detected at any of the tested distances with any of the used marker sizes at any light conditions, making it the best candidate to be used as a visual marker in a localization system that requires robustness and real-time performance.

As an extra test of performance, it was also tested the detection of the AprilTags for different angles between the marker's normal vector and the camera vector. The results showed that for angles below 60 degrees the detector has no problem identifying the marker, for angles bigger than 60 degrees the results start to get worse to the point that no marker is detected.

3.3.3 AprilTag Detector

The reference [28] introduces one of the most used and well-known algorithms for AprilTag detection. This algorithm possesses a wrapper for Robotic Operating System (ROS) and can detect all the artificial markers in a grayscale image with a low false-positive rate and a high detection ratio, making the algorithm reliable and robust.

The first step of the algorithm is to apply a threshold to the grayscale image to convert the image into a completely black-white image. This is done using an adaptative threshold where the idea is to find the minimum and maximum values in a region around each pixel.

The second step is to find edges in the binarized image which might form the boundary of a marker. The author proposes to segment the edges based on the identities of the black and white components from which they arise.

The third step is to fit a quad to each one of the unordered boundary points, partitioning the points into four groups corresponding to each line segment.

The fourth step is to do quick decoding of the detected tag, this is done by applying an XOR between the detected tag with each one of the codes from the tag family. The codes for each tag family are stored in a hash table to speed up the decoding process.

Finally, if localization accuracy is desired for pose estimation, the algorithm also refines the edges in the image to better detect the position of the tag, this process makes the detection slower but more accurate. If the user only wishes to detect the tag in the image without much pose accuracy this step can be skipped.

The experimental results show that the algorithm has high robustness against false positive detections since in 13,623,725 detections only 6 were falsely leading to a false positive rate of 0.000044%. The algorithm can detect almost 100% of the tags up to 7 m away from the camera. The orientation error keeps below 1 degree for off-axis angles between 20 degrees and 80 degrees. The detector is also able to estimate the relative pose of the marker with errors below 10 cm up to 6 meters away from the camera. Using an Intel Xeon E5-2640 with a single core running at 2.5 GHz the detector took 78 ms to process a 640 x 480 image making it possible to use this algorithm in a real-time scenario at an average frame rate.

3.4 State of Art Discussion

Taking into account all the state-of-art presented in this thesis, it is stated that almost all indoor localization algorithms and technologies rely heavily on the usage of vision systems since this type of sensor can solve the localization problem with high accuracy and in real-time.

Considering that the workspace of the UAV will be in a warehouse scenario, the usage of algorithms that relies on artificial features for localization can be an advantage when solving the localization problem.

From all the artificial markers available, the AprilTags were the ones with the best results [26], for this reason, they will be the main type of markers used in the developed localization algorithm.

Since using only artificial markers is impossible in real-world applications, the developed algorithm will need to either use natural features or other types of sensors to aid in solving the localization problem.

Finally, because the UcoSLAM algorithm showed promising results and it is an optimal algorithm to be used in a warehouse 4.0 scenario, it will be used to compare with the developed algorithms.

Esta página foi intencionalmente deixada em branco.

Chapter 4

Theoretical Concepts

This chapter will present the mathematical theoretical concepts that were used during the development of the localization algorithm, namely the concepts behind the Graph-SLAM and the Smoothing and Mapping.

4.1 SLAM

In [23] factor graphs are graphical models capable of modeling complex estimation problems, such as SLAM problems. These graphs have big advantages when solving this type of problem since they allow to describe the optimization problem adequately [29].

To formulate the factor graph as a SLAM problem, one must first define two types of variables, the ones responsible for the localization, $X = \{x_i\}$, and the ones responsible for the mapping, $L = \{l_j\}$, the first ones represent all the poses from the robot and the latter represent the poses of all the landmarks that create the map. When talking about factor graphs, these variables represent states that need to be estimated. Additionally, factors describe the connection between states in the factor graph and represent functions on subsets of those variables. The edges, responsible for connecting the factors and the states, indicate that a certain state depends on a certain factor.

Whenever a landmark is introduced to the SLAM problem it is also identified and associated with a landmark state in the factor graph, allowing the correct placement of factors between that landmark and the state. This association problem is widely known as data association. The data association problem will not be explained since it is already solved. This fact will be explained in future sections of this thesis.

In the next diagram it is presented an example of a factor graph where α_i are motion inputs that relate the state x_i with the state x_{i-1} and ζ_n are measurement inputs that

relate a certain l_j with a certain x_i.

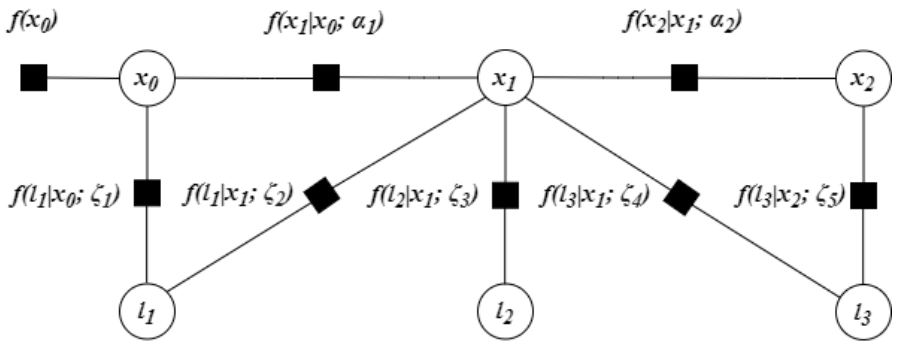

Figure 4.1: Example of a factor graph [4].

As seen in [4], the previous factor graph written as a joint probability of a belief network yields

$$P(X, L, Z) \propto P(x_0) \sum_{i=1}^{M} P(x_i|x_{i-1}, \alpha_i) \sum_{n=1}^{N} P(\zeta_n|x_{i_n}, l_{j_n}) \tag{4.1}$$

On the previous equation, the probability $P(x_0)$ represents a prior knowledge probability on the initial state regarding the localization of the robot. The $P(x_i|x_{i-1}, \alpha_i)$ probability is the motion model taking into account the α_i motion input. Finally the $P(\zeta|x, l)$ probability, represents the measurement or observation model taking in consideration the measurement inputs ζ_n. Note that even though $P(X, L, Z)$ also depends on some scale factor, this can be omitted since the SLAM will be ultimately handled as an optimization problem.

Assuming that the motion and measurement models behave as Gaussian models they can be defined as

$$x_0 = f_0(x_0) \Leftrightarrow P(x_0) \propto 1 \tag{4.2}$$

$$x_i = f_i(x_{i-1}, \alpha_i) + w_i \Leftrightarrow P(x_i|x_{i-1}, \alpha_i) \propto exp(-\frac{1}{2}||f_i(x_{i-1}, \alpha_i) - x_i||^2_{\Sigma_i}) \tag{4.3}$$

$$\zeta_n = h_n(x_{i_n}, l_{j_n}) + v_n \Leftrightarrow P(\zeta_n|x_{i_n}, l_{j_n}) \propto exp(-\frac{1}{2}||h_n(x_{i_n}, l_{j_n}) - \zeta_n||^2_{\Lambda_n}) \tag{4.4}$$

In the equation 4.3, f_i represents the motion model equation and w_i a zero-mean motion noise with co-variance matrix Σ_i. Regarding the equation 4.4, h_n represents the

measurement model equation and v_n the zero-mean motion noise with co-variance matrix Λ_n. Notice that in equation 4.2, $f_0(x_0)$ and $P(x_0)$ are equal to one since it is assumed that x_0 is known, this is true because in practice the origin of the coordinate system is arbitrary and usually x_0 is fixed at the origin [30].

Since the main objective is to solve the SLAM as an optimization problem, it must be converted into a Maximum-A-Posteriori (MAP) for the whole trajectory. Considering $\Omega := (X, L)$, it is possible to obtain the MAP estimate by maximization of the equation 4.1, giving rise to

$$\Omega^* := \underset{\Omega}{\operatorname{argmax}}\, P(X, L \backslash Z) \tag{4.5}$$

Furthermore, applying the negative logarithmic law to the equation 4.3, the maximization is then converted in a non-linear least-squares minimization problem given by

$$\Omega^* := \underset{\Omega}{\operatorname{argmin}}(\sum_{i=1}^{M} ||f_i(x_{i-1}, \alpha_i) - x_i||^2_{\Sigma_i} + \sum_{n=1}^{N} ||h_n(x_{i_n}, l_{j_n}) - \zeta_n||^2_{\Lambda_n}) \tag{4.6}$$

.

Trying to solve the optimization problem with non-linear equations is not efficient, for this reason, the equation 4.6 needs to be linearized using the associated Jacobians [31].

The linearization of the motion and measurement terms in equation 4.6 result in

$$f_i(x_{i-1}, \alpha_i) - x_i \approx \{f_i(x^0_{i-1}, \alpha_i) + F_i^{i-1}\partial x_{i-1}\} - \{x_i^0 + \partial x_i\} = \{F_i^{i-1}\partial x_{i-1} - \partial x_i\} - a_i \tag{4.7}$$

$$h_n(x_{i_n}, l_{j_n}) - \zeta_n \approx \{h_n(x^0_{i_n}, l^0_{j_n}) + H_n^{i_n}\partial x_{i_n} + J_n^{j_n}\partial l_{j_n}\} - \zeta_n = \{H_n^{i_n}\partial x_{i_n} + J_n^{j_n}\partial l_{j_n}\} - c_n \tag{4.8}$$

In the equation 4.7 the term F_i^{i-1} represents the Jacobian of the motion inputs at the linearization point x^0_{i-1}. Regarding the equation 4.8, $H_n^{i_n}$ and $J_n^{j_n}$ are the Jacobians of the measurement terms when x_{i_n} and l_{j_n} suffer a change evaluated at the linearization point $(x^0_{i_n}, l^0_{j_n})$. The terms $a_i := x_i^0 - f_i(x^0_{i-1}, \alpha_i)$ and $c_n := z_n - h_n(x^0_{i_n}, l^0_{j_n})$ are the prediction errors associated with the motions and the measurements.

The previous Jacobians can be defined as

$$F_i^{i-1} := \frac{\partial f_i(x_{i-1}, \alpha_i)}{\partial x_{i-1}} | x^0_{i-1} \tag{4.9}$$

$$H_n^{i_n} := \frac{\partial h(x_{i_n}, l_{j_n})}{\partial x_{i_n}} | (x_{i_n}^0, l_{j_n}^0) \tag{4.10}$$

$$J_n^{j_n} := \frac{\partial h(x_{i_n}, l_{j_n})}{\partial l_{j_n}} | (x_{i_n}^0, l_{j_n}^0) \tag{4.11}$$

.

Taking into account the previous non-linear equation 4.6 and the linearized motion and measurements models present in equations 4.7 and 4.8, along the previous Jacobians, the equation 4.6 can be rewritten as

$$\delta^* := \underset{\delta}{\operatorname{argmin}}(\sum_{i=1}^{M} ||F_i^{i-1} \partial x_{i-1} - \partial I_{(d \times d)} x_i - a_i||_{\Sigma_i}^2 + \sum_{n=1}^{N} ||H_n^{i_n} \partial x_{i_n} + J_n^{j_n} \partial l_{j_n} - c_n||_{\Lambda_n}^2) \tag{4.12}$$

The δ^* vector is now a linear least-squares problem where d represents the dimension of x_i. Since δ^* is linear, this is now a pure optimization problem that can be solved efficiently.

4.2 SAM - Smoothing and Mapping

Smoothing implementations of SLAM are starting to become state-of-the-art when talking about the localization problem. This is happening because this type of implementation has a big advantage when compared to the old ways of filtering such as the widely known Extended Kalman Filter (EKF). For example, the EKF marginalizes prior information at each iteration making previous data inaccessible in future iterations [29]. This can be a problem since there is no way to correct previous errors that were admitted in the filter, even using more data will not correct these errors.

On the other hand, with smoothing it is always possible to re-linearize subsets of the graph, or even the whole graph, and since the objective behind the smoothing is to find the MAP solution, the smoothing is always concerned in the most probable trajectories and landmarks position. For this reason, the admitted past errors are corrected every time a re-linearization step happens. The EKF is much more computationally efficient when talking about small maps with a low quantity of landmarks since the update complexity increases quadratically with the number of states and landmarks. For bigger maps with an increased quantity of landmarks, smoothing algorithms are preferred since the information matrix can be kept sparse and the marginalization of the information

does not happen, resulting in a more efficient and accurate solution to the localization problem.

As explained in the previous subsection, the equation 4.12 is already a linear least-squares problem that can be solved with smoothing, but since it is not written in a standard way it must be rewritten to be used by least-squares solvers like Gauss-Newton or Levenberg-Marquardt [29].

As seen in [30], if the Jacobian matrices 4.9, 4.10 and 4.11 are collected in a matrix A and prediction errors a_i and c_n are stored in a right-hand side vector b it is possible to rewrite the equation 4.12 as

$$\delta^* := \underset{\delta}{\operatorname{argmin}} ||A\delta - b||_2^2 \tag{4.13}$$

The matrix A can get very big, but it is always quite sparse. If d_x is the number of states and d_l, d_ζ the dimensions of landmarks and measurements the size of A can be calculated by $(Nd_x + Kd_\zeta) \times (Nd_x + Md_l)$ with M being the number of poses, N the number of landmarks and K the number of measurements. For $M = 3$, $N = 2$ and $K = 4$, A and b have the following aspect

$$A = \begin{bmatrix} -I_1 & & & & & \\ F_2^1 & -I_2 & & & & \\ & F_3^2 & -I_3 & & & \\ H_1^1 & & & J_1^1 & & \\ H_2^1 & & & & J_2^2 & \\ & H_3^2 & & J_3^1 & & \\ & & H_4^3 & & J_4^2 & \end{bmatrix}, b = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \tag{4.14}$$

Finally, with the equation 4.13 and the matrices A and b the smoothing problem can be solved by any Least-squares solver and the SLAM problem efficiently resolved.

Esta página foi intencionalmente deixada em branco.

Chapter 5

Project

Taking into consideration the architecture presented in Figure 2.1, this chapter will describe all the decisions made in order to develop the implemented blocks. At the end of the explanations, a proposed architecture of the project will be presented.

5.1 Robotic Platform

The decisions behind the choice of the best robotic platform for the project take into account the fact that the drone will be used to fly inside a warehouse. For this reason, the platform must be of small dimensions while also being reliable and stable. The platform must also possess at least a vision system, composed of a camera and a gimbal, an IMU and an obstacle detection system. To facilitate the development of the project and the possibility of integration in different warehouses it was opted to use a commercial platform (TRL-9) easily available in the market. Since the platform will be a commercial one, it must supply some kind of Software Development Kit (SDK) to access the sensor data coming from the platform and to be able to control it. Taking into account these requirements it was decided that the best platform to use was one from the DJI's Mavic model.

Among the options in the DJI Mavic model the contemplated platforms were, the DJI Mavic Air, the DJI Mavic 2 Pro, the DJI Mavic 2 Zoom, and the DJI Mavic 2 Enterprise. Even though the DJI Mavic Air 2 introduces interesting characteristics to the project there was no available SDK when the architecture planning was made.

The following table represents a comparison between the drones previously mentioned.

Table 5.1: Comparison table between the DJI's Mavic range platforms.

	DJI Mavic Air	**DJI Mavic 2 Pro**	**DJI Mavic 2 Zoom**	**DJI Mavic 2 Enterprise**
Price	850 €	1500 €	1250 €	2300 €
Dimensions (Unfolded)	168×184×64 mm (L×W×H)	322×242×84 mm (L×W×H)	322×242×84 mm (L×W×H)	322×242×84 mm (L×W×H)
Flight Time	21 Min	31 Min	31 Min	31 Min
Maximum Speed	68 kph	72 kph	72 kph	72 kph
Range	4000 m	8000 m	8000 m	8000 m
Frontal Obstacle Detection	Measurement Range: 0.5 - 12 m Detectable Range: 0.5 - 24 m Effective Sensing Speed: ⩽ 8 m/s FOV: Horizontal 50 , Vertical ±19	Measurement Range: 0.5 - 20 m Detectable Range: 20 - 40 m Effective Sensing Speed: ⩽ 14m/s FOV: Horizontal: 40 , Vertical: 70	Measurement Range: 0.5 - 20 m Detectable Range: 20 - 40 m Effective Sensing Speed: ⩽ 14m/s FOV: Horizontal: 40 , Vertical: 70	Measurement Range: 0.5 - 20 m Detectable Range: 20 - 40 m Effective Sensing Speed: ⩽ 14m/s FOV: Horizontal: 40 , Vertical: 70
Backwards Obstacle Detection	Precision Measurement Range: 0.5 - 10 m Detectable Range: 0.5 - 20 m Effective Sensing Speed: ⩽ 8 m/s FOV: Horizontal 50 , Vertical ±19	Precision Measurement Range: 0.5 - 16 m Detectable Range: 16 - 32 m Effective Sensing Speed: ⩽ 12m/s FOV: Horizontal: 60 , Vertical: 77	Precision Measurement Range: 0.5 - 16 m Detectable Range: 16 - 32 m Effective Sensing Speed: ⩽ 12m/s FOV: Horizontal: 60 , Vertical: 77	Precision Measurement Range: 0.5 - 16 m Detectable Range: 16 - 32 m Effective Sensing Speed: ⩽ 12m/s FOV: Horizontal: 60 , Vertical: 77
Downwards Obstacle Detection	Altitude Range: 0.1 - 8 m Operating Range: 0.5 - 30 m	Altitude Range: 0.1 - 8 m Operating Range: 0.5 - 30 m	Altitude Range: 0.1 - 8 m Operating Range: 0.5 - 30 m	Altitude Range: 0.1 - 8 m Operating Range: 0.5 - 30 m
Upwards Obstacle Detection	———	Precision Measurement Range: 0.1 - 8 m	Precision Measurement Range: 0.1 - 8 m	Precision Measurement Range: 0.1 - 8 m
Lateral Obstacle Detection	———	Precision Measurement Range: 0.5 - 10 m Effective Sensing Speed: ⩽ 8 m/s FOV: Horizontal: 80 , Vertical: 65	Precision Measurement Range: 0.5 - 10 m Effective Sensing Speed: ⩽ 8 m/s FOV: Horizontal: 80 , Vertical: 65	Precision Measurement Range: 0.5 - 10 m Effective Sensing Speed: ⩽ 8 m/s FOV: Horizontal: 80 , Vertical: 65
Internal Storage	Micro SD, 8GB Onboard	Micro SD, 8GB Onboard	Micro SD, 8GB Onboard	Micro SD, 24GB Onboard
Camera	12 MP	20 MP	12 MP	12 MP
Sensor	1/2.3" CMOS	1" CMOS	1/2.3" CMOS	1/2.3" CMOS
Lens	FOV: 85 Aperture: f/2.8	FOV: 77 Aperture: f/2.8–f/11	FOV: 83 (24 mm) : 48 (48 mm) Aperture: f/2.8 (24 mm)–f/3.8 (48 mm)	FOV: 83 (24 mm) : 48 (48 mm) Aperture: f/2.8 (24 mm)–f/3.8 (48 mm)
Zoom	Não	Não	2x Digital 2x Electronic	2x Digital 3x Electronic
Gimbal (Mechanical Range)	Tilt: -100 to 22 Roll: -30 to 30 Pan: -12 to 12	Tilt: -135 to +45 Pan: -100 to +100	Tilt: -135 to +45 Pan: -100 to +100	Tilt: -135 to +45 Pan: -100 to +100
Gimbal (Controllable Range)	Tilt: -90 to 0	Tilt: -90 to +30 Pan: -75 to +75	Tilt: -90 to +30 Pan: -75 to +75	Tilt: -90 to +30 Pan: -75 to +75
Max Bitrate	100 Mbps	100 Mbps	100 Mbps	100 Mbps
Battery	2375 mAh	3850 mAh	3850 mAh	3850 mAh
SDK	Mobile, UX and Windows	Mobile, UX and Windows	Mobile, UX and Windows	Mobile, UX and Windows

From the table, it is possible to verify that the DJI Mavic Air has fewer obstacle detections sensors than the Mavic 2 Pro, the Mavic 2 Zoom, and the Mavic 2 Enterprise, for this reason, this platform was the first one to be discarded. From the remaining three, the one that was chosen was the DJI Mavic 2 Zoom since it has the lowest price among all of them while also maintaining the required specifications.

5.2 SDK

Considering that the chosen robotic platform was the Mavic 2 Zoom, it was also necessary to explore the SDK available for the drone. The available SDK were the Mobile, the UX, and the Windows.

The mobile SDK was created to be used in mobile devices, such as Android and iOS smartphones. With this SDK, a developer can control the drone using high or low-level commands like velocities or orientations, have access to the drone telemetry data and live video feed from the main camera, control the gimbal and its telemetry, obtain information about detected obstacles, use commands such as take-off and landing, etc.

The Windows SDK was created to be used in devices that run the Windows Operating System (OS), it has almost all functions of the mobile SDK with the big disadvantage that is unable to control the drone using high or low-level flight commands. This disadvantage renders this SDK useless for the main purpose of the project.

The UX SDK was also created to be used in mobile devices but this one uses the official DJI application as a development sandbox. Even though this SDK would make the development of an application faster and easier it could bring future problems given the fact that it uses the official application as a sandbox.

Based on the information previously described, the SDK that follows the requirements is the Mobile SDK.

5.3 Drone and Ground Station Communication

Since it is impossible to have onboard processing running in the commercial DJI drones it was decided to use a ground station to run all the localization and navigation algorithms.

To connect the smartphone to the drone, it was required to connect the device via Universal Serial Bus (USB) to the remote control that comes with the platform. This requirement is due to the fact that the vehicle only communicates via smartphone. The remote controller communicates with the drone using the OcuSync 2.0 wireless communication protocol, this is a propriety communication protocol from DJI. Since the USB port from the smartphone was already connected to the remote controller it was decided to use a wireless access point between the mobile device and the ground station. In order to have an environment of development of new features to the drone, a software layer able to integrate the framework ROS [32] in the application was developed. This integration was perform by using the ROSJAVA [33] Bridge between the android OS and the Linux Ground Station.

5.4 Localization Algorithm

Considering the sensor data accessible in the drone and the main objective of implementing the project in a warehouse scenario, it was decided that the developed localization algorithm would consist of a fusion between the motion data coming from the drone and artificial markers present in the warehouse.

The goal behind the localization algorithm is being capable of doing hybrid visual SLAM where instead of using natural features to locate the vehicle while there are no artificial markers visible, the algorithm will use the velocity and orientation data coming from the drone. Since the onboard algorithms that calculate the velocity and orientation do not perform any kind of loop closing, the velocity and orientation data will suffer drift over time, but this drift will be adjusted every time the camera finds an artificial marker.

Considering that the warehouse can be a big map with lots of artificial markers a Graph-SLAM approach will be used for the reasons explained in section 4.2. To create the factor graph and solve the Graph-SLAM problem the GTSAM framework will be used. Since this framework only creates the factor graph, the Incremental Smoothing and Mapping 2 (iSAM2) [34] algorithm is used to solve the optimization problem.

The indoor localization processing pipeline architecture is present in Figure 5.1.

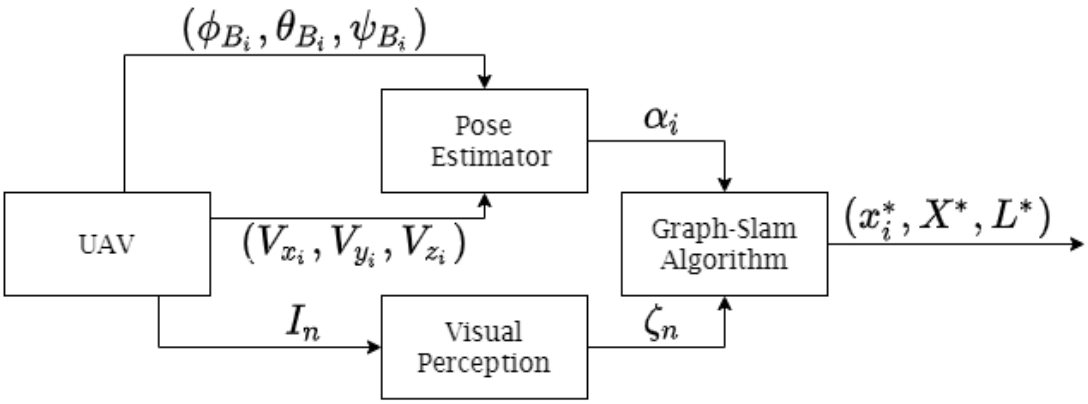

Figure 5.1: Localization algorithm architecture.

With respect to the artificial markers, it was decided to explore the AprilTags, described in section 3.3.1, since from the state-of-art, these Tags are the ones that have presented higher performance for real-time localization.

To detect these AprilTags and calculate the relative positions between the drone and the marker, the AprilTag Detector present in the subsection 3.3.3 was applied. This was

the chosen algorithm since it has a high relative position accuracy, a low false-positive ratio, and is capable of running in real-time.

Since GTSAM only accepts poses and relative poses, a Pose Estimator block was also required. This block integrates the velocities coming from the drone transforming them into positions, joining them with the orientations. The final result is a pose that can be used by the GTSAM framework.

Finally, as explained before, the Graph-SLAM Algorithm block will consist in the creation of a factor graph using the GTSAM framework, the poses coming from the Pose Estimator, and the relative poses between the drone and the artificial markers coming from the AprilTag Detector. This block will also take care of the optimization using the iSAM2 algorithm. The output of the block will be an optimized pose, an optimized trajectory, and an optimized set of landmarks which will be the detected visual markers.

Additionally, the UcoSLAM algorithm presented in the state-of-the-art will be used to compare the results of the developed algorithm with the results of a state-of-the-art algorithm.

5.5 Proposed Architecture

In figure 5.2 it is presented the proposed architecture for the project. The blue blocks represent the part that is already developed by DJI, the red blocks represent the perception and localization part developed and present by the author of this thesis, and the green blocks are the guidance navigation and control part developed by another person and presented in another thesis. The yellow blocks were implemented by both the author and the other person. Regarding the tf block, all the measurement transformations were made solely by the author, and the remaining ones were developed by both. This low-level architecture has the navigation part simplified, for the non-simplified architecture see Appendix A.

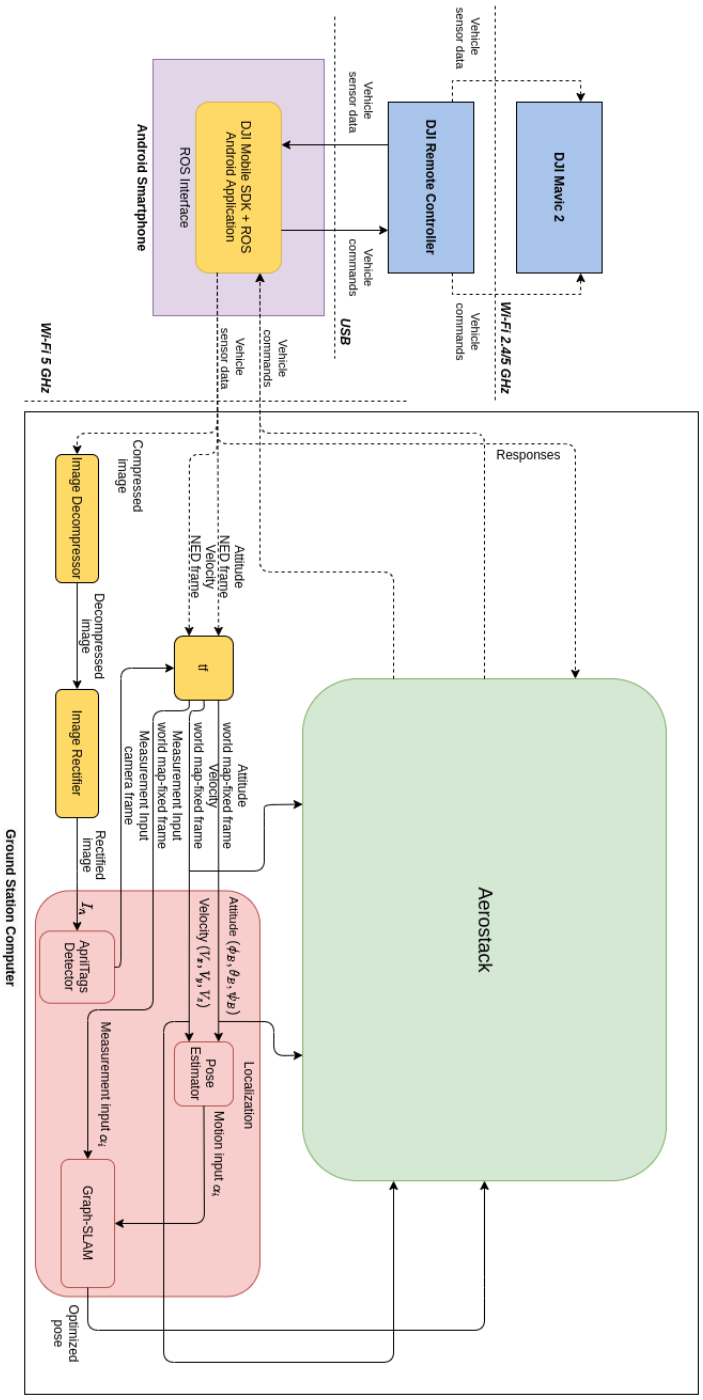

Figure 5.2: Proposed Architecture.

Chapter 6

Implementation

In this chapter it will be presented the algorithms and applications developed during this thesis, namely the localization algorithm and the android application used to communicate with the drone.

6.1 Mobile SDK + ROS Application

The developed android application uses the DJI Mobile SDK to communicate with the remote controller that in turn communicates with the drone. The ROS framework is used to transmit data between the ground station and the mobile device. The communication between the smartphone and remote controller is made using USB while the interface between the smartphone and the ground station is made using a 5 GHz Wi-Fi access point.

To develop the android application the Android Studio Integrated Development Environment (IDE) was used together with the Mobile SDK version 4.14 and the ROSJava version 0.4. Note that even though the latest version for the Mobile SDK is 4.15, it was required to use the 4.14 version because there are unresolved dependencies conflicts between the 4.15 version and the RosJava.

DJI makes available a sample code [35] for the Mobile SDK that already creates the USB connection and receives the video feed coming from the main camera, for this reason, this code was used as a start point for the developed application. Examples of what is already developed in the sample code are shown in the Figures 6.1 and 6.2.

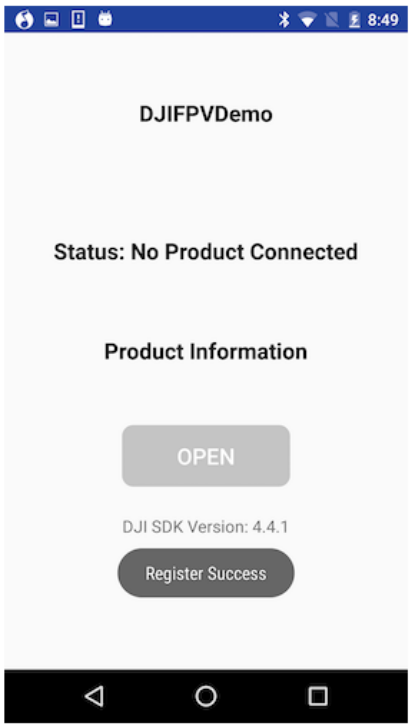

Figure 6.1: Sample code USB connection layout [35].

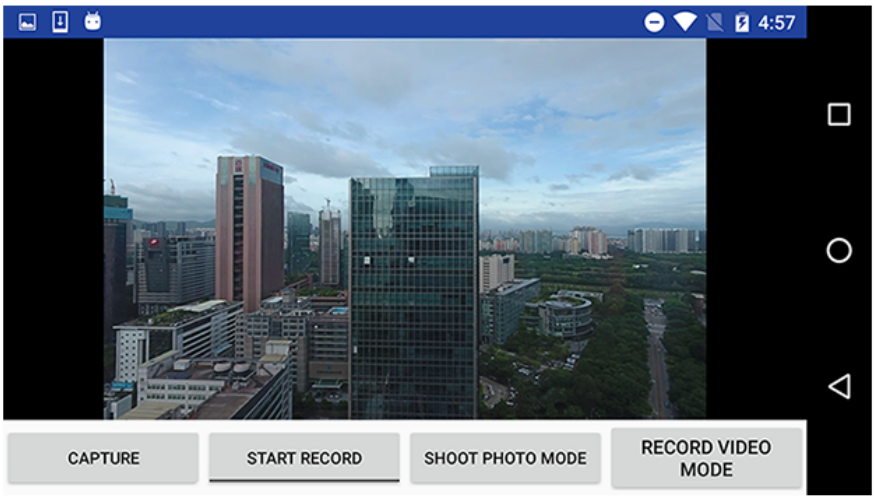

Figure 6.2: Sample code video feed layout [35].

Note that DJI has a policy where the first time the user opens a developed application the smartphone must have access to the internet to register the application in the DJI

database. The registration is done using an access token that the developer must redeem logging in the DJI Developer Website [36]. After the first time, the application does not require an internet connection again.

The first change made in the sample code was the ability to make the connection to the roscore running in the ground station. This step is needed to share data between the ground station and the mobile application. The connection is performed once the user clicks on the OPEN button present in Figure 6.1.

In order the establish the ROS connection to the ROS master running the roscore the lines of code presented in the Listing 6.1 were applied.

```
1 NodeConfiguration nodeConfiguration = NodeConfiguration.newPublic(
      getRosHostname());
2 nodeConfiguration.setMasterUri(getMasterUri());
```

Listing 6.1: Establishing the connection to the ROS master.

The next step after making the ROS connection between the smartphone and the ground station was to create the publishers responsible for sending the sensor data to the ground station. In the RosJava library, there are two ways of creating publishers, one is made by declaring the publisher as a Publisher type variable defining the name and type of the topic to be published the other way is by defining the publisher exactly like the previous one but associating a thread running at the desired frequency to the created publisher. The Listings 6.2 and 6.3 manifest the two ways of creating a publisher.

```
1 Publisher<std_msgs.Int32> Responses = connectedNode.newPublisher("drone0/
      Responses", "std_msgs/Int32");
```

Listing 6.2: Creating a publisher without an associated thread.

```
1 final Publisher<geometry_msgs.QuaternionStamped> Velocity = connectedNode.
      newPublisher("drone0/Velocity",
2                 "geometry_msgs/QuaternionStamped");
3     connectedNode.executeCancellableLoop(new CancellableLoop() {
4         protected void loop() throws InterruptedException {
5             Get_Velocity_Altitude();
6             geometry_msgs.QuaternionStamped Value = (geometry_msgs.
                  QuaternionStamped) Velocity.newMessage();
7             geometry_msgs.Quaternion Value_Point = Value.getQuaternion();
8             std_msgs.Header Head = Value.getHeader();
9             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 6.3: Creating a publisher with an associated thread.

Taking into account that one must ask the drone for the sensor data, it becomes really useful to assign a thread to the publisher. The thread objective is to ask the drone for the desired data and send it to the ground station periodically. For this reason, every publisher associated with sensor data will be created with an associated thread.

Alongside the sensor data publishers, two extra topics were also created, these publishers were responsible for sending information about eventual commands sent to the drone, that is, if the command was completed or not. One of these topics was created to be used by eventual algorithms in the ground station, the other was created for debugging purposes. Since these topics do not need to ask for any type of sensor data, they were created without an associated thread.

As already said, to get the vehicle sensor data one must ask for it, this is done by inquiring about the module responsible for the sensor state as shown in the Listing 6.4. All the information about modules, sensors, and how to access them is available in the DJI android Application Programming Interface (API) documentation [37].

```
private void Get_Aircraft_Attitude() {
        FlightController flightController = ((Aircraft) product).
            getFlightController();
        Roll_Drone = flightController.getState().getAttitude().roll;
        Pitch_Drone = flightController.getState().getAttitude().pitch;
        Yaw_Drone = flightController.getState().getAttitude().yaw;
    }
```

Listing 6.4: Example of how to obtain the drone attitude.

In RosJava it is also possible to define ROS subscribers. This is done by declaring a Subscribe type variable defining the name and type of the topic to be subscribed. It is

also possible to associate a callback to the created subscriber, this is useful if one desires to run code once a new message is received. In the developed application one subscriber was needed to receive eventual commands to be executed by the vehicle. The code used to create the subscriber and the callback is presented in the Listing 6.5.

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

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

Listing 6.5: Subscriber and callback responsible for receiving aircraft commands.

A Parse_Command function was created to detect what command the drone must execute. The code in the Listing 6.6 represents the referred 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));
```

```
23      }
24      else if (Command.startsWith("Gimbal")) {
25          Rotate_Gimbal(Command);
26      }
27      else if (Command.startsWith("Velocity")) {
28          Send_Flight_Control_Data_To_Aircraft(Command);
29      }
30      else if (Command.equals("Take Photo"))
31      {
32          Take_Photo();
33      }
34      else {
35          Send_Command_Result(0);
36          Send_Debug_Message("Invalid Command");
37      }
38  }
```

Listing 6.6: Parse_Command function.

Similar to the process of receive sensor data, to send commands to the drone one must send the command to the module responsible for the action that the user wants to execute. The example in the Listing 6.6 sends a velocity command to the vehicle.

```
1  FlightControlData velocity_data;
2  velocity_data = new FlightControlData((float) Velocity_X, (float) Velocity_Y, (
       float) Velocity_Yaw, (float) Velocity_Z);
3  ((Aircraft) product).getFlightController().sendVirtualStickFlightControlData(
       velocity_data, djiError -> {
4      if (djiError == null) {
5          Send_Command_Result(1);
6          Send_Debug_Message("Velocity Received");
7      } else {
8          Send_Command_Result(0);
9          Send_Debug_Message("Velocity Error");
10     }
11
12 });
```

Listing 6.7: Example of how to send a velocity to the aircraft.

Alongside the sensor data, the application also transmits the video feed of the main camera. Since the DJI sample code already receives the uncompressed video feed, it

was only needed to transmit an array of bytes containing each frame of it. Each frame of the video is uncompressed as a java SurfaceTexture variable type, but since RosJava only accepts ChannelBufferOutputStreams variables as ROS compressed images, it was required to convert the previous SurfaceTexture in a ChannelBufferOutputStream. The first step was to convert the SurfaceTexture in a Bitmap with a resolution of 1280x720 pixels, the second step was to convert the previous Bitmap in a byte array, this is done by compressing the image frame as a JPEG image, in this case, the JPEG image was compressed with 90% of quality to achieve the lowest transmission delay possible while also maintaining a good frame quality. Once the image is converted in a byte array, a flag is set to true to indicate that a new frame is ready to be sent by another thread. The Listing 6.8 expresses the previously explained process.

```
1 public void onSurfaceTextureUpdated(SurfaceTexture surface) {
2     Bitmap image;
3     image = mVideoSurface.getBitmap();
4     Bitmap Image = getResizedBitmap(image, 1280, 720);
5     ByteArrayOutputStream stream = new ByteArrayOutputStream();
6     Image.compress(Bitmap.CompressFormat.JPEG, 90, stream);
7     byteArray = stream.toByteArray();
8     flag_done = true;
9 }
```

Listing 6.8: Conversion of a SurfaceTexture in a byte array.

Note that, the onSurfaceTextureUpdated function works as a callback, that is, every time a new video frame is received the code inside this function is automatically executed.

The last step is for the publisher thread to access the latest image frame and transmit it to the ground station via ROS. The thread is running every 60 milliseconds and it only sends the image frame if a new one is available. To transform the byte array in a ChannelBufferOutputStream, the previous byte array is first transformed in a Little Endian ChannelBuffer and finally converted in the needed ChannelBufferOutputStream. After the image frame is a ChannelBufferOutputStream variable type the last step is to transmit it via ROS. This is done using the code presented in the Listing 6.9.

```
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 6.9: Transmission of a video feed image frame.

The Tables 6.1 and 6.2 express all the developed ROS topics and the programmed drone actions.

Table 6.1: Presentation and explanation of all the developed ROS topics.

Topic Name	Topic Type	Topic Explanation	Topic Frequency (Hz)
/drone0/Attitude_RPY	geometry_msgs/PointStamped	Transmits the drone attitude as Roll, Pitch and Yaw in degrees. x: roll; y: pitch; z: yaw.	20
/drone0/Battery	sensor_msgs/BatteryState	Transmits the real time battery state.	1
/drone0/Gimbal_Attitude	geometry_msgs/PointStamped	Transmits the absolute gimbal attitude as Roll, Pitch and Yaw in degrees. x: roll; y: pitch; z: yaw.	20
/drone0/Velocity	geometry_msgs/QuaternionStamped	Transmits the velocity and absolute altitude of the drone in m/s and m. x: velocity in x; y: velocity in y; z: velocity in z w: altitude.	20
/drone0/Video_Feed	sensor_msgs/CompressedImage	Transmits the image frame.	16
drone0/Responses	std_msgs/Int32	Publishes 0 if command failed to be executed. Publishes 1 if command is executed successfully.	———
drone0/Responses_Debug	std_msgs/String	Publishes a debug message explaining why the command failed.	———
drone0/Mavic_Commands	std_msgs/String	Receives the commands that the drone must execute.	———

Table 6.2: Presentation and explanation of all the implemented vehicle commands.

Command String	Arguments	Response	Description
Motor X	X=On; X=Off	0=failed 1=success	Turn the motors on or off.
Takeoff	———————-	0=failed 1=success	Starts the takeoff of the drone.
Land	———————-	0=failed 1=success	Starts the landing of the drone.
Joystick X	X=On; X=Off	0=failed 1=success	Activates or deactivates the joystick mode in order to control the drone using the ground station.
SetUp X Y Z W	X=0; X=1 (Vertical Control Mode - 0=Velocity; 1=Position) Y=0; Y=1 (Roll Pitch Control Mode - 0=Angle; 1=Velocity) Z=0; Z=1 (Yaw Control Mode - 0=Angle; 1=Angular Velocity) W=0; W=1 (Flight Coordinate System - 0=Ground; 1=Body)	0=failed 1=success	Set up the drone to use the wanted configuration.
Gimbal X Y Z W	X=Roll Y=Pitch Z=Yaw W=Time	0=failed 1=success	Rotates the gimbal by the desired relative roll,pitch and yaw in the desired time. The angle values are expected to be in degrees and the time in seconds.
Velocity X Y Z W	X=Velocity X; Y=Velocity Y; Z=Velocity Z; W=Angular Velocity Yaw;	0=failed 1=success	Controls the velocity of the drone in the X,Y and Z directions and the angular velocity of the yaw angle. The velocities are expected to be in meters per second and the angular velocity in degrees per second.
Take Photo	————————————————————	0=failed 1=success	Takes a photo and stores it in the memory.

6.2 Decompression Calibration and Rectification

As seen previously the image frames reach the ground station as compressed JPEG images via the topic /drone0/Video_Feed. For these frames to be usable in the localization algorithm they must be uncompressed and rectified, and for this, the camera must be calibrated to obtain the intrinsic and lens distortion parameters.

Regarding the decompression of the image frames, the OpenCV [38] framework already has tools capable of doing it. The lines of code in the Listing 6.10 represent the decompression of the image frames.

```
1 Mat image = cv::imdecode(cv::Mat(image_compressed.data),IMREAD_ANYCOLOR);
2 std_msgs::Header header=image_compressed.header;
3 cv_bridge::CvImage img_bridge = cv_bridge::CvImage(header, sensor_msgs::
      image_encodings::BGR8, image);
4 img_bridge.toImageMsg(raw_video_msg);
```

Listing 6.10: Decompression of the video feed.

The uncompressed image is then published in a topic named /drone0/sensor_measurements/camera/image_raw of type /sensor_msgs/Image.

Concerning the calibration of the camera, initially the Single Camera Calibrator Application [39] from Matlab was used to obtain the necessary parameters. After three calibrations it was concluded that the estimation of the lens distortion parameters was not correct, for this reason, another tool to acquire the parameters was used.

The tool used to realize the calibration was the one that is available in the UcoSlam framework. This tool uses a bundle of ArUco markers to calculate the intrinsic and distortion parameters. The Figures 6.3 and 6.4 represent the bundle of markers and the Graphical User Interface (GUI) used to perform the calibration.

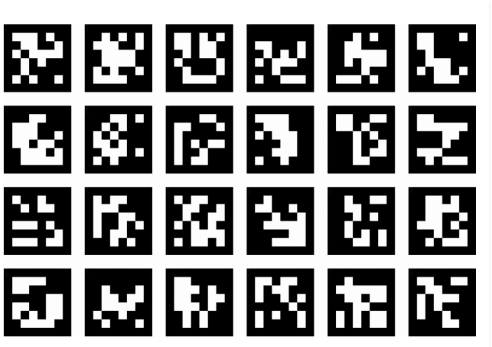

Figure 6.3: ArUco markers bundle used to perform the calibration.

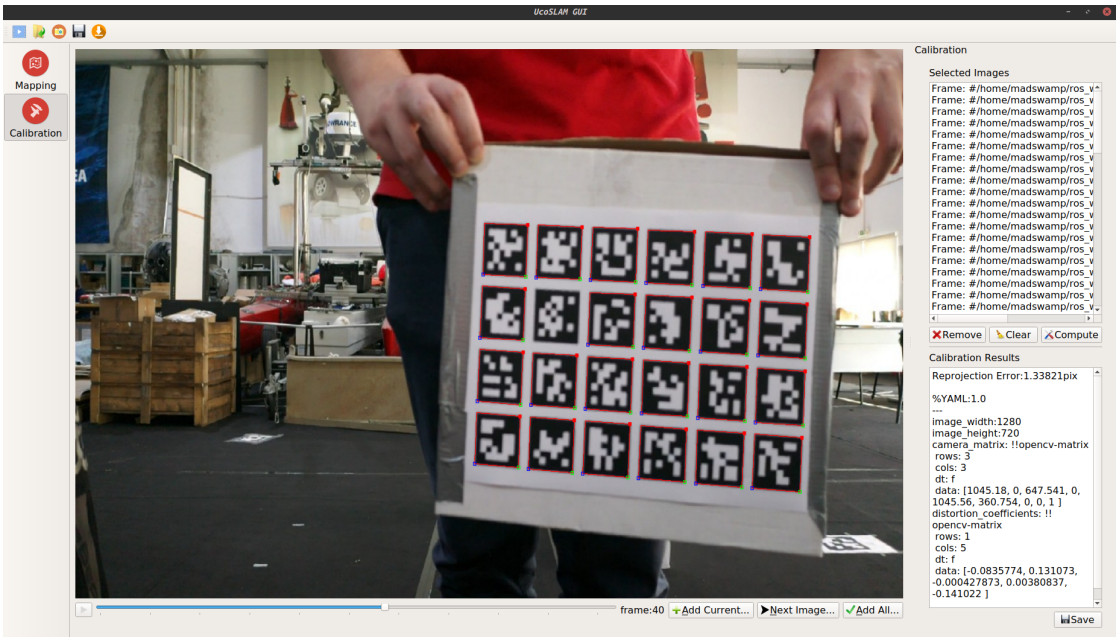

Figure 6.4: Tool used the perform the camera calibration.

Using the previous tool to calibrate the camera it was possible to obtain usable intrinsic and lens distortion parameters. These parameters are then published in a topic named /drone0/sensor_measurements/camera/camera_info of type /sensor_msgs/CameraInfo.

Regarding the rectification of the image, the image_proc [40] package available in the ROS framework is used. This package receives as inputs the distorted image and the camera parameters, outputting the undistorted image frame. This image frame is then published in a topic named /drone0/sensor_measurements/camera/image_rect of the same type.

6.3 Frame Transformation and Pose Estimator

The coordinate system in which the localization was designed to work, is a system fixed at the origin of the trajectory, that is, the first pose of the drone when the localization algorithm starts. This coordinate system makes it easier to analyze the results without compromising the accuracy of the localization. From this point on this coordinate system will be called **origin** and it is shown in the Figure 6.5. Note that all the coordinate system transformations will be implemented using the tf [41] framework present in ROS.

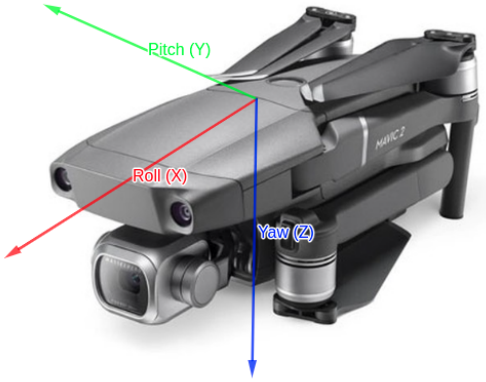

Figure 6.5: Coordinate system used for the localization algorithm.

Since DJI velocity and orientation data is given in North East Down (NED) coordinates, a frame that represents the NED system was also created, this frame was named **ned**. The average of 50 measurements was used to determine the yaw angle that relates the **origin** frame to the **ned** frame. The Listing 6.11 explains this process.

```
1  if(count_average==50)
2  {
3      geometry_msgs::TransformStamped static_transform_message;
4      yaw_frame=yaw_frame/count_average;
5
6      yaw_frame=yaw_frame*M_PI/180;
7
8      static_transform_message.header.stamp=ros::Time::now();
9      static_transform_message.header.frame_id="origin";
10     static_transform_message.child_frame_id="ned";
11
12     static_transform_message.transform.translation.x=0;
13     static_transform_message.transform.translation.y=0;
14     static_transform_message.transform.translation.z=0;
15
16     tf2::Quaternion q;
17     q.setRPY(0, 0, -yaw_frame);
18
19     static_transform_message.transform.rotation.x=q.x();
20     static_transform_message.transform.rotation.y=q.y();
21     static_transform_message.transform.rotation.z=q.z();
```

```
22      static_transform_message.transform.rotation.w=q.w();
23      tf_broadcaster_frame.sendTransform(static_transform_message)
24  }
25  else
26  {
27      yaw_frame=yaw_frame+attitude->point.z;
28      count_average++;
29  }
```

Listing 6.11: Creating of the **ned** frame regarding the **origin** frame.

The creation of both these frames makes it easier to convert the velocity and attitude data coming from DJI to the main coordinate system, that is, the **origin** system.

The next step was to calculate a raw estimation of the drone localization using only the transformed velocities and orientations given by DJI. The raw data from the estimated pose will be later fed in the Graph-SLAM algorithm as motion inputs α_i. This raw estimation represents the pose estimator block of the low-level architecture and it is done by integrating the DJI transformed velocities along the time. The integrated translation is then joined with the transformed orientation giving origin to the estimated raw pose. This raw pose is published in a topic named /drone0/raw_localization/raw_pose of type geometry_msgs/PoseStamped as shown in the Listing 6.12.

```
1
2   void mavic_raw_localization::velocity_callback(const geometry_msgs::
        QuaternionStampedConstPtr& msg)
3   {
4       if(!flag_velocity)
5       {
6           prev_integration_time=ros::Time::now();
7           velocity_integrator.point.x=0;
8           velocity_integrator.point.y=0;
9           velocity_integrator.point.z=0;
10          flag_velocity=true;
11      }
12      else
13      {
14
15          ros::Time current_integration_time=ros::Time::now();
16          double diffTime=(current_integration_time - prev_integration_time).nsec
                / 1E9;
```

```
17          velocity_integrator.point.x=velocity_integrator.point.x+msg->quaternion.
                x*diffTime;
18          velocity_integrator.point.y=velocity_integrator.point.y+msg->quaternion.
                y*diffTime;
19          velocity_integrator.point.z=velocity_integrator.point.z+msg->quaternion.
                z*diffTime;
20
21          prev_integration_time=current_integration_time;
22
23          geometry_msgs::PoseStamped pose_msg;
24          tf2::Quaternion raw_attitude_aircraft;
25          raw_attitude_aircraft.setRPY(attitude_data.point.x*M_PI/180,
                attitude_data.point.y*M_PI/180,attitude_data.point.z*M_PI/180);
26
27
28          pose_msg.header.frame_id="origin";
29          pose_msg.header.stamp=ros::Time::now();
30          pose_msg.pose.orientation.x=raw_attitude_aircraft.x();
31          pose_msg.pose.orientation.y=raw_attitude_aircraft.y();
32          pose_msg.pose.orientation.z=raw_attitude_aircraft.z();
33          pose_msg.pose.orientation.w=raw_attitude_aircraft.w();
34          pose_msg.pose.position=velocity_integrator.point;
35          raw_pose_pub.publish(pose_msg);
36      }
37
38  }
```

Listing 6.12: Calculation of the raw pose of the vehicle.

Since the Graph-SLAM algorithm needs the raw pose of the landmarks in the **origin** frame to be able to estimate the optimized position of the drone, it was also required to create a frame that expresses the raw localization of the drone, this frame is used to declare the position where the landmark is seen referent to the **origin**. This frame is called **drone0** and the transformation from this frame to the **origin** frame is given by the calculated raw pose, this process is exemplified in the Listing 6.13.

```
static tf2_ros::TransformBroadcaster tf;
geometry_msgs::TransformStamped transformStamped;

transformStamped.header.stamp=ros::Time::now();
transformStamped.header.frame_id="origin";
transformStamped.child_frame_id="drone0";
transformStamped.transform.translation.x=velocity_integrator.point.x;
transformStamped.transform.translation.y=velocity_integrator.point.y;
transformStamped.transform.translation.z=velocity_integrator.point.z;

transformStamped.transform.rotation.x=raw_attitude_aircraft.x();
transformStamped.transform.rotation.y=raw_attitude_aircraft.y();
transformStamped.transform.rotation.z=raw_attitude_aircraft.z();
transformStamped.transform.rotation.w=raw_attitude_aircraft.w();
tf.sendTransform(transformStamped);
```

Listing 6.13: Creation of the **drone0** frame.

The drone main camera is placed in a gimbal, for this reason, the image frames are rotated when compared to the vehicle body, this rotation is given by the relative attitude of the gimbal. The relative attitude that is provided by DJI SDK, makes it possible to create a frame that relates the gimbal with the drone. The gimbal coordinate system is expressed in the Figure 6.6.

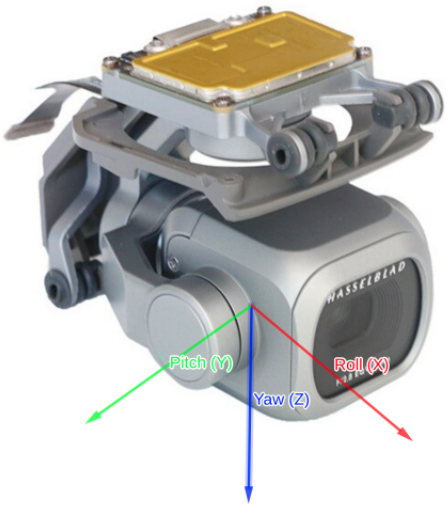

Figure 6.6: Coordinate system used by DJI gimbal.

Note that, since the translation between the center of the drone body and the center of the gimbal was not calculated, it is considered that the gimbal is exactly in the center of the vehicle. The frame that represents the gimbal was named **drone0_gimbal** and it was created using the code in the Listing 6.14.

```
static tf2_ros::TransformBroadcaster tf;
geometry_msgs::TransformStamped transformStamped;
transformStamped.header.frame_id="drone0";
transformStamped.child_frame_id="drone0_gimbal";
transformStamped.transform.translation.x=0;
transformStamped.transform.translation.y=0;
transformStamped.transform.translation.z=0;
attitude_gimbal.setRPY(gimbal_data.point.x*M_PI/180,gimbal_data.point.y*M_PI
    /180,gimbal_data.point.z*M_PI/180);
transformStamped.transform.rotation.x=attitude_gimbal.x();
transformStamped.transform.rotation.y=attitude_gimbal.y();
transformStamped.transform.rotation.z=attitude_gimbal.z();
transformStamped.transform.rotation.w=attitude_gimbal.w();
tf.sendTransform(transformStamped);
```

Listing 6.14: Creation of the **drone0_gimbal** frame.

As present in Figure 6.7, the coordinate system of the gimbal is different from the coordinate system of an image, for this reason, a transformation between them was also necessary.

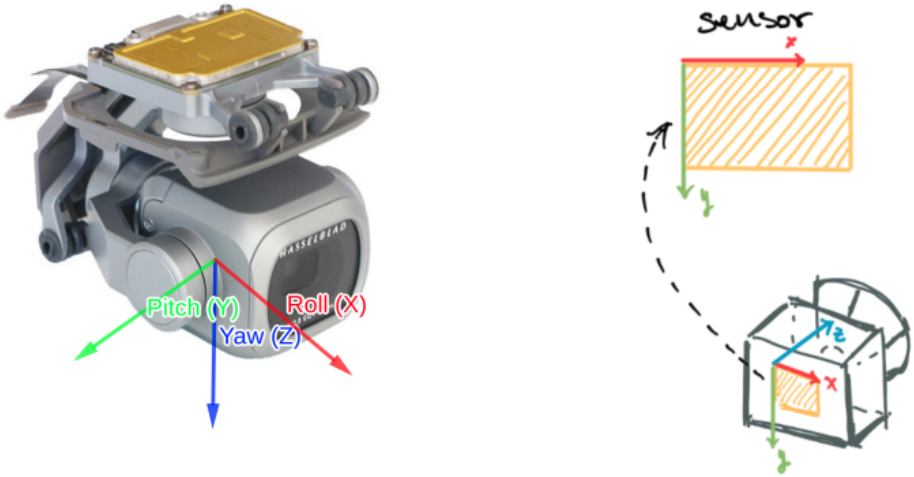

Figure 6.7: Difference between the coordinate systems of a DJI gimbal and an image [42].

Taking as origin the leftmost pixels of an image, the coordinate system is expressed as the X-axis going along the width of the image, the Y-axis going along the height, and the Z-axis, also called the depth axis, going outside the image. Given this, the rotation that transforms the gimbal coordinate system in the image coordinate system is a rotation of 90 degrees in the pitch axis of the gimbal followed by a 90 degrees rotation in the yaw axis. The created frame that represents the image was named **drone0_image** and the creation of this frame is presented in the Listing 6.15.

```
transformStamped.header.frame_id="drone0_gimbal";
transformStamped.child_frame_id="drone0_image";

transformStamped.transform.translation.x=0;
transformStamped.transform.translation.y=0;
transformStamped.transform.translation.z=0;

tf2::Quaternion q0,q1;
q0.setRPY(0,90*M_PI/180,0);
q1.setRPY(0,0,90*M_PI/180);

rotation_gimbal_image=q0*q1;
transformStamped.transform.rotation.x=rotation_gimbal_image.x();
transformStamped.transform.rotation.y=rotation_gimbal_image.y();
transformStamped.transform.rotation.z=rotation_gimbal_image.z();
transformStamped.transform.rotation.w=rotation_gimbal_image.w();
tf.sendTransform(transformStamped);
```

Listing 6.15: Creation of the **drone0_image** frame.

6.4 AprilTags Detector

The algorithm used to detect the AprilTags was the one explained in 5.4 but although this algorithm has state of art results and it is already available as a ROS package, it was still necessary to do some changes to the source code. The first change implemented in the detector was the ability to create a frame for each detected AprilTag. The frame of each AprilTag is created using the relative pose between the marker and the camera, note that the relative pose is already calculated by the detector algorithm. For distances above 4 meters, the calculated relative pose starts to lose accuracy, for this reason only markers with lower distances are considered in the localization algorithm. This first add

on is exemplified in the Listing 6.16.

```
for (unsigned int i=0; i<tag_detection_array.detections.size(); i++)
{
    double x=0,y=0,z=0;
    x=tag_detection_array.detections[i].pose.pose.pose.position.x;
    y=tag_detection_array.detections[i].pose.pose.pose.position.y;
    z=tag_detection_array.detections[i].pose.pose.pose.position.z;
    if(sqrt(pow(x,2)+pow(y,2)+pow(z,2))<4)
    {
        geometry_msgs::TransformStamped tag_transform;
        tag_transform.header.frame_id="drone0_image";
        tag_transform.child_frame_id=marker_id[i];
        tag_transform.header.stamp=ros::Time::now();
        tag_transform.transform.translation.x=tag_detection_array.detections[i].
            pose.pose.pose.position.x;
        tag_transform.transform.translation.y=tag_detection_array.detections[i].
            pose.pose.pose.position.y;
        tag_transform.transform.translation.z=tag_detection_array.detections[i].
            pose.pose.pose.position.z;
        tag_transform.transform.rotation.x=tag_detection_array.detections[i].
            pose.pose.pose.orientation.x;
        tag_transform.transform.rotation.y=tag_detection_array.detections[i].
            pose.pose.pose.orientation.y;
        tag_transform.transform.rotation.z=tag_detection_array.detections[i].
            pose.pose.pose.orientation.z;
        tag_transform.transform.rotation.w=tag_detection_array.detections[i].
            pose.pose.pose.orientation.w;
        tf_broadcaster_.sendTransform(tag_transform);
    }
}
```

Listing 6.16: Creation of a frame for a detected marker.

The second change was the development of a topic that publishes the id of all the detected tags in a given image frame alongside the relative pose between the marker and the camera, note that even though this relative pose is not used, it is published because of eventual future developments. The id automatically solves the data association that is a requirement for the localization algorithm. The published topic is named /drone0/tag_detection_array and its type is nav_msgs/Path. The publication of this topic is shown in the Listing 6.17.

```
for (unsigned int i=0; i<tag_detection_array.detections.size(); i++)
{
    tag_detected_mine.pose.position.x=tag_detection_array.detections[i].pose.
        pose.pose.position.x;
    tag_detected_mine.pose.position.y=tag_detection_array.detections[i].pose.
        pose.pose.position.y;
    tag_detected_mine.pose.position.z=tag_detection_array.detections[i].pose.
        pose.pose.position.z;
    tag_detected_mine.pose.orientation.x=tag_detection_array.detections[i].pose.
        pose.pose.orientation.x;
    tag_detected_mine.pose.orientation.y=tag_detection_array.detections[i].pose.
        pose.pose.orientation.y;
    tag_detected_mine.pose.orientation.z=tag_detection_array.detections[i].pose.
        pose.pose.orientation.z;
    tag_detected_mine.pose.orientation.w=tag_detection_array.detections[i].pose.
        pose.pose.orientation.w;

    if(detection_names[i].find("tag")<-1)
    {
        tag_detected_mine.header.seq=std::stoi(detection_names[i].substr(4,
            detection_names[i].length()));
        array_tag_id.data.push_back(std::stoi(detection_names[i].substr(4,
            detection_names[i].length())));

    }
    tag_detection_array_mine.poses.push_back(tag_detected_mine);
}
```

Listing 6.17: Publication of the marker id along side the relative pose.

The AprilTag detector provides the possibility of detecting standalone markers or a bundle of markers. The big advantage of using a bundle of markers is the possibility of having N X 4 points to estimate the relative pose, where N represents the number of markers in the bundle. The more points the algorithm has to estimate the relative pose, the more accurate the calculations are, for this reason, if accuracy is desired it is preferable to use bundles of markers. The visual difference between the standalone markers and the bundle of markers is shown in the Figures 6.8 and 6.9.

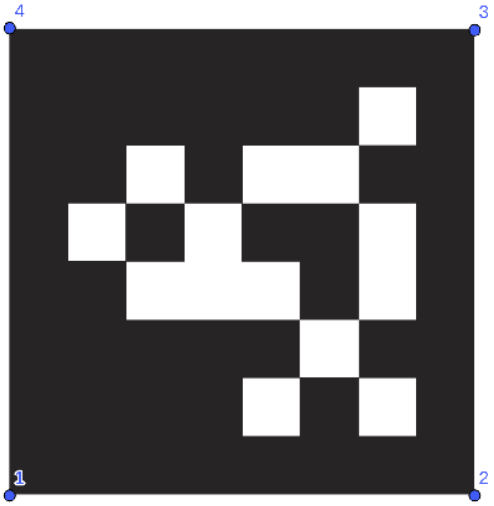

Figure 6.8: Standalone AprilTag.

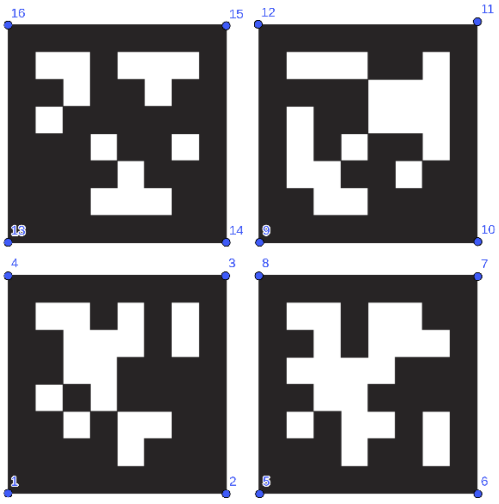

Figure 6.9: Bundle of AprilTags.

When using standalone markers the origin of the coordinate system of the markers is placed right in the middle of the marker, with the X-axis pointing to the right of the marker, the Y-axis pointing to theupper part of the marker, and the Z-axis pointing to the outside of the markers. This coordinate system is shown in Figure 6.10.

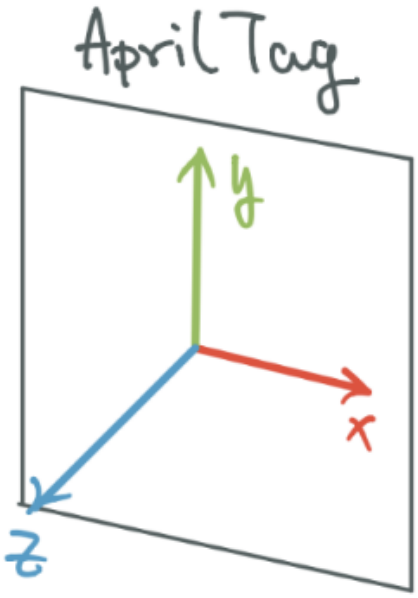

Figure 6.10: AprilTag coordinate system.

The coordinate system of a bundle is the same as a standalone marker, but in this case, it is centered in a marker called master tag, that is, the center of this marker is the center of the reference system of the bundle. The pose of all the other tags in the bundle is the relative pose between the center of the tag and the center of the master tag. This relative pose is important because it allows a better relative pose estimation between the camera and the bundle yielding better detection results. To estimate the relative pose between the master tag and all the other tags, an initial calibration of the bundle is crucial. The framework used to detect the tags already provides an algorithm capable of doing all the calibration. As explained previously, the detection of bundles of tags is more accurate than the detection of standalone markers. Unfortunately, during the development, it was noticed that the loss in size, to be able to fit four tags in an A4 sheet, made the accuracy of the detection worse when compared to the detection of standalone markers, for this reason, standalone markers were used in the final tests.

To use the detector it was required to create a YAML file with the size of each tag and the id, this file was named tags.yaml and it has the aspect shown in the Listing 6.18.

```
standalone_tags:
  [
            {id: 84, size: 0.190},
            {id: 85, size: 0.190},
            {id: 86, size: 0.190},
            {id: 87, size: 0.190},
            {id: 88, size: 0.190},
            {id: 89, size: 0.190},
            {id: 90, size: 0.190},
            {id: 91, size: 0.190},
            {id: 92, size: 0.190},
            {id: 93, size: 0.190}

  ]
```

Listing 6.18: Example of the tags.yaml file.

Alongside the tags.yaml file, the detector also uses a setting.yaml file to configure certain parameters of the algorithm. This file indicates the type of tag family to be detected, the number of threads to use, if it should do the detection faster with less accuracy or slower with more accuracy, etc. It was also needed to add two extra parameters to the settings.yaml file to indicate the vehicle name, in this case, drone0, and if the algorithm should create the previously explained frames. The new settings.yaml file is exemplified in the Listing 6.19.

```
tag_family:      'tag36h11' # options: tagStandard52h13, tagStandard41h12,
    tag36h11, tag25h9, tag16h5, tagCustom48h12, tagCircle21h7, tagCircle49h12
tag_border:      1
tag_threads:     4          # default: 2
tag_decimate:    0.0        # default: 1.0
tag_blur:        0.0        # default: 0.0
tag_refine_edges: 1         # default: 1
tag_debug:       0          # default: 0
tag_refine_pose: 1
tag_refine_decode: 1

# Other parameters
publish_tf:      true
drone_name: 'drone0'
```

Listing 6.19: Example of the settings.yaml file.

6.5 Graph-Slam Algorithm

The Graph-Slam algorithm is the key component of the localization system since it will be in this block that the localization is estimated. The algorithm relies on the GTSAM framework to create the factor graph and in the iSAM2 algorithm to solve the optimization and smoothing problem. This block receives as motion inputs the raw poses of the drone in the **origin** frame, that is, the poses used to create the **drone0** coordinate frame and as measurement inputs, it receives the AprilTags poses, also in the **origin** frame.

The GTSAM framework also needs a noise model associated with each one of the inputs, in this case, a noise model associated with the raw poses of the vehicle and a noise model associated with the AprilTags pose estimation. This noise models represent the motion noise covariance matrix $\mathbf{\Sigma_i}$ and the measurement noise covariance matrix $\mathbf{\Lambda_n}$ presented in Section 4.1.

Both matrices are 6 by 6 diagonal matrices where the diagonal line first three terms represent the noises in the orientation and the last three terms the noises in translation. From this, the $\mathbf{e_v}$ and $\mathbf{e_l}$ matrices were created, the first one takes into account the raw pose noise and the latter the markers pose estimation noise.

$$\mathbf{e_v} = \begin{bmatrix} e_{\phi_V} & 0 & 0 & 0 & 0 & 0 \\ 0 & e_{\theta_V} & 0 & 0 & 0 & 0 \\ 0 & 0 & e_{\psi_V} & 0 & 0 & 0 \\ 0 & 0 & 0 & e_{x_V} & 0 & 0 \\ 0 & 0 & 0 & 0 & e_{y_V} & 0 \\ 0 & 0 & 0 & 0 & 0 & e_{z_V} \end{bmatrix}, \mathbf{e_L} = \begin{bmatrix} e_{\phi_L} & 0 & 0 & 0 & 0 & 0 \\ 0 & e_{\theta_L} & 0 & 0 & 0 & 0 \\ 0 & 0 & e_{\psi_L} & 0 & 0 & 0 \\ 0 & 0 & 0 & e_{x_L} & 0 & 0 \\ 0 & 0 & 0 & 0 & e_{y_L} & 0 \\ 0 & 0 & 0 & 0 & 0 & e_{z_L} \end{bmatrix} \quad (6.1)$$

In the pose noise matrix the $(e_{\phi_V}, e_{\theta_V}, e_{\psi_V}, e_{x_V}, e_{y_V}, e_{z_V})$ parameters represent the orientation noise in roll, pitch and yaw in radians and the translation noise in *x, y, z* in meters. In the observation noise matrix the $(e_{\phi_L}, e_{\theta_L}, e_{\psi_L}, e_{x_L}, e_{y_L}, e_{z_L})$ also represent the orientation noise in roll, pitch and yaw in radians and the translation noise in *x, y, z* in meters.

Considering the fact that the DJI does not provide any type of information about the velocity and attitude coming from the vehicle, the noise was obtained from the resolution of this data. For the x, y, and z velocities the resolution is 0.1 m/s , since this value is being integrated every 20 Hz, that is every, 0.05 seconds the noise for the translation components is $(e_{x_V}, e_{y_V}, e_{z_V}) = (0.1 \times 0.05, 0.1 \times 0.05, 0.1 \times 0.05) = (0.005, 0.005, 0.005)$. For the orientation, the roll, pitch and yaw components resolutions are 0.1 degrees, which is approximately 0.00175 radians, so the noise for the orientation components is given by $(e_{\phi_V}, e_{\theta_V}, e_{\psi_V}) = (0.00175, 0.00175, 0.00175)$.

For the AprilTags pose estimation, it was made manual tests in order to estimate the accuracy of the detected poses, for the translation errors the following values were obtained $(e_{x_L}, e_{y_L}, e_{z_L}) = (0.15, 0.15, 0.10)$, for the orientations the values were $(e_{\phi_L}, e_{\theta_L}, e_{\psi_L}) = (0.122, 0.122, 0.052)$. The creation of both noise matrices is exemplified in the Listing 6.20.

```
1 noiseModel::Diagonal::shared_ptr motion_noise = noiseModel::Diagonal::Sigmas((
      Vector(6) << 0.00175,0.00175,0.00175,0.005,0.005,0.005).finished());
2
3 noiseModel::Diagonal::shared_ptr measurement_noise = noiseModel::Diagonal::
      Sigmas((Vector(6) << 0.122,0.122,0.052,0.15,0.15,0.10).finished());
```

Listing 6.20: Definition of the motion and measurement noises.

As explained in the Section 4.1 the first pose of the drone is used as a prior, initializing the origin of the localization as being this pose, note that sometimes this is not true, since the origin of the localization reference frame is in reality the marker from where the vehicle takes off, nevertheless this can be done since the optimization is able to solve this error and calculate the real initial position of the drone. After this first pose, every motion input is inserted in the factor graph as a relative pose between the previous pose and the most recent obtained pose. Finally, the raw pose of the vehicle is inserted as an initial estimate to be later used by the iSAM2 algorithm. This process is shown in the Listing 6.21.

```
1 void mavic_slam_gtsam::add_motion_to_graph()
2 {
3     Pose3 Pose_Odometry=Pose3(Rot3(pose_received.pose.orientation.w,
4                                    pose_received.pose.orientation.x,
5                                    pose_received.pose.orientation.y,
6                                    pose_received.pose.orientation.z),
7                               Point3(pose_received.pose.position.x,
```

```
                                pose_received.pose.position.y,
                                pose_received.pose.position.z));
    if(motion_factors_count==0)
    {
        factor_graph.emplace_shared<PriorFactor<Pose3>>(symbol('x', 0),
                                                Pose_Odometry,motion_noise);
    }
    else
    {
        factor_graph.emplace_shared<BetweenFactor<Pose3>>(symbol('x',
            motion_factors_count-1),
                symbol('x', motion_factors_count),
                Prev_Pose.between(Pose_Odometry), motion_noise);
    }
    initialEstimates.insert(symbol('x', motion_factors_count), Pose_Odometry);
    Prev_Pose=Pose_Odometry;
    motion_factors_count++;
}
```

Listing 6.21: Addition of the motion inputs to the factor graph.

Regarding the measurement inputs, the first step is to associate the received measurement with a previously seen landmark, this is done using the id of the AprilTag. If the tag was never seen during the localization, it is added for the first time to the factor graph with the relative pose between the marker and the most recent pose of the drone, it is also added as an initial estimate using the absolute pose relative to the **origin** reference frame which is the same as the localization reference frame. If the markers are already initialized only the relative pose between the marker and the latest vehicle pose is added to the factor graph as seen in Listing 6.22.

```
void mavic_slam_gtsam::add_landmark_to_graph()
{
    for (int i = 0; i < tag_array.poses.size(); i++) {
        geometry_msgs::TransformStamped transformStamped;
        id[i] = tag_array.poses.at(i).header.seq;

        transformStamped = tf_buffer.lookupTransform("origin","tag_" + to_string
            (id[i]),ros::Time(0));

        Pose3 Pose_Landmark = Pose3(Rot3(transformStamped.transform.rotation.w,
```

```
            transformStamped.transform.rotation.x,transformStamped.transform.
            rotation.y,transformStamped.transform.rotation.z),
        Point3(transformStamped.transform.translation.x,transformStamped.
            transform.translation.y,transformStamped.transform.translation.z));

        if (array_landmarks_seen[id[i]] == 0) {
            initialEstimates.insert(symbol('l', id[i]), Pose_Landmark);
            array_landmarks_seen[id[i]] = 1;
        }
        factor_graph.emplace_shared<BetweenFactor<Pose3>>(symbol('x',
            odometry_factors_count - 1),symbol('l', id[i]), Prev_Pose.between(
            Pose_Landmark),Landmark_observation_noise);
    }
}
```

Listing 6.22: Addition of the measurement inputs to the factor graph.

Assuming that the factor graph is already correctly formed, the last step is to optimize the graph using the iSAM2 algorithm. This optimization is done every time a new measurement input is inserted in the graph or every 40 motion inputs. To optimize the graph it is only needed to feed the iSAM2 algorithm with the constructed graph and the initial estimates. From this point, it is possible to calculate the optimized vehicle poses and trajectory and the landmark poses. It is also possible to calculate the covariance matrices associated with the previous poses. The optimization is exemplified in the Listing 6.23.

```
void mavic_slam_gtsam::optimize_factor_graph()
{
    isam.update(factor_graph,initialEstimates);
    optimized_estimates.insert(symbol('x',odometry_factors_count-1),isam.
        calculateEstimate(symbol('x', odometry_factors_count-1)));

    Pose3 optimized_pose;
    optimized_pose=optimized_estimates.at<gtsam::Pose3>(symbol('x',
        odometry_factors_count-1));

    Matrix3 Pose_Rotation = optimized_pose.rotation().matrix();
    Quaternion Optimized_Pose_Orientation(Pose_Rotation);
    Covariance_Pose=isam.marginalCovariance(Symbol('x', odometry_factors_count
        -1)).data();
```

```
12      initialEstimates.clear();
13      factor_graph.erase(factor_graph.begin(),factor_graph.end());
14  }
```

Listing 6.23: Optimization of the factor graph.

Note that every time the factor graph is added to the iSAM2 algorithm, the previous factor graph, and initial estimates must be cleared, this is necessary because the iSAM2 algorithm already stores the old data in its memory.

The last step is to publish the optimized pose for it to be used by the navigation algorithms. This pose is published in a topic named /drone0/self_localization/pose of the type geometry_msgs/PoseStamped.

6.6 UcoSlam

As explained in Section 5.4, the UcoSlam algorithm will be used as a comparison between the developed localization algorithm and a state-of-the-art algorithm. Since this algorithm does not have a direct implementation in the ROS framework, it was required to create a wrapper to use in the project. This wrapper subscribes to the distorted uncompressed image frames coming in the /drone0/sensor_measurements/camera /image_raw topic sending them to the UcoSlam algorithm. After receiving the images the UcoSlam calculates the real-time pose of the drone and the *x, y, z* position of the detected ArUco markers. The vehicle pose is then published in a /drone0/uco_slam/camera_pose topic of the type geometry_msgs/PoseStamped, and the markers pose in a /drone0/uco_slam/camera_markers of the type nav_msgs/Path. This process is shown in the Listing 6.24.

```
1   void uco_slam::image_callback(const sensor_msgs::ImageConstPtr &msg)
2   {
3       image_frame=*msg;
4       flag_new_image=true;
5   }
6
7   void uco_slam::do_uco_slam()
8   {
9       cv::Mat image_frame_mat;
10      cv_bridge::CvImagePtr cv_ptr;
11      try
12      {
```

```
13          cv_ptr = cv_bridge::toCvCopy(image_frame, sensor_msgs::image_encodings::
                TYPE_8UC3);
14          image_frame_mat=cv_ptr->image;
15      }
16      catch (cv_bridge::Exception& e)
17      {
18          ROS_ERROR("cv_bridge exception: %s", e.what());
19          return;
20      }
21      cv::Mat posef2g= SLAM.process(image_frame_mat,cameraParams,frame_counter);
22      ...
23  }
```

Listing 6.24: Obtainment of the image and calculation of the drone pose.

Note that, the UcoSlam algorithm calculates the vehicle poses and markers location taking into account an image coordinate system as shown in Figure 6.7. For this reason, it was required to convert the UcoSlam results in the desired coordinate system. The developed coordinate system was the same as the Graph-SLAM algorithm with its origin being the first pose of the drone, this reference frame was named **uco_origin**. The first step was to convert the results from the **uco_image_frame** to the **uco_origin**, this is done using a rotation of 90 degrees in pitch following a rotation of 90 degrees in yaw. After that, since the results were already in the desired reference frame it was only needed to convert the camera frame to the vehicle frame.

This conversion was made in two stages, first between **uco_camera_frame** as seen in Figure 6.7 and the desired **uco_drone0_frame** as seen in Figure 6.5. This was done by doing a rotation of -90 degrees in roll following a rotation of -90 degrees in yaw. The second step was to apply the extrinsic rotation of the gimbal to convert the camera frame to the real vehicle frame. This process is similar to what was done in the Graph-Slam algorithm using the DJI gimbal attitude. Once more, all the coordinate systems transformations were done using the tf library from the ROS framework. After being transformed, the drone pose was published in a topic named /drone0/uco_slam/aircraft_pose of the type geometry_msgs/PoseStamped and the markers in a topic named /drone0/uco_slam/markers of the type nav_msgs/Path.

Esta página foi intencionalmente deixada em branco.

Chapter 7

Results

This chapter will present all the results that were obtained by the developed algorithms and applications validating the utility of the android application and the accuracy and usability of the Graph-Slam algorithm. A comparison between the Graph-Slam algorithm and the state of art UcoSlam algorithm will also be made.

7.1 Mobile SDK and ROS Application

This section will validate the developed android application, showing the layout and the ROS topics publishing the vehicle sensor data. Note that the application was mainly used in a Xiaomi Redmi 9 with a Mediatek Helio G80 chipset with an octa-core (2x2.0 GHz Cortex-A75 and 6x1.8 GHz Cortex-A55). The smartphone also has a Mali-G52 MC2 Graphical Processing Unit (GPU) and 4 GB of Random Access Memory (RAM). As explained previously, the smartphone is connected to the ground station using a 5.0 GHz Wi-Fi hotspot to be able to have the lowest latency possible. Since the 5.0 GHz has a low range and propagates very poorly between obstacles the smartphone was kept close to the ground station at all times.

Has it is possible to see in Figure 7.1, in a first instance the application starts by doing the previously explained registration process and validation of the connection between the smartphone and the remote controller, during this process the OPEN button remains greyed out and inaccessible. Once all the validations are completed the OPEN button becomes blue and it is possible to click on it to proceed to the next window.

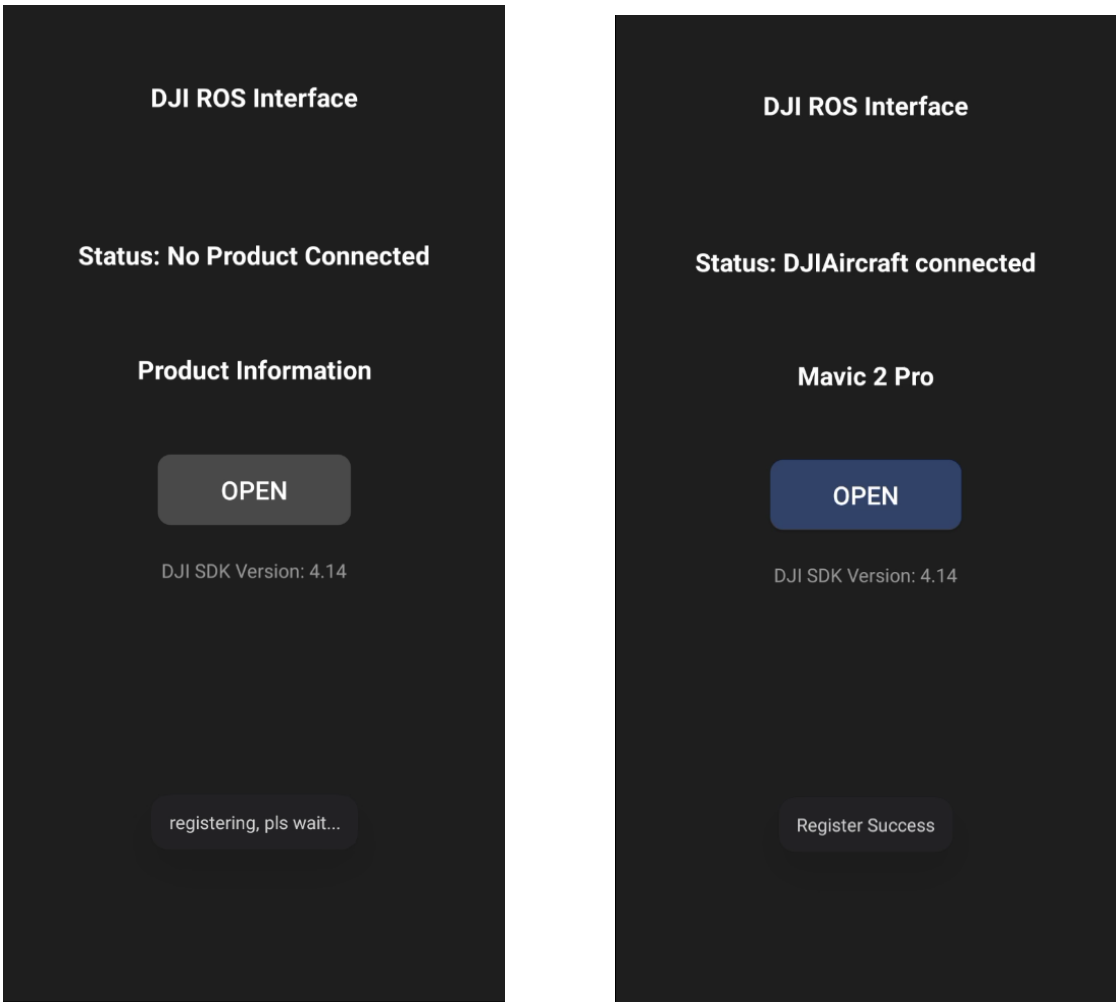

Figure 7.1: Application entry layout.

The Figure 7.2 represents the second layout of the application. On this screen, the user can write the IP address of the machine running the roscore, that is, the ROS Master. It is also possible to obtain this IP address by reading a QRCode. Once the user writes the IP address in the text box and clicks on the connect button, the application proceeds to establish the connection to the Master. If the connection is done successfully the application proceeds to the last layout and starts the publishing and subscribing of the ROS topics. If the connection fails the application shows an error message and asks the user to enter a valid IP address.

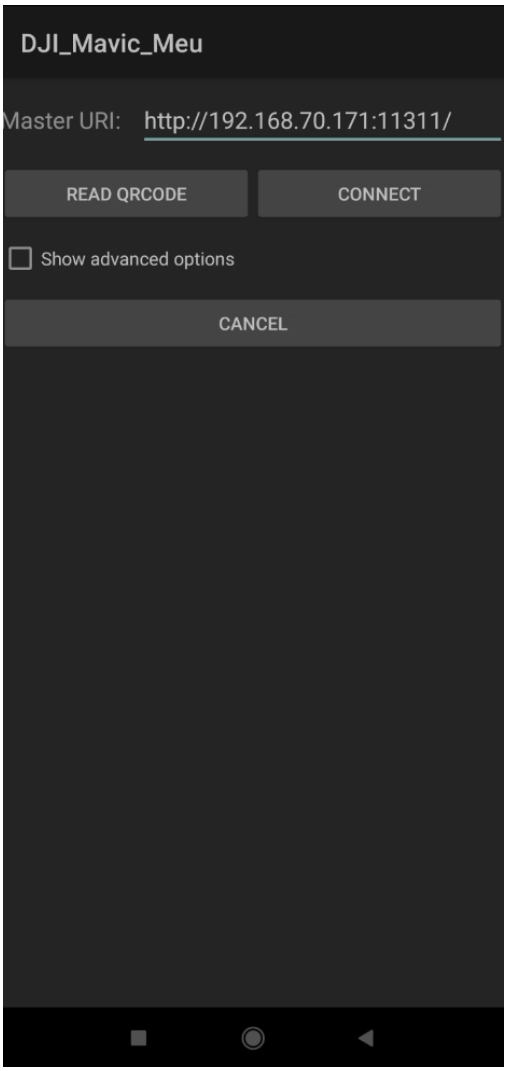

Figure 7.2: ROS connection layout

The last and main layout of the application is presented in Figure 7.3. In this layout is possible to check the Flying First Person View (FPV) of the drone while also being able to take photos or record videos. Alongside these functionalities, the application is also sending the sensor data to the ground station using the ROS framework. The Rosgraph is represented in the Figure 7.4 and the topics received in the ground station are shown in the Figures 7.5, 7.6 and 7.7.

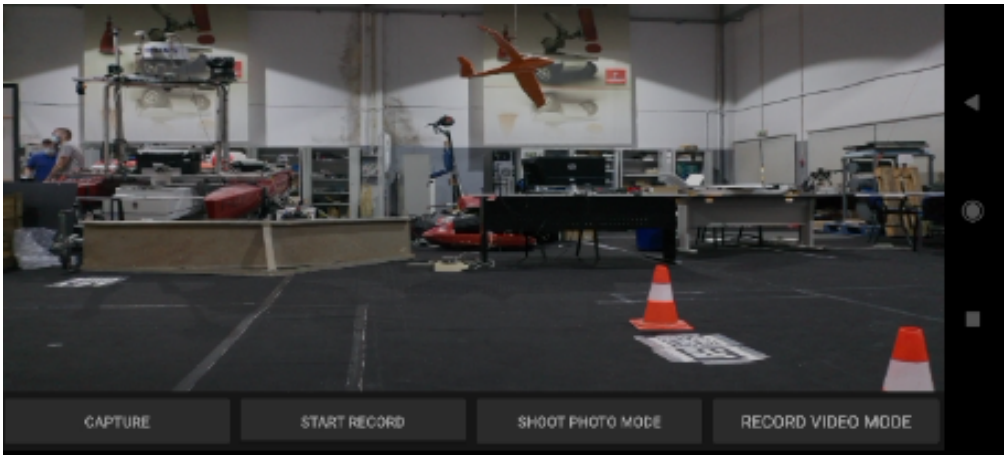

Figure 7.3: Application main layout.

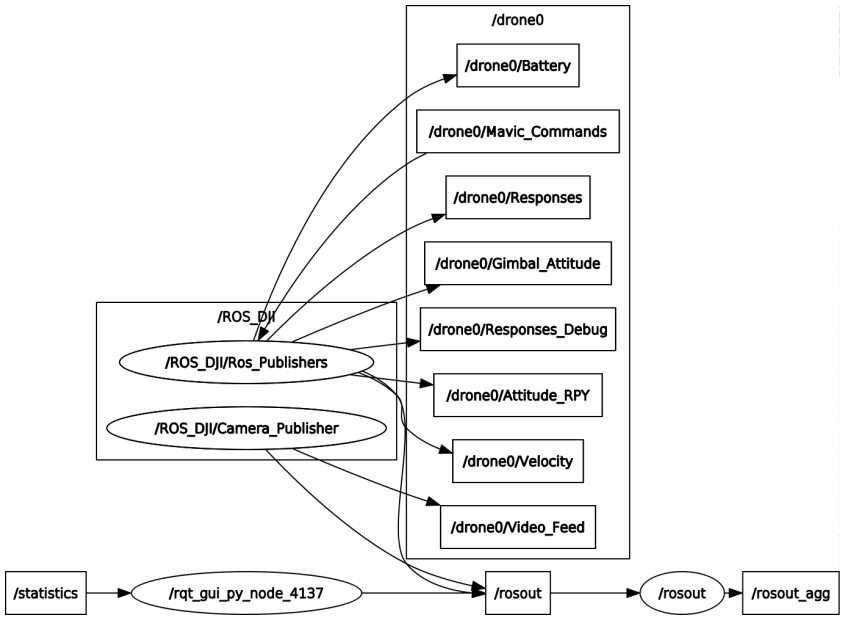

Figure 7.4: Rosgraph of the android application.

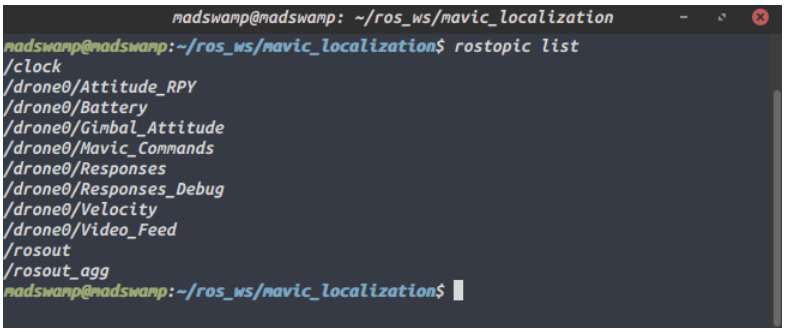

Figure 7.5: ROS topics shared between the android and the ground station.

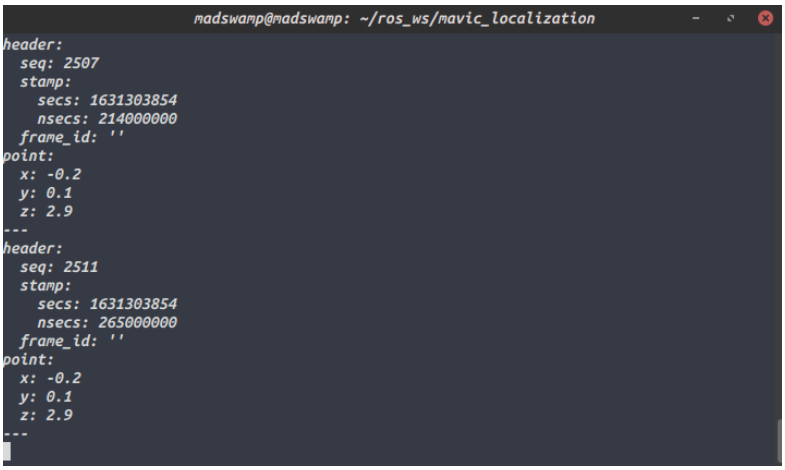

Figure 7.6: Vehicle attitude topic.

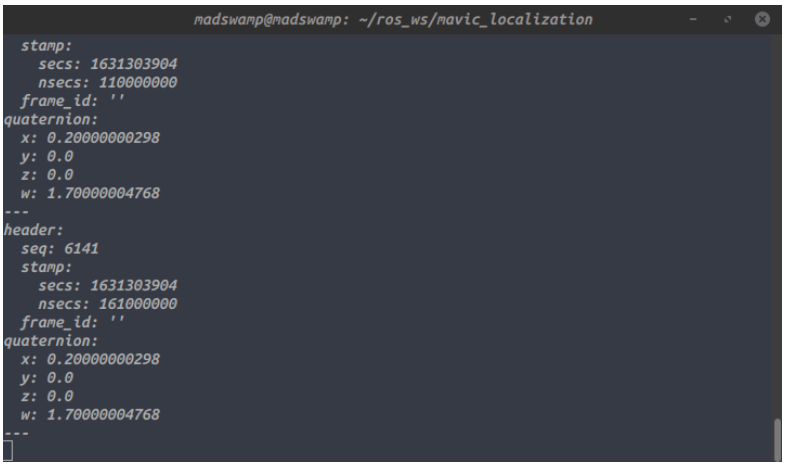

Figure 7.7: Vehicle velocity and altitude topic.

As seen in the previous figures the ROS topics shared between the android application and the ground station are all working as intended validating the development of the application and the connection between the two.

The topics are all being published with the desired frequency, that is 20 Hz for the Attitude, Gimbal attitude, and Velocity topics, 16 Hz for the Video Feed topic, and 1 Hz for the Battery topic. Regarding the Video Feed topic, the images are arriving at the ground station with a delay of more or less 200 ms which is enough for the project objectives.

7.2 Decompression and Calibration

As explained previously the image frames reach the ground station compressed, for them to be usable in the algorithms they must be previously decompressed. The Figure 7.8 validates the decompression of the image by showing a decompressed image frame in the rqt_image_view tool.

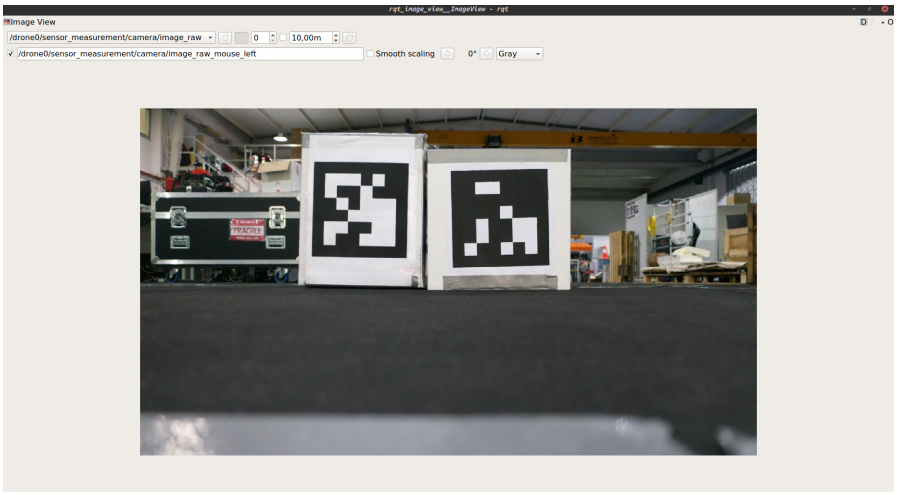

Figure 7.8: Image frame obtained after the decompression.

Regarding the calibration of the camera, the obtained intrinsic matrix is the following.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1068.0297 & 0 & 640.5154 \\ 0 & 1072.7975 & 342.7160 \\ 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

The obtained distortion parameters were the following.

$$D = \begin{bmatrix} k_1 & k_2 & t_1 & t_2 & k_3 \end{bmatrix} = \begin{bmatrix} 0.1521 & -0.4710 & -0.0091 & -0.0020 & 0.7405 \end{bmatrix} \quad (7.2)$$

7.3 Localization Algorithms

Before starting to present the results referent to the localization algorithms it is important to refer that all the tests were performed with a laptop with an Intel Core i5-6300HQ Processor with 4 cores and 4 threads with a base frequency of 2.3 GHz and a maximum turbo frequency of 3.2 GHz, the laptop also has 32 GB of RAM. At the end of this section a benchmark between the Graph-Slam and the UcoSlam will also be made.

7.3.1 Box of Tags Tests

The first set of tests was made to validate the detection of the markers, the frame transformations, and the functionality of the Graph-Slam algorithm and the UcoSlam. Note that, this first set of tests was not intended to compare the accuracy of the algorithms.

For the Graph-Slam algorithm, the test was conducted using a circular motion around a tower of AprilTags while also changing the altitude to the ground. For the UcoSlam the test was also conducted using a circular motion around a tower of ArUco markers but without changing the altitude. The tags in the towers had different orientations and positions and the camera was always focused on them. The Graph-Slam used bundles of markers and the UcoSlam used standalone markers. The examples of both scenarios are represented in the Figure 7.9 and 7.10.

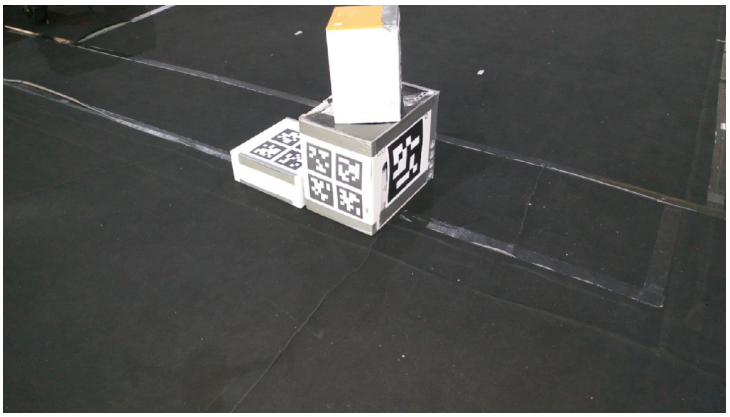

Figure 7.9: Tower of AprilTags scenario.

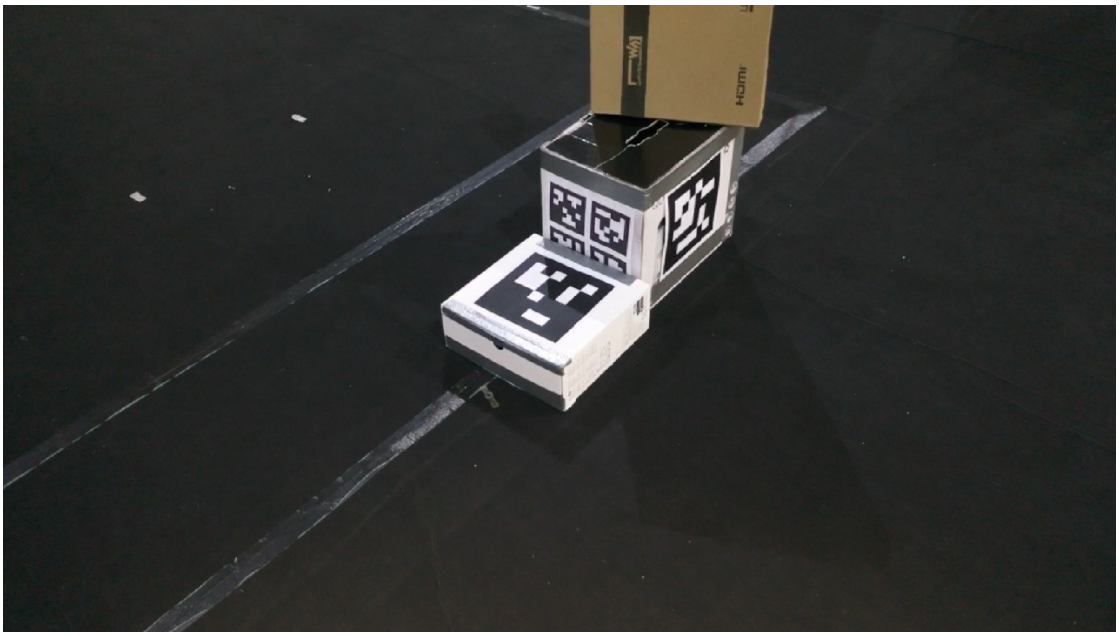

Figure 7.10: Tower of ArUco tags scenario.

7.3.1.1 Graph-Slam

The Figure 7.11 represents the RViz view of the Graph-Slam algorithm running in real-time.

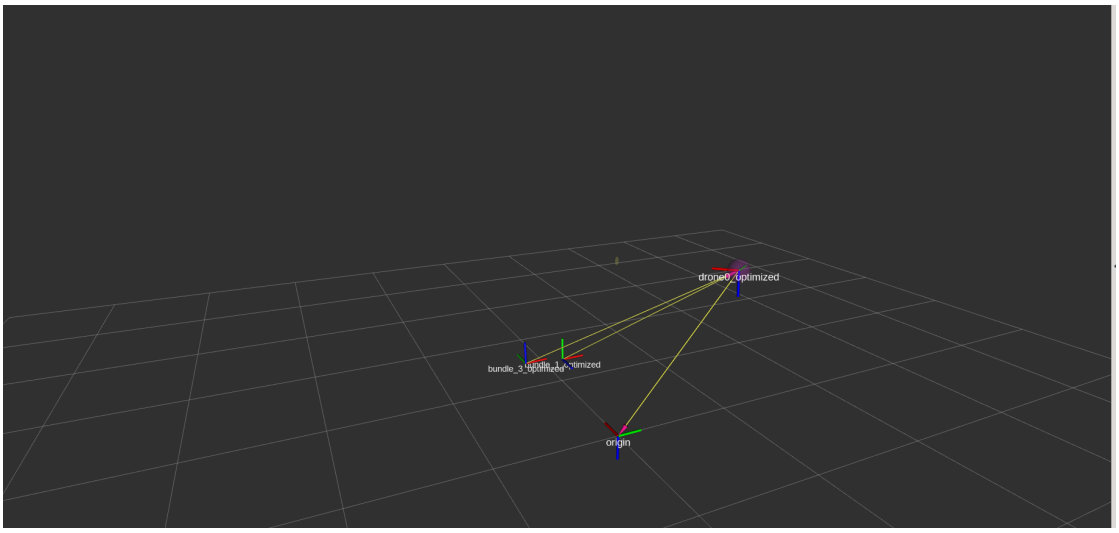

Figure 7.11: RViz view of the drone and tags estimated position from the Graph-Slam Algorithm.

The Figures 7.12, 7.13, 7.14 and 7.15 represent the trajectory done by the drone and the pose of the detected AprilTags.

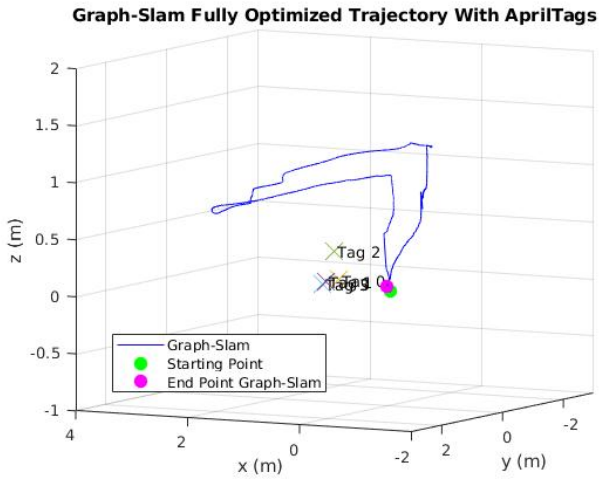

Figure 7.12: Graph-Slam trajectory with markers during the initial tests.

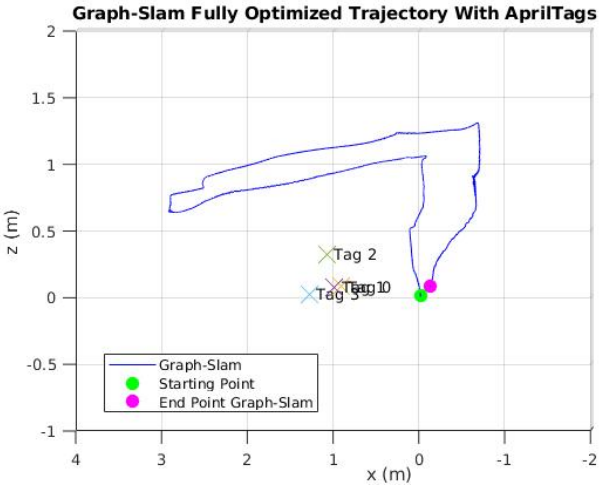

Figure 7.13: Graph-Slam trajectory with markers during the initial tests (XZ).

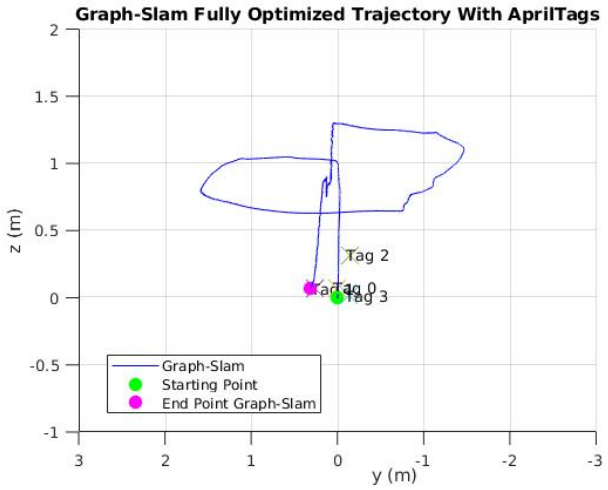

Figure 7.14: Graph-Slam trajectory with markers during the initial tests (YZ).

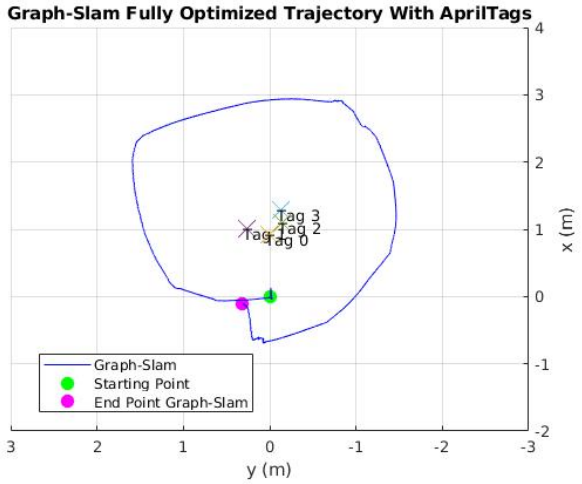

Figure 7.15: Graph-Slam trajectory with markers during the initial tests (YX).

By analyzing the previous figures it is possible to conclude that the algorithm is working properly and as intended since the estimated trajectory is close to the one that was made. The marker's position is also close to the real ones.

Unfortunately, during a later test where the drone was further away from the markers, it was possible to observe that the reduced size of the markers in the bundles made the accuracy of the detection worse, especially in the estimation of the orientation. For this reason, the definitive test was made using standalone markers with bigger sizes.

7.3.1.2 UcoSlam

The Figures 7.16 and 7.17 represent the RViz view of the UcoSlam algorithm also running in real-time, and the GUI that is available when using the UcoSlam framework.

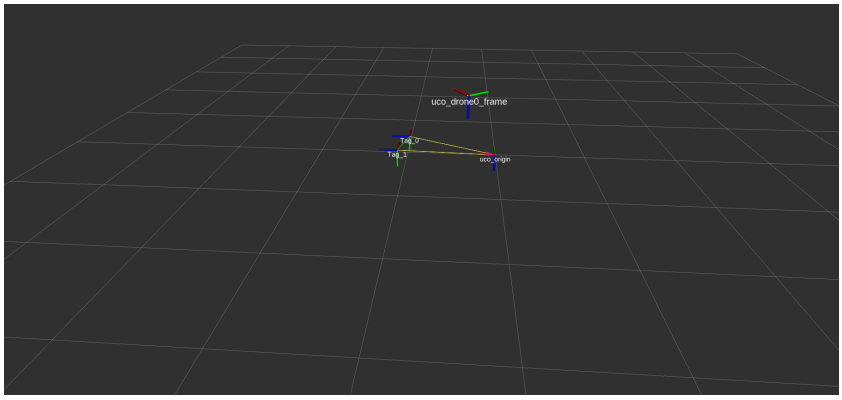

Figure 7.16: RViz view of the drone and tags estimated position from the UcoSlam Algorithm.

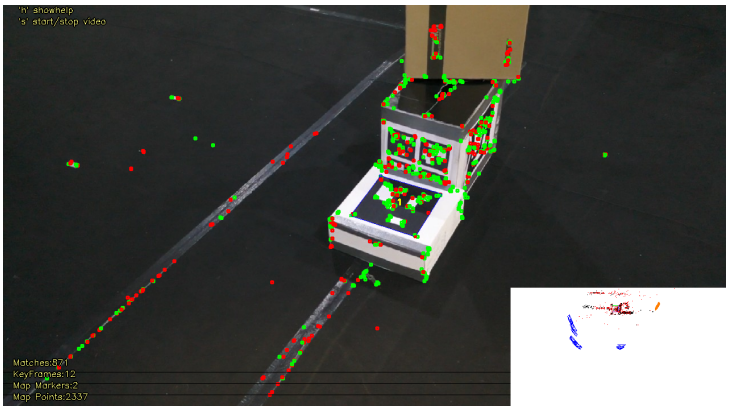

Figure 7.17: GUI available in the UcoSlam framework.

The trajectory made by the drone and the localization of the markers is shown in the Figures 7.18, 7.19, 7.20 and 7.21.

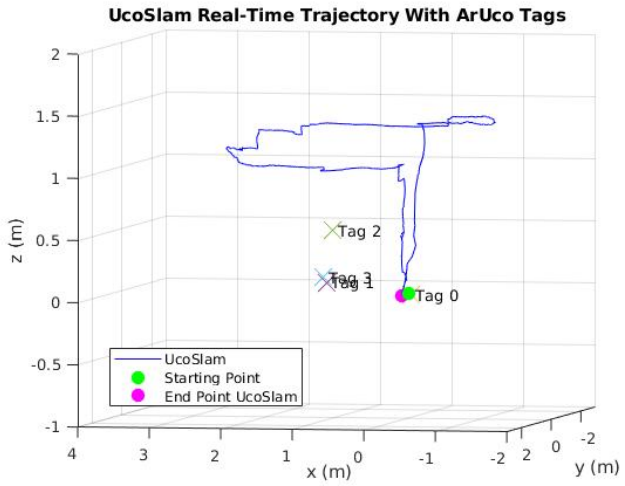

Figure 7.18: Uco-Slam trajectory with markers during the initial tests.

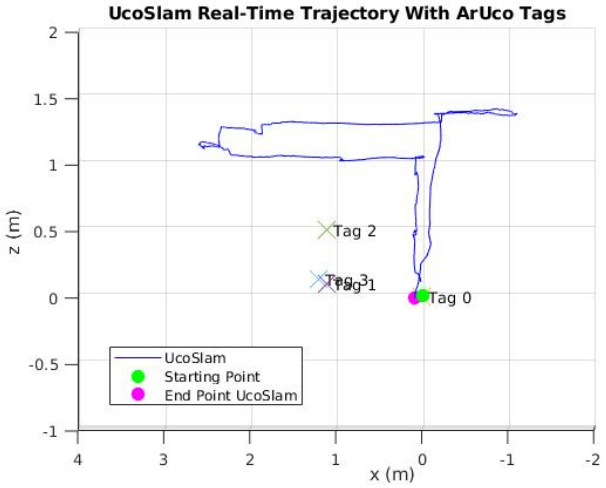

Figure 7.19: Uco-Slam trajectory with markers during the initial tests (XZ).

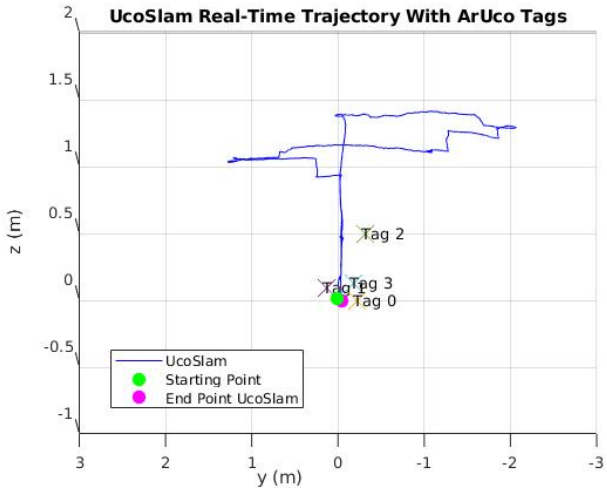

Figure 7.20: Uco-Slam trajectory with markers during the initial tests (YZ).

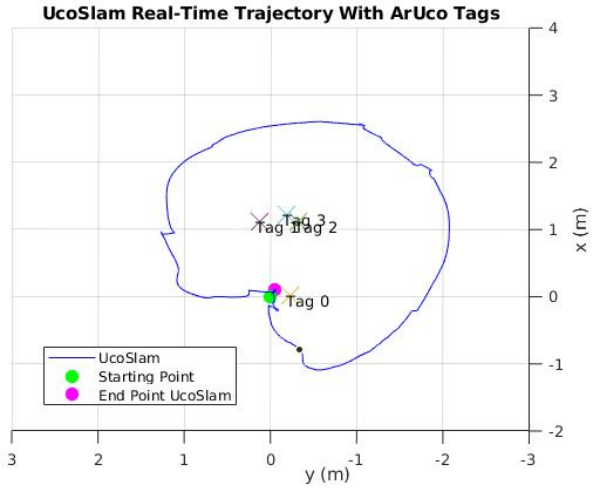

Figure 7.21: Uco-Slam trajectory with markers during the initial tests (YX).

By observation of the previous figures, it is possible to validate the usability of the implemented modifications done to the UcoSlam. The obtained trajectory and markers pose is pretty much the same as the one done with the vehicle.

7.3.2 Evaluation of the Localization During a Warehouse Mission

The last and definitive set of tests were conducted in a warehouse look-alike scenario that was instrumented inside the laboratory. The instrumentation of the laboratory consisted of the positioning of AprilTags and ArUco tags in random positions and orientations on the ground and walls of the laboratory. For the Graph-Slam, 9 tags with 19 cm of size were printed in A4 paper sheets, for the UcoSlam, 10 tags with 16.7 cm of size were also printed in A4 paper sheets. Both algorithms had a marker placed in the ground at the take-off/landing position, the objective of these markers is to localize the drone during the take-off and the landing procedures and to fix the origin of the localization referential in it. In the case of the UcoSlam, an extra marker was needed in front of the drone when it is landed in the initial position, this marker helped in the estimation of the real-world scale. In both cases only the position of the marker located in the take-off/landing position is known since this marker is considered the origin of the localization frame. All the IDs of all the markers are known since they are used to solve the data association problem.

The objective of these tests was to compare and validate the accuracies and the computational demands of both the Graph-Slam and the UcoSlam algorithms. The Figure 7.22 represents the scenario where the tests were conducted.

Figure 7.22: Instrumented warehouse look-alike scenario.

Unfortunately, it was impossible to use the same flight to test both the algorithms, this is due to two problems that arises when using the UcoSlam. The first problem is the

fact that the UcoSlam has a problem in keeping the vehicle located when it is not flying at speeds below 2 m/s. The second problem is the processing power demands that the UcoSlam requires after more or less five minutes of flight time. The Graph-Slam does not have this kind of problem, making it possible to be used when the drone has higher flight speed and longer flight times.

7.3.2.1 Graph-Slam

To test the Graph-Slam, a manual flight of about 560 seconds was done, the take-off and landing positions were approximately the same. The drone trajectory consisted of a come-and-go movement between the take-off/landing position and the most distant marker placed in the laboratory, the trajectory variations in the Z position were minimal as supposed in a warehouse scenario. The main variations were made in the X and Y positions, the one with the most variation was the X. Four come-and-go loops were made to observe the impact that the loop closing has on the slam results. The Figures 7.23 and 7.24 exemplifies the RVIZ visualization of the Graph-Slam algorithm.

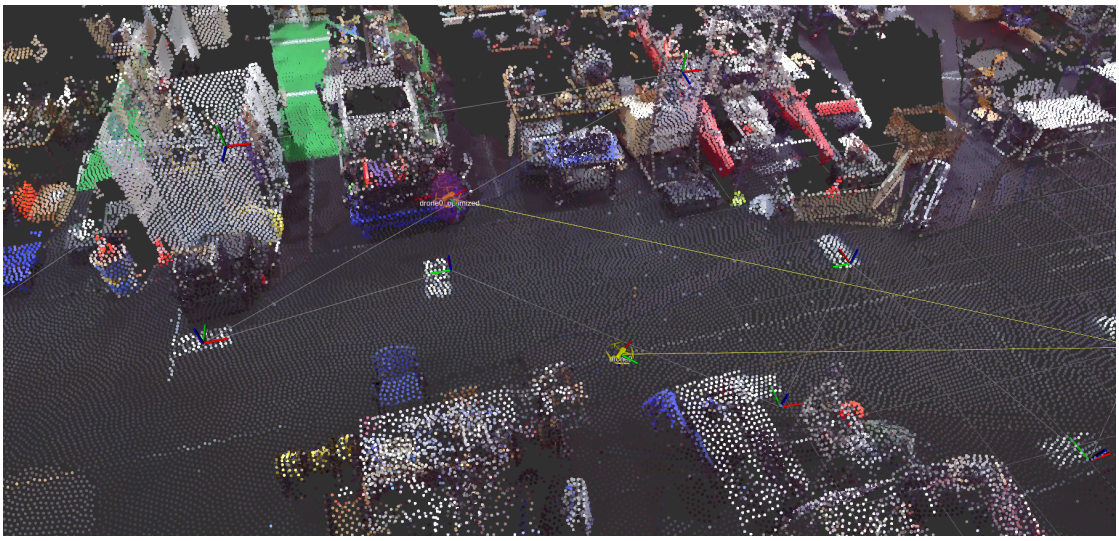

Figure 7.23: Real-Time visualization of the Graph-Slam algorithm.

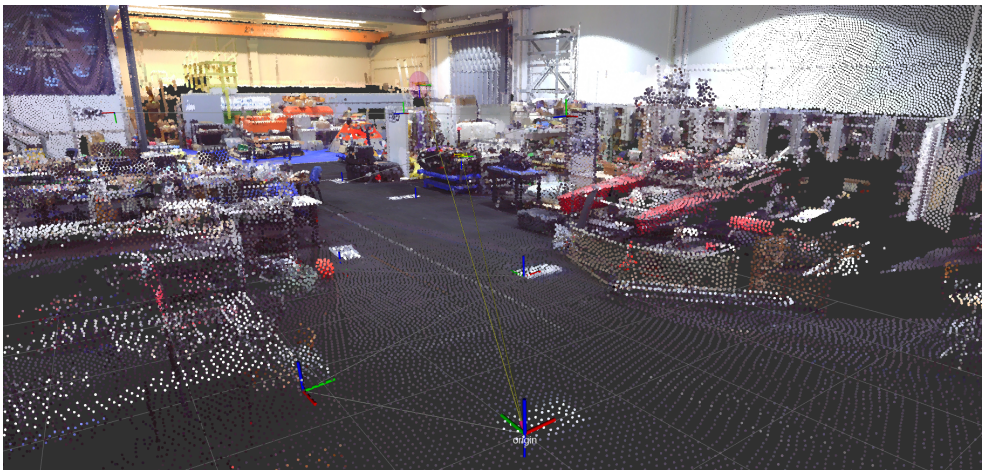

Figure 7.24: Real-Time visualization of the Graph-Slam algorithm from another perspective.

The Figure 7.25 represents the processing time of the Graph-Slam algorithm along the flight time.

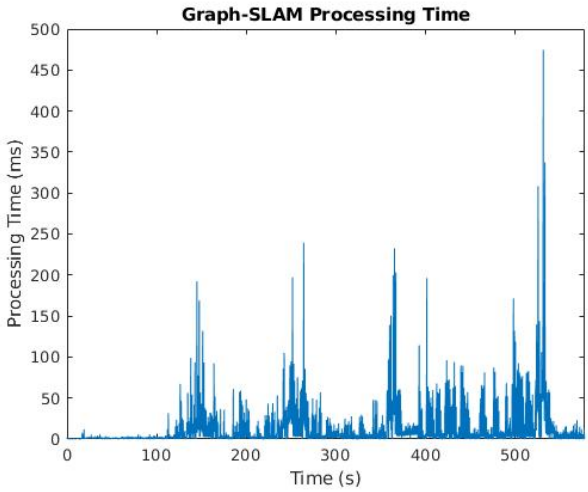

Figure 7.25: Graph-Slam processing time along the flight time.

The average processing time was more or less 6.91 milliseconds. Note that during the first 100 seconds of flight, the drone did not revisit any markers, for this reason, no loop closing happened which make the processing time during this section lower.

The Figures 7.26, 7.27, 7.28 and 7.29 represent the fully optimized trajectory performed by the drone and the detected markers.

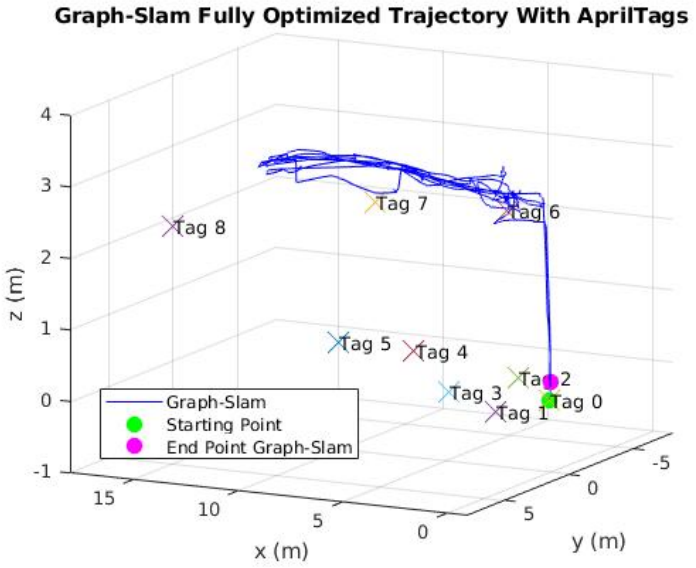

Figure 7.26: Graph-Slam optimized trajectory with markers during the final tests.

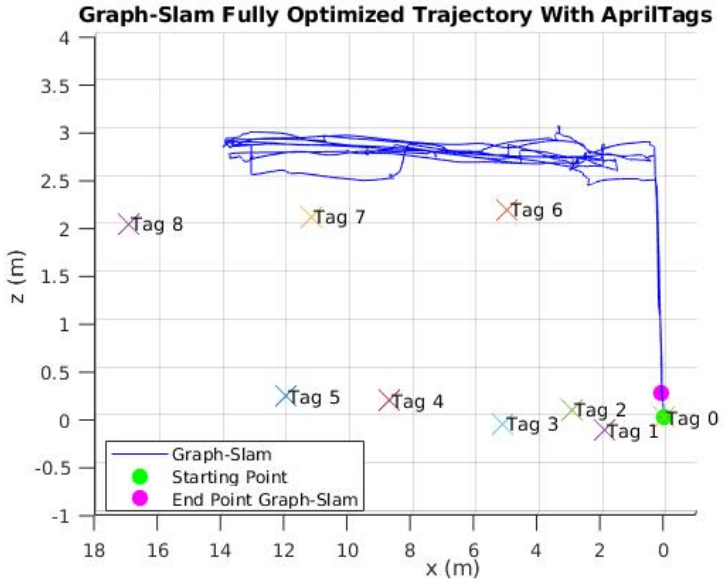

Figure 7.27: Graph-Slam optimized trajectory with markers during the final tests (XZ).

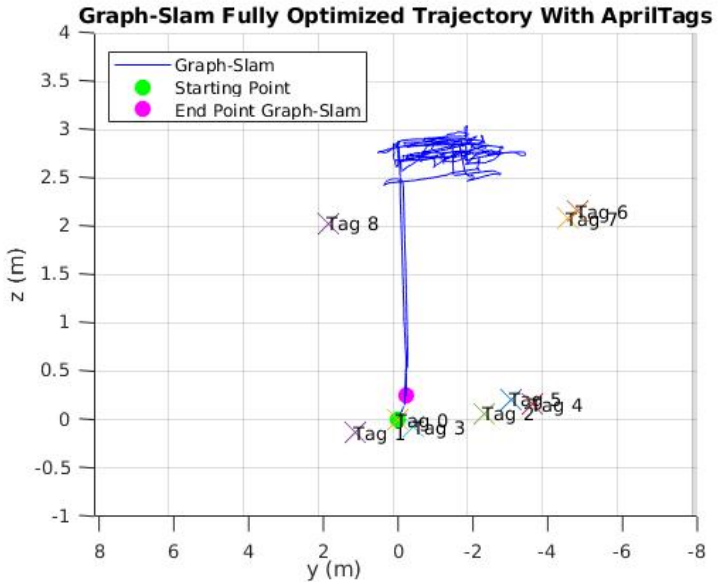

Figure 7.28: Graph-Slam optimized trajectory with markers during the final tests (YZ).

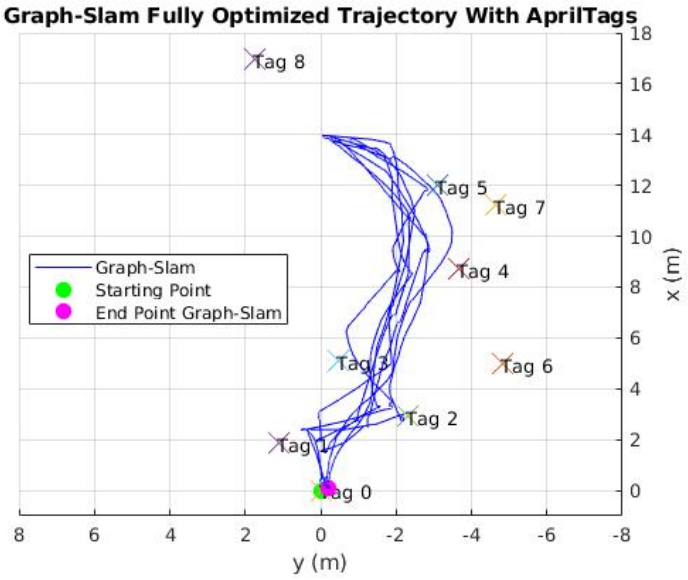

Figure 7.29: Graph-Slam optimized trajectory with markers during the final tests (YX).

By observation of the previous results, it is possible to conclude that the obtained trajectory is close to the one that was manually made during the flight. And the tags are dispersed in the same way as the ones in the real world.

The Figures 7.30, 7.31, 7.32 and 7.33 represent a comparison between the position obtained by only integrating the velocities coming from the DJI SDK, and the Graph-Slam trajectory.

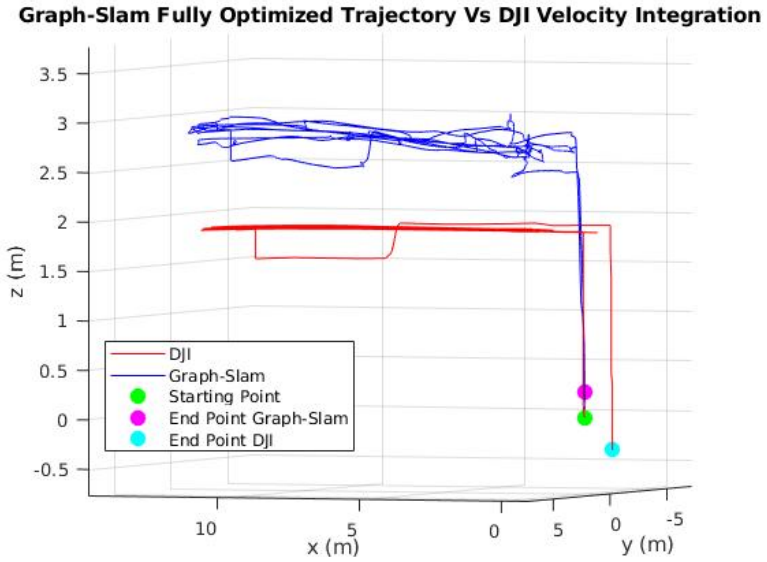

Figure 7.30: Graph-Slam optimized trajectory vs raw data provide by DJI SDK.

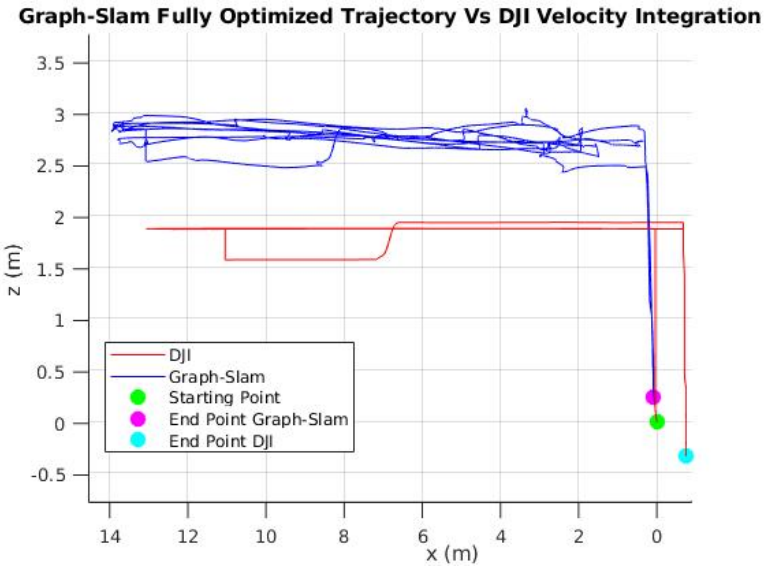

Figure 7.31: Graph-Slam optimized trajectory vs raw data provide by DJI SDK (XZ).

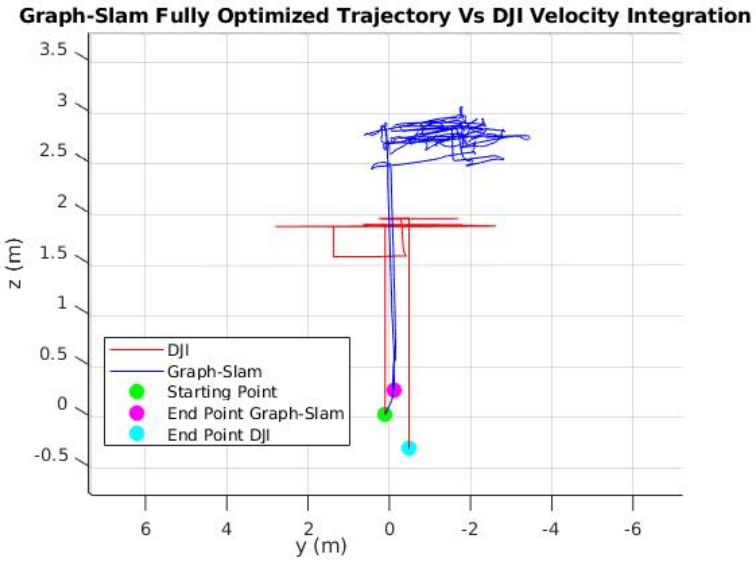

Figure 7.32: Graph-Slam optimized trajectory vs raw data provide by DJI SDK (YZ).

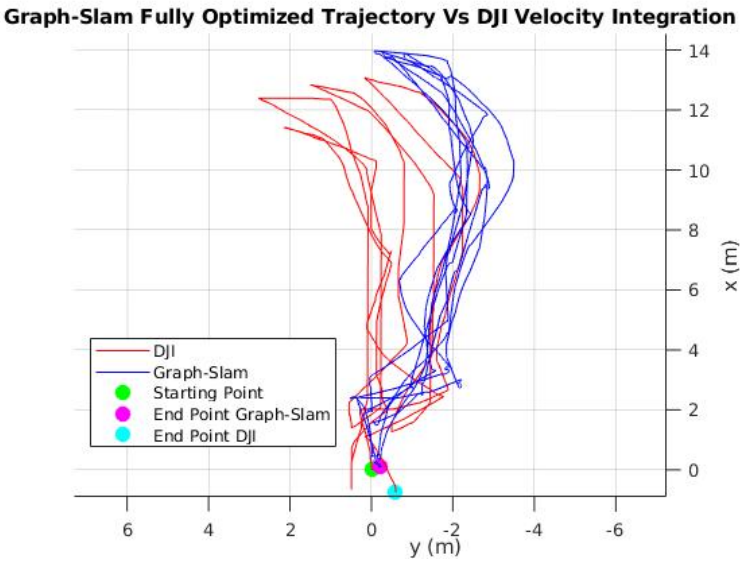

Figure 7.33: Graph-Slam optimized trajectory vs raw data provide by DJI SDK (YX).

Even though the raw localization is somewhat close to the real trajectory it still has significant problems mainly in the Z component since during the flight the vehicle never flew lower than the 2 meters in Z. Regarding the X component, the trajectory cannot be a straight line since during the flight the drone always flew a little to the right.

Alongside the fully optimized trajectory, it is also important to know the real-time

trajectory of the vehicle. This trajectory is important because it shows the positions that the algorithm effectively estimated at a defined timestamp.

The Figures 7.34, 7.35, 7.36 and 7.37 show the real-time trajectory compared with the fully optimized trajectory.

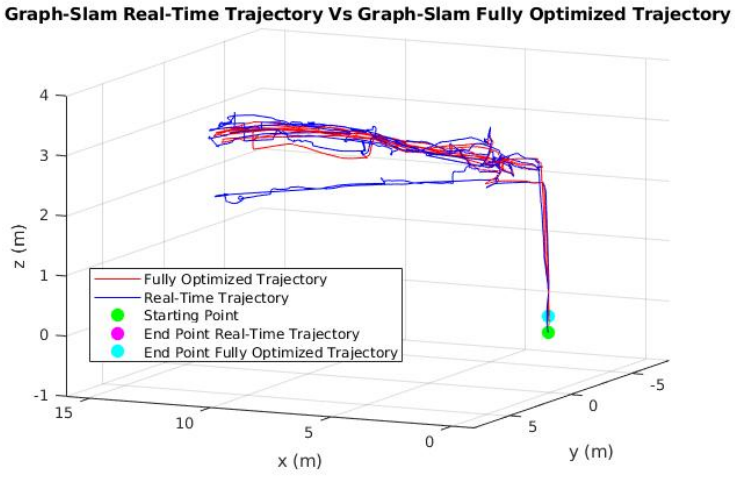

Figure 7.34: Real-time trajectory vs fully optimized trajectory.

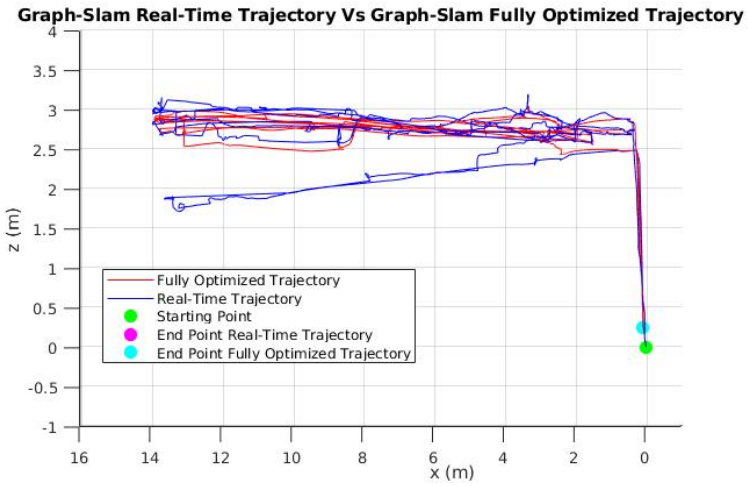

Figure 7.35: Real-time trajectory vs fully optimized trajectory (XZ).

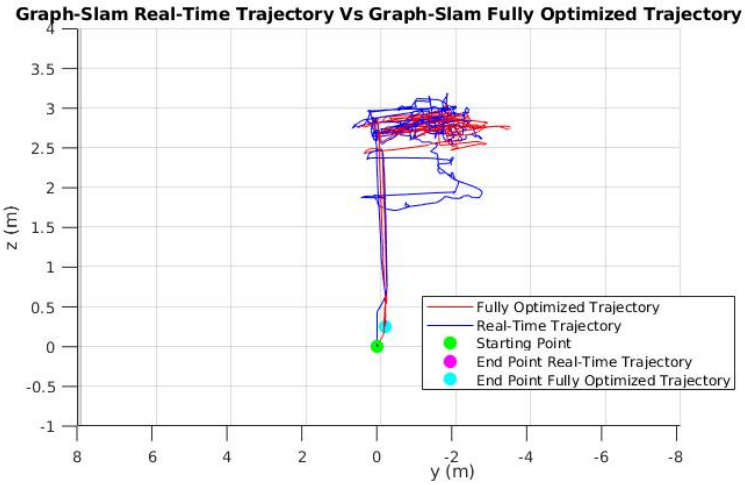

Figure 7.36: Real-time trajectory vs fully optimized trajectory (YZ).

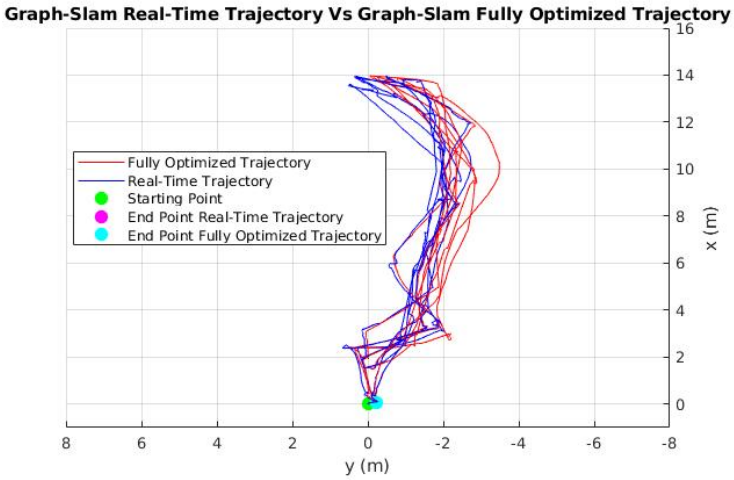

Figure 7.37: Real-time trajectory vs fully optimized trajectory (YX).

During the first loop, the real-time trajectory of the drone was not completely correct, but over time this trajectory tended to the fully optimized one. The only loop that was far from the optimized trajectory was the first one, which is normal since all the AprilTags that the vehicle is observing during this loop are completely new and the pose of the drone has high uncertainty.

7.3.2.2 UcoSlam

To test the UcoSlam a manual flight was also done. This flight took about 806 seconds and it was only possible to do a loop around the instrumented scenario because of the high processing power that was needed to run the algorithm after after about five minutes and the speeds below 2 m/s that was necessary for the algorithm not to lose itself and enter in relocalization mode. Similar to the Graph-Slam test, the drone also took off and landed in the same position. The trajectory was pretty much the same as the previous one and all the 10 ArUco markers were detected.

The Figure 7.38 represents the processing time of the UcoSlam algorithm along the flight time.

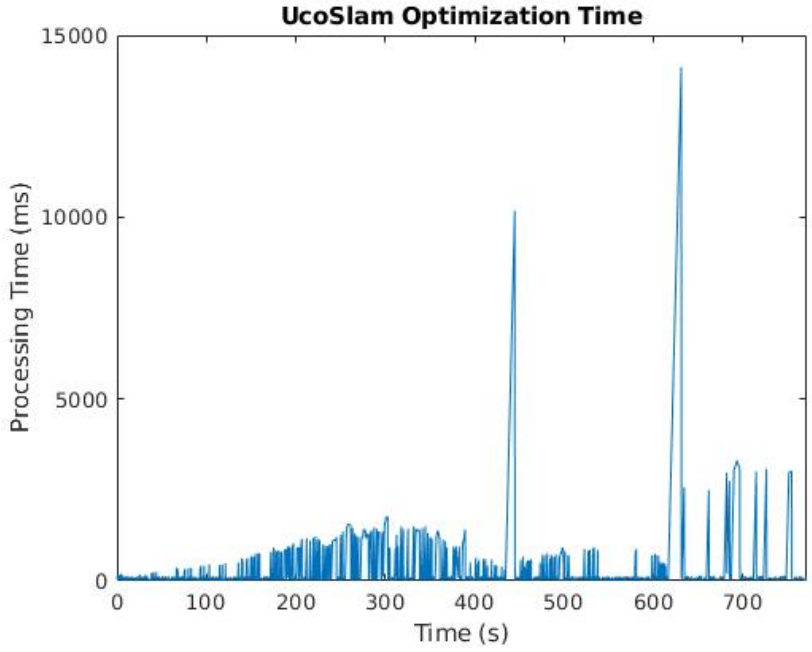

Figure 7.38: UcoSlam processing time along flight time.

The average processing time for the UcoSlam was more or less 371 milliseconds. This processing time is a big problem when talking about the real-time localization of an UAV.

The estimated trajectory of the UcoSlam and the position of the ArUco markers are shown in the Figures 7.39, 7.40, 7.41 and 7.42. Note that when using the UcoSlam it is impossible to access the fully optimized trajectory of the vehicle, for this reason only the real-time trajectory will be shown.

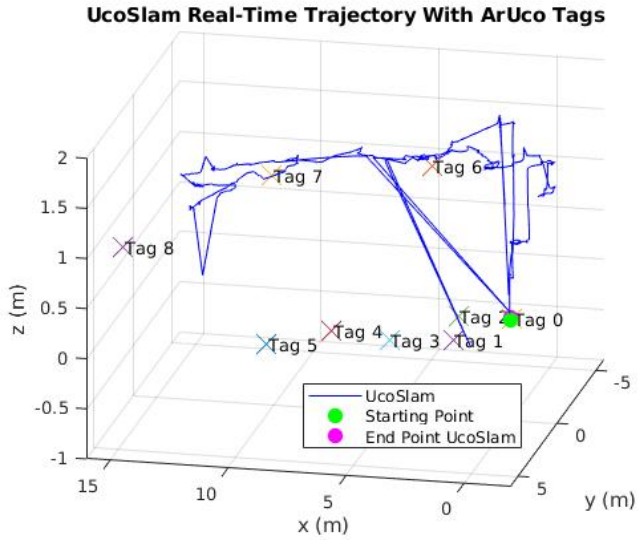

Figure 7.39: UcoSlam real-time trajectory with markers during the final tests.

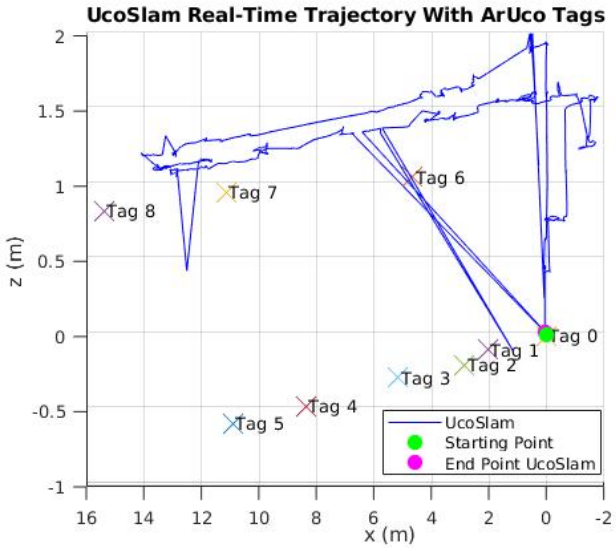

Figure 7.40: UcoSlam real-time trajectory with markers during the final tests (XZ).

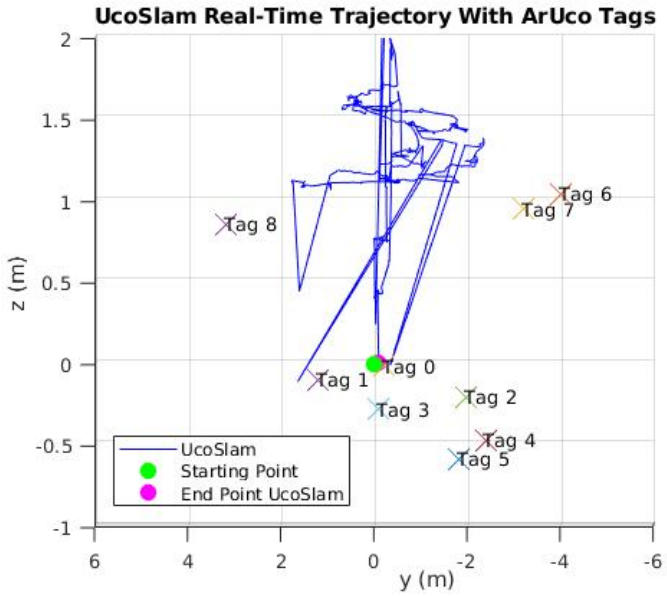

Figure 7.41: UcoSlam real-time trajectory with markers during the final tests (YZ).

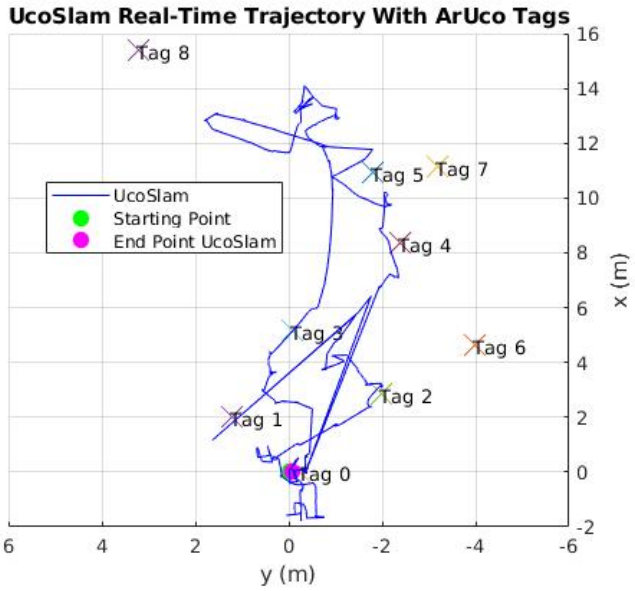

Figure 7.42: UcoSlam real-time trajectory with markers during the final tests (YX).

Not considering some outliers made by the loop closings, the trajectory also looks close to the real one, except for the altitude that should be closer to 2.2 meters and not below the 2 meters.

7.3.2.3 Graph-Slam Vs UcoSlam Benchmark

Regarding the benchmark of the processing time and computational power, the Graph-Slam is the one with the lower average processing time of 6.91 milliseconds while the average time of the UcoSlam was 371 milliseconds. It is also possible to use the Graph-Slam algorithm with higher flight speed since there are no relocalization problems and the pose calculation is much faster.

Regarding the benchmark of the accuracy, since there was no way to get the ground truth trajectory of the drone, the accuracy will be evaluated taking into account the markers pose. To get the true pose of the tags, a scan of the laboratory was made using the LIDAR, FARO Focus3DS 120. The output of this scan allows detecting the pose of the markers with millimeter accuracy. With the ground truth pose of the markers and the pose estimated by both localization algorithms, it is possible to compare the markers pose, and in this way understand which one of the algorithms performed better in terms of accuracy, this is possible to be done because the trajectory accuracy is directly related with the estimated pose of the tags.

When talking about the accuracy, the Figures 7.43, 7.44, 7.45 and 7.46 show a comparison between the estimated tags made by the Graph-Slam and ones obtained using the FARO scan.

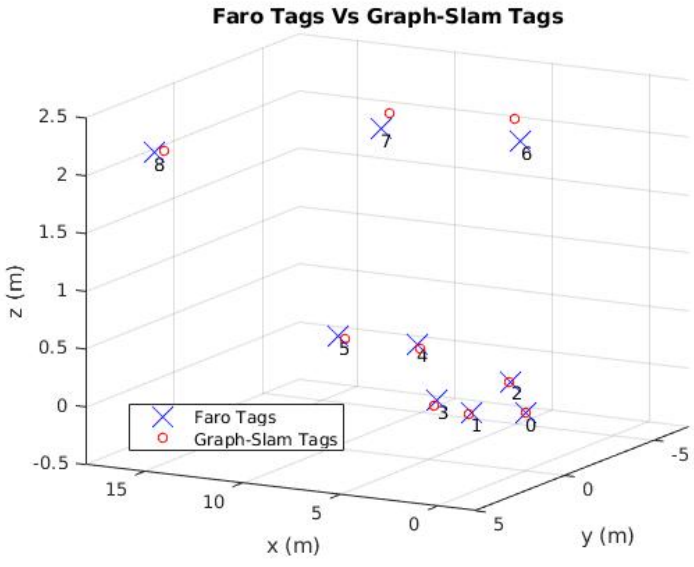

Figure 7.43: Faro estimated tags compared to Graph-Slam estimated tags.

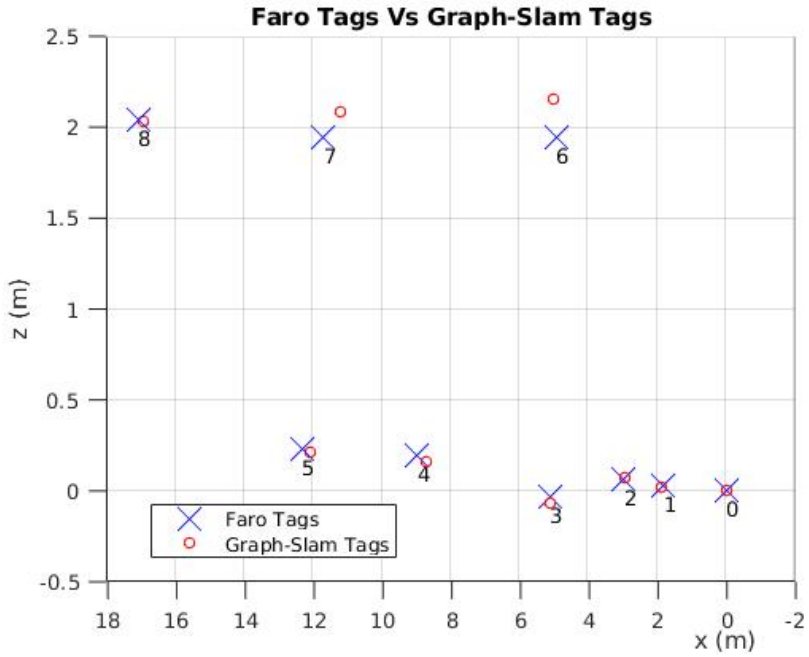

Figure 7.44: Faro estimated tags compared to Graph-Slam estimated tags (XZ).

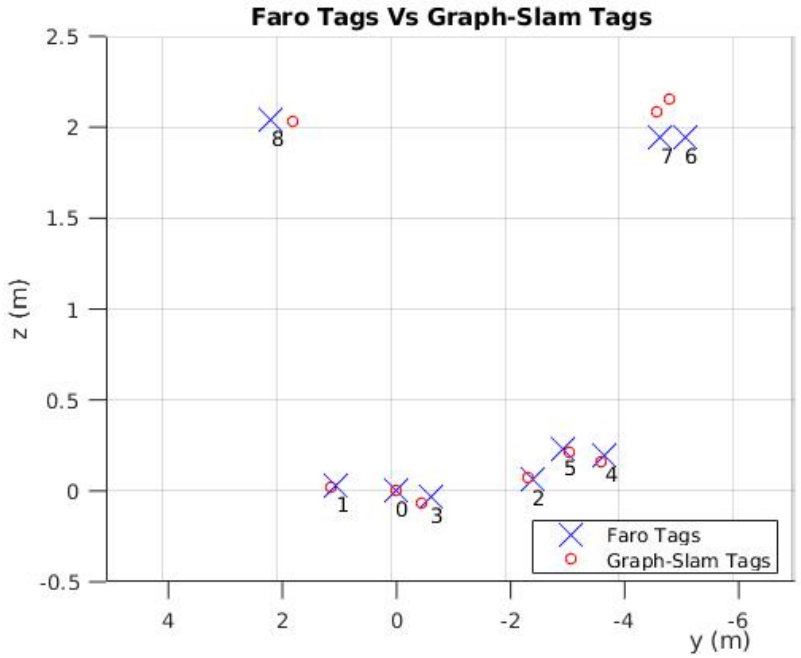

Figure 7.45: Faro estimated tags compared to Graph-Slam estimated tags (YZ).

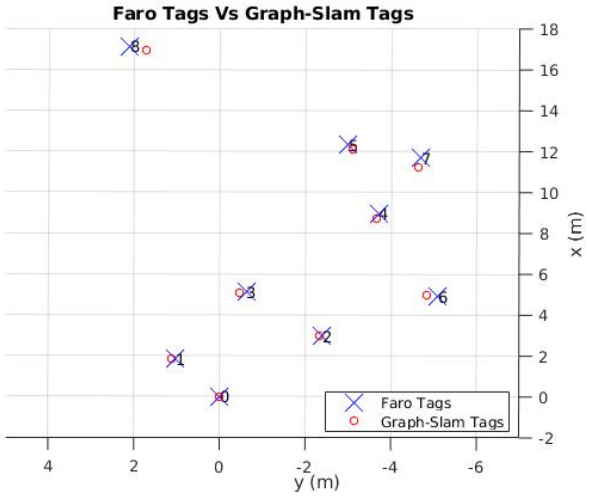

Figure 7.46: Faro estimated tags compared to Graph-Slam estimated tags (YX).

The previous results show that the markers estimated by the Graph-Slam algorithm are close to the ones obtained by the FARO scan, which displays a good accuracy regarding the developed algorithm. These results could be further improved by changing the A4 paper sheets in which the tags were printed for a better material with fewer irregularities. Note that the tags that were placed in the two vertical cardboards had the worst accuracy, this happened because the cardboards were not completely flat yielding in worst landmarks position estimation.

The UcoSlam results are presented in Figures 7.47, 7.48, 7.49 and 7.50.

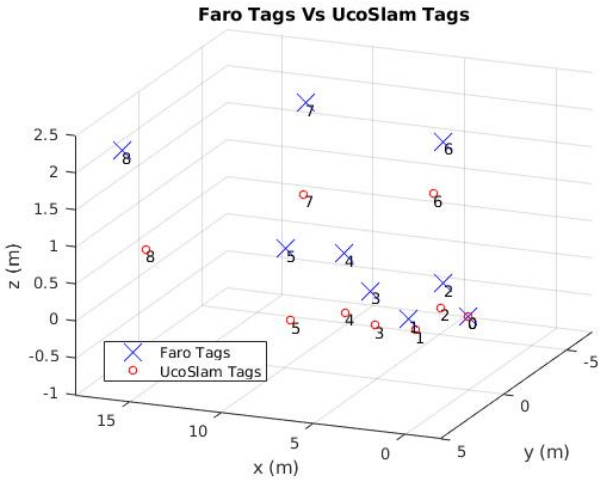

Figure 7.47: Faro estimated tags compared to UcoSlam estimated tags.

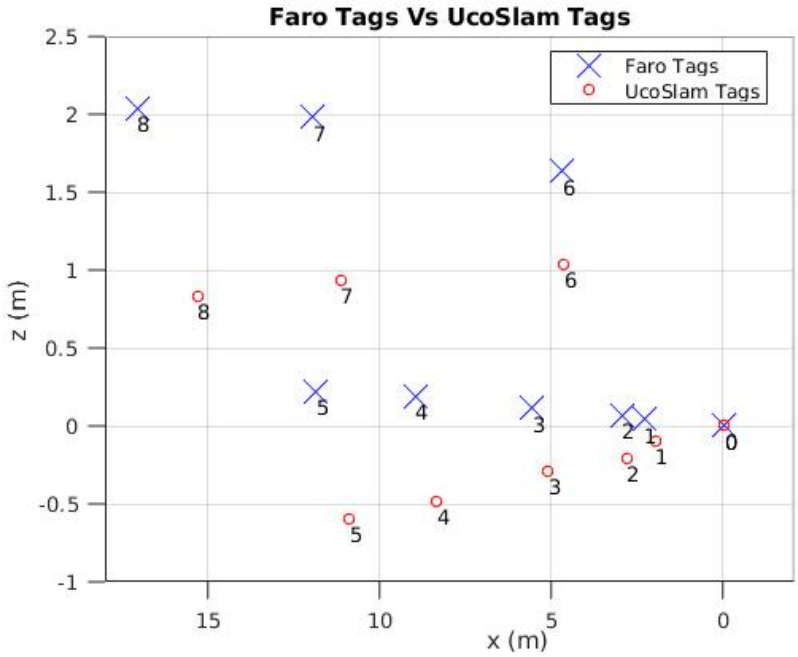

Figure 7.48: Faro estimated tags compared to UcoSlam estimated tags (XZ).

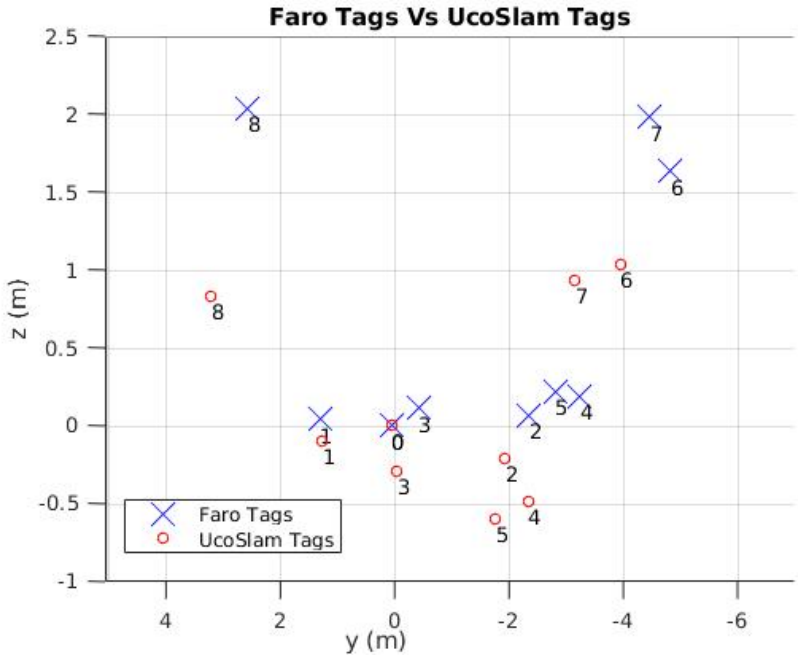

Figure 7.49: Faro estimated tags compared to UcoSlam estimated tags (YZ).

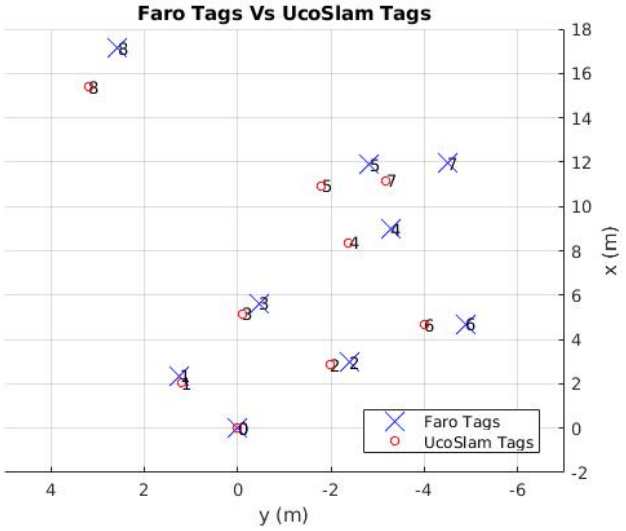

Figure 7.50: Faro estimated tags compared to UcoSlam estimated tags (YX).

Comparing the estimated tags obtained with the UcoSlam with the tags obtained by the scan, it is possible to conclude that the UcoSlam had a worse performance when compared to the Graph-Slam algorithm since the estimated tags are further from the ground truth ones.

The following tables show the quantification of the errors obtained by both algorithms estimations.

Table 7.1: Graph-Slam estimated tags errors.

Graph-Slam	Error X (m)	Error Y (m)	Error Z (m)	3D Error (m)
Tag 0	0	0	0	0
Tag 1	0.0371	0.0864	0.0090	0.0944
Tag 2	0.0086	0.0801	0.0020	0.0806
Tag 3	0.0190	0.1597	0.0403	0.1658
Tag 4	0.2326	0.0575	0.0310	0.2416
Tag 5	0.2278	0.1103	0.0207	0.2539
Tag 6	0.0807	0.2597	0.2109	0.3442
Tag 7	0.4762	0.0620	0.1412	0.5005
Tag 8	0.1596	0.3795	0.0084	0.4118
Mean (Not Counting Tag 0)	0.1552	0.1494	0.0579	0.2616

Table 7.2: UcoSlam estimated tags errors.

UcoSlam	Error X (m)	Error Y (m)	Error Z (m)	3D Error (m)
Tag 0	0	0	0	0
Tag 1	0.3119	0.0376	0.1383	0.3433
Tag 2	0.1295	0.4143	0.2731	0.5129
Tag 3	0.4636	0.3703	0.4023	0.7170
Tag 4	0.6246	0.8989	0.6724	1.2846
Tag 5	0.9527	1.0324	0.8211	1.6272
Tag 6	0.0392	0.8760	0.5977	1.0613
Tag 7	0.7998	1.3017	1.0474	1.8523
Tag 8	1.7490	0.6271	1.2059	2.2151
Mean (Not Counting Tag 0)	0.6338	0.6948	0.6448	1.2017

By observation of the previous tables, it is possible to conclude that in terms of accuracy the developed Graph-Slam algorithm performed much better than the UcoSlam algorithm since the UcoSlam mean 3D Error is more or less 4.6 times bigger than the Graph-Slam error.

This benchmark showed that for the project in question the Graph-Slam algorithm is much better in terms of accuracy and computational power.

7.4 Results Discussion

Regarding the developed android application, it is possible to conclude that the shown results reveal that the application works as intended, publishing all the ROS topics with the desired frequency while having an acceptable delay regarding the video feed.

The decompression of the video feed also worked as supposed, doing the decompression of the image without any type of problems or frame losses.

The Graph-Slam algorithm also performed flawlessly while having superior results when compared to the state of art algorithm UcoSlam. It is important to refer that during the performed test the UcoSlam performed much worse than what was expected, this might be due to the lack of distinctive natural features in the laboratory where all the tests were conducted and the lack of processing power from the computer where the algorithm was tested. The developed algorithm also has room for improvement in terms of accuracy, and this can be done by changing the source of the used raw localization

and by changing the type of material in which the AprilTags are being printed since the paper induces errors in the estimation of the orientation.

Finally, the Graph-Slam was also tested in conjunction with the navigation that was developed in another thesis. In these tests the vehicle had a defined mission to complete, that consisted in taking off, going to a point that simulated the "inventory" in a warehouse 4.0 scenario, taking a photo of the "inventory", return to the initial position, and land. Note that before starting the autonomous mission, the vehicle made three manual flights to have the best localization estimation possible. Even though the drone was able to complete the mission, it was noticed that there was some problem with the localization that happened during some trials. The problem was related to the raw localization used to estimate the pose of the drone between markers. This problem exists because of the resolution of the velocity coming from the DJI SDK which is a 0.1 m/s resolution. For example, when the navigation is sending a velocity of 0.05 m/s to the drone, even though it is moving with this velocity, the data coming from the sensors are telling that the drone is stopped. If the camera is not seeing any AprilTag, the localization considers that the vehicle is indeed stopped, which is not true. If eventually, the AprilTags detector detects a previously seen marker, the localization closes the loop and corrects the pose of the drone, but if no marker is detected during a long time, the localization gets compromised and the mission fails. Unfortunately, there is no way to change the resolution of the velocity coming from the DJI SDK sensor data, for this reason, the only way of fixing the problem is to use more markers or to change the raw localization source. Since using more markers is not a suitable option regarding the project objectives, the only viable option is to change the raw localization source in the future. A good alternative to the actual raw localization is to use visual odometry techniques to estimate the velocity or pose of the drone.

Chapter 8

Conclusions

The work present in this thesis addressed the development of an indoor localization framework for drones in the context of logistic operations inside a warehouse. The technique that has been developed is based on Graph-SLAM, and the obtained results allow us to expect to integrate the algorithm during real-time logistic operations. A benchmark has been performed to validate the developed method by comparing the drone estimated position with the UcoSlam algorithm and an exogenous system composed of a static LiDAR to extract the Tags position. Even though the algorithm performed better than the UcoSlam in terms of accuracy and computational power, it still has a lot of room for future improvements.

One of the big advantages of the Graph-SLAM is that it can be easily adapted to be used in other aircraft platforms as long as the platforms have a camera and an estimated velocity and attitude which is the case with all DJI platforms. The algorithm is also able to solve the localization problem without the need for any extra sensors, which was a main requirement of the project since in most cases it is not easy to add extra hardware to premade UAV platforms.

The first version of the Graph-SLAM was published in the 2021 ICARSC international conference with the title Graph-SLAM Approach for Indoor UAV Localization in Warehouse Logistics Applications [4].

Regarding the developed android application, it satisfied all the project requirements by being able to map all the sensor data as ROS topics to be used in a ground station.

For all the previous reasons it is possible to conclude that all the project objectives and requirements were completed.

For future works, it is planned to change AprilTags printed on paper by another type of material capable of providing a better estimation mainly in the estimation of the aircraft

orientation. One limitation of the DJI SDK is not providing the IMU data to estimate the drone displacement. Therefore, it is also intended to explore a visual odometry method, able to be integrated with the Grpah-SLAM, to improve drone localization during navigation between tags.

Bibliography

[1] FlytBase. Drone Automation For Warehouse 4.0. pages 0–14, 2019.

[2] Lukas Wawrla, Omid Maghazei, and Prof. Dr. Torbjørn Netland. Applications of drones in warehouse operations. (August):13, 2019.

[3] Tiago M. Fernández-Caramés, Oscar Blanco-Novoa, Iván Froiz-Míguez, and Paula 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 (Basel, Switzerland)*, 19(10), 2019.

[4] A. Moura, J. Antunes, A. Dias, A. Martins, and J. Almeida. Graph-SLAM Approach for Indoor UAV Localization in Warehouse Logistics Applications. *2021 IEEE International Conference on Autonomous Robot Systems and Competitions, ICARSC 2021*, pages 4–11, 2021.

[5] Robin Amsters, Ali Bin, Samrat Roy, Peter Slaets, Peter Aerts, Maarten Verheyen, and Eric Demeester. A Literature Study of Indoor Positioning Systems and Map Building Software. 2017.

[6] Rainer Mautz. Indoor Positioning Technologies Habilitation Thesis. *ETH Zurich, Department of Civil, Environmental and Geomatic Engineering, Institute of Geodesy and Photogrammetry*, page 10, 2012.

[7] Anca Morar, Alin Moldoveanu, Irina Mocanu, Florica Moldoveanu, Ion Emilian Radoi, Victor Asavei, Alexandru Gradinaru, and Alex Butean. A comprehensive survey of indoor localization methods based on computer vision. *Sensors (Switzerland)*, 20(9):1–36, 2020.

[8] Francisco J. Perez-Grau, Ricardo Ragel, Fernando Caballero, Antidio Viguria, and Anibal Ollero. An architecture for robust UAV navigation in GPS-denied areas. *Journal of Field Robotics*, 35(1):121–145, 2018.

[9] Tao Yang, Peiqi Li, Huiming Zhang, Jing Li, and Zhi Li. Monocular vision SLAM-based UAV autonomous landing in emergencies and unknown environments. *Electronics (Switzerland)*, 7(5), 2018.

[10] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: An efficient alternative to SIFT or SURF. *Proceedings of the IEEE International Conference on Computer Vision*, 2011.

[11] Heng Deng, Usman Arif, Kun Yang, Zhiyu Xi, Quan Quan, and Kai Yuan Cai. Global optical flow-based estimation of velocity for multicopters using monocular vision in GPS-denied environments. *Optik*, 219(May):164923, 2020.

[12] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[13] Bruce D. Lucas and Takeo Kanade. ITERATIVE IMAGE REGISTRATION TECHNIQUE WITH AN APPLICATION TO STEREO VISION. 2, 1981.

[14] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering, Transactions of the ASME*, 82(1):35–45, 1960.

[15] Simon Zingg, Davide Scaramuzza, Stephan Weiss, and Roland Siegwart. MAV navigation through indoor corridors using optical flow. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3361–3368, 2010.

[16] Jianbo Shi and Carlo Tomasi. Good features to track. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1994.

[17] Raul Mur-Artal and Juan D. Tardos. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[18] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment – a modern synthesis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1883, 2000.

[19] Dorian Gálvez-López and Juan D. Tardós. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.

[20] Rafael Muñoz-Salinas, Manuel J. Marín-Jimenez, and R. Medina-Carnicer. SPM-SLAM: Simultaneous localization and mapping with squared planar markers. *Pattern Recognition*, 86(September):156–171, 2019.

[21] Bernd Pfrommer and Kostas Daniilidis. TagSLAM: Robust SLAM with Fiducial Markers. 2019.

[22] Simon Parsons. Probabilistic Graphical Models: Principles and Techniques by Daphne Koller and Nir Friedman, MIT Press. *The Knowledge Engineering Review*, 26(2), 2011.

[23] Frank Dellaert. Factor Graphs and GTSAM. *Technical Report*, (GT-RIM-CPR-2012-002):1–27, 2012.

[24] Rafael Muñoz-Salinas and R. Medina-Carnicer. UcoSLAM: Simultaneous localization and mapping by fusion of keypoints and squared planar markers. *Pattern Recognition*, 101, 2020.

[25] Bernhard L Ecklbauer. A Mobile Positioning System for Android based on Visual Markers. *Academia.Edu*, page 100, 2014.

[26] Gaetano C. La Delfa, Salvatore Monteleone, Vincenzo Catania, Juan F. De Paz, and Javier Bajo. Performance analysis of visualmarkers for indoor navigation systems. *Frontiers of Information Technology and Electronic Engineering*, 17(8):730–740, 2016.

[27] B. Benligiray, C. Topal, and Cuneyt Akinlar. STag: A stable fiducial marker system. *Image and Vision Computing*, 89:158–169, 2019.

[28] John Wang and Edwin Olson. AprilTag 2: Efficient and robust fiducial detection. *IEEE International Conference on Intelligent Robots and Systems*, 2016-November:4193–4198, 2016.

[29] Erik Strömberg. *Smoothing and Mapping of an Unmanned Aerial Vehicle Using Ultra-wideband Sensors*. PhD thesis, 2017.

[30] Frank Dellaert and Michael Kaess. Square root SAM: Simultaneous localization and mapping via square root information smoothing. In *International Journal of Robotics Research*, volume 25, 2006.

[31] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, and John J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[32] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. `https://www.ros.org`. Online; Accessed on Aug. 25, 2021.

[33] Kohler, Damon and Queiro, Rodrigo and Corbellini, Ernesto. rosjava. `https://github.com/rosjava`. Online; Accessed on Aug. 25, 2021.

[34] Michael Kaess, Hordur Johannsson, Richard Roberts, Viorela Ila, John J. Leonard, and Frank Dellaert. ISAM2: Incremental smoothing and mapping using the Bayes tree. *International Journal of Robotics Research*, 31(2):216–235, 2012.

[35] DJI. Dji camera application. `https://developer.dji.com/document/06724d27-23cf-4741-b128-fc17d2891981`. Online; Accessed on Sept. 4, 2021.

[36] DJI. Dji developer. `https://developer.dji.com/`. Online; Accessed on Sept. 4, 2021.

[37] DJI. Dji mobile sdk documentation. `https://developer.dji.com/api-reference/android-api/Components/SDKManager/DJISDKManager.html`. Online; Accessed on Sept. 4, 2021.

[38] OpenCV. Opencv. `https://opencv.org/`. Online; Accessed on Sept. 4, 2021.

[39] MathWorks. Single camera calibrator app. `https://www.mathworks.com/help/vision/ug/single-camera-calibrator-app.html`. Online; Accessed on Sept. 4, 2021.

[40] ROS. image_proc - ros wiki. `http://wiki.ros.org/image_proc`. Online; Accessed on Sept. 9, 2021.

[41] ROS. tf - ros wiki. `http://wiki.ros.org/tf`. Online; Accessed on Sept. 9, 2021.

[42] William Wang. Apriltag frame transforms. `http://willshw.me/2018/04/27/apriltag2-ros-frame.html`. Online; Accessed on Sept. 9, 2021.

Appendix A

Non Simplified Proposed Architecture

Proposed Architecture in the next page.

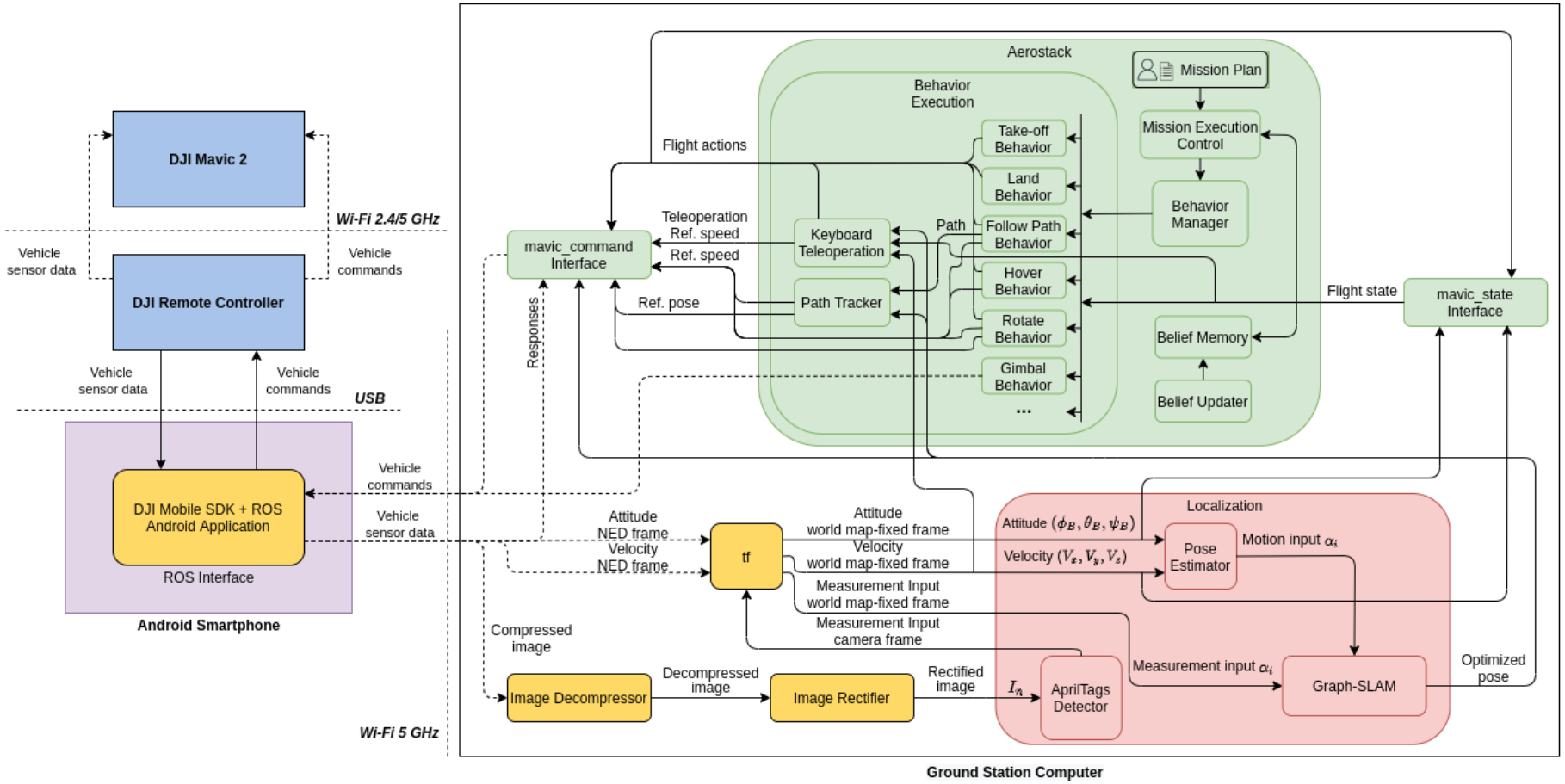

Figure A.1: Non Simplified Proposed Architecture.