University of Padua

Department of Information Engineering

Master Degree
in Control System Engineering

Visual inertial SLAM dense mapping for indoor autonomous navigation

Supervisor: Prof. Angelo Cenedese
Co-Supervisors: Prof. Julien Hendrickx
and Ing. François Wielant

Student:
Lorenzo Pasini
2012267

Academic Year: 2021/2022 13/10/2022

To my family.

Acknowledgements

My heartfelt thanks to my family and friends for being present during this journey, to professors Julien Hendrickx for its valuable advise, to François Wielant for its daily technical support and to professor Angelo Cenedese for his important suggestions.

Abstract

The field of Simultaneous Localization and Mapping (SLAM) algorithms is gaining a lot of importance nowadays due to his importance in mobile robotics application. In fact even if the problem has been explored since the 80's only in recent times the increasing hardware power at our disposal and the growth of interest in the field allow the creation of really effective algorithms to track the position of robots and map the environment at the same time. Simultaneous Localization and Mapping algorithms can be built on a wide range of sensors, or a combination of them. The most used are for example Lidar, radar, camera, accelerometer, depth cameras and many others. Cameras in particular are really interesting since they are ubiquitous and really cheap. The problem with camera based SLAM algorithm is that they usually produce a sparse map of the environment. A dense map is really better for navigation purposes so in this thesis the principal aim is to enhance a visual slam based algorithm to build a navigable dense map of the environment. In particular the algorithm that will be used and analyze is ORB-SLAM3 in the configuration in which the true scale of the map is recovered using the integration of accelerometer data. The dense map will be produced using a relative depth estimation neural network and the visual slam algorithm will be used to rescale the neural network estimation and for tracking purposes.

Introduzione

Motivation

Find motivations for a project like the one that will be described in this thesis is not so difficult. Robots are in fact ubiquitous nowadays and in particular mobile robots are more and more often present both in industrial and in home environment. Robots can be used to transport various type of material autonomously around a factory, but they can also clean the floor of our houses and cut the grass of our garden. To complete all these type of task more efficiently they need to answer two fundamental question

- *Where am I?*
- *What is around me?*

These question are tricky also for a human being. For us is natural to move inside any environment without thinking about all the process that our brain is completing to allow us to move around without any difficulties. Simultaneus Localization and Mapping Algorithm try to give the ability to answer these question to a robot, and in a really general way the answer is found as follow

- First of all the robot need to collect data from the environment at each time step. To complete this task a lot of different sensor can be used like Lidars, radars, cameras, depth sensors, accelerometers etc. It will be clear in all this thesis that choosing the right sensor configuration and setting up them in the right way will be fundamental to obtain good results.

- As a second step the algorithm try to find a change in the pose of the robot that justify the change in the observations in different moment in time. To do that we usually need to solve an optimization problem

- The third element is to build a coherent map of the environment based on all (or at least a great amount) of previous and current information

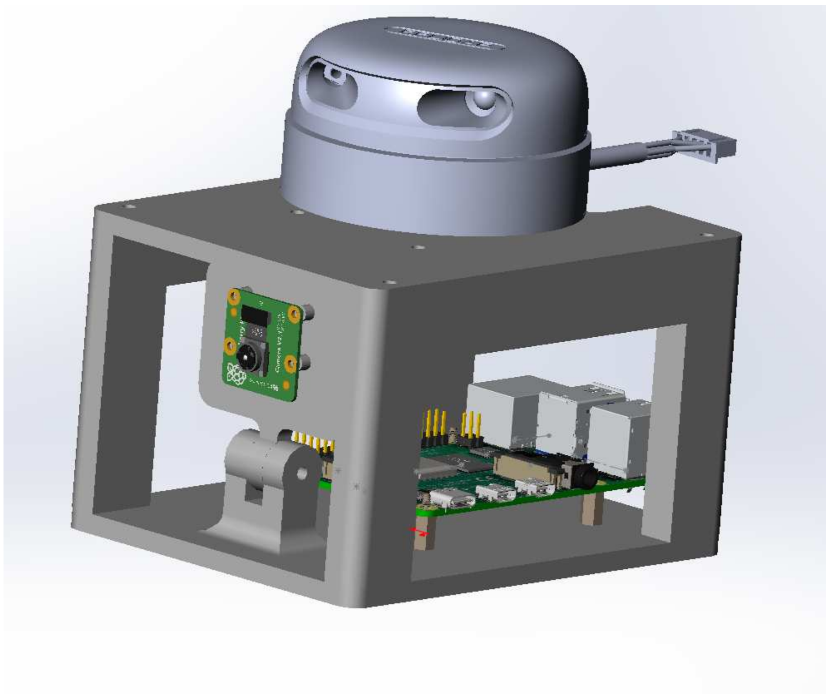


Figure 1: CAD image of the sensor setup built by previous year student. Image taken from their GitHub repository

Obviously complete SLAM algorithm are far more complex than that and they implement a lot of different routine to solve problem like loop closure, increase robustness or manage the built map in the most efficient way. All these things will be describe in greater details in the thesis but the above points can be considered as a really general description of the principal task that a SLAM algorithm need to accomplish.

Project goals

This project is part of a multi-years effort of the Universitè Chatolieque de Louvain which goal is to build a drone able to explore autonomously an indoor environment. It will be clear that the distinction between indoor and outdoor environment is still really important, in particular during the choice of the sensor setup. All the previous works [1] [2] made use of the A.R Drone 2.0 platform and in particular they focused on the visual information coming from the camera to build the slam algorithm. Last year a decision to change the approach was taken and a new platform (shown in figure [1]) was build. The sensor chosed were a 2D Lidar, an accelerometer, and a rgb camera. The entire setup will be described in more details later but the general idea of last year project was to use a neural network to estimate a depth map from the rgb image provided by the camera, and then try to correct this estimate using the data coming from the 2D lidar. Once they have produce a point cloud in this way they want to feed that point cloud to an already build RGBD SLAM algorithm to estimate both the position and the map structure. The possible difficulties that one can encounter using this approach are in particular two

- Neural Network that estimate absolute depth are not so precise for the moment. In particular the Neural Network used last year was AdaBins [3].
- The single line of points provided by the 2D Lidar do not add a lot of information that allow us to correct the depth map produced by the Neural Network.

However the principal aim of last year project was to build a dense navigable map for the robot. In fact the vast majority of SLAM algorithm that use cameras as the main sensor (which are called visual SLAM algorithm) produce a sparse [4] [5] or a semi-dense map [6] [7], which are not suitable for indoor navigation. Last year result were promising but the authors underline that their implementation was not ready for real time SLAM application in particular because of the precision of the neural network and the time needed to run it online.

Strategy choosed for this thesis work

To improve last year result and to be able to build a 3D navigable map we try to take a different approach. First of all we try to use a different Neural Network to achieve better precision in the depth estimation. Our choice was at the end MiDaS [8] [9], which is a Network that estimate the relative depth between point in the image and not the absolute depth. It is able to achieve quite good precision, but the depth estimate need to be rescaled to be able to obtain a metric depth map. To do that we decide to use information coming from an already existing visual-inertial SLAM algorithm, called ORB SLAM 3 which is able to estimate the metric depth of a few hundred points in the image. Once we have the depth of these few hundred points we can do a simple linear regression to rescale the Neural Network depth estimation and produce a point cloud. It is possible to do this procedure only at each keyframe to avoid the computational comlpexity of estimate the depth with the NN at each frame coming from the camera. Moreover the tracking thread of ORB SLAM 3 will give us also the global position of each

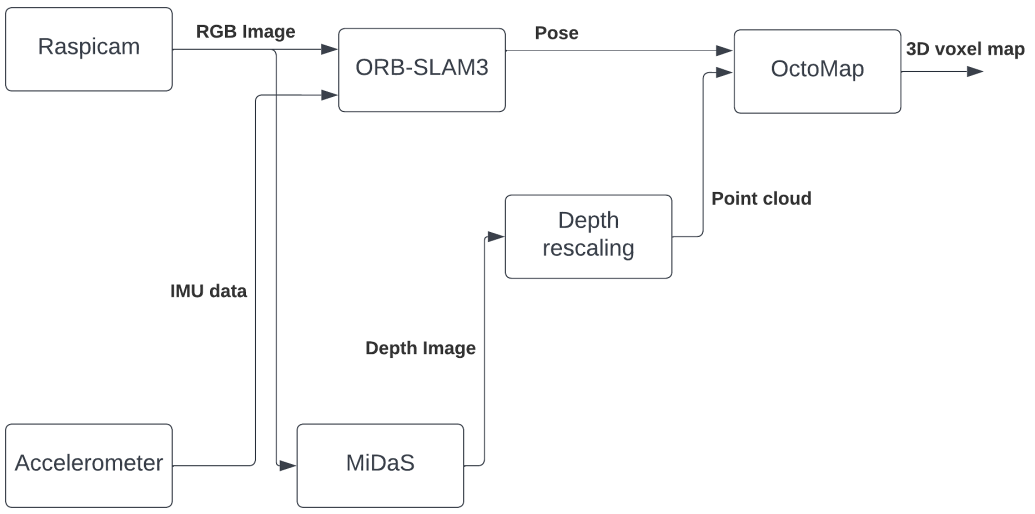


Figure 2: General high level graphical description of the choosen approach

keyframe. In this way we will be able to produce point clouds with their global position and give them to OctoMap, which is an algorithm able to combine them to produce a global voxel probabilistic map. This type of approach solve last year problems regarding the time needed by the neural network to produce a single estimate, and concerning the precision of the neural network. The principal difficulties that we have encounter implementing this type of approach are

- setting up ORB-SLAM3. Starting from the installation to all the practical measures that one need to take in consideration to make the algorithm work with a particular experimental set up

- reading and modifying the code of ORB-SLAM3 to be able to obtain the pose and the depth information avaiable in the ROS network

- syncronize the depth sparse data from the ORB-SLAM3 with the RGB image coming from the camera

once these problems are solved a 3D navigable voxel map of the environment can be produced. Obviously the main problem of this approach is the use of a neural network for the depth estimation since NN can be really powerful but sometimes really unpredictable. An concise graphical description of the method can be seen in figure 1.4.

Thesis structure

The Thesis will be structured in the following way

- *Chapter 1*: General description of SLAM algorithms, their history and description of the most important existing techniques based on the sensor setup that can be choosen. Moreover the mathematical tools needed to understand SLAM algorithm will be explained.
- *Chapter 2*: Introduction to visual SLAM algorithms, description of basic computer vision concepts, and of other important techniques used to solve the problem.
- *Chapter 3*: Deep description of ORB-SLAM3. Analisys of the modification introduced to the ROS Wrapper to extract the pose and depth information.
- *Chapter 4*: Description of the MiDaS NN and of the procedure used to rescale its depth estimate. Description of OctoMap and explanation of the result that we have been able to obtain.
- *Conclusion*: Strenght and witness of our approach will be described and the possible future developement will be explained. Moreover the lesson learned during the implementation of this project will be reported.

Contents

List of Figures

List of Tables

Chapter 1

Simultaneous Localization and Mappping

The aim of this first chapter of the thesis is to give a general overview of the concepts needed to understand Simultaneous Localization and Mapping algorithms and the sensors usually used to implement them. Moreover the history and main mathematical concepts needed to understand SLAM will be described. In particular the chapter will be divided in the following three section

- The first section will be dedicated to a general discussion of SLAM algorithms with the focus on the main piece of software needed by a modern and complete algorithm of this type. Some of the most famous algorithm based on the choice of the sensors will be explained.

- The second section will be dedicated to the description of the main mathematical tools needed to understand SLAM. In particular the most important mathematical instruments are linear algebra for pose representation and non linear optimization tools.

- The third and last part will be used to described the history of SLAM following a particularly useful historical division given by [10] and with the help of other two important surveys [11] [12].

The main objective of this chapter will be to give general concepts that apply to all SLAM algorithms regardless of the sensor chosen. The reader must taking into account that this thesis will focus on visual-inertial SLAM algorithms, so algorithms where the sensors used are a camera and an accelerometer.

1.1 SLAM algorithms structure

The first section of this chapter will be based on the subdivision given in the first part of [13]. This book can be considered as a really good introduction in particular to visual SLAM, starting from the mathematical concepts to the code implementation of the single modules needed by a SALM algorithm. A Simultaneous Localization and Mapping Algorithm is usually divided in the following modules

- data collecting and frontend odometry
- backend optimization
- loop closing
- mapping or reconstruction

each of these components will be describe in the following. Before starting with this description I want also to underline the multidisciplinarity of SLAM as a research study field. A lot of different topics must be understood to be able to understand SLAM. Some of the most important subjects that are involved in SLAM are

- computer vision: which is really fundamental in visual SLAM when you need for example to extract feature used as landmark by the SLAM algorithm
- optimization: important both for position tracking and for backend map and trajectory optimization
- electronic: sensor setup and communication but more in general all the hardware needed for achieving real time SLAM. Take in consideration that for implementing a SLAM system you will need to know how to synchronize and configure all the physical element of the system
- computer science: for the efficient code implementation of the algorithm it is really useful for example to correctly manage the memory of the system but also to build the program in such a way that it can run in a really fast way also in low power embedded systems

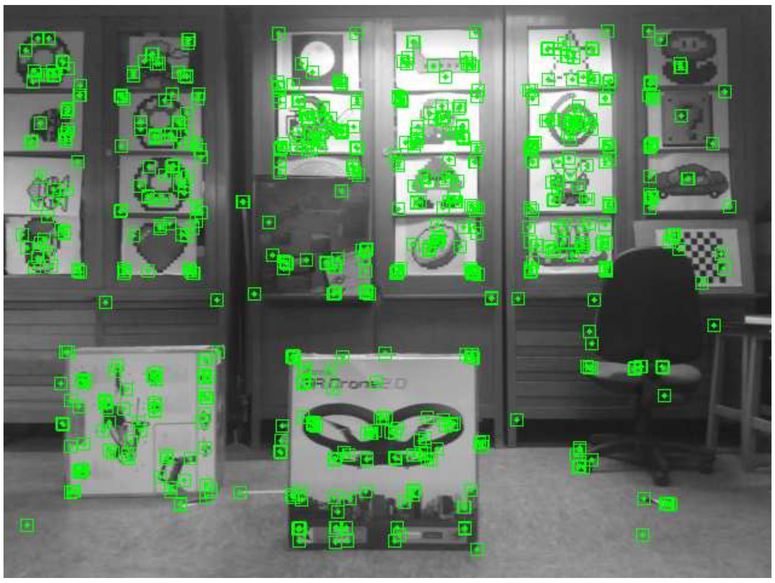


Figure 1.1: Example of ORB features extracted by ORB-SLAM3 from a black and white image taken in the lab

1.1.1 Data collecting and frontend odometry

To locate itself and map the environment a robot need to collect data or observation of the landscape. To do that we need sensors and the choice of the sensors is really fundamental to understand which SLAM algorithm we will use. Collecting data in real time from different sensors and synchronize them could seem a simple task but it is really fundamental to extract good performance from the SALM algorithm. Once the data are collected (data can be distances given by a 2D lidar, images coming from a camera, acceleration coming from an mpu) the frontend part of the algorithm will do two principal things.

Feature extraction. Once we have the raw data usually we can extract only the ones that are particularly interesting to track the position of the robot. Take as an example data coming from a 2D Lidar. In this case we have the distance of points all around the robot at a certain height. In this case we can be interested only in simple reobservable points in the environment such as the corners of the room. The same things can be said for images, in fact really often we are interested only in relevant points in the image and to find them we will need a feature extractor.

Data association With the given relevant points this part of the algorithm will try to estimate the pose of the robot given two subsequent observations of the environment. The concept is quite simple: taking again the example of the 2D Lidar and suppose that we are in a perfectly squared room. Given that the four extracted corner of the room were at particular distance and that now these corner are detected in a different position, the question is: which is the most probable position of the robot now? Answering this type of question often means solving optimization problems.

It is important to notice that extracting feature from raw data is not mandatory and in fact there exist algorithms that solve the optimization problem on all the 2D Lidar readings (this is called scan matching), and direct visual SLAM algorithms which solve an optimization problem on all the pixel intensity of two subsequent images. There are pro and cons with both the approaches in fact even if extracting features from the raw data could be really time consuming, the following optimization problem that we need to solve to estimate the robot position will be lighter if we need to solve it considering only the extracted features. Moreover choosing to solve the optimization problem only with feature points rather than with all the data coming from the sensors could be less reliable or more efficient depending on how good the data filtering algorithm is.

1.1.2 Backend optimization

Once we have the estimated pose and the observations collected from the frontend part of the algorithm the backend part comes into play to build and optimize trajectory and map. This part of the algorithm is really fundamental because it take in consideration that measurements from any sensor are noisy. For example Lidar measurement can be affected by usual electronic noise or background light. With cameras the problem is even more relevant if we take in consideration that cameras are continuously adjusting their white balance for example. This could seem a really simple problem to solve, because for human, even if two images are the same except for the white balance with which they are acquired, it is really simple to recognize them as the same image. A microprocessor see these images only as matrices of numbers so these two images will be for him totally different images. The backend part of the SLAM algorithm is usually considered as the one which is really more interesting for pure SLAM researchers. This part is usually seen as a state optimization problem and the base formulation of the problem will be a probabilistic one. This probabilistic approach will allow us to filter the previous mentioned noises. Since this part of the algorithm is solving in practice a state estimation problem there are multiple way to implement it. We can use for example Extended Kalman Filters, Particle Filters which are called filtering approach and are considered the most classic ones, or we can use maximum a

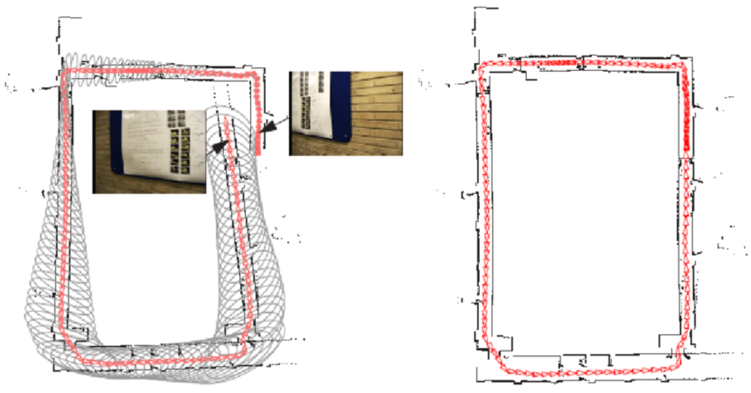


Figure 1.2: Example of a map before the loop closing algorithm correct the drift caused by normal imperfection in the SLAM system. Image from [14]

posteriori estimation. See section II of [10] for a wider explanation.

1.1.3 Loop Closing

The loop closing problem is one of the most famous problem discussed in the SLAM community. It is always caused by the fact that sensors and algorithms has not infinite precision. The problem is that the algorithm will always be affected by some error which accumulated during time. This error will cause a wrong estimation of the position and when the robot come back to an already visited position it will be probably be affected by a drift which can be small or big depending on how good the SLAM algorithm is. We can take an extreme example in which we use only data coming from an accelerometer to estimate the position of the robot. Suppose the accelerometer gives us 100 samples per second and that we have a mean error of $1mm/s^2$, this can cause a drift in the estimate of the position of few meters in some seconds. This is totally a non negligible error. This example is obviously extreme since we are using data that need a double integration for estimating the position of the robot, but it makes clear how important is to solve the loop closing problem. It is obvious that we need a part of the algorithm which try to check if the robot is observing an already visited position and once he found such a correspondence it is able to optimize again the entire map and trajectory obtaining a coherent representation of the environment. This part of the software is called loop closing algorithm and it is always present in a complete SLAM library.

1.1.4 Mapping or Reconstruction

We can talk in a dedicated section about the mapping part of the algorithms in particular because the type of maps that a SLAM algorithm can produce are very different and the type of map that we can choose depend on the objective of our algorithm. One first distinction between possible maps is the following

- *Sparse:* a sparse map is a map in which the position of only few point of the environment is estimated. This type of map are really light and easy to manage and they are suitable for localization purposes and an initial general description of relevant point of the environment. The problem with this type of map is that sparse map are not navigable because of the sparsity of the map representation. In fact usually points that are represented in a sparse map are recognizable points (features). A white wall for example will probably not present in a sparse representation because it is devoid of relevant features.

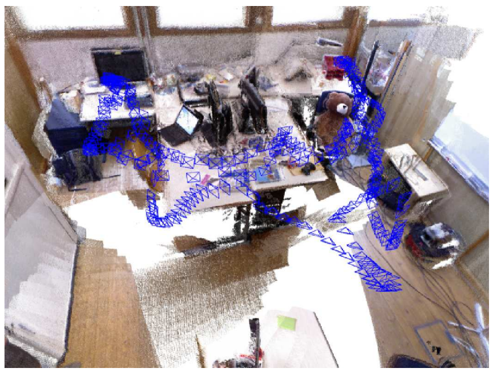
- *Dense:* a dense representation of the map is a representation of the environment in which a really high number of points are used to build a complete map of the environment which can be represented as a dense point cloud, a voxel occupancy grid, a set of 3D meshes etc. This type of maps are usable for navigation but since they contains a higher level of information, they are more difficult to manage in particular from the point of view of memory occupancy.

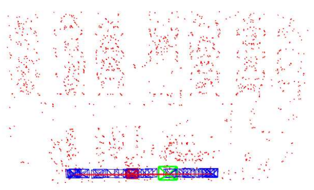
- *Semi-dense:* This type of maps are a compromise between the first two type of maps

Another really general and really important distinction consider the logical structure of the map. In fact in general we can have:

- *Metric Maps:* These are memory expensive maps, but they are suitable for navigation. This type of representation try to build a metric scale model of the environment and they can be 2D or 3D. The choice of a three-dimensional of bi-dimensional map strongly depend on the application for which we are going to use it. For example a mobile robot on wheels can simply used a 2D map of the environment. Instead, if we want to build a map for a flying drone we will probably prefer a 3D map of the environment.

- *Topological Maps:* These other types of maps are really light since they tried to collect only the most relevant information of the environment. This


(a) Example of dense map that can be produced by a ORB-SLAM2. Image taken from [15]


(b) Example of sparse map produced by ORB-SLAM

representation are graph with nodes connected by edges which symbolize the relation between two points of the map. These map are usable for localization purposes but not for navigation.

Now that we have defined these distinctions between maps types it is clear which type of map we will need for our project. In particular we will use a map which is dense and in metric scale which is the right choice for navigation purposes. Actually our algorithm will use two maps at the same time. The first map, produced by ORB-SLAM3, will be used for localization and rescaling purposes, the second one produced by OctoMap using the rescaled version of the point clouds produced by the neural network will be used for navigation purposes.

1.2 Important sensors and SLAM algorithms

The goal of this section is to describe to the reader the most important sensors that can be used to build a SLAM algorithm and the corresponding most known algorithms that are really famous inside the SLAM community. I want to underline that only the most relevant sensors that I have encounter during the work on this project will be discussed. In fact we can build SLAM algorithms also using data coming from a small sonar for example, however sonars are not used so often to build this type of algorithms. I want also to remember that the performance of a SLAM algorithm depend on the application and a particular algorithm could performance better than another one depending on the situation. Finally I want to underline the fact that this thesis is focused on visual SLAM so the vast majority of algorithm that I have encounter are based on different type of cameras.

1.2.1 Monocular cameras (-accelerometer)

Monocular cameras are ubiquitous nowadays. In a modern smartphone you can find even four of them. Cameras are cheap and they collect a lot of information about the environment. Cameras will be explored in more details in the chapter dedicated to visual SLAM but we can state the most important specifications that are relevant for our objective

- *framerate:* the framerate is fundamental because monocular SLAM algorithms assume straight baseline between frames
- *lens:* usually a fish-eye lens is well regarded since it can increase the field of view of the camera
- *RGB/BW:* SLAM algorithms usually work with black and white images but in our case the neural network will need an RGB image as an input, so we will need an RGB camera.
- *rolling/global shutter:* a global shutter camera will be usually preferred, but rolling shutter cameras are really cheaper and could guarantee decent performance

In general pure visual SLAM algorithm based on monocular cameras could be really effective but they can found difficulties in featureless environment and with pure rotation movements. Moreover they can reconstruct the environment only in a non metric random scale which will usually depend on the algorithm initialization. All these problems will be better explained later but it is sufficient to know that there exists multiple way to recover the true scale of the environment usually integrating some other sensor in the system. One of the best way to do that is to introduce an accelerometer. This type of algorithms are called visual-inertial SLAM algorithm and one of them will be the central part of this thesis. There are multiple visual SLAM library which are really often open source. The two most important ones that have been encountered during this project are

ORB-SLAM. This algorithm was a pure feature based visual SLAM algorithm in its first iteration [4], then it add support to depth and stereo cameras with its second version [15]. With the third update called ORB-SLAM VI [5] also the support to acceleremeter which introduce the capability to build metric scale map of the environment even when only a monocular camera is used. The last version of the algorithm ORB-SLAM3 will be the center of the third chapter of this thesis being one of the fundamental element of this work.

LSD-SLAM. This second algorithm [6] differs from ORB SLAM because it is a direct visual SLAM program and because it is able to build semi-dense map. That is, it produce map of the environment with a really higher number of points with respect to the first algorithm mentioned in this section. This algorithm is also really light and there are successive version [7] [16] which can integrate the accelerometer to recover the true scale of the world.

1.2.2 Stereo cameras

In general monocular SLAM algorithm try to estimate 3D position of relevant points in the image doing a triangulation between two successive frame in which a particular point is present. Stereo cameras are simply doing the same thing but instead of doing triangulation between two frames taken by the same camera in different moment in time, we have two (or more) synchronize monocular cameras. The principal advantage in this case is that the relative pose of cameras is fixed and known. So in this case the scale ambiguity problem does not exists. Moreover pure rotation will be not a problem for stereo camera based algorithms. The principal drawback is that the sensor will be more expensive and usually bulkier. In fact the precision of the depth estimation will depend (in addition to the definition of the cameras) also to the distance between them. To achieve good precision in an outdoor environment we will need really high distance between cameras, which is possible on a car but not in a drone for example. Both LSD-SLAM and ORB-SLAM have version of themselves that can work with stereo cameras but there are other algorithm such as

S-PTAM. This algorithm principal characteristic is to highly exploit the parallelizble nature of the SLAM problem [17] [18]. It is important to notice that all the mentioned algorithm try to exploit this fact also dividing the tracking, mapping and optimization problems in different thread. We need to remember that high optimization of the code is fundamental since we are talking about really complex algorithm that need to work in real time.

1.2.3 RGB-D cameras

RGB-D cameras work in a really different way with respect to the previous mentioned sensors. In fact they can recover the depth of points in the image using a totally different principles. There are in general two type of RGB-D cameras.

- the first type of depth cameras are the one based on the structured light

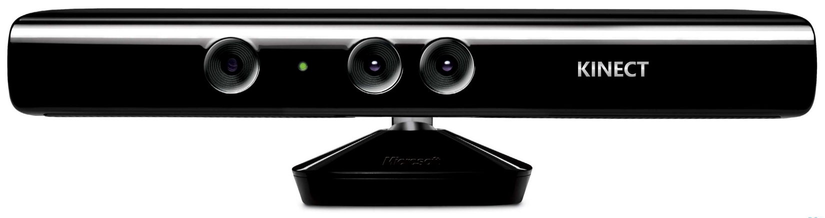


Figure 1.4: The Microsoft Kinect V1. We can see the presence of an RGB camera, an infrared light projector and an infrared light receiver

principle. These type of camera like the $KinectV1$, project a known infrared pattern in the environment and they can recover the distance of the points based on the distortion of the known pattern.

- the second type of depth cameras are based on the Time of Flight principle (in fact they are called TOF sensors), and they estimate the distance of points based on the time needed by infrared light to come back to the receiver.

RGB-D cameras are really effective in indoor environment since they relieve the SLAM algorithm from all the needed computation to estimate the depth of points. The output of this type of sensor could be an already prepared point cloud that can be given to any type of SLAM algorithm that can accept them. These cameras can be also really cheap. The kinect V1 for example cost only around 15 euros. The problem with them is that they cannot be used in outdoor since they have usually limited range and they can suffer the presence of other infrared light source like daylight. There exist variations of both ORB-SLAM and LSD-SLAM able to use RGB-D cameras but we can also mention

RTAB-Map. This algorithm is really famous it can work also with stereo cameras and it is even available for iOS and Android [19]. Obviously you will need a device with an incorporated depth/stereo cameras or 3D lidar.

1.2.4 2D and 3D Lidar

With 2D/3D Lidar the distance of the objects measure the time interval between the emission of the laser pulse and its reception. The difference with TOF RGB-D cameras is that they use infrared light instead of laser pulse. 2D Lidar are usually rotating laser pulse emitter and receiver (as the one at our disposal). They can be really suitable for indoor 2D mapping since they have a really bigger range and higher precision than RGB-D cameras. They can have problems with reflecting surface, they are really expensive and since they have a moving part they can be prone to mechanical failure. Moreover the vibration caused by the rotations could be not negligible especially if the Lidar is mounted on a Drone. There are multiple SLAM algorithms able to exploit the information coming from 2D Lidar but two in particular are rally famous especially in the open source community

Hector SLAM. This is so famous that there exist a direct installation for ROS [20]. It is really simple to setup and it can achieve really good performance. It require really low computational resources and it produce a 2D occupancy grid of the environment at different resolution. It can also be augmented to achieve 3D capability using data coming from IMU, GPS, or other similar sensors.

Google Cartographer. This algorithm work both with 2D and 3D Lidar and it can achieve really high performance and its main characteristic is to be able to perform loop closing from Lidar data with really low computational resources [21].

At the end of this section there are in particular two concepts to bring home

- a lot of SLAM algorithm are compatible with different sensors. This means that once you have build a particular type of backend for the algorithm you can adapt the frontend to achieve compatibility with different type of hardware

- The main discriminant when you are making the first decisions on how to build your SLAM system is to decide if you want to use your system in an indoor or an outdoor environment. In fact as we have seen some sensor can be really efficient in an indoor environment but non performing in outdoor situation, or vice-versa

name	light sens.	output	preferred env.
monocular	medium	RGB image	indoor/outdoor
stereo	medium	multiple RGB images	indoor/outdoor
RGB-D	high	point cloud	indoor
2D Lidar	low	planar distances	indoor
3D Lidar	low	point cloud	outdoor

Table 1.1: Table that sum up the main possible sensors used by SLAM algorithm

1.3 Mathematical tools

This third section of the first chapter is dedicated to the two principal mathematical tools needed for SLAM algorithm. In particular we will take a look on how the 3D pose of the robot can be represented in a mathematical way and, after that, we will take a look at the basics of non linear optimization. Before explain these two topics we will take a look at the most basic mathematical formulation of the SLAM problem.

1.3.1 Mathematical formulation of SLAM problem

In this first section we will see how the SLAM problem can be formulated as a simple state estimation problem. Before doing that we need to define some of the variable that we will use for the formulation of the problem.

- $\boldsymbol{x}_k$ represent the pose of the robot at time k
- $\boldsymbol{y}_j$ represent the state of the $j'th$ landmark (relevant point of the environment)
- $\boldsymbol{u}_k$ represent the input to the system at time k. This variable is usually used to take in consideration odometry data coming for example from accelerometeer or rotary encoders
- $\boldsymbol{z}_{k,j}$ represent the observation of the $j'th$ landmark given that the actual state of the robot is $\boldsymbol{x}_k$
- $\boldsymbol{w}_k$ is the noise affecting the relation between the state at $k-1$, the input at time k and the state $\boldsymbol{x}_k$
- $\boldsymbol{v}_{k,j}$ is the noise affecting the observation of landmark j given that the robot is in the state $\boldsymbol{x}_k$
- $\mathcal{O}$ is the set of (k, j) that represent at which pose a particular landmark has been observed
- K is the last time step that we are considering

Once we have defined these variables we can describe the two fundamental equation that we need.

$$\begin{cases} \mathbf{x}_k = f\left(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k\right), \quad k = 1, \cdots, K \\ \mathbf{z}_{k,j} = h\left(\mathbf{y}_j, \mathbf{x}_k, \mathbf{v}_{k,j}\right), \quad (k, j) \in \mathcal{O} \end{cases} \tag{1.1}$$

In this system of equation the first line represent the so called *motion equation*. The function f in fact connect the actual state of the robot with the previous one depending on the input and the noise that affect our system. The function f is really often a non linear function. The second equation represent the *observation equation* which connect through the (usually also non linear) function h, the measurement of landmark $\boldsymbol{y}_j$ obtained from the sensors, given that the robot is in state $\boldsymbol{x}_k$ and that the measurement system is affected by the noise $\boldsymbol{v}_{k,j}$. Now that we have built this equations that govern the time evolution of our robot pose taking in consideration the position of the landmarks (which are assumed to be static in the vast majority of SLAM algorithm) and the input to the system we can clearly see how the SLAM problem is simply a *state estimation* problem. The quantities that we want to estimate are the pose and trajectory (collection of poses through time) of the robot and the position of the landmarks. To solve these problems we can use a more classical approach which imply the use of filters like the Extended Kalman Filter (EKF), otherwise we can try to solve it using more modern non linear optimization technique. These optimization technique will be better explained at the end of this section.

1.3.2 Pose representation

The first thing that we need to do is to better define what is the state of the robot $\boldsymbol{x}_k$. In the SLAM framework the state of the robot correspond to the actual pose of the robot inside the map. To represent the pose we can use the rototraslation between the fixed world frame and the robot frame. To see how these rototraslation can be represented we can talk about the translation and the rotation independently.

Traslation. Representing a 3D translation is really simple and we can do that simply using a three-dimensional vector $\boldsymbol{t} \in \mathbb{R}^3$. This is simply the vector connecting the origin of the world frame and the origin of the robot reference frame.

Rotation. Rotation are usually represented using rotation matrix. For 3D rotation we simply have 3x3 rotation matrix which have two particular characteristic. They are orthogonal and their determinant is equal to one since they represent a pure rotation and they cannot stretch our vector. So we can define the special orthogonal group which contains all the possible rotation matrices of an n dimensional space as follow

$$\mathrm{SO}(n) = \left\{\mathbf{R} \in \mathbb{R}^{n\times n} \mid \mathbf{R}\mathbf{R}^T = \mathbf{I}, \det(\mathbf{R}) = 1\right\} \tag{1.2}$$

Note that since rotation matrices are orthogonal $\boldsymbol{R}^T$ represent the inverse rotation with respect to $\boldsymbol{R}$. Multiplying a rotation matrix by a particular vector gives us the rotated version of that vector in the same reference frame or the coordinates of the same vector in a rotated reference frame.

To obtain the coordinate of a particular vector $\mathbf{a}$ in any rototrasleted reference frame we simply need to multiply it by the specific rotation matrix and then add the translation vector. We need to be particularly careful with the translation vector since we need to have really clear in mind in which reference frame the translation vector is represented. Now that we have a description of a complete rototraslation we can try to represent it using a single matrix. We can do that simply using homogeneous coordinates. To do that we simply add a 1 at the bottom o f our 3D vector and at this point we can represent a complete rototralsation with a single Transform matrix $\mathbf{T}$ as follow

$$\begin{bmatrix} \mathbf{a}' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ 1 \end{bmatrix} = \mathbf{T} \begin{bmatrix} \mathbf{a} \\ 1 \end{bmatrix} \tag{1.3}$$

the advantage of using transform matrix is that now we have a linear relation between the coordinates of a vector and the coordinates in the rototraslated frame. This is really important since now we can compose multiple rototraslation in a really streightforward way. As before we can define the *special euclidean group* as follow

$$\mathrm{SE}(3) = \left\{\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} \mid \mathbf{R} \in \mathrm{SO}(3), \mathbf{t} \in \mathbb{R}^3\right\} \tag{1.4}$$

this group contains all possible transform matrices that could be applied to the homogeneous version of three-dimensional vectors. Representing rototraslation using transform matrix is inefficient because rotation matrix has some constrain and they have nine degree of freedom even if a 3d rotation has only 3 DoF. So we can use other way to represent 3D rotation.

Rotation Vectors. One simple way to fully express a rotation is to use a vector which direction is parallel to the axis of rotation and the length is equal to the angle of rotation. The relation between rotation vectors and rotation matrices is expressed by the *Roudrigues' formule* wich is reported here only for the reader knowledge

$$\mathbf{R} = \cos\theta\mathbf{I} + (1-\cos\theta)\mathbf{n}\mathbf{n}^T + \sin\theta\mathbf{n}^\wedge \tag{1.5}$$

where $^\wedge$ represent the conversion of a vector in skew-symmetric matrix obtained as follow

$$\mathbf{a}^\wedge = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}^\wedge = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{1.6}$$

Euler angles. Euler angles are really intuitive since they decompose a 3D rotation in three different rotation around three axis. The most important thing to consider is that the axis of rotation and the order in which these rotation must be applied need to be clearly defined. Fortunately depending on the field these notion are usually a standard. The most important problem of Euler angles is the famous *Gimbal Lock* problem, which cause the loss of one degree of freedom for the rotation because of alignment between two rotation axis caused by the first rotation.

Quaternions. We have seen that rotation matrix are redundant and that Euler angles (or any other three dimensional representation of a three dimensional rotation) suffer of singularity. However there exist another possible representation of rotation using an extension of imaginary numbers called quaternions. Quaternions are formally elements of a non commutative division algebra but we can give a more practical description of them if we consider them as an extension of complex numbers. A quaternion is an entity such as

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k \tag{1.7}$$

where q_0, q_1, q_2, q_3 are real numbers and i, j, k are imaginary units that satisfy the following equations

$$\begin{cases} i^2 = j^2 = k^2 = -1 \\ ij = k, ji = -k \\ jk = i, kj = -i \\ ki = j, ik = -j \end{cases} \tag{1.8}$$

we can use quaternions to represent any 3D rotation and in fact there exist a map from quaternion to rotation matrix. Moreover to apply the rotation to a 3D vector we simply need to extend a 3D point as an imaginary quaternion and then left multiply it by the quaternion and right multiply it by the inverse of the quaternion.

1.3.3 Lie Group, Lie Algebra, derivatives

Since we have just shown how to represent the pose and we know that the vast majority of time we will need to solve optimization problem where the unknown are poses (of the camera or of the landmark in the 3D space) we can surely note that there is a particular problem. Take for example the group of rotation matrices equipped with the classical matrix multiplication. Since it is a group it is close under the matrix multiplication operation, but for example the sum of two rotation matrix do not return a rotation matrix. This is a problem when we deal with optimization problem where the unknown is for example a rotation matrix. In fact to solve this type of problem a notion of derivative is needed and the notion of derivative need some kind of addition and subtraction operation to represent small variations, and so define the derivative operation. To do that the following elements will be introduced in this section

- Lie group
- Lie algebra
- exponential/logarithmic map
- addition/subtraction operation and right derivative

it is worth noticing that these concepts are used so often in the SLAM community, and in general in real robotics implementation, that there exists a $C++$ library called Sophus which is built for the creation and manipulation of these type of objects. The name of the library correspond to the name of the mathematician that develop the theory behind all these mathematical tools which was Sophus Lie.

Lie group. We first need to define what a group is. A group by definition is a couple $(\mathcal{G}, *)$ where $\mathcal{G}$ is a set and $*$ is a binary operation between elements in the set. The set and the operation are such that for every element $\mathcal{X}, \mathcal{Y}, \mathcal{Z} \in \mathcal{G}$

closure: $\mathcal{X} * \mathcal{Y} \in \mathcal{G}$

identity element: $\mathcal{E} * \mathcal{X} = \mathcal{X} * \mathcal{E} = \mathcal{X}$

inverse: $\mathcal{X}^{-1} * \mathcal{X} = \mathcal{X} * \mathcal{X}^{-1} = \mathcal{E}$

associativity: $(\mathcal{X} * \mathcal{Y}) * \mathcal{Z} = \mathcal{X} * (\mathcal{Y} * \mathcal{Z})$

a Lie group is not only a group but it also has the proprieties of a differentiable manifolds. The idea that one need to understand here is that locally the group could resemble a Euclidean space. An important propriety of Lie group is that they have the same local structure everywhere, due to the group properties. This characteristic is simple to visualize if we think about the group compose by the set of imaginary numbers of norm one paired with the multiplication between imaginary numbers. The set can be represented in the imaginary plane as a circle which has the same local structure everywhere. For this reason also the tangent plane to the Lie group has the same proprieties everywhere. A really important practical characteristic about Lie group is that we can define the group action on element of a set $\mathcal{V}$ as follow

$$\cdot : \mathcal{M} \times \mathcal{V} \rightarrow \; V : (\mathcal{X}, v) \rightarrow \; X \cdot v \tag{1.9}$$

which need to satisfy the following proprieties

$\mathcal{E} \cdot v = v$

$(\mathcal{X} * \mathcal{Y}) \cdot v = \mathcal{X} \cdot (\mathcal{X} \cdot v)$

to explain what this group action is we can simply take as an example the group of 3D rotation matrix $SO(3)$. This group is able to act on elements of the set $\mathbb{R}^3$ through the usual matrix vector multiplication.

Lie algebra. The fact that the Lie group has the proprieties of a differentiable manifold allow us to have a well define tangent space in each point of the Lie group. We can represent the tangent space to the Lie group $\mathcal{M}$ at point $\mathcal{X}$ as $T_{\mathcal{X}}\mathcal{M}$. Now we can define the Lie algebra as the tangent space at the identity element of the group $\mathcal{E}$. That is, the Lie algebra associated to the Lie group $\mathcal{M}$, denoted by $\boldsymbol{m}$ is by definition

$$\boldsymbol{m} := T_{\mathcal{E}}\mathcal{M} \tag{1.10}$$

the Lie algebra is a vector space so we can choose a base and build two isomorphisms (one the inverse of the other) usually called *hat* and *vee* which create a one-to-one correspondence between the tangent space and the calssical *Cartesian vector space* $\mathbb{R}^m$. These two operation are usually simbolyzed respectively by the symbols $\wedge$ and $\vee$. So when we write $\tau^\vee$ we means the element of the tangent space that correspond to the vector τ in the corresponding vector space.

Exponential and logaritmic map. A correspondence between element in the Lie algebra and element in the Lie group could be created considering the time-derivative of the of the group constrain. Take as an example the rotation matrix group. The constrain for this group is

$$\boldsymbol{R}(t)\boldsymbol{R}(t)^T = \boldsymbol{I}$$

and when we time-differentiate it we obtain

$$\dot{\boldsymbol{R}}(t)\boldsymbol{R}(t)^T + \boldsymbol{R}(t)\dot{\boldsymbol{R}}(t)^T = 0$$

so we can notice that $\dot{\boldsymbol{R}}(t)\boldsymbol{R}(t)^T$ is a skew-symmetric matrix and if we consider $\boldsymbol{R}(t) = I$ we obtain that

$$\dot{\boldsymbol{R}}(t) = \tau^{\vee}$$

$\tau^{\vee}$ is a three dimesnional vector in the tangent space at the identity element of the group, so in the Lie algebra. Considering the solution of the last differential equation we can find the expression of the exponential map and of its inverse, the logaritmic map, which have the following expression

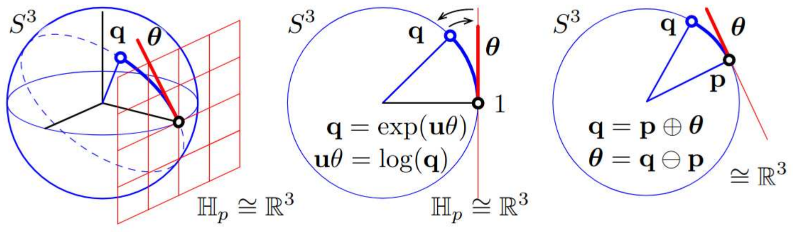
$$\begin{aligned} \exp: \quad & m \to \mathcal{M} \quad ; \quad \tau^{\wedge} \mapsto \mathcal{X} = \exp\left(\boldsymbol{\tau}^{\wedge}\right) \\ \log: \quad & \mathcal{M} \to m \quad ; \quad \mathcal{X} \mapsto \tau^{\wedge} = \log(\mathcal{X}) \end{aligned} \tag{1.11}$$

To visualize how these maps work we can look at figure 1.5, where the geometric representation of the set of unit quaternion as a sphere is given. In that figure we can see how the Lie algebra is the tangent space at the identity element. Moreover we can see how the exponential map is acting as a wrapping function which take an element in the tangent space and "wrap" it around the sphere. Vice-versa the logarithmic map unwrap elements of the Lie group mapping them directly in the tangent space.

Addition, subtraction operation and right derivative. Now that we have define the Lie algebra corresponding to a particular Lie group we can define operations between elements of the Lie group and Lie algebra as follow

$$\begin{aligned} right - \oplus: \quad & \mathcal{Y} = \mathcal{X} \oplus {}^{\mathcal{X}}\boldsymbol{\tau} := \mathcal{X} * \operatorname{Exp}\left({}^{\mathcal{X}}\boldsymbol{\tau}\right) \in \mathcal{M} \\ right - \ominus: \quad & {}^{\mathcal{X}}\boldsymbol{\tau} = \mathcal{Y} \ominus \mathcal{X} := \log\left(\mathcal{X}^{-1} * \mathcal{Y}\right) \in T_{\mathcal{X}}\mathcal{M} \end{aligned} \tag{1.12}$$

the first thing that we need to notice is that we have defined the right-addition



S^3
$\mathbf{q}$
$\boldsymbol{\theta}$
$\mathbb{H}_p \cong \mathbb{R}^3$
1
$\mathbf{q} = \exp(\mathbf{u}\theta)$
$\mathbf{u}\theta = \log(\mathbf{q})$
$\mathbf{p}$
$\mathbf{q} = \mathbf{p} \oplus \boldsymbol{\theta}$
$\boldsymbol{\theta} = \mathbf{q} \ominus \mathbf{p}$
$\cong \mathbb{R}^3$

Figure 1.5: Visualization of the Lie group of unit quaternion the corresponding Lie algebra and the exponential and logarithmic map. Image from [22]

(and subtraction). This is important since if we have the first element of the addition operation, equal to an element of the Lie algebra, differently from the above expression, the definition of the addition operation will change. Another thing to explain is the symbol ${}^{\mathcal{X}}\boldsymbol{\tau}$ that represent the element of the Lie algebra but collocated in the tangent space corresponding to the $\mathcal{X}$ element of the Lie group. Now that we have define the right addition and subtraction operation we can define the right Jacobians as an extension of the the usual derivative operation as follow

$$\frac{{}^{\mathcal{X}}Df(\mathcal{X})}{D\mathcal{X}} := \lim_{\boldsymbol{\tau}\to 0} \frac{f(\mathcal{X} \oplus \boldsymbol{\tau}) \ominus f(\mathcal{X})}{\boldsymbol{\tau}} \tag{1.13}$$

more deep discussion about Lie algebras and Lie groups applied to robotics can be found in [22]. For reader who are interested in a formal mathematical description of these concept [23] and [24] could be really good resources.

1.3.4 Non linear Optimization

The last mathematical tools that need to be introduced is the Gauss-Newton and the Levernberg-Marqatdt method for non-linear optimization problems. In fact this is one of the most used approach to solve strongly non linear least square problem usually present in SLAM algorithms such as Bundle Adjustment. The Gauss-Newton approach is basically based on the first order approximation of the error function, and not of the squared norm of the error. Suppose that we want to find the solution of the following problem

$$\min_{\boldsymbol{x}} \quad F(\boldsymbol{x}) = \min_{\boldsymbol{x}} \quad \frac{1}{2}||f(\boldsymbol{x})||_2^2 \tag{1.14}$$

as every numerical method, since $f(\boldsymbol{x})$ is usually non linear, we want start from an initial guess and then compute the change $\Delta\boldsymbol{x}$ which allow us to get closer to the solution. In the Gauss-Newton approach we consider the first order approximation of $f(\boldsymbol{x})$ and then we compute the change as follow

$$\Delta\boldsymbol{x} = \arg\min_{\Delta\boldsymbol{x}} \quad \frac{1}{2}||f(\boldsymbol{x}) + \boldsymbol{J}(\boldsymbol{x})^T\Delta\boldsymbol{x}||^2$$

where $\boldsymbol{J}(\boldsymbol{x})$ is the Jacobian of $f(\boldsymbol{x})$. We can solve this minimization problem simply computing the derivative and then imposing it equal to zero. As a result we obtain the following normal equation

$$\underbrace{\mathbf{J}(\mathrm{x})\mathbf{J}^T}_{\mathbf{H}(\mathrm{x})}(\mathrm{x})\Delta\mathrm{x} = \underbrace{-\mathbf{J}(\mathrm{x})f(\mathrm{x})}_{\mathbf{g}(\mathrm{x})} \tag{1.15}$$

at this point the only thing that we need to do is to solve this equation at each step of the algorithm. The problem is that solving this equation means that $\boldsymbol{H}$ needs to be invertible. Since we cannot guarantee this, the methods will not surely converge. The levenberg-Marqardt method is only a modified version of the Gauss-Newton method in which we solve the same optimization problem to find $\Delta\boldsymbol{x}$ but with the difference that the problem is constraint inside a *trust region* which dimension depend on how good the first order approximation of the function is.

1.4 Brief History of SLAM

The aim of this last section is to summarize the history of SLAM and the information are principally taken from three really important papers [10] [11] [12]. The main division of slam history given in [10] is the following

- classical age
- algorithmic age
- the future

these three ages will be briefly discussed in this section. Before doing that I can report the answer of the last cited paper to the question *"is SLAM a solved problem?"*. This is a really interesting question since when you first dig into the field, the SLAM problem seems a really open one but the more you study it the more you have the impression that the problem could be considered already well explored. The answer of the authors of the papers (with which I agree) is the following: even if in really strict condition and in quite controlled environment the SLAM problem could be considered solved, a lot of work need to be done to find general and in particular robust solution. The robustness is one of the main subject of today research and will be part of the last portion of this section.

1.4.1 Classical Age 1986-2004

The first event in which probabilistic SLAM appears was the 1986 IEEE Robotics and Automation Conference held in San Francisco. As always, the first step that one need to take in consideration when dealing with a new problem is to formulate the problem in the correct way. For SLAM the probabilistic representation was really the key aspect since it is a problem in which noises and uncertainties have such a great impact. As underline in [10] the most important conceptual breakthrough in this first period of the SLAM history was the understanding that the estimation problem of the position of the robot and of the landmark, has a convergent solution, if they are considered simultaneously. This structure of the SLAM problem was described only in 1995 [25]. It was in this classical period where the principal approaches to solve the problem were proposed like
Extended Kalman Filters. In this case the motion and observation equation are linearized to allow the use of the classical Kalman Filter.
Particle Filters. These filters try to deal with non linearity without the need of linearization of the system. To give a simple practical explanation on how a particle filter work we can summarize the step followed by them in a simple localization task

- first they sample the actual probability distribution representing their beliefs on the position of the robot
- they assign different credibility to each of the sample based on the real measurement coming from the sensor
- they update the probability distribution representing their beliefs
- they repeat the above steps until they converge to the real position of the robot

Maximum Likelihood estimator. Which try to maximize the posterior distribution

In these first twenty years the SLAM research community delineated the general SLAM problem and the principal challenges related to the real implementation of an algorithm that try to solve it

1.4.2 Algorithmic Age 2004-2015

The second period of SLAM history was dedicated to the development of more robust theoretical foundation concerning topic such as convergence and consistency of the problem solutions. Moreover the typical structure of SLAM algorithm described above started to become mainstream. In particular the part of the algorithm that we now consider part of the backend were developed. In [10] there is a table reporting the most important papers published about SLAM during this particular period. An important aspect of this period is that papers published during these years start to concentrate on a particular aspect of the SLAM algorithm. In fact, as mentioned above, SLAM is a really multidisciplinary field and after twenty years from the beginning of the the studies on it, making improvement on a particular aspect is possible only being a real expert of that particular problem. It is important to notice that the vast majority of library used in SLAM start to be built in this period.

1.4.3 The future of SLAM research

In the last part of [10] the possible future development of SLAM research field are deeply described. In general the desired improvement are the following:
Robustness. As mentioned before SLAM algorithms need to work with data coming from noisy sensors. Moreover they usually work better in controlled environment. Take as an example a feature based visual SLAM algorithm which need visual features in the field of view of the camera to be able to locate itself. One of the open problem in SLAM is how to decrease the rate of failure of these algorithms making them more robust to noisy data and different type of environment. Another thing to notice is that usually we make the assumption of static environment which is usually not true in the real world. So another improvement concerning the robustness of the algorithms is to make them reliable also in environment that could change a little bit during time.

Scalability. SLAM algorithm can be really demanding from the point of view of computational power, in particular when the size of the optimization problem that we need to solve became really high. Moreover they can be demanding also from the point of view of memory since the space occupied by the map can increase unboundly over time. So the second aspect that can be the focus of future SLAM research can be the production of more efficient way to solve minimization problem and to represent the map.

Map Representation. The map not only need to be light but also it can include semantic information about the environment. This is not usual in available SLAM algorithm. Moreover another possible improvement of maps produced by SLAM algorithms is the representation of it as a set of distinct object rather than a collection of 3D point or a single mesh.

Improvement in theoretical tools. An improvement on any theoretical or mathematical tools that is used to solved the SLAM problem can have a big impact on the community. The main theoretical contribution can involve the improvements of solvers for minimization problems and in particular way to detect wrong local minima problem solutions.

Active SLAM. The final possible future development mentioned in the paper is the possibility of create Active SLAM algorithm which not only act in a passive way, only estimating the map and the position, but also actively, guiding the robot in the environment exploration.

Chapter 2

VisualSLAM

This second chapter is dedicated to the general description of the most basic way to build a Visual SLAM algorithm. It will be structured in the following way:

- At the beginning the very basic information about how images are represented in computers, and about the basic pinhole model of the camera will be described
- In second place the way in which the camera need to be calibrated will be explained and the particular camera that we are using in this work (Raspicam V2) will be described
- the accelerometer structure and in particular the MPU used in this work (MPU 6050) will be described. Moreover the way in which the motion processing unit need to be calibrated will be discussed
- basic concept of computer vision will be described
- other important concept needed to understand the function of a visual SLAM algorithm will be explained (Pose estimation, Bundle Adjustment, Bag of Words)

This chapter is intended to build basic intuition about how visual SLAM algorithm work since in chapter 3 ORB-SLAM3 will be analyze in more depth.

2.1 Image representation and camera model

In this first part we need to describe how the principal information that we need to elaborate is represented. Moreover we need also to describe the main sensor that we will use and how it is mathematically described.

2.1.1 Images

Images are usually represented in computers as matrices in which each entries contains information about the corresponding pixel of the image. This information depends on the type of image that we are considering. In particular we have encountered three type of images during this work.

Colored images. In this images each entry of the matrix contains the intensities of the three color component of a particular pixel, so the blue, green and red components. It is important to take into account that the correct order of the color must be selected. There are in fact function that have as default option the BGR representation such as the *imshow* function of OpenCV, while other software components, such as the neural network that we will use for the depth estimation, will assume that the input image is in RGB format. Typically each color intensity is represented using eight bits.

Grey scale images. Grey scale images are formatted as colored images but each entry of the matrix contain eight bit of information only about the grey intensity captured by the camera in that particular pixel. Grey scale images are often really less heavy than the corresponding colored one and they contain enough information to be used by visual SLAM algorithm to compute descriptors and recover the movement of the camera. Neural network for depth estimation will need the colored image because since this images contains much more information about the environment it is simpler for the network to learn how to estimate depth.

Depth Images. This images are equal to greyscale images but each entries usually contains 16 bit of information which are enough to represent depth in millimeter up to 65 meters. Usually if we want to show them we will need to rescale them in such a way that they contains values from 0 to 255.

2.1.2 Pinhole camera model

The second thing that we need to describe is the most simple model that we can consider for a camera. A model of a camera describe the geometric shape of the camera in such a way that we can obtain a mathematical projection of a 3D point to the camera image plane. Here we will consider the pinhole camera model in which the camera is considered to be compose of a small hole in which light pass to be then captured by a CCD or CMOS sensor positioned behind the hole in what we will call the camera image plane. The small hole is usually called the

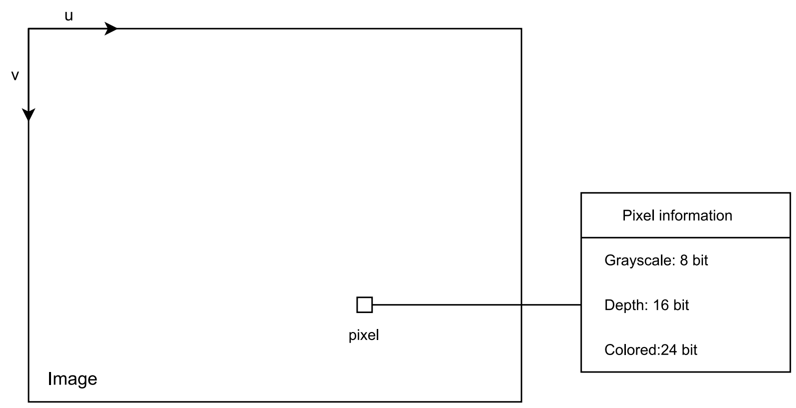


Figure 2.1: Graphic representation on how pixel information is stored for different type of images

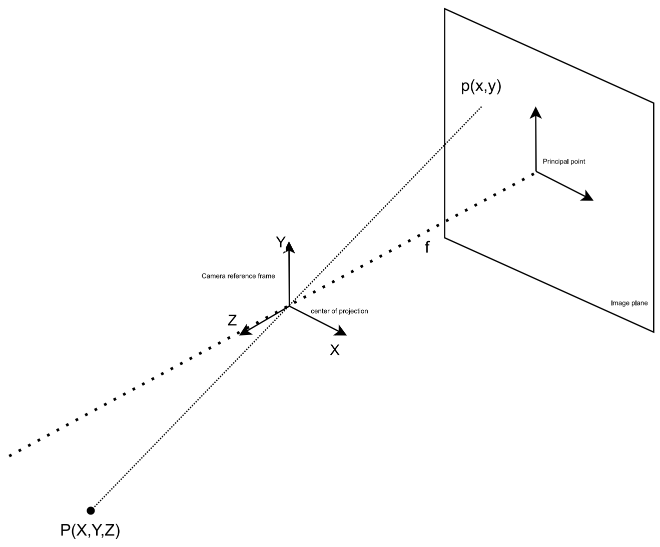


Figure 2.2: Schematic representation of the pinhole camera model

camera optical center (which is the point in lens axis of the camera in which all light ray pass before hitting the image plane). If we consider the camera reference frame (O, x, y, z) and image plane reference frame (O', x', y') due to the triangle similarity we can see that a 3D point $P = [X, Y, Z]$ once projected in the camera reference frame it is represented by the point $[X', Y']$ which can be computed using the following formulas

$$X' = f\frac{X}{Z}, \quad Y' = f\frac{Y}{Z} \tag{2.1}$$

where f represent the focal length. Note that we want to obtain the pixel coordinate $[u, v]$ of the 3D point. To obtain the pixel coordinate we need to shift an scale X' and Y'. We can do that as follow

$$\begin{cases} u = f_x \frac{X}{Z} + c_x \\ v = f_y \frac{Z}{Z} + c_y \end{cases} \tag{2.2}$$

where c_x and c_y represent the translation between the origin of the camera image plane and the pixel coordinate frame, and f_x f_y are the focal lenght f multiplied by the scaling factor of the two axis. At the end we can rewrite that formula as follow

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z} \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \frac{1}{Z}\mathbf{KP} \tag{2.3}$$

the matrix $\boldsymbol{K}$ is called intrinsic camera matrix. The extrinsic camera matrix is instead the rototraslation matrix that describe the position of the camera with respect to the world fixed frame. Moreover there is always a lens mounted on the camera and in SLAM application this is often the case since a fisheye camera is usually welcomed since it increase the field of view of the camera. In this case we need a more complex model to take in consideration also the distortion introduced by the lens. In general the intrinsic camera matrix and the distortion matrix are obtained through the camera calibration, and once the extrinsic matrix of the camera is known we can project any 3D point represented in world coordinate in the camera image plane in pixel coordinate. Note that our objective will be often exactly the opposite, so knowing the pixel coordinate of the point in multiple image we would like to reconstruct the 3D position of the point. Another problem which we want to solve is to compute the position of the camera based on the 3D position of a point and its pixel coordinate in the camera image plane. To be fair we will also be interested in solve both these problem simultaneously.

Camera calibration

Since to use images coming from the camera we need intrinsic and distortion parameters we need to use a procedure to determine them. This procedure is called camera calibration and it works following these steps

- The first step is to provide a fixed recognizable pattern in the environment in which distances between point in the pattern is known. The most used pattern are the following one

  Chessboard. A chessboard is usually used because angle of the squares has really high gradient in the image so they can really simply been identified.

April grid. Alternatively more and more frequently pattern of April tags are used [26]. The most simple pattern is the one in figure 2.3. April tags follow the same concept of QR Code but they are made to be able to encode less information, which could be seen as a disadvantage but it is a desired feature in robotics since in this way they can be identified more rapidly. April grids has some advantages with respect to chessboard in fact for example a frame in which even only a partial part of the grid is present can be used anyway to extract points for calibration, in fact while in a chessboard all the white or black squares are equal in an April grid each square is identified by the April tags. This allow to obtain better calibration since it is simpler to obtain calibration points even in external part of the image which is the place in which the distortion is worse. Another advantage of April grid, due to the same reason, is that the pose of the April grid is fully resolved. This is not true for chessboard since if we rotate a chessboard of 180 degrees we obtain exact the same pattern that we had before rotation.

- The second step is to collect a lot of image in which the pattern is present in a lot of different positions, scales and orientations.

- The third step is arbitrarily defined the world fixed reference frame for example in the higher left corner of the pattern with the z axis coming out from the plane of the pattern and x and y axis parallel to it. Then we need to compute the position of each recognizable point in the pattern (in the case of April grid the angle of the squares) for each image in the word reference frame. This is something that we can do since the real world distance between them is known

- As the fourth and last step since we know the 3D position in the world frame of each feature point we can estimate the distortion and intrinsic parameter and the intrinsic matrix choosing the ones that allow us to project the 3D points in the correct position in the image plane

To calibrate our camera we use the ROS camera calibration package using a chessboard at the beginning. At the end we finally use the Kalibr package using an April grid. The advantage of using the Kalibr package, particularly when you are dealing with camera-imu setup, will be more clear later. A more in

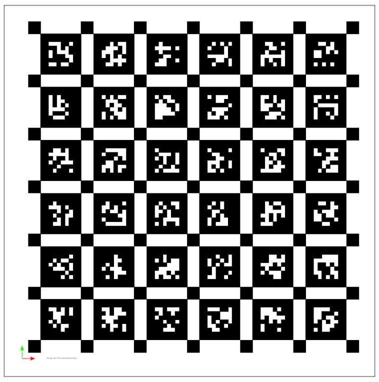


Figure 2.3: Example of an April grid used for calibration

depth description of the theoretical aspect related to the rolling shutter camera calibration used by Kalibr are presented in [27].

2.1.3 Raspicam V2

We will now describe the camera that we have used in our experiment, the Raspicam V2. We will describe only the main aspect of the camera and in table 2.1 more specification will be included. To obtain a complete description of the module you can refer to [28]. The camera is really cheap ($\sim$ 30£) and the underlying sensor is a Sony IMX219 which has $3280x2464$ pixels. The camera is able to shoot videos at different framerates and at different resolutions. Obviously increasing the resolution will decrease the possibility of the robot to record data at an higher framerate. The most important characteristic of this module is that this camera is a **rolling shutter camera**. This means that the sensor is scanned horizontally or vertically so the pixel are read sequentially and not all at the same time (as it happens in **global shutter camera**). The difference in these two type of cameras, and the reason for which the second type cost more than the first one is in the sensor circuit configuration which is more complicated for global shutter camera. Rolling shutter camera are usually a problem for SLAM algorithm since this way of capturing images can cause distortion when there is a rapid object in the field of view of the camera or the camera is moving fast through the environment. Because of these distortion the algorithm is not able to recognize a particular scene because the way in which it will be reproduced in the image will depend on the velocity and direction in which the camera is moving. The last important feature that I want to underline is that the Raspicam has a quite wide field of view as can be seen in Tab 2.1. This is the reason why we use it instead

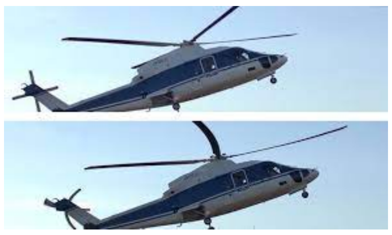


Figure 2.4: Differences between images acquired with a global and with a rolling shutter camera. Frame from a video of the official Sony youtube channel.

Price	$\sim$ 30£
Weight	3g
Sensor Resolution	3280 $\times$ 2464 pixels (8 Megapixels)
Video modes	1080p30, 720p60 and 640 $\times$ 480p60/90
Sensor	Sony IMX219
Sensor image area	3.68 x 2.76 mm (4.6 mm diagonal)
Pixel size	1.12 µm x 1.12 µm
Horizontal field of view	62.2 degrees
Vertical field of view	48.8 degrees

Table 2.1: Principal specificaition of the Raspicam V2

of another global shutter camera at our disposal (PointGrey camera). In fact this second camera has a zoom lens (which is really bad for SLAM purposes) mounted on it and lenses have a really high cost for that type of camera.

2.2 The accelerometer

The accelerometer is the other key component of our setup and it is the reason for which we will talk about visual-inertial SLAM. A really good reference for accelerometers industry and technology history is [29]. The real big shift in accelerometer technology in recent years was caused by the introduction of MEMS (Micro-Electro-Mechanical-System). This silicon based technology allow the production of really cheap and small electro-mechanical devices. Accelerometers nowadays can be so small that they can fit inside a modern earbuds. This section will briefly explain the functioning principles of accelerometers, the procedure needed to calibrate them alone and together with the camera and the specification of the particular MPU (Motion Processing Unit) used in our real world implementation. Note that in this section the word accelerometer is often used even if we are considering the whole MPU which include also the gyroscope.

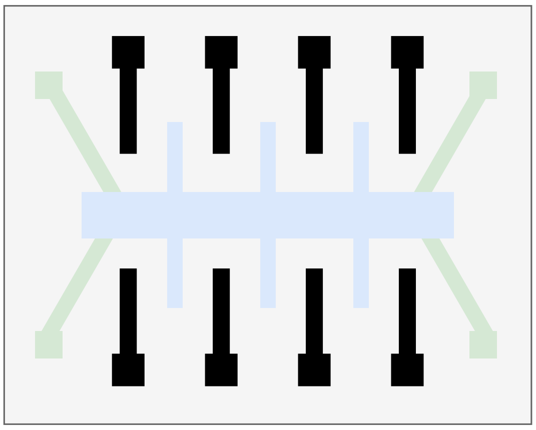


Figure 2.5: Schematic representation of the accelerometer

2.2.1 Physical functioning of the acclerometer

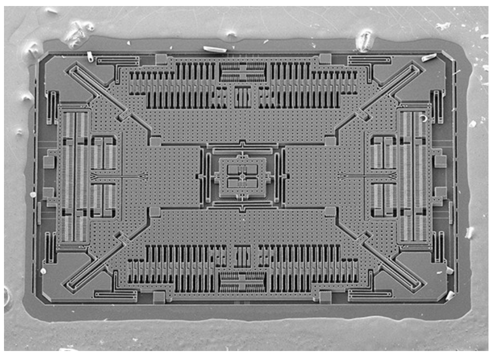
Accelerometer usually provide acceleration but they can also integrate a gyroscope to provide angular velocity data. To refer to the all system (so accelerometer and gyroscope together) it is better to talk about motion processing unit (MPU). Here the functioning of the accelerometer will be first introduced and after that a brief description of the gyroscope will be given.

Accelerometer. The comoponents of a MEMS accelerometer are

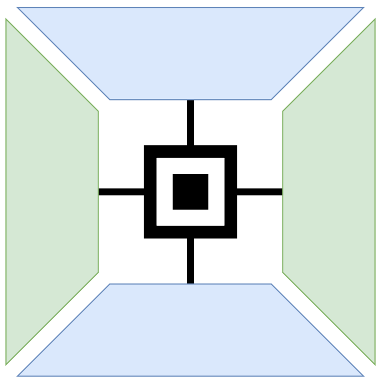
- A suspended mass
- Some fixed plates
- Polysilicon springs to which the suspended mass is attached

since MEMS devices can be produced with similar techniques with which chips are produced, everything is obtained from a silicon wafer. The structure of the device can be seen in figure 2.5 and we can imagine that three of this devices are present in the MPU one for each axis. When you accelerate the device along a particular axis the suspended mass will be subjected to a displacement changing the capacitance between the fixed plates. In this way the acceleration can be detected measuring the voltage variation between the fixed plates.

Gyroscope. The structure of the gyroscope is a little more involved since it is composed of four plates connect with springs to a central anchor. These plates are kept in perpetual oscillating motion along their symmetry axis, in such a way



(a) A microscope image of the L3GD20HTR MEMS gyroscope from ST Microelectronics. Credit Adam McCombs.



(b) Schematic representation of the gyroscope

Figure 2.6: Microscope and schematic representation of a gyroscope

that they almost always posses a velocity different from zero. It is known that if a rotation is applied to the system, depending on the axis in which it is applied, the plates will start moving also along others direction due to the Coriolis force, which act in a direction perpendicular both to the velocity of the plate and the axis of rotation. This displacement can be measure again as a change in voltage and converted in angular velocity values. In image 2.6 both the schematic structure and a real microscope image of a MEMS gyroscope inside an MPU can be seen.

2.2.2 Calibration

As the camera also the accelerometer need to be calibrated for the right functioning of the system. In particular we want to obtain some calibration data that are needed by ORB-SLAM3 to work. The data that we need are the following

- the frequency of the accelerometer (which is fixed by us)
- the noise density of the accelerometer
- the noise density of the gyroscope
- the random walk of the accelerometer
- the random walk of the gyroscope

these data are needed because of the model used to represent the acceleroemter noises both in the package used to calibrate the imu-camera system (Kalibr) and in ORB-SLAM3. In fact in these packages the measurement of the imu are modeled in the same way for each axis of the accelerometer and the gyroscope as follow

$$\bar{\omega}(t) = \omega(t) + b(t) + n(t). \tag{2.4}$$

where $\bar{\omega}(t)$ is the output of the accelerometer or of the gyroscope, $\omega(t)$ is the true value of the acceleration or angular velocity along one of the axis. The other two terms represent the noises that we will obtain from the calibration procedure.

- $n(t)$ is a white Gaussian noise represented by a white Gaussian noise process with variance σ_g, so by definition it has the following characteristics

  $$\begin{aligned} &E[n(t)] \equiv 0 \\ &E\left[n\left(t_1\right) n\left(t_2\right)\right] = \sigma_g^2 \delta\left(t_1 - t_2\right) \end{aligned} \tag{2.5}$$

  σ_g correspond to the noise density of one of the two sensor (accelerometer or gyroscope).

- $b(t)$ which is the one that was previously called the random walk of one of the two sensor. This represent slow variation in the sensor bias which are modeled as Brownian motion. $b(t)$ is obtained integrating a white Gaussian noise with σ_{bg} variance, so it satisfy the following equation

  $$\dot{b_g}(t) = \sigma_{bg}\omega(t) \tag{2.6}$$

To compute these two parameters a package which implement the Allan deviation analysis has been used [30]. To compute the Allan deviation it is sufficient to compute the square root of the Allan variance which can be computed as follow (in its simplest non-overlapping form)

$$\sigma^2(T) = \frac{1}{2(K-1)} \sum_{k=1}^{K-1} \left(\bar{\Omega}_{k+1}(T) - \bar{\Omega}_k(T)\right)^2 \tag{2.7}$$

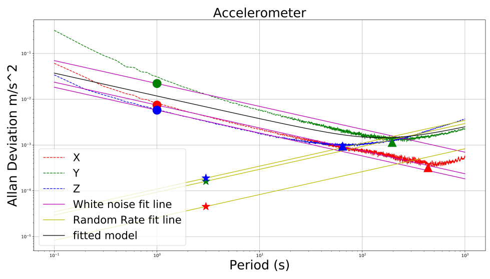
where T is the dimension of the clusters in which we want tot divide our data, K is the total number of cluster that we have, $\bar{\Omega}_k(T)$ is the average values of our data in the $k'th$ cluster of dimension T. We can compute this Allan variance for a lot of value of T and then plot it w.r.t this values. We can derive information about different noises that affects our system looking at this plot since it represent the variation of the mean value of our data for different dimension of the time window in which we are tacking the average. The Allan variance analysis was in fact used at the beginning as a tool to study the stability of oscillators but a complete overview of the history and theory behind it can be found in [31]. As a real general description different zones of the graphs gives information about different noises that affect our system.

2.2.3 The MPU 6050

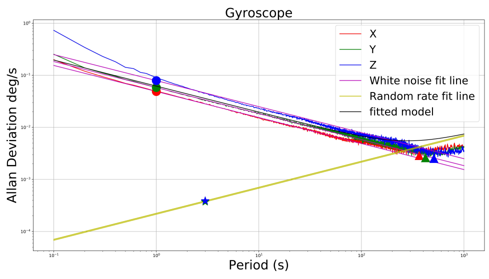
The motion processing unit that has been used in this project is the MPU 6050 mounted on the GY-521 module. The module is compose by the MPU, an LD3985 3.3V regulator (which allow us to power the module with voltage slightly different from 3.3V), a LED to show that the module is on and multiple resistor. This module allow really low power consumption and the MPU contains an accelerometer a gyroscope and a digital motion processor (DMP). The Digital Motion processor allow the MPU to elaborate data before putting them in the registers to be read by the microcontroller. This could be a useful feature to offload some of the computation from the central CPU. Moreover the MPU provide a digital low pass filter, with a low pass frequency that can be set. Another important feature of this MPU is that it is highly configurable and the configuration take place setting the desired values in the configuration registers. Some of the quantities that can be configured are

- the accelerometer range
- the gyroscope range
- the sample rate divider
- the digital low pass filter frequency

a lot more information about the module and about the register map can be found in [32] [33]. In figure 2.8 a collection of 1000 samples from the accelerometer place in a flat surface can be seen. In this image we can see the offset of the measurement in the three axis (which can also be caused by a not perfect orientation of the flat surface on which the accelerometr is placed), and the different dispersion on the values in the three different axis of measurement. The Allan variance plot of both the gyroscope and of the accelerometer can be seen in figure 2.7 and in table 2.2 the estimated noises parameters of the MPU 6050 can be observed. The MPU is able to communicate with the microcontroller (or the Raspberry in our case) through I2C protocol which is a synchronize (the clock is shared) serial (data are transmitted one after the other) protocol in which one Master is able to select



(a) Acceleration Allan curve



(b) Gyro Allan curve

Figure 2.7: Allan analysis curves used to extract noises parameters of both the accelerometer and the gyroscope

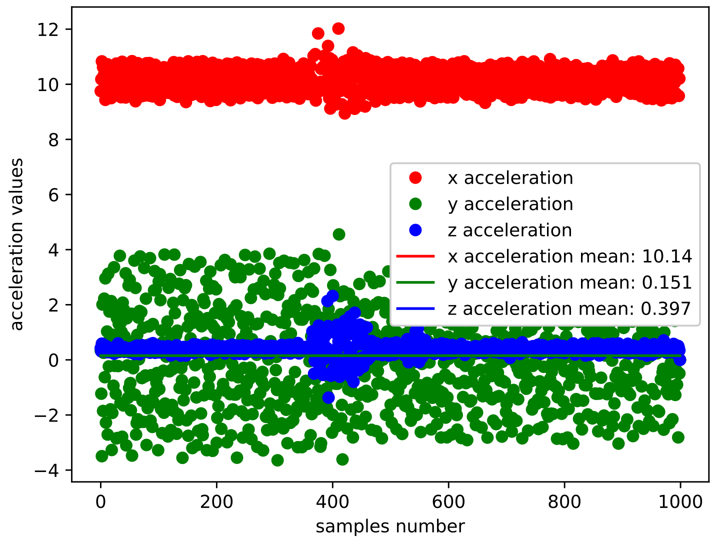


Figure 2.8: One thousand samples collected from the accelerometer in a static position. What can be noticed are biases and different dispersion in each axis

the slave from which he want to read data, and it is the one who decide when to start and end the communication. The I2C protocol is really simple and since it does not implement particular measure against noises it is usually used for board-to-board communication or at least for communication between two really near module.

2.2.4 IMU-Camera calibration

Once both the MPU and camera intrinsic parameter has been calculated following the procedure explained above, another important matrix is required by ORB-SLAM3. In particular this matrix is the transformation matrix between

accelerometer noise	0.011772986113862747 $m/s^{3/2}$
accelerometer random walk	0.0001321283869975406 $m/s^{5/2}$
gyro noise	0.0010812959383010065 $rad/\sqrt{s}$
gyro random walk	2.306075388456348 $* 10^{-6}$ $rad/s^{3/2}$
mpu frequency	150 hz

Table 2.2: Motion processing unit 6050 calibration parameter

the camera reference frame and the IMU reference frame. This matrix can be computed using the Kalibr package and the one obtained for this project is the following one (with entries rounded at the fifth decimal number)

$$\begin{bmatrix} 0.00655 & -0.99978 & 0.01943 & 0.00448 \\ -0.99988 & -0.00682 & -0.01381 & -0.00412 \\ 0.01394 & -0.01934 & -0.9997 & -0.02083 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

once we have obtained all these data we can write the configuration file needed by ORB-SLAM3.

2.3 Feature extraction and Matching

This section will explore one of the more important basic algorithmic element that is needed to implement a feature based visual SLAM algorithm. In fact visual SLAM algorithm are divide in two big group

- feature based methods: these are the methods that will be explored more in this thesis work since ORB-SLAM3 belongs to this group. Feature based methods extract feature and compute descriptors for a particular frame then they try to find the same points in the next images and based on the change of positions of these points they try to compute the camera trajectory

- direct methods: these methods directly work on the entire image without the need of feature extraction. These methods try to find the transformation matrix that minimize the photometric error, which differs from feature based methods which try to minimize the reprojection error.

so one of the foundational element of any feature based SLAM algorithm is the feature that will be used in the program. A feature is only a particular point in the image that can be represented in a really simple way. A feature point must have the following characteristics

- Repeatability: which means that the same point can be simply re-observed in multiple different images taken with different prospective
- Distinctive: the point must be distinguishable from all other points in the image and in particular from all other feature in the same image
- Efficiency: The features must be only a small subset of the pixels of the images and they must be identify by an efficient algorithm (this is true in particular for SLAM application)
- Locality: to find a features we need to use only a small subset of the pixels of the image

There exist multiple type of feature that we can extract from an image and in particular when we talk about a feature we need to talk about the corresponding key points (which is the position of the point in the image), the way in which we found them, and the corresponding descriptors (which is usually a vector describing the keypoint). Here SIFT features will be introduced before the ORB features because ORB is born, more that a decade after SIFT, to be a more efficient alternative to SIFT.

SIFT. To talk about SIFT we need first to talk about how the features point are detected. To do that we need to compute the scale space of the image which is define as as follow

$$S(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{2.8}$$

where

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\left(x^2+y^2\right)/2\sigma^2} \tag{2.9}$$

then we can compute $\sigma^2 \nabla^2 S(x, y, \sigma)$ and if we found a maximum in this function for some point (x, y, σ) we can claim that we have found a candidate feature point. In general we will discard some of the points in order to maintain only the most promising one. It is important to notice that as well as find a possible feature point we have also found its characteristic scale σ which allow SIFT to be scale independent. As a second step we need also to find the principal orientation of the feature point. To do that we simply compute the gradient of each pixels near enough to our candidate point (the notion of near here depends on the

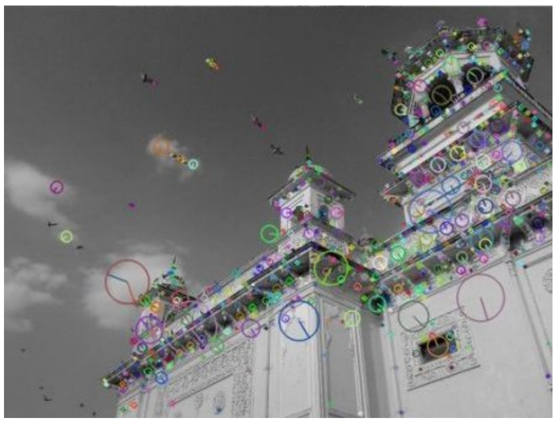


Figure 2.9: Image with SIFT features. In this representation the scale information is encoded in the dimension of the cirlces and the principal direction in descripted by the segment inside the circles. Image from the official OpenCV SIFT page [35].

characteristic scale that we have already computed) and we build an histogram of the orientation to find the most frequent one. This one will be selected as the principal orientation of the feature point. With this second step also orientation invariance has been obtained. The last thing that we need to do is to build a descriptor or a "signature" of a particular feature point. The SIFT descriptor is built based on normalized histograms of the gradients in the section of the image corresponding to the feature point which must be first rescaled w.r.t. the scale factor and rotated according to the principal direction. The descriptor is a vector, and if two descriptor are similar enough we can claim that we have found a match. In image 2.9 we can see examples of SIFT features and they can be applied in a lot of field in addition to SLAM, for example collage of image to build panoramic photos. SIFT is a really effective feature extractor and descriptor but the problem is that it impose a really high computational cost. In the years after SIFT introduction a lot of energy has been spent to produce a more efficient alternative to SIFT. The first results has been obtain with SURF but here ORB will be discussed since it has been introduced later, it has better performance and it is the one used by ORB-SLAM3. All the details regarding SIFT can be found in the original paper [34].

ORB. As before we need to describe how the ORB feature is extracted and then how the descriptor is computed. ORB is based on FAST for key points detection. FAST is a kind of corner point extractor which is really efficient. It extract point as follow

- select a pixel and check its brightness
- take the nearest 16 pixels around the ones that has been previously selected
- if there are N consecutive points which has brightness greater than the brightness of the selected pixel plus a threshold or smaller than the brightness of the selected pixel minus the threshold then we select that point as a feature point.

Since FAST collect no information about the scale and the rotation in ORB these detector is modified to obtain scale and rotation invariance. To obtain the first a pyramid of subsampled image is build and feature are detected in all the subsampled image. Moreover matches can be done also between points belonging to different level of the piramyd of different image. To obtain rotation invariance the intensity centroid method is used. The centroid is computed as

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right) \tag{2.10}$$

where

$$m_{pq} = \sum_{x,y \in B} x^p y^q I(x,y), \quad p,q = \{0,1\} \tag{2.11}$$

then we can connect the geometric center of the feature with the centroid to obtain the rotation direction of the feature. As before we have obtain scale and rotation invariance with a modification of the FAST corner detection, now we need to compute a descriptor for the feature point. ORB use BRIEF descriptor. The BRIEF descriptor is a 128-bit array in which the value of each element of the array (so a 1 or a 0) is decided selecting two random points in the feature and checking if the first one has a bigger or lower intensity than the second one. Actually BRIEF is modified in ORB a little bit to obtain better rotation invariance. Now it should be clear why ORB is called oriented FAST and Rotated BRIEF. Also in this case the complete description of ORB can be found in the original paper [36].

Once features has been extracted the feature matching is another big problem to solve. In fact even if we can take the most naive approach simply claiming a match every time that we found the couple of features in two images that minimize some sort of distance (euclidean for SIFT descriptors and Hamming distance for ORB) if we use this approach we can encounter multiple problems. To mention one of them take for example an image with the same texture repeated on all the figure. In this case a lot of feature point with a really similar descriptor will be present. One of the possible solution to this problem is presented in [37].

2.4 Pose estimation techniques

The core element of any SLAM algorithm is the estimation of the camera roto-traslation and the world position of the map points. In this section some types of methods to estimate in particular camera position will be explained. These methods differs because they use different information to recover the camera position. The first two method that will be explain will used the pixel coordinate of multiple corresponding point in two subsequent image to recover the camera rototraslation up to a scale factor. The third method try to recover the camera position based on the 3D world position of points and 2D pixel coordinate of the same points in the camera image plane.

2.4.1 Essential and Fundamental Matrix

The first method rely on the epipolar constrain which follow from a really simple consideration. Consider two image in which the projection of the same 3D point is present. We can immediately say that origin of the camera in the two positions and the position of 3D point must be on the same plane, or in other words must be coplanar. Now, given a point $\boldsymbol{P} = [X, Y, Z]$ we define

- $\boldsymbol{p}_i$ as the pixel coordinates of point $\boldsymbol{P}$ in the $i'th$ image plane
- $\boldsymbol{x}_i$ as the coordinates of point $\boldsymbol{P}$ in the $i'th$ normalized image plane

moreover we consider $\boldsymbol{K}$ the camera intrinsic matrix, $\boldsymbol{R}$ the rotation matrix between the two frames and $\boldsymbol{t}$ the transaltion vector between the two frames. From the pinhole camera model we know that

$$s_1\boldsymbol{p}_1 = \boldsymbol{K}\boldsymbol{P}, \quad s_2\boldsymbol{p}_2 = \boldsymbol{K}(\boldsymbol{R}\boldsymbol{P} + \boldsymbol{t})$$

where s_1 and s_2 are the depth value of the 3D point in the two camera reference frame that we are considering. We can remove these two scalar value from the equation substituting the equality with an almost equal symbol which in this scale means "up to a scale". Then we know that it is sufficient to left multiply $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ by $\boldsymbol{K}^{-1}$ to obtain $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. After these consideration we can conclude that

$$\boldsymbol{x}_2 \simeq \boldsymbol{R}\boldsymbol{x}_1 + \boldsymbol{t} \tag{2.12}$$

then we can left multiply both sides by $\boldsymbol{t}^\wedge$ and $\boldsymbol{x}_2^T$ to obtain the epipolar constrain which has the following expression

$$\boldsymbol{p}_2^T \boldsymbol{K}^{-T} \boldsymbol{t}^\wedge \boldsymbol{R} \boldsymbol{K}^{-1} \boldsymbol{p}_1 = 0$$

the matrix $\boldsymbol{E} = \boldsymbol{t}^\wedge \boldsymbol{R}$ is known as essential matrix and the matrix $\boldsymbol{F} = \boldsymbol{K}^{-T} \boldsymbol{E} \boldsymbol{K}^{-1}$ is called fundamental matrix. Now we can take 8 couple of points between the two frames, consider the epipolar constraint for each couples and set up a linear system of equation where the unknowns are the coefficients of the essential matrix. With that system we can find these coefficient up to a scale factor in fact we know that one degree of freedom is inevitable due to the fact that we are using monocular cameras. This problem is not present when using stereo cameras since in that case we have the additional information of the rototraslation between the two camera which is exactly what we are trying to reconstruct here. Once we know the essential matrix (or the fundamental matrix since the only difference between them is the intrinsic camera matrix which is assumed to be known in SLAM) we can recover the rototraslation between the two frame, again up to a scale.

2.4.2 Homography

A very similar method to the one explained above is the one that consider the homography matrix usually symbolized by $\boldsymbol{H}$. Also in this case a relation between the projection of the same 3D point in two different camera plane is found. This relation bring us to the definition of homography matrix. Using four different 3D point we can compute this particular matrix up to a scale factor, solving the system of equation that we build considering these four points. Then we simply need to recover the rototraslation based on the homography matrix that we have just computed. This method differs from the previous one because it assume that all the 3D points that we consider to compute $\boldsymbol{H}$ are on the same plane. Usually this plane is assumed to be parralel to the image plane of the camera.

2.4.3 PnP

The Perspective-n-Point (PnP) problem is a problem in which we try to reconstruct the camera pose given the 3D world coordinate of n points and their projection in the camera plane. Usually we will need to solve these type of problems when the visual SLAM algorithm loose track of its position and need to retrieve it based on the points that it is seeing and the one already part of the map that it is building. There are multiple way to solve these problem but here the Efficient PnP will be explained since it is one of the most simple and efficient techniques even if ORB-SLAM3 use a different method called ML-PnP. E-PnP is presented in [38] but the main priciples behind it are the following. We can consider each of the n reference points as a linear combination of four control points such that the sum of the coefficient of the combination is equal to one ($\sum_{j=1}^{4} \alpha_{ij}$). In this way we know that

$$\begin{aligned} \boldsymbol{p}_i^w &= \textstyle\sum_{j=1}^{4} \alpha_{ij} \boldsymbol{c}_j^w \\ \boldsymbol{p}_i^c &= \textstyle\sum_{j=1}^{4} \alpha_{ij} \boldsymbol{c}_j^c \end{aligned}$$

where the apex specify if the point is represented in the camera reference frame or in the world reference frame. Then we can consider the usual pinhole camera model

$$s_i \boldsymbol{p}_i^i = \boldsymbol{K} \sum_{j=1}^{4} \alpha_{ij} \boldsymbol{c}_j^c \tag{2.13}$$

where $\boldsymbol{p}_i^i$ are the pixel coordinate in the camera image plane of the point $\boldsymbol{p}_i$. From this equation for each point $\boldsymbol{p}_j^c = [x_j^c, y_j^c, z_j^c]$ we can obtain other two equation which has the following form

$$\begin{aligned} \textstyle\sum_{j=1}^{4} \alpha_{ij} f_x x_j^c + \alpha_{ij} \left(u_0 - u_i\right) z_j^c = 0 \\ \textstyle\sum_{j=1}^{4} \alpha_{ij} f_y y_j^c + \alpha_{ij} \left(v_0 - v_i\right) z_j^c = 0 \end{aligned} \tag{2.14}$$

if we consider n points we can build a system of equations in which the unknowns are the position of the control points in the camera reference frame. The solution to the system is found considering it as a linear combination of the right singular vector of the coefficient matrix of the system, as better explained in the paper. Once the system is solved the solution can be refined using Gauss-Newton method and then the poisition of the camera can be computed as the one that minimize the reprojection error of the control points.

2.4.4 RANSAC

The random sample consensus method (RANSAC) [39] is an iterative method which can be applied to all model estimation problem to find the best model even in the presence of outliers, so it can be applied to all the pose estimation

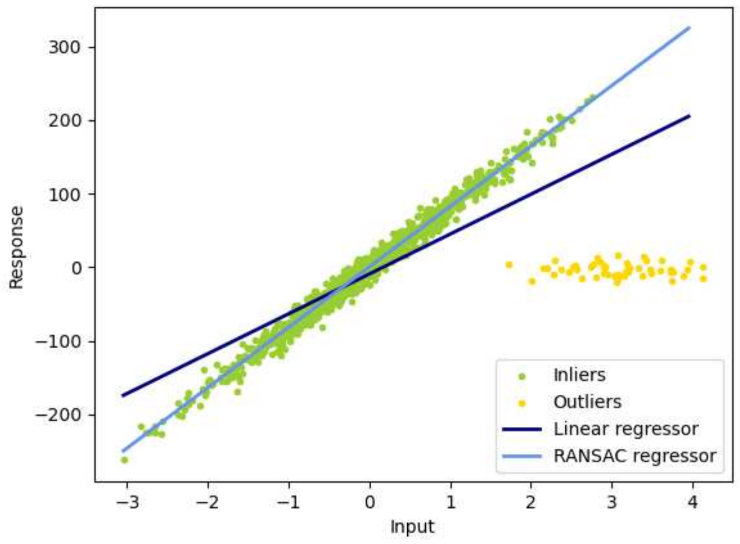


Figure 2.10: The result of a linear regression using RANSAC or using the classical method. Image taken from the official scikit learn RANSAC page [39].

techniques explained above. The algorithm work as follow

- randomly select the less number of data that you need to fit the model. (for example for a linear regression model the minimum number of data points is two)
- fit the model and test all the data on the fitted model (also the ones that have not been selected in the first step)
- repeat these two step for a certain number of iteration
- select the model that has the best performance

In image 2.10 can be seen the result of a linear regression applied to all point in the dataset and the result obtained using RANSAC. It is important to notice that even if RANSAC is able to obtain great result it is not a deterministic algorithm so its capability to estimate good models depend on the number of iteration that we will fix. In general RANSAC is heavily utilized in all the model estimation problem which are present in SLAM.

2.5 Bundle Adjustment

The bundle adjustment problem is really fundamental since its solution will be heavily utilized by ORB-SLAM3. In general solving a bundle adjustment problem means to find the positions of the landmarks, the position of the camera (exstrinsic parameters) and the intrinsic parameter of the camera, that minimize the reprojection error, given multiple view of a particular scene. Bundle adjustment is heavily utilized in SLAM but also in other field such as 3D reconstruction. As any minimization problem that we need to solve with a numerical method, initializing it in the correct way is really fundamental to not remain stuck in a local minimum.

2.5.1 Problem formulation and solution

Since the simplest camera model (pinhole) has already been introduced the formulation of the problem is quite simple. In fact we can suppose to have multiple images of a particular scene and that we have extract some features from each image. Obviously a lot of features match between these images is necessary to have a possibility to solve the problem. This means that the same 3D point must be present in the field of view of the camera when it is set in many different position. Now we know that we can in the order

- transform the 3D coordinate of a point in the camera frame coordinate
- apply the distortion model to undistort the image
- compute the pixel coordinate using the intrinsic parameter

so if we consider the function $h(\boldsymbol{x}, \boldsymbol{y})$ that map 3D point into the camera image plane, where $\boldsymbol{x}$ is the pose of the camera and $\boldsymbol{y}$ is the position of the 3D point, we can see that it is a strongly non linear function. Now we can simply consider the following cost function

$$\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\|\mathbf{e}_{ij}\|^2 = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\|\mathbf{z}_{ij} - h\left(\mathbf{T}_i, \mathbf{p}_j\right)\|^2 \tag{2.15}$$

where

- $\boldsymbol{z}_{ij}$ is the actual pixel coordinate position of the $j'th$ point when the camera is in the $i'th$ pose

- $\boldsymbol{T}_i$ is the $i'th$ pose of the camera represented by the trasform matrix between the global reference frame and the camera frame
- $\boldsymbol{p}_j$ is the position of the point j in global coordinates

this non linear minimization problem can be solved for example using Levenberg-Marquardt method (explained in the math section of this thesis) which reduce to find the solution of the normal equation. The problem with this equation is that its dimension can be really large. In fact usually the features points in each of the multiple images that we are considering could be hundreds. So if we try to solve it directly inverting the hessian matrix it could lead to an explosion in the computational power that we need. This is a really big problem since our aim is to implement these type of algorithm on a drone which has low power hardware, and the program need to run this software in real time. There is a characteristic of the hessian matrix for this type of problem which is really important to achieve real time performance when we are solving this system of equation which is sparsity.

2.5.2 Sparsity

The sparsity of the hessian matrix directly derive from the sparsity of the jacobian matrix. In fact we need to consider the fact that the term $\boldsymbol{e}_{ij}$ of the loss function is only influenced by $\boldsymbol{T}_i$ and $\boldsymbol{p}_j$. This is due to the fact that the 3D position of another point in the image or the pose of the camera which is different from the i position does not influence the reprojection error of the point j in the camera plane when the camera is in the pose i. This means that all the other derivatives are zeros. It can be shown that the hessian matrix $\boldsymbol{H}$ in the bundle adjustment problem, if we order the vector of the unknowns in such a way that it contains all the unknown poses in the first part and all the 3D positions of the points in the second, has the following structure

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{B} & \boldsymbol{E} \\ \boldsymbol{E}^{\boldsymbol{T}} & \boldsymbol{C} \end{bmatrix} \tag{2.16}$$

where $\boldsymbol{B}$ is a block diagonal matrix with dimension that is proportional to the number of camera poses (so it is very smaller than $\boldsymbol{e}_{ij}$ since the number of points in every image is really higher than the number of image itself), $\boldsymbol{C}$ is also a block diagonal matrix with dimension proportional to the number of features in all the images. Given this structure of the hessian matrix we can left multiply both sides of the linear equation, of the Levenberg-Marquardt method, by the following matrix

$$\begin{bmatrix} \boldsymbol{I} & -\boldsymbol{EC^{-1}} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix} \tag{2.17}$$

we obtain

$$\begin{bmatrix} \mathbf{B} - \mathbf{E}\mathbf{C}^{-1}\mathbf{E}^{T} & \mathbf{0} \\ \mathbf{E}^{T} & \mathbf{C} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_c \\ \Delta\mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{v} - \mathbf{E}\mathbf{C}^{-1}\mathbf{w} \\ \mathbf{w} \end{bmatrix} \tag{2.18}$$

where $\Delta\mathbf{x}_c$ and $\Delta\mathbf{x}_p$ are the increment that we want to obtain at each step of the numerical method and $\mathbf{w}$ $\mathbf{v}$ are the partition of $\boldsymbol{g}$ as it is defined in the math section according to the partition of $\boldsymbol{H}$ that we have shown in this section. The first line is solvable independently and has a really smaller dimension than the original one. Moreover once we have the solution of the first equation the solution to the second line is simply

$$\Delta\mathbf{x}_p = \boldsymbol{C}^{-1}(\boldsymbol{w} - \boldsymbol{E}^{T}\Delta\mathbf{x}_c) \tag{2.19}$$

which is simpler to solve than the original one because $\boldsymbol{C}$ is a block diagonal matrix so is really simpler to compute its inverse.

2.6 Bag of Words

The last thing that need an introduction in this chapter is the Bag of Words approach used by visual SLAM algorithm to build a database of already visited position in such a way that the loop closing problem could be solved, recognizing an already visited scene and then correcting the entire trajectory in a coherent way. The bag of word representation was first used in natural language processing and following this particular scheme you can simply represent a text as an histogram of the frequency of the words that it contains. In this section the way in which the BoW representation of an image is computed will be discussed and the structure of images database that can be built for loop closing detection will be explained. In particular the database and the basics needed for understand the loop closing technique used by ORB-SLAM will be shown [40].

2.6.1 BoW image representation and database structure

The first thing to explain is how the BoW representation of an image is computed. First of all we need to build a tree with the following procedure

- select a database of random images
- compute the features in each of the image belonging to the database
- cluster the features in the features space using k-medians clustering to build the first nodes of the tree
- build the child node of each node built in the previous step using again k-medians clustering
- repeat the previous step until you reach W leaf node

now we will consider the leaf node of the tree that we have just built as words. To compute the BoW representation of an image we need to take each feature in the image and compute the corresponding word following the tree from the top to the leaf node selecting at each step the node that minimize the Hamming distance with the features that we are considering. We repeat the procedure for each feature in the image and then we can compute the BoW representation of the image considering the frequency of the words which are present in the image itself. The database of the images is not only compose by the sequence of BoW representation of seen images but also by other two field

- *inverse index:* in which for each word in the vocabulary a list of the images in which the word is present and the weight of the word in that corresponding image is maintained.
- *direct index:* in which for each image and for each tree depth (the root node is at level N the leaf node are at level 0) the node touched by some feature in the image and the features that activate these nodes are stored

both of these set of data are updated every time a new image is inserted in the database. Even if could seem quite a complex structure each of the index will be useful in the next section to find the loop closure candidates.

2.6.2 Loop detection

Once a new image is acquired we need to check for possible correspondence with images already present in our database to find loop closure candidate. To do that the inverse index allow us to reduce the dimension of the set in which we need to search since we can restrict our research only on the set of images that share some words with the one that we have just acquired. Then we simply need to find images that obtain an high normalized similarity score with our new images, which is computed as follow

$$\eta(\boldsymbol{v}_t, \boldsymbol{v}_{t_j}) = \frac{s(\boldsymbol{v}_t, \boldsymbol{v}_{t_j})}{s(\boldsymbol{v}_t, \boldsymbol{v}_{t-\Delta t})}$$

where

$$s(\boldsymbol{v}_1, \boldsymbol{v}_2) = 1 - \frac{1}{2}|\frac{\boldsymbol{v}_1}{|\boldsymbol{v}_1|} - \frac{\boldsymbol{v}_2}{|\boldsymbol{v}_2|}|$$

and $\boldsymbol{v}_i$ respresent the BoW representation of the image i. In particular $\boldsymbol{v}_{t-\Delta t}$ is the BoW representation of the prevoius image in cronological order with respect to the new image for which we are searching a match. A particular clustering between images taken near in time is described in the same paper that describe this entire procedure [40]. This clustering is needed to avoid that images that are similar because they have been acquired in a similar moment in time, could compete to be the one selected for the loop closure. At the end to check if one of the selected candidates could be considered for a real loop closure we can check the correspondence of features, and of their distribution in the two images. To do that the direct index is helpful since it allow us to speed up the feature correspondence (in this case the parameter considering the depth of the tree in which we want to find the correspondence is useful as a trade-off between complexity and number of feature correspondence).

Chapter 3

ORB-SLAM 3

As explained in the introduction of this thesis, the aim of this work is to build a 3D map of the environment in such a way that the map is navigable. This means that the map need to be dense. In this chapter the basic SLAM algorithm that will be used to track the camera movement and to estimate the true depth of some sparse points will be presented. In Chapter 4 these first information produced by this algorithm will be completed by the ones given by a relative depth estimation neural network to produce the necessary dense map. ORB-SLAM3 is a visual/visual-inertial/RGB-D feature-based SLAM system developed by an equip of the University of Zaragoza. It has been developed through several years of research and the principal step of its development history can be reported here

- *ORB-SLAM: a Versatile and Accurate Monocular SLAM System (2015)* [4]: with this paper the first full implementation of ORB-SLAM as a complete SLAM system was presented. Obviously multiple papers was previously published to solve specific problem such as feature detection and extraction or relocalization, but with this particular work the first full implemented version of this SLAM library was presented and can already achieve good performances
- *ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras (2017)* [15]: this second version of the algorithm was totally based on the first one but with the crucial modifications which allow it to support both stereo and RGB-D cameras. The principal difference between monocular and stereo/RGB-D SLAM is that the real world scale depth information of the pixels is already available and we do not need any

triangulation between pixels in successive frames to recover it.

- *Visual-Inertial Monocular SLAM with Map Reuse (2017)* [41]: with this third paper the modifications needed to integrate information coming from an accelerometer were introduced. With those information the algorithm is able to achieve better performances even when poor visual information are available and it is also able to recover the true world scale of the environment even in the monocular case
- *ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM (2021)* [5]: with this last implementation of ORB-SLAM the algorithm is now able to manage and merge multiple map. Moreover various improvement in the code, a better IMU initialization procedure and an improved place recognition algorithm allow to achieve state of the art performances.

ORB-SLAM3 is now one of the most precise visual SLAM algorithm and in the first section of this chapter all the module needed for its functioning will be presented. Then in the second section the troubleshooting for its installation on Ubuntu 20.04 and ROS Noetic will be presented. This second section could seem useless but since ORB-SLAM3 is built for Ubuntu 18.04 and ROS Melodic the time spent to debug all the installation problem justify the presence of this small section, which I wish could be helpful for other people which are trying to install ORB-SLAM3 with the same set-up. In the last section the modification introduced in the ROS Wrapper of ORB-SLAM3 to make depth and pose information available to the ROS network will be presented.

3.1 Algorithm description

In this first section the elements that compose ORB-SLAM3 will be described. Moreover the principal modules of this algorithm will be analyzed. This third version of the algorithm is based on the first monocular implementation so particular emphasis will be given to the addition needed to include data from the accelerometer to build a real scale map of the environment

3.1.1 Important element desciption

In this first part a list of the principal elements that compose the algorithm will be given

- **Map Points.** Map points are extracted using ORB features extractor and descriptor. Their depth is estimated using triangulation and then refined with bundle adjustment. They contain information about their 3D world position, the viewing direction towards the optical center of each keyframe in which they are visible, the ORB descriptor, and the maximum and minimum distance at which it can be observed according to the limits imposed by the ORB features

- **Keyframe.** Keyframe are created when enough different keypoint with respect to the previous keyframe are found in a frame. It store all the ORB features and the camera pose in the world coordinate frame. Keyframe are generated very frequently but then a really rigid culling policy is used to eliminate keyframes that contain redundant information. In this way the map do not grow really fast during the exploration.

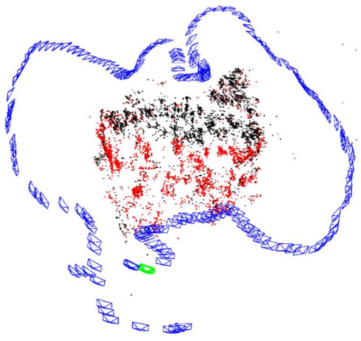
- **Active Map.** The active map is in practice the collection of keyframe that the system is using in the local mapping and tracking of the camera position. In case that the tracking is lost and the system cannot relocalize a new active map is generated and the system continuously try to merge it with the previous active map in the most coherent way

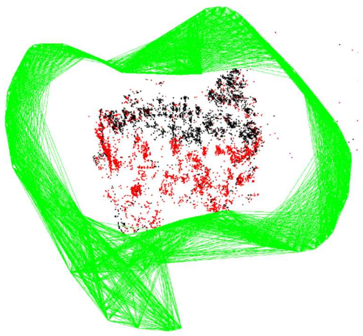
- **Atlas.** The Atlas is a set of disconnected map which contain the active map and all the other non-active map.

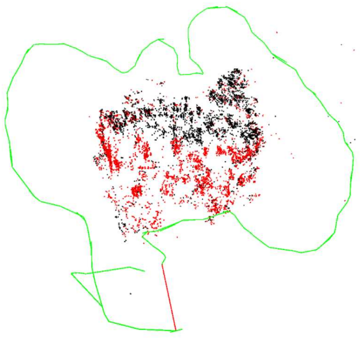
- **Keyframe Database.** The Keyframe database is built as described in the previous chapter using Bag of Words representation of keyframe. It also store direct and inverse index as explained before. This database is fundamental for relocalization purposes.

These are the most important objects used in the code to store information about the map and the camera position. There are also three really important graph that are built when the algorithm is running which represent the relationship between different keyframe. These graph are really important since, for example, they allow the algorithm to select which keyframe it need to use in the local map bundle adjustment problem. These graph are

- **Covisibility graph.** The covisibility graph has one node for every keyframe produced by the algorihtm and the connection between nodes is stronger or weaker depending on the number of map points visible from both keyframes.


(a) Representation of keyframe and map points


(b) The covisibility graph representation


(c) The spanning tree representation

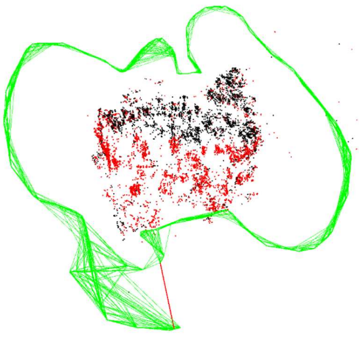

(d) The essential graph representation

Figure 3.1: Principal graph and object used by ORB-SLAM3. Images from the original ORB-SLAM3 paper.

- **Spanning tree.** The spanning tree contain each keayframe that has not been eliminated by the culling poilicy and each time a new keyframe is inserted it is connected only to the keyframe with which it has more covisible map points

- **Essential graph.** The Essential graph contains the spanning tree, all the connection which are strong enough (over a certain treshold) and the loop closure edges. This graph is really helpful when for example we need to solve the usual pose adjustment problem after a relocalization. In fact it gives us the information about which keyframes we can use to set the minimization problem that we need to solve to adjust the pose of our camera.

The last thing to mention is that, to maintain real time performance, the software is compose of three different threads: the first thread is the tracking thread and it is in charge of estimate the camera displacement frame by frame, the second thread is the local mapping thread and it is in charge of decide when to insert a new keyframe and to solve the local Bundle Adjustment problem to optimize the local map, the third thread is the Loop and Map merging thread which try to continuously find loop closure and try to merge different maps, when a loop closure is identified with a keyframe coming from another map a global bundle adjastement is launched to merge the two map. All these threads will be described in this section of the thesis and in addition the way in which the IMU is initialized will be briefly explained.

3.1.2 IMU initialization

The first important part of the algorithm is the IMU and the map initialization. In the calibration file that you need to give to the algorithm the biases of the accelerometer are not requested. Moreover the gravity vector direction is not known and the scale factor needed to rescale the map that we can obtain from the monocular camera is not known either. So the firsts steps taken by the algorithm are the following one.

- it try to estimate a map using only the camera information. In this way it can obtain a good map and trajectory estimate but in a totally wrong scale. In the same time data from the accelerometer are stored in memory
- as a second step, knowing the out of scale trajectory estimated in the first step, and the accelerometer data, it tries to find the best IMU biases, gravity vector and scale factor that maximize the probability of observing them given the inertial measurement that has been collected. Once the problem is solved the map can be rotated according to the gravity vector and scaled according to the scale factor.
- as a last initialization step a joint visual and inertial map estimation is brought on given the initial estimate given by the first two step

it is important to notice that, even if the gravity vector and the bias of the accelerometer are assumed to be well estimated after this first initialization, the scale factor is refined considering more keyframe and more accelerometer data until the map reach one hundred keyframe or 75 seconds have passed until the map initialization. More details about the IMU initialization can be found in the ORB-SLAM3 original paper, this type of initialization is in fact a novelty of this last version of the algorithm and differs from the initialization proposed with ORB-SLAM VI.

3.1.3 Tracking

The tracking part of the algorithm is instead totally inspired by the one used in ORB-SLAM VI. The tracking procedure is different if a map update has just happen or not. If a map update has just happened the pose of the camera is estimated finding the camera rototraslation that minimize the reprojection error (this is connected to the visual part of the system and is explained in chapter 2), and the sum of inertial residuals as they are defined in [42]. So in case that a map update has just happened the optimization problem is the following one

$$\begin{aligned} \theta &= \left\{\mathbf{R}^j_{\mathrm{WB}}, {}_W\mathbf{p}^j_{\mathrm{B}}, {}_W\mathbf{v}^j_{\mathrm{B}}, \mathbf{b}^j_g, \mathbf{b}^j_a\right\} \\ \theta^* &= \underset{\theta}{\operatorname{argmin}} \left(\sum_k \mathbf{E}_{\mathrm{proj}}(k,j) + \mathbf{E}_{\mathrm{IMU}}(i,j)\right) \end{aligned} \tag{3.1}$$

where $\mathbf{E}_{\mathrm{proj}}(k,j)$ is the usual reprojection error, $\mathbf{b}^j_g$ and $\mathbf{b}^j_a$ are respectively the gyroscope and the accelerometer biases. The error related to the IMU measurement is the following one

$$\begin{aligned} &\mathbf{E}_{\mathrm{IMU}}(i,j) = \rho\left(\left[\mathbf{e}_R^T \mathbf{e}_v^T \mathbf{e}_p^T\right] \mathbf{\Sigma}_I \left[\mathbf{e}_R^T \mathbf{e}_v^T \mathbf{e}_p^T\right]^T\right) + \rho\left(\mathbf{e}_b^T \mathbf{\Sigma}_R \mathbf{e}_b\right) \\ &\mathbf{e}_R = \log\left(\left(\Delta\mathbf{R}_{ij} \operatorname{Exp}\left(\mathbf{J}^g_{\Delta R}\mathbf{b}^j_g\right)\right)^T \mathbf{R}^i_{\mathrm{BW}}\mathbf{R}^j_{\mathrm{WB}}\right) \\ &\mathbf{e}_v = \mathbf{R}^i_{\mathrm{BW}}\left({}_W\mathbf{v}^j_{\mathrm{B}} - {}_W\mathbf{v}^i_{\mathrm{B}} - \mathbf{g}_{\mathrm{w}}\Delta t_{ij}\right) \\ &- \left(\Delta\mathbf{v}_{ij} + \mathbf{J}^g_{\Delta v}\mathbf{b}^j_g + \mathbf{J}^a_{\Delta v}\mathbf{b}^j_a\right) \\ &\mathbf{e}_p = \mathbf{R}^i_{\mathrm{BW}}\left({}_W\mathbf{p}^j_{\mathrm{B}} - {}_W\mathbf{p}^i_{\mathrm{B}} - {}_W\mathbf{v}^i_{\mathrm{B}}\Delta t_{ij} - \frac{1}{2}\mathbf{g}_{\mathrm{W}}\Delta t^2_{ij}\right) \\ &- \left(\Delta\mathbf{p}_{ij} + \mathbf{J}^g_{\Delta p}\mathbf{b}^j_g + \mathbf{J}^a_{\Delta p}\mathbf{b}^j_a\right) \\ &\mathbf{e}_b = \mathbf{b}^j - \mathbf{b}^i \end{aligned} \tag{3.2}$$

where we are considering the last keyframe i and the current frame j. Moreover ${}_W\mathbf{p}^j_{\mathrm{B}}$ or ${}_W\mathbf{v}^j_{\mathrm{B}}$ indicates the position or the velocity of the IMU in the world frame, $\Delta\mathbf{R}_{ij}$, $\Delta\mathbf{v}_{ij}$, $\Delta\mathbf{p}_{ij}$ are the variation of orientation velocity and position between the keyframe and the current frame which can be computed through IMU data pre-integration, Δt_{ij} is the time interval between the last keyframe and the current frame, $\mathbf{R}^i_{\mathrm{BW}}$ and $\mathbf{R}^j_{\mathrm{WB}}$ are respectively the rotation matrix between the body reference frame and the world reference frame considering the last keyframe and the rotation matrix between the world reference frame and the body reference frame considering the current frame and ρ is the Huber cost function. $\mathbf{\Sigma}_I$ and $\mathbf{\Sigma}_R$ are the information matrices of the preintegration and of the bias random walk.

The Jacobian terms $\mathbf{J}_{(\cdot)}^{(\cdot)}$ are used to take into account the effect of changing the biases. They can be efficiently computed online using IMU data as they are arriving, as explained in [42]. $\mathbf{g}_{\mathrm{W}}$ is the gravity vector. If the tracking is not considered after a map update the same optimization problem is solved but in addition the prior computed in the previous step are considered and an additional error term is added to minimize the difference between the new estimated velocity position and translation, and the one estimated for the previous frame. It is important to notice that at the beginning the algorithm proceed with a pure monocular SLAM as described in [4] so alternatively estimating the camera movement using the Essential or the Homography matrix methods and then choosing the most suitable one between them. Only after this first trajectory estimation and the IMU initialization the tracking procedure summarized in this section and better described in the paper is used. The tracking thread is also responsible for the insertion of new keyframe. They are inserted if

- the last global localization has happened at least 20 frames ago
- at least 20 frame has passed since the last keyframe insertion
- the current frame has at least 50 keypoints
- the current keyframe has at most 90% of keypoints tracked by the previous keyframe

it can be seen that keyframe are generated in a very generous way but then, as it has been said before, a strict policy (principally connected to the additional information contained in a particular keyframe) is used to cut useless keyframe.

3.1.4 Mapping

Also the local mapping part of the algorithm is the same that has been built for ORB-SLAM VI. The mapping thread simply consider the last N keyframe and perform local bundle adjustment to optimize the local map. The cost function that will be optimized in this part of the algorithm is slightly different from the one described in the second chapter of this thesis since it contains also a term related to the data coming from the accelerometer. Only the last N keyframe are selected since the computational complexity of the bundle adjustement problem rapidly increase when we consider too much keyframe, and real time performance are a really key component for a SLAM algorithm. The local mapping thread is also responsible of the local keyframe culling. This is really fundamental to achieve long time performance since in this way the quantity of keyframe (which are used in particular for BA) can remain bounded over time. The discard policy eliminate all the keyframe in which at least 90% of all the points are contained in at least other three keyframe at an equal or finer scale. In this case since we are using also data coming from the accelerometer and these data are less reliable when two keyframe are far away in time a keyframe can be discarded only if its elimination do not create a time gap higher than $0.5s$ between two keyframe in the local map.

3.1.5 Loop Closing and Map Merging

The place recognition technique used by ORB-SLAM3 is based on the bag of words keyframe representation as explain in the previous chapter and consist in the following six principal step

- first of all the BoW database produce three candidates keyframes based on the closeness of the BoW representation of the keyframes
- for each candidate a local window with its best covisible keyframe is build and keypoints matches between these keyframe and the reference keyframe is found with the help of the database direct index
- the best possible rototraslation considering all the keypoints contained in each local window and the reference keyframe is computed
- more matches between the reference keyframe and all the keypoints in each local windows are found using the rototraslation estimated in the previous step
- the process continue only of place recognition is fired in three consecutive keyframe
- the gravity vector is verified to see if the place recognition hypothesis can be finally accepted

the best place recognition hypothesis is then selected. In this case two possible situation could happened. In the first case the place recognition connect the current position to a place collocated in the current active map. In this case the classical loop closing algorithm is started and a global bundle adjustment (which

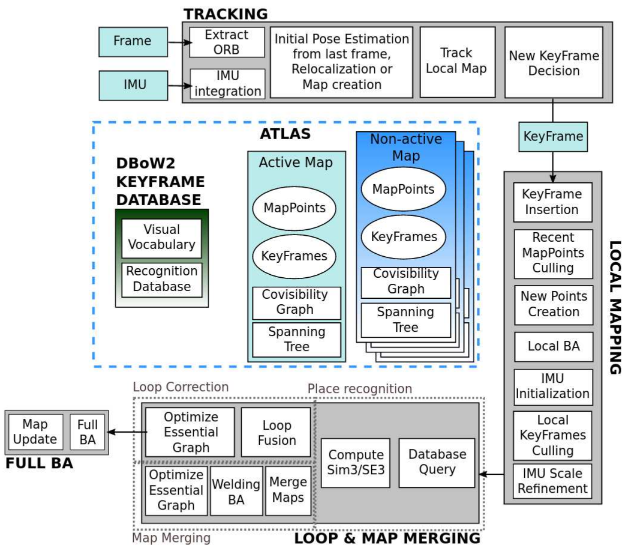


Figure 3.2: Schematic summary of ORB-SLAM3 functioning. Image from the original ORB-SLAM3 paper.

is performed only on a subset of keyframe if the number of them is too high to avoid the explosion on the computational cost) is launched to readjust the entire trajectory and map according to the new loop closing information. In the second case the place recognition find a correspondence with a position present in a non active map. In that case a particular merging map algorithm is triggered, as it is explained in the paper.

3.2 ORB-SLAM3 installation

This small section of the thesis is dedicated to the explanation of the installation of ORB-SLAM3. The principal problem, as mentioned before, is that ORB-SLAM3 has been built and tested on Ubuntu 18.04 and ROS Melodic but in this thesis setup the version of Ubuntu utilized is the 20.04 "Focal Fossa" and the version of ROS is Noetic. So, as explained in the GitHub page of ORB-SLAM3 the installation in a different set up could lead to some problems that need to be solved. Since these problem could be not so easy to solve, in particular if the installation is brought on by someone with low experience, I wish that this section could be helpful for others who are trying to make ORB-SLAM3 work in a similar set-up. In particular in the first part of the section the principal dependencies used by ORB-SLAM3 will be listed and the second part will be dedicated to a small description of the principal problem encountered during the installation.

3.2.1 Principal Dependencies

The principal dependencies needed by ORB-SLAM3 to work are strictly related to the algorithms described in the chapter 2 of this thesis. This will be really clear when the single dependencies will be explained in this subsection

Pangolin. Pangolin is the only dependency that is not important for the functioning of the algorithm but it is only needed for the visualization and the user interface of ORB-SLAM3. In fact it is used in particular in file such as *MapDrawer.cc* which is used for visualization purposes

OpenCV. OpenCV is one of the most important dependencies since it include already built function not only to extract ORB features but also to match them. Moreover OpenCV support all type of image manipulation function so it can be used to rectify the image but also to change its color encoding. It is important to notice that, even if they can't be used to obtain ORB-SLAM calibration files, OpenCV contains already built function to obtain camera calibration data.

Eigen3. Eigen3 is a really fundamental C++ library for creation and manipulation of linear algebra objects (vectors, matrices etc.). Eigen3 is particularly important for its efficiency and it is critical for the functioning of ORB-SLAM3 because it is needed by the next dependency that will be described here which is g2o.

g2o. g2o is an open source general graph optimization C++ library used to solve non linear optimization problem. In particular it is heavily utilized (as in this case) to solve Bundle Adjustment problem which are one of the most important problem to solve in SLAM algorithm.

DBoW2. DBoW2 is another really important C++ open source library which is able to implement BoW representation of images and so is heavily used to build the keyframe database and to implement loop closing algorithm. The library support by default the ORB descriptor so it is very suitable to be utilized in ORB-SLAM. The library need a pre-computed ORB vocabulary which is in fact required when ORB-SLAM is started from the command line.

Sophus. Sophus is also a C++ library based on Eigen which implement everything that is related with Lie Group and Lie algebra, so to represent 2D and 3D rigid body transformation.

It is important to notice that all the dependencies need to be installed in their right version to avoid particular problem during the build operation. We will see that this will be one of the problem encountered during the installation.

3.2.2 Installation Troubleshooting

In this section the most important problem encountered during the installation of ORB-SLAM3 and its dependencies will be listed

1. The first thing to notice is that, during the installation of Pangolin, the last build that consider also the python component of the library could lead to errors that do not allow to complete the installation with "the python stuff". The thing is that the installation of pangolin could be considered ended once the first basic build using the command `cmake --build build` end succesfully

2. To avoid any type of library conflict due to other version of OpenCV installed in the computer it is useful to install it by source following for example the guide at [43] (if you select this guide it is important to disable the OpenCV support to CUDA since we will not need it). After this second step no other problem with dependencies installation should happened.

3. at this point if the use of C14 is forced adding the following line

   ```
   set(CMAKE_CXX_STANDARD 14)
   set(CMAKE_CXX_STANDARD_REQUIRED ON)
   set(CMAKE_CXX_EXTENSIONS OFF)
   ```

   to the CMake file in the root directory of ORB-SLAM3 the building process without ROS support using the `./build.sh` file should go on without problems. If these lines are not added errors regarding standard libraries will pop up.

4. The next thing to solve is a directory conflict due to the name of the root folder of the project and the name of the ORB-SLAM3 folder inside the ROS

directory of the project. To do that the only way that I have found is to redo all the installation procedure changing the name of the root folder (in my case I have named it ORB-SLAM3_folder). At this point the directory present in the `bash.rc` and in the `build_ros.sh` files need to be changed consequently. In particular the name *Examples* in the `build_ros.sh` file need to be changed in *Example_old* since the ROS directory is present only in the older version of the folder

5. Now the following two lines need to be added to the CMake file that can be found in the ORB_SLAM3 folder inside the $/ROS/ORB\_SLAM3$ directory

   ```
   set( OpenCV_DIR "/home/lorenzo/Downloads/
   opencv-4.5.2/build" )
   ${PROJECT_SOURCE_DIR}/../../../Thirdparty/Sophus
   ```

   the first one can be added at the beginning of the file behind the line needed to find the OpenCV package, the second one need to be added inside the *include_directories*. These two line allow the compiler to find these two dependencies which cannot be found otherwise.

6. the last things to do is to force also in the CMake file mentioned in the previous point the use of C14 using the same line reported above and commenting the following line in the same file

   ```
   # Node for monocular camera (Augmented Reality Demo)
   #rosbuild_add_executable(MonoAR
   #src/AR/ros_mono_ar.cc
   #src/AR/ViewerAR.h
   #src/AR/ViewerAR.cc
   #)

   #target_link_libraries(MonoAR
   #${LIBS}
   #)
   ```

in this way we can disable the build of the augmented reality demo which is not needed and which can lead to the appearance of other errors

at this point also the installation of ORB-SLAM with ROS support should ended without further errors and the needed code modification can be implemented to extract the necessary information from the algorithm.

3.3 ROS Wrapper modification

The last part of this section is dedicated to the explanation of the modification implemented to the ROS Wrapper of ORB-SLAM3 to be able to extract data from the algorithm and publish them as ROS messages to be able to build another node for the densification of the map. The data that need to be publish are in particular

1. the camera pose
2. the visible map point corresponding to the camera pose. In particular we want to obtain a depth map that contains the depth of pixels in the keyframe that correspond to the 3D map point.

to try to reduce the computational cost, since the node that use the neural network for the densification is quite demanding in terms of resources, these data are published only when a keyframe is generated and not for every frame. In the following subsection the principal modification to the code will be shown and commented.

3.3.1 Comment on code

The first thing that need to be done is the creation of the messages to transport information about the position of points visible in a keyframe (which will be called depth map) and the pose of the camera corresponding to the same keyframe. To publish the camera position is sufficient to use a simple transform broadcaster and to publish the depth map a LaserScan message can be used. In this case the message need to be reshaped by the NN node to recover the right dimension of the depth map.

The second really important thing is to check if at least the first keyframe has been created before the ROS cycle started. This is due to the fact that if the first keyframe has not been generated yet the program will crush because inside the cycle you will refer to a place in the memory that has not been allocated yet. To do that we need simply to do the following check

```
if(((*SLAM.mpTracker).mCurrentFrame.mpLastKeyFrame) != nullptr)
```

in this first case we can also analyze how the pointer to the first keyframe is found. The `SLAM` object contains all the SLAM system and the `mpTracker` is the object containing everything that is needed for tracking purposes. In particular it contains the object corresponding to the current frame which contains the pointer to its reference keyframe. Moreover the keyframe object contains a method to obtain the inverse pose of the camera in the world reference frame so using the following line we can recover it

```
Sophus::SE3f actual_pose = lastKeyFrame->GetPoseInverse();
```

note that the pose of the camera is represented as expected, in ORB-SLAM, as a $SE3$ Sophus object. Then, using the following lines we can build the translation vector and the quaternion corresponding to the $SE3$ rototraslation

```
geometry_msgs::Pose pose;
Eigen::Vector3f translation = actual_pose.translation();
pose.position = eigenToPointMsg(translation);
Eigen::Quaternionf quaternion = actual_pose.unit_quaternion();
pose.orientation = eigenToQuaternionMsg(quaternion);
```

at the end we can simply create a Transform broadcaster which can publish a `tf::Transform` object which need to be created and set up with the current rotation and translation just extracted. The following line of code do exactly that

```
static tf::TransformBroadcaster br;
tf::Transform transform;
transform.setOrigin(tf::Vector3(translation(0),translation(1),
translation(2)));
tf::Quaternion q(pose.orientation.x, pose.orientation.y,
pose.orientation.z, pose.orientation.w);
transform.setRotation(q);
```

at this point we can publish the camera pose with the following line

```
br.sendTransform(tf::StampedTransform(transform,
ros::Time((*SLAM.mpTracker).mCurrentFrame.mpLastKeyFrame->
mTimeStamp), "world", "camera_frame"));
```

it is important to notice that the last keyframe object contains the timestamp corresponding to the the moment in which the keyframe has been acquired. This is really important since using this information we can synchronize the pose of the camera, the depth map and the RGB image coming directly from the Raspicam node. Now that the pose of the camera is published, also the depth map need to be constructed and published. The first thing that we need is to find all the map points which are visible in that particular keyframe and then translate them in the camera reference frame and project them in the camera image plane. Fortunately a method which is present in the mapPoint object allow us to get its world position and a method inside the mpCamera object allow us to project the map point into camera image plane. We can do everything using the following line of code

```
for (auto elem : mapPointSet) {

pcl::PointXYZ point;
Eigen::Vector3f pointWorldPosition = elem->GetWorldPos();
Eigen::Matrix3f R = quaternion.toRotationMatrix();
Eigen::Vector3f pointCameraPosition = R.inverse() *
(pointWorldPosition - translation);
Eigen::Vector2f positionInImagePlane =
(*SLAM.mpTracker).mCurrentFrame.mpLastKeyFrame->mpCamera->
project(pointCameraPosition);
}
```

at this point it is sufficient to associate for each element of LaserScan message vector the depth of the corresponding pixels in the image if it is available. Also in this case it is fundamental to set the timestamp in the header of the LaserScan object equal to the timestamp saved in the lastKeyFrame object to achieve synchronization as explained above.

These are the fundamental modification to the ROS Wrapper needed to make ORB-SLAM3 publish the needed data in the ROS network. In the next and last chapter of the thesis the implementation of the last node that used MiDaS to densify the map produced by ORB-SLAM, and the Octomap package used to combined all the point cloud produced by this new densification node, will be explained.

Chapter 4

Map Densification

In this fourth and last chapter of the thesis the use of the neural network for the map densification will be discussed. In particular the chapter will follow the following scheme

- In the first part a review of the physical setup and of the ROS network will be reported
- In second place the node built to use the neural network for the depth estimation, to rescale its estimate and to produce and publish the point cloud will be presented
- As a last step the Octomap package will be briefly described and the result obtained using the experimental setup will be showed

After this final chapter the conclusion regarding this project will be summarized considering the problems that characterized this solution and the possible future development.

4.1 Set-up Review

The Raspicam and the accelerometer has been already deeply describe at the beginning of the thesis in this section the various elements that compose the set-up will be briefly listed and particular attention will be given to the way in which they are physically connected one to the other. As a second step the description of the final ROS network will be given with all the nodes, the names and types of messages used.

4.1.1 Physical Set-up

The hardware elements that has been used to implement the experimental system are

- A Raspberry Pi 4 (4GB) which is the only elements that has not been already described. It is equipped with Ubuntu 20.04 (Server version) and ROS noetic. The Raspberry is used as an hub for all the sensors that are needed for the project and it is connected to the local area network to sent all the sensors data to the workstation for the elaboration
- The workstation is an MSI Laptop equipped with an 8th gen i7 processor, 16 GB of RAM, 512 GB of SSD and a GTX 1050 graphic card. It is used to run ORB-SLAM3, the node for the map densification and the OctoMap package to produce the global voxel map.
- the Raspicam which has been already described and which is directly connected to the Raspberry through a 15 pins camera serial interface (CSI) connector.
- the accelerometer which communicate with the raspberry through the I2C protocol. The most important problem of the accelerometer is that it is the first sensors to slow down if the Raspberry is engaged with other heavy tasks such as the recording of data coming from the sensors. This slow down is really serious since ORB-SLAM3 need that data coming from the accelerometer are published at an high and stable rate

the important thing to notice is that the set up is really cheap. In particular the sensors used are not expensive and the costly part of the set up is related to the computational power needed by all the nodes. Image 4.1 summarized the physical set up.

4.1.2 Review of the ROS Network

The second important thing to describe is the architecture of the ROS network which underpins the entire system. Here the nodes used and the messages published by them will be described

- The Raspicam node is the driver of the Raspicam and it published a lot of messages starting from the camera info (which contains for example the

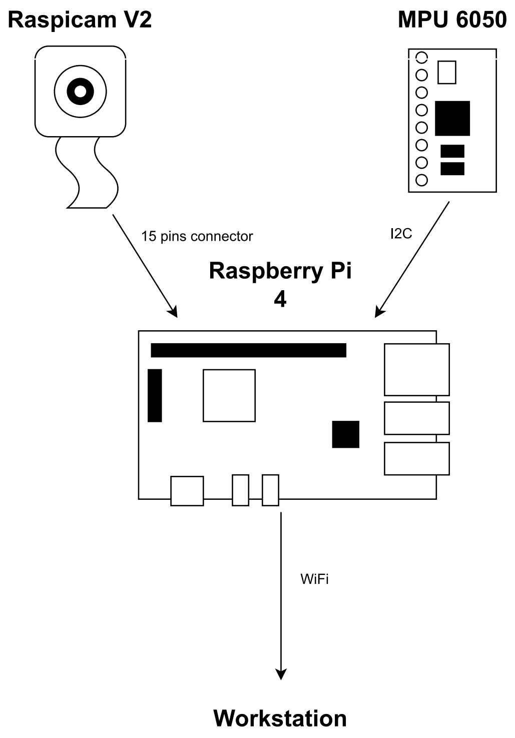


Figure 4.1: Schematic representation of the hardware setup

calibration data of the camera) to arrive to the uncalibrated and calibrated version of the images. The only message that is needed in this project is the */raspicam_node/image* which is then remmapped in our network as */camera/image_raw*. This message (which is an Image type message) contains the raw colored image produced by the camera with dimension 640x480 and with a frequency of 30hz.

- The accelerometer node publish data coming from the accelerometer with a frequency of 150hz. The type of the published message is the Imu type, so it contains data coming from the accelerometer and also from the gyroscope

- the densification node which is used to implement the Neural Network depth estimation and rescaling. This node is subscribed to the information produced by the modified ORB-SLAM3 Wrapper and the RGB image published by the Raspicam node to elaborate them and publish the point cloud of the visible environment.

- the ORB-SLAM3 Wrapper node which manage the interface between ORB-SLAM3 and the ROS network and that has been already deeply describe

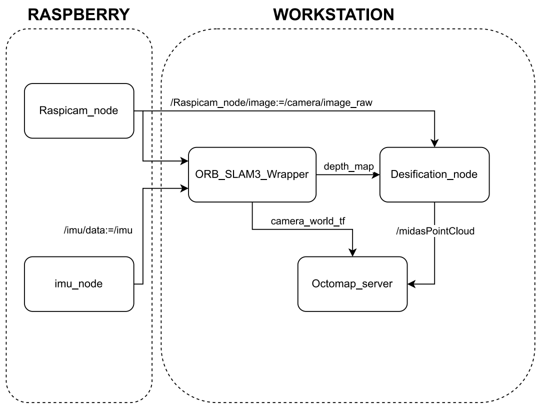


Figure 4.2: Schematic representation of the ROS network

at the end of Chapter 3.

- The OctoMap server node that is subscribed to the point clouds coming from the desnification node and which publish the 3D voxel probabilistic map.

A graphical representaion of the ROS network is presented in 4.2.

4.2 Densification node

The second section of this Chapter is dedicated to the description of the node added for the map densification. The messages to which the node is subscribed and the one which is published by it has been already described. Here the first validation of the sinchronization of messages to which the node is subscribed, the Neural Network used to produce the dense depth map and the OctoMap package will be rapidly discussed.

4.2.1 Synchronization and ORB-SLAM3 depth estimation validation

As already explained the node is subscribed to data coming from ORB-SLAM3 ROS Wrapper (the sparse depth map) and data coming directly from the Raspicam (the RGB image). The first thing to check is to understand if the node is able to subscribe to these two piece of information with the correct synchronization and to understand how accurate ORB-SLAM3 could be in the estimation of the depth of certain points in the image with the actual experimental set-up. This sparse depth information in fact will be used to rescale the depth map which is the output of the Neural Network. Moreover even if the accuracy of ORB-SLAM can be seen in the paper the experimental set-up used in this thesis has worse specification than the one used in the paper. The two most important deficit are

- the Raspicam is a rolling shutter camera rather than a global shutter one
- the accelerometer can be push to 150hz at most rather than 200hz. This limit is due to the bottleneck given by the Raspberry computational power and not to the IMU model. A lot of effort has been pushed to try to increase that frequency to 200hz, for example the node has been rewritten in C++ rather than Python and the frequency of the I2C bus has been increased but nothing can make the IMU run at a faster rate than 150hz.

To check if the synchronization is good once all the data are collected by the node the RGB image can be plotted and on the image some of the detected features can be drawn. Then one can check if the features remains stable even if the camera is moved. If, in addition to plotting the features, we also draw the estimated distances by ORB-SLAM3, we can also check how good is the depth estimation with our set-up. In image 4.3 an example of the plotted images can be seen, and in image 4.4 an Histogram of the absolute errors in the depth estimation is showed.

4.2.2 MiDaS

Now a brief discussion over the used neural network for depth estimation is needed. In particular Midas is Neural network for relative inverse depth estimation. The real novelty of this network is the dataset on which it has been trained. In fact the greatest problem when dealing with training depth estimation neural network (and neural network in general) is data availability. In this case the authors of the paper where able to train the network on multiple data set even if they used different type of annotations or incompatible data structures. Another key contribution of the paper is the use of 3D movies as source of information for the training since with them the relative depth information can be extracted through stereo matching. This source of training data really increment a lot the availability of information for the training procedure. In image 4.5 a depth map obtained with an image coming from the Rapsicam and using MiDaS can be observed. The depth map produced by the neural network is obviously

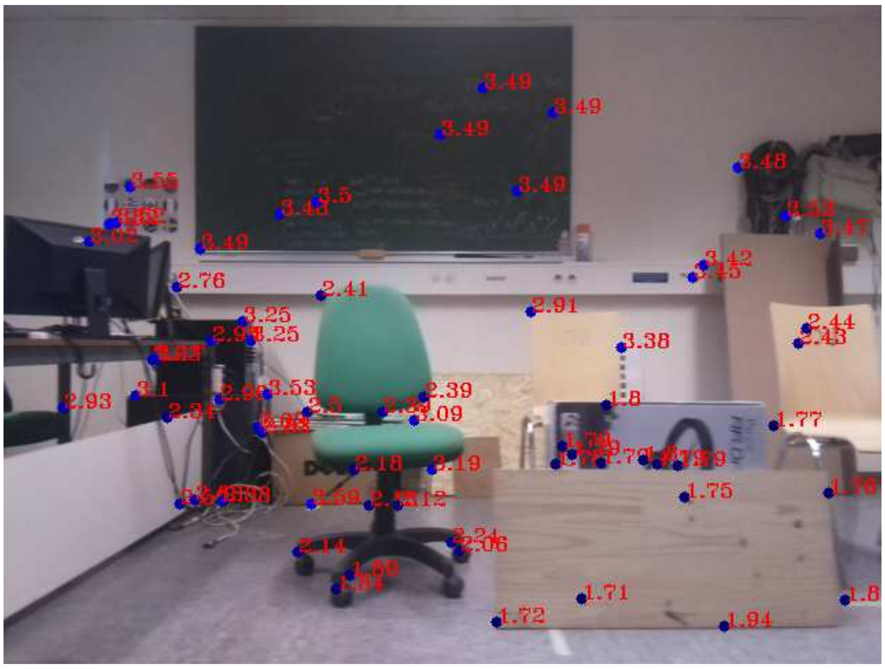


Figure 4.3: Example of an image with some of the detected orb features with the corresponding estimated distances. This images have been used to estimate the precision of ORB-SLAM3 with the actual experimental setup. The stability of the features between succesive images allow lso to check the synchronization of the RGB image with the depth map.

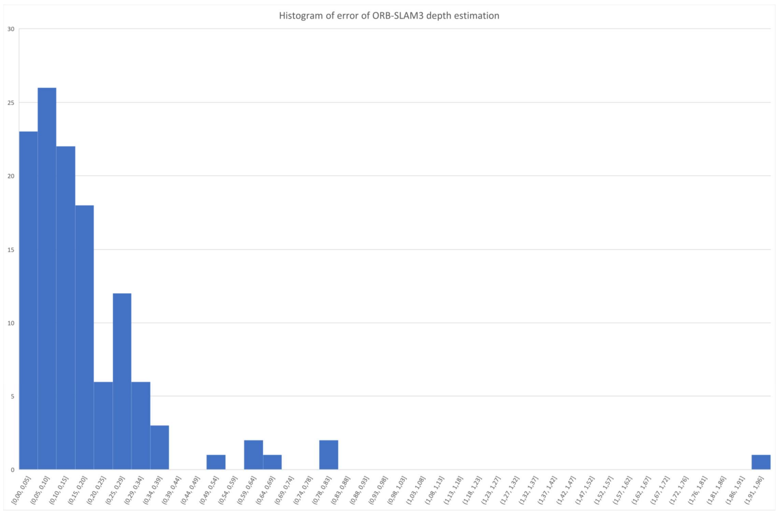


Figure 4.4: Histogram of depth estimation errors of ORB-SLAM3. The mean error is 18 cm and its variance is 0.049.

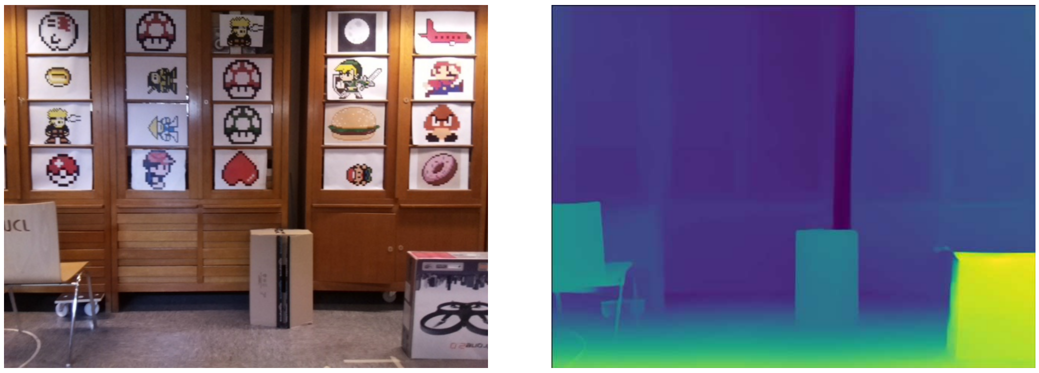

(a) RGB image given as an input to the network (b) Inverse depth output of the network which need to be rescaled

Figure 4.5: Input and output image of the MiDaS neural network

dense but it is represented in inverse depth and it is totally out of scale. So, once the non inverse depth has been recovered, simply calculating the inverse of the output of the network, the only remaining task is to rescale it.

4.2.3 Rescaling the NN output

The rescaling procedure is quite simple and it is based on the reasonable assumption that there is a linear relation between distances estimate by the neural network and distances coming from ORB-SLAM3. If we assume that, we can try to understand which is this linear relation and rescale all the NN output. To do that we simply assume that the true depth of each pixel is simply the output of the really simple linear model

$$y = ax + b$$

where y is the true depth corresponding to a particular pixel (given by ORB-SLAM3) and x is the relative depth estimate by the neural network. So if we collect all the points which are present in the sparse depth map produced by ORB-SLAM3 and the corresponding points in the relative depth neural network map we can set up a linear regression problem which, once is solved, allow us to obtain the a and b parameters. Once we know these parameters we can use them to rescale the entire relative depth map and obtain a real world scale depth images. Now using the the camera calibration data and the depth information that we have just obtained we can project points in the 3D space and produce a PointCloud2 message that can be used by OctoMap in combination with the current pose of the camera to produce the global map. In image 4.6 an example of the linear regression problem solved at each keyframe generation is showed. In

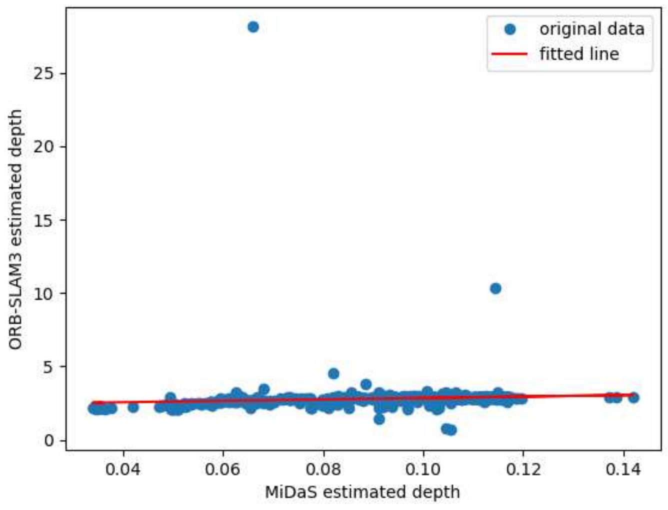


Figure 4.6: Example of a linear regression problem that is solved at each keyframe. The presence of outliers can be noticed.

that image we can see how the hypothesis of linear dependence can be considered quite reasonable.

4.3 OctoMap and final results

In this final section of the thesis a rapid high-level overview of the Octomap package is given and some example of the final voxel map that can be obtained using this method will be given. The important point here is that once the Octomap is available, path planning algorithm can be built using information coming from this map, allowing for autonomous exploration of unknown environment.

Octomap is a C++ library also available as a ROS package which allow to produce voxel map of the environment starting from a point cloud. In particular it is able to combine multiple point cloud of the environment if the position in which these point clouds are taken is known. The principal characteristics of OctoMap, as explained in the paper are the following one

- it use a probabilistic representation of the environment. This is really crucial since if a moving obstacle (for example a person) will pass in front of the robot we want that the robot sees it, but we do not want that the robot update the map with a fixed obstacle in each position in which this moving obstacle has been seen. So using this probabilistic approach allow to achieve dynamic map updating

- it represent also free space inside the environment in addition to occupied ones. This information is fundamental for path planning algorithm and so to achieve autonomous exploration
- it is really efficient from the point of view of memory management. This is another key point since when the map start to grow in size the memory requirement could increase a lot and so manage memory in the right way is crucial.

The two principal observation about the functioning of OctoMap are the following one

- *The map representation:* The map is represented as an octree [44] which is build subdividing a particular volume of space recursively into eight sub volume. The principal characteristic is that if all the child node of a particular node representing a volume are in the same state (occupied or not), they can bu pruned.
- *The free space.* The free space in the map is computed assigning the "free space" label to each volume of space between a detected occupied voxel and the current position of the camera (in our case), using simply raycasting to determine which of the voxels are in this particular position.

In this work there is no deep description of the underline functioning of Octomap but additional information for the interested reader can be found in [45].

In the final images presented in this thesis 4.7 an example of multiple points clouds and the 3D voxel map produced thanks to the combination of them can be seen. To validate these map, since confronting them with a ground truth needed the use of instrument for the measurement of 3D environment such as 3D Lidar, relative distances between points inside the map has been used. Here an important disclaimer on how these images have been obtained need to be done. In fact even if real time performance of the system could be achieved the problem is that, due to the cheap hardware that compose the set up, the performance of ORB-SLAM3 are not stable enough to allow our system to build a voxel map of the environment. So to obtain the map the procedure adopted has been the following one

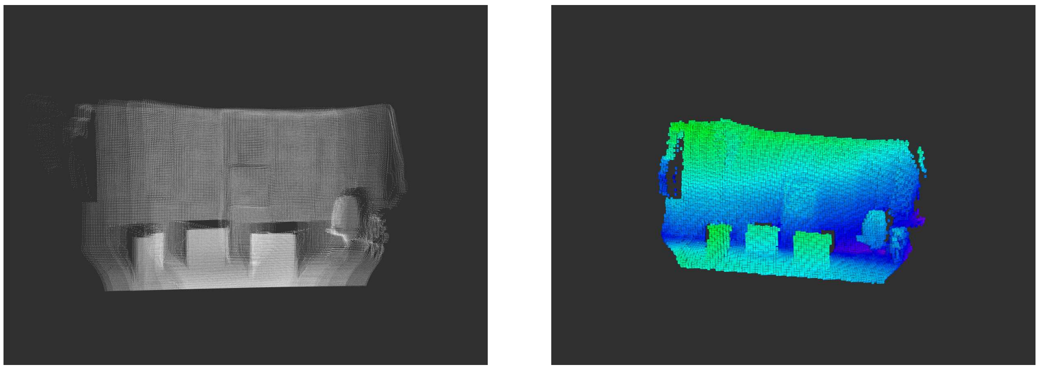

(a) Example of multiple point cloud obtained through the NN output rescaling **(b)** The resulting voxel map produced by OctoMap

Figure 4.7: Combination of multiple pointcloud produced by OctoMap

- data from the sensors has been recorded in a bag file
- these data were elaborated only by ORB-SLAM3 and the output of the algorithm (so the camera pose and the depth map) were recorded in a second bag file. In this way section of the sequence in which ORB-SLAM3 produced good result could be saved
- the recorded output of ORB-SLAM3 was given to the densification node and to the octomap package to produce the 3D voxel map

the procedure is quite complex and to achieve real time performance a more powerful workstation, a global shutter high-speed camera and a way to make the accelerometer run at stable 200 hz are needed. All these problem are mentioned again in the conclusion. The mean relative absolute distance error in the 3D voxel map is of about 10 cm with variance almost equal to $0,0053$ cm. Obviously other disclaimers need to be done. The mean relative error is computed on a not so large sample because produce each scene is quite complex as explain above. Moreover the relative measure are taken in the center of the scene where the neural network has better performance because putting obstacle in the high corner the room is difficult. It is true that because the robot can move it can try to change it position to achieve better understanding of part of the map for which it has not built a clear map. However these observation are necessary to explain one more time that this project has the goal to demonstrate the feasibility of this particular approach.

Conclusions

In this thesis work a way to densify the map produced by an inertial-visual SLAM algorithm has been defined. Moreover a real implementation of that procedure has been constructed. There are some problems with this first iteration of the project which are in particular

- the computational power needed is quite high since ORB-SLAM3, MiDaS and OctoMap need to run at the same time.
- the hardware at disposal is not powerful enough to extract the best possible performance from ORB-SLAM3, which sometimes can randomly change scale in an unreasonable way or loose track of the position (this problem lead to the use of a particular offline procedure to obtain the 3D voxel map as explained at the end of the last chapter of the thesis).
- the neural network used is able to achieve good performance in depth estimation but as always neural network are not a deterministic way to determine the depth of a particular scene and so they can sometimes lead to unexpected results.

These are obviously problem that need to be solved but the project succeed as a proof of concept, and demonstrate that with additional adjustment work it could lead to monocular dense SLAM with relatively cheap hardware. The first adjustment needed is more powerful hardware, in particular a global shutter camera could lead to really better performance and the second one is the better general organization of the project to obtain better efficiency regarding the use of computational resources. The most promising future development in my opinion is the use of depth completion network, which take both the RGB image and the sparse depth information coming from an inertial-monocular SLAM algorithm like ORB-SLAM3, to produce a depth map. This could lead to higher quality depth map since the network has additional information to estimate depth in addition to the RGB image. The challenge in this case is on how to build the dataset and on how to structure the training procedure. There exist already trained network for this particular task [46] [47] but they are really harder to integrate in a particular project differently from MiDaS.

At the end of this thesis I want also to list the most important things learned during this particular project

- the first thing is the mastering of the ROS environment and an increase experience with the use of programming languages like C++ and Python
- increasing understanding of the Linux operating system
- better understanding of the use of real sensors (camera and IMU calibration, communication protocol etc)
- improved ability to orient myself at the beginning of a really open project and totally unknown problem
- general knowledge of the focal point of the SLAM problem and history with particular attention to visual SLAM algorithm

SLAM algorithm are improving at an always faster rate and achieving a really general and robust solution to SLAM could lead to a real revolution in mobile robotics. I wish that this master thesis could be a good starting point for everyone who want to try to learn visual SLAM basic concepts to be able to explore then some of the key problems still unresolved in this field of study.

Bibliography

Bibliographic reference

[1] Dehem Boris. “Three dimensional monocular SLAM for autonomous drone navigation”. In: (2017) (cit. on p. viii).

[2] Kechtban Louai. “Toward an optimized 3D Monocular SLAM”. In: (2018) (cit. on p. viii).

[3] Shariq Farooq Bhat, Ibraheem Alhashim, and Peter Wonka. “AdaBins: Depth Estimation Using Adaptive Bins”. In: (June 2021). DOI: 10.1109/cvpr46437.2021.00400. URL: https://doi.org/10.1109%2Fcvpr46437.2021.00400 (cit. on p. ix).

[4] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardos. “ORB-SLAM: A Versatile and Accurate Monocular SLAM System”. In: *IEEE Transactions on Robotics* 31.5 (Oct. 2015), pp. 1147–1163. DOI: 10.1109/tro.2015.2463671. URL: https://doi.org/10.1109%2Ftro.2015.2463671 (cit. on pp. ix, 8, 51, 57).

[5] Carlos Campos et al. “ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM”. In: *IEEE Transactions on Robotics* 37.6 (Dec. 2021), pp. 1874–1890. DOI: 10.1109/tro.2021.3075644. URL: https://doi.org/10.1109%2Ftro.2021.3075644 (cit. on pp. ix, 8, 52).

[6] J. Engel, J. Sturm, and D. Cremers. “Semi-Dense Visual Odometry for a Monocular Camera”. In: (Dec. 2013) (cit. on pp. ix, 9).

[7] J. Engel, V. Koltun, and D. Cremers. “Direct Sparse Odometry”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (Mar. 2018) (cit. on pp. ix, 9).

[8] René Ranftl et al. “Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer”. In: (2019). DOI: 10.48550/ARXIV.1907.01341. URL: https://arxiv.org/abs/1907.01341 (cit. on p. ix).

[9] René Ranftl, Alexey Bochkovskiy, and Vladlen Koltun. “Vision Transformers for Dense Prediction”. In: (2021). DOI: 10.48550/ARXIV.2103.13413. URL: https://arxiv.org/abs/2103.13413 (cit. on p. ix).

[10] Cesar Cadena et al. “Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age”. In: *IEEE Transactions on Robotics* 32.6 (Dec. 2016), pp. 1309–1332. DOI: 10.1109/tro.2016.2624754. URL: https://doi.org/10.1109%2Ftro.2016.2624754 (cit. on pp. 1, 5, 20–22).

[11] H. Durrant-Whyte and T. Bailey. “Simultaneous localization and mapping: part I”. In: *IEEE Robotics Automation Magazine* 13.2 (2006), pp. 99–110. DOI: 10.1109/MRA.2006.1638022 (cit. on pp. 1, 20).

[12] T. Bailey and H. Durrant-Whyte. “Simultaneous localization and mapping (SLAM): part II”. In: *IEEE Robotics Automation Magazine* 13.3 (2006), pp. 108–117. DOI: 10.1109/MRA.2006.1678144 (cit. on pp. 1, 20).

[13] Xiang Gao et al. “14 Lectures on Visual SLAM: From Theory to Practice”. In: (2017) (cit. on p. 2).

[14] Xuqin Wu et al. “Loop Closure Detection for Visual SLAM Based on SuperPoint Network”. In: *2019 Chinese Automation Congress (CAC)* (2019), pp. 3789–3793 (cit. on p. 5).

[15] Raul Mur-Artal and Juan D. Tardos. “ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras”. In: *IEEE Transactions on Robotics* 33.5 (Oct. 2017), pp. 1255–1262. DOI: 10.1109/tro.2017.2705103. URL: https://doi.org/10.1109%2Ftro.2017.2705103 (cit. on pp. 7, 8, 51).

[16] Lukas von Stumberg and Daniel Cremers. “DM-VIO: Delayed Marginalization Visual-Inertial Odometry”. In: *IEEE Robotics and Automation Letters* 7.2 (Apr. 2022), pp. 1408–1415. DOI: 10.1109/lra.2021.3140129. URL: https://doi.org/10.1109%2Flra.2021.3140129 (cit. on p. 9).

[17] Taihú Pire et al. "Stereo parallel tracking and mapping for robot localization". In: (2015), pp. 1373–1378. DOI: 10.1109/IROS.2015.7353546 (cit. on p. 9).

[18] Taihú Pire et al. "S-PTAM: Stereo Parallel Tracking and Mapping". In: *Robotics and Autonomous Systems* 93 (2017), pp. 27–42. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2017.03.019. URL: https://www.sciencedirect.com/science/article/pii/S0921889015302955 (cit. on p. 9).

[19] 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. 10).

[20] S. Kohlbrecher et al. "A Flexible and Scalable SLAM System with Full 3D Motion Estimation". In: (Nov. 2011) (cit. on p. 11).

[21] Wolfgang Hess et al. "Real-Time Loop Closure in 2D LIDAR SLAM". In: (2016), pp. 1271–1278 (cit. on p. 11).

[22] Joan Solà, Jeremie Deray, and Dinesh Atchuthan. "A micro Lie theory for state estimation in robotics". In: (2018). DOI: 10.48550/ARXIV.1812.01537. URL: https://arxiv.org/abs/1812.01537 (cit. on p. 19).

[23] J. Stillwell. "Naive Lie Theory". In: Undergraduate Texts in Mathematics (2008). URL: https://books.google.it/books?id=SuR5OAgxyDIC (cit. on p. 19).

[24] Roger Howe. "Very basic Lie theory". In: *The American Mathematical Monthly* 90.9 (1983), pp. 600–623 (cit. on p. 19).

[25] Nebot Durrant Rye. "Localisation of automatic guided vehicles". In: *Robotics Research: The 7th International Symposium (ISRR'95)* (1996), pp. 613–625 (cit. on p. 21).

[26] Edwin Olson. "AprilTag: A robust and flexible multi-purpose fiducial system". In: (May 2010) (cit. on p. 29).

[27] Luc Oth et al. "Rolling Shutter Camera Calibration". In: (2013), pp. 1360–1367. DOI: 10.1109/CVPR.2013.179 (cit. on p. 30).

[28] "Introducing the Raspberry Pi Cameras". In: (-). Accessed: 09-09-2022 (cit. on p. 30).

[29] Patrick Walter. "Review: Fifty Years Plus of Accelerometer History for Shock and Vibration (1940–1996)". In: *Shock and Vibration* 6 (Jan. 1999), pp. 197–207. DOI: 10.1155/1999/281718 (cit. on p. 31).

[30] "Allan Variance ROS". In: (-). Accessed: 09-09-2022. URL: https://github.com/ori-drs/allan_variance_ros (cit. on p. 34).

[31] David W Allan and Judah Levine. "A historical perspective on the development of the Allan variances and their strengths and weaknesses". In: *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 63.4 (2016), pp. 513–519 (cit. on p. 35).

[32] "MPU 6050 datasheet". In: (-). Accessed: 09-09-2022. URL: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf (cit. on p. 35).

[33] "MPU 6050 register Map". In: (-). Accessed: 09-09-2022. URL: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6500-Register-Map2.pdf (cit. on p. 35).

[34] D.G. Lowe. "Object recognition from local scale-invariant features". In: 2 (1999), 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410 (cit. on p. 39).

[35] "Introduction to SIFT (Scale-Invariant Feature Transform)". In: (-). Accessed: 16-09-2022 (cit. on p. 39).

[36] Ethan Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: (2011), pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544 (cit. on p. 40).

[37] Marius Muja and David Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." In: *VISAPP 2009 - Proceedings of the 4th International Conference on Computer Vision Theory and Applications* 1 (Jan. 2009), pp. 331–340 (cit. on p. 41).

[38] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. "EPnP: An accurate O(n) solution to the PnP problem". In: *International Journal of Computer Vision* 81 (Feb. 2009). DOI: 10.1007/s11263-008-0152-6 (cit. on p. 43).

[39] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: https://doi.org/10.1145/358669.358692 (cit. on pp. 43, 44).

[40] Raul Mur-Artal and Juan D. Tardós. “Fast relocalisation and loop closing in keyframe-based SLAM”. In: (2014), pp. 846–853. DOI: 10.1109/ICRA.2014.6906953 (cit. on pp. 47, 49).

[41] Raul Mur-Artal and Juan D. Tardos. “Visual-Inertial Monocular SLAM With Map Reuse”. In: *IEEE Robotics and Automation Letters* 2.2 (Apr. 2017), pp. 796–803. DOI: 10.1109/lra.2017.2653359. URL: https://doi.org/10.1109%2Flra.2017.2653359 (cit. on p. 52).

[42] Christian Forster et al. “On-Manifold Preintegration for Real-Time Visual–Inertial Odometry”. In: *IEEE Transactions on Robotics* 33.1 (Feb. 2017), pp. 1–21. DOI: 10.1109/tro.2016.2597321. URL: https://doi.org/10.1109%2Ftro.2016.2597321 (cit. on pp. 56, 57).

[43] “How to install OpenCV 4.5.2 with CUDA 11.2 and CUDNN 8.2 in Ubuntu 20.04”. In: (-). Accessed: 16-09-2022. URL: https://gist.github.com/raulqf/f42c718a658cddc16f9df07ecc627be7 (cit. on p. 61).

[44] Donald J. Meagher. “Geometric modeling using octree encoding”. In: *Comput. Graph. Image Process.* 19 (1982), p. 85 (cit. on p. 75).

[45] Armin Hornung et al. “OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees”. In: *Autonomous Robots* (2013). Software available at https://octomap.github.io. DOI: 10.1007/s10514-012-9321-0. URL: https://octomap.github.io (cit. on p. 75).

[46] Jinsun Park et al. “Non-Local Spatial Propagation Network for Depth Completion”. In: (2020) (cit. on p. 78).

[47] Alex Wong and Stefano Soatto. “Unsupervised Depth Completion with Calibrated Backprojection Layers”. In: (2021), pp. 12747–12756 (cit. on p. 78).