POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Cooperative SLAM Optical Sensor-Based Architecture for Multi-UAVs systems

Supervisors

Giorgio GUGLIERI

Simone GODIO

Candidate

Nicola RUTIGLIANO

April 2021

Abstract

Simultaneous Localization And Mapping (SLAM) is a well-known problem in a variety of cases, such as ground robots, drones, cars, and underwater robots. To perform efficiently SLAM is even more crucial when autonomous navigation must be guaranteed. UAVs are widely taking the scene in many applications, like search and rescue missions, real-estate applications, agriculture, system logistic, and package delivery, thanks to their dexterity in movements, even in narrow environments. Many of the state-of-the-art SLAM algorithms find a common factor in the usage of Visual - Visual Inertial Odometry (VO-VIO).
Indeed, more and more solutions to the problem are developed, bringing new approaches, or improving existing ones. Recent researches include neural-networks too. [1][2]
The search for more information to integrate into SLAM algorithms has brought to investigate systems with multiple agents that collaborate to build a global map and use the shared information to improve the localization process.
The objective of this thesis is the development of a strategy to perform multi-agent SLAM for a fleet of drones. Different combinations of inertial sensors, cameras, and range finders are tested, along with sensor fusion algorithms with the idea of improving the localization process as far as possible.
Then, general SLAM algorithms are described, analyzed, and tested. The development is thought for a case in which each drone is provided with the same sensor suite, that comprehends a tracking camera module, a depth camera module, and a low-budget onboard computer. Anyway, it is possible to use the same logic to combine data coming from different robots, e.g. system which combines both ground robot and drones, with different sensor suites.
This project is carried on in the ROS environment, which allows taking advantage of already existent libraries and communication protocols. Precisely, it is shown how a ROS package has been created and developed to generate a set of nodes that provide features for the multi-agent SLAM, such as data compression and decompression, point cloud filtering, and concatenation. It is shown that, even with limited computational resources and a standard wireless connection, it is possible to share information without loss of data consistency and a global map can be created and then forwarded to each agent.

Table of Contents

List of Tables

List of Figures

Acronyms

SLAM
Simultaneous Localization and Mapping

UAV
Unmanned Aerial Vehicle

UGV
Unmanned Ground Vehicle

IMU
Inertial Measurement Unit

ROS
Robot Operating System

GUI
Graphic User Interface

Lidar
Laser imaging, detection, and ranging

PnP
Perspective-n-Points

BA
Bundle Adjustment

SDK
Software Developement Kit

FOV

Field of View

VPU

Vision Processing Unit

DOF

Degrees Of Freedom

MRPT

Mobile Robot Programming Toolkit

EKF

Extended Kalman Filter

UKF

Unscented Kalman Filter

AI

Artificial Intelligence

PCL

Point Cloud Library

FPS

Frames Per Second

API

Application Programming Interface

Chapter 1

Multi-UAV SLAM: introduction

1.1 The SLAM problem

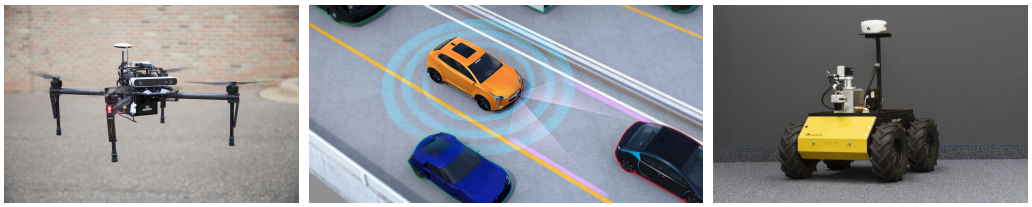
This section addresses the simultaneous localization and mapping problem, commonly known as SLAM. This problem arises when it is required for a robot to construct or update a map and, simultaneously, keep track of its location. Given those premises, it is easy to understand the complexity of the problem, especially compared to the two main tasks taken singularly:

- Localization problem: the complexity lies on the unknown map, thus the absence of a-priori information about the environment.
- Mapping problem: the complexity lies in the lack of information on the pose.

On top of this, there must be added project constraints, such as:

- Autonomy
- Computational cost
- Real-time requirements
- Memory management
- Available sensor suite
- Budget

There are many applications in which SLAM is employed, such as self-driving cars, unmanned ground vehicles (UGVs), and unmanned aerial vehicles (UAVs). The last one is the subject of interest for this project.
Indeed, the SLAM problem is far more complex for a flying vehicle such as a drone with respect to a ground vehicle, since both localization and mapping should be performed in a three-dimensional fashion because a larger amount of data occurs, other than more sources of noise and the difficult task of building a 3D map.



(a) UAV (b) Car (c) UGV

Figure 1.1: Examples of SLAM applications. The SLAM problem is of great interest in many different areas

Even if SLAM suites are based on a variety of hardware and software possibilities and different combinations of the two, the mathematical formulation of the problem is quite the same.
Given a series of controls u_t and sensor observations o_t over discrete time steps t, the SLAM problem is to compute an estimate of the agent's state x_t and a map of the environment m_t.
All quantities are usually probabilistic. If this problem only involves the estimation of variables that persist at time t, it is called *online SLAM*, while it is called *full SLAM* if past measurements and controls are not discarded, as in the online, but processed. Statistical techniques used to estimate those variables include Kalman filters and particle filters.
However, *Visual SLAM* is the main subject of interest here. Visual SLAM is a framework that provides the fusion of landmark features obtained from both the visual modalities within an environment.
For applications in mobile robotics (ex. drones, service robots), it is valuable to use low-power, lightweight equipment such as monocular cameras. Visual SLAM can also allow for the complementary function of such sensors, by compensating the narrow field-of-view, feature occlusions, and optical degradation common to lightweight visual sensors with the full field-of-view.
At the state of the art, there are a lot of different algorithms that allow achieving visual SLAM, which will be briefly presented later.
The main differences between those algorithms are related to sensor interface,

usually monocular or stereo camera and IMUs, data processing, map building techniques, and output generation.
However, there is a factor that is quite common to all those algorithms, i.e. the development platform ROS. The following section will describe its basics.

1.1.1 ROS basics

Although the name may suggest it, ROS is not properly an operating system, but it's a collection of software frameworks and it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.
At the core of ROS are four different concepts, underpinning its philosophy:

- *Plumbing*: which means that different programs can run at the same time and ROS enables the communication between different pieces of software. ROS acts as a "plumbing" in the sense that it provides the device drivers needed for communication between software and hardware.

- *Tools*: the second core part of ROS is the set of tools that it provides. Many basic tools are already implemented by ROS, being it simulations, visualization, GUIs, or basic tools for data logging'.

- *Capabilities*: capabilities can be thought of as high-level tools. They are software that can be installed on ROS and enable mapping, localization, planning, etc. These capabilities are provided by the Open Source community and enable the research teams to focus on single areas while being able to rely on the state-of-the-art solution to be customized to their specific needs.

- *Ecosystem*: being the de facto research standard, ROS boasts vast documentation, tutorials provided for most of its software, and great compatibility.

The inner-workings of ROS can be best understood by looking at its basic components:

- *Nodes*: processes that perform computation. A robot control system usually comprises many nodes. Nodes are used for different purposes, such as wheel motors control, localization, path planning, graphical view, and so on.
  Nodes are combined together into a graph and communicate with one another using streaming topics. The use of nodes in ROS provides several benefits to the overall system. Nodes architecture provides benefits: additional fault tolerance, as crashes are isolated to individual nodes and reduced code complexity in comparison to monolithic systems.

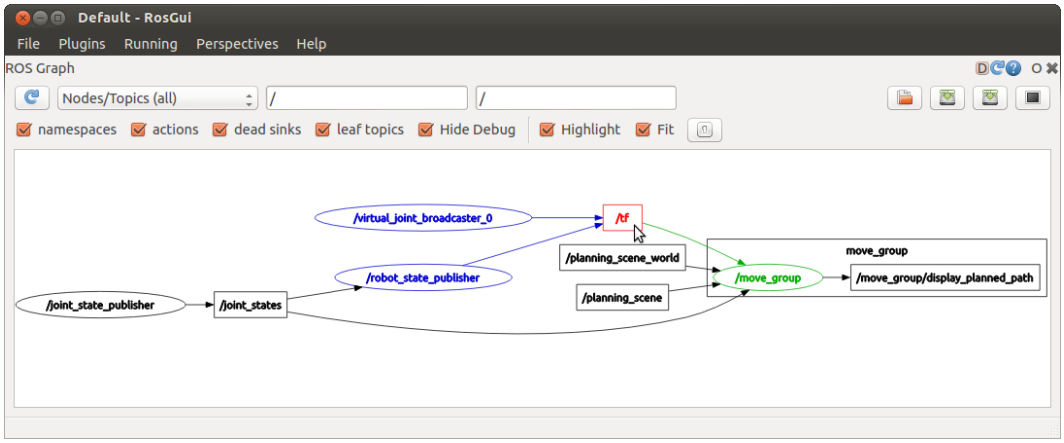


Figure 1.2: RQT graph: example of nodes architecture with one of the tools provided by ROS

- *Messages*: ROS nodes communicate with each other with messages: they are a simple data structure, comprising typed fields.
  Standard primitive types (integer, floating-point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).

- *Topics*: Messages are routed via a transport system with publish-subscribe semantics. A node sends out a message by publishing it to a given topic.
  The topic is a name that is used to identify the content of the message.
  A node that is interested in a certain kind of data will subscribe to the appropriate topic.
  There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.
  In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption.

- *Services*: the publish-subscribe model is a very flexible communication paradigm, but its "one-way" transport is not appropriate for request-reply interactions, which are often required in a distributed system.
  Request-reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply.
  A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

- *Master*: ROS master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics.
  The role of the Master is to enable individual ROS nodes to locate one another

and to communicate with the peer-to-peer protocol.
Nodes connect to other nodes directly; the Master only provides lookup information.
Nodes that subscribe to a topic will request connections from nodes that publish that topic and will establish that connection over an agreed upon connection protocol.

ROS's core functionality is augmented by a variety of tools that allow developers to visualize and record data, easily navigate the ROS package structures, and create scripts automating complex configuration and setup processes.
The addition of these tools greatly increases the capabilities of systems using ROS by simplifying and providing solutions to a number of common robotics development problems.
These tools are provided in packages like any other algorithm, but rather than providing implementations of hardware drivers or algorithms for various robotic tasks, these packages provide task and robot-agnostic tools. Following, a brief description of those tools, that will make the explanation of this project clearer.

- *rviz*: it is a three-dimensional visualizer used to visualize robots, the environments they work in, and sensor data. It is a highly configurable tool, with many different types of visualizations and plugins.

- *rosbag*: it is a command-line tool used to record and playback ROS message data.
  It uses a file format called bags, which log ROS messages by listening to topics and recording messages as they come in.
  Playing messages back from a bag is largely the same as having the original nodes which produced the data in the ROS computation graph, making bags a useful tool for recording data to be used in later development.

- *catkin*: catkin is the ROS build system, it is based on CMake, and is similarly cross-platform, open-source, and language-independent.

- *roslaunch*: it is a tool used to launch multiple ROS nodes both locally and remotely, as well as setting parameters on the ROS parameter server.
  roslaunch configuration files, which are written using XML can easily automate a complex startup and configuration process into a single command.
  roslaunch scripts can include other roslaunch scripts, launch nodes on specific machines, and even restart processes that die during execution.

ROS contains many open-source implementations of common robotics functionality and algorithms. These open-source implementations are organized into *packages*. Many packages are included as part of ROS distributions, while others may be

developed by individuals and distributed through code-sharing sites.
Among the packages included in ROS, there are some worth mentioning, such as *tf*, which provides a system for representing, tracking, and transforming coordinate frames, and *gazebo*, which integrates tools to use a simulation environment.
The understanding of ROS functionalities, along with the possibility of creating customized nodes, are important pre-requisites that allow to use or create a SLAM package.

1.1.2 Sensing

This section will present different types of sensors and different sensor suite configurations, along with *Sensor Fusion* algorithms, which are best suited for SLAM. Sensors are a fundamental component to achieve robust localization and detailed mapping.
Of course, it doesn't exist a unique solution, but it depends on the task a robot must achieve, along with external factors.
When a sensor suite is chosen, it is always recommended to take into account:

- Types of robots: configurations might change a lot from a flying robot to a ground robot or an underwater robot.
- Robot's payload: usually this is a much bigger problem for flying robots, so heavier or bulky sensors are highly not recommended.
- Available power supply: autonomy is an important issue, and some kinds of sensors are quite demanding.
- Budget: some kinds of sensors are very expensive and not always suited for a project.
- Environment information: e.g. indoor, outdoor, shadows, windows, lights.

Moreover, the task of each sensor might be related only to localization, only to mapping or it can perform both.
At the state of the art, the most used solutions for the mapping task are based on cameras and lasers.
Laser-based systems are one of the most popular choices for solving the SLAM problem. Laser-based systems can obtain robust results in both indoor and outdoor environments.
The high speed and high accuracy of laser range finders enable them to generate highly precise distance measurements. The most popular laser-based system is Lidar.
Lidar is commonly used to make high-resolution maps. It uses ultraviolet, visible,

or near-infrared light to image objects. It can target a wide range of materials, including non-metallic objects, rocks, rain, chemical compounds.
An example of 3D Lidar sensor is shown in Figure 1.3

Figure 1.3: Example of 3D Lidar sensor: this model is suited for indoor applications

This technology is being used in robotics for the perception of the environment as well as object classification. The ability of lidar technology to provide three-dimensional elevation maps of the terrain, high precision distance to the ground and approach velocity can enable the safe landing of robotic and manned vehicles with a high degree of precision.
If the mapping task is considered, nothing is better than Lidar: its level of details, along with a very wide field of view and amazing precision is off the charts.
Moreover, all those perks might hide some drawbacks: not only it's a very expensive sensor, but also it is bulky, heavy and its power consumption isn't negligible, especially for a small/medium-sized drone.
For these reasons, it will not be considered in the further analysis on the sensor suites, nor in the deepening of SLAM algorithms that are based on this technology.
There is, actually, another reason that justifies this choice: the research on SLAM algorithms all over the world, even with few exceptions, is strongly focused on visual or visual-inertial systems.
The progress in the computer vision field in the last years is remarkable and a lot of environmental information can be extracted from images. A lot of the algorithms that will be described later depend always on a camera, stereo or monocular, and in many cases, on inertial sensors, i.e. IMUs.
An IMU is an electronic device that is used a lot in navigation systems, especially in aircraft applications: it consists of a combination of an accelerometer, a gyroscope

and, usually, a magnetometer.
The latter is used as a heading reference, other than provide data useful for calibration procedures. Figure 1.4 illustrates an example of IMU.

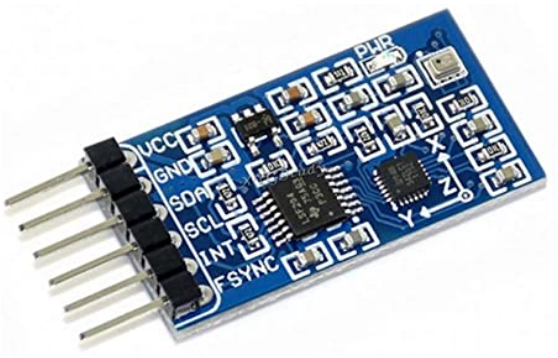


Figure 1.4: Example of IMU: to make the usage easier, it is mounted on a PCB which already provides all the needed pins

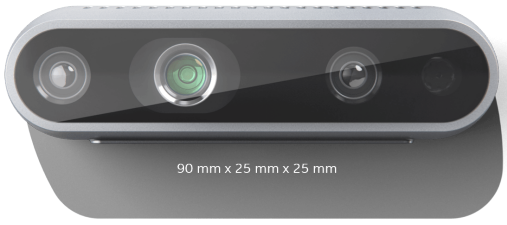
The algorithms that are based on cameras and IMUs technologies are part of what it's called *Visual SLAM* or *V-SLAM*.
The main drawbacks of those solutions lie in the fact that the complexity of the algorithm is way higher, since cameras have a limited field of view, in opposition to Lidars, that allows 360° mapping.
Besides, these algorithms require that the 3D pose of the camera over time should be estimated with enough precision.
Even if IMUs are quite useful when it is required to track the motion of an object, it is quite clear that an inertial sensor, even a high-quality one, tends to provide divergent data.
The choice has then switched to the usage of cameras to perform tracking. Moreover, a lot of approaches use a combination of camera and image data to obtain reliable tracking.
This is one of the great advantages of the visual approach: with just one sensor it is possible to carry on both localization and mapping tasks.
The firsts works in the V-SLAM area were based on monocular cameras and a *features-based* approach.
Then, the evolution of camera technologies and vision has brought to the usage of more and more detailed images, which now carry on depth information too.[3]
This is done through stereo cameras, based on the stereo vision concept: it is the extraction of 3D information from digital images; by comparing information about a scene from two vantage points, 3D information can be extracted by examining the relative positions of objects in the two panels.

It is necessary to describe how a V-SLAM approach works before going any further in the description of monocular and stereo cameras.
First of all, it is necessary to pay attention to coordinate systems: a global reference frame should be defined, along with a reference frame attached to the camera, to allow camera pose estimation and perform environment reconstruction.
Tracking and mapping are performed to continuously estimate camera poses.
The tracking task is carried on using correspondences between the image and the map through feature matching or feature tracking in the image. Then, the camera pose is computed by solving the Perspective-n-Point (PnP) problem.
Usually, V-SLAM algorithms assume that intrinsic camera parameters are previously calibrated. Some of the most recent devices don't even need calibration or, in worst cases, the SDK provides all the tools. This allows defining the camera pose as a roto-translation in the global reference frame.
There are two more features inside V-SLAM algorithms that recur very often:

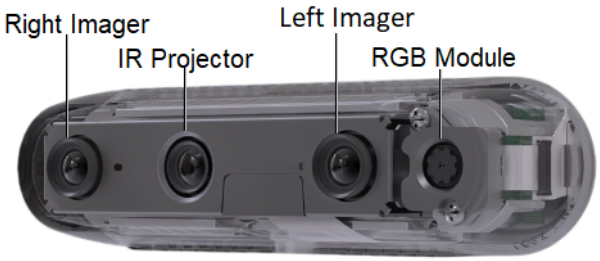
- Re-localization
- Global map Optimization

Re-localization is the ability of an algorithm to find the robot pose when, for some reason, the tracking is lost. This problem is also known as the *"kidnapped robot problem"*.
Losing track usually happens when fast camera motion, bumps, or external disturbances occur. If it happens, the robot should recover its pose from the map, otherwise, any SLAM algorithm becomes useless.
Another important feature that, as well as re-localization is always integrated into SLAM algorithms, is the *"global map optimization"*. As the robot, therefore the camera, keeps moving, the produced map can include more and more errors. If this error accumulates and keeps growing, the map is useless.
Global map optimization comes in handy in this situation: the map is refined by taking into account the whole map information. To make this happen, the region of the map in which the starting point of the robot movement is should be re-visited. This allows global optimization to calculate the error and adjust the map. The improvement is not limited to that region, but all the map is corrected accordingly.
Loop closing is a technique to acquire reference information. A closed-loop is defined by searching for a match between a current image and a previously acquired one.
If the loop is detected, it means that the camera captures a previously seen scene. In this case, the accumulative error can be estimated.
Bundle adjustment (BA) is another content that can minimize the error of the map by optimizing both the map and the camera poses. When a task requires to inspect a large environment, this procedure minimizes estimation errors in the

most efficient way. In small environments, since the error is small, BA may be performed without loop closure.
Cameras are responsible of *visual odometry* too. Odometry is to estimate the sequential changes of sensor positions over time using sensors such as wheel encoder to acquire relative sensor movement.
Visual odometry brings quite a different approach, providing the same information as classic odometry, but with cameras.
The advantage of this solution is the independence from encoder data: it is well known that wheel odometry is not always reliable, since the estimation of a robot pose from wheels suffers from measurement errors due to encoders and external factors such as friction and rough environments.
Moreover, classic odometry cannot be applied to robots that have no wheels, such as underwater robots or flying ones, like drones.
The usage of visual odometry does not rule out the usage of wheel odometry when it's available. This usually depends on the task a robot is required.
One of the most efficient products available on the market is the *Intel Depth Camera*.



(a) D435 Front view



(b) D435 Modules

Figure 1.5: Intel D435 Depth Camera: front view of the device and single module description

The Intel RealSense depth camera D435 is a stereo solution, offering reliable depth data. Its wide field of view is perfect for applications such as robotics or augmented and virtual reality.
With a range of up to 10 meters, this small form factor camera can be integrated into any solution with ease.
The combination of a wide field of view and global shutter sensor on the D435 make it the preferred solution for applications such as robotic navigation and object recognition. The global shutter sensors provide great low-light sensitivity allowing robots to navigate spaces with the lights off.
Other than the remarkable hardware specifications, the sensor is paired with a standalone application that, not only allows to visualize all the images but also lets the user explore among a large variety of configurations, resolutions, data format, calibration, filtering, and many post-processing tools.
Moreover, there's also a version of this product that embeds a motion module, employing an IMU, which makes it even a better fit for SLAM. It seems obvious that this product fits perfectly in Visual SLAM applications. Indeed, the presence of a ROS wrapper, called *realsense-ros*, and the presence of a wide SKD, called *librealsense*, makes the integration in SLAM systems as easy as possible. More details will be discussed later.
Another important detail about cameras is that with new technologies a lot of producers included graphic processors inside their hardware.
In this way, image acquisition, pre-processing, and post-processing can be performed on the camera, by avoiding this kind of burden to the main embedded hardware of the robot.
The Intel RealSense tracking camera T265, which is shown in Figure 1.6, belongs to this category.

Figure 1.6: T265 module: two fisheyes and an IMU to provide robot tracking

This sensor is born with the intent to run VIO algorithms completely on the camera. To do this the hardware set-up consists essentially of two wide fisheye cameras 165-degree circular FOV.
It also incorporates an IMU and the Intel® Movidius™ Myriad™ 2 VPU. The embedded processor runs the entire SLAM algorithm onboard, analyzes the stereo images, and fuses all sensor information.
This device is intended to take care of an important part of SLAM by itself, but it allows to explore alternatives too, since it is possible to manage all the modules inside this camera, included the IMU.
Obviously, in that case, the main difference between the two described products is related to the fact that D435 cameras have a smaller field of view. Indeed, SLAM should be run on the embedded platform, increasing the required power and cost. The producer usually suggests using these two products together.
There is another sensor that it's worth mentioning, i.e. the ZED 2 camera, shown in Figure 1.7.

Figure 1.7: ZED 2 camera: combination of tracking and depth functions in one, high-performance sensor

It's safe to say that the performances of this stereo module are excellent with respect to most of the products on the market, which makes the ZED 2 one of the most recommended stereo cameras available.
It benefits a 9-DOF IMU, that includes an accelerometer, gyroscope, and magnetometer, a wide field of view camera, a long baseline, and thermal calibration.
If compared to the devices described previously, it can be said that most of the features of the tracking camera and the depth camera are included in this device. Unfortunately, the ZED 2 camera is quite large and its power consumption is considerable.
Therefore, it is not recommended for applications like drones, in which the volume, weight, and power are relevant constraints.
The last solution that it will be presented is one of the most recent products by Intel, i.e. the Intel RealSense L515 Lidar camera. Figure 1.8 illustrates this device.

Figure 1.8: Lidar camera L515: it consists of a RGB camera and a laser scanner.

This product consists of a laser scanner based on Lidar technology, a full-HD RGB camera, and an IMU. The combination of these sensors makes it a state-of-the-art solution in many fields, such as logistics and environmental scanning.
Since its depth range is quite limited, especially if compared to the ones of the depth cameras available on the market, it is best suited for indoor operations.
Even if its main purpose is related to tasks like 3D object scanning, it possesses all the characteristics to be a source of information for SLAM algorithms, since it provides depth information and it has a motion module.
Its reduced size and limited power consumption are features that make the L515 Lidar camera a potential best fit for drone applications.
Anyway, in this case, it is not considered for some reasons. The first one is the price: if compared with the D435 and T265 suite, it is very expensive.
Moreover, it has a practical range that is very limited (about 4 meters). Finally, it is a quite recent product, therefore it is hard to find a lot of algorithms that are suited for its usage and, even if some packages available online can use the L515 data, only a few users tried to do it. Thus, it is hard to find online support in the SLAM community.
With this in mind, the Lidar camera solution has been discarded, but it might be analyzed in the future.
Among all the presented solutions, the choice fell on the Intel suite, which consists of T265 and D435 cameras.

1.2 ROS packages

1.2.1 Sensor Fusion Packages

At the state-of-the-art, many SLAM packages are based on VIO technologies, which basically can be seen as a fusion between the graphic, odometric input, and inertial measurement. However, nothing prevents using, on the side, other localization inputs that increase the robustness of the system.
Involving a sensor fusion package in the SLAM system is very convenient, especially if there's a concrete chance that the primary localization system fails, or if the robot has two (or more) comparable sources of localization data.
In this context, comparable means that their accuracy is quite similar, i.e. the noises affecting data are all more or less in the same range.
Aerial vehicles usually must respect, more than other cases, tight weight and size constraints, so it is rare to see a large sensor suite.
Indeed, it might not be so uncommon to find a drone that embeds a camera, one or more IMUs, and global position sensors, like GPS.
Another thing to keep in mind is that many of the state-of-the-art sensors used for localization already include some sensor fusion functionalities, calibration tools, and noise filtering tools.
With these functionalities, some basic operations, like extracting orientation from IMU data, are already implemented and streamed by the sensor itself.
ROS supports some packages that can perform sensor fusion.
Usually, those packages are based on well-known filters, such as median, Madgwick, or Kalman filters. A possible flow chart for a sensor fusion package is shown in Figure 1.9.

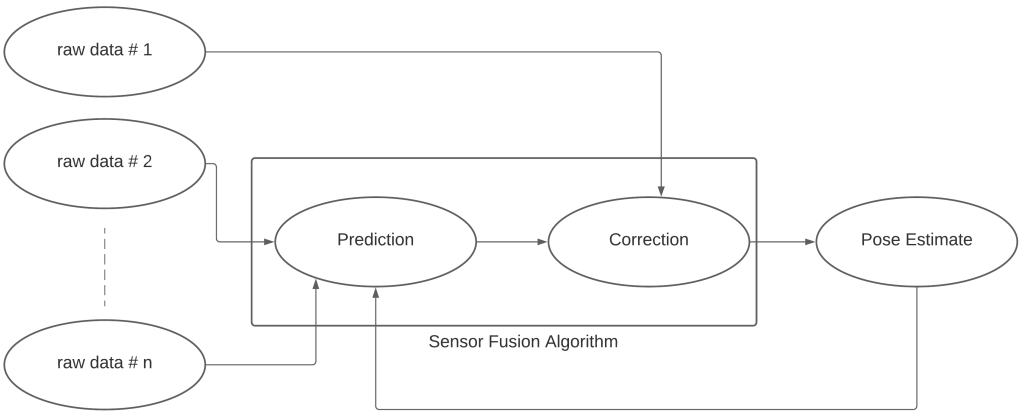


Figure 1.9: Example of sensor fusion algorithm based on the Kalman Filter technology

Among those packages, it will be discussed Robot Localization.
It is a collection of state estimation nodes, each of which is an implementation of a nonlinear state estimator for robots moving in 3D space.
It contains two state estimation nodes, based on Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF). Also, the package provides another node, which aids in the integration of GPS data.
The state estimation nodes use an omnidirectional motion model to project the state forward in time and correct that projected estimate using perceived sensor data.
Among the many perks of this package, it is worth mentioning some important ones:

- unlimited number of input
- complete freedom in the choice of the variable to fuse for a single sensor
- many possible configurations
- advanced parameter tuning

In the drone context, especially if the primary source of the system for what concerns localization is VO, it might be considered the hypothesis to fuse odometry and external IMU data.
Another idea might be the usage of multiple odometry sources, such as the T265 tracking camera and a VO package, or the usage of two odometry data from the T265 camera and, for example, the ZED 2 camera.
This package doesn't put a limit to the testable configurations, anyway the set-up of all the parameters and the choice of the right configuration is not an easy task. Moreover, it must be taken into account, as a rule of thumb, that the basic working principle of Robot Localization takes into account just the least noisy sensor.
To fuse different sensors, with noise values of a different order, might be needed some data manipulation.

1.2.2 SLAM packages

The first step in a project that requires a SLAM algorithm is the choice of the algorithm itself, i.e. the selection of a SLAM package. The goal is to find a package that grants the best trade-off between performances and computational cost. There are a lot of possible reasons that can make the decision easier:

- *sensor interface*: some packages don't allow to interface with certain kinds of sensors or they allow the usage of different sensor suites, but they are best inclined to work with a type of sensor instead of another.

- *external resources*: SLAM packages can run completely on board, or they can run completely on a ground station, or there even exists some mixed solutions.

- *computational effort*: the first thing to understand before choosing a package is if that package will be able to run on the selected device.

- *documentation*: the presence of detailed documentation must not be taken for granted. The wider is the documentation and the number of tutorials, the easier is the installation, the configuration, the usage, and the parameter tuning

- *3D-SLAM support*: not every package is designed to perform 3D-SLAM, therefore, if the project requires it, it is necessary to rule out some packages.

- *releases*: SLAM packages keep being released for over a decade by now and this doesn't seem to stop any soon.
  Therefore, one must pay attention to each release date and the number of releases: the tendency is to rule out the oldest packages, deemed as obsolete, in favor of the newest.
  Anyway, recently released packages present drawbacks as well: the newer is a package, the higher is the chance to encounter bugs and issues of different kinds.
  Moreover, time helps to create a community of users that report bugs, propose fixes and discuss different strategies, with the consequence that it becomes easy to find useful information by browsing online.

Eventually, even ruling out the packages that don't match a project requirements, there are a lot of possible choices for a package.
Possibly, the best choice is to try to install and use different packages and select the best possible fit. Following, a list of the most promising packages analyzed.

- **Cartographer** [4]: Cartographer is a lidar-based system that provides real-time SLAM in 2D and 3D across multiple platforms and sensor configurations, it is used by Google Street View for mapping building interiors.
  It was built for backpack mapping platforms and achieves real-time mapping and loop closure at a 5 cm resolution.
  To achieve real-time loop closure, a branch-and-bound approach has been used for computing scan-to-submap matches as constraints.
  Experimental results and comparisons to other well-known approaches show that, in terms of quality, Cartographer is competitive with established techniques. Among its perks, it should be reported that this package already supports 3D mapping, the sensor suite it has so far used to map 3D environments includes an IMU too, there is large and well-detailed documentation.

The critical problems with Cartographer are related to the fact that it is a lidar-based approach mainly focused on 2D map construction, that it requires a considerable computational effort, and that it cannot receive any information from cameras.
Indeed, it looks clear that, although it is not impossible to use this package for drone applications, it's not quite recommended, especially as a first choice.

- **Gmapping[5]**: GMapping is a highly efficient Rao-Blackwellized particle filter to learn grid maps from lidar data. Rao-Blackwellized approaches associate to every particle an individual map of the environment, accordingly, the main focus of the package is to reduce the number of particles needed to produce the occupancy grid map.
  GMapping achieves this goal using a proposal distribution that considers the accuracy of the robot's sensors to draw particles in a highly accurate manner, and an adaptive resampling technique that maintains a reasonable variety of particles and in this way enables the algorithm to learn an accurate map while reducing the risk of particle depletion.
  GMapping is designed for ground robots. It assumes that the laser is mounted horizontally and at a fixed height, and it makes a 2D map based on those assumptions.
  When the UAV wobbles or changes height while it is moving, the UAV breaks those assumptions. It has nevertheless been used with MAVs before but the sensor suite consisted of lidar and an IMU.
  Moreover, GMapping yields a 2D occupancy grid, not a 3D one, it can however build 2.5D maps. The limited possibilities of configuration, as long as the poor documentation, don't make this package a reasonable solution for drones.

- **Hector[6]**: Hector is a package designed with the idea to get a low-computational cost algorithm capable of combining a 2D SLAM approach based on lidar with a 3D inertial sensor (IMU).
  This package consists of a set of tools mainly used for mapping purposes and IMU managing. Hector is a SLAM approach that finds its main perk in the fact that it can be used without odometry as well as on platforms that exhibit roll/pitch motion (of the sensor, the platform, or both).
  It leverages the high update rate of modern lidar systems and provides 2D pose estimates at the scan rate of the sensors. While the system does not provide explicit loop closing ability, it is sufficiently accurate for many real-world scenarios.
  The system has successfully been used on Unmanned Ground Robots, Unmanned Surface Vehicles, Handheld Mapping Devices, and logged data from quadrotor UAVs.
  At first sight, this package shows some drawbacks. Except for some tutorials,

it does not have detailed documentation and its sensor interface appears to be limited to laser sensors and IMU.

- **MRPT**[**7**]: MRPT project is a set of cross platform C++ libraries and applications that can allow multiple third-party libraries to work together. The main focus of MRPT are Simultaneous Localization and Mapping (SLAM), computer vision, and motion planning algorithms.
  Interesting SLAM algorithms are provided by this library but only one is able to perform 3D SLAM, 3D Kalman Filter-based Range-Bearing SLAM. This means that there is no choice about the SLAM algorithm to implement with MRPT libraries. The main features of this algorithm are:

  1. The initial probability density of a feature is approximated with a particular weighted sum of Gaussians.
  2. This initial state is expressed in the robot frame, and not in the global map frame, so that it is decorrelated from the stochastic map, until it is declared as a landmark and added to the map
  3. Many features can enter the initial estimation process at a low computational cost, and the delay can be used to select the best features

  Regarding sensor interface, most common sensors can be used without any kind of incompatibility problems, but despite the wide hardware interface most of them are dated.
  This package is however quite complex in its usage and it doesn't allow, if necessary, to decouple the SLAM problem from the path planning and the computer vision part. Even if this could be possible, it doesn't look like the best solution.

- **Vins-Fusion**[**8**][**9**]: VINS-Fusion is an optimization-based multi-sensor state estimator, which achieves accurate self-localization for autonomous applications.
  VINS-Fusion is an extension of VINS-Mono, which supports multiple visual-inertial sensor types (mono camera + IMU, stereo cameras + IMU, even stereo cameras only).
  It, if needed, allows GPS integration as well. The approach of the core of this package, i.e. VINS-Mono, is based on the idea of calibrating temporal offset between visual data and IMUs measurements.
  This kind of dynamic calibration has proven to get better results in terms of calibration than offline tools. Indeed, the main characteristic of this package, if compared with other state-of-the-art packages, regards the improved fusion algorithm, which is capable of fusing efficiently local pose information (e.g. IMUs) through the VIO with global information, such as GPS, magnetometer,

barometer. Its computational cost is bearable even for low-budget embedded systems.

- **ORB-SLAM2[10]**: ORB-SLAM2 is a SLAM package for Mono, Stereo, and RGB-D cameras. It can detect loops and relocalize the camera in real-time. It provides a ROS node to process live monocular, stereo, or RGB-D streams. It includes a lot of the features of a complete SLAM package, as the presence of an efficient feature descriptor (ORB), the usage of local BA for estimation, and the possibility of relocalization.
  It requires a medium computational effort, it cannot be used on low-budget and low-memory boards. There are some versions adapted for low-budget systems, but they are not as performing as the complete version.
  The drawbacks lie in the most common visual-based SLAM problem, i.e. the loss of track in environments with low features too fast movements, and the impossibility to integrate IMU data.

- **ORB-SLAM3[11]**: If compared with its previous version, ORB-SLAM3 shows some clear improvements. First of all, it has been integrated a VIO system, which allows integration of IMU data as well.
  Indeed, it has a lot of possible configurations, such as Mono, Stereo, Mono + IMU, stereo + IMU, fisheye. Moreover, an improved place recognition method has been included in the package, by letting the system work properly even in situations with poor visual inputs from the environment.
  From the data collected on the most used datasets, it appears that the package assures accuracy and robustness level off the charts if compared with similar solutions.
  The only drawback that stands out is the fact that an official 1.0 version of the package has yet to be released. There is a beta version, but it might be still too unstable.

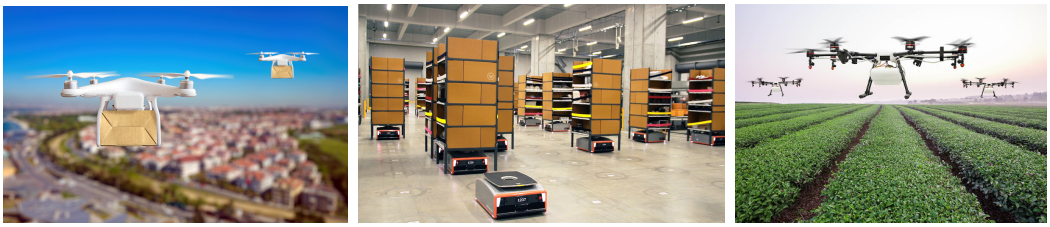
- **S-PTAM[12][13]**. S-PTAM: is a Stereo SLAM system. It separates the time-constrained pose estimation from less pressing matters such as map building and refinement tasks.
  On the other hand, the stereo setting allows the reconstruction of a metric 3D map for each frame of stereo images, improving the accuracy of the mapping process with respect to monocular SLAM.
  Different versions are available, which present slightly different approaches. Anyway, the dated information available, along with the lack of documentation don't suggest this as the primary choice for a SLAM package.

- **RTAB-Map[14]**: RTAB-Map (Real-Time Appearance-Based Mapping) is an RGB-D, Stereo, and Lidar Graph-Based SLAM approach based on an incremental appearance-based loop closure detector.

The loop closure detector uses a bag-of-words approach to determine how likely a new image comes from a previous location or a new location.
When a loop closure hypothesis is accepted, a new constraint is added to the map's graph, then a graph optimizer minimizes the errors in the map.
A memory management approach is used to limit the number of locations used for loop closure detection and graph optimization so that real-time constraints on large-scale environments are always respected.
RTAB-Map can be used alone with a handheld stereo camera, or a 3D lidar for 6DoF mapping, or on a robot equipped with a laser rangefinder for 3DoF mapping.
There are a lot of perks using RTAB-Map, starting from the fact that, even if it is relatively old compared to other famous packages, it has been continuously updated, and its structure is well prepared to be modified as new VO-VIO algorithms are developed.
Moreover, its detailed documentation, along with a discrete number of tutorials, and an active community, makes the usage as easy as possible. Unlike most of the packages presented above, RTAB-Map doesn't perform VO by itself, but relies on external packages to do it: the main node can take as an input an odometry message, that can come out of wheel odometry as well as tracking modules, like T265 and ZED 2 camera ones.
If none of these messages are provided, the package allows performing VO starting from subscribed images. The package presents many possible configurations: it can work with cameras as well as with lidars, the produced map can be visualized and saved in different formats, the parameter tuning enables the user to manage parameters on both the VO and the mapping side.
Finally, its contained computational effort guarantees the possibility to run the package in real-time situations, even on low-budget embedded devices.

1.3 The Multi-agent SLAM problem

Even if the research about SLAM is mainly focused on the development of more and more accurate, robust, and portable packages, recently the implementation of some systems for multi-agent SLAM have been studied and there already exists some implementations.
First of all, it is important to make clear why these kind of systems have been studied, and, to do that, the first thing is to understand the reasons behind the employments of robots' fleets.
The possibility to use a large number of robots, even of different kind, which are able to communicate and accomplish a task together, opens up to a lot of options in a large variety of applications, such as security systems, logistics, delivery, farming, search and rescue, exploration.



(a) Delivery (b) Logistic (c) Farming

Figure 1.10: Applications of robots fleets on different tasks: multiple robots open to the possibility of multi-robot SLAM

There is a certain number of missions that require a system in which a fleet of mobile robots, usually autonomous robots, is required to accomplish tasks in unknown environments. In autonomous applications, SLAM represents a key role in assuring safe and successful navigation.
The idea behind collaborative SLAM consists of taking advantage of the data produced by each robot and use them to get a consistent boost in the performance of each robot of the fleet.
From the mapping point of view, it is possible to merge the maps produces by each agent to get a global map, that can be then forwarded to all the agents. In this way, a mobile robot can acquire environmental information about a certain zone even if it hasn't ever been there. Not only mapping, but localization benefits of multi-agent SLAM too, getting an increased accuracy and, most of all, robustness.
These systems usually adopt a centralized architecture, in which a ground station manages the entire fleet, it is responsible for the data networking and performs the most resource-demanding parts of the SLAM algorithm.

Figure 1.11: Sketch diagram for Multi agent logic. Each agent is able to send and receive data from the ground station, i.e. a notebook

To be more precise, the agents usually compute the local map and perform the essential tasks, kine real-time VO.
Tasks as bundle adjustment and place recognition are usually performed on the ground station, along with map fusion and optimization.
The advantage of the centralized architecture lies in the possibility to perform all of these tasks on a powerful machine, by reducing the computational power needed on the drones.
Simultaneously, it must guarantee data consistency, data access from the agent, and time delays managing, which is quite a challenge, especially for wireless networks. Indeed, the system must guarantee that each drone can, eventually, perform SLAM on its own: if a loss of connection or data inconsistency happens, the single agent must be able to react properly.
To achieve independence from any kind of ground station, some systems tried to implement a decentralized system too. However, the data consistency and the process of sharing information is way more difficult.
Therefore, it does not appear to be the best possible solution yet, especially considering that there's also the problem that a decentralized system requires more powerful on-board computers.

Chapter 2

Package Development

2.1 Project Structure

This section addresses the structure of the proposed solution to the multi-agent SLAM problem. It will be discussed all the premises, preliminary hypothesis, and the choices behind the development of the package.
The system is thought to work in an environment in which the communication among a ground station (server) and the drones (agents) is enabled, with the goal to exchange information about localization and mapping and enhance the robustness of the SLAM algorithm. The information flow is managed by ROS.
The idea on which this system is based is to get a simple, efficient and with low-computational effort package to share information among the server and the agents with a wireless network.
In the following paragraphs, it will be described the hardware configuration, both for what concerns the agents and the server, and the software development, carried on with the creation of a ROS package, starting from well-known libraries.

2.1.1 Hardware set-up

In this section it will be described the hardware set-up and the specification of each member of this system. In Figure 2.1 it is shown the configuration of the hardware components mounted on the agent.

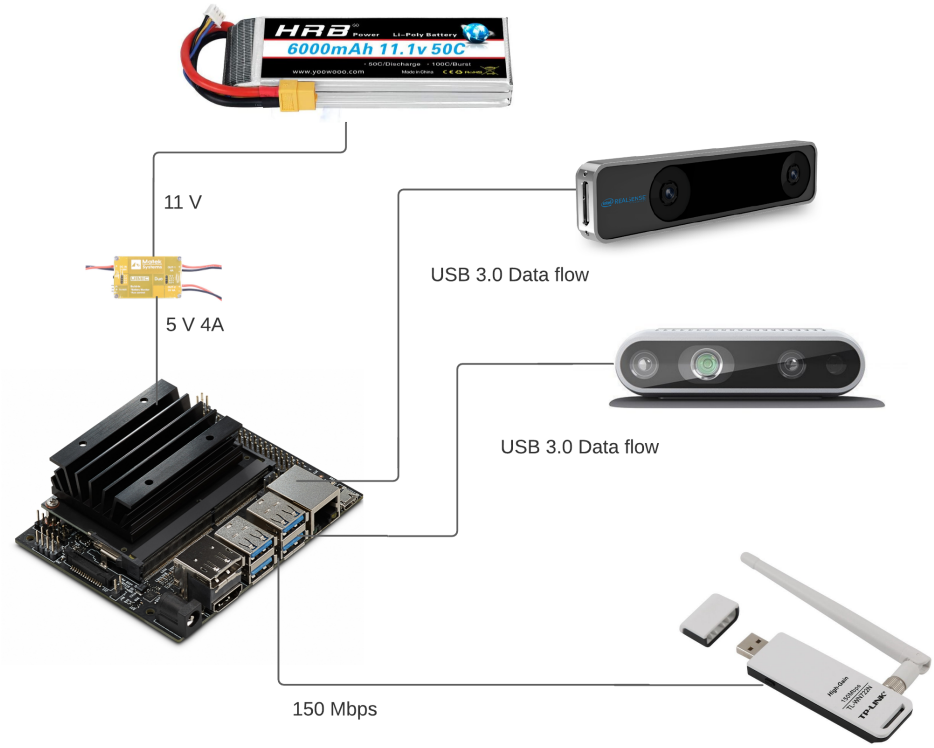


Figure 2.1: Single-agent hardware architecture. It is a battery-powered Jetson Nano, a USB wireless module and the RealSense D435 and T265 cameras

A brief about the devices is reported in Table 2.1:

Device	Connectivity	Tech Specs
Nvidia Jetson Nano	USB,Ethernet	CPU: Quad-core ARM, Memory: 4 GB 64-bit LPDDR4
TP-Link wireless adapter	USB	Wireless 150Mbps, 2.4GHz
D435 Depth Camera	USB 3.0	RGB module Depth module Size: 90 mm x 25 mm x 25 mm
T265 Tracking Camera	USB 3.0	2 x Fisheye module Motion module Size: 108 mm x 24.5 mm x 12.5 mm
HRB LiPo battery	XT60	Capacity : 6000 mAh Output Voltage : 11.1 V
Matek UBEC DUO Dual	T plug	Ouptut: 4 A - 5 V

Table 2.1: Hardware components: brief description

As described above, the core of the system is represented by the Jetson Nano onboard computer. As well as all the Nvidia Jetson products, and some of the most diffused embedded devices on the market, this board is based on the ARM architecture. ARM processors are best suited for a device designed with the idea of guaranteeing high portability, since their low cost, reduced power consumption, and low heating dissipation.
Moreover, the Jetson Nano is way cheaper than the other Jetson models, and its size is comparable to the Jetson Xavier NX, but it is a lot smaller than the Jetson TX2. The boards are shown in Figure 2.2.

(a) Jetson Nano (b) Jetson Xavier NX (c) Jetson TX2

Figure 2.2: Jetson boards: the three main onboard computers produced by Nvidia, organized (from left to right) from least to most expansive and powerful

Indeed, getting high portability inevitably affects the performance. Anyway, the Jetson Nano is one of the best products on the market, especially considering its cost-quality ratio. This is a low-cost platform, which is provided with support for full desktop Linux. Its reduced power consumption, along with the small size, makes it a perfect candidate for drone applications. This device is usually employed in a variety of scenarios: AI, deep learning, and robotics. One of the most interesting features is the presence of NVIDIA CUDA-X, a collection of over 40 acceleration libraries that enable modern computing applications to benefit from NVIDIA's GPU-accelerated computing platform.
Unfortunately, the Jetson Nano doesn't embed a wireless module, therefore in this project has been decided to adopt an external adapter, connected via USB port. It is clear that this is not the best possible choice, especially considering that the cost of higher-performance wireless adapters is only slightly higher than the one employed, and a lot of products on the market take up lesser space. Hence, the final system deployment should include a more adequate wireless adapter.
The power supply is realized with a LiPo battery and a buck regulator.
The sensor suite consists of two Intel cameras, the D435 depth camera and the T265 tracking camera, both connected with a USB cable. Moreover, the T265 camera embeds a VPU, that allows the camera to perform image processing and stereo-visual odometry on-board, by lifting this burden from the Jetson Nano.

The possibility to insert more sensors to improve the localization or the mapping procedure has been considered, but it has been discarded, since the Intel products already add to the system visual and inertial information, and they keep to a low level the total volume occupation and weight of the sensor suite.
These cameras take advantage of the USB 3.0 connection, which guarantees a continuous image flow and it reduces possible time delays.
The devices are mounted together through 3D-printed support, as shown in Figure 2.3, which usage is recommended by Intel since it makes easier not only for a safer and more compact assembly, but because this support helps with the set up of the reference frames on the software side.

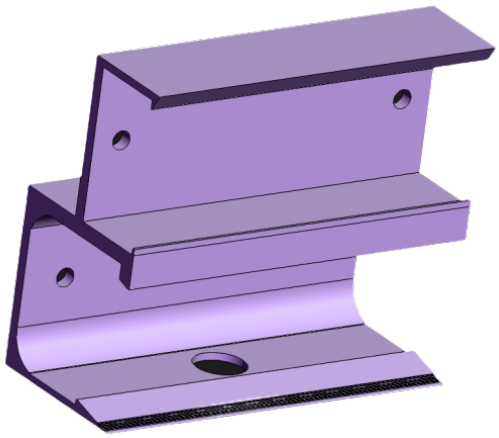


Figure 2.3: T265 + D435 camera 3D-printed support: it helps both in the testing phase and in the real-time usage

This set-up has the advantage of using a low-budget board, with limited hardware capacity, with two advanced sensors that guarantee small volume, high performance, and low power consumption. Anyway, the onboard computer and the sensors are powerful enough to run some SLAM packages completely on board, which means that this can guarantee if needed, independence from any kind of ground station. This system is suited to work well in a multi-agent architecture since it keeps acceptable the overall cost of the fleet. Moreover, the proposed solution doesn't even require a very powerful ground station.
All the experiments that will be shown in the next sections have been realized by using, as a server, a laptop with an Intel Core i7 processor of 8th generation, along with a reduced amount of RAM, 8 GB.
Theoretically, there is a possibility that might make the overall cost even lower, especially if the drone swarm is large. The usage of monocular cameras for SLAM could reduce the cost of the sensor suite by far, other than implying a reduction of

the power consumption and a reduction of volume and weight.
All these advantages might suggest that the monocular camera is an even more portable solution than the one proposed in this project, but it has its drawbacks, which not only regards the well-known differences between monocular and stereo vision.
The usage of the T265 camera is not just related to its accuracy, especially because in some contexts it might be less precise than other open-source VIO packages.
As shown in [15] and [16], when it's a matter of which VIO algorithm to choose, there isn't an unique solution.
Some algorithms guarantee better performances in terms of accuracy, but they may imply high computational demands to ensure real-time.
Indeed, the accuracy also depends strongly from the kind of movement that the robot should be able to do: there are algorithms that work well with slow movements, and other that provide better performances with fast ones.
All of these assumptions suggest that a system with a monocular camera and an IMU, with a VO algorithm, for example Vins-Mono, might guarantee better performances.
The issue of this solution consists in the fact that all the effort weights on the onboard computer. A low-budget device such as the Jetson Nano is not powerful enough to handle it. The T265, on the contrary, allows decoupling the workload, by taking care of VO. The result might be as not as precise, but it is cheaper and it allows to use of the Jetson Nano, being less power demeaning.

2.1.2 Software set-up

This section addresses the preliminary work that led to software development.
There will be a focus on the set-up of the working environment, the ROS installation, some preliminary tests, packages installation, and generation of a new package.
The main idea behind this project is to use ROS as a global framework for the multi-agent system, in the sense that a ROS master will run on the server and each agent will be able to communicate with it.
Indeed, some other issues need to be taken care of, such as the choice of the SLAM package for the single agent.
Since the objective of this project is not the development of a SLAM algorithm for a single mobile robot, there' no need to explore the hypothesis to create a new package from scratch.
The advantage of this choice is that the system development could be adapted to more than one SLAM package, other than the fact that the creation of a new package is very complex and time-demanding.
The disadvantage of this idea is that the SLAM package is not easily manageable and open to be edited, when necessary, but it can be considered as a black box: it

is possible to modify its input and its outputs, but not the package itself.
To be precise, editing a package is a possibility, especially since many of them are released with an open-source license.
However, the structure of these packages is very nested, with a high level of complexity
Therefore, modifying a code is not easy and it might have dangerous consequences on the entire package.
Anyway, it is necessary to specify that these changes, intended as code editing, are referred to as changes in the logic of the algorithm: the parameter tuning usually it is possible and it allows to modify of a lot of parameters of a package, but the main logic remains the same.
Other than the choice of the package, it is important to understand how to generate a new package, to be able to get an interface between the multi-agent logic and the single agent SLAM package.
Moreover, the generation of a package allows the development of a nodes architecture, which is helpful for what concerns possible bugs, errors, or system failures.
This structure is also useful in this problem since if, for example, one of the nodes responsible for the multi-agent logic stops working, the nodes responsible for the SLAM on each agent can keep running, by avoiding further problems.
To start analyzing these problems it is necessary to start by installing ROS on the server and the agents to create an environment on which to develop the algorithm.
First of all, it is necessary to pick a ROS distribution to be installed.
No limitation is forced, since all the releases have an open-source license.
ROS allows to install anyone of the available versions. Some of the distributions are older and provided with long-term support, which makes them more stable, while others are newer and provided with shorter support, but they can work with recent platforms and more recent versions of the ROS packages.
It is obvious that the oldest versions have been ruled out, both because of compatibility reasons and because they are no longer supported by ROS developers.
Indeed, the ROS community for these versions is no longer existent, since it is focused on the newest releases.
It has been also considered the hypothesis to use a release of ROS2, but this hypothesis has been ruled out as well, exclusively for compatibility reasons.
The upgrade of the system to ROS2 is a concrete possibily for the future.
Actually, ROS recommends one of the following distribution:

- **ROS Kinetic Kame**, released in 2016, supported until 2021
- **ROS Melodic Morenia**, released in 2018, supported until 2023
- **ROS Noetic Ninjemys**, released in 2020, supported until 2025

(a) ROS Kinetic Kame (b) ROS Melodic Morenia (c) ROS Noetic Ninjemys

Figure 2.4: ROS distributions: ROS recommend the installation of one of these three distribution

Among these releases, the choice has fell on ROS Melodic Morenia (ROS Melodic), since ROS Noetic is too recent and it does not support all the packages, which doesn't make it a stable release.
ROS Kinetic has been ruled out as well, since, even if it's the more stable release, it's older then Melodic, and the long term support for this version will end very soon. To be able to install ROS both on the server and on the Jeton Nano, Ubuntu has been installed.
For compatibility reasons, the Linux distribution **Ubuntu 18.04.5 LTS (Bionic Beaver)** has been chosen.
ROS makes available different kinds of installations, based on the needs of the user.

- **Desktop-Full Install**: ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators and 2D/3D perception
- **Desktop Install**: ROS, rqt, rviz, and robot-generic libraries
- **ROS-Base**: ROS package, build, and communication libraries. No GUI tools.

For development purposes, the Desktop-Full version has been installed on both the onboard computer and the server. In the final deployment, it might be sufficient to install the Desktop version.
Once ROS is installed, it is necessary to create a workspace. To do that, it is used the **catkin** package. Catkin is the build system of ROS.
A build system is responsible for generating 'targets' from raw source code that can be used by an end-user.
These targets may be in the form of libraries, executable programs, scripts, interfaces (e.g. C++ header files). In ROS terminology, source code is organized into

'packages' where each package typically consists of one or more targets when built. To build targets, the build system needs information such as the locations of toolchain components, source code locations, code dependencies, external dependencies, where those dependencies are located, which targets should be built, where targets should be built, and where they should be installed.
This is expressed in a configuration file read by the build system. With CMake, it is specified in a file typically called **CMakeLists.txt**. The build system utilizes this information to process and build source code in the appropriate order to generate targets.
ROS utilizes a custom build system, catkin, that extends CMake to manage dependencies between packages.
Catkin combines CMake macros and Python scripts. This package is designed with the idea to allow better distribution of packages, better cross-compiling support, and better portability. It allows building multiple, dependent projects at the same time.
The catkin package is used to create a catkin workspace. A catkin workspace is a folder where it is possible to modify, build, and install catkin packages. It can contain up to four different spaces which each serve a different role in the software development process.

- **Source space**: it contains the source code of catkin packages. Each folder within the source space contains one or more catkin packages.
  This space should remain unchanged by configuring, building, or installing. The root of the source space contains a symbolic link to the CMakeLists.txt file. This file is invoked by CMake during the configuration of the catkin projects in the workspace.
- **Build space**: it is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here.
- **Development space**: it is where built targets are placed before being installed. This provides a useful testing and development environment which does not require invoking the installation step.
- **Install space**: it is the space in which built targets are installed.

Once ROS is installed and the catkin workspace has been created, the user can create, edit, build its own packages, and/or clone and install packages from the ROS library or source git.
Among the necessary packages for SLAM, it is important to install the SLAM packages, along with their dependencies, and the packages that allow handling

sensors. **realsense-ros** belongs to the latter category: it is the package that allows communication with RealSense cameras.
Properly, realsense-ros is a ROS Wrapper, which contains all the needed nodes and nodelets to get camera data in the ROS format: messages.
The installation, other than the package, will install all the needed dependencies, included the RealSense SDK: **librealsense**.
The SDK allows depth and color streaming and provides intrinsic and extrinsic calibration information.
The library also offers synthetic streams (point cloud, depth aligned to color, and vice-versa), and built-in support for record and playback of streaming sessions.
The SDK also includes a viewer, a screenshot of which is shown in Figure 2.5, that allows to view depth stream, visualize pointclouds, configure some settings, post-processing, and more.

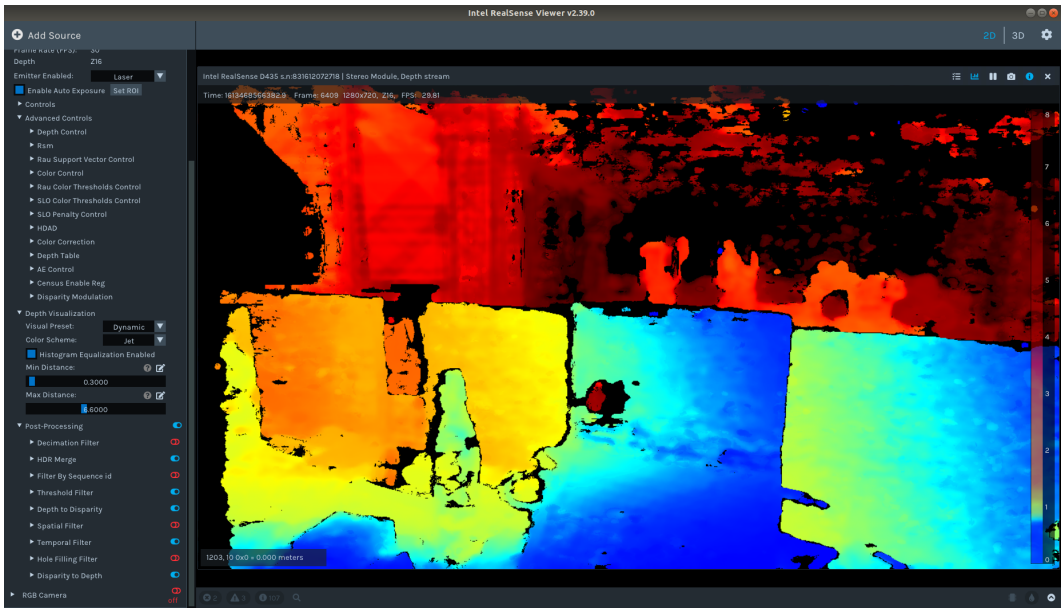


Figure 2.5: RealSense Viewer: one of the SDK tools, useful for visualize, configure and post-processing data. Here it's shown a depth image acquired with D435 camera

Moreover, librealsense includes some debug tools, along with tools that can test the quality of the depth image. Finally, the SDK is provided with some other code examples, which are thought to use some advanced features of the cameras or to simply use multiple cameras together.
Once the ROS Wrapper is downloaded and installed, it will be possible to access a folder that contains a series of *.launch* files. Those files are the method used to start the camera node.
When the node is launched, the node starts the cameras and enables the data

stream. From that moment on, it is possible to see a list of all the published topics, which depends on the setting of the parameter that has been specified in the **.launch** file. Usually, for what concerns the D435 camera, it can be present a depth, infrared, and/or color stream, along with a point cloud stream, while the starting of the T265 camera enables the stream of fisheye, odometry, and IMU topics. Figure 2.6

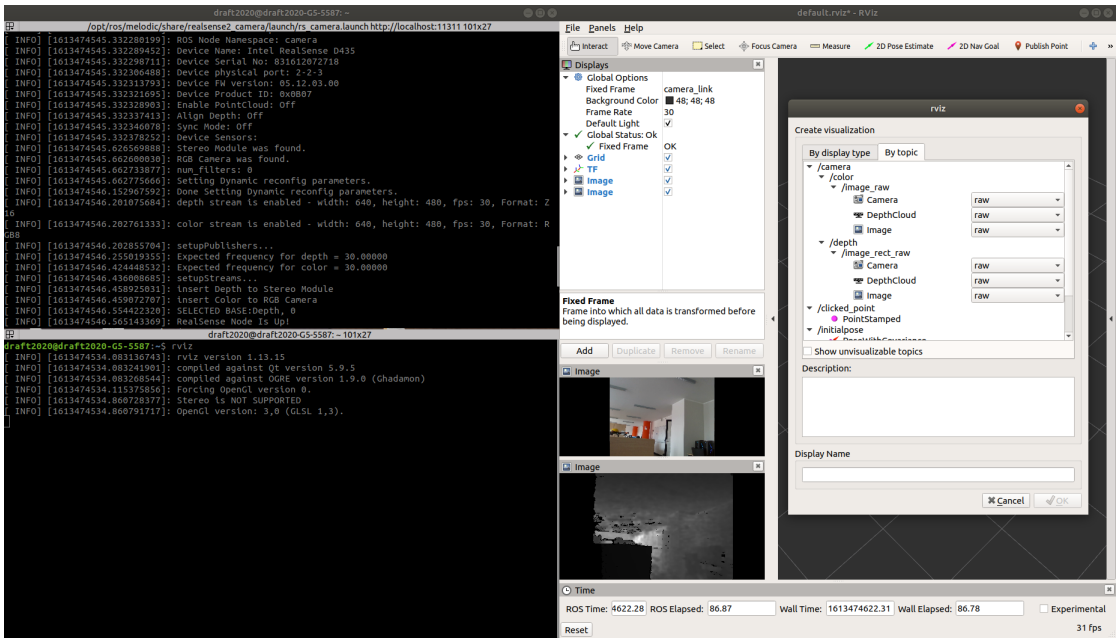


Figure 2.6: D435 camera launched from a terminal. On the right Rviz, visualization tool. Here it's used to show color and depth image, and the topic selection

If the user wants to launch both cameras simultaneously, it can, by using a unique *.launch* file. The Wrapper will generate in every case just one node, that handles everything.
Starting the camera in this way allows to calibrate a lot of parameters, such as the FPS, or it allows to enable/disable certain streams, or it allows to manage some post-processing tools and filters. More detail on this will be discussed later.
There is another information that the cameras provide, and it is stored in the **tf** [17]. The tf library is one of the most important features of ROS: it can manage and keep track of coordinate frames. Thanks to this library, it is possible to transform any kind of data among different reference frames. It aims to simplify the managing of coordinate frames since the manual application of a transformation to certain data can easily be affected by errors.

Figure 2.7 shows an example of tf tree generated by the system when the T265 camera and D435 camera are running.

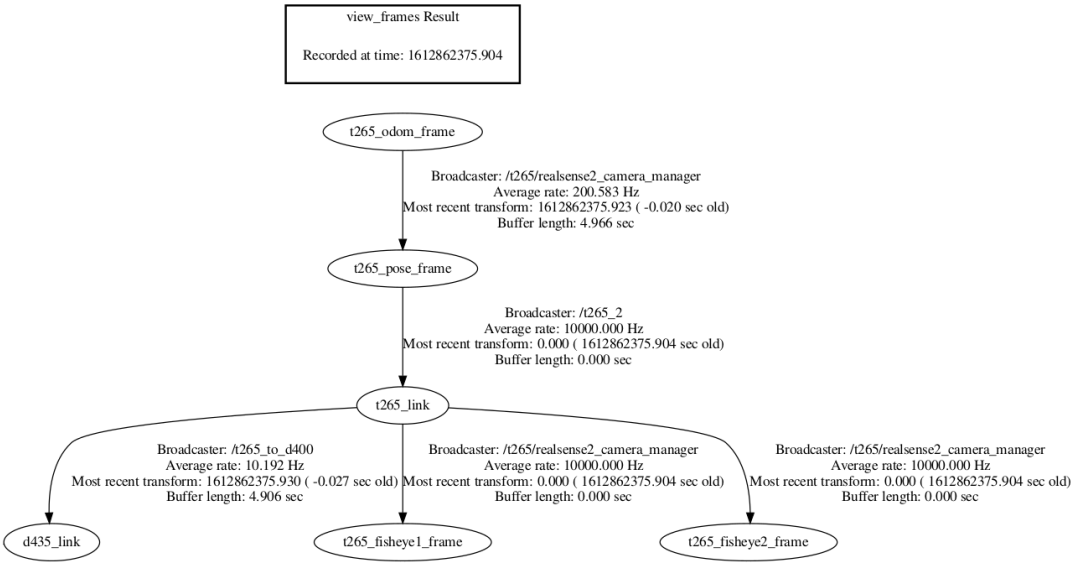


Figure 2.7: Example of tf tree: D435 camera and T265 camera system broadcasts these coordinate frames, that are both linked through static and dynamic transforms.

The tf package is responsible to keep track during time of all the involved frames over time, both for dynamic and static frames. All the relationships among the coordinate frames are then represented with a tree structure, commonly called *tf tree*.
However, the tf package doesn't work exclusively behind the scenes: the user can interact with the package, and there are mainly two ways to do this: listening to transforms and broadcasting transforms-

- tf listener: Receive and buffer all the coordinate frames in the system, and it allows to access any specific existing transform between two frames.

- tf broadcaster: any part of the system can create a broadcaster, which is a structure that provides information about a certain frame with respect to the rest of the frames in the system.

Once the cameras have been installed, and the user gets a good comprehension of the tf associated with images and odometry streams, it is possible to perform a testing phase.

2.1.3 Preliminary tests: Localization

Some tests have been carried on two sides: pure localization and SLAM.
The first test that has been carried on has the objective to understand the accuracy of the localization performed by the T265 camera and the accuracy of the output of the sensor fusion package **robot localization**.
In order to perform a comparison, some experiments have been carried on by considering two different sources of odometry: the one produced by the T265 camera and the one produced by the ZED 2 camera. To use the ZED 2 camera, the ZED Wrapper has been installed. The camera is turned on by a .launch file.
The idea is to understand if the combination of these two sources of information with an EKF can generate a filtered output that is more accurate than the single ones.
There's no particular intention of using both these cameras in a real-time application: the experiment aims to try to understand if it's worth combining two or more pose data.
Two possible outcomes might suggest adopting such a strategy: if the fused odometry is more accurate than the input data, or if it is possible to fuse a principal source of odometry with a secondary source that works as a backup localization source.
The tests that will now be described are performed with a Jetson Nano, placed on a wheeled cart. The ZED 2 camera and the T265 camera are placed on the cart, on their respective tripod. The system is battery-powered.
The hardware set-up is shown in Figure 2.9 and Figure 2.8.

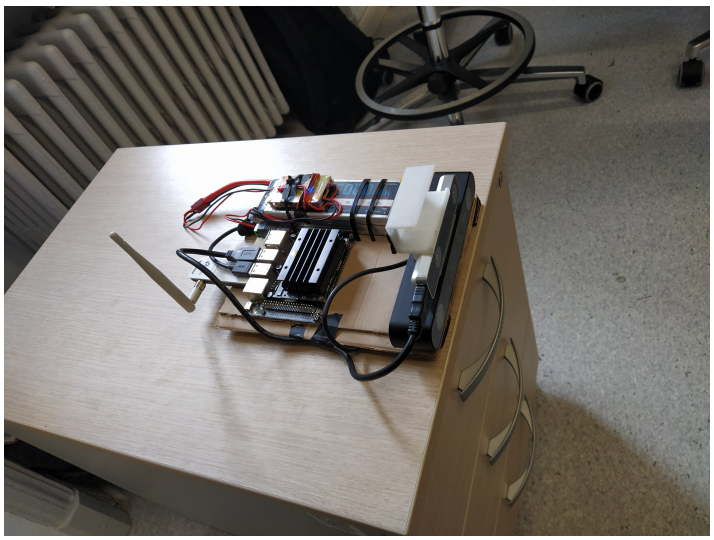


Figure 2.8: Test hardware set-up: the suite is placed on the moving cart and it is battery powered

Figure 2.9: Test hardware set-up: T265 camera and ZED 2 camera are connected via USB to the Jeston Nano board

(a) Sector a-b-i (b) Sector b-c-d (c) Sector h

Figure 2.10: Test course: a scotch tape has been used on the floor for designing the course

The tests cover a course illustrated in Figure 2.11. In Table 2.2 are specified the dimensions of the course.

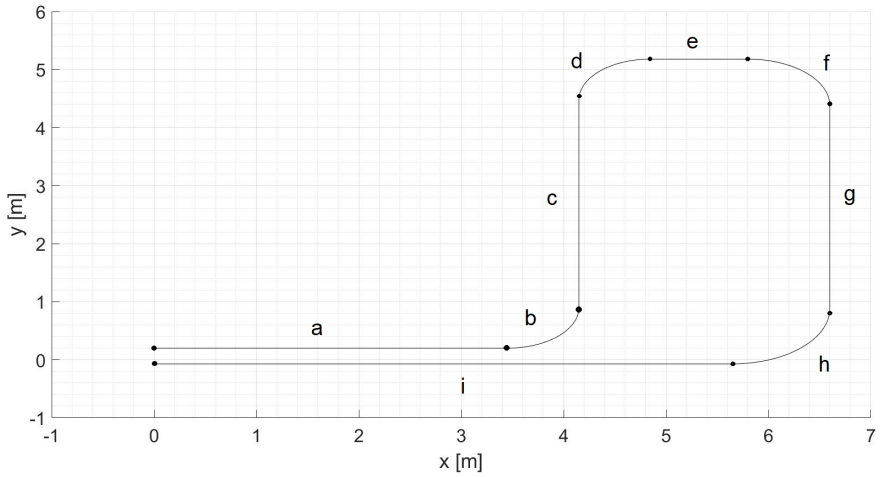


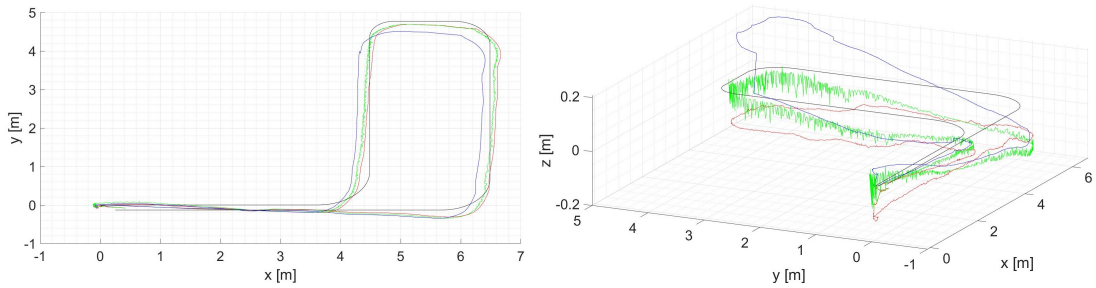
Figure 2.11: Test course: the model has been measured and designed on Matlab, that makes the data comparison easier

Section Name	Measure Type	value (m)
a	straight	3.45
b	radius	0.7
c	straight	3.58
d	radius	0.7
e	straight	0.95
f	radius	0.8
g	straight	3.5
h	radius	0.95
i	straight	5.65

Table 2.2: Tests course: definition of the length of the course

Once the course has been established and measured, and the hardware has been set-up, tests have been performed.
The evaluation criteria of these tests are especially related to the z-axis value: since the platform is flat, as well as the floor, the expected outcome is that the value should be approximately zero.
Any difference from zero can be considered a measurement error. Different tests have been performed, since, among them, changes have been made to the configuration parameters of the filter.

- **test 1**: it consists of a single lap on the course. The filter parameters are left as default. Results are shown in Figure 2.12

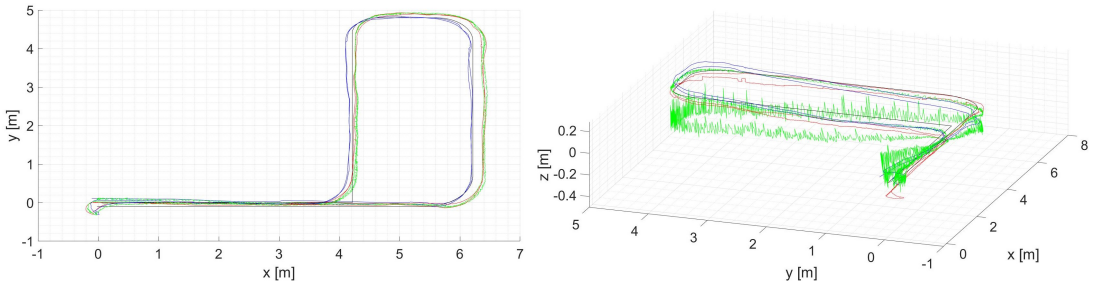


(a) X-Y plane (b) 3D representation

Figure 2.12: Test 1: single lap. The reference course is represented in black. The T265 data is red. The ZED 2 data is blue. The EKF output is green.

The first test has shown that the EKF works well in giving a solution that is halfway between the two odometry inputs.
The ZED 2 camera kept a very good behavior at the beginning, but it has shown an unexpected discontinuity that caused a change in the position value along the z-axis of about 15 cm.
The camera managed to adjust the issue, and the results can be seen after some time, but the large instantaneous variation caused a considerable drift.
The T265 camera shows a more oscillatory behavior over time, but, after an initial drift of about 10 cm, it tracked the position quite well.
No significant discontinuities appeared. The EKF output in this case appears to give out better performances, but it has shown an oscillatory behavior.

- **test 2**: it is basically the same as the previous test, with the exception that this time the cart has done two laps around the course.
  This test helps to understand the effect of IMU measurements and VO data on the overall localization value.
  The motion module with whom these cameras are supported is thought to compensate for the issues of the two above described measurements.
  When the VO faces an environment that is poor of key-points, the IMU helps not losing track. When a task requires a lot of time, the VO helps to compensate for the long time drifts.
  Then, if the camera goes through a place in which it has already been before, it should be able to use re-localization tools to adjust its pose. The results of this test are shown in Figure 2.13

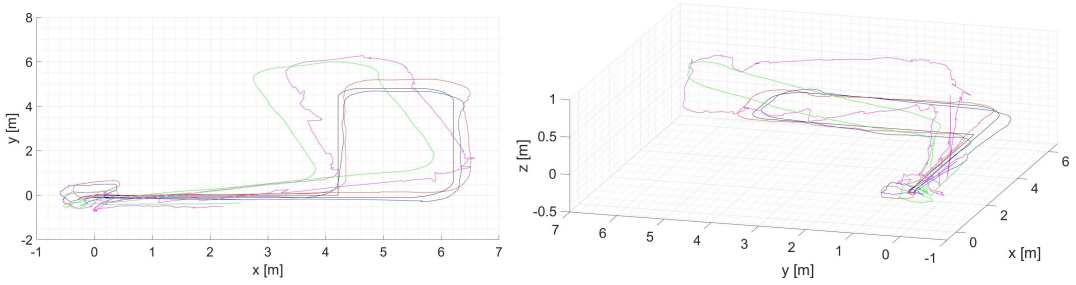


(a) X-Y plane (b) 3D representation

Figure 2.13: Test 2: two laps. The reference course is represented in black. The T265 data is red. The ZED 2 data is blue. The EKF output is green.

This test has shown that cameras behave quite well even after a long time. The track on the X-Y plane is very good, while the error on the z-axis is bounded between -40 cm and 40 cm.
In this case, the EKF provided a good result in the first lap, but a bad one, if compared with the cameras ones, in the second lap.
Moreover, it keeps showing a lot of oscillations.

- **test 3**: it consists of two laps, and this time two filters, both belonging to robot localization, have been launched: the EKF and the UKF. The results of this test are shown in Figure 3.2



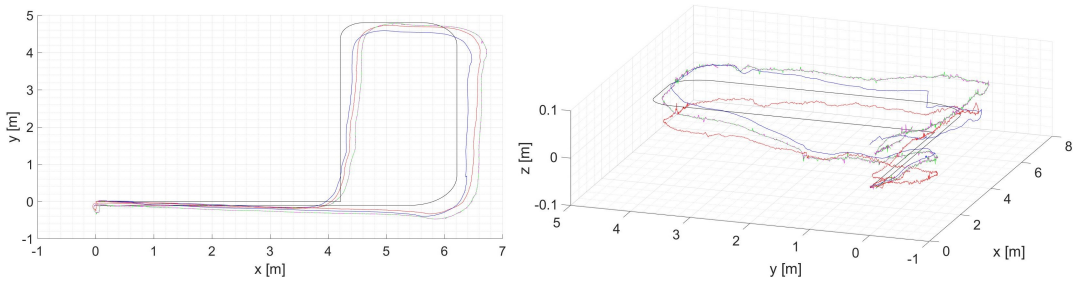
(a) X-Y plane (b) 3D representation

Figure 2.14: Test 3: two laps and differential parameter true for both inputs. The reference course is represented in black. The T265 data is red. The ZED 2 data is blue. The EKF output is green. The UKF output is purple.

Another detail differentiates this test from the previous ones: the usage of the *differential parameter*. If this parameter is set to true in the *.launch* file, then pose information are integrated differentially.

Considering the acceleration measurement at time *t*, it is subtracted the measurement at the time *t-1*, and then this quantity is converted to a velocity.
This parameter should be useful if the system can provide more than one source of the absolute pose.
When more than one source is considered, the filter output may have oscillations due to sync problems among the inputs.
By integrating one, two, or more differentially, this scenario might be avoided.
However, this test has shown that setting to true the differential parameter of both odometry inputs, both filters produce outputs that are way worse than the input data.
Therefore, the option of using two differential sources of data has not been successful. On the bright side, it has been verified that the differential configuration reduces to a minimum value the oscillation on both filter outputs.

- **test 4**: it consists of a single lap on the course, and the differential parameter has been applied only to the odometry input produced by the T265 camera.The results of this test are shown in Figure 2.15



(a) X-Y plane (b) 3D representation

Figure 2.15: Test 4: one lap and differential parameter true for T265. The reference course is represented in black. The T265 data is red. The ZED 2 data is blue. The EKF output is green. The UKF output is purple.

The results of this test are better than the previous ones.
The data in the X-Y plane are quite good, even if they're not as good as test 2 data. The oscillation on the output of both filters has been reduced a lot, if compared to the tests executed without the differential parameter.
The error along the z axis is restrained to a value smaller than 10 cm. Both EKF and UKF outputs are good enough for the first part of the course: they tend to follow the ZED 2 odometry more than the T265 one, but the global result is really good. However, during the second part of the course the error is higher than the cameras ones.

- **test 5-6**: these two tests are wrapped together in order to help visualize the differences between the case in which the differential is set to true on the T265 odometry input or on the ZED 2 odometry input.
  There's one more detail that is added with respect to the previous tests, i.e. the choice of the parameters to fuse: by definition, the odometry message's structure is the one represented in Figure 2.16.

```
Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

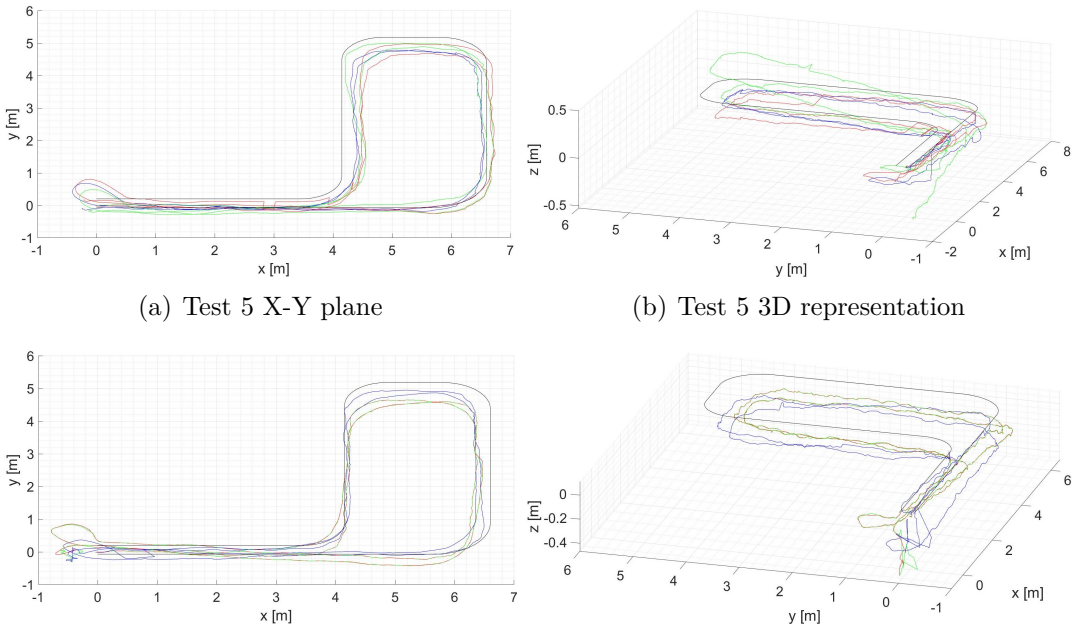
Figure 2.16: ROS documentation about odom message. It shows all the fields, included each field data format.

Robot localization doesn't dictate a precise rule on what kind of data should be fused. There are some rules of thumb described on the documentation, but it must be kept in mind that those rules are thought in the case that odometry is provided by wheels' encoders. Anyway, robot localization fuses data as the boolean table defined in the *.launch* file dictates: Table 2.3 is an example.

Variable type	x-axis	y-axis	z-axis
linear position	true	true	true
angular position	true	true	true
linear velocity	false	false	false
angular velocity	false	false	false
linear acceleration	false	false	false

Table 2.3: Odometry param setting : boolean table that specifies what is fused with the filter

The two tests are illustrated in Figure 2.17. The boolean table has been set in order to fuse exclusively position and orientation.



(a) Test 5 X-Y plane

(b) Test 5 3D representation

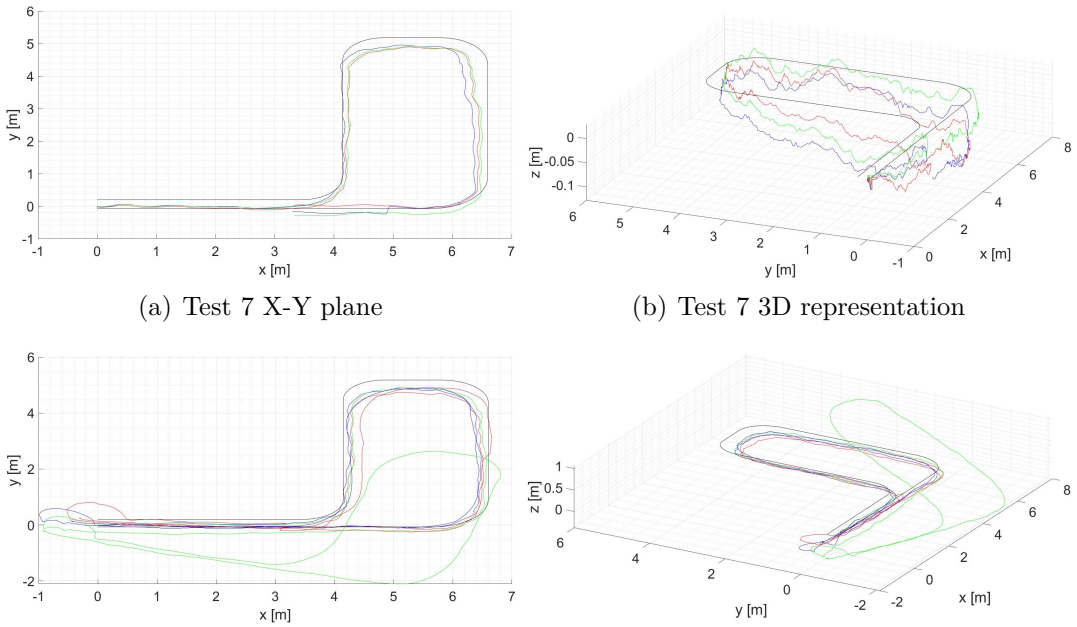
(c) Test 6 X-Y plane

(d) Test 6 3D representation

Figure 2.17: Test 5-6: two laps and differential parameter true for T265 (test 5) and ZED 2 camera (test 6). The reference course is represented in black. The T265 data is red. The ZED 2 data is blue. The EKF output is green.

The two tests have shown some already known problems: the EKF output, after one lap, gets a lot less accurate.
Moreover, in test 5 the filter output does no improve the localization process. The same thing can be said about test 6.
The differential set to true for ZED 2 odometry inputs makes the output of the EKF follow almost exactly the T265 odometry input for almost the entire time. Therefore, fusing position and orientation doesn't improve the accuracy.

- **test 7-8**: these two tests are performed with the differential parameter set to true only for the T265 odometry input. The only difference between the two tests consists in the fact that test 7 fuses both positions and velocities, while test 8 fuses both positions and only linear velocity.
  The results are shown in Figure 2.18



(a) Test 7 X-Y plane

(b) Test 7 3D representation

(c) Test 8 plane

(d) Test 8 3D representation

Figure 2.18: Test 7-8: one lap and differential parameter true for T265. Two boolean table configurations. The reference course is represented in black. The T265 data is red. The ZED 2 data is blue. The EKF output is green.

These two tests have shown two undesired behaviors. Test 7 is good, the average error on the z-axis is low, the X-Y plane shows a good track, but the filter doesn't improve the localization. Test 8 showed a significant error along each axis. The filter output might be correct, but it is rotated with respect to the real one, which means something occurred that messes up the EKF.

These preliminary tests have shown that robot localization is not completely necessary. It requires accurate calibration of all the parameters, and finding the right combination of these parameters is really complex and even if found, nothing assures that the results will be good enough.
In most tests, the filter output has shown a behavior that was even worse than the inputs. Indeed, some tests have shown good results, but they all coincide with oscillations on the output that does not help to improve the localization.
Another thing that emerges from the tests is that robot localization gives much more importance to less noisy data. This means that if the process noise covariance matrix of two inputs is different by two orders of magnitude, the filter will neglect the worst one.
This behavior is theoretically correct, but the implication is that if the package can't combine two accurate data as the ones taken into account here, there is very little chance that the fusion of a less precise sensor, like an IMU, could improve the localization of a video tracking device.
For what concerns the risks of failure, it looks obvious that there are methods that are less computational demanding to provide a safety device.

2.1.4 Preliminary tests: SLAM packages

This section will present some of the preliminary tests that have been carried on in order to choose a SLAM package to be used for the single drone SLAM.
As addressed in Chapter 1, there are a lot of available packages online that could be suited for the single-agent SLAM.
Anyway, some packages, like Cartographer, Hector, Gmapping, and MRPT have been ruled out, for sensor incompatibility or software issues.
Based on the sensor suite, it is reasonable to choose a package that allows taking the most possible advantage from it.
With these premises, it is obvious that the choice should have fallen on a V-SLAM approach. Even the T265 tracking camera has been listed among the available components, other strategies have been tested, in order to understand which is the better solution.
The idea of this approach is to understand if better performances are achieved when the SLAM problem is decoupled between the T265 camera, for the VO, and a SLAM package, or when the SLAM problem is dealt with only the package.
In the second case, the usage of two cameras is redundant.
The selection has brought to consider three packages: ORB-SLAM2, Vins-Fusion and RTAB-Map. ORB-SLAM3 has not been considered because it has been released since not long ago, and therefore a stable release is not available yet.
The tests that will now be described have been carried on with monocular and

stereo cameras. The D435 provided the stereo image, while the Raspberry Pi Camera Module V2 has been used for the tests with monocular images.
This camera can be easily configured to be used in ROS applications, it's cheap, and its volume is really negligible in most drones.
The connection to the Jetson Nano is realized by means of a ribbon cable. The sensor is showed in Figure 2.19.

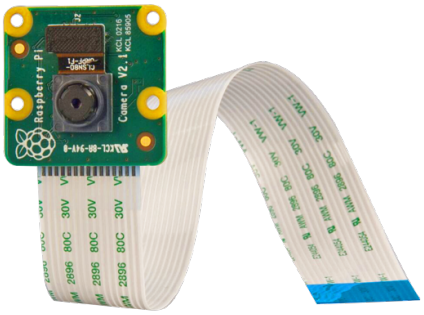


Figure 2.19: Raspberry Pi Camera Module V2. Monocular camera compatible with most of the onboard computers available on the market.

No IMU has been introduced; in the case of ORB-SLAM2 this happened because it is not possible to subscribe to an IMU topic.
The first test concerned ORB-SLAM2. To use this package it is necessary to install it correctly. This might imply more than a few issues, since ORB-SLAM2 is strongly related on a lot of dependencies, that should be manually installed before the package installation.

- **Pangolin**: it is a development library to manage display OpenGL display and abstracting video input. It provides a series of visualization tools that are needed to execute the package and visualize the video output of ORB-SLAM2. It is used because it requires a minimum amount of computational effort to be used, especially considering that it allows video interactions and 3D representations.

- **OpenCV**: it is an open-source library which provides all sort of tools in the fields of computer vision and machine learning.
  OpenCV includes a large set, over 2500, algorithm, divided between classic perception and computer vision algorithms and state-of-the-art computer vision and machine learning algorithms.
  The ductility of this library makes it one of the most used libraries among the ones available online. As a matter of fact, OpenCV can be employed in tasks like image recognition, object identification, features classification in videos, tracking of moving objects, 3D models extrapolation, point cloud production,

tracking of camera motion, etc. ORB-SLAM2 uses this library to manipulate images and features.

- **Eigen**: it is a library that provides all sorts of functions for linear algebra, matrix and vector operations, and geometrical transformations.

- **DBoW2**: it is an open-source library that is used to index and convert images into a bag-of-word representation. It is the library responsible for the creation of a visual vocabulary. It can work with any kind of feature descriptor, included ORB, the ones used by ORB-SLAM2.

- **g20**: it is an open-source general graph optimization framework. It is used to optimize graph-based nonlinear error functions. It has been specifically created for SLAM and BA applications.

Two installations have been tried. The first one on a notebook, the second one on the Jetson Nano. In the first case, there were few issues in the installation, especially related to the OpenCV version, but it has been easy to overcome the problem.
The package installation has been done by means of the official repository; the standalone version of ORB-SLAM2 works perfectly, while the ROS version causes many problems.
Anyway, it is possible to overcome these problems and to start the node without any issue.
For the sake of simplicity, instead of using the monocular camera, to test ORB-SLAM2 on the notebook, it has been used the color camera of the D435 Depth camera.
This choice is convenient for two reasons: the USB cable of RealSense cameras simplifies the connection, and the camera intrinsic and extrinsic parameters are not only already provided, but also included in a ROS topic.
When the ORB-SLAM2 node starts, it loads a configuration file inside which there should be inserted the camera parameter.
Cameras usually introduce distortion to images. Usually, these are taken care of inside more advanced cameras, therefore, among the data streamed, there are the distortion parameters and information about camera geometry.
The two most important kinds of distortions are :

- *radial distortion*: it causes straight lines to appear curved. It causes small distortions at the center of the image, and larger ones far from the center.

- *tangential distortion*: it appears when the image plane is not perfectly aligned to the lens.

Five parameters are used to take into account distortions. In addition to these five, it is necessary to find the camera matrix, which includes information about focal length and optical centers. OpenCV provides a code to calibrate the camera.
Once the parameters are found, they can be added to a *.yaml* file that ORB-SLAM2 loads at the start.
Figure 2.20 shows an example of calibration pattern on a chessboard.
The process has been done identically with the Raspberry Pi Camera Module V2 and the D435 color camera: a series of images of a chessboard held in different position have been acquired and then the calibration tool showed the pattern and printed the calibration parametes.

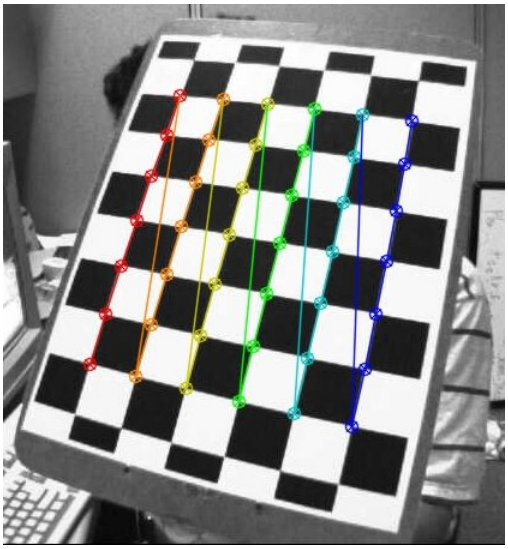
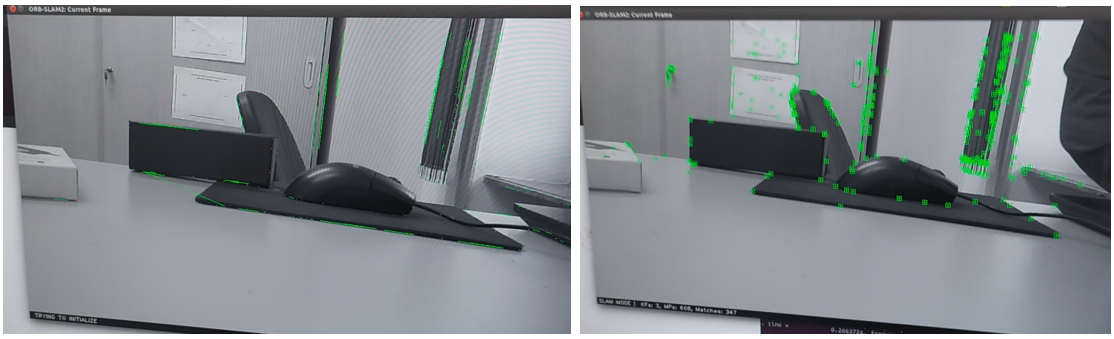


Figure 2.20: Raspberry Pi Camera Module V2 calibration pattern. The method consists in taking snapshots of a chessboard printed on a sheet. The tool provide the pattern shown in figure.

This will allow obtaining a calibrated image. For what concerns the installation on Jetson Nano, the procedure is more complicated. By flashing Jetson Nano, a series of libraries are automatically added to the system, included a version of OpenCV. The installed version causes compatibility problems with ORB-SLAM2, therefore it is needed to substitute it with another version.
Once this is done, and all the other dependencies of the package are installed, ORB-SLAM2 can be installed too.
Anyway, the official repository causes a lot of problems with Jetson Nano and it ended up not working, a lot of times.
Finally, the installation has been successful, but the package is very resource-demanding, so the board cannot guarantee data consistency and real-time SLAM.

A possible solution is found by installing a version of the package that takes advantage of the GPU enhancement available on Nvidia boards.
The version is called ORB-SLAM2-CUDA, and it causes less problem with the installation, but the results are not as powerful as the ones obtained with the original version.
Some tests were performed on the same course shown in Figure 2.11. Two types of tests were attempted. The first one is done with the hardware positioned on a wheeled cart, while the second one is done with the platform held by hand.
The tests have shown some common behaviors. The first thing to notice is that the package requires some time to be initialized.
Until the package doesn't find enough features in the image, it is not able to start. Therefore, an environment with a reduced number of features could imply that the SLAM algorithm doesn't start at all.
The same thing happened if the initial movement of the device is too fast: since there is no stereo camera that can provide depth information, ORB-SLAM2 performs a triangulation among the features detected in two consecutive frames and, once it founds a bigger enough number of key-frames, it can initialize.
As a consequence, by keeping the camera perfectly still, there's no variation in the frames and it is not possible to initialize.
Figure 2.21 shows the initialization phase of the package.



(a) Pre-initialization (b) Initialized

Figure 2.21: ORB-SLAM2:initialization phase. In the beginning, the software looks for features in the environment, once the number is high enough the algorithm starts,

Anyway, this is not the only problem. The algorithm suffers a lot of fast turns: it works better with slow rotations or with a rotation of a small angle.
This might be a problem in the drone context. Moreover, the main issue is the absence of a supplementary localization device.
If the camera ends up in a place with a low number of features, or if for any

reason, the key-frame matching doesn't work, the package displays a "Tracking Lost" message. Even if it is usually able to recover, the global localization and mapping lose data, accuracy and in some cases, no more points are added and the plotted trajectory stops at the moment in which the track is lost.
Moreover, by comparing the position values with the T265 tracking camera ones, it is clear even with just a qualitative analysis that the T265 camera provides better performances.
For what concerns the mapping side of the algorithm, it appears that the number of points inserted in the map are really low and if a scene is filmed from a different point of view, the re-localization process doesn't work well enough.
Figure 2.22 shows the visualization tool of ORB-SLAM 2 during a test.

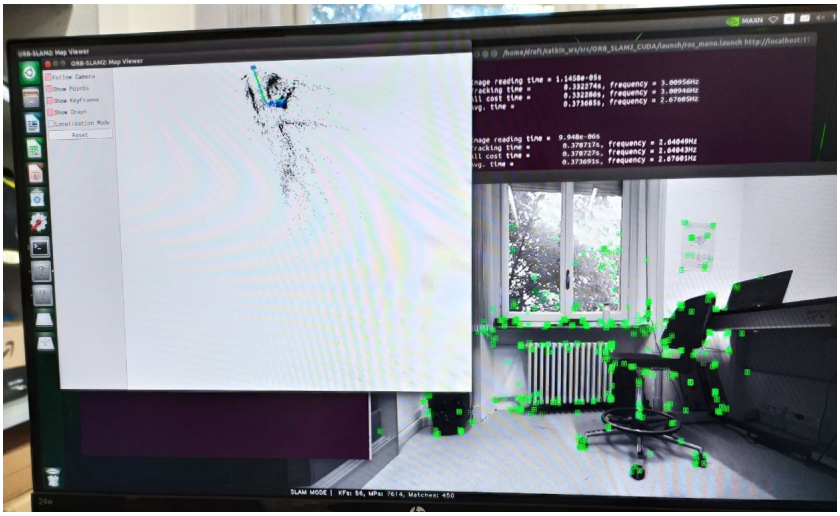


Figure 2.22: ORB-SLAM2:running phase. The tools allow to visualize the motion module and the map update.

For these reasons, and for the fact that the compatibility on Jetson Nano may cause problems, this package has not been considered anymore.
Another package has been taken into account as potential candidate to be chosen as the one for single-agent SLAM: Vins-Fusion.
This package has been only briefly tested on a laptop. It has shown great performance in terms of accuracy and robustness, and the localization data are very close to the T265 camera ones.
The mapping procedure is very similar to the ORB-SLAM2 one.
Unfortunately, the package require a discrete computational effort to be executed in real-time, and the Jetson Nano couldn't provide the required power. Therefore, it has not been further considered.
The third and last package that has been considered is RTAB-Map. There are a

lot of reasons to choose this package. The first important reason is that is the only package, among the tested ones, that is able to take the T265 odometry data as an input, by relieving this task from the SLAM package.
Anyway, if the T265 is not available, it is possible to use a lot of VO packages to extract localization data.
The second important reason is that it has been tested on Jetson Nano and it runs smoothly, guaranteeing real-time SLAM.
To be sure of this choice, some tests have been made. The set-up consists of a Jetson Nano with T265 and D435 cameras connected via USB ports.
The system is battery-powered and it has been held by hand in a mission which consisted in mapping an office environment, with some problematic details, such as shadowed zones and tight corners with low features.
RTAB-Map installation is very simple, since it belongs to the libraries that can be installed from the ROS library list.
To run the package it is sufficient to insert a node, with the appropriate syntax, inside of the *.launch* file.
Before explaining and illustrating the results of the tests, it should be clear how the output data of the algorithm are displayed and therefore analyzed.
The visualization will be performed by means of the ROS tool Rviz. The SLAM outputs are the odometry output and the produced map.
The first one is displayed by means of the tf representation. It is possible to see the movement of the tf associated to the moving camera with respect to the tf associated to the global map, which usually coincide with the tf generated by the camera at its starting point. In the following analysis of data, these frames will be called respectively **map frame** and **odom frame**. The mobile frame attached to the camera will be called **pose fame**.
For what concerns the map, as for most of the analyzed SLAM packages, the output is represented as a point cloud. Theoretically, the map can also be seen as a two-dimensional occupancy grid, but this isn't a useful solution if the objective is to visualize a 3D map, as the one needed in the case of the drone.
Obviously, both localization and mapping data are accessible since published as ROS topics.
However, in order to allow an easier visualization and a to get a data format that is at the same time accurate, but not too heavy from the memory point of view, RTAB-Map includes, for 3D maps, the octomap.
Octomap is a 3D mapping framework based on octrees [18].
Octrees are data structures that are used for spatial subdivision in 3D.
When visualized, every point of the map is represented as a cube, that is usually called a voxel.
An example of the map is illustrated in Figure 2.23.

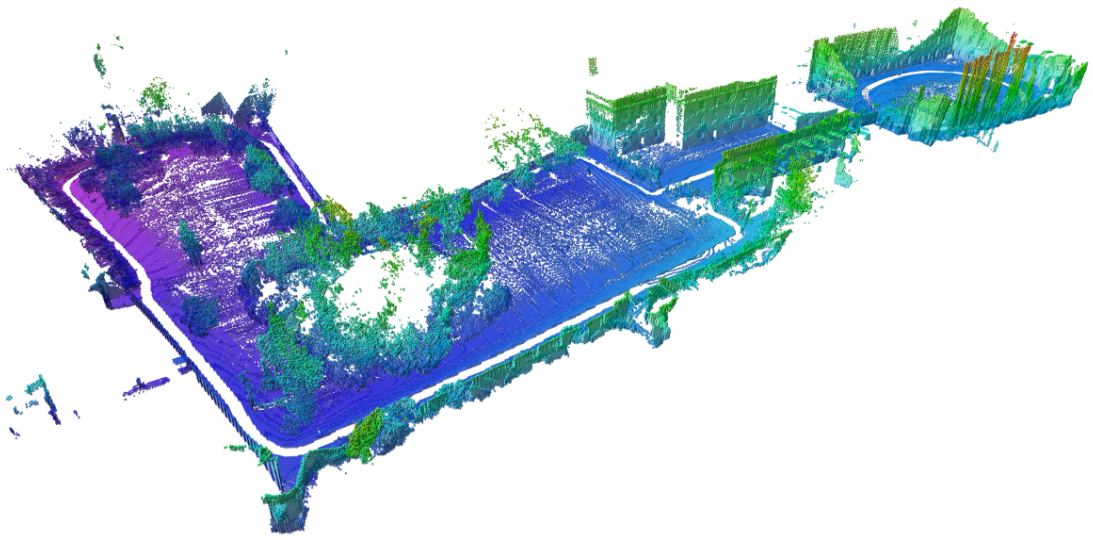


Figure 2.23: Example of 3D occupancy grid realized with Octomap, in this case the color represents the height of the map, from purple (lowest) to green (medium) to red(highest).

The Octomap are 3D occupancy grid and they are very helpful in the data analysis and saving.
Moreover, it is possible to use different kinds of color legends, based on the information about data.
For example, cube colors can be sorted by altitude, with respect to the map frame, or by occupancy probability.
Another useful feature of octomaps makes it possible to visualize the points of the map only in a specified range of height: if the user wants to create a map of a building on multiple floors, or the map of a single floor but covering both floors and ceilings, the resulting map could become very hard to read.
Cutting the map to a certain height is useful exactly for this reason.
Anyway, if all the system in which RTAB-Map is used needs only the map in real-time, the successive steps, e.g. path planning, will need to just subscribe the map topics.
Indeed, if there is interest in saving the map of an environment, in order to use it later or re-use it for a multi-session mapping, it is possible to save the map in a database file (*.db*).
RTAB-Map can load this map as soon as it is launched.
To launch a RTAB-Map node, it is necessary to know the desired configuration, among the available ones, and tune the parameters in the *.launch* file accordingly.
Figure 2.24 illustrates a possible configuration.

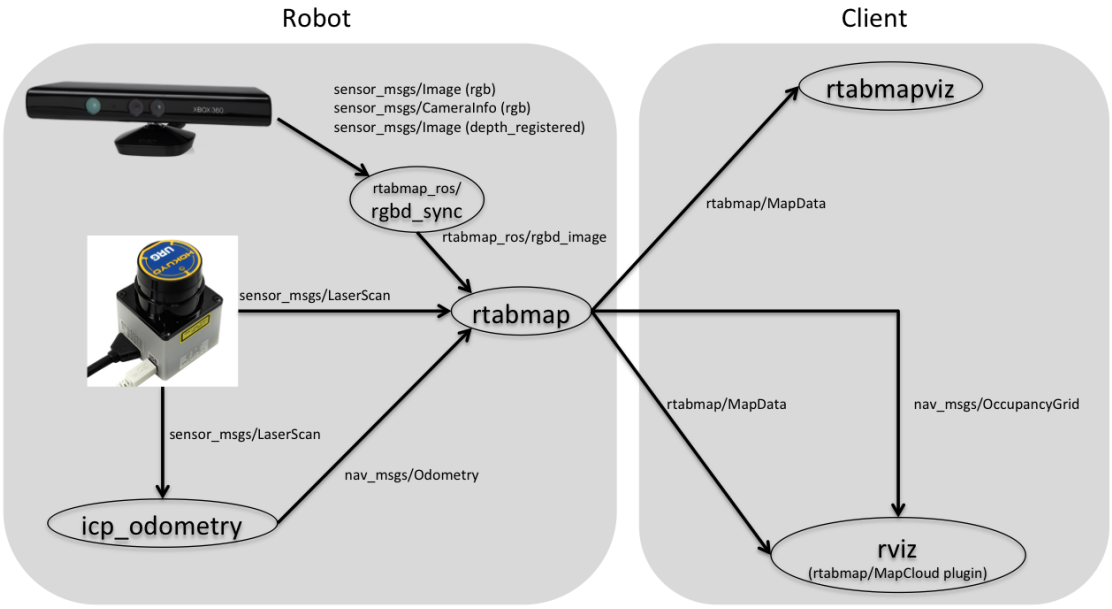


Figure 2.24: RTAB-Map possible working configuration: camera, lidar, external source of odometry [19]

There are many possible configurations, which depend on the selected hardware and the choices of the user.
The idea is that the package should be able to receive or compute odometry information and receive information about the environment.

- **Odometry**: by default, the parameter *visual_odometry* is set to true, which means that the package will launch a VO node, i.e. it will subscribe to a depth image, a color image, and a camera info topic, and it will extract localization data.
  However, if an odometry topic has been subscribed, i.e. the parameter *odom_topic* has been filled independently from its origin, it is used instead of the VO node.
  There is a third parameter that supersedes the other two.
  If *odom_frame_id* is filled with the name of the frame associated with the movement of the robot, the odometry data will be extrapolated from the dynamic transformations between the reference frame and the moving frame.
  If this happens, neither VO nor input odometry methods are used. Basically, the package has been developed in such a way that only one option at the time can be used, therefore it is always possible to tell the package which mode the user wants to use.

- **Images**: the default configuration dictates that the package uses a RGB-D image, which is a combination of a color and a depth image, such that each pixel is represented both as a depth information and a color information.
  In the default configuration, the parameter *depth* is set to true.
  This is done by subscribing to the depth image, the color image, and the camera info topics.
  These are the topics that are also needed for the VO node.
  If the depth parameter is set to false, the package can be switched to stereo configuration, by setting to true the parameter *stereo*.
  In this case, it will need to subscribe to a left image, a right image, and camera info topics.
  If both depth and stereo parameters are set to false, the package can use scans.

- **Scans**: the other method used by RTAB-Map to extract environment information is the laser scan.
  This information can be subscribed both if it is encapsulated in a laser scan message or a point cloud message.
  In the default configuration, the parameter *subscribe_scan* and the parameter *subscribe_scan_cloud* are set to false.
  If one of the two parameters is set to true, the package will use that information for the mapping, instead of the depth image one.
  Anyway, there are two possible configurations: if the depth parameters are set to true, RTAB-Map will use the scan for mapping purposes and the images to perform other SLAM tasks such as loop closures and some optimizations.
  On the contrary, if the depth parameters are set to false, there will be no additional tasks other than the mapping.
  In any case, it is needed to specify the scan cloud topic.

Other details about the parameter set-up will be discussed later.
Some tests were performed by using the default configuration of the package, with the exception that the odometry topic of the T265 tracking camera has been used instead of VO.
The hardware suite consists of a battery-powered Jetson Nano and the set of T265 and D435 cameras, held by hand and carried around an office environment.
This represents a valid indoor test, as demonstrates the fact that a lot of data sets available on line for package performances benchmarking are related to offices or industrial environments. Figure 2.25 shows the final map.
It is possible to see the accuracy of the map even in identifying tight corners and the definition of the entire map shows that, even if a loss of tracking happened, the package managed to solve the problem without any issue.

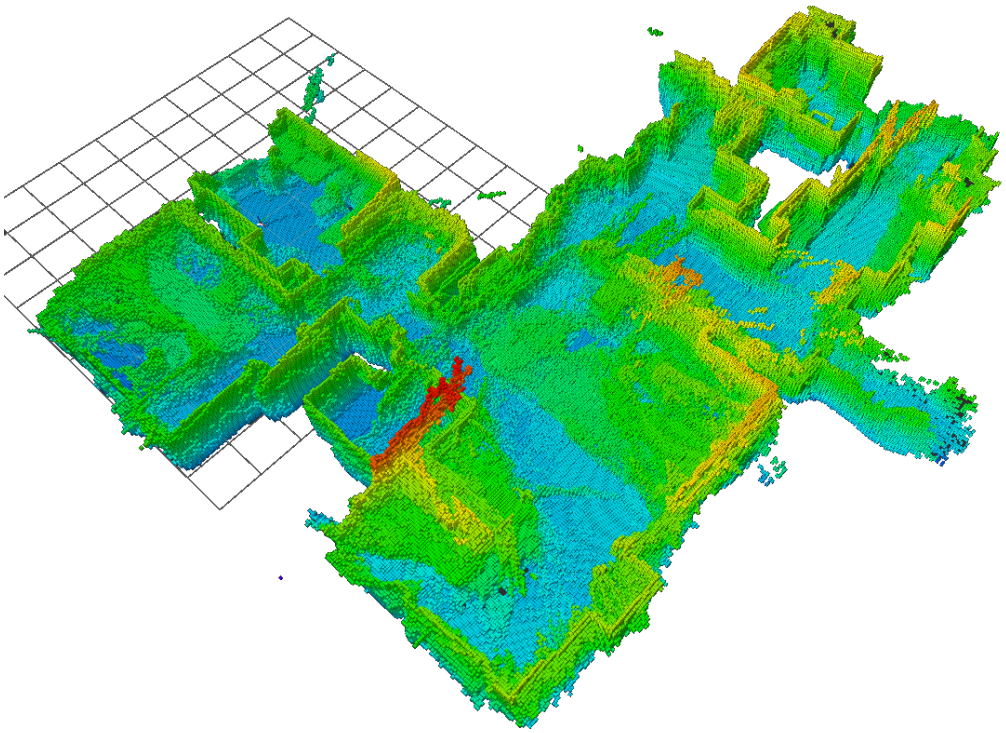


Figure 2.25: Octomap produced by a real-time test with RTAB-Map. The map is color coded by height. Rviz has been used for visualization

At [20] it is possible to see the video of the real-time test. The video has been accelerated at ten times the real velocity.
All the perks described in these last paragraphs make clear that RTAB-Map has been chosen as the reference package for this project.
Anyway, as said before, this does not mean that the strategy that will be developed for multi-agent SLAM is suited only for RTAB-Map.
For the sake of simplicity, in the following sections it will be described the RTAB-Map case, but the package can be easily edited to be adapted to any SLAM package. Obviously, in the simplest case it's only a matter of ROS topics remap, in the hardest the codes should be modified, but the

2.2 Package creation

This section addresses the creation of the package **pcl_handler**, that has the purpose of creating a set of utilities that can be used to perform multi-agent SLAM. Before entering in the details, it is important to describe the general idea behind this package.
The idea is to create a system that is able to share and combine data coming from the agents, using the powerful resources of the server.
The choice to create multiple nodes derives from the fact that this allows a decoupling of the entire logic, making easier the installation and usage on the agents. Moreover, the nodes architecture helps the development phase, since it is possible to localize eventual issues to a certain node.
In order to be able to exchange information along a standard wireless connection, it is required that data dimension is small enough to guarantee data consistency and real-time sharing. Therefore a method has been developed to share information with ROS interface.
The method consists in taking advantage of the fact that RealSense camera provide, through the ROS wrapper, compressed images, that can be then de-compressed and used without any problem.
On the server side, it is possible to run more computational demanding tasks, like the one related to the creation of a map, starting from the map provided by each agent.
Moreover, the way to compute the map is related to the usage, the combination and the filtering of point clouds. This kind of architecture brings a lot of advantages, like the possibility to extend this logic to an indefinite number of agents.
Indeed, these agent shouldn't necessarily be identical: it is possible to combine systems made of ground robots and aerial robots, or systems in which the agents have different sensor suites, or both.
In order to start with this, the first thing to do is to understand how to create a ROS package from scratch. Then, it will be analyzed the first steps on image computation. Finally, it will be discussed the data combination and the re-integration in the SLAM system.

2.2.1 Create a ROS package

The first step in the development of the package is its creation. *catkin* tools provide all the necessary commands to create a basic version of a package.
It is possible to create a standalone catkin package as well as creating a package inside a *catkin_workspace*. In this case, it will be used the latter one.

The catkin command **create_catkin_pkg** enables the user to create a package folder, with all the needed sub-folders and two important files that are absolutely necessary to build the package:

- **CMakeLists.txt**
- **package.xml**

Once the package has been created, it is needed to specify in the *package.xml* file all the dependencies. Fortunately, when the user builds the catkin workspace, all the libraries are imported and loaded. Indeed, it will automatically load all the nested dependencies, therefore there is no need to do it manually.
Then, it is needed to edit the *CMakeLists.txt* file. Some fields should be compiled.

- *Required CMake version*: the required version of CMake needed. Catkin requires version 2.8.3 or higher.
- *Package Name*: name of the package which is specified by the CMake project function.
- *Find Package*: which other CMake packages need to be found to build the project.
- *Catkin Package*: it is the required macro to specify catkin-specific information to the build system. This function is called before declaring any target. Here should be included all the catkin and non-catkin projects that this project depends on.
- *Specify build targets*: here should be included all the executable targets and the library targets

Each time a new library is required for the project, it will be specified in both these files. Each time a new executable will be created, it will be specified in the *CMakeLists.txt*. If this doesn't happen the catkin make command will not, in the first case, recognize dependencies or it will not, in the second case, build the executable.
The libraries used in this project belong in part to ROS libraries, which help to manage messages and data input/output.
The other libraries will be described in detail when used in the executable files. Anyway, some common libraries are added, such as Eigen, OpenCV, PCL. The firsts two are already been described in Section 2.1.4, in conjunction with the installation of ORB-SLAM2, while the third one is used in many SLAM packages and it is the Point Cloud Library. It contains a set of functions that enable computing, convert, rotate, merge, and other operations on point clouds.
In the following sections, the created nodes will be explained.

2.2.2 Image compression and decompression

The first issue that will be faced in this development is data sharing through a wireless connection. To keep consistency, it should be assured that, if the connection is stable, the data can travel from an agent to the server and vice-versa.
This means that whatever is shared with the connection should be small enough to travel without any issue.
As it will be shown, a lot of the data that the camera can provide are really big, and not suited for real-time implementation.
The proposed solution to this problem is based on [21]. It consists of the development of two nodes that can decompress a compressed depth image and reconstruct it with low losses in the compression phase.
The idea is to enable each agent to send a compressed image with the wireless network and, once it gets to the server, it is decompressed, reconstructed and then it can be used in a lot of possible ways. Figure 2.26 shows the data flow.

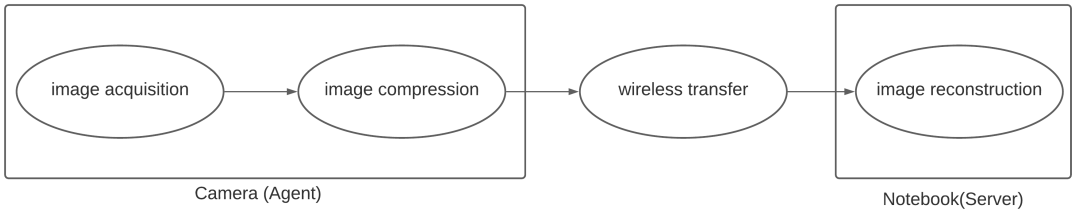
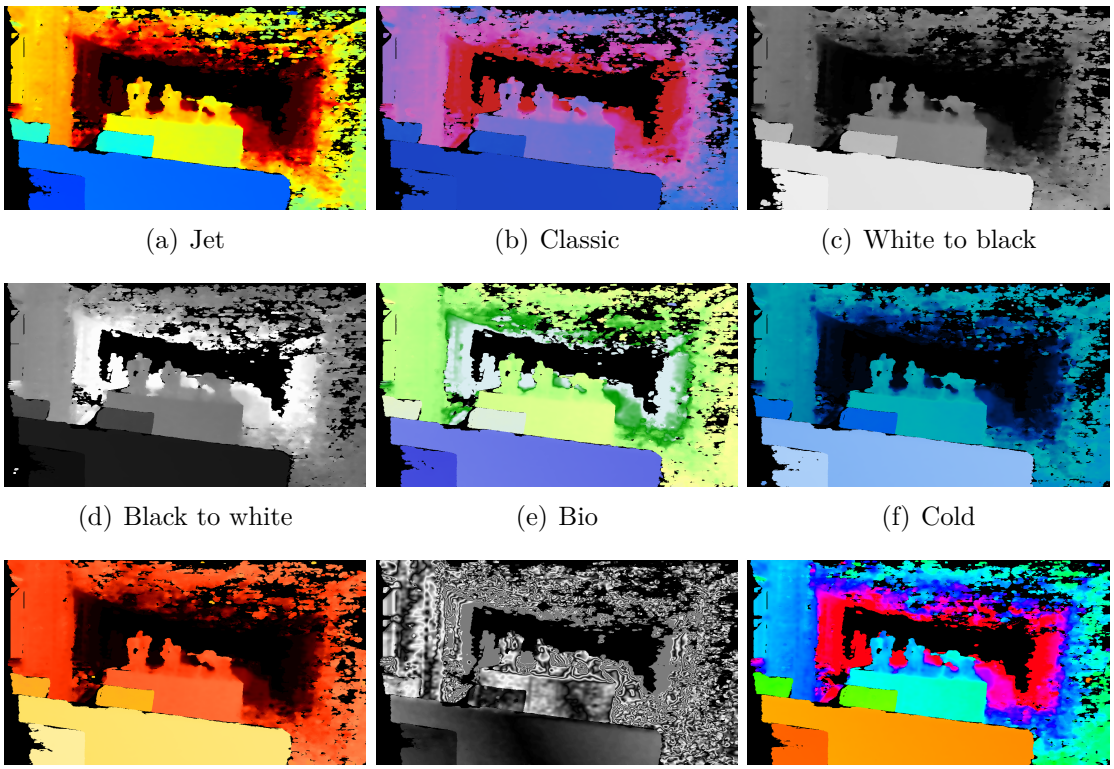


Figure 2.26: Image processing. Depth images are generated from the camera and compressed. They are shared via wireless network and then reconstructed.

The study presented in the white-paper concerns the development of such a solution employing a C++ script, which, even if it is available as a standalone script, still belongs to the *librealsense* SDK. In this case, the code has been edited and adapted to be applied directly to ROS messages.
The D435 depth camera, once launched with ROS, produces a series of topics that are related to color images, depth images, extrinsic and intrinsic parameters. The topic list includes also compressed images. For reasons that will be explained in a bit, it is not possible to directly use these images without some post-processing on images.
In order to understand how these images are processed, it is needed to start from the side where images are generated, i.e. the camera side.
The camera always performs depth image compression, but for this compression to make sense, it is needed a process called *colorization*.
Depth images are represented as a gray-scale image: it is a 16-bit resolution image. Instead of using compression techniques directly applied to depth images, it is

possible to colorize the image. This implies that the image can be then treated as a 24-bit resolution RGB image. As a consequence, many different compression methods available in the literature can be used.
Before exploiting the size reduction accomplished with this process, it is necessary to dig a little further into the colorization. This process can be automatically done by the camera just by using a filter called *colorizer*. It is enough to specify the name of the filter in the corresponding field in the *.launch* file and it is done.
Indeed, RealSense provides nine different color representations for depth images. Figure 2.27 illustrates the options.



(a) Jet (b) Classic (c) White to black

(d) Black to white (e) Bio (f) Cold

(g) Warm (h) Pattern (i) Hue

Figure 2.27: RealSense D435 depth image colorization options. Jet is default. With no colorization active, the default is black to white.

If no colorization filter is active, the depth image can be seen with the *black to white* representation. (darkest pixels represent closer objects, white pixels represent farthest ones). However, if the colorization filter is active, Jet is the default color space. Anyway, the white-paper suggests using the Hue color space, which is shown in Figure 2.28.

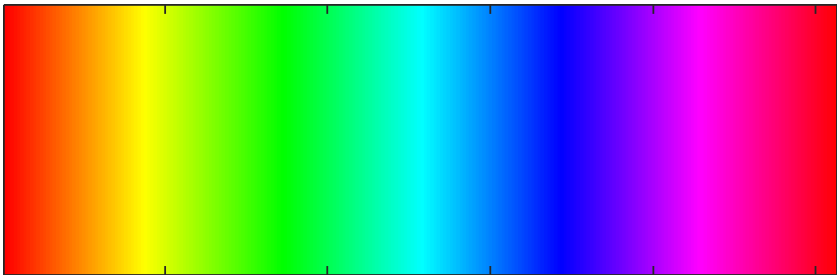


Figure 2.28: Hue color space. It has six gradations for R, G, and B, and one of them is almost always 255, so the image is never too dark

This suggestion is due to the fact that, if the color space is analyzed, it is possible to see that, in the conversion process from gray-scale to hue, the image doesn't become too dark, by guaranteeing a high level of details.
Once all the details about image compression are defined, it is time to discuss the image reconstruction.
Other than the colorizer filter, other filters can be applied in the post-processing phase. *realsense-ros* makes different filters available, and it is sufficient to specify them in the *.launch* file, in the filter field to be used. Moreover, it is possible to use the dynamic reconfigure package to change the settings of these filters from the default parameters.

- **Decimation**: filter used to reduce the complexity of the depth scene. It can be tuned to change the decimation factor. In this case, since the aim is to get a compressed, high-resolution image, this filter is ignored.
- **Spatial**: filter used to perform high-quality edge-preserving [22]. Here it is used with the default settings.
- **Temporal**: filter used to improve the depth data persistency. If used, when pixel information is missing, the filter decides if the value should be rectified with stored data. Not used, since it is may introduce blurring if the scenes are not static.
- **Holes Filling**: this filter uses the nearest neighbor to a pixel that is missing data and rectifies it. Used, with default settings.

Filters parameter are defined as dynamic: ROS discerns nodes parameters between static and dynamic. Once edited a static parameter, it is necessary to restart the node to see the changes. Dynamic parameters can be edited at runtime, without having to restart the node. To do this, the package **dynamic_reconfigure** is used. This package contains a set of utilities, like the possibility to list the reconfigurable nodes, and, for such nodes, the list of the reconfigurable parameters. Indeed, it

includes the possibility to set a parameter value.
In this case, the important reconfiguration is related to the colorizer filter: as mentioned before, the default color code is the Jet, therefore it is needed to be modified. Instead of opening the **rtq** tool to reconfigure the filter, a *.launch* file has been used. This starts a *dynamic_reconfigure* node.
Another important parameter to take care of is the *histogram equalization*. If this parameter is set to true, the colorization is not performed as it is needed, therefore the image reconstruction will fail. Finally, in Table 2.4 it is shown the size of ROS *.bag* files for different image formats.

Bag name	Resolution	FPS	Info	Size per sec. (MB)
bag1	640x480	30	default	14.1
bag2	640x480	90	max. fps	43.0
bag3	1280x720	30	max resolution	40.6
bag4	848x480	60	medium res. and fps Hue colorization	41.5
bag5	848x100	100	flat image and max. fps Hue colorization	19.9
bag6	640x480	30	compressed Jet	1.3
bag7	640x480	30	compressed Hue	0.92
bag8	640x480	90	compressed and max. fps Hue colorization	1.83
bag9	1280x720	30	compressed and max. resolution Hue colorization	0.93
bag10	848x100	100	compressed and max. fps flat image Hue colorization	0.797
bag11	848x480	60	compressed medium res. and fps Hue colorization	0.866

Table 2.4: ROS bag tests on depth images: the difference of size between regular and compressed images is substantial

The bag size makes clear that sharing standard images is possible, but it might not guarantee real-time. Anyway, this solution will need a powerful connection.
On the contrary, the compressed image is from 10 to 40 times smaller than the uncompressed one.
This implies that real-time Wi-Fi sharing is possible, even with a normal connection.

Once it is assured that the image transmission is possible and it works, it is time to discuss the image reception.
To recover the 16-bit resolution depth image from the colorized one, it is necessary to know some information about the camera and the filter settings.
First of all, the color space number of discrete levels should be known, as well as the equations that are used to convert the depth values into R, G, and B values. Moreover, there is another important couple of data that is crucial in both compression and decompression: the minimum and maximum depth parameters.
When colorization is performed, other than the color code, it is possible to specify the minimum and maximum distance parameters: the camera adjusts the colorization based on these values.
Therefore, it is necessary to know their value and, of course, to use the same ones in the decompression.
Now that the theoretical principle behind the image transmission has been described, it is time to discuss the code.
As said before, the compression side is managed by the camera, so the code will deal only with the decompression.
To perform this kind of process, it is necessary to know what is the system input and what is the expected output of the node.
The node based on the white-paper methodology requires the colorized depth image as input and it outputs a gray-scale image.
To perform this operation with ROS, it is necessary to subscribe to the image topic (**sensor_msgs/CompressedImage**) and publish another one.
After the subscription is done, the image must be converted to an OpenCV image, i.e. to a camera matrix.
The conversion is handled by the library **cv_bridge**, which is an OpenCV library for ROS that provides all the functions that allow to convert ROS topics into OpenCV images, or vice-versa.
This library API has been released both in Python and in C++.
Unfortunately, the functions that can convert compressed images into OpenCV images have been defined only in Python. This is not good, since a large part of the already existing code that will be used is written in C++.
For a first solution, two nodes have been developed: the first one for conversion, the second one for depth map recovery. The idea is to export the part of the code that exists from C++ to Python, so it is possible, in the final release of the package, to use only one node. Thus, the first node has been written in Python.
The details are described in the Pseudo-code 1. Before digging into the code explanation, it is important to remember what a *callback* function is: a callback is a function that is passed as an argument to another function, which is expected to execute the argument at a given time.
This comes in handy both with synchronous and asynchronous cases.

Thus, it is an excellent way to perform a certain set of instructions each time a ROS topic is subscribed by the node.

Algorithm 1 Compressed Depth converter node. It subscribes to a CompressedImage topic and it publishes a Image topic.

```
1: procedure CLASS DEFINITION
2:     procedure ___INIT___(self)
3:         ▷ ROS subscriber initialization: CompressedImage
4:         ▷ ROS publisher initialization: Image
5:     end procedure
6:     procedure CALLBACK(self,data)
7:         encoding = bgr8
8:         cv_image = cv_bridge.compressed_imgmsg_to_cv2(data,"encoding")
9:         ▷ cv_bridge function for conversion. It needs to know the encoding
10:        output_image = cv_bridge.cv2_to_imgmsg(cv_image,"rgb8")
11:        ▷ cv_bridge function for conversion to ROS message
12:        ▷ ROS publisher
13:    end procedure
14: end procedure
15: procedure MAIN(args)
16:    ▷ Node initialization and cleanup
17:    ▷ Call to class
18:    ▷ Node run (ROS spin)
19: end procedure
```

Once this script has been completed, it has been possible to start working on the second script. It consists of a ROS subscriber to the image published by the Python node, a function that converts RGB images back to depth gray-scale images and a portion of code that takes the image information and produces a *PointCloud2* message. Moreover, there is a subscriber to the *CameraInfo* topic, which is needed to get camera intrinsic and extrinsic parameters.
This node's details are specified in Pseudo-Code 2. This nodes includes the OpenCV, the PCL_ROS and the Eigen libraries, other than ROS standard libraries and sensor_msgs libraries.
This second node is not only responsible for the image recovery, but it starts to implement the data manipulation, which will be the focus of the next section.

Algorithm 2 Color to Depth converter node. It subscribes to a CompressedImage topic and it publishes a Image topic.

procedure CLASS DEFINITION(ImageConverter)
    ▷ ROS subscriber initialization: CameraInfo
    ▷ ROS subscriber initialization: Image
    ▷ ROS publisher initialization: PointCloud2
    $width, height$
    $min\_depth \leftarrow 0.29$
    ▷ minimum distance. To avoid depth inversion, it's offset from 0.3 to 0.29
    $max\_depth \leftarrow 6.0$
    $depth\_units \leftarrow 0.0010$ ▷ depth value scale factor, from RealSense viewer
    $hue\_value \leftarrow 0$ ▷ used later in the conversion
    $K, invK$
    ▷ K is the intrinsic parameter camera matrix, but invK is used
    procedure INFO_CALLBACK(*CameraInfo msg)
        ▷ Intrinsic matrix, height and width are copied in the respective variables
    end procedure
    procedure RGBTOD(r,g,b)
        ▷ it takes R, G, B inputs and returns the depth value
    end procedure
    procedure COLORIZEDDEPTHTODEPTH(color_mat, depth_mat)
        ▷ It takes the color image matrix and returns the depth image matrix
    end procedure
    procedure IMAGE_CALLBACK(*Image msg)
        cv_ptr ▷ Subscribed Image declaration
        $cv\_ptr = toCvCopy(msg, encoding : rgb8)$
        $depth\_mat$ ▷ depth image matrix initialization
        $ColorizedDepthToDepth(cv\_ptr, depth\_mat)$
        ▷ Here it is performed the conversion from color image to depth image
        ▷ PointCloud2 message initialization
        ▷ PointCloud2 fields filling
        ▷ PointCloud2 message header fields filling
        ▷ ROS publisher
    end procedure
end procedure
procedure MAIN(args)
    ▷ Node initialization and cleanup
    ▷ Call to class
    ▷ Node run (ROS spin)
end procedure

2.2.3 Point cloud manipulation

This section addresses the generation and manipulation of Point Clouds.
Being able to manage these data is important since it allows to perform operations on both the input and output data of SLAM packages.
Usually, the packages produce two fundamental topics: the localization topic and the map topic. The first information can be encapsulated in a *odom* topic, or in the *tf*. The second one depends on the mapping sensor and the package structure.
For UAVs, it is recommended to use a 3D map, therefore the possibility to use **occupancy grids** is ruled out.
Therefore, it will be possible to use point clouds or Octomaps.
For what concerns the input data, it is possible to take into account point clouds or laser scans.
Theoretically, RTAB-Map, as well as many packages that can take a point cloud as input, allows the subscription to these inputs because it is the most appropriate way to take into account Lidars. Therefore, the point cloud input is originally thought to be the data acquired from a 3D Lidar.
Anyway, independently of the motivations, it is clear that it is possible to provide point clouds as inputs, so handling them is important.
The nodes that will have as a common factor the usage of functions that belong to the libraries Eigen and PCL.
The first thing to do in this analysis is to explain why a point cloud manipulation is necessary.
One possible reason concerns the size of the clouds. The data transfer, along with the usage of these data as input for SLAM nodes, are difficult tasks if the point cloud size is too big: even by providing the system with a large bandwidth connection and/or with a high-budget onboard computer, there's no guarantee that the transfer is feasible and the real-time SLAM can run.
On top of that, there's one more problem: even on a powerful laptop, it is not possible to save ROS bags that last more than a few seconds.
The process of saving data in bags is managed by means of a buffer.
If the buffer is full, which happens after a certain amount of time, some data received from the sensor will be lost.
Depth cameras like the D435 can output a point cloud. To do that, it is sufficient to set to true the corresponding parameter in the camera *.launch* file.
Anyway, this cloud is very dense and it contains information about the color too. If compared with a point cloud without color information, it looks clear that there is a substantial difference in size. It is enough to imagine that the color information occupies 24 bits for each point in the cloud.
Indeed, as it will be shown in a bit, removing the color information couldn't be enough. The high density is still a substantial problem that should be solved before

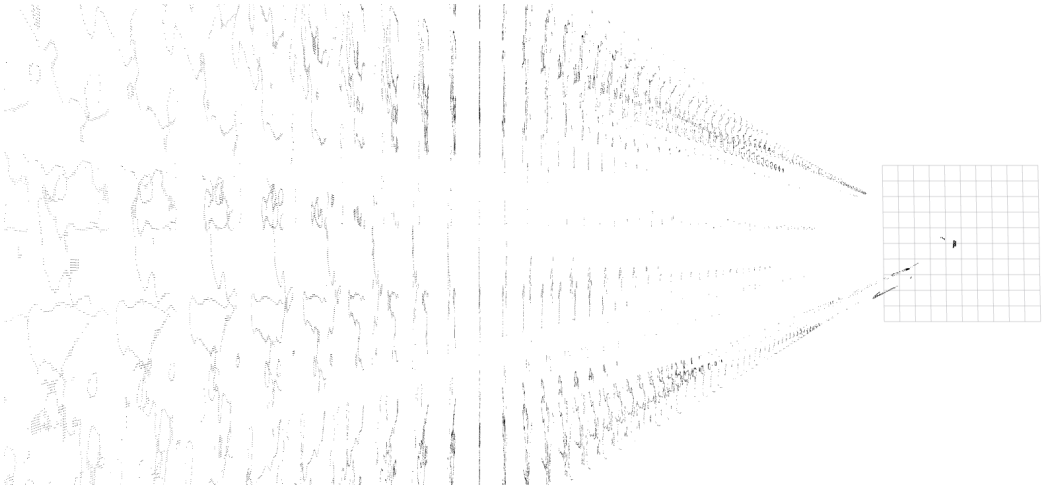
thinking of combining data. Of course, going in the opposite direction, i.e. getting a much less dense cloud, could be a problem.
If the density is too low, there's a high risk to lose important environmental information, causing a substantial loss of accuracy in the map building.
Moreover, if the density is low, problems related to cloud matching may occur.
As it will be shown later, when the matching of two clouds is attempted, it is enough that one of them has low-density to cause the impossibility to find a match.
Some tests have been performed in order to quantify the size of the clouds in different situations. Table 2.5 illustrates the results.

Bag name	Cloud type	FPS	Info	Size per sec. (MB)
bag1	RGB	30	camera default	42.4
bag2	reconstructed no color	30	axis limit 5.5 m	17.8
bag3	reconstructed no color	30	axis limit 5.5 m voxel filter box side 0.1 m	0.135
bag4	reconstructed no color	30	axis limit 5.5 m voxel filter box side 0.05 m	0.45
bag5	reconstructed no color	30	axis limit 5.5 m voxel filter box side 0.03 m	0.975
bag6	reconstructed no color	30	axis limit 5.5 m voxel filter box side 0.01 m	4.0

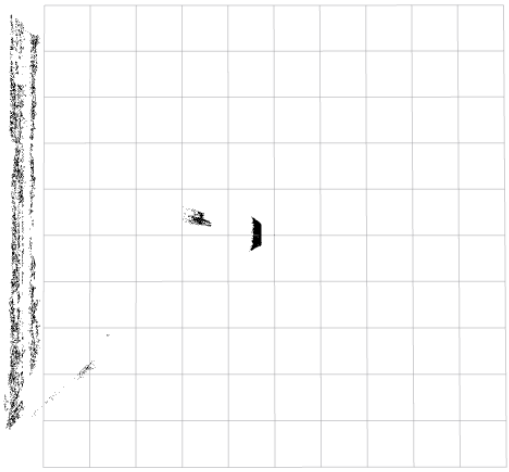
Table 2.5: ROS bag tests on point clouds: the difference in size between different configurations is clear.

As highlighted by the collected data, the filtering process is an important part of the point cloud post-processing.
Indeed, it is needed to enter into details about the filtering.
This analysis will not consider the noise removal filters, but it will be based on the noiseless assumption.
This category allows to remove the so-called *shadow points*, i.e. points that belong to the cloud, but don't actually exist.
The two types of filtering strategies that will be here considered are the axis-limit

and the down-sampling.
The first one is a simple filter that just removes from the point cloud all the points such that the value along one of the axis is higher than a selected threshold.
The D435 is capable of acquiring depth information at a range up to ten meters, but it is practically very unlikely that objects are detected that far.
However, assuming that objects are detected, there is little information about them and the depth points that are found in this way have small precision.
Therefore, it is reasonable to set a limit condition to the depth data. There are different ways to do this. PCL provides both C++ functions and ROS nodelets that implement this kind of filtering. In the second case the nodelet, that belongs to the package *pcl_ros* is called **passthrough**.
It is enough to edit the launch file, by remapping the input topic to use it. It is always necessary to remember that, in order to a nodelet work, the published topic should be subscribed by another node, otherwise it is impossible to visualize it with any of the tools that ROS provides.
Anyway, this tool has been used and it is helpful in the package design process, but in this case, it will not be applied in the final release. The reason behind that can be found in the image recovery node, especially in Pseudo-Code 2.
As specified before, the depth parameters should be the same both on the compression and decompression side. Among these parameters, the maximum and minimum depth distance is specified.
During the process of image recovery, the points of an image that have a depth value larger than the maximum distance are discarded.
The consequence is that, if a point cloud is produced starting from the reconstructed image, it will be automatically limited to this value.
Finally, it is important to discuss the practical meaning of using this filtering technique: first of all, it reduces the point cloud size and it globally increases its accuracy; second of all, it avoids taking into account nonexistent points in certain situations.
The latter can be experienced, for example, in indoor environments: a reflecting surface, such as a mirror, a television screen, or a window may cause the laser installed inside the D435 camera to not work properly.
As a consequence, the point cloud will contain points that are very far and in the real world don't even exist.
By applying an axis-limit filter it is possible to avoid this kind of problems. The implications lie on the fact that a certain quantity of data could be lost in this way. Therefore, it is recommended to perform experiments to find a suitable trade-off between depth capabilities and unwanted point removal.
Figure 2.29 illustrates the comparison between two point clouds of the same scene.



(a) No filter Point Cloud

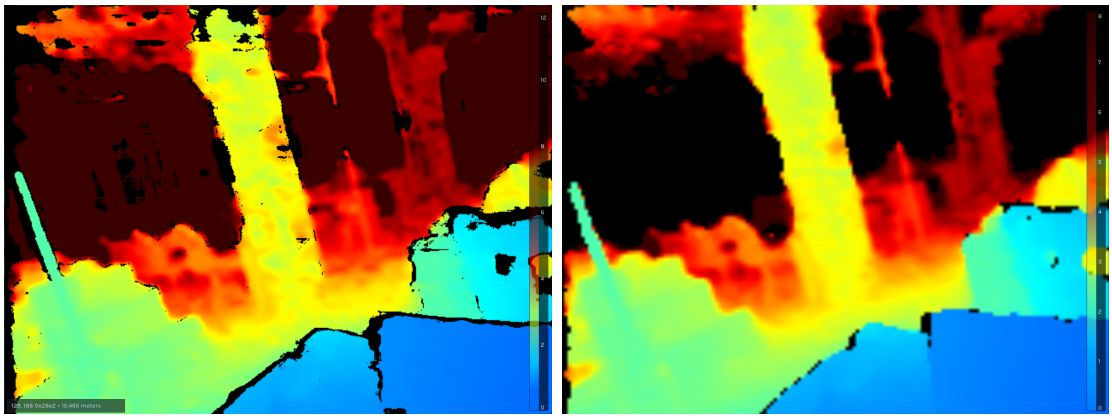


(b) Axis-limit filter Point Cloud

Figure 2.29: Effect of the axis-limit filter on a scene that includes a reflecting surface. The axis limit is set to *5.5 m*

As it is clear, the camera is pointing at a window, that messes up the laser and introduces a lot of non-existent points. The filter doesn't solve the problem completely, but it helps reducing it.
A possible solution consists in the development of a Computer Vision algorithm that automatically recognizes troublesome surfaces and discards the associated pixels.

The second category of filtering that it will be analyzed is the **voxel**. There are two ways to reduce the complexity of a depth scene: the first one might be the decimation process, i.e. applying to an high-resolution image a post-processing filter that reduce the image resolution. An example of the effect of decimation is shown in Figure 2.30.

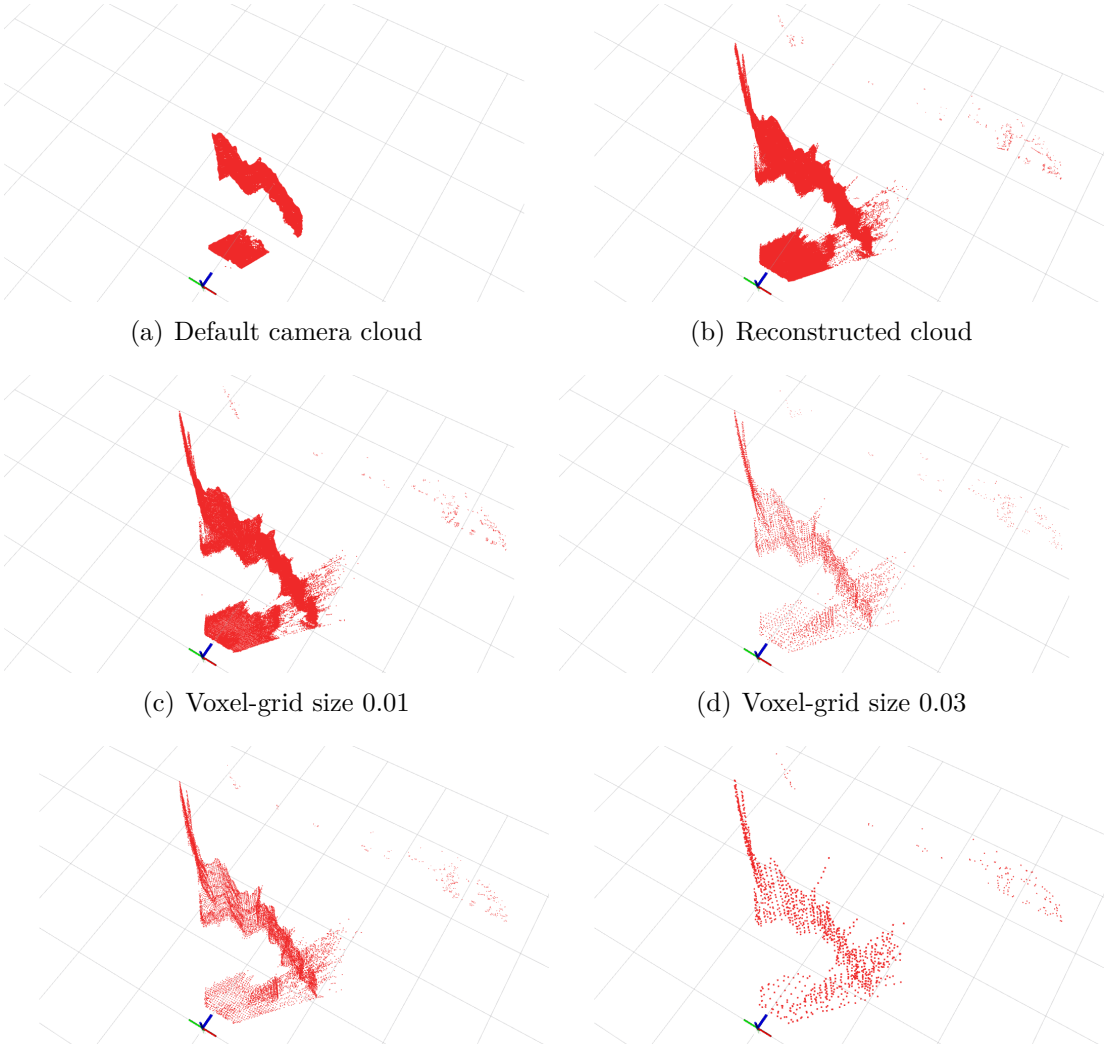


(a) Standard Image (b) Decimation Filter

Figure 2.30: Effect of the decimation filter on a depth colorized image. The decimation factor is equal to 3.

Even if it's subtle, it is possible to notice that the decimated image is blurry, and the detail level of the image gets way worse, implying a loss in the depth scene information.
Since the data flow that are used for this package requires that the compressed image and the reconstructed image have the same characteristics, it is not recommended to use a decimation filter.
It might be possible to use it, but then the reconstruction algorithm should be modified as well, and it doesn't seem the best solution or, at least, the most simple and immediate one.
Therefore, the down-sampling strategy has been chosen. It looks like a valid choice also because the main drawback of a filtering strategy like this one is not too relevant: to down-sample an image requires a certain amount of time, that inevitably slows down the frequency of publication of point cloud topics.
Anyway, since the operations on point clouds are performed entirely on the server, and they don't necessarily have to satisfy real-time requirements, the time delay does not represent an important issue.
Indeed, there's a reason why the voxel filter introduces this delay: a standard down-sapling of a point cloud consists is dividing the entire environment in little boxes of specified dimension and then replacing all the points inside one box with

its center. The voxel does a slightly different procedure, since it approximates the points contained in a box with the centroid, which belongs to the original point cloud.
This procedure avoids to introduce "fake" points and it guarantees to keep an high level of accuracy.
There's another concern that regards the usage of the voxel filter, i.e. on which point cloud to use it.
If it is assumed to face a situation in which two or more point clouds should be merged, it is possible to apply a filter on both the input point clouds, or it is possible to filter the resulting merged cloud.
The importance of understanding this concept lies in the goal of the point clouds managing nodes: the idea is to find a way to combine the depth information coming from different agents, therefore one reasonable approach might be to combine together multiple point clouds.
If applied before the merging, the advantage in terms of computational cost that is obtained is substantial, since the points' manipulation regards a much smaller amount of data. The drawback of this solution is related to the number of clouds. If more clouds are fused, it might happen that in certain situations, when more cameras are pointed towards the same direction, the merged cloud is very dense and too detailed.
In this way the advantage of an accurate but at the same time less complex scene is lost.
A possible solution can be found in the application of the voxel grid filter to the merged point cloud too. It causes an additional delay to the system, but the map density will be uniform.
The other way, i.e. to apply only the filter to the merged cloud, is without any doubt the worst one: until the number of clouds is low the velocity of the merging and filtering combined might be still acceptable, even if not optimal. If the number of clouds increases, the computation becomes too heavy and unfeasible.
The size issue is relevant even in the hypothesis of data sharing through a wireless connection. As already stated before, it is not possible to share a raw point cloud via Wi-Fi, therefore some kind of computation is necessary.
On top of that, this solutions can introduce some problems in the testing phase, since it makes hard any data visualization method and it is impossible to save ROS bags, for the same reasons described before.
Finally, the best solution is to apply a voxel filter for each cloud, and determine the number of clouds beyond which the merged map needs filtering too. This solution is acceptable also because the process is decoupled between agent and server: for size reasons, the single cloud down-sampling must occur on the agent side, while the merged cloud down-sampling is done on the server side.
Figure 2.31 shows the aspect of the point cloud in different voxel-filter sizes.



(a) Default camera cloud

(b) Reconstructed cloud

(c) Voxel-grid size 0.01

(d) Voxel-grid size 0.03

(e) Voxel-grid size 0.05

(f) Voxel-grid size 0.1

Figure 2.31: Effect of the axis-limit filter on a scene that includes a reflecting surface. The axis limit is set to *5.5 m*

2.2.4 Point cloud Merging

After all the necessary computations on a single point cloud have been described in details, it is necessary to address the problem of merging two or more point clouds. The data flow, shown in Figure 2.32, is thought to allow the server to receive multiple point clouds and fuse them in a unique map.

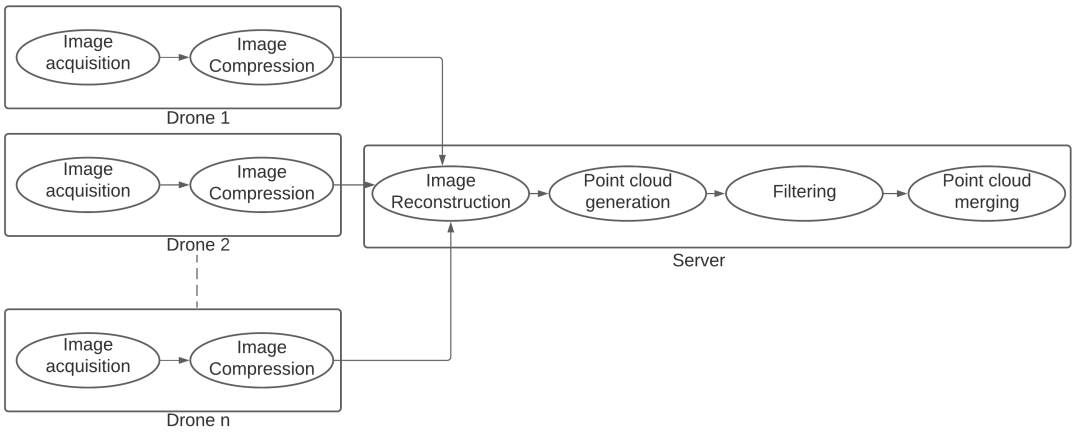


Figure 2.32: Data flow for the drone fleet: the server manages the point cloud computation and cloud merging

There are different ways that are worth exploring for what concerns point clouds merging and integration in SLAM algorithms.
For sake of simplicity, the following paragraphs will refer to a two clouds case.
Indeed, it is also important to specify that an important assumption has been made: to know exactly the starting point of each agent.
This origin is helpful because, by knowing the reference frame with respect to which the odometry of each agent is defined, it is possible to know the agents' position in a global reference frame.
As a consequence, the points belonging to the clouds, that are defined in the agent reference frame, have known coordinates in the global reference frame.
Now, if the initial position is unknown, each agent performs SLAM on its own, until at least two of them are located in the same area.
To being able to determine the absolute coordinates of the points in the unknown position case, it is needed to perform a real-time matching check: if two points extracted from two different agents are a match, the system can apply a multi-agent SLAM strategy.
Anyway, this solution has its drawbacks: there's a possibility, depending on the

fleet's mission, that the agents don't get in touch at all.
This is, for example, the case of exploring missions, in which it is necessary to cover the largest possible area as fast as possible.
Moreover, there's the issue that a point matching algorithms is indeed time demanding and it could corrupt the real-time requirement. Also, it increases a lot the complexity of the algorithm.
By knowing the coordinates of the origin in the global frame as an a priori information, it is possible to perform SLAM without any constraint.
Once the preliminary assumptions are explained, the first way, that has been analyzed and tested, can be described: it is the one that could guarantee the fastest system.
It consists of combining each sample of point cloud received from the agents in a merged cloud, then this cloud is filtered and it is sent to each agent to work as an input of the SLAM system.
As it has already been told, the RTAB-Map node can subscribe to a *PointCloud2* topic, so technically it is possible.
This approach wants to use the multi-agent logic to create an input for the system that is, basically, a substitute for the lidar input.
Anyway, there are several reasons why this solution doesn't always work.
The first one lies in the data consistency: it is important that each cloud, once processed on the server, has the same publication rate and the same clock timer.
Unfortunately, this is not always possible, mainly because the rate depends on the size of the cloud and the wireless communication time.
Luckily, there's a simple solution, i.e. using a time synchronizer node.
The package **message_filters** includes a node that can subscribe to a number of topics that goes from 2 to 10 and it synchronizes them.
Therefore, it is just a matter of writing a node that subscribes to the received clouds and re-publish them.
This solution makes the information consistent, but it implies issues if the connection between an agent and the server is too slow.
Moreover, it implies inevitable data loss and the merged cloud might have a low publication frequency.
The tests performed with two clouds have shown that all the computation can bring the merged cloud to a publishing frequency of about 3 *Hz*.
The second reason why this solution is troublesome lies in the RTAB-Map node: this algorithm has been written in a way that from, the point cloud input, it expects a lidar-kind stream, that moves with the odometry data.
The merged cloud is subject to rotations and translations, therefore, in most cases, no cloud or just a portion of the cloud are processed by the SLAM package.
This happens because RTAB-Map isn't able to detect this situation and it is designed to think that the environment is moving, not the robot, therefore the

moving points won't be added to the global map.
The last reason that will be presented here is related to the localization data.
In order to bring the points of the cloud in the global reference frame, it is necessary to apply a homogeneous transformation, for which angles and distances of each axis are needed.
The only available information is the odometry of the T265 camera.
However, these data represent the position of the robot with respect to the starting point of the agent, which can be stored in a transformation matrix.
Anyway, the needed transformation is not this one, but its inverse.
The inverse of a matrix, especially one that includes nonlinear functions as the sine and cosine ones, can lead to a discontinuity that might corrupt the data.
All these reasons make clear that this has not been the chosen path for the final release of the package.
It is now possible to describe the proposed solution.
A further assumption is made for this solution to make sense: the multi-agent SLAM logic is though for a multi-session SLAM: the fleet works in two phases: in the first one all the computational effort is focused on the mapping, while in the second one is focused on the localization and the map update.
Each agent will send images in real-time to the server, that will produce an "updatable" cloud: each time the point cloud of an agent is published, it does not only contain the actual cloud but the pasted ones too. It is basically a local map related to a single agent.
Then the algorithm fuses all the local maps together.
Once this is done, it is sufficient to have just a few seconds between the first and the second SLAM phase in order to load the complete map as an RTAB-Map database file in each agent SLAM node. When the second phase starts, the first thing that RTAB-Map does is load the map.
Therefore, the agent will see not only its map but the other agents' ones too.
The main drawback of this solution is that it is not possible to enforce the SLAM logic for a single-phase mission.
Now the details on how this procedure works will be described in detail.
Figure 2.33 shows the entire data flow. It is possible to notice the presence of a package called **octomap_server**.
This is a package that works on the octomap principle, but it can build incremental point clouds.
The advantage of using this package, instead of writing a brand-new node lies in its simplicity: it just need to know the odometry reference frame and the point cloud topic name and it can produce a map.

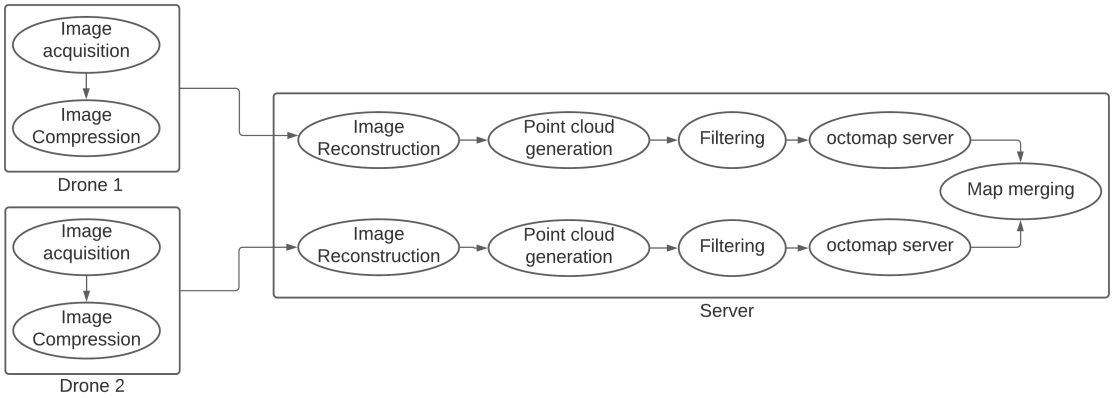


Figure 2.33: Data flow for a two agents system. There is a new block, octomap_server. It acts on the cloud right before the merging.

In order to use the *octomap_server* package, it is enough to set-up a *.launch* file. Among the possible outputs, there is the possibility to choose an occupancy grid, an octomap or a point cloud.
The latter has been the choice for this project. Once an incremental cloud is built for each agent, it is possible to merge the clouds.
The Pseudo Code 3 shows how this is done. The libraries *pcl* and *Eigen* have been used in this code.

Algorithm 3 Point cloud merging code. It subscribes to two point clouds, it applies transformations and it concatenates them.

procedure CLASS DEFINITION(PointCloudMerge)
    ▷ ROS subscriber initialization: PointCloud2 n.1
    ▷ ROS subscriber initialization: PointCloud2 n.2
    ▷ ROS publisher initialization: PointCloud2
    $cloud1, cloud2, cloud\_final$ ▷ PointXYZ data structure
    $m1$ ▷ Eigen, Matrix4f data structure
    ▷ The following variables are the position and orientation of the second agent starting point wrt the global frame. The global frame coincides with the first agent starting point
    $x\_static \leftarrow 0$
    $y\_static \leftarrow -0.5$ ▷ (meters)
    $z\_static \leftarrow 0$
    $roll\_static \leftarrow 0$
    $pitch\_static \leftarrow 0$
    $yaw\_static \leftarrow 0$

```
14:     procedure CLOUD1_CALLBACK(*PointCloud2 msg)
15:         procedure FROMROSMSG(*msg, cloud1)
16:         end procedure
17:         ▷ pcl library function: converts PointCloud2 message in pclXYZ format
18:         cloud_final.header.frame_id ← msg− > header.frame_id
19:         cloud_final.header.seq ← msg− > header.seq
20:         cloud_final.header.stamp ← msg− > header.stamp
21:     end procedure
22:     procedure CLOUD2_CALLBACK(*PointCloud2 msg)
23:         procedure FROMROSMSG(*msg, cloud2)
24:         end procedure
25:     end procedure
26: end procedure
27: procedure MAIN(args)
28:     ▷ Node initialization and cleanup
29:     ▷ Call to class
30:     ▷ Node run (ROS spin)    ▷ cloud1 and cloud2 subscriber    ▷ cloud_final
    publisher
31:     transformed_cloud
32:     concatenated_cloud
33:     ms ← Identitymatrix
34:     ▷ Eigen, Matrix4f structure ▷ Static transform from reference frame n.2 to
    reference frame n.1
35:     procedure TRANSFORMPOINTCLOUD(cloud2,transformed_cloud,ms)
36:     end procedure
37:     concatenated_cloud = cloud1
38:     concatenated_cloud += transformed_cloud
39:     ▷ cloud_final publish
40: end procedure
```

The obtained solution starts from the images and it produces the fused point cloud. This is the final obtained map. The inclusion on the SLAM package RTAB-Map, as discussed before, happens between two phases. Once the map has built and stored in a PointCloud2 message, it is enough to run instantaneously a RTAB-Map node to convert it in a database file. From that moment, it is possible to use the produced map. Anyway, as it will be clarified in the next chapter, the user must take care of the fact that a minimum density is required for the point cloud, otherwise during the two phases, it might happen that the package can't find matches between the database map and the new data.

Chapter 3

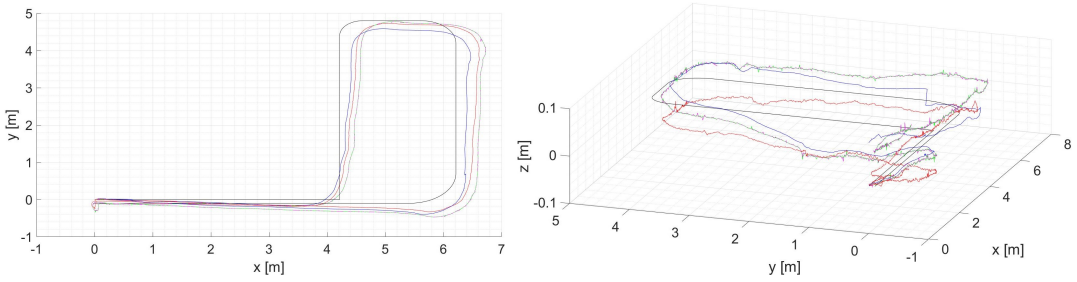
Data analysis and Simulation results

In this chapter, the results of the package implementation will be analyzed and discussed, by highlighting the perks and the drawbacks of the implemented solution. Finally, a series of possible future developments will be presented.

3.1 The localization problem

3.1.1 Results

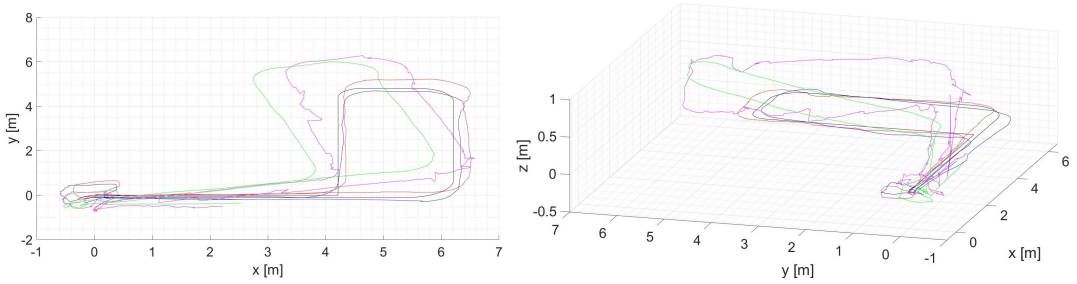
As shown in 2.1.3, it is possible to insert in the SLAM for each drone a sensor fusion strategy, which can fuse odometry and inertial data.
The mathematical relevance of the Kalman Filter is helpful in the operation and, among the open-source solutions, it still looks like the best possible option, since it provides an excellent balance between set-up and usage complexity and efficiency. Some tests have shown that the possibility of enhancing the VIO performances is, actually, possible. Figure 3.1 shows an example in which an EKF improves the localization process.



(a) X-Y plane (b) 3D representation

Figure 3.1: Example of situation in which the sensor fusion is effective in improving the localization process.

Anyway, the tests have only shown the effects of the fusion on two sources of localization data that are, by themselves, already reliable.
Even in this case, it is not always guaranteed that the sensor fusion package is worth using. It requires the presence of other sensors, which in the drone situation means increased weight and volume.
On top of this, there's the computational effort issue: even if sensor fusion packages are not too demanding, the onboard computer must guarantee the necessary power to run independent SLAM running along with all the features needed to use the sensor suite and allowing the drone to fly.
Finally, there's the issue that the employed package needs a lot of tuning and it doesn't always guarantee good performances.



(a) X-Y plane (b) 3D representation

Figure 3.2: Example of a situation in which the sensor fusion is not effective, and it makes things worse.

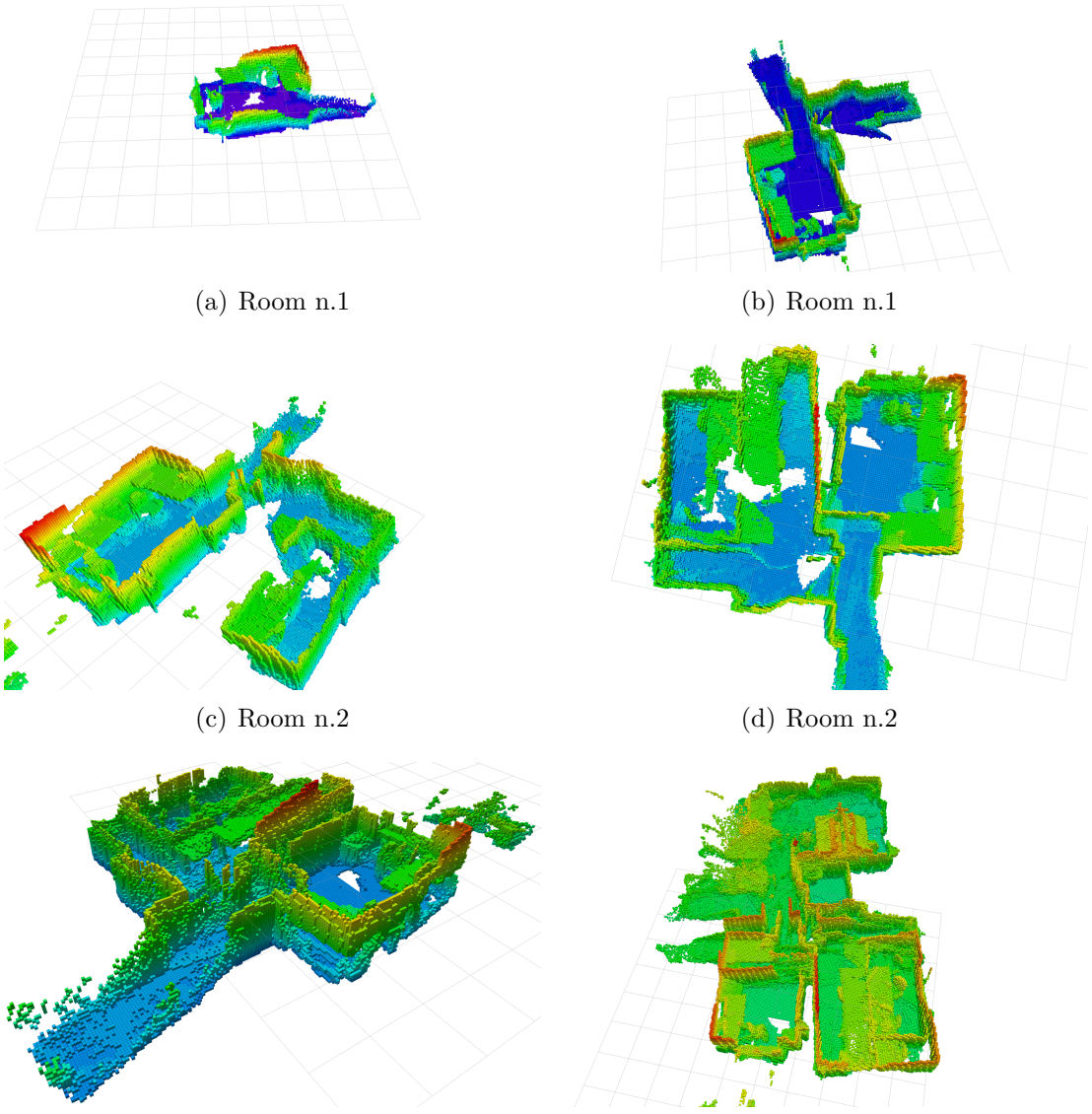
3.1.2 Future Developments

The presence of a sensor fusion package can improve both the accuracy and the reliability of the localization system.
The investigation of this kind of solution is indeed important, and it should be done starting from the package **robot_localization** and the sensor suite.
The latter opens to a lot of possibilities, like usage of multiple IMUs systems, or usage of supplementary localization devices.
Another possibility might be fusing the localization output of a SLAM or VIO package with another localization source, to improve the information as much as possible.
The package exploration, however, might be as helpful, since a good tuning might guarantee a certain consistency among the results and make the filter work properly.
All these developments could be summarized in the search of a trade-off between performances, computational effort, weight and volume, cost.

3.2 The Single-agent SLAM problem

3.2.1 Results

The proposed solution shows that, at the state-of-the-art, RTAB-Map is still one of the most reliable systems for mobile robot solutions that don't depend on ground stations for heavy operations.
Indeed, the localization part is accurate enough to use the data for navigation purposes.
Finally, there's the important feature of the map building: if compared with many SLAM packages, the map produced by RTAB-Map is not only easier to read and analyze thanks to the octomap tools, but also very detailed. Figure 3.3 shows an example of exploration of an office environment made of three rooms. It is easy to see how well-detailed is the environment and the objects that are part of it.
The map has been represented in **rviz** with an height limitation, since the presence of the ceilings prevents from a clear view.



(a) Room n.1

(b) Room n.1

(c) Room n.2

(d) Room n.2

(e) Room n.3

(f) Room n.3

Figure 3.3: Example of exploration of an office environment with RTAB-Map. The octomap is represented with the axis-color code.

3.2.2 Future development

The system uses RTAB-Map in a configuration in which it is used the tracking information of the T265 as an odometry topic and the depth and color image of the D435 camera.
For starters, there's the possibility to consider a system that still employs RTAB-Map, but it uses a VIO algorithm instead of the T265 camera.
This kind of solution is not too hard in the implementation, since RTAB-Map is easily configurable with many famous VIO packages, and once installed, a single *.launch* file is enough to run the entire SLAM system.
Other than RTAB-Map, there's the possibility to use different SLAM and VIO packages: as mentioned in Section 1.2.2, there are some packages that can run on onboard computers without the usage of a ground station.
Therefore, it is possible to test the SLAM performances on a single agent with a different SLAM approach.

3.3 The Mutli-agent problem

3.3.1 Results

The proposed solution enforces the capability of the system to map an environment with a fleet of drones. The system guarantees the independence of each agent SLAM process, but at the same time, it guarantees the possibility to fuse data coming from different drones.
The great advantage of this solution consists in its portability, i.e. the possibility to use a multi-agent logic even on low-budget embedded computers.
Moreover, it is important to remember that this solution does not depend on RTAB-Map, but it is easily editable in order to adapt to another package.
The following paragraphs will show the capability of the system of point cloud merging.
The tests illustrated in Figures 3.5,3.6, and 3.7 show the evolution of the output of a single point cloud, as the output of the **octomap_server** package.
They are performed in an office environment, where there are some desks and some pieces of furniture. Figure 3.4 represents a sketch map of the environment, other than the starting point of two agents, represented by a blue and red dot, accordingly to the color of the respective clouds.

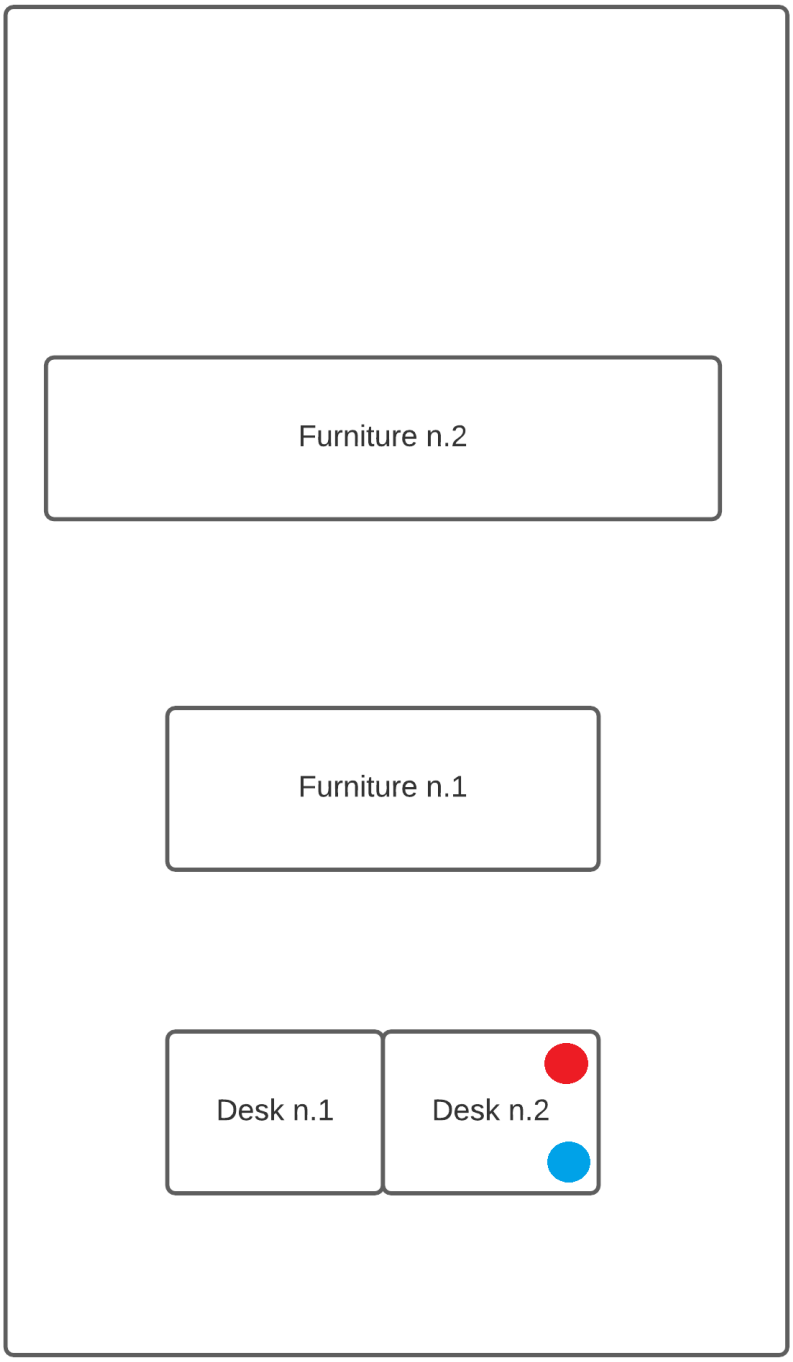
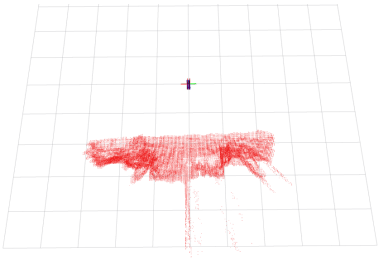
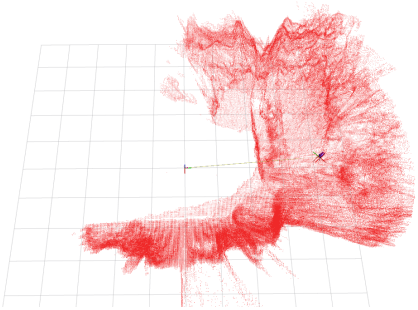


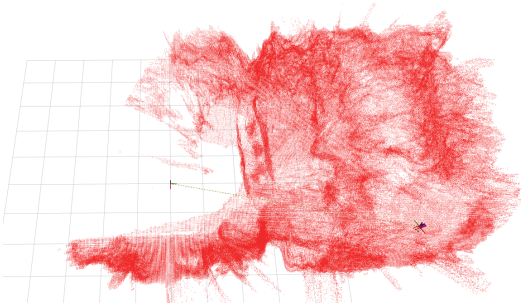
Figure 3.4: Sketch diagram of the office environment. Blue and red dots represent the starting point of the two involved agents.



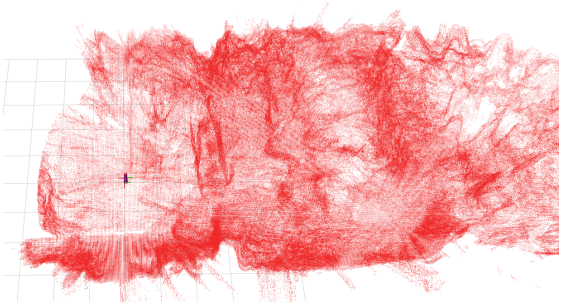
(a) Stage 1



(b) Stage 2

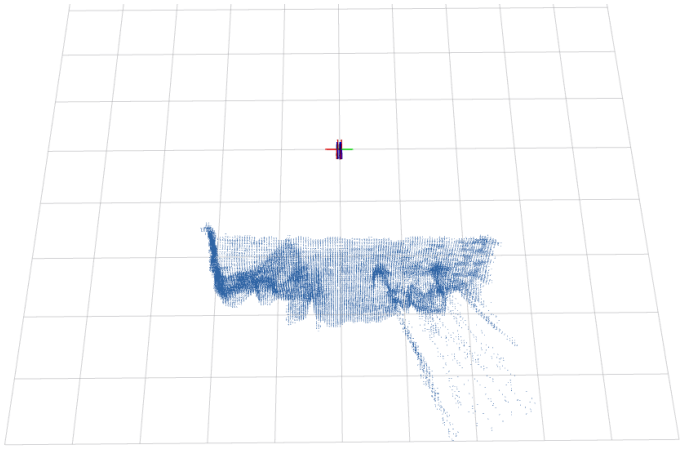


(c) Stage 3

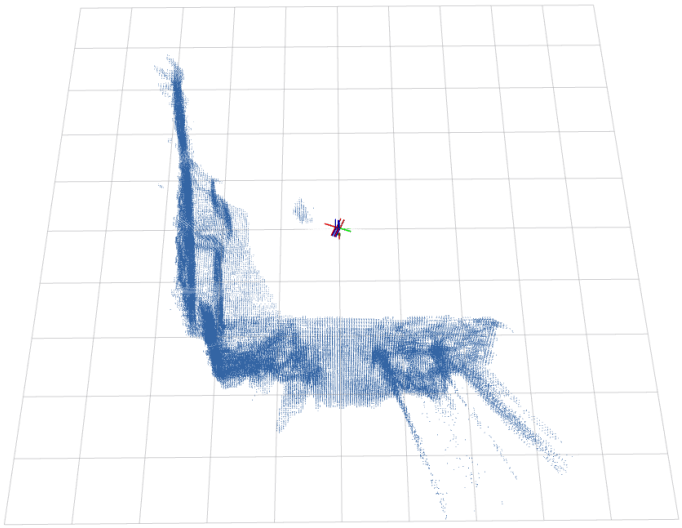


(d) Stage 4

Figure 3.5: Example of environment exploration for a multi-agent system: the red cloud is the first agent map

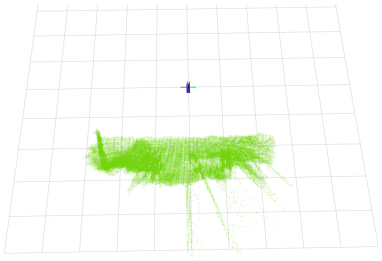


(a) Stage 1

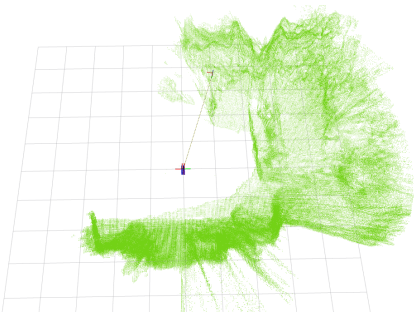


(b) Stage 2

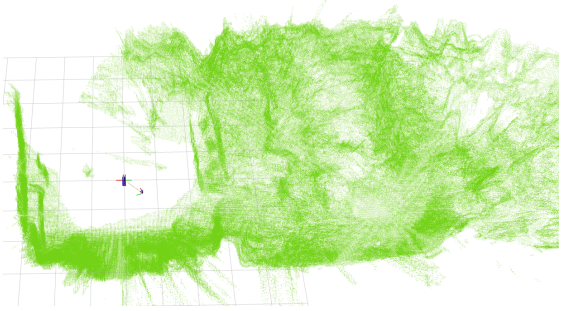
Figure 3.6: Example of environment exploration for a multi-agent system: the blue cloud is the second agent map.



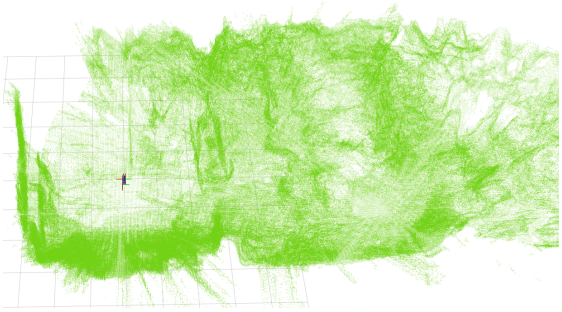
(a) Stage 1



(b) Stage 2



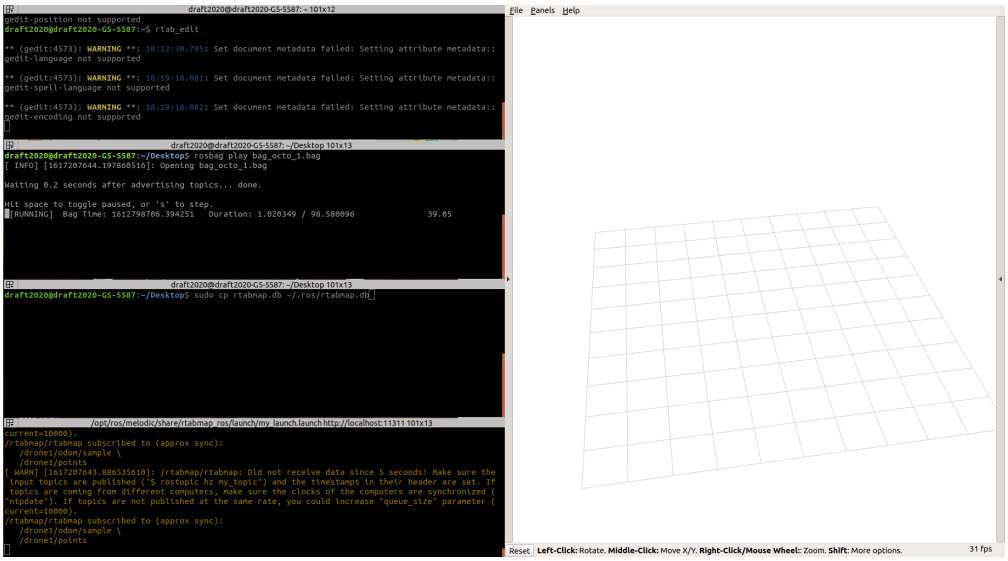
(c) Stage 3



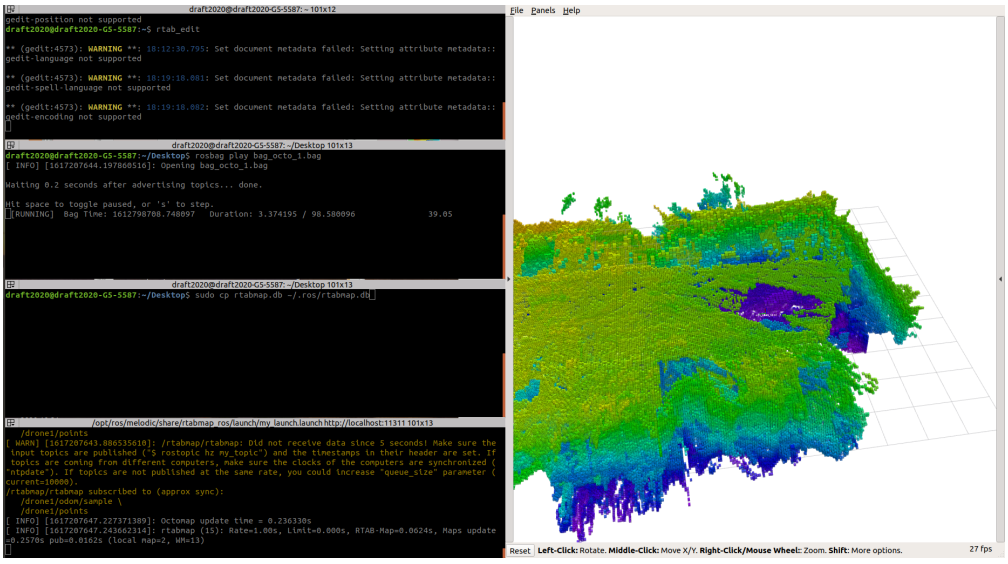
(d) Stage 4

Figure 3.7: Example of environment exploration for a multi-agent system: the green cloud is the server cloud.

The results show that the map merging is effective and the global map is accurate. As it will be shown soon, RTAB-Map can use this map as a database file and it can use it properly.
Figure 3.8 illustrates the "wake-up" phase of RTAB-Map, by showing that the load is possible and it works.



(a) Starting ROS bag



(b) Starting ROS bag

Figure 3.8: RTAB-Map "wake-up" phase: the packages takes less than three seconds to pre-load the map

Of course, it is important to specify that the package displays an error if the cloud is not dense enough: a matching between the real-time data and the pre-loaded ones is possible only if the density is sufficiently high, otherwise, the loaded map will be discarded because it is not consistent with the actual data.
Finally, Figure 3.9 illustrates the octomap version of the pre-loaded map.

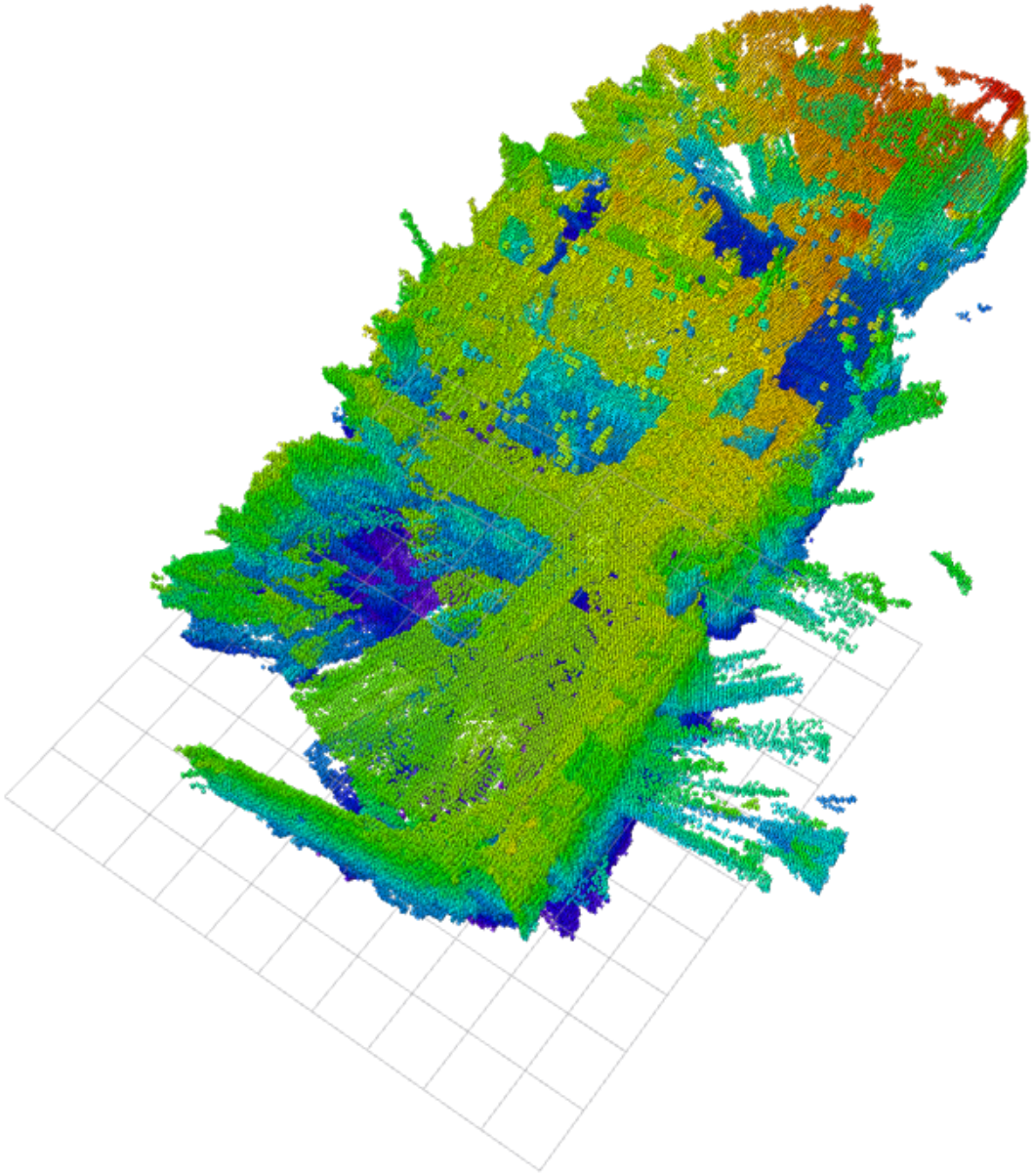


Figure 3.9: RTAB-Map loads the octomap version of the merged map stored in the database file. This is the loaded map.

3.3.2 Future Developments

The advantage of this system consists in its simplicity and in the fact that it does not require particularly expensive hardware components, which means that the computational effort is quite limited.
Anyway, this happens because the described approach does not include a lot of the features typical of SLAM systems.
First of all, there is the real-time problem: the described system works well in a two-phase mission, but it cannot be properly applied in a single-phase one.
One of the possible developments should be the definition of a solution that guarantees real-time usage: the ideal multi-agent SLAM package should be able to take advantage of the presence of many agents to enhance the performances of each single one of them.
Realizing a system like this means that the single-agent package should be properly modified to combine local and global information.
As explained before, many SLAM packages present a series of executables and libraries in a very nested structure, therefore it is no easy task to apply changes.
The easiest way might be to start from an existing visual odometry approach and then proceed by developing most of the package features.
Anyway, this possibility opens up to many other features typical of SLAM packages: to improve both phases it is important to apply loop closure, bundle adjustments, map correction, and other tools appropriately.
The presented system can fuse two or more maps, but it still lacks a correction phase, which could improve the performances.

Chapter 4

Conclusions

The Multi-agent SLAM problem is becoming more and more actual, especially due to the progress on autonomous vehicles, ground robots, and drones.
Some fields in which groups of robots are employed already exist, and new ones are emerging: many systems are based on the idea of a swarm that can interact and shares information without any human interaction. Localization and mapping data are perfect examples of shareable information.
Even if solutions to SLAM for a single-vehicle exist, the research is keeping going further, by trying to improve the capabilities of already existent approaches or trying new ones. It appears that the common factor in all the state-of-the-art models is the vision approach: the usage of advanced cameras allows to get small, but performing systems.
Thus, the aim of this kind of research is, basically, to use the sensor suite at its fullest in order to allow a robot to navigate by itself, even in an unknown environment.
The multi-agent problem is different both in the research and in the aim for many reasons: for starters, there's the fact that the higher is the number of robots that the user wants to include in a mission, the higher is the overall price, since, other than the cost of the single agent, it must include a strong ground control station and a powerful communication interface.

The objective of this thesis is to explore the available solutions for the SLAM problem on a single agent, to pick the best one in terms of a trade-off between performances and computational effort, and to use it as a starting point for a multi-agent system that can enhance the classic SLAM performances, by keeping the overall system simple and cheap.
The tests have shown that a low-budget embedded computer, alongside a state-of-the-art visual and inertial sensor suite, can provide satisfying results.
The presented solution has great possibilities to be adapted to any kind of fleet, even one that combines ground robots and aerial robots.
Moreover, it can be used for systems in which not all the agents have the same hardware structure.
Another perk lies in the fact that there's no permanent link to a certain SLAM package, but there's the possibility to adapt the package to many different solutions.
Anyway, it is important to specify that each agent is fully capable of operating by itself: this means that if the connection with the control ground station is lost, the agent can face the SLAM problem by itself.
The tests have shown that data sharing is possible even with tight-bandwidth connection if data are manipulated properly: image compression and reconstruction is an important feature in this context, and it has shown great results.
Anyway, there still are a lot of issues to solve, especially related to the fact that the package developed in this project cannot guarantee multi-agent SLAM performances in a single-phase mission.
Indeed, the map merging has proven to work properly, but some advanced map updates should be evaluated, along with the fact that the package can work only if the initial position of each agent is known with respect to a global reference frame.
Finally, it can be observed that the possibility of creating a multi-agent SLAM system with this logic is a concrete possibility and it could open up to a reliable system for many kinds of missions, from indoor tasks as in logistics to outdoor missions, related both to exploration and security.

Bibliography

[1] Dongjiang Li, Xuesong Shi, Qiwei Long, Shenghui Liu, Wei Yang, Fangshi Wang, Qi Wei, and Fei Qiao. «DXSLAM: A Robust and Efficient Visual SLAM System with Deep Features». In: *arXiv preprint arXiv:2008.05416* (2020) (cit. on p. i).

[2] Devendra Singh Chaplot, Dhiraj Gandhi, Saurabh Gupta, Abhinav Gupta, and Ruslan Salakhutdinov. «Learning To Explore Using Active Neural SLAM». In: *International Conference on Learning Representations (ICLR)*. 2020 (cit. on p. i).

[3] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. «Visual SLAM algorithms: a survey from 2010 to 2016». In: *IPSJ Transactions on Computer Vision and Applications* 9 (Dec. 2017). DOI: 10.1186/s41074-017-0027-2 (cit. on p. 8).

[4] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. «Real-Time Loop Closure in 2D LIDAR SLAM». In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278 (cit. on p. 16).

[5] G. Grisetti, C. Stachniss, and W. Burgard. «Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters». In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 34–46. DOI: 10.1109/TRO.2006.889486 (cit. on p. 17).

[6] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. «A Flexible and Scalable SLAM System with Full 3D Motion Estimation». In: *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE. Nov. 2011 (cit. on p. 17).

[7] *MRPT: official link*. URL: https://www.mrpt.org/ (cit. on p. 18).

[8] Tong Qin, Jie Pan, Shaozu Cao, and Shaojie Shen. *A General Optimization-based Framework for Local Odometry Estimation with Multiple Sensors*. 2019. eprint: arXiv:1901.03638 (cit. on p. 18).

[9] Tong Qin, Shaozu Cao, Jie Pan, and Shaojie Shen. *A General Optimization-based Framework for Global Pose Estimation with Multiple Sensors*. 2019. eprint: arXiv:1901.03642 (cit. on p. 18).

[10] Raúl Mur-Artal and Juan D. Tardós. «ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras». In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262. DOI: 10.1109/TRO.2017.2705103 (cit. on p. 19).

[11] Carlos Campos, Richard Elvira, Juan J. Gomez, José M. M. Montiel, and Juan D. Tardós. «ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM». In: *arXiv preprint arXiv:2007.11898* (2020) (cit. on p. 19).

[12] Taihú Pire, Thomas Fischer, Javier Civera, Pablo De Cristóforis, and Julio Jacobo berlles. «Stereo Parallel Tracking and Mapping for robot localization». In: *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 1373–1378. DOI: 10.1109/IROS.2015.7353546 (cit. on p. 19).

[13] Taihú Pire, Thomas Fischer, Gastón Castro, Pablo De Cristóforis, Javier Civera, and Julio Jacobo Berlles. «S-PTAM: Stereo Parallel Tracking and Mapping». In: *Robotics and Autonomous Systems (RAS)* 93 (2017), pp. 27–42. ISSN: 0921-8890. DOI: 10.1016/j.robot.2017.03.019 (cit. on p. 19).

[14] Mathieu Labbé and François Michaud. «RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation: LABBÉ and MICHAUD». In: *Journal of Field Robotics* 36 (Oct. 2018). DOI: 10.1002/rob.21831 (cit. on p. 19).

[15] J. Delmerico and D. Scaramuzza. «A Benchmark Comparison of Monocular Visual-Inertial Odometry Algorithms for Flying Robots». In: (2018), pp. 2502–2509. DOI: 10.1109/ICRA.2018.8460664 (cit. on p. 27).

[16] Alexandre Alapetite, Zhongyu Wang, John Paulin Hansen, Marcin Zajączkowski, and Mikołaj Patalan. «Comparison of Three Off-the-Shelf Visual Odometry Systems». In: *Robotics* 9.3 (2020). ISSN: 2218-6581. DOI: 10.3390/robotics9030056. URL: https://www.mdpi.com/2218-6581/9/3/56 (cit. on p. 27).

[17] Tully Foote. «tf: The transform library». In: Open-Source Software workshop (Apr. 2013), pp. 1–6. ISSN: 2325-0526. DOI: 10.1109/TePRA.2013.6556373 (cit. on p. 32).

[18] Hornung A., Wurm K. M., Bennewitz M.and Stachniss C., and Burgard W. «OctoMap: an efficient probabilistic 3D mapping framework based on octrees». In: (Feb. 2013), pp. 189–206. DOI: 10.1007/s10514-012-9321-0 (cit. on p. 49).

[19] M.Labbé. *RTAB-Map: ROS wiki*. URL: `http://wiki.ros.org/rtabmap_ros/Tutorials/SetupOnYourRobot#Remote_mapping` (cit. on p. 51).

[20] N.Rutigliano C.Caputi. *RTAB-Map: test in PIC4SeR research Lab*. URL: `https://www.youtube.com/watch?v=mV2zqIpptpY` (cit. on p. 53).

[21] Anders Grunnet-Jepsen Tetsuri Sonoda. *Depth image compression by colorization for Intel® RealSense™ Depth Cameras*. URL: `https://dev.intelrealsense.com/docs/depth-image-compression-by-colorization-for-intel-realsense-depth-cameras` (cit. on p. 56).

[22] Eduardo S. L. Gastal and Manuel M. Oliveira. «Domain Transform for Edge-Aware Image and Video Processing». In: *ACM TOG* 30.69 (2011). Proceedings of SIGGRAPH 2011, 69:1–69:12 (cit. on p. 58).