Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE GRAU

TÍTOL DEL TFG: Navegació autònoma per quadròpters voladors usant VSLAM i intel·ligència artificial.

TITULACIÓ: Enginyeria tècnica aeronàutica, especialitat Aeronavegació

AUTOR: Pere Coll Fernández

DIRECTOR: Juan López Rubio

DATA: 04/08/2020

Títol: Navegació autònoma per quadròpters voladors usant VSLAM.

Autor: Pere Coll Fernández

Director: Juan López Rubio

Data: 04/08/2020

Resum

Quan es tracta de robots que es desplacen, hi ha un gran inconvenient en la navegació autònoma, ja que afegir el sentit de la vista a una màquina pot ser molt complicat. Hi ha altres sensors que poden ajudar a un robot a interactuar amb l’entorn, com els sensors de distància d'ultrasons, utilitzats per evitar col·lisions amb objectes o sensors de pressió, per a conèixer l’altura del robot. A més, els mètodes de navegació amb GPS poden ser molt útils per gestionar els moviments del robot en un espai obert de manera més o menys precisa.

Tanmateix, per a l'aplicació en interiors on el senyal GPS és deficient o inexistent, hem de confiar plenament en les dades dels nostres sensors. Un sensor que, per a aquest projecte, serà una càmera 2D sense cap informació de profunditat.

En aquest document estudiarem el Sistema LSD-SLAM amb integració de la tecnologia ROS. Totes les proves es realitzaran usant el simulador Gazebo, en un espai virtual. Tots els procediments seguits en aquest projecte han sigut provats en Ubuntu 16.04.

Al llarg d’aquest document es descriuen tots els passos seguits per instal·lar i configurar la tecnologia ROS, juntament amb el simulador esmentat anteriorment i el sistema LSD-SLAM. A més a més, es duran a terme unes proves de vol en espais interiors i exteriors on comprovarem l’efectivitat de l’algorisme. Aquesta serà quantificada i comparada entre els dos escenaris.

Tots els paquets necessaris per poder reconstruir el projecte els podeu trobar a les referències del mateix.

Title: Autonomous navigation for flying quadcopters using VSLAM.

Author: Pere Coll Fernández

Director: Juan López Rubio

Date: 04/08/2020

Overview

When it comes to moving robots, there is a major drawback in autonomous navigation, as adding the sense of sight to a machine can be very complicated. There are other sensors that can help a robot interact with the environment, such as ultrasonic distance sensors, used to prevent collisions with objects or pressure sensors, to know the height of the robot. In addition, GPS navigation methods can be very useful for managing robot movements in an open space more or less accurately.

However, for indoor applications where the GPS signal is deficient or non-existent, we must fully rely on the data from our sensors. A sensor that, for this project, will be a 2D camera without any depth information.

In this paper we will study the LSD-SLAM System with integration with ROS technology. All tests will be performed using the Gazebo simulator, in a virtual space. All procedures followed in this project have been tested on Ubuntu 16.04.

This document describes all the steps to install and configure ROS technology, along with the aforementioned simulator and the LSD-SLAM system. In addition, flight tests will be carried out indoors and outdoors where we will check the effectiveness of the algorithm. This will be quantified and measured between the two scenarios.

All the packages needed to be able to rebuild the project can be found in the references.

INDEX

CHAPTER 1. INTRODUCTION AND OBJECTIVES 9

1.1. Introduction 9

1.2. Objectives 9

CHAPTER 2. INITIAL DESCRIPTION 12

2.1. The rise of the UAVs..12

2.2. Visual Odometry (VO)..14

2.3. The beginning of the SLAM techniques 15

2.4. The Robot Operating System (ROS)..16

2.4.1. Description...16

2.4.2. Ubuntu Installation 18

CHAPTER 3. WORKING ENVIRONMENT 20

3.1. Gazebo 20

3.1.1. Description...20

3.1.2. Integration with ROS 21

3.1.3. Installation Commands...21

3.2. The Parrot AR drone..24

3.2.1. Description...24

3.3. AR drone Implementation for Gazebo 26

3.4. ROS development Studio 30

CHAPTER 4. THE LSD-SLAM TECHNOLOGY 31

4.1. LSD-SLAM: Large Direct Monocular SLAM..31

4.1.1. Description...31

4.1.2. The LSD-SLAM Algorithm 31

4.1.3. LSD-SLAM for ROS Kinetic...33

4.1. LSD-SLAM running and testing 34

4.1.1. Simulation Steps 34

4.2. Testing in a Simulated Environment..37

4.2.1. The Word Scale Ambiguity Problem 37

4.2.2. Path and environment setup..40

4.2.3. Simulation Results 43

4.3. Results..47

4.3.1. Performance Analysis 47

CHAPTER 5. CONCLUSIONS AND FUTURE WORK 50

5.1. Conclusions..50

5.2. Future Work 51

BIBLIOGRAPHY 52

LIST OF FIGURES

Illustration 1 - Reaper UAV 12
Illustration 2 - Control room for the Reaper Drone 12
Illustration 3 - Phantom 4 Pro design 13
Illustration 4 - Visual Odometry Visualization 14
Illustration 5 - Google driverless car or Waymo 15
Illustration 6 - Robot Operating System (ROS) logo 16
Illustration 7 - Gazebo Simulator 20
Illustration 8 - Empty Gazebo Simulation 22
Illustration 9 - Available ROS topics 23
Illustration 10 - Parrot AR done 2.0 with indoors configuration 24
Illustration 11 - All parts of the AR drone 2.0 25
Illustration 12 - Directory tree structure at this point for our workspace 27
Illustration 13 - Gazebo Simulation with the Ardrone 28
Illustration 14 - Teleop keyboard control keys 29
Illustration 15 - ROS development studio interface 30
Illustration 16 - Block diagram showing the LSD-SLAM algorithm 32
Illustration 17 - LSD-SLAM obtained point cloud 36
Illustration 18 - Simulated environment 36
Illustration 19 - Scaling factor problem visualization 37
Illustration 20 - Height data comparison vs IMU data and vSLAM system 1. 38
Illustration 21 - Height data comparison vs IMU data and vSLAM system 2. 39
Illustration 22 - Height data comparison vs IMU data and vSLAM system in different environment 39
Illustration 23 - Experimental path 1 40
Illustration 24 - Experimental path 2 40
Illustration 25 - Experimental path 3 40
Illustration 26 - Test Environment 1: Outdoors Scenario 42
Illustration 27 - Test Environment 2: Indoors Scenario 42
Illustration 28 - Path 1 results (3d view) 44
Illustration 29 - Path 1 results (x-y view) 44
Illustration 30 - Path 2 results (3d view) 45
Illustration 31 - Path 2 results (x-y view) 45
Illustration 32 - Path 3 results (3d view) 46
Illustration 33 - Path 3 results (x-y view) 46
Ilustration 34 - LSD-SLAM system mapping and tracking visualization 47
Illustration 35 - Mean Absolute Deviation Formula 47
Illustration 36 - MAD error comparison (m) 48
Illustration 376 – Max error distance comparison (m) 49

CHAPTER 1. INTRODUCTION AND OBJECTIVES

1.1. Introduction

Sight or vision is probably one of the most relevant senses when talking about how humans interact with the environment. Especially when walking or moving, having an image of the surrounding area will be crucial for evaluating not only our position, but also the next movements that we need to perform to complete a certain objective.

When it comes to moving robots there is a big challenge in autonomous navigation, as adding sight to a machine can be very challenging. There are other sensors that can help a robot interact with the environment such as ultrasonic distance sensors, used to avoid collisions with objects, or pressure sensors, to know the robot height. Also, methods as GPS guidance can be very useful when managing the robot's movements in an open space area in a more or less accurate way.

However, for indoors application where there GPS signal is poor or inexistent, we need to fully rely on our sensors data. A sensor, that for this project is going to be a camera without any depth information, just the plane pixel data.

All the testing and implementations will be done using a virtual environment in Gazebo simulator and using ROS technology. All the procedures and steps followed in this project have been tested in Ubuntu 16.04.

1.2. Objectives

The main objective of this project is to research and implement a Visual Slam system called LSD-SLAM and analyse its viability for the use in drones for autonomous navigation in indoors space. Also, it will be studied if the use of simulated environments could be a good option for demoing and learning about the self-navigation in drones.

The task of processing and tracking video frames is commonly a very power consuming task and takes lots of processing time. The chosen SLAM system is one of the lightest available and therefore can be used in smaller drones. This make it even more attractive for indoors navigation, as the drone may need to fly through tiny areas.

Another important objective for the project is to evaluate the SLAM system in indoors and outdoors environments. This way it will be possible to numerically determine the algorithm precision in both environments and obtain conclusions about the impact of having more and closer objects or facing an open space.

The installation of all the technologies used in the project may sometimes be difficult. Building an environment that will work with all the programs needed will also be a challenge. It will be explored the best solution for the implementation of ROS technologies along with the simulator and the SLAM system.

This project will also document all the steps and procedures needed to install and setup the working environment. It will also be explained the usage of the components of the system.

CHAPTER 2. INITIAL DESCRIPTION

In this chapter we will be taking about the concepts needed to understand the project. We will go through the evolution of the drones, we will introduce the Visual Odometry concept and the SLAM techniques. Finally, we will have a look into the Robot Operating System.

2.1. The rise of the UAVs

Like many other current technologies, the reason why the UAV's were developed was for military purposes. This new technology led to the creation of the quadcopters, that nowadays anybody can own as a toy or experimental tool for a relatively small amount of money.

Illustration 1 - Reaper UAV

The word UAV means Unmanned Aerial Vehicle, which can be driven autonomously or remotely piloted by one or more operators. This is very useful and has many functionalities as surveilling war areas with dangerous access or launching missies without risking the life or the aircraft pilot.

Illustration 2 - Control room for the Reaper Drone

As mentioned above, the fast technology evolution has brought these vehicles to evolve to a commercial use, developing small multirotor vehicles such as the parrot AR drone family [1], that is a quadcopter with four motors mainly designed for development purposes, or the Phantom family [2], designed for high quality footage.

Illustration 3 - Phantom 4 Pro design

Nowadays flying quadcopters are used for many applications: from sending medical equipment to areas with difficult access, to recording videos of ourselves doing sport. In this project we will be looking at flying quadcopters as flying cameras, because we will be using them as another sensor for the robot.

When referring to indoor flight, those robots can help analysing the structure of a building after a natural disaster hits, mapping buildings inside for architecture, planning factories space distribution… among many others. All those tasks have a thing in common: they require a skilled human pilot controlling the robot at all times.

In this project we will be looking SLAM techniques in order to improve the flying autonomy of these quadcopters in indoor flight and relieve stress from the pilot.

2.2. Visual Odometry (VO)

Visual Odometry is the use of a camera to estimate its position and orientation in real time by observing a sequence of images of its environment. This has a wide variety of applications in robotics and it was first introduced for planetary rovers operating in Mars.

In navigation, Odometry is used to estimate the position of the vehicle over time by analysing the data from the motion sensors. The sensors need to be very accurate and well calibrated in order to get a proper positioning. For large periods of time this positioning method can lead to large errors due to the integration of velocity over time.

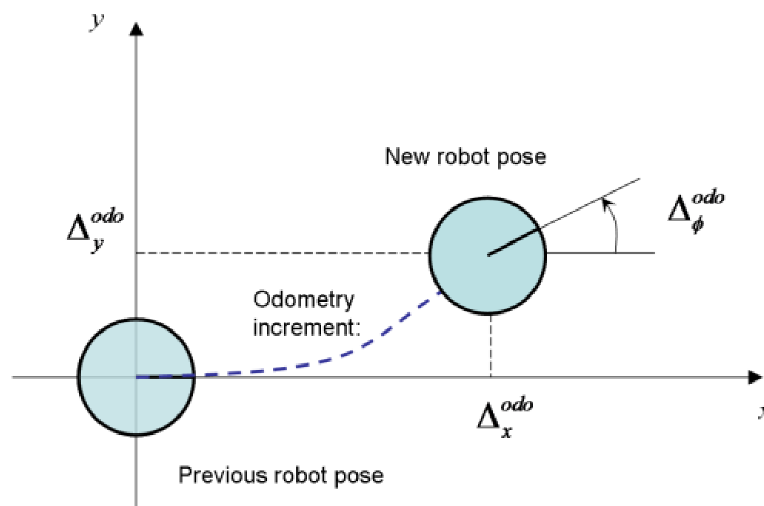

Illustration 4 - Visual Odometry Visualization

The word Odometry has been taken from the Greeks meaning “Route Measure”.

As we will be using visual Odometry for this project we need to look into different types of data sensors, in our case cameras. In recent years the VO methods proposed can be divided into monocular or stereo cameras. A stereo camera such as an RGB-Depth camera has a sensitive device that provides the depth information about an image. A monocular camera returns colour information within each pixel.

2.3. The beginning of the SLAM techniques

The Simultaneous Location and Mapping (SLAM) techniques aim to evaluate the state of a robot equipped with the on-board sensors, while building and updating a 3D map of the environment. This process can run in real time or using datasets. SLAM techniques are very useful on spatial missions, undersea or underground explorations and indoors navigation.

The state of the robot is described by its pose, composed by the position and orientation although there may also be additional parameters such as the linear or angular velocity. The 3D map is created by tracking the corners of the objects and landmarks [3].

A huge challenge when working with monocular cameras is the estimation of the world scale, as it cannot be observed and will drift all the time creating an important source of error. On the other side, this leads to an important advantage, which is that no matter the size of the environment you are working in, allowing it to work in small rooms or in large-scale outdoors scenarios. It’s important to notice that stereo cameras have a limited range, making it useless for large environments.

Illustration 5 - Google driverless car or Waymo

In this project, as we will be looking on camera sensors the SLAM is going to be visual (vSLAM) although it can also be accomplished by using distance sensors. A very interesting application for vSLAM is augmented reality, as it’s becoming very popular in the mass-market.

2.4. The Robot Operating System (ROS)

The robot Operating System is a flexible framework for writing robot software. It's important to say that ROS is not a real operating system as it runs on top of a Linux Ubuntu. It was developed in 2007 in the Artificial Intelligence laboratory in Stanford and has continuously been under development. In the following pages we will see some definitions and the needed steps for its installation.

2.4.1. Description

ROS technology offers a set of libraries and tools to help software developers create robotics applications. It is one of the pioneer Multi-Robot Systems (MRS) that allows the implementation of multiple robots to perform certain tasks together. ROS environment is so flexible that includes multiple platforms, coding languages, compilers and many libraries developed by investigators. All it's open source so any developer can use and modify them in order to improve or create new features. This way, ROS technology is in constant growth.

Due to the constant expansion of ROS, there are many versions available. Some of the latest are:

- ROS melodic (from May, 2018)
- ROS lunar (from May, 2017)
- ROS kinetic (from May, 2016)
- ROS indigo (from July, 2014)

It is important to mention that ROS license in under BSD, which means that it allows commercial or non-commercial use, making it very interesting for single developers or companies.

Illustration 6 - Robot Operating System (ROS) logo

To better understand how this technology works we will look through some basic concepts about ROS features.

- **Package**: The ROS software is organized in packages. These packages would be what it's called projects for common programming languages. A package may contain ROS nodes, configuration files, datasets, ROS-independent library or anything that constitutes a useful module [4]. The packages can be organized as stacks and can be easily downloaded from GitHub repositories and installed as you can see below:

```
cd ~/catkin_ws/src
git clone https://github.com/tum-vision/tum_simulator.git
cd ..
catkin_make
```

- **Node**: A node is an executable file within a package. ROS has been designed to use multiple nodes simultaneously to control the robot. Nodes communicate with other nodes using a client library and can publish or subscribe a topic. As an example, there is a node that controls the camera of the robot.

  The nodes will be the processes that will perform all the computations. ROS nodes aim to build large process with many functionalities split into small and simple processes. This division also helps in being able to distribute the functionalities among different computers in the same system.

- **Topic**: As said above the nodes can publish or subscribe to a topic. The topics are the channel information between nodes, that allow identifying the message and generating the interactions between nodes. For example, the topic */AR drone/front/camera_info,* would emit a topic which would be the camera specifications.

- **Message**: The information emitted by a node that is listened by one or more nodes. The message channel is the topic mentioned above.

  The information between nodes is unidirectional, therefore for a proper communication the message type from the publisher and subscriber must be the same.

In order to start the server and client to allow communication between nodes we need to run the ROS master. This provides naming and registration for all the nodes in the ROS server. This will allow the nodes locate one to each other.

2.4.2. Ubuntu Installation

In this section we will see how to install ROS kinetic in Ubuntu 16.04. It's important to notice that kinetic will only work with Ubuntu 15 (Wily) and 16 (Xenial).

First, we need to set up the source.list in order to allow the ROS packages from the ROS server. The following command feds the source.list:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
-sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

The next step will be to set the keys, to make the repository trusted by Ubuntu.

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --
recv-key 0xB01FA116
```

To install we need to make sure we have the latest version of all Ubuntu packages.

```
sudo apt-get update
```

Now, we can download and install ROS, which in our case will be Kinetic full desktop version. This may take a while to install.

```
sudo apt-get install ros-kinetic-desktop-full
```

This will install the most important ROS packages. The installation of ROS is over, but there are few more steps that will be useful in the future.

In order to install dependencies for ROS packages we need to initialize rosdep as follows:

```
sudo rosdep

init rosdep update
```

It is also very convenient to have the terminal automatically set ROS variables whenever we open it. To do that we will run:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

And to finish, we need to install some more programs in order to be able to build ROS packages we will need in the future.

```
sudo apt-get install python-rosinstall python-rosinstall-generator
python-wstool build-essential
```

As we have seen, installing and setting ROS basics may take some time and it’s somehow challenging if you are not familiar with Linux systems. It may also lead to errors if the steps are not followed carefully.

Each version of ROS is only compatible with few Ubuntu versions, therefore upgrading from an old ROS version to a new one will mean a complete system upgrade that will probably lead to several errors.

Also, using ROS technology in a Windows or MAC computer will lead to a very slow performance and continuous error fixing due to its low development and compatibility. In addition, there are not many forums to seek help and the traffic of those is very poor.

CHAPTER 3. WORKING ENVIRONMENT

The initial intention for this project was to do the testing process using a Parrot Ardrone 2.0 in the real World, but due to the current country situation due to the Covid-19 Pandemics will be entirely carried out in a simulator.

This hazard makes the project more tedious as in the simulated world you need to spend lots of computing time and face many errors making all the components work with the simulated environment.

In this chapter we will describe and explain the software and its installation process for the simulation.

3.1. Gazebo

3.1.1. Description

Gazebo is a 3D simulation system that allows testing the behaviour of robots on a simulated environment. This software also allows to generate robot models and create virtual worlds of indoors or outdoors scenarios. Its physics engines and wide variety of sensors allow the user to mimic the real world [5].

Beside all this, the importance of this simulator is in the fact that can be synchronized with ROS technologies. Therefore, the robots can publish the information from their sensors in the nodes and receive the commands back from the computer.

Illustration 7 - Gazebo Simulator

Gazebo inner functioning runs an Open Dynamics System (ODS), which is a physics engine with a set of high-performance libraries capable of cinematic and dynamic simulation of rigid bodies. For the 3D rendering uses a motor called OGRE.

Having a visual simulation tool is very important so it's easier to test the algorithms and commands sent to the AR drone. Gazebo also contains a simulation for the Ardrone robot so it won't be needed to import it from somewhere else.

3.1.2. Integration with ROS

In order to run ROS technology in gazebo we need to install the meta-package called *gazebo_ros_packages [6]* that contains the following sub-packages:

- gazebo_dev: Provides the configuration of the installed Gazebo version and allows the execution of Gazebo dependencies.

- gazebo_msg: Contains the services (.srv) and messages (.msg) in order to interact with Gazebo form ROS.

- gazebo_plugins: Groups the plugins to use all the sensors, motors and dynamic reconfigurations.

- gazebo_ros: Is the package that wraps the Gazebo server and client and provides the ROS interface for messages, services and configuration.

3.1.3. Installation Commands

The fact that the AR drone 2.0 is already installed in the Gazebo simulator will make the installation steps easier. Despite, we will look through the installation commands and setup for Gazebo and its integration with ROS. It's important to notice that we will be installing Gazebo version 7, as it's the only version that supports LSD-SLAM package along with ROS kinetic.

To start with Gazebo installation, we will follow the same pattern we did in the ROS install, and we will set up the keys and update the source.list. The commands are the following:

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-
stable `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-
stable.list'
```

And for the keys:

```
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key
add -
```

As in ROS install, once we have made sure that Debian database is up to date, we can proceed and install Gazebo 7.

```
sudo apt-get install gazebo7

sudo apt-get install -y libgazebo7-dev
```

To check proper Gazebo install, we can run this command and Gazebo should prompt:

```
gazebo
```

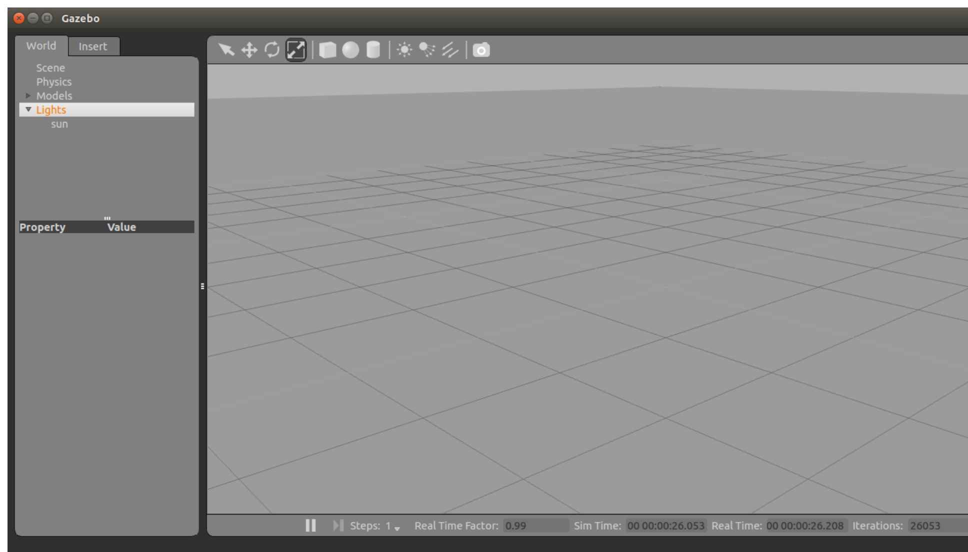

Illustration 8 - Empty Gazebo Simulation

Now that Gazebo and ROS are installed we can proceed with their integration. We will need to set our workspace in order to install the packages. To set our workspace, we will do the following:

```
mkdir -p ~/simulation_ws/src

cd ~/simulation_ws/src

catkin_init_workspace

cd ~/simulation_ws

catkin_make
```

We need to source to the setup script as follows:

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

Now we can download the *gazebo_ros_packages [6]* from GitHub into our workspace src folder and check and install for any missing dependencies using rosdep:

```
cd ~/simulation_ws/src

git clone https://github.com/ros-simulation/gazebo_ros_pkgs.git -b
kinetic-devel

rosdep update

rosdep check --from-paths . --ignore-src --rosdistro kinetic

rosdep install --from-paths . --ignore-src --rosdistro kinetic -y
```

Once the package is ready, we will proceed to build it with the *catkin_make* as follows:

```
cd ~/simulation_ws/

catkin_make
```

The installation is over, and it's time to open gazebo and look for the available ROS topic in order to confirm a proper install.

```
rosrun gazebo_ros gazebo
```

Gazebo should open as before, with now we should see the following available ROS topics when running *rostopic list* command:

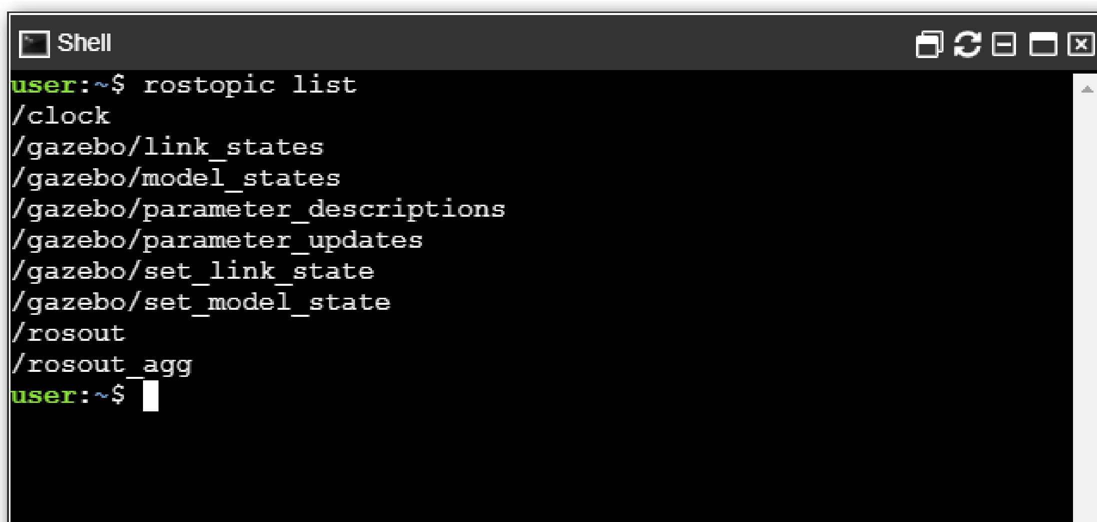

Illustration 9 - Available ROS topics

3.2. The Parrot AR drone

In this section we will describe the drone we have selected for this project. The AR drone has been a great option as it is very affordable in terms of budget, it's small, it's modular and the university owns some of these drones for development purposes.

Moreover, this drone is compatible with ROS technologies so we will be able to simulate in Gazebo and add the packages needed for the LSD-SLAM system.

3.2.1. Description

In this project we will be using the Parrot AR drone 2.0 for the simulations [1]. This model series was first shown on 2010 in the International Consumer Electronics Show with the purpose of civil recreational use. The builder of this drone is the French Parrot company.

The control of this drone is very simple, as it can be operated by a computer but also a mobile or a tablet using an app. It communicates with the operator via WIFI and supports autonomous navigation via GPS.

This model has a configuration of quadcopter (4 electric independent motors) with a very simple modular structure allowing easy and fast modifications of its parts. This structure is made of carbon fibre in a cross shape. To protect the propellers there is a foam structure to cover them that is typically used for indoor flight.

Illustration 10 - Parrot AR done 2.0 with indoors configuration

The weight of the drone in outdoors configuration if of 380g. When the drone is equipped with the indoors protection the total mass of the robot is of 420g. The four brush motors have 14.5W power each and are capable of rotating at 28.500 RPM.

As CPU this drone has an ARM microprocessor with preinstalled linux system and many available sensors. Among these sensors we find two cameras, one pointing downwards and the other looking in the forwards direction.

The forward camera has a resolution of 720p with 92º lens and can record video up to 30fps. The camera pointing downwards has a QVAG 60FPS sensor and it's mainly used for the stabilization of the drone's flight and measuring ground speed.

Apart from the cameras, this robot has an ultrasound sensor pointing downwards that measures the ground altitude. Other sensors such ad an accelerometer, magnetometer, pressure sensor and gyroscope are also included in the drone.

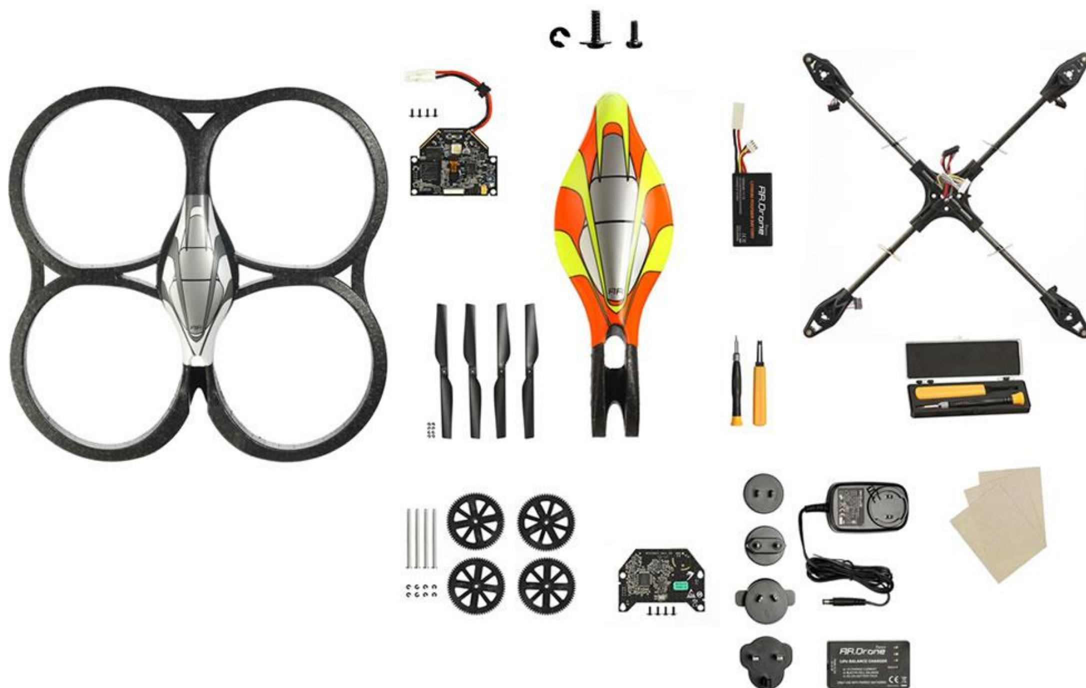

Illustration 11 - All parts of the AR drone 2.0

The battery for this model is of 1000mah or 1500mah, and the approximate flight time (no matter what battery) is approximately of 12 minutes, depending on the conditions of the flight.

3.3. AR drone Implementation for Gazebo

In order to run the AR drone in a Gazebo simulation we will need to install the following two packages:

AR drone autonomy package [7]:

Contains the drivers to fly the quadcopter.

Tum Simulator package [8]:

This package contains the simulation for the AR drone 1.0 and 2.0 to run in the ROS environment using Gazebo Simulator. Contain the following files:

- **cvg_sim_gazebo:** contains the models for objects, sensors and quadcopters.
- **cvg_sim_gazebo_plugins:** contain the plugins for the quadcopter and for its sensors.
- **message_to_tlf:** this package creates the ROS node.
- **cvg_sim_msgs:** contains messages forms for the simulator.

So, with these two packages we will be able to simulate and control an AR drone flying in different scenarios. To install those packages, we need to move to the catkin workspace we previously created. Inside that folder we will clone the two packages form Github. First, we will install the ardrone_autonomy package.

```
cd ~/simulation_ws/src
git clone https://github.com/AutonomyLab/ardrone_autonomy
cd ..
rosdep install --from-paths src --ignore-src
catkin_make
```

After that, we will proceed with the tum_simulator:

```
cd ~/simulation_ws/src
git clone https://github.com/iolyp/ardrone_simulator_gazebo7
cd ..
catkin_make
```

And finally we will source the environment.

```
source devel/setup.bash
```

At this point, our simulation workspace should look like this:

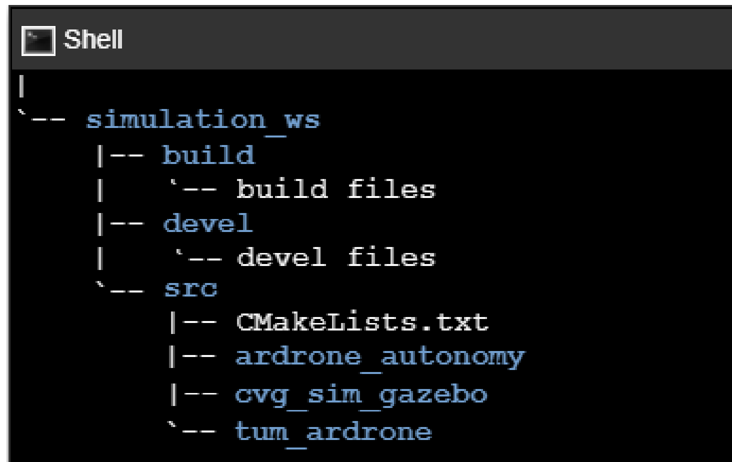

Illustration 12 - Directory tree structure at this point for our workspace

Initial running and testing

Simulations launch commands:

```
roslaunch <package_name> <world_launch_file>

roslaunch cvg_sim_gazebo ardrone_testworld.launch
```

With that command the Gazebo simulation should start and, if both packages are installed correctly, an AR drone 2.0 will appear in the middle of a small village.

Illustration 13 - Gazebo Simulation with the Ardrone

- **Moving the AR drone commands:**

To move the quadcopter in the simulated Gazebo world, we will be looking at two methodologies although there may be others.

Moving by directly sending the commands to the node:

```
#take off

rostopic pub -1 /ardrone/takeoff std_msgs/Empty


#move forward

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{linear:  {x:1.0, y:0.0, z:0.0}, angular: {x:0.0, y:0.0, z:0.0}}'


#land

rostopic pub -1 /ardrone/land std_msgs/Empty
```

Note here, that by modifying the x, y and z values we can move the AR drone in any direction. By changing the angular values, we will make the robot turn. In our case the robot will not turn at any moment, therefore we will only be changing the linear velocity values in order to move but not rotate.

- **Moving by using keyboard commands:**

We will need to install the teleop_twist_keyboard application thought the “apt” command:

```
sudo apt-get install ros-kinetic-teleop-twist-keyboard
```

After that, by interacting with the keyboard the messages will be automatically sent to the correspondent node. Run the keyboard with the following command:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

See the available controls below:

```
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5       turn 1.0
```

Illustration 14 - Teleop keyboard control keys

3.4. ROS development Studio

Setting up ROS can be a very tricky process due to its long installation process and its lack of integrity with other installed packages versions. In addition, there is not much information in the Internet about how to solve those errors [9]. In addition, running a real time simulator requires a very powerful device, which makes the task not accessible to everyone.

Therefore, after attempting to install all the required programs and facing with many errors and extremely poor performance we decided to use a service called the ROS development studio [10]. This is a free online platform that hosts an Ubuntu server with preinstalled ROS software. You can create your account and test your projects with a powerful CPU. It is a very user-friendly platform that incorporates tools such as Gazebo for simulation visualization, IDE, shell… among others.

The ROS version chosen for this project is the Kinetic, which runs with an Ubuntu 16.04. This is because for the last ROS version (Melodic), the LSD-SLAM packages are not updated. For previous versions of ROS the packages are outdated.

This environment has been developed by the company The Construct Sim which is formed by a group on engineers located in Barcelona.

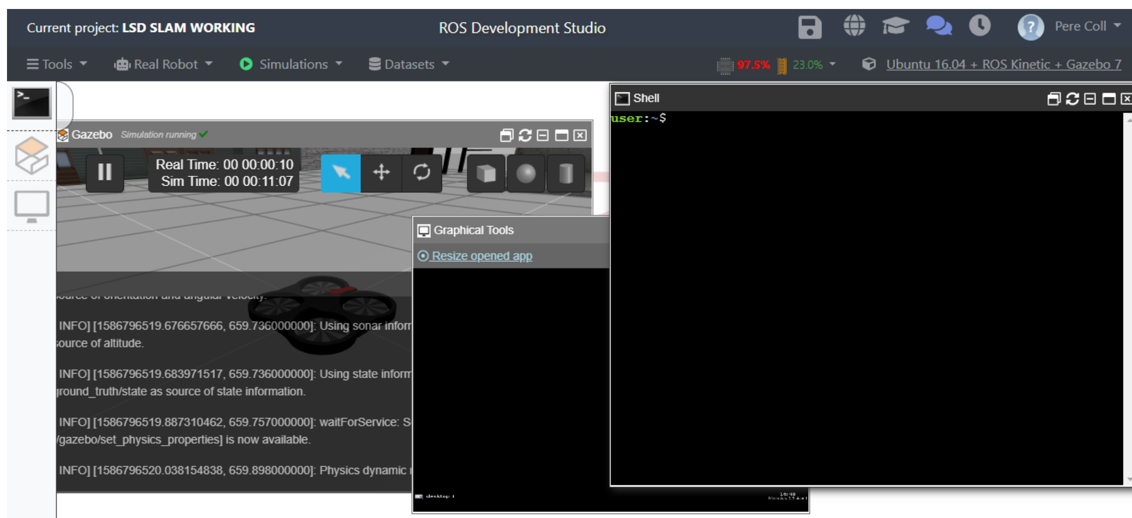

Illustration 15 - ROS development studio interface

CHAPTER 4. The LSD-SLAM technology

4.1. LSD-SLAM: Large Direct Monocular SLAM

4.1.1. Description

This technology was developed by Jakob Engel, Thomas Schöps and Daniel Cremers from the Technical University of Munich and presented in the 13th European Conference on Computer Vision, ECCV 2014, held in Zurich, Switzerland [11].

This project aims to build a visual SLAM method for monocular cameras, with the capabilities of building *large-scale and consistent maps of the environment along with the highly accurate pose estimation [14].*

This method uses the contrast of the images for its functioning. It applies filters to the obtained image in greyscale. Also, it takes objects geometry information that can be very useful for augmented reality.

This method represents the world by a number of key frames connected by position restrictions that can be optimized using an optimization graph.

4.1.2. The LSD-SLAM Algorithm

The LSD-SLAM algorith is divided in three main parts: the tracking, the depth map estimation and the map optimization. In this section we are going to have a look at which tasks are performed in each of them.

To initialize the algorithm, the camera needs to have a translational movement in the first seconds in order to lock a certain depth configuration. In order for the algorith to obtain a correct depth configuration, it will need to receive more than one keyframes.

In the image below, we can see the three algorithm modules.

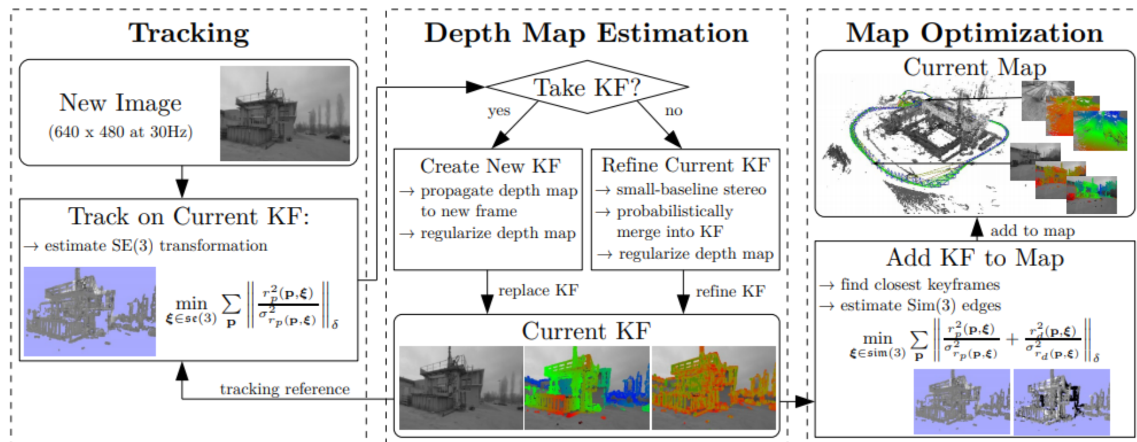

Illustration 16 - Block diagram showing the LSD-SLAM algorithm

Tracking Module

The tracking module purpose is to try estimate the camera pose with respect to the current keyframe, using the previous frame for the initialization. Therefore, this part is constantly tracking the new images obtained by the camera.

Depth Map Estimation

In this part is where a decision between if an image should be taken as a new keyframe or should be used in order to refine the current keyframe.

This decision is made taking into account two variables: the relative distance from the current frame and the relative angle to the current frame. If the weighted sum of both parameters is larger than a certain threshold, then the image will be taken as a new keyframe.

In the case that the image is taken as a new keyframe it will be needed to propagate the depth map to the new frame. This is done by projecting the projecting the points from the previous frame to the new one. After that, some spatial regularization is done.

In the other hand, the image is used to update the current frame. Therefore, multiple baseline stereo comparisons will be done where the accuracy is large enough in order to include then in the previous keyframe.

Map optimization

As each keyframe is scaled in the previous modules, all the keyframes may have different scale. So, when they are aligned it's important to take this different scale into account. This will be accomplished minimizing an error function.

When the keyframe is added into the 3D map, the closest keyframes are used to detect a *loop-close**. The map will be optimized using pose graph optimization.

* A loop-close primary purpose is to correct the drift that has been accumulated in the pose calculations along the time, as the visual SLAM or Odometry are surely accumulating errors overtime. Therefore, using visual recognition, some places are detected in order to close the loop and match the position with previous images.

4.1.3. LSD-SLAM for ROS Kinetic

The last package that we need in order to perform the visual Odometry will be the LSD-SLAM package [12]. It has also been installed via the catkin_make command. It is important to notice that the repository can be found in the reference number [12] has been adapted for kinetic version and Ubuntu 16.04, otherwise there may be several compatibility errors.

With these three packages installed we should be able to run an AR drone simulation and perform LSD-SLAM in order to generate a 3D map for the environment and guide the quadcopter.

This package is divided into two sub packages, the lsd_slam_core and the lsd_slam_viewer. The lsd_slam_core is the one that will execute all the SLAM calculus while the viewer will display the optional 3D mapping. This process can be performed in real time or using your own dataset. In our case we will be interested in the first scenario.

It is important to notice that the LSD-SLAM package needs QT version 4 instead of the default installed version 5.

4.1. LSD-SLAM running and testing

4.1.1. Simulation Steps

In this section we will see how to run the LSD-SLAM while the live simulation is running. First of all, we will need to open the simulation as described in the previous section.

After that we will open the LSD-SLAM viewer with the rosrun command:

```
rosrun lsd_slam_viewer viewer
```

This will open a grey window where the 3D point cloud representation will be displayed.

Next will be to start the lsd_slam_core and feed it with the camera image and information.

```
rosrun lsd_slam_core live_slam /image:=<camera_image_topic>
/camera_info:=<camera_info_topic>
```

In order to see the topics available, we can run the following command while the simulation is running:

```
rostopic list
```

So the command should look like this.

```
rosrun lsd_slam_core live_slam /image:=/AR drone/front/image_raw
/camera_info:=/AR drone/front/camera_info
```

The first time you execute this previous command will lead to errors because the default camera calibration file image dimensions do not fulfill the required. The dimensions should both be a multiple of 16 in order for the software to scale it down. Moreover, the camera calibration parameters are not the ones that match the AR drone 2.0 camera. Therefore, we will need to create a new calibration file and specify it in the “rosrun” command.

Camera configuration file

This file must contain the following parameters:

```
#focal lengths (fx, fy), camera optical centers (cx, cy) and lens
distorsion matrix (d, composed by k1 k2 p1 p2)

fx/width fy/height cx/width cy/height d

#camera size in pixels
```

```
input_width input_height
#camera image settings
"crop" / "full" / "none" / "e1 e2 e3 e4 0"
#output image size in pixels
output_width output_height
```

In order to obtain those parameters, we need to use a camera calibration package [13], and calibrate the AR drone monocular camera using a chess-type board.

As the simulator AR drone 2.0 has known parameters we will be using them for our simulations. The parameters are the following:

```
0.50355332 0.894045767 0.494061013 0.509863 0
640 360
crop
400 288
```

Where there is no lens distortion matrix and image is cropped to two multiple numbers of 16.

To run this configuration with this specific calibration file we will run:

```
rosrun lsd_slam_core live_slam /image:=/AR drone/front/image_raw
/camera_info:=/AR drone/front/camera_info
_calib:=<calibration_file_path.cfg>
```

We will also run the viewer in order to see the live point cloud as we did before and we will be using the keyboard to move the quadcopter around:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

After some flying you can obtain results as the following:

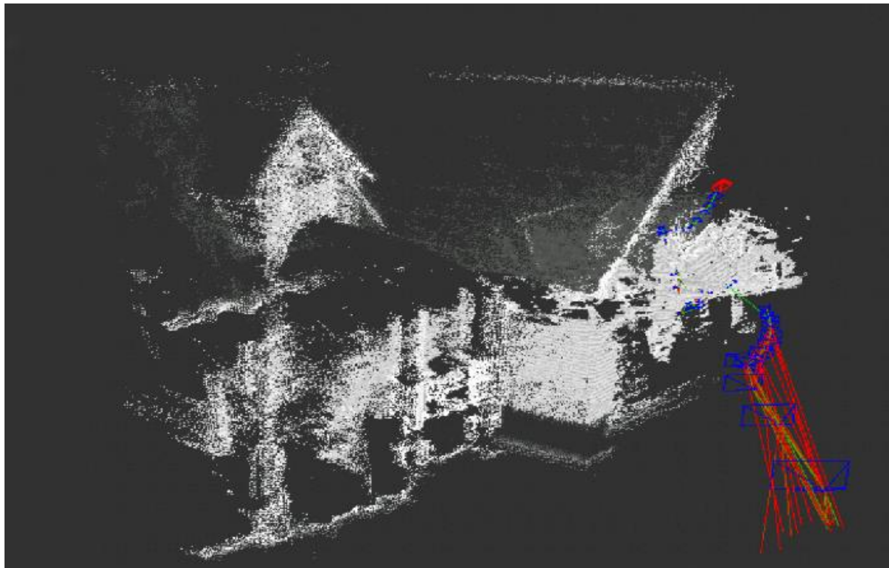

Illustration 17 - LSD-SLAM obtained point cloud

The point cloud above contains a set of points in the space. It is the mapping result of the contrast points followed by the SLAM system. Each point represents a solid portion of an element, and a set of points could be transformed into a solid object. The results of the mapping a relatively accurate and the solids can be detected easily.

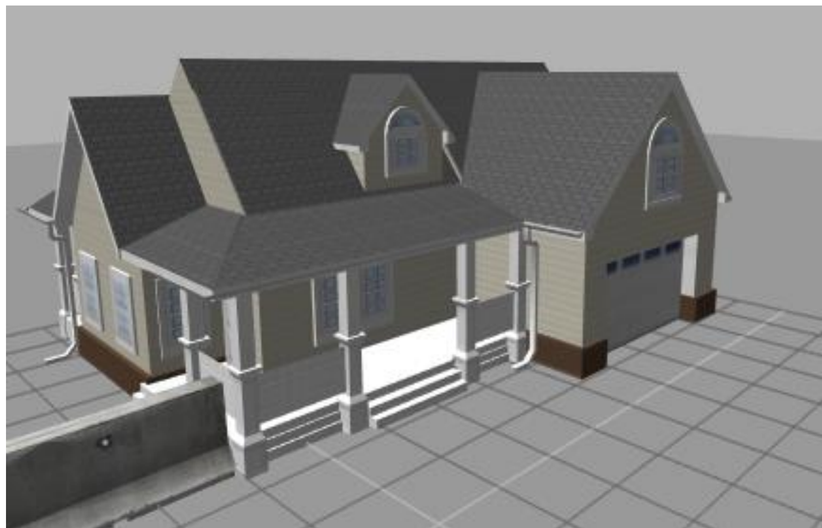

Illustration 18 - Simulated environment

It is important to say that the tracking system is not very good as for turning movements the camera gets lost very quickly. You need to combine translation with slow rotation in order for the tracking module not to get lost. Also, as there is no background in the simulation it's harder for the LSD-SLAM system to follow the tracking points.

4.2. *Testing in a Simulated Environment*

4.2.1. The Word Scale Ambiguity Problem

One of the major problems when using monocular cameras for gathering the environment information is the scale factor problem. The 2D cameras are not able to obtain the absolute distance of its displacement by just observation during the translational movement. This has a very clear explanation, as if you look to a chair without any depth information you will be unable to tell the size of that chair. Therefore, when calculating your own movement, the scale will be incorrect.

In order to compute the trajectory these types of systems need to compute the key points from the images by comparing the features obtained in two consecutive video frames. As the distance from the key points to the camera is unknown, the system won't be able to scale its movement properly.

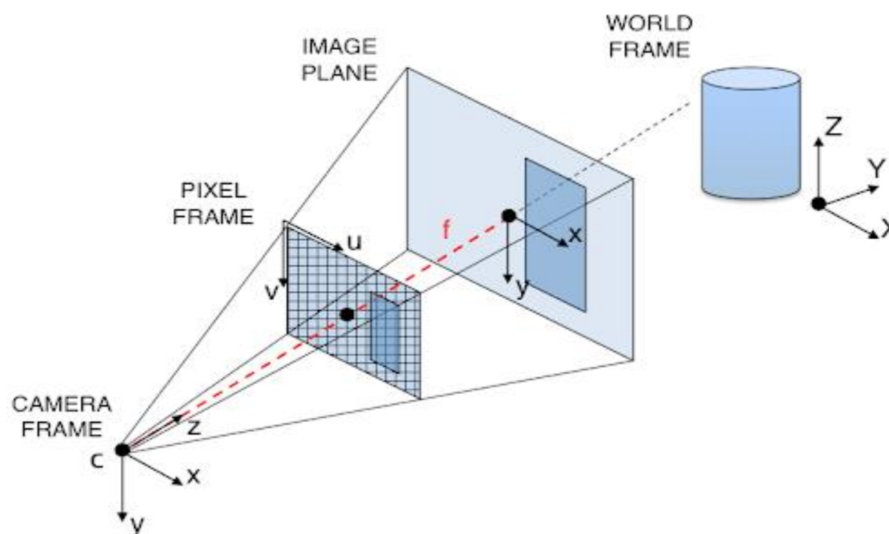

Illustration 19 - Scaling factor problem visualization

However, there is one possible solution for this problem as we have the IMU data from the ultrasonic sensor. This sensor is pointing downwards and will be measuring the real distance in de Z axis. Thanks to that, we can perform a vertical movement and compare de distance measured by the sensor and by the LSD-SLAM system.

In order to obtain the scaling factor, we will need to obtain the quotient between the real height measurement from the ultrasound sensor and the SLAM system as follows:

$$\text{Scale factor} = \frac{\text{Sensor height}}{\text{SLAM height}}$$

Find below a plot that shows the real height and the LSD-SLAM predicted height:

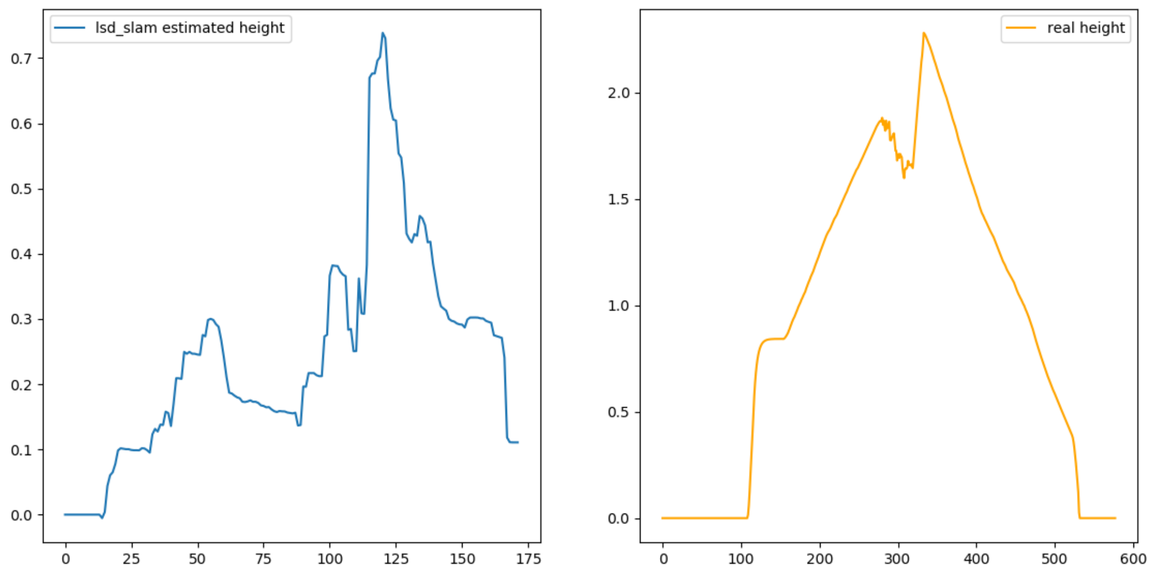

Illustration 20 - Height data comparison vs IMU data and vSLAM system 1.

In this particular example the real height (orange color) ranges between 0 and 2.28m, while the LSD-SLAM data only arrives at 0.74m. This would mean a scaling factor of approximately 3.1. This means that the results obtained by the visual SLAM system need to be multiplied by a factor of 3.1.

Nevertheless, if this experiment is performed in different simulations the scaling factor differs significantly from the other simulations. Even with the exact same simulation is carried out, the scaling factor may be different.

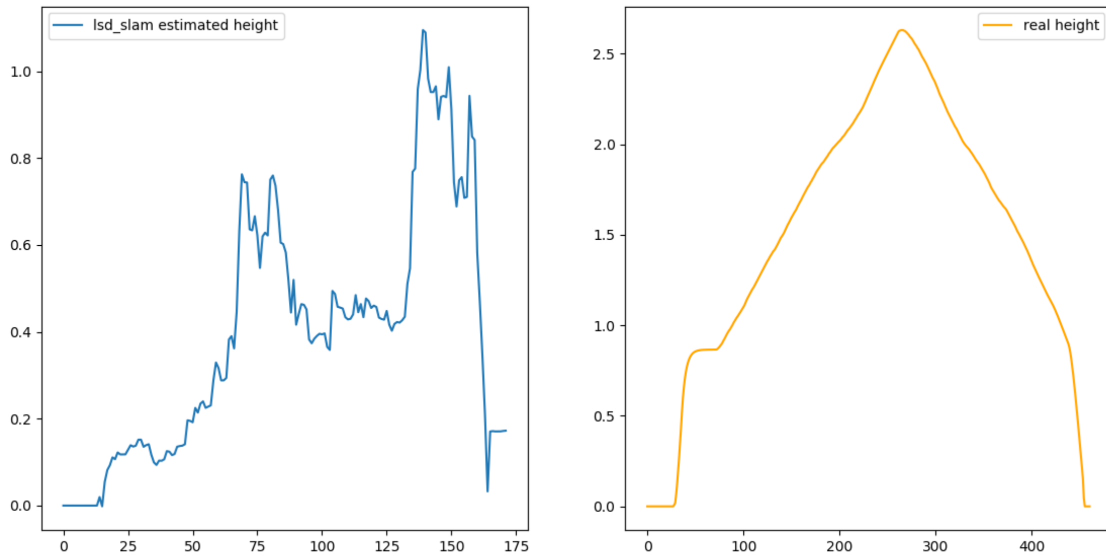

Illustration 21 - Height data comparison vs IMU data and vSLAM system 2.

If we have a look to another environment, in wider area where the objects are further from the camera we can clearly see that the observed scale is even smaller. In the example below we can see a scaling factor about 20.

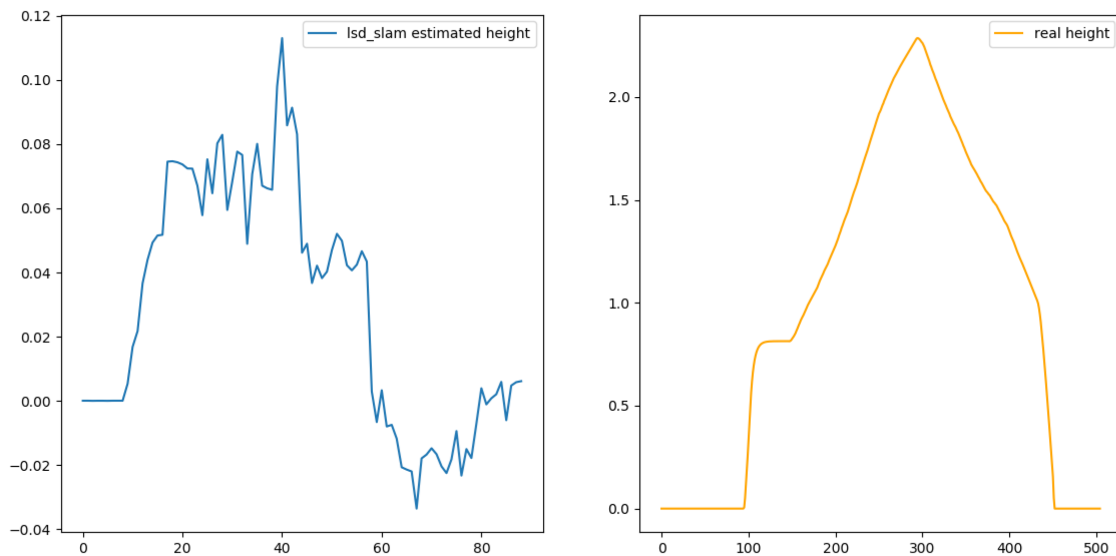

Illustration 22 - Height data comparison vs IMU data and vSLAM system in different environment

Therefore, if we want to get accurate data from the LSD-SLAM system we will need to perform the Z translational movement in order to compute the proper scale.

4.2.2. Path and environment setup

4.3.2.1 The Paths

In order to test the performance of the LSD-SLAM algorithm we will be testing different paths in an in an outdoors and indoors scenarios.

The paths decided for this experiment will be the following:

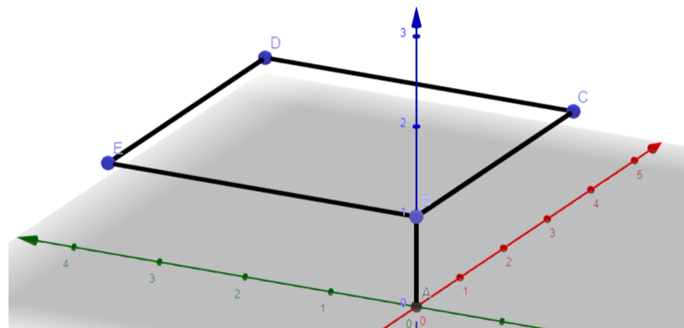

Illustration 23 - Experimental path 1

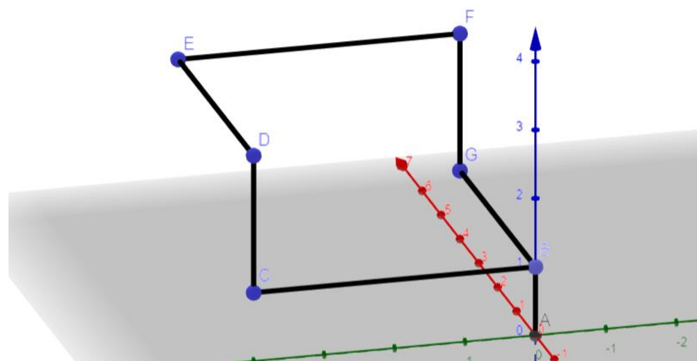

Illustration 24 - Experimental path 2

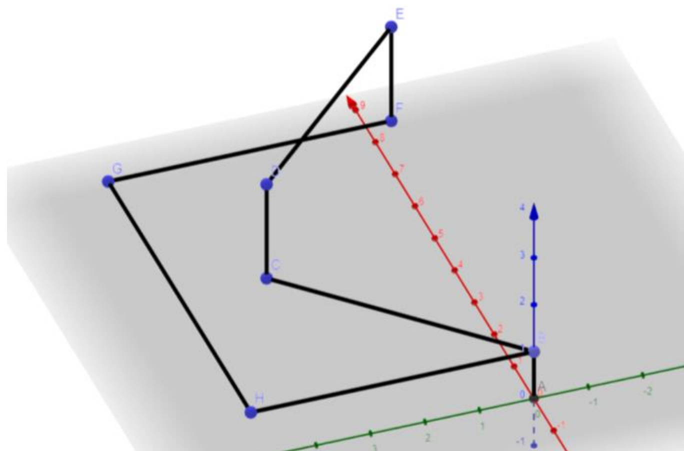

Illustration 25 - Experimental path 3

The chosen paths are all formed by pure translational movement, meaning that the drone will always be pointing to the same direction without turning in the corners. This means, that the robot will be moving forward, to the left and right and backwards.

This way we will avoid losing the track, as we have said before that rotating the drone within the same position may cause it.

The instructions sent to the drone to reproduce those paths will be simple directive instructions to move in the desired direction. This will be done programmatically using an executable batch file.

The maximum speed of the drone will be 0.2 m/s.

See below an example of the intructions file for the second track.

```
#!/bin/bash

rostopic pub -1 /ardrone/takeoff std_msgs/Empty;

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.0, y: 0.2, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}' & sleep 20 ; kill $!

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.0, y: 0.0, z: 0.2}, angular: {x: 0.0,y: 0.0,z: 0.0}}' & sleep 10 ; kill $!

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.2, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}' & sleep 20 ; kill $!

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.0, y: -0.2, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}' & sleep 20 ; kill $!

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.0, y: 0.0, z: -0.2}, angular: {x: 0.0,y: 0.0,z: 0.0}}' & sleep 10 ; kill $!

rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{linear:  {x: -0.2, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}' & sleep 20 ; kill $!

rostopic pub -1 /ardrone/land std_msgs/Empty;
```

4.3.2.2 The Environments

The scenarios chosen for the tests will be the following. Both scenarios were prebuilt in the ROS packages and it's implementation is very simple. The first environment will be the outdoors, with the ardrone facing to a direction where there are not many objects and data points to track. It's important to notice that the objects are far in the distance.

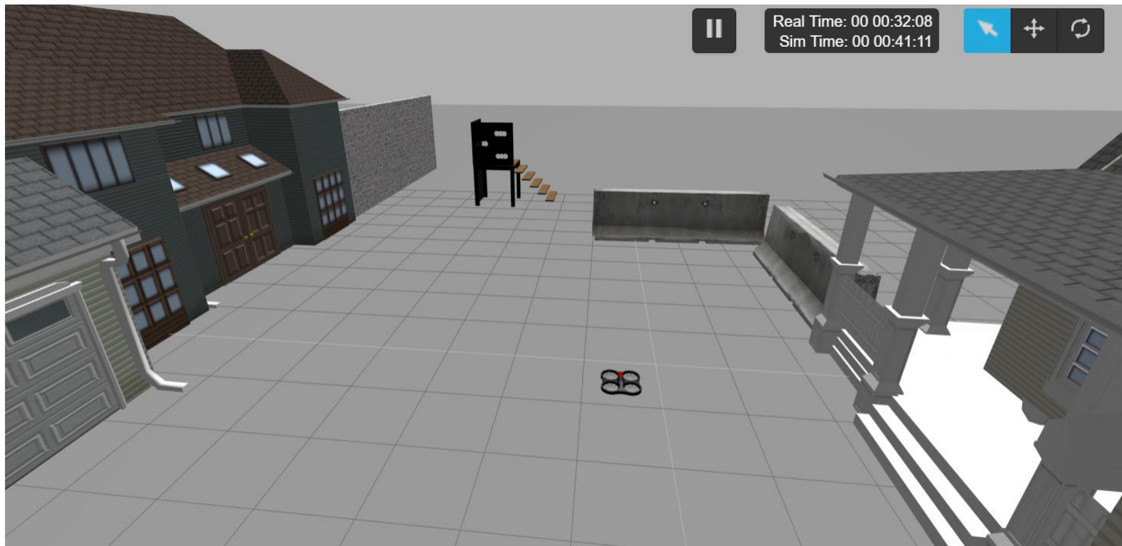

Illustration 26 - Test Environment 1: Outdoors Scenario

The second scenario will be the indoors, where a coffee place has been chosen. In this environment there are many points to track and the objects are closer than in the first case.

Illustration 27 - Test Environment 2: Indoors Scenario

4.2.3. Simulation Results

In order to analyse the accuracy of the LSD-SLAM method we will need to track the real position of the drone along with the estimated position obtained by the VO System.

To obtain those parameters we will need to save the data form the nodes corresponding to the variables needed. In order to obtain the nodes, we need to run the simulation with the SLAM System active. Then we will run the following command to get the list of nodes:

```
rostopic list
```

After doing that, we will realise that the nodes we are looking for are the following:

```
/ground_truth/state    #Real position of the done
/lsd_slam/pose         #Estimated position of the LSD-SLAM system
```

Therefore, we can create two batch executable files that echo the data from the nodes and save it to a file.

```
bash -c 'rostopic echo /lsd_slam/pose > test.txt'
bash -c 'rostopic echo /ground_truth/state > test2.txt'
```

With all this set, we can proceed to perform the tests.

In order to visualise the results, we will plot a 3d plot of both paths and a 2d plot for the x-y axis field. To do that, a simple python code was made that took into account the scaling factor and arranged the coordinates at each timestamp.

The orange line will always represent the real path followed by the quadcopter while the blue line will be the estimated trajectory computed by the LSD-SLAM algorithm.

Result Path 1:

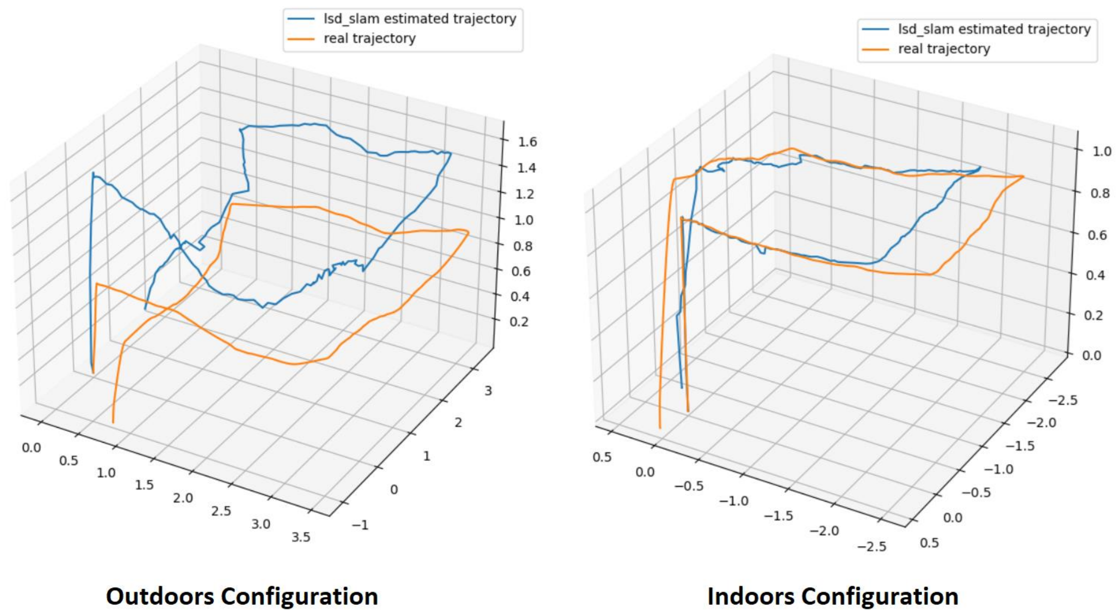

Illustration 28 - Path 1 results (3d view)

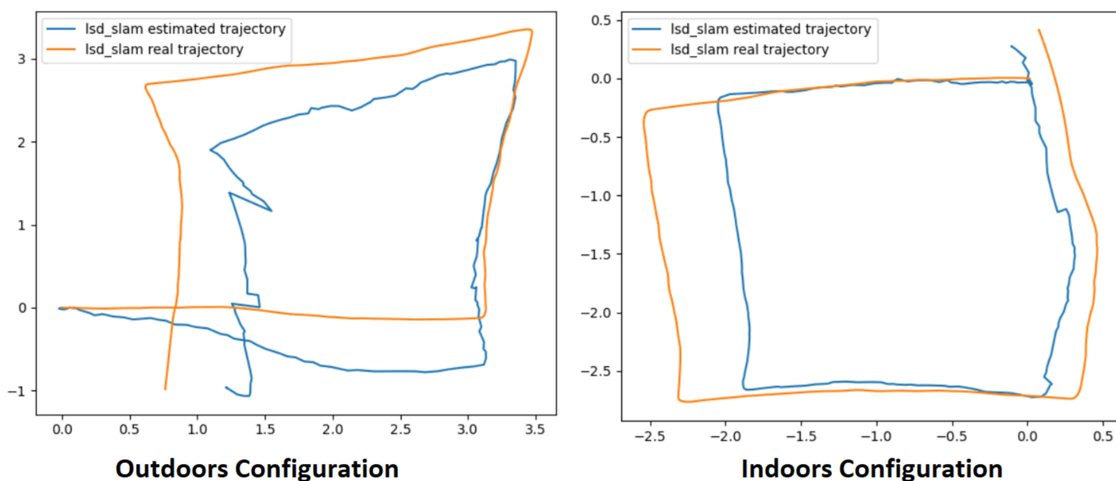

Illustration 29 - Path 1 results (x-y view)

Result Path 2:

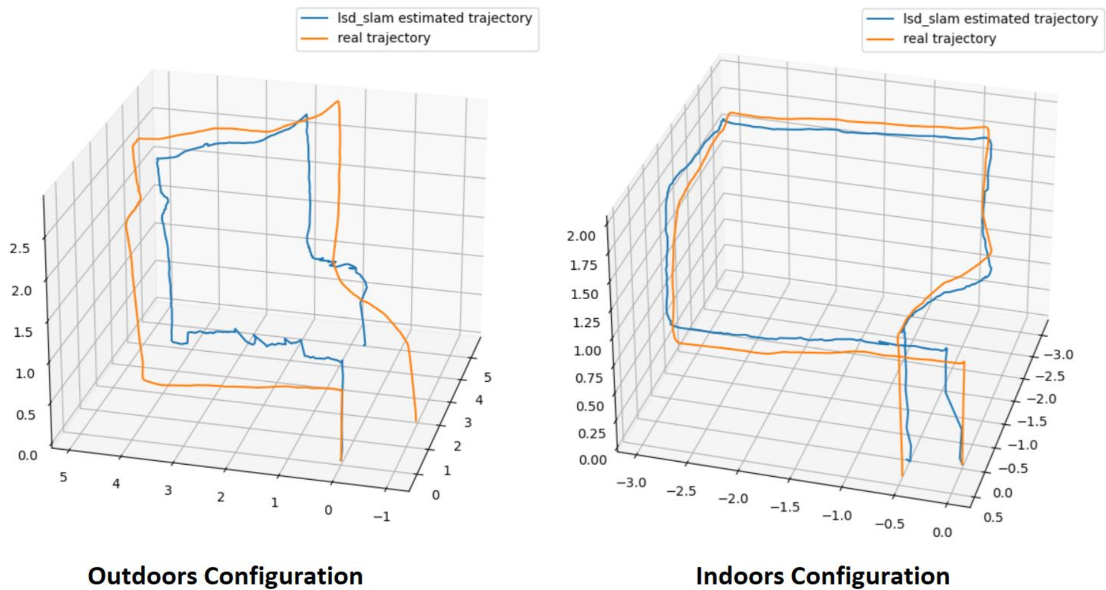

Illustration 30 - Path 2 results (3d view)

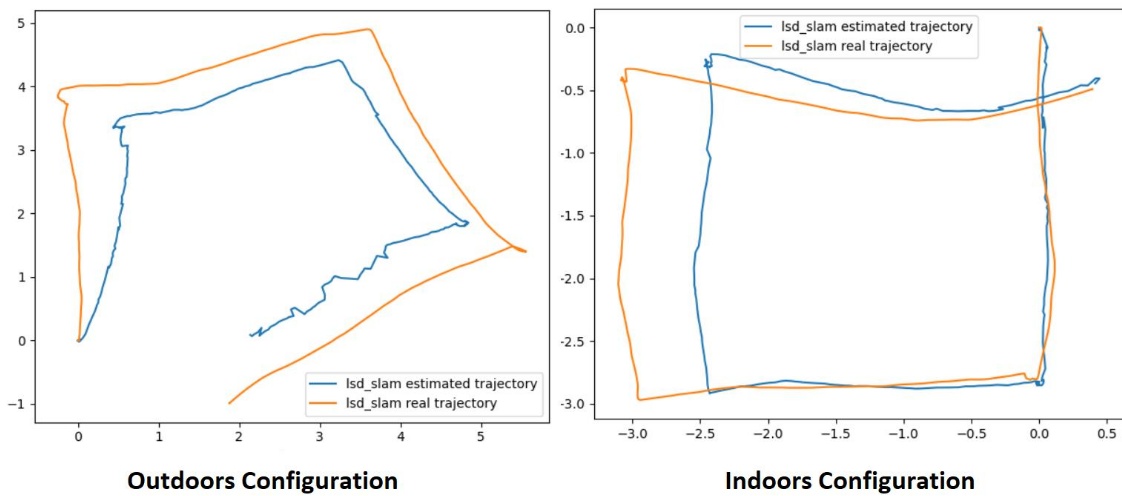

Illustration 31 - Path 2 results (x-y view)

It's clear to see that the results obtained in the indoors scenario are more accurate than the performance in the outdoors case. The precision of the estimated path is way higher in a closed scenario, where there are more objects and features to track in a closer distance from the drone.

We can also see that the estimated trajectory is smoother in the indoors case.

Result Path 3:

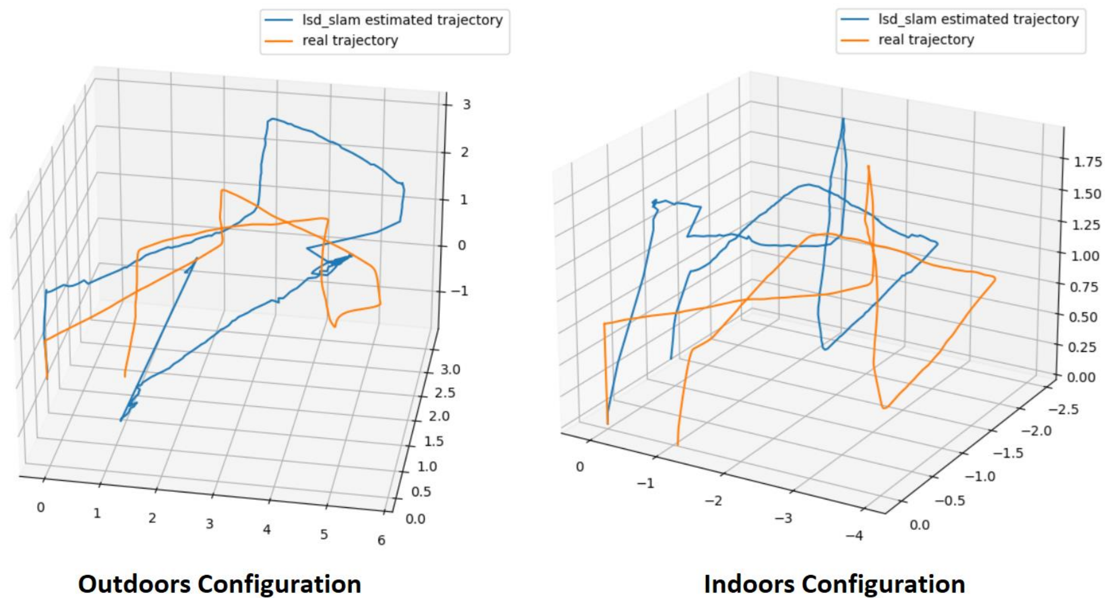

Illustration 32 - Path 3 results (3d view)

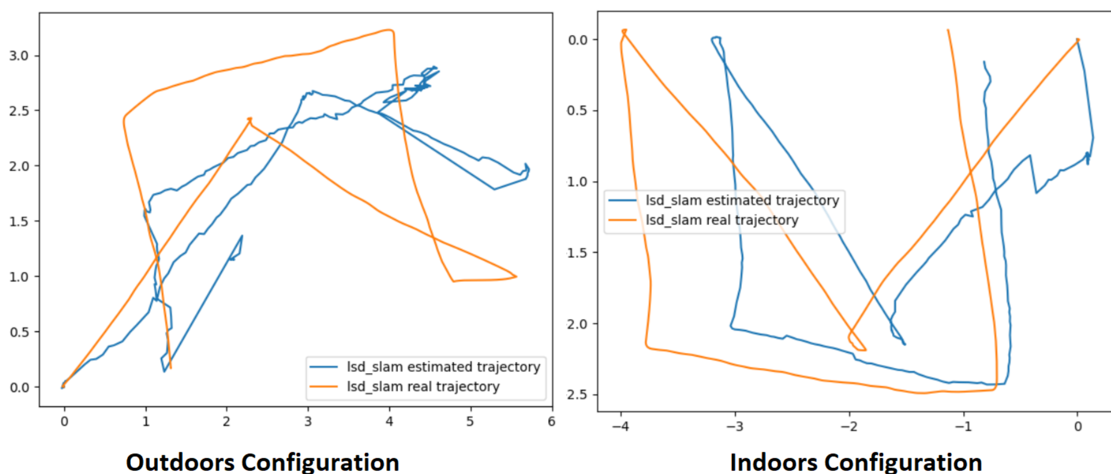

Illustration 33 - Path 3 results (x-y view)

In this specific case we can see a very important hazard of the SLAM Systems which is the cumulative error caused by an uncertainty in the beginning of the tracking process. This problem can be solved by adding known areas in the environment, where the SLAM System would revaluate the drone position and mitigate the cumulative errors.

4.3. Results

4.3.1. Performance Analysis

In this section we will discuss the results obtained in the different tests performed in the previous section. In order to have a numerical value for the precision in each environment we will compute the Mean Absolute Deviation (MAD) and the maximum deviation value for each case.

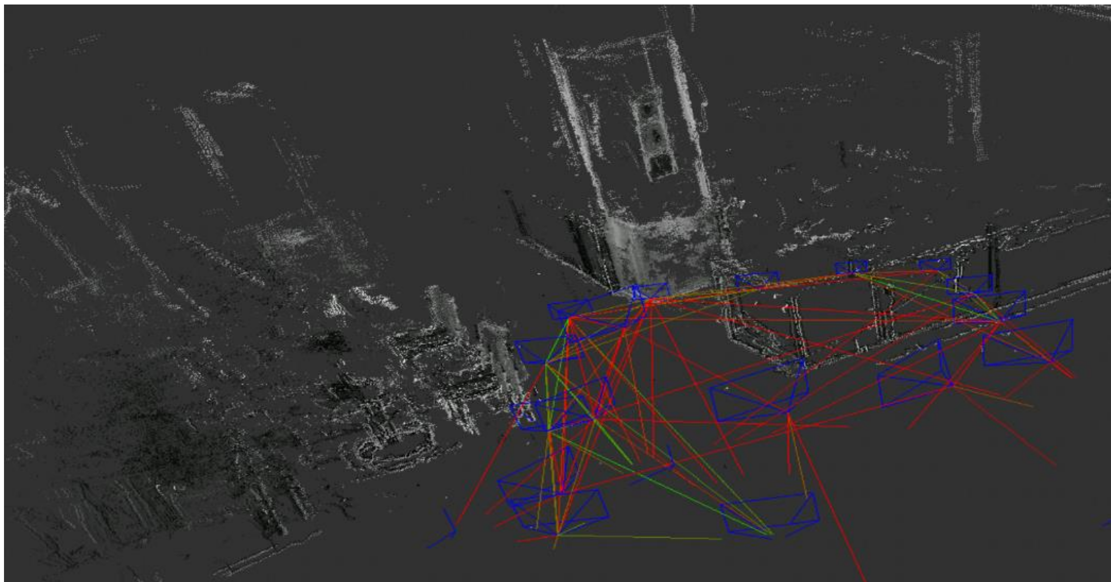

Illustration 34 - LSD-SLAM system mapping and tracking visualization

To start, we will have a look at the definition for the MAD to briefly explain how is calculated.

- The Mean Absolute Deviation: it is a very good way to express the variation within a dataset. Its value corresponds to the averaged value between all the absolute errors within a dataset. In our case, the averaged number will be the real position of the drone given by its navigation data. See the formula below:

$$M.A.D. = \frac{\sum|x - A|}{n}$$

Illustration 35 - Mean Absolute Deviation Formula

In the formula above, the x represents the drone pose value (x, y or z) that has been given by the LSD-SLAM system for each time step. The A represents the real position for the quadcopter, given by the simulation navigation data. The n states for the number of data points studied in each scenario.

Now we will see the given error for each track and scenario, so we will be able to evaluate each case:

	Outdoors Environment	***Indoors Environment***
Track 1 (x value)	0.240	0.271
Track 1 (y value)	0.445	0.135
Track 1 (z value)	0.580	0.134
Track 2 (x value)	0.441	0.305
Track 2 (y value)	0.390	0.070
Track 2 (z value)	0.262	0.068
Track 3 (x value)	0.394	0.424
Track 3 (y value)	0.511	0.173
Track 3 (z value)	1.009	0.292

Illustration 36 - MAD error comparison (m)

As expected, the performance of the SLAM algorithm is more precise in the indoors environment than in the open world environment. This is due to the number of objects to track: the higher the features to track, the smaller the error we will obtain.

In the open world scenario there is always the distance problem, where we can have the blue sky without any possible tracking points. In the case of this simulation there nothing else behind the near objects, showing an infinite white wall. Therefore, the algorithm could not track many points what is clearly reflected in the MAD results.

This problem gets even larger when we talk about cumulative errors, as the beginning of the tracking is crucial in order not to have them. The fact of having more objects and in different distances makes it easier to start a more accurate pose estimation.

It can also be interesting to see the maximum deviation for each scenario. See the values below:

	Outdoors Environment	***Indoors Environment***
Track 1 (x value)	0.696	0.610
Track 1 (y value)	0.844	0.422
Track 1 (z value)	0.96	0.478
Track 2 (x value)	0.775	0.633
Track 2 (y value)	1.127	0.193
Track 2 (z value)	1.029	0.493
Track 3 (x value)	0.995	0.887
Track 3 (y value)	1.718	0.834
Track 3 (z value)	2.202	0.782

Illustration 376 – Max error distance comparison (m)

Again, the values obtained are better in the indoors scenario. However, for autonomous navigation between small spaces such as going across a door or navigating along a small corridor, could result in a drone crash.

In order to get smaller navigation values we would need to add waypoints along the track, where the system could recalibrate its position and mitigate the errors.

So, as overall results, the evaluation of the algorithm in indoors and outdoors spaces has concluded as it was expected. The algorithm works better in an indoors scenario. Also, we have learned how to obtain the data from the ROS nodes in order to represent the flight plan of the drone.

CHAPTER 5. Conclusions and Future Work

5.1. Conclusions

Along the project it has been used a Parrot AR drone quadcopter with ROS integration and Gazebo Simulator. All the tests and results have been obtained by using the simulated environment. The algorithm used for the position estimation has been the LSD-SLAM.

The first thing to notice when working with the Robot Operating System is the complicity on its installation. It's lack of integration with another operating system rather than Linux makes it harder to implement. Also during the setup it's very common to hit errors, due to different versions on the Ubuntu preinstalled packages, or outdated ROS packages. The task of setting up the ROS, the gazebo simulator and the LSD-SLAM system has been one of the toughest parts of the project.

The use of RDS Studio has been a great tool for the research, avoiding the poor performance of my laptop. The 8 free hours a day are more than enough to test and implement your simulations. The LSD-SLAM package is extremely out-to-date. It uses old package versions which will make you face many errors.

Leaving to a side the difficulties encountered in the installation process we can conclude that the use of simulated environment for testing is a great tool, especially if you cannot access a drone for any reason.

The data obtained from the indoors and outdoors tests show the expected results: the algorithm is more accurate in closed spaces where there are closer objects. Although being more accurate, the difference between the real and the predicted trajectory is too large for the robot to have accurate navigation through doors, for example.

An important problem of the LSD-SLAM is the loss of tracking when turning or heavily shaking. The algorithm gets completely lost when that happens and the re-tracking feature struggles to get the tracking back. This gets worst in simulations, as the control of the drone is usually even trickier.

All the steps followed during the project have been described in order to make it easy to reproduce in the future. All the packages with the proper version are linked in the references so there are no errors in future simulations.

5.2. Future Work

This project opens many positions for upcoming and future work as the field of the VO and the SLAM algorithm is very recent. Below I will propose some ideas for future projects:

- The LSD-SLAM tracking works with a certain precision when the drone is performing a translation movement but gets lost easily when turning. Therefore, it could be studied the angular ratio for rotations in order not to lose the track, along with trying to improve the algorithm in order to be capable of improving accuracy when turning. That would be very interesting as it is a big issue in the studied algorithm.

- The autonomous navigation for aerial robots is nowadays very unexplored, as robots are commonly piloted by an operator. Combining the LSD-SLAM system mapping with solid reconstruction the robot should be able to avoid collision with the walls and be able to travel from a point A to a point B on its own. Studying the possibilities of self-navigation using this SLAM system can also be a very interesting topic to do some research on.

- The LSD-SLAM system studied uses a simple 2D camera to map and track the drone position. It could also be a very attractive project doing a study about the differences in building and implementing a monocular SLAM system vs a stereo system. Doing some research on computational power used, size of the drone, cost of the equipment, accuracy of both systems…

- The SLAM systems can also be mainly used mapping purposes. There are applications such as game development or scene reconstruction where being able to create a 3D design based on reality can be very useful. Doing some research in which SLAM technique is better for this purpose could also be an interesting project.

Bibliography

[1] Parrot AR done 2.0 official site https://www.parrot.com/es/drones/parrot-AR drone-20-elite-edition

[2] Phantom 4 Pro official site https://www.dji.com/es/phantom-4-pro-v2?site=brandsite&from=landing_page

[3] Waymo or Google Self-driving car https://waymo.com/

[4] ROS Organization “ROS file system concepts”, http://wiki.ros.org/ .

[5] Gazebo Simulator Page http://gazebosim.org/

[6] Gazebo Integration with ROS https://github.com/ros-simulation/gazebo_ros_pkgs

[7] AR drone Autonomy Package https://github.com/AutonomyLab/AR drone_autonomy

[8] Tum AR drone Package https://github.com/iolyp/ardrone_simulator_gazebo7

[9] ROS help forum https://answers.ros.org/questions/

[10] ROS Development Studio, https://www.theconstructsim.com/rds-ros-development-studio/

[11] Engel, J., Sturm, J., Cremers, D. “Semi-dense visual Odometry for a monocular camera.”, *Intl. Conf. on Computer Vision (ICCV), (*2013).

[12] LSD-LSAM ROS Kinetic https://github.com/kevin-george/lsd_slam.git

[13] Camera Calibration for AR drone http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration

[14] LSD-SLAM: Large-Scale Direct Monocular SLAM (J. Engel, T. Schöps, D. Cremers), ECCV 2014. https://vision.in.tum.de/_media/spezial/bib/engel14eccv.pdf