ALMA MATER STUDIORUM - UNIVERSITÁ DI BOLOGNA

SCUOLA DI INGEGNERIA

DIPARTIMENTO DI
INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE
"Guglielmo Marconi"
DEI

MASTER DEGREE IN AUTOMATION ENGINEERING

GRADUATION THESIS
in
System Theory and Advanced Control M

Embedded SLAM algorithm based on ORB-SLAM2

CANDIDATE:
LUIGI CHIOCCI

SUPERVISORS:
CHIAR.MO PROF LORENZO MARCONI
DOTT. ING. BIAGIO TRIMARCHI

ACADEMIC YEAR 2022/2023
GRADUATION SESSION II

Abstract

The goal of this dissertation was to present a comprehensive analysis of the ORB-SLAM2 algorithm. By introducing the basis of points triangulation and ORB features, the whole structure of the algorithm has been analyzed, with a centralized focus on the graph-based optimization involved, as well as the place recognition mechanism. Additionally, the original code has been analyzed and optimized, resulting in a substantial increase in time performance while keeping a similar accuracy to the original one, proved by several simulations performed. The final goal of this thesis has been the testing of the obtained algorithm on a real quadcopter and the analysis of its outcomes: the results were affected by the limited computational resources available in the vehicle, obtaining a lower accuracy with respect to the simulation, but still proving the efficiency of the improvements applied on the original code.

Contents

Introduction

Nowadays, Simultaneous Localization and Mapping (SLAM) has emerged as a fundamental problem in robotics, involving a vast variety of topics, from computer vision to mathematics. SLAM algorithms are involved in a wide range of applications, including for example autonomous vehicles, drones, and augmented reality.

The main obstacle to the development of efficient and robust SLAM algorithms on embedded systems is their high computational demand for achieving high accuracy.

Among the various SLAM algorithms, the families of the ORB-SLAM ones have become a milestone in the realm of visual SLAM. Starting from the first algorithm introduced in [1], its authors have continued to evolve it over the years creating ORB-SLAM2, introduced in [2], up to to the most recent ORB-SLAM3, brought to life in [3]. This family of algorithms, thanks to their robustness, efficiency and open-source availability, has gained a great response from the scientific community.

The purpose of this thesis is to dive deeply into the functioning of the ORB-SLAM2 algorithm, keeping as goal the understanding of the weaknesses and strengths of the original implementation and to try to optimize it, by increasing its real-time capability while keeping the same performance. This is done in the optics of an implementation on a real embedded device, more precisely a quadcopter, to see if it's possible to bring the power of state-of-the-art algorithms into a platform with limited capabilities. This work hopes to highlight the possibilities of improvements of already existing SLAM algorithms, by showing that a deep knowledge of the original material can be exploited to enhance its qualities.

More precisely, the thesis is structured in the following way: in the chapter 1 an introduction of the SLAM thematic, its classifications and some famous example are illustrated, along with a more extended focus on the visual SLAM scenario; in the chapter 2 a brief explanation of common computer vision topics and problematic, together with a detailed explanation of the ORB features used by the ORB-SLAM family is present; then, in the chapter 3 an illustration of the fundamentals of graph based optimization and its applicability to SLAM algorithms, with an enhanced focus on its use in the ORB-SLAM case, is performed; in the chapter 4 the main techniques for place recognition exploited by the ORB-SLAM2 algorithm are presented, along with their use in loop closing and relocalization; the chapter 5 presents some topics and ideas that can be investigated in the future for further increase the efficiency of the ORB-SLAM2 algorithm, some of which are taken directly from ORB-SLAM3 but needs more study in order to be applicable in embedded platforms; finally, in the chapter 6 an explanation of the changes performed on the original code is presented, followed by the results obtained both in simulations (performing a comparison with the original code) as also the result of the test performed onto the real quadcopter.

Chapter 1

SLAM introduction

Simultaneous Localization and Mapping (SLAM) is a key element of every application involving an autonomous vehicle, especially for situations where information regarding position and environment is not known in advance. Given this property, a vast range of techniques have been developed to face the problem, resorting to different types of sensors and assumptions both on the environment and the vehicle considered.
Some techniques, for example, are specifically designed for terrestrial vehicles, exploiting the advantage of having to estimate only 2D-pose (this means needing two coordinates for translation and one for the orientation) to fully localize the vehicle in an environment.
The more challenging scenario is the one regarding aerial vehicles, which needs a 3D-pose to be correctly localized, so it requires 3 variables for translation and 3 for the orientation.

1.1 SLAM classification

The various existing SLAM algorithms can be categorized based on the type of map created, the type of sensors used and the type of functions implemented.

1.1.1 Map types

SLAM algorithms can build three main different types of maps: metric, topological and semantic.
Metric maps are detailed reconstructions of the environment where the position of each object or landmark is defined by a set of coordinates. The maps reconstructed in this way can be dense, sparse or semi-sparse, depending on how many points of the environment are triangulated and used. Dense maps carry more information at the cost of being more memory-consuming, while

sparse maps are more efficient, containing fewer points, but less useful for tasks such as accurate reconstruction of the environment.
Topological maps are instead more focused on creating a map based on topological information (like mutual connections) regarding the different positions or areas of the environment. A common example of this type of map is the generalized Voronoi graph [4], which divides the environment space into areas, depending on the distance from obstacles and other details, constructing a type of representation pretty useful for path planning.
Semantic maps are becoming more and more powerful in SLAM applications involving neural networks. They incorporate semantic information on the observed environment, like for example objects' labels, which can be used for advanced tasks like obstacle detection. Given they're nature, they are mostly used in applications regarding advanced autonomous driving and when the computational burden is not an issue.
SLAM algorithms can of course use a mixed representation of the environment, without limiting themselves to only one, increasing the power and usability of the map.

1.1.2 Sensor types

SLAM algorithms can resort to different types of sensors to capture information about the environment and/or the robot's state. The type of sensor strongly impacts the type of map that could be created and also the type of algorithms that could be used.
However, rather than using a single sensor type, multi-sensor SLAM algorithms have been proposed, allowing to enrich the accuracy and the type of information provided. Of course, using more sensors results in higher economic costs and also higher computational demands, given the necessity of elaborating more types of information.

LIDAR A vast gamma of SLAM algorithms involves the usage of LIDAR (Light Detection and Ranging) sensors to perceive the environment and build a map of it. They can be 2D or 3D, depending on the environment's nature to observe: a terrestrial vehicle will need only a 2D LIDAR, while an aerial one should use a 3D one, in order to reconstruct a wider portion of the environment with a single scan.
LIDAR sensors can build a map of the surroundings with high accuracy, detecting even small particles. This allows, in the case of 3D LIDAR, to construct also point-clouds of the environment that can be used to build a dense map of it, almost effortlessly. For those reasons LIDAR sensors have spread enormously in SLAM applications, both for 2D and 3D localization.

The main drawback of LIDAR sensors is their high cost, especially for high-quality and 3D ones, which limits their usage in various applications and some fields.

Inertial Sensors Another set of sensors commonly used in SLAM are inertial sensors like accelerometers and gyroscopes. They are useful to read information about the robot state, like linear acceleration and angular velocities, beside the environment.
The main advantages of this sensors are being economic and lightweight, but they suffer from high disturbances and a severe error accumulation, especially the information regarding the linear accelerations, that makes their usage basically useless without any additional information. For this reasons they are almost never used alone, but always in collaboration with other sensors to enrich the information provided to the SLAM algorithm.

Camera The sensors that bring the higher balance between cost and information submitted are cameras. They are mostly economic and can bring rich information about the environment.
The information contained in an image can be used to reconstruct 3D positions of the points observed but, compared to the LIDAR sensors, their accuracy decreases when moving away from the camera. In addition to this, the information provided is bi-dimensional so, in order to correctly triangulate a point of the environment, more observations of the same point are required. A common solution to this problem is the usage of a special type of multi-camera, like for example stereo ones, which are composed of more cameras aligned on the same axis, that allows triangulating the positions of the observed points from a single observation from all the cameras involved.
SLAM systems based on cameras are also spreading in a very fast manner, especially in the last few years. Since localization using only camera information is more challenging with respect to using sensors such as LIDAR (because it involves further elaboration of the data, computer vision techniques, and is more prone to environmental disturbances), it has become a main topic in SLAM literature.

RGB-D RGB-D sensors are special types of cameras that provide color and depth information in a single step, enhancing the quality of data provided to the algorithm with respect to classic cameras. They have a higher cost than cameras, with in general smaller resolutions. However, the high quality of the information provided has caused a high spread of the use of this type of sensor.

1.1.3 Algorithms

SLAM systems are mostly based on statistical techniques, which aim to produce an accurate estimation of the robot's state and the sensed map. Since all the variables in the general setting are unknown, all the quantities describing the problem are probabilistic. The most common techniques used are Extended Kalman Filters (EKF), particle filters and maximum a posteriori estimation (MAP).

Extended Kalman Filter One of the oldest methods of SLAM is the one using Extended Kalman Filter (EKF) to estimate the agent's pose, the environment or both.
The EKF is the non-linear version of the Kalman Filter, which is the optimal linear estimator for linear systems subjected to additive white noise. Since the systems in which SLAM algorithms are involved are strongly non-linear (as most of the systems of interest in engineering), studies were performed to extend the theory and the advantages of the Kalman Filter to non-linear systems. In a compact mathematical form, the system of interest in which the EKF can be applied is the following:

$$\begin{aligned} \boldsymbol{x}_{t+1} &= f(\boldsymbol{x}_t, \boldsymbol{u}_t) + \boldsymbol{w}_t \qquad & \boldsymbol{w}_t &\sim \mathcal{N}(0, Q_t) \\ \boldsymbol{z}_t &= h(\boldsymbol{x}_t) + \boldsymbol{v}_t & \boldsymbol{v}_t &\sim \mathcal{N}(0, R_t) \end{aligned} \tag{1.1}$$

where $\boldsymbol{x}_t$, $\boldsymbol{u}_t$ and $\boldsymbol{z}_t$ are respectively the system state, the control input and the measurement, all considered in discrete-time, and $\boldsymbol{w}_t$ and $\boldsymbol{v}_t$ are the process and observations noise, both assumed zero-mean multivariate Gaussian noises with covariance Q_t and R_t. As stated, both the system model and the measurement one are non-linear.
Given a measurement of the real variable $\boldsymbol{z}_t$, the steps to estimate the system state are mainly two: first, a prediction is performed, exploiting the system knowledge, then the correction using the real measurement is applied.

$$\begin{array}{ll} \text{Predicted State Estimation} & \hat{\boldsymbol{x}}_{t|t-1} = f(\hat{\boldsymbol{x}}_{t-1|t-1}, \boldsymbol{u}_t) \\ \text{Predicted Covariance Estimate} & P_{t|t-1} = F_t P_{t-1|t-1} F_t^T + Q_t \\ \text{Measurement Residual} & \tilde{\boldsymbol{e}}_t = \boldsymbol{z}_t - h(\hat{\boldsymbol{x}}_{t|t-1}) \\ \text{Residual Covariance} & S_t = H_t P_{t|t-1} H_t^T + R_t \\ \text{Kalman Gain} & K_t = P_{t|t-1} H_t^T S_t^{-1} \\ \text{Update State Estimate} & \hat{\boldsymbol{x}}_{t|t} = \hat{\boldsymbol{x}}_{t|t-1} + K_t \tilde{\boldsymbol{e}}_t \\ \text{Update Covariance Estimate} & P_{t|t} = (I - K_t H_t) P_{t|t-1} \end{array} \tag{1.2}$$

In the (1.2), the variables F_t and H_t are respectively the Jacobians of the system model and the measurement one. This algorithm is general and has become a standard technique for SLAM. In the most general case, the system state represents both the agent position and the environment landmark coordinates, while the measurement model depends of course on the type of sensor considered. Its advantages are that is easily implementable and requires low computational power (directly related to the state dimensions), but it heavily depends on the initial estimate of the state and the correctness of the model, so its performances drop quickly in case of non-modeled elements or other type of disturbances, like a non-static environment. Another drawback of EKF is directly related to their state dimension that, since it involves all the landmarks present in the map, could easily bring an explosion of computational complexity if used for large environments and for long periods of time. Despite that, the SLAM algorithms based on EKF are very common and their performances are reasonable, especially when aided with high-precision sensors like GPS.

Particle Filters Particle filters, also known as sequential Monte Carlo methods, are a type of algorithms that aims to find an estimate of the internal state of a dynamical system. This type of methods doesn't take any assumptions on the type of errors and disturbs involving the variables, as long as the system state behaves as a Markov process (by mean, the current state only depends on the previous one).
The Monte Carlo localization at each step tries to find the robot pose by finding the optimal one in a set of N *particles* $\mathcal{X}_t = \{\boldsymbol{x}_t^1, \boldsymbol{x}_t^2, \ldots, \boldsymbol{x}_t^N\}$, which are predicted states of the system, each one with a proper weight. They are usually initialized with a uniform distribution along the environment and then, as long as new sensor information comes in, the particles' positions are updated to find the best state estimate.
At every time step each particle is updated according to the system model:

$$\hat{\boldsymbol{x}}_{t+1}^i = f(\boldsymbol{x}_t^i, \boldsymbol{u}_t) \tag{1.3}$$

where w_t is a disturbance with an unknown distribution. Each particle's weight is then updated using an *importance function*, a special type of probabilistic function that takes as input the previous weight, the predicted state, the predicted observations (according to the (1.1)) and the sensor information. These weights are then normalized.
These updates, going on with time, cause most of the particle weights to go to almost zero, causing a degeneration of the performances. This problem is reduced by *resampling* the particles, eliminating the ones with small weights and creating new ones associated with the higher weights. This method allows the particles to converge to the correct state of the system if a sufficient number of observations are provided.
The main drawbacks of this method are the assumptions of the Markov model, which eliminates the possibility of using it in a non-static environment, and

the computational effort, which grows linearly with the number of particles and with the state dimensions. Since having a high number of particles assures higher accuracy, a compromise must be found. Fortunately, techniques such as Kullback-Leibler divergence (KLD) sampling can be used, which adapts the number of particles used for prediction according to the error estimate.

Maximum a posteriori estimation This type of techniques (known as MAP) are very popular when the sensors involved provide visual data of the environment, such as 2D image or, more in general, when a structure resembling an optimization problem can be found. MAP algorithms, in general, are strongly related to the Maximum likelihood (ML) estimation since they aim to maximize the likelihood function $f(z|x)$, the probability of z when the parameter is x, with respect to the model parameter. In the specific SLAM case the observations are the information provided by the sensors and the model parameters are the system state's coordinates. In addition to the classic ML estimation, the assumption of MAP methods is that a prior distribution of x, such as $g(x)$, is known, and this allows to calculate the likelihood function using the Bayes' theorem:

$$f(x|z) = \frac{f(z|x)g(x)}{\int_{\mathcal{X}} f(z|x)g(x)dx} \tag{1.4}$$

where $\mathcal{X}$ is the domain of the density function $g(x)$.
The MAP method then estimates x as the mode of the posterior distribution of the variable:

$$\hat{x}_{\text{MAP}} = \arg\max_x f(x|z) = \arg\max_x f(z|x)g(x), \tag{1.5}$$

where the denominator in the (1.4) has been canceled out since it doesn't depend on x. Bundle adjustment (BA) techniques take the theory of MAP estimates and adapt them to be used with image data, relating the likelihood function to the reprojection error, allowing to minimize a cost function that connects the system state (such as landmarks and robot position) to the pixels data provided by the sensors.
The biggest advantage of this approach is that allows the optimization of not only the current system state using only the current data, but also the optimization of past states using past data, allowing a better estimation of the current state and reducing the error drift due to errors accumulation in the measurement of the environment.
Despite the solution of an optimization problem could be in general more computationally demanding with respect to the other methods seen, the nature of the problem allows the use of special optimization algorithms that exploit the structure of the problem (like Levenberg–Marquardt, one of the best

ones for non-linear least square optimization problems) and the sparsity of the same (due to the lack of interaction among the different parameters of the various poses). These peculiarities, along with an increase in the computational power availability of commercial devices have led to a huge spread of this type of algorithms in the literature.

1.2 Visual SLAM

As stated before, SLAM algorithms that use cameras as main sensors have spread more and more in recent years [5]. These types of algorithms, which use only visual information, are specifically referred as visual SLAM (vSLAM) algorithms. With respect to the ones that use other sensors, vSLAM techniques present in general a higher difficulty, because cameras can acquire less visual input from a limited field of view, and the information provided requires in general further elaboration before they could be used for position estimation.

1.2.1 Modules of vSLAM

Basically, all vSLAM algorithms follow a common modular structure, with two main modules always running (*tracking* and *mapping*) plus an *initialization* one used to initialize the system and the map. In addition to these modules, other two important ones are usually implemented in order to obtain a robust and accurate vSLAM algorithm: *relocalization* and *global map optimization.*

Initialization The initialization is mandatory for every algorithm since it defines the global coordinate system and an initial map of the environment is computed in order to start to estimate the pose. Depending on the type of sensors (monocular, stereo or RGB-D) initialization could be more or less demanding. A RGB-D or stereo camera can track immediately the depth of the observed points (even if with uncertainty depending on the distance from the sensor), allowing an immediate first triangulation of the point of interest in the environment. A monocular sensor, instead, needs more observations of the target points in order to correctly triangulate them; this requires so more sensor information and more time in order to correctly estimate an initial map of the environment.

Tracking Once the system is running, the tracking module is in charge of estimating the camera position given the last or the latest observations. The basic idea, not depending on the specific type of algorithm used, is to match

already mapped points with the ones observed and to use the correspondence between the 2D coordinates on the camera and the 3D poses of the points to reconstruct the pose of the camera. Since the position of the drone could be used by other critical time algorithms, the new pose prediction must be performed as quickly as possible.
The accuracy of the tracking not only depends on the disturbances of the sensors, but also on the reliability of the mapped points. Poor accuracy of the map causes a degrading in the performance of the tracking, which could result also in wrong pose estimates.

Mapping The mapping module is in charge of reconstructing and enriching the environment map using new observations. The construction of the map is a crucial task because the pose estimation can be only as good as the map is accurate. An accurate map could also be used by other algorithms like planning or obstacle avoidance.
New observations can not only be used to track new points, but also to refine the ones already observed, augmenting the map's quality.
Since the triangulation of the observed points strongly depends on the camera position, mapping and tracking are strongly coupled modules, where errors in one task impact the other and vice versa.

Relocalization When fast motion occurs, or when the camera encounters strongly disturbed scenarios (like blur, strong light change and so on), the track could fail. It's exactly in this situation that the relocalization module starts its task. It has the job of estimating again the position of the robot with respect to the map.
Without a proper and efficient relocalization technique, most of the SLAM algorithms become practically useless, because if their tracking module fails, the position of the robot is lost forever. This scenario is also referred to as the ”kidnapped robot problem” in literature.

Global map optimization As said before, since tracking and mapping are strongly related, they suffer of mutual errors, that usually accumulate over time, resulting in an increasing pose estimation drift. For long tracking scenario this could result in unusable pose prediction.
The job of this module is precisely to reduce and correct, when possible, this accumulated error. The basic idea is to use the whole global map (while tracking only needs the portion that the camera is able to see) and to recognize already visited places. When a positive match is encountered, information is then used to enforce a loop constraint, used to suppress the accumulated error.

1.2.2 vSLAM approaches

The families of all the vSLAM algorithms belong to two main typologies: *feature-based* approach and *direct* approach. The first ones are based on extracting feature points from the images and using them to build the map, being also the first approach introduced, in the early 2000s. On the contrary, direct approaches are based on using the whole image for tracking, without extracting features points. This type of algorithms are especially useful with texture-less or feature-less environments.
In the last few years, there has been also an increase of vSLAM algorithms that uses neural networks to capture semantic information on the environment, but their computational requirements are high and not easily satisfied by most of the common mobile devices such as robot and vehicles, resulting more interesting in application like autonomous driving. For this reason they will be not treated here.

Feature-based methods This type of methods involves filter-based algorithms and also BA-based ones. Their performance depends not only on the type of estimation algorithm used, but also on the feature extraction methods: algorithms extracting high-level features are often more computationally demanding, but allow a higher accuracy of the map created. Given the nature of the features, and also depending on the type of sensor used, the map created by this type of method is usually sparse.
MonoSLAM [6] is the first monocular vSLAM, developed in 2007 by Davison et al. Based on feature extraction, it estimates both the camera position and the 3D structure of the environment with an EKF, while the initialization is performed by observing a known object with a defined global coordinate system. As stated before, the main drawback of EKF methods is that its computational demand increases with the environment's size: so if the explored ambiance is large, the increasing dimension of the state vector can easily lead to a loss of real-time constraints.
PTAM [7] is the first algorithm that introduced the splitting of the tracking and mapping in two different threads, allowing the division of the computational burden of the two modules. The tracking is performed by projecting the mapped points on the camera to make 3D-2D correspondences, while the 3D features are optimized in a BA algorithm that is run in the mapping thread. The initialization is performed using the 5-Point algorithm: a technique that allows to estimate the relative position of two cameras using a minimal set of features' correspondences. The second main contribution of PTAM is the introduction of keyframes-based mapping. Essentially, new map points are triangulated only from certain positions, called keyframes, which are frames with a substantial disparity between them and the other keyframes. The triangulation performed in this way is more accurate than

using standard frames. In the mapping thread some keyframes are then used to perform a local BA, to increase the accuracy of the tracked points. Finally, a global BA run to optimize all the keyframes to reduce the accumulated error. Since the accuracy of vSLAM algorithms depends on the number of feature points, an EKF approach has worse performances with respect to BA methods, since the latter allows the maintenance of a bigger map without a degradation in the performances. The global BA allows also to maintain the geometric consistency of the whole map, even if problems of local minimum could arise and cause a degradation in the results. After the introduction of PTAM, other algorithms were constructed to improve the performances and to correct the weaknesses. The ORB-SLAM [1] algorithm and its evolved versions [2] [3] are, so far, one of the best circulating vSLAM algorithms.

Direct methods This type of method, in contrast to the previous ones, use directly the input image without any further elaboration. To estimate the position of the camera photometric consistency is often used as an error measurement, calculating the intensity of the pixels with respect to the predicted ones.
DTAM, proposed by Newcombe et al. [8], is the first monocular SLAM system that creates a dense 3D surface model and uses it for camera tracking. The mapping is performed using a multi-baseline stereo (to achieve a high depth's accuracy) and then optimized by considering space continuity. Tracking is instead performed by registration of the observed image and the 3D map model, while initialization is performed with stereo measurements. The density of the problem requires however a GPU to allow the system to track the camera pose with real-time performances.
LSD-SLAM [9] is another leading method above the direct ones. With respect to the previous algorithm, LSD-SLAM tracks the position of the camera reconstructing only areas with high-intensity gradients, by so ignoring texture-less areas. In the mapping modules, the depth values are first set randomly and then optimized using photometric consistency. LSD-SLAM implements also loop-closure detection and a 7 DoF global pose-graph optimization (with monocular cameras, the depth of the point is an additional parameter of the problem since it cannot be fixed with a single observation), to assure high accuracy and small errors. With respect to DTAM, this semi-dense approach allows real-time performance also on CPUs, resulting in higher integrability on mobile devices.

Chapter 2

Triangulation basis and ORB features

Since the algorithm treated is specifically designed for use with cameras, an illustration of the basic theory of triangulating new points from images is now performed, as well as the definition of the type of features used, ORB, along with the techniques for their extraction.

2.1 Triangulation basis

The process that allows to determine a point in the 3D space given its projection on an image, is known as triangulation in computer vision theory. The basic formulation of the problem assumes a pinhole model for the camera: no lenses are present and the aperture of the camera is seen as a point. Given its simplicity, this model can be used only as a first-order approximation for the mapping of 3D points to 2D ones.
However, this geometry can be adapted to several types of cameras, depending on the lens and optics structure used. However, independently from the camera type, the inverse mapping procedure (so from a 2D view to a 3D correspondence), is more complex, since it requires more than one observation from different positions to correctly estimate the scale of the point.

2.1.1 Perspective projection

The general geometry of the system is reported in the figure 2.1, where the point P with coordinates x, y, z in the camera coordinate system (with the z axis called also as *optical axis*) is projected onto the point Q, with coordinates u, v, on the camera plane trough the optical center C. The focal plane, the place where the projecting hole is situated, is parallel with distance f, called *focal length*, to the focal one.

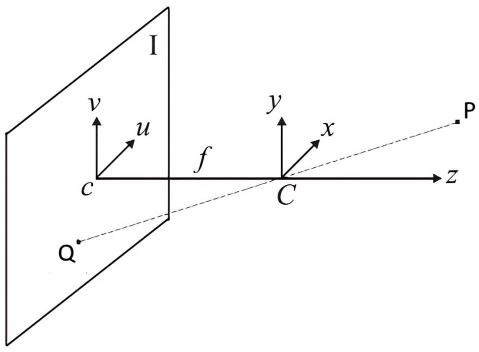


Figure 2.1: *Scheme of a general pinhole camera.*

The relationship between the coordinates of the point Q and P is given by the simple optical projection law:

$$\frac{u}{x} = \frac{v}{y} = -\frac{f}{z} \quad \rightarrow \quad \begin{cases} u & = -x\frac{f}{z} \\ v & = -y\frac{f}{z} \end{cases} . \tag{2.1}$$

Since this projection leads to a rotation of the observed points, a change of coordinates is introduced, assuming the origin of the camera coordinate system on the upper left corner of the camera plane, with the v axes pointing downward. This change of reference system cancels out the minus sign from the (2.1) and requires adding a translation represented by the coordinates of the camera center c (this point is obtained by intersecting the optical axis with the camera plane). In practice the f coefficient is not the same for x and y, due to flaws in the camera sensor, errors in camera calibration and other sources of non-ideality. For this reason they are distinguished in two different coefficients, f_x and f_y, and the resulting projection equation is:

$$\begin{cases} u & = f_x\frac{x}{z} + c_x \\ v & = f_y\frac{y}{z} + c_y \end{cases} \tag{2.2}$$

where the c_x and c_y are the coordinates of the camera center.
In reality the coordinates of the projected point, u and v, are not continuous but discrete, since they are associated with the pixels of the image. For this reason, the parameters are usually expressed in pixel units as measurement units. Using pixels unit, the inequality between f_x and f_y can be seen as a non-perfect square form of the single pixel.
This parameters are also used in literature to build the intrinsic matrix of a camera, defined as

$$K = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.3}$$

where the parameter γ is introduced in the cases of axis skew (that causes shear distortion of the image) and is usually zero. This matrix can so be used to project the normalized 3D coordinates $\{x/z, y/z, 1\}$ onto the point $\{u, v, 1\}$ by a simple multiplication.
The pinhole model is impracticable in reality since a point-dimensional hole would never ensure sufficient light intensity for the sensor to correctly capture an image. Lens use was introduced exactly for this reason, since it permits to focus more light on a single point, enhancing the quality of the image.
Lens causes however several distortions to the captured images, so a calibration of the camera is mandatory in order to estimate the distortion parameters and compensate them, without which the information acquired would be useless. There are two major kinds of distortion studied in the literature for pinhole cameras: radial and tangential. The first causes straight lines to appear curved, enhancing this effect the farther the points are from the image center, and the distortion can be modeled as:

$$\begin{cases} x_{\text{distorted}} & = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{\text{distorted}} & = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{cases} \tag{2.4}$$

where r is the distance of the considered pixel from the center of the camera and k_1, k_2, k_3 are characteristic parameters of the lens used. The second type, the tangential one, occurs when the lens plane is not perfectly parallel to the image plane, causing some areas to look closer than expected. This distortion is usually modeled as:

$$\begin{cases} x_{\text{distorted}} & = x + [2p_1 xy + p_2(r^2 + 2x^2)] \\ y_{\text{distorted}} & = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \end{cases} \tag{2.5}$$

where p_1 and p_2 also depends on the lens. So, to correctly rectify the image (name of the process that allows to eliminate the distortion effects and to apply the (2.2)), at least 5 parameters are needed, k_1, k_2, p_1, p_2, k_3, where the last parameter is usually set to zero to reduce the order of the radial effects and so to simplify the image elaboration.
Fortunately there are plenty of open-source libraries, like OpenCV, with already implemented functions that allow to easily calibrate cameraa (the process to extract the characteristic parameters of the camera) and to elaborate the images provided.

Kannala-Brandt model Despite being very simple, the pinhole camera model (2.2) cannot be applied to all types of cameras, even if the distortion models (2.4) and (2.5) are considered. This is the case of the fisheye lens, invented in 1906 by Robert W. Wood: a particular type of photographic lens with an ultra-wide angle (usually around 180°, sometimes even bigger). In this dissertation, fisheye lenses are taken under observation because the type of cameras mounted on the drone used are Intel RealSense Tracking Camera T265, and they involve the use of fisheye lenses to capture images.
This particular type of view causes the sensor to capture a spherical vision and to project it on a circular image, causing naturally a severe distortion, making, therefore, the previous model useless.
Kannala and Brandt, in [10], have introduced a new type of model with the purpose of canceling out the difficulties of adapting the classic pinhole model to the fisheye lens.
While pinhole cameras obey the classic (2.1), that can be summarized with the relationship

$$r = f \tan\theta \tag{2.6}$$

where r is the distance of the projected point from the image center and θ is the angle between the optical axis and the projecting ray, the fisheye lens is more suited to be modeled with the so-called *equidistance projection*

$$r = f\theta \tag{2.7}$$

where points are projected through an observing semi-sphere instead of a plane like the classic model.
The distortion model is in this case directly applied on the angle θ, becoming

$$\theta_{\text{distorted}} = \theta + k_1\theta^3 + k_2\theta^5 + k_3\theta^7 + k_4\theta^9. \tag{2.8}$$

Then, from the (2.7) with the distorted angle, is possible to reconstruct the two pixel coordinates u and v as

$$\begin{cases} u &= f_x(x' + \gamma y') + c_x \\ v &= f_y y' + c_y \end{cases} \qquad \begin{aligned} x' &= \left(\frac{\theta_{\text{distorted}}}{r}\right)\frac{x}{z} \\ y' &= \left(\frac{\theta_{\text{distorted}}}{r}\right)\frac{y}{z} \end{aligned} \tag{2.9}$$

where, by naming $f'_x = f_x\theta_{\text{distorted}}/r$, $f'_y = f_y\theta_{\text{distorted}}/r$ and by putting γ equal to zero as in the usual case, a structure equivalent to the (2.2) can be obtained, allowing to interface it with all the structure using pinhole models effortless.

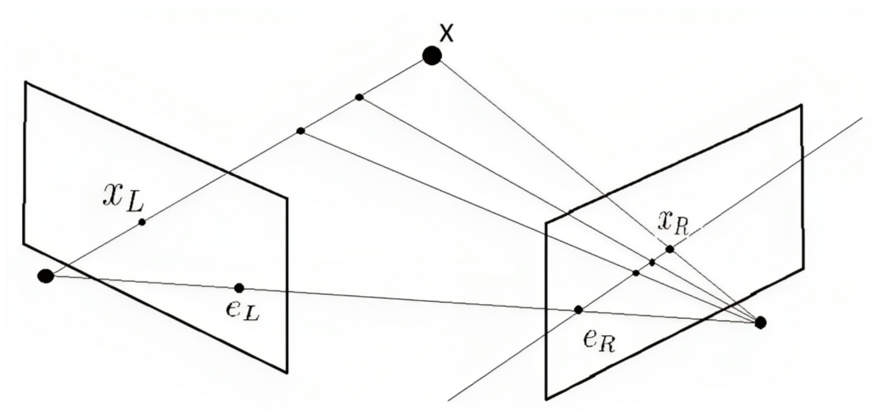


Figure 2.2: *Scheme of epipolar constraints of two cameras observing the same point.*

Naturally, a calibration technique to obtain the intrinsic parameters of the camera (pinhole, fisheye or any other model) is mandatory, and this is usually done by capturing an image with a known pattern and then reconstructing the parameters from a set of measurements. Different techniques, both for pinhole and fisheye models exist, but they are outside of the purpose of this dissertation and will not be reported. The parameters used for rectification and undistortion are all already provided or in the dataset used for simulations, or as data directly published by the T265 camera.

2.1.2 Epipolar Geometry

As it can be seen by the (2.2), with a single image point is impossible to recover the full 3D pose of the original point, since the system of equations' rank is less than three, having more variables than equations at disposal. Using the pinhole camera model it can be proven that in fact all the points that lie on an optical ray (a line that passes through the pinhole), are projected onto the same 2D point, resulting in insufficient information to reconstruct the original position.
However, this situation is solved when the same point is observed by two or more distinct positions since certain geometrical relationships arise between the different projected points. These geometric constraints are called *epipolar constraints* and are the basis for stereo vision and point triangulation.
Given two pinhole cameras that look at the same point X, as reported in the figure 2.2, different characteristics can be observed: for simplicity, in order to avoid the inversion of the projection in this example, the two image planes have been brought in front of the optical center.
The projection of each camera's optical center into the other camera's plane

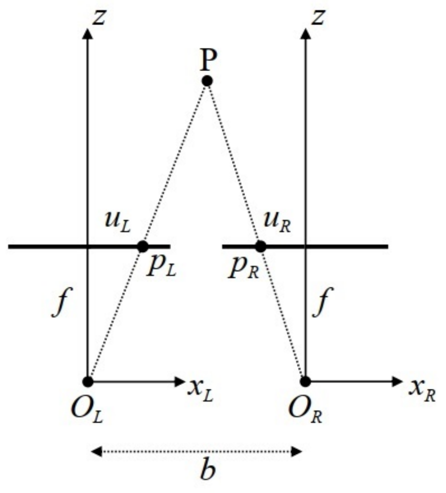


Figure 2.3: *Illustration of the standard stereo projection.*

creates a pair of points, called *epipoles*, respectively e_L and e_R. Given their construction, both the epipoles and the two optical centers lies on the same 3D line.
Given x_L and x_R, respectively the projections of point X on the two camera's plane, two special lines can be defined, called *epipolar lines*, that connect respectively the points x_L and e_L on one camera and x_R and e_R on the other. A peculiarity of the epipolar lines so constructed is that each point lying on the ray that connects the point X to its projection on one of the two camera's plane (x_L or x_R), is projected on the epipolar line of the other images.
This constraint is used to enforce the triangulation of the same 3D point seen from different positions and also to check if two projected points belong to the same 3D point.
A further simplification of the epipolar geometry happens when the two camera's plane coincides, as in the case of stereo cameras. This type of camera, precisely designed for obtaining a more accurate scale estimation of the observed points, is simply obtained by putting two cameras aligned on the same plane, with a certain translation along the horizontal axis. In this case the two epipolar lines coincide and all the epipolar lines of the images are parallel to the line that connects the two optical centers.
In the resulting scheme, as it can be seen in the image 2.3, the coordinates of the 3D point, with respect to the two different cameras' coordinate systems, are

$$\begin{aligned} x_L &= x_R + b \\ y_L &= y_R \quad , \\ z_L &= z_R \end{aligned} \tag{2.10}$$

where b, called *baseline*, is the distance between the two optical centers. Now, by writing the (2.2) for both the cameras, a new constraint connecting the two coordinates u_L and u_R can be written:

$$u_L - u_R = b\frac{f_x}{z}. \tag{2.11}$$

This difference between u_L and u_R is usually called *disparity* in literature.
The equations (2.11) and (2.2) can be put together to find a system of equations with full rank:

$$\begin{cases} u_L &= f_x \frac{x}{z} + c_x \\ v_L &= f_y \frac{y}{z} + c_y \\ u_R &= f_x \frac{x-b}{z} + c_x \end{cases} \tag{2.12}$$

where, for simplicity, the intrinsic parameters have been assumed equal for both cameras.
Those relationships can finally be inverted to find the coordinates of the 3D point given the pixel coordinates on the two images:

$$\begin{cases} z &= \frac{bf_x}{u_L - u_R} \\ x &= z\frac{u_L - c_x}{f_x} \\ y &= z\frac{v_L - c_y}{f_y} \end{cases}. \tag{2.13}$$

2.1.3 Linear triangulation

While the equation (2.13) holds for points that are close to the camera, its precision decreases the further the landmarks are from the image plane. This is caused by the geometry of the projection mechanism that directly links the precision of a point projected into a pair of pixels to its distance from the camera, resulting in a bigger amount of points being projected in the same image area.
For this reason the above benefits of stereo cameras are used only for fast close points triangulation, while for far ones the solution is to resort to standard triangulation techniques that use multiple observations from different positions. Therefore, in the algorithm considered, not only the points whose a stereo match has been found are kept, but also all the other ones, since other matches could be found from other observations.
There are several ways of performing this triangulation, and the one used in the ORB-SLAM2 is the one using tools from linear algebra to solve it.
Since the equations (2.2) and (2.13) perform a projection starting from a point already in the camera coordinate reference system, a relationship between the 3D pose of the point in the global coordinate system and the one expressed in the camera frame must be defined, such as

$$\boldsymbol{x} = T\boldsymbol{X}, \quad T \in \mathbb{R}^{3\times 4} \tag{2.14}$$

where the T matrix, known as *projection matrix*, describes the projection from the world coordinate system, expressed in homogeneous form ($\boldsymbol{X} = [x_w, y_w, z_w, 1]^T$) into the camera frame ($\boldsymbol{x} = [x_c, y_c, z_c]^T$), where the 4th homogeneous coordinate has been dropped to simplify the mathematical structure, given its redundancy. The matrix T can in fact be obtained from the first three rows of the homogeneous matrix T_{cw}, the one describing the position of the world coordinate system with respect to the camera one. The camera point $\boldsymbol{x}$ can then be projected into the camera plane (or reconstructed from it) using the standard (2.2).
In this case the quantity $\boldsymbol{X}$ is the unknown variable to be determined and, by not having reliable information on the scale from one single observation, at least two of them are needed. Writing so the linear system obtained by putting together two of the (2.14), gained from two different positions, an exact solution could ideally be found, but given the noises and flaws over the two measurements, this usually leads to a non-solvable problem.
However, the equation (2.14) can be seen as a similarity transformation, since if a scale factor is involved the result doesn't change: every point on the same ray as $\boldsymbol{X}$ and $\boldsymbol{x}$ is projected into the same one, and those points can be found by multiply the (2.14) by any scale factor. Given this property, instead of using the whole vector $\boldsymbol{x} = [x_c, y_c, z_c]^T$ that has only two independent coordinates, being derived from the (2.2), a scaled quantity of this vector can be used, namely $\boldsymbol{x}' = [x', y', 1]^T$, where $x' = x_c/z_c$ and $y' = y_c/z_c$ can be obtained directly from the pinhole model.
This type of problem is tackled with the *Direct Linear Transformation* (DLT) method, which consists of scale factor removal and transformation into a linear system, that can be solved with the tools of linear algebra.
The basic idea of DLT is to apply the cross-product property of parallel vector (that is equal to zero) to the scaled quantity in the (2.14):

$$\boldsymbol{x} = \alpha T\boldsymbol{X} \quad \rightarrow \quad \boldsymbol{x} \wedge T\boldsymbol{X} = 0. \tag{2.15}$$

So, by performing the due calculations, the linear system resulting from the cross-product can be written as

$$\boldsymbol{x} \wedge T\boldsymbol{X} = \begin{bmatrix} y'\boldsymbol{t}_3^T\boldsymbol{X} - \boldsymbol{t}_2^T\boldsymbol{X} \\ \boldsymbol{t}_1^T\boldsymbol{X} - x'\boldsymbol{t}_3^T\boldsymbol{X} \\ x'\boldsymbol{t}_2^T\boldsymbol{X} - y'\boldsymbol{t}_1^T\boldsymbol{X} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \qquad T = \begin{bmatrix} \boldsymbol{t}_1^T \\ \boldsymbol{t}_2^T \\ \boldsymbol{t}_3^T \end{bmatrix} \tag{2.16}$$

where the last row is a linear combination of the other two.
Now, by putting together two of the (2.16) originated from two observations of the same point and by defining $x_i' = \frac{u_i - c_x}{f_x}$ and $y_i' = \frac{v_i - c_y}{f_y}$ (where the sub-

scripts i are referred to the camera considered), the following homogeneous linear system can be obtained:

$$A\boldsymbol{X} = \boldsymbol{0}, \qquad A = \begin{bmatrix} x'_1\boldsymbol{t}_{31}^T - \boldsymbol{t}_{11}^T \\ y'_1\boldsymbol{t}_{31}^T - \boldsymbol{t}_{21}^T \\ x'_2\boldsymbol{t}_{32}^T - \boldsymbol{t}_{12}^T \\ y'_2\boldsymbol{t}_{32}^T - \boldsymbol{t}_{22}^T \end{bmatrix} \tag{2.17}$$

where the first subscript on t_{ij} specifies the row of the matrix taken, while the second refers to the matrix itself (referring to the first or to the second camera pose).
Since the equality to zero will never be reached, given the non-ideality of the observations, the solution is to find the value of X that minimizes the (2.17), and this is given by the right eigenvector associated with the smallest eigenvalues of A. This can be easily done by performing a SVD of A, which can be done by many numerical calculus libraries.

2.2 ORB features

Oriented FAST and Rotated BRIEF (commonly called ORB) was developed by Ethan et al. in [11] as an efficient alternative to the, at the time, most used feature detectors, SIFT and SURF: these two algorithms, despite having optimal accuracy, where first of all patented, so not for free use, and also computationally expensive, with no real-time performance on devices without GPUs. The idea of ORB features is to combine two methods, one for the extraction and one for the description, from which it takes the name: the FAST detector, with additional scale and orientation components, and a modified version of the BRIEF descriptor.

2.2.1 Oriented FAST

FAST detector, as the name suggests, is a rapid method for extracting features from an image. By recalling computer vision fundamentals, a point in an image can defined as a feature if it represents a recognizable and distinguishable entity, like an edge or similar, of a 3D correspondence. The FAST detector, to identify features, simply compares the intensity of a pixel with the ones in a fixed size circle around it: if more than a fixed threshold (that is one of the parameters that can be customized) of them are darker or brighter than the target pixel, the latter is set as a keypoint. The basic idea behind this is that if a point represents an element like an edge, it will be on the corner of the observed object, and so an observable change of intensity around it should be observable. Of course this idea is not flawless, since shadow and

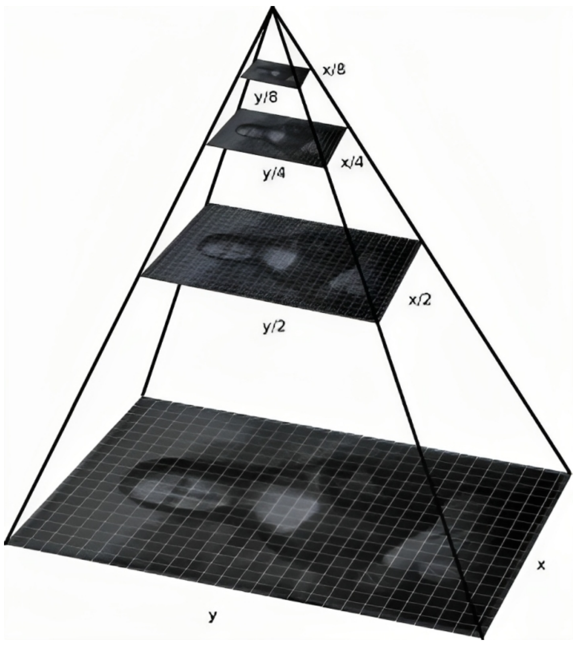


Figure 2.4: *Example of pyramid construction from an image.*

illumination, for example, could lead to a misdetection of the feature. These errors, however, can be mitigated by performing a substantial number of observations followed by an outlier rejection policy, as will be stated later.
Since FAST doesn't measure the goodness of the keypoints detected, a Harris corner measure is implemented to keep the best N points of the ones detected, in order to discard the ones considered as misdetection of edges.
Another addiction to FAST, performed by the author of the article, is the production of multi-scale features with the construction of a multi-scale image pyramid, as it can be observed in the figure 2.4: at each level of the pyramid (that is a scaled version of the previous layer), FAST keypoints are detected and then measured with the Harris detector. The first layer of the pyramid is usually the image itself, while the scale factor is a parameter of the ORB features that can be set in advance.
The best keypoints individuated are kept along with the level of the layer at which they were identified: this assures a partial scale-invariant property of the ORB features, being able to identify the same 3D keypoint on multiple scales, and also to partially estimate the accuracy on the measurement of the observed point.
The last addition to the FAST detector is an orientation measure, applied with the *intensity centroid* method, since the original version of FAST doesn't bring a rotation measurement. The assumption of this technique is that a corner's intensity is offset from its center, so the idea is to use the vector that connects the corner's center to the centroid to estimate the orientation of the detected

feature.
The centroid is calculated as:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \qquad m_{pq} = \sum_{x,y} x^p y^q I(x,y), \tag{2.18}$$

where the sum on x and y is performed on the pixels inside a fixed radius circle around the keypoint. Once the the vector that connects the keypoint to the centroid has been constructed, the orientation is easily computed as

$$\theta = \text{atan2}(m_{01}, m_{10}). \tag{2.19}$$

By using this evolved version of FAST detector, keypoints from a picture can easily be detected, each one with an associated rotation and scale.

2.2.2 rBRIEF descriptor

Once the coordinates of a keypoint have been identified, is mandatory also to compute a descriptor: a quantity that must be as unique as possible for the observed 3D point, in order to correctly identify the same one even if observed by different positions.
The standard BRIEF (Binary Robust Independent Elementary Features) descriptor is essentially a bit string description of an image patch (a set of pixels around a target one). It is computed by randomly taking two pixels inside this patch of which the intensity is compared, according to the following formula

$$\tau(p; x, y) := \begin{cases} 1 & : p(x) < p(y) \\ 0 & : p(x) \geq p(y) \end{cases}, \tag{2.20}$$

where the quantity $p(x)$ is the intensity of the pixel at position x.
The feature descriptor is then defined as a vector of n binary tests:

$$f_n(p) := \sum_{1 \leq i \leq n} 2^{i-1} \tau(p; x_i, y_i). \tag{2.21}$$

Of course, the bigger n, the better the descriptor is. This test could also be extended to compare 3 or 4 points for each test, obtaining two bits for each one of them. A common way to choose the random points is to use a Gaussian distribution around the interested keypoint.
Before performing the test is important to smooth the images, to obtain more robust descriptors, and this is usually done with a Gaussian filter.
However, the original BRIEF descriptor is not rotational invariant, so a modification was mandatory to increase its performance. The solution was to use the angle θ obtained in the (2.19) to *steer* the BRIEF descriptor, incorporating

in this way the orientation information into the vector itself. This is done by defining the matrix S as

$$S = \begin{bmatrix} x_1, & x_2, & \dots, & x_n \\ y_1, & y_2, & \dots, & y_n \end{bmatrix}, \tag{2.22}$$

filled with the coordinates of all the pixels chosen for the binary test, and steering it with the rotation expressed by θ:

$$S_\theta = R_\theta S. \tag{2.23}$$

Finally, the rBRIEF descriptor, by putting together the original BRIEF descriptor and the new information about the feature orientation, can be computed as

$$g_n(p, \theta) := f_n(p) | (x_i, y_i) \in S_\theta. \tag{2.24}$$

To speed up the computation of the descriptor, the angle θ is discretized into increments of 12 degrees ($2\pi/30$) and the BRIEF patterns can be precomputed in advance and put into a look-up table.
To measure how similar a keypoint is to another, the two points' descriptors are compared. Since the two vectors are binary, the Hamming distance can be used, being basically a count of the number of bits that are different for each position, which can be done in a very fast way.
The combination of FAST and rBRIEF (the "r" is added to denote the rotation application) allows so to obtain a feature detector and descriptor, that is an order of magnitude faster with respect to SIFT and SURF extractor, being easily implementable on simple CPU's only hardware.

Chapter 3

Graph-based SLAM

As stated in chapter 1, the PTAM algorithm was one of the first to use the idea of a graph-based SLAM, but not the first one. The idea was first introduced in [12] where, in contrast with the other, at the time, SLAM techniques (that used mostly incremental approaches), the idea of keeping the data of all the passed frames and then performing a maximum likelihood estimation was proposed. In this way the accumulated errors of the various measurements could be all taken into account together, resulting in an accuracy increase.

3.1 Illustration of graph-based SLAM

The basic idea of graph-based SLAM is to organize the SLAM problem into, as the name says, a graph that highlights the spatial structure of the problem. Each robot pose is modeled as a node in this graph, each one labeled with its position. All the information acquired from environmental observations and/or from odometry measurements are modeled as edges between two nodes, as stated in [13].
Each edge, essentially, represents a constraint between the two connected nodes' poses and, since the information and data involved are all affected by uncertainties and/or errors (since they originate from sensor measurements), they can mathematically be modeled as probability distributions over the relative transformations between the two poses while, for the same reasons, the nodes can be modeled as distributions over the actual poses of the robot or agent considered.
Each constraint can be built by taking into account odometry related information, directly connecting two poses, or can be determined by aligning the observations information taken by two distinct robot locations, as it is in the case of vSLAM. In this latter case, the 3D poses of the landmarks observed could be included as variables in the graph itself (despite not being actually nodes), so that they could be optimized later along with the nodes (even the

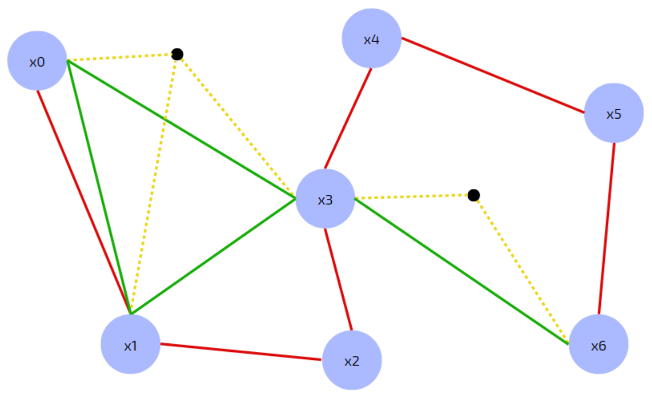


Figure 3.1: *Example of graph construction from a set of measurements. In the picture all the nodes report the agent pose and the connections between them are highlighted: the red lines are edge created from odometry measurements, while the green lines are edges created from observation of mutual landmarks (the observations are reported in yellow dotted lines).*

map points' poses can be modeled as random distributions, constrained by all the robot's poses that observe it). An example of the structure of the graph here described is reported in the figure 3.1.

Given these premises, a graph-based SLAM can be usually decoupled in two subsequent tasks: the *graph construction* (building of the graph structure from the sensors measurements) and the *graph optimization* (finding the most likely configuration of edges and poses given the measurements).

Since both nodes and edges originate from probability distributions that could very likely overlap each other, multiple potential edges connecting different poses could be generated by a single observation, leading to the description of the connectivity of the graph itself as a probability distribution. Since this could easily leads to an exploding complexity (in a combinatorial speed), in most practical approaches a simplification is applied, by restricting the possibles topologies of the graph to the most probable ones. This can be done by interleaving the graph construction with the graph optimization, in a way that at each new node insertion the graph structure is as close as possible to the most likely one.

Having stated the general idea for the graph construction, the specific details heavily depend on the desired implementation and could vary a lot from one algorithm to another. As far as concern the optimization, instead, a deeper analysis can be exploited, in order to highlight the general structure of the problem: by assuming a Gaussian distribution for the noise affecting the observations and recalling the MAP estimate (1.5), the graph optimization can be reduced to the computation of a Gaussian a posteriori approximation over the robot poses. Thanks to the special structure of a generic multivariate Gaussian distribution

$$f(\boldsymbol{x}) = \frac{1}{\sqrt{2\pi^n \det \Sigma}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right) \quad \boldsymbol{x}, \boldsymbol{\mu} \in \mathbb{R}^n, \Sigma \in \mathbb{R}^{n\times n} \tag{3.1}$$

with mean $\boldsymbol{\mu}$ and covariance matrix Σ, this maximization problem can be simplified. It can be proven that maximizing a likelihood function depending on a random variable with a given probability density function is equivalent to the maximization of the density itself. By also observing that the likelihood function could be substituted by its logarithm (given its crescent monotony), the maximization of the problem involving the (3.1) automatically translates into a non-linear least squares form

$$\log f(\boldsymbol{x}) = -\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu}), \tag{3.2}$$

where all the constants that don't play any role in the maximization have been canceled out. It is then easy to understand that the variable that maximizes the (3.2) is the mean itself, so the optimization is performed in order to find it. The (3.2) is essentially the form of the standard residual of the problem: the general cost function to optimize the graph can be obtained by summing up all the residual regarding the robot's and the map points' poses. By recalling the system and the observation models, $f(\boldsymbol{x_i}, \boldsymbol{u_i})$ and $h(\boldsymbol{x_i}, \boldsymbol{p_j})$ (where x_i is the agent pose and p_j is the landmark pose), the general optimization problem can be written as

$$\min_{\boldsymbol{x}_i, \boldsymbol{p}_j} \sum_{i=0}^{N-1} ||f(\boldsymbol{x_i}, \boldsymbol{u_i}) - \boldsymbol{x}_{i+1}||^2_{\Lambda_i} + \sum_{i=0}^{N} \sum_{j=1}^{M} ||h(\boldsymbol{x_i}, \boldsymbol{p_j}) - \boldsymbol{z}_{ij}||^2_{\Sigma_{ij}}, \tag{3.3}$$

where N and M are respectively the number of robot poses and landmarks, Λ_i and Σ_{ij} are the covariance matrices associated with the error in the odometry and in the observation, and finally $\boldsymbol{z}_{ij}$ is the measurement of the landmark j from the position i.

By analyzing the (3.3), the high coupling of the problem can be highlighted, especially by the second term. However, the special construction of the graph, with edges and nodes, makes the structure of the second term pretty sparse, since only the terms that correspond to actual edges are different from zero. This, of course, doesn't mean that the poses are independent (given the strong connectivity of the graph, all the poses depend on the others, as well as the map points), but only that the corresponding error term is not present in the (3.3) and that the information is propagated in an indirect way.

This sparsity of the problem can be exploited by several numerical algorithms in order to speed up the computation and the resolution of the general problem. Of course, given the non-linearity of the terms, a proper initialization is crucial for achieving a solution close to the optimal one (given the presence of local minima). This reflects the precedent issue of correctly interspersing the graph construction with the graph optimization because leaving too much behind the latter will result in an incorrect estimation of the solutions.
A common technique done for reducing the computational effort of performing the (3.3), is to optimize only a portion of the graph, usually around the last node inserted: this reduces the burden of the computation but of course, could lead to an increasing drift (due to error accumulation) as it would happen with standard incremental SLAM algorithm, so it should be properly managed.

3.2 Graph structure and creation

The ORB-SLAM algorithm [1] resorts to a graph-based structure performing the construction as done in the PTAM algorithm [7]: without using all the camera poses, but only a subset of them, called *keyframes* (where a classic camera pose is usually referred as a *frame*). This leads to a graph reduced in size but with still enough information to perform an accurate estimation, making the problem faster to be solved without impacting the quality of the result.
The main difference with respect to the formulation previously seen, given its implementation for a vSLAM algorithm, is that there are no odometry edges since every connection of the graph is only due to mutual landmarks' observations. This could seem like a deterioration of the general graph structure, but the observations measurement provides still enough information to build a well-posed graph connectivity, resulting in good accuracy and a decrease in the computational burden.

3.2.1 Connections

In order to obtain a connectivity that brings enough information for a correct estimation, but at the same time doesn't become too redundant for keeping the computational effort low, the connections between keyframes are created in a selective manner: basically, only keyframes sharing (by mean, both observing) a minimum number of points are considered connected. The connections created with this procedure form the so-called *covisibility graph*, where two keyframes are linked only if they share at least 15 points.
While the covisibility graph is used especially for local optimization, so for retrieving connectivity information only for a small portion of the graph, global

optimizations (such the one brought by a loop closure) use a more restrictive spanning graph of the covisibility one, where only links that are composed by more than 100 points are kept: this new graph is called *essential graph*. This is done to keep low the effort done for optimizing the complete structure but still preserving enough information to assure a correct estimation of the poses involved.
To ensure a fast keyframe retrieval, another connectivity information of the graph that is kept: the *spanning tree*. This is updated every time a node is inserted or deleted, and basically links each keyframe to a parent, designed as the one with the most point in common among the one in its covisibility neighbors. This is a minimal representation of the graph, by keeping only the more promising edges for each pair of keyframes, which allows the retrieval and the propagation of information among them in a very fast manner. In order to not have orphan keyframes, every time that a node with children is deleted, they are repartitioned between their covisibility connections and the culled keyframe's parent.
The last type of link that could be present in the graph used is a *loop edge*, which is created every time a loop closure is performed, by strongly connecting two keyframes observing the same place. This type of connection is the stronger one, since it binds two keyframes together, not only with a covisibility mechanism but also enforcing a loop connection that helps in reducing the accumulated error during the life of the algorithm.

3.2.2 Keyframes

The process of selection of keyframes is crucial since too many of them would cancel out the benefits of a reduced graph size, while too few would impact negatively the graph connectivity, making it useless. To maintain the number of keyframes stable, a double control is performed: a selection procedure assures keyframe insertion only when needed, and a culling one assures the elimination of redundant keyframes that do not bring additional information.
The general condition to insert a new keyframe is that it must bring enough new information with respect to the others keyframes already present in the graph. To make the tracking more robust to fast camera movements (such as rotation) and rapid scenario changes, this procedure is not too tight, allowing a pretty fast insertion of keyframes and leaving the job of removing the redundant ones to the local mapping module. With respect to the original PTAM procedure, which used a distance criterion to insert new keyframes, the set of conditions imposed by the ORB-SLAM algorithm (and their evolution in ORB-SLAM2 [2]) are less restrictive and more functional, increasing the tracking efficiency. Given the tight coupling between the tracking and

the graph structure, several of the conditions imposed refer to the reference keyframe of the actual frame, which is basically the one with the most points in common.
Given the specific formulation for stereo cameras of the ORB-SLAM2, a new distinction between far and close points is reported (as previously stated in the section 2.1.3), that allows the insertion of additional conditions to guarantee a sufficient amount of close points tracked. So essentially, to insert new keyframes, all the following conditions must be met:

- No loop closure procedures must be running, to avoid keyframes insertion during the graph update.
- At least a fixed number of frames (usually determined by the fps frequency of the camera) must have passed from the last global relocalization, to ensure a good relocalization.
- The current frame observes less than 75% of the points observed by the reference keyframe or a sufficient amount of new close points could be inserted. In any case, more than 15 points must be tracked.
- To allow good interfacing with the local mapping module, if it is not idle, new keyframes can be inserted only if a sufficient number of keyframes have passed from the last keyframe insertion or if the tracking is weak (insufficient amount of close points or less than 25% of the points tracked with respect to the reference keyframe).

The 3rd condition allows a good tracking, while the 4th one is kept to avoid the that local mapping could impact negatively the tracking. The condition on the close points is structured in a way that new keyframes are inserted if not sufficient close points are found between the ones already tracked in the map, but a substantial new amount of them could be created. This is done since only close points guarantee strong position tracking (while the far ones are good mainly for orientation).
In parallel, in the local mapping module, keyframes are culled by performing, each time that a new keyframe is inserted, a search in the latter covisibility neighbor, by eliminating the keyframes whose 90% of the tracked map points are seen in at least others three keyframes, in the same or finer scale (according to the ORB scale definition). This procedure permits a compact structure of the graph, keeping at the same time only the map points observed with the highest level of precision.
Of course, to avoid conflicts with the loop closing module, the elimination of each keyframe that is being managed by a loop detection or correction mechanism is delayed until its completion. In the case of keyframes connected

by a loop edge, this culling is postponed indefinitely because the information provided by the link is stronger with respect to the benefits of avoiding keyframes' redundancy.

3.2.3 Map points

Not only the robot's poses, but also the map points' positions are quantities that must be kept in consideration when constructing a graph-based SLAM. For this reason, each keyframe has a kind of link with the map points tracked (even if they do not appear directly as nodes): basically, each keyframe memorizes the map point tracked by it, associating each one of them to the observed 2D features. In this way, additional information as descriptors and observed scale are directly linked to the interested map point.

To allow a fast recall of the keyframes that observe a map point, also each one of them keeps in a container the information regarding the observing keyframes, by creating a doubly linked connection between them.

As explained in the sections 2.1.2 and 2.1.3, map points are created by triangulating stereo observations of close points, while the further ones are reconstructed by combining multiple observations of more keyframes, enforcing epipolar constraints for points matching.

To avoid duplication of the same points a larger search is performed after new points creation from the new keyframe (extending the searching area also to keyframes not covisible with the current one), and a fusion procedure is then adopted (by relating the 3D poses of the points with the feature extracted from the keyframes involved) by merging the duplicated points into one, modifying in this way the keyframes observations. Of course, this fusing usually leads to the creation of more connections with other keyframes.

Since new map points are inserted in a pretty fast manner, a culling procedure is required to settle the landmarks' validity (since incorrect triangulation could arise from spurious data association). The restrictive test is performed for every map point during the first three keyframes insertion after its creation: during this period, to be retained in the map, a point must be found from the tracking in at least the 25% of the frames in which is predicted to be visible. Another condition that is instead kept during the whole point's life is that it must be observed from at least three keyframes. If one of these two conditions is violated, map points are culled.

This procedure, despite its restrictiveness, allows a compact (by keeping only points that are useful for tracking) and robust (by rapidly discarding outliers) sparse map representation of the environment.

3.3 Graph optimization

As stated before, the second main part of a graph-based SLAM is the graph optimization, which has the job of finding the most likely values and disposition for nodes and edges given the observation data.
An efficient graph optimization is mandatory since it will simplify also the graph construction, by allowing the introduction of new edges and nodes on an already quasi-optimal graph, reducing the randomness of the new entries and making the optimization itself easier. This tight coupling between the two parts requires an efficient and reliable graph optimization, that should not be too computationally demanding to not impact too much the tracking performance but also reliable enough to still achieve a good accuracy.
In the ORB-SLAM algorithm, more precisely, more than one type of optimization (regarding the graph and its nodes and connections) is performed, depending on the modules in which they run and the targets whom they aim.

3.3.1 Motion-only bundle adjustment

The basic optimization procedure performed is the one regarding the current frame, namely the current position of the robot. This type of optimization actually doesn't involve directly the graph, but its main purpose is to estimate the current frame pose, given map points matches. Naming $\mathcal{X}$ the set of matches between the tracked map points and the keypoints detected, the reference optimization problem is:

$$\{R_{cw}, \boldsymbol{t}_{cw}\} = \underset{R_{cw}, \boldsymbol{t}_{cw}}{\arg\min} \sum_{i \in \mathcal{X}} \rho \left(||\boldsymbol{x}_i - \pi(R_{cw}\boldsymbol{X}_i + \boldsymbol{t}_{cw})||^2_{\Sigma_i} \right) \tag{3.4}$$

where the quantities R_{cw} and $\boldsymbol{t}_{cw}$ represent the transformation linking the world coordinate system to the current camera's one, $\boldsymbol{X}$ is the position of the map point in the world frame, while $\boldsymbol{x}$ contains the coordinates of the matched keypoint in the image (for a stereo observation $\boldsymbol{x} = [u_L, v_L, u_R]^T$, while for a monocular one $\boldsymbol{x} = [u_L, v_L]^T$). The transformation $\pi(\cdot)$ is the projection from the points in the camera coordinate frame into the keypoints in the image plane: having two types of observations, stereo for tracked close points and monocular for the far ones, this transformation either assume the form (2.13) or (2.2), depending on the type of points involved. These two types of projection are both needed since not all the keypoints tracked have a corresponding stereo match.
The weighting matrix Σ is due to the uncertainties in the observed keypoints coordinate, and is directly linked to the scale at which the interesting keypoint has been observed:

$$\Sigma_i = \begin{bmatrix} \sigma_i & 0 & 0 \\ 0 & \sigma_i & 0 \\ 0 & 0 & \sigma_i \end{bmatrix}, \qquad \sigma_i = \frac{1}{s^{2n_i}}, \tag{3.5}$$

where s is the scale factor associated to the multi-scale pyramid of the ORB extraction, and n_i is the level at which the keypoint has been observed. Recalling the definition of the MAP estimate (3.2), the matrix Σ is basically the inverse of the covariance of the residual.
To increase the robustness of the optimization to outliers (that could originate from spurious data association), each term of the (3.4) is subjected to the robust Huber loss function:

$$\rho_\delta(a^2) = \begin{cases} \frac{1}{2}a^2 & , |a| \leq \delta \\ \delta\left(|a| - \frac{\delta}{2}\right) & |a| > \delta \end{cases}, \tag{3.6}$$

where δ is a parameter that must be chosen depending on the type of problem. This loss function increases the robustness of the optimization, reducing the impact on the overall cost of the residuals that fall too far with respect to the other ones.
Every time the (3.4) is involved, four subsequent optimizations are actually performed: after every optimization a χ^2-test is performed on the final residuals obtained, and all the points classified as outliers are excluded from the next optimization, in order to further increase the robustness of the solution.

3.3.2 Local bundle adjustment

The local bundle adjustment is run instead in the local mapping module and has the job of optimizing the poses and the map points regarding the last keyframe inserted, along with its covisibility neighbor.
After having processed a new keyframe (created its connections, triangulated new map points and fused the cloned ones) the local mapping launches an optimization in order to assess the local graph structure. Using covisibility information, the modules extract all the keyframes that are connected to the current one in the actual graph, proceeding then to retrieve all the map points that are seen from this group. Finally, it retrieves also every other keyframe that observes the map points considered. The poses of this last group of keyframes, however, don't take an active part in the optimization, since they are set as fixed, but they are only used to bring additional information for the optimization of the map points.
By defining the set of local keyframes as $\mathcal{K}_L$, the set of fixed ones as $\mathcal{K}_F$, the set of observed points as $\mathcal{P}_L$ and the set of matches between the map points and the keypoints in a certain keyframe k as $\mathcal{X}_k$, the following optimization problem can be written:

$$\{\boldsymbol{X}_i, R_{lw}, \boldsymbol{t}_{lw} | i \in \mathcal{P}_L, l \in \mathcal{K}_L\} = \underset{\boldsymbol{X}_i, R_{lw}, \boldsymbol{t}_{lw}}{\arg\min} \sum_{k \in \mathcal{K}_L \cup \mathcal{K}_F} \sum_{j \in \mathcal{X}_k} \rho(E_{kj}) \tag{3.7}$$

where E_{kj} is the standard reprojection error

$$E_{kj} = ||\boldsymbol{x}_j - \pi(R_{kw}\boldsymbol{X}_j + \boldsymbol{t}_{kw})||^2_{\Sigma_{kj}} . \tag{3.8}$$

The quantities R_{kw} and $\boldsymbol{t}_{kw}$ represent, of course, the transformations that express the world coordinate system into the keyframe k reference frame. Also in this case the robust Huber cost function ρ is applied and the matrix Σ_{kj} is calculated in the same way as in the (3.5), depending in this case both on the map point and the observing keyframe.
Also in this case an additional policy for outlier removal is applied and, more precisely, two sequential optimization are performed: the outlier detected from the first one are removed for the second one. All the points classified as outliers are then removed from all the keyframes involved. Naturally, given the necessity of high interactivity with the tracking module, the optimization can be stopped at any time if a new keyframe is inserted, according to the policy presented in the section 3.2.2.

3.3.3 Essential graph optimization

After having detected a loop (by mean, a place that has already been visited before) a new type of connection, called *loop edge* can be established between the two keyframes involved.
This new edge brings with itself the relative position between the two keyframes and, once established, it can be used to propagate the correction to the remaining part of the graph, in order to reduce the accumulated drift due to uncertainties and errors in the measurements.
To reduce the computational burden of the procedure, the optimization is performed on the *essential graph*. To be sure of not having isolated keyframes, also parent's links and previously detected loop edges are added in the resulting graph.
This type of optimization doesn't involve directly the observed map points, since it is performed only on the keyframe poses using as constraint precisely the relative poses between them. Since several of these have been corrected by the loop detection, this optimization basically propagates this correction over the whole graph, by refining the keyframes' poses.
By naming $\mathcal{G}$ as the set of all the keyframes in the graph and $\mathcal{E}$ as the set of the edges present in the essential graph, the optimization problem involved can be formulated as:

$$\{T_{iw} | i \in \mathcal{G}\} = \underset{T_{iw}}{\arg\min} \sum_{j,k \in \mathcal{E}} \left(e_{jk}^T \Lambda_{jk} e_{jk} \right) \tag{3.9}$$

where

$$e_{jk} = \log_{SE(3)} \left(T_{jk} T_{kw} T_{jw}^{-1} \right), \quad e_{jk} \in \mathbb{R}^6. \tag{3.10}$$

Since the homogeneous transformation are a redundant representation of pose in space, the error term (3.10) is projected into the $SE(3)$ tangent space by the logarithmic transformation $log_{SE(3)}$, to obtain a vector in $\mathbb{R}^6$.
This projection involves mainly rotations since they are usually better represented using redundant parameterization, such as rotation matrices in $\mathbb{R}^{3\times3}$ or quaternions in $\mathbb{R}^4$, to avoid singularities. In the optic of a minimal parameterization for the optimization procedure, a projection called *logarithmic map* into the tangent space of the manifold considered is performed[1].
The matrix $\Lambda_{jk} \in \mathbb{R}^{6\times6}$ is chosen as an identity because additional information on the uncertainties of the edges is not present, so no residual of the (3.9) must have a higher or lower weighting than the others.

3.3.4 Global bundle adjustment

The local bundle adjustment (3.7) is designed to optimize only a local portion of the graph: this is done to speed up the optimization but causes other parts of the graphs to be misaligned with the updated one, resulting in a decreased accuracy when revisiting old areas. As a compromise between obtaining the optimal solution and a small overhead on the computation, a global bundle adjustment is performed, but only after an essential graph optimization. To further reduce the computational request, this procedure is run in another thread so that the system can continue to detect loops and create the map, to allow the insertion of new keyframes and map points during the global bundle adjustment.
The general structure of the optimization problem recalls the one from the (3.7), more precisely

$$\{\boldsymbol{X}_i, R_{kw}, \boldsymbol{t}_{kw} | i \in \mathcal{P}, k \in \mathcal{G}\} = \underset{\boldsymbol{X}_i, R_{kw}, \boldsymbol{t}_{kw}}{\arg\min} \sum_{k \in \mathcal{G}} \sum_{j \in \mathcal{X}_k} \rho(E_{kj}) \tag{3.11}$$

where this time all the keyframes belonging to the graph $\mathcal{G}$ are involved, as well as the map points belonging to the global map $\mathcal{P}$; $\mathcal{X}_k$ is always the set of tracked map points for each keyframe and E_{kj} is exactly the same as the (3.8). The only fixed keyframe in this optimization is the one representing the

[1]A wider explanation of this mathematical relationship is reported in the appendix A

origin (introduced in the initialization of the algorithm) and is mandatory to have it in order to eliminate the gauge freedom.
Since the global BA is asynchronous with respect to the loop closing and the local mapping modules, there is the need to define a procedure to merge the output of the global optimization with the current state of the graph, since new keyframes and map points could be present. Additionally, if a new loop is detected while the global BA is running, the latter is aborted immediately and restarted once the loop closure procedure has finished, in order to perform the global optimization on the latest updated graph available.
Once the global bundle adjustment has finished, the corrections are propagated through the whole graph, using the spanning tree: starting from the first keyframe created, the correction is propagated to the children until completion. If a keyframe has been involved in the optimization, its corrected pose is already available and so this is directly used. For all the other keyframes not processed (so that were created during the global bundle adjustment execution) a special procedure is used: each child pose is updated from the parent pose, using its relative position with it.
More precisely, by naming T_{pw}^{new} and T_{pw}^{old} the corrected and the old poses of the parent and as T_{cw}^{old} the uncorrected pose of the child, the new child's pose can be computed as:

$$T_{cw}^{new} = T_{cp} T_{pw}^{new}, \qquad T_{cp} = T_{cw}^{old} \left(T_{pw}^{old} \right)^{-1} \tag{3.12}$$

where as always the subscripts on T_{ij} state that the transformation express the coordinate system of j into the reference system of i.
A similar procedure is run for the map points: the ones processed by the global optimization are directly updated, while the other ones that were added later are updated using the relative position with respect to their reference keyframe (basically, the keyframe from which they were first observed). By naming $\boldsymbol{X}_w$ the position of the map point in the world coordinate system and $\boldsymbol{X}_c$ the position of the map point in the reference keyframe coordinate system (that, given its origin from the observation, it remain the same during all his life), this procedure can be summarized as follow:

$$\boldsymbol{X}_w^{new} = T_{wc}^{new} \boldsymbol{X}_c, \qquad \boldsymbol{X}_c = T_{cw}^{old} \boldsymbol{X}_w^{old}. \tag{3.13}$$

Chapter 4

Place recognition

As stated in the chapter 1, one of the most important features of a robust SLAM algorithm is an efficient method to perform place recognition. This is mandatory in both relocalization frameworks and for loop detection: in the first one, place recognition is used to relocalize the robot when the default tracking algorithm falls (due for example to motion blur, sudden motions or severe ambient occlusions), while in the latter is used to recognize places visited before, in order to correct the drift originated from error accumulation.
In visual SLAM two main types of techniques for performing place recognition exist: map-to-image and image-to-image. The first type consists, as the name says, in finding a match between the actual image seen and the points already present in a memorized map. This is pretty efficient in the case of small environments, but in the case of larger ones this method could lead to an increase in computational burden (caused by the need to perform a search over a bigger set of map points). For this reasons image-to-image methods, that match the actually observed image with information directly extracted from the already seen ones, scale much better.
The idea behind image-to-image methods is to rely on a database built from the images collected by the robot during the exploration so that, when a new image is captured, all the similar ones can be retrieved and compared. A common way to build this type of database is to use the Bag of Words (BoW) model. Having this in mind, the ORB-SLAM's authors opted for the use of the DBoW mechanism, [14], in order to construct the database.

4.1 DBoW

Given the goal of constructing an image database, one of the best ways of doing it is to memorize a set of features associated with each image, instead of the image itself. The chosen features extractor is an ORB one, illustrated in chapter 2, since it's both fast and reliable.

The idea of a BoW technique originated from document classification, where the purpose was to classify a piece of text by counting how many times each word would appear on it, without considering grammar laws or words' order. To reduce the impact of commonly used words, the frequencies of the words counted are normalized using *Term Frequency-Inverse Document Frequency*, TF-IDF, as measurement. This method provides a useful way to adjust the words' frequency, by taking into account not only how many times a word appears in a document with respect to every other word, but also how many times it appears in all the other documents, by so reducing the weight of the commonly used ones.
By adapting the technique of the BoW to the features extracted from an image, it's therefore possible to obtain a set of elements with which to construct a database. To efficiently organize it, a vocabulary tree is used, [15], that uses a direct and an inverse index to speed up the database access.

4.1.1 Vocabulary tree

Inheriting the definition from document classification, a vocabulary must be constructed in order to apply the BoW classification idea. The main advantage of using a vocabulary is that it can be constructed offline from visual descriptors obtained from an image dataset. This is done by dividing the hyperspace generated by all the ORB descriptors extracted into smaller and smaller subsets until a *word* is obtained, that is in practice only a set of descriptors very similar to each other. By doing that, each descriptor extracted from any other image can be associated with a word in the vocabulary, allowing the construct of the BoW for the image interested.
To construct the vocabulary a partition is basically performed. The sets of all the ORB descriptors describe a binary space with high dimensionality so the idea is to recursively split this high dimension space into smaller chunks and, for doing that, a tree structure is used: to construct this tree only two parameters are needed, k and L, that defines the number of children of each node and the maximum level of deepness reached.
Starting from the raw descriptor space, it is partitioned into k clusters by a k-means process, that basically joins together the descriptors that are more similar to each other. Formally, this goal can be defined as an optimization problem: by defining $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k\}$ the set of clusters that must be found, each one defined by its mean

$$\boldsymbol{\mu}_i = \frac{1}{\#\mathcal{S}_i} \sum_{\boldsymbol{x} \in \mathcal{S}_i} \boldsymbol{x} \tag{4.1}$$

where $\#\mathcal{S}_i$ is simply the cardinality of the set considered, the general form of the problem becomes:

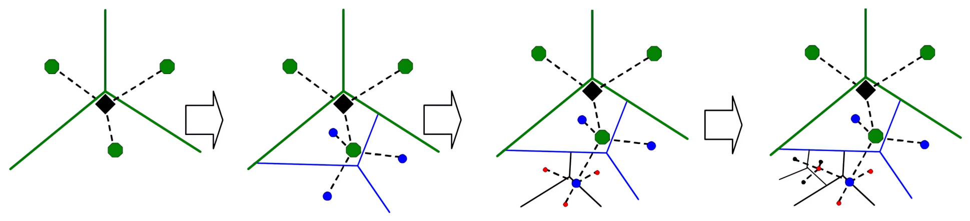


Figure 4.1: *Illustration of the process. At each level the parent cluster is divided into k sub-clusters (in this example $k = 3$).*

$$\arg\min_{\mathcal{S}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in \mathcal{S}_i} ||\boldsymbol{x} - \boldsymbol{\mu}_i||^2 . \tag{4.2}$$

An illustration of this procedure is depicted in the figure 4.1.
The problem can however be pretty demanding from the computational point of view (depending of course on the number of descriptors considered), but since all its computations are performed offline, this does not bring any disadvantage.
Each one of this cluster is then partitioned again in other k clusters, using always the same technique (4.2), recursively performing this process till an L number of subdivisions have been performed.
The process is therefore equivalent to a tree construction, by assigning to each node the clusters generated from its partition as children. The tree created in this way directly defines the visual vocabulary (being the leaf nodes the designed *words*) and also an efficient search procedure. In fact, to find the word assigned to a specified descriptor during the online phase, it is sufficient to simply compare the interested descriptor to the k candidate cluster centers (each of whom is associated with one of the k children) for each level until the final level, and so the designed word, is obtained. Since in this case the descriptors used are binary vectors, the comparison can be done really fast by simply performing an *xor* operation between all the elements of two vectors, to obtain the Hamming distance.
By performing this search for the descriptors of all the features extracted from an image, and by so finding a *word* for each one of them, the BoW of the specified image can finally be computed.

4.1.2 Scoring

Given two BoW vectors, computed as said before, a way to measure their similarities must be provided, to be able to match a query image with the ones in the database.

First, to each tree's leaf node (namely to each *word*) a weight is assigned, following the TF-IDF measure. More precisely, the weight of each word in a BoW of an image t, following the idea expressed in [16], can be calculated as:

$$v_i^t = \frac{n_{it}}{n_t} \ln \frac{N}{n_i} \tag{4.3}$$

where n_{it} is the number of occurrences of the word i in the image t, n_t is the total number of words in the image considered, N is the total number of images in the database and n_i is the number of images with at least one descriptor vector through the node i. The first term of the (4.3), as it can be seen, depends only on the image interested (that could already be in the database or be the one queried), while the second depends only on the whole database. So, as the name said, the weight of each word is a mix between its frequency in a given document, for the contribution of the first term, and its frequency in the whole database, thanks to the second term.
Given two images, each one with their BoW vectors $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$, whose elements are calculated according to the (4.3), the similarity score between two images can be calculated as:

$$s(\boldsymbol{v}_i, \boldsymbol{v}_j) = 1 - \frac{1}{2} \left| \frac{\boldsymbol{v}_i}{|\boldsymbol{v}_i|} - \frac{\boldsymbol{v}_j}{|\boldsymbol{v}_j|} \right| . \tag{4.4}$$

The values of this score (where a L_1-norm is used, since it gives better result with respect to the classic L_2-norm), lies in $[0, 1]$, being the closer to one the more similar the two vectors are.

4.1.3 Direct and inverse index

As often done for BoW databases, an inverse index is maintained along with the bag of words: each word w_i in the vocabulary stores a list of the images I_t where it is present. This allows a very fast database query, by easily retrieving all the images with some words in common, to easily perform comparisons between them. To speed up the efficiency, the inverse index is augmented by storing pairs $\langle I_t, v_t^i \rangle$, in order to quickly retrieve also the weight of the referenced words in the image pointed. The inverse index structure is updated, of course, every time a new image is added to the database.
In addition, a direct index is also used to better store the features of each image: the idea in this case is to keep, for each image, the index of the words that belong to it. More precisely, the quantities stored are the ancestor nodes of the words present in the image I_t, as well as the list of local features associated with each node. This permits to speed up the geometrical verification of two observed images by computing correspondences only between the features that belong to the same words, or with common ancestors (by mean,

parent nodes) at a certain level. The utility of that is exploited, for example, in the tracking module when correspondences with the reference keyframe of the currently processed frame must be found (since a reliable relative pose between them is not available) or also for speeding up the matches' search between two keyframes. As for the inverse index structure, also this one is updated every time a new image is added to the database.

4.2 Loop detection

In the original work [14], place recognition was performed by searching the image database to find the best candidates, and temporal and geometric consistency validations were then performed, in order to filter out the candidates retrieved and find the best one.

Using a graph-based SLAM, these steps are still performed but in a different way, since covisibility information can be used to speed up the various steps.

4.2.1 Candidates selection

First of all, when a new keyframe arrives, the similarity scores between its BoW and its neighbor's ones in the covisibility graph are computed, and the minimum among them, $s_{\min}$, is then searched. This score is then used as lower bound for the database query, by discarding all the memorized keyframes having a score lower than the minimum obtained. This technique allows to obtain more robust candidates to match, by essentially performing a normalization of the last keyframe's score along its covisibility neighbor since it depends deeply on the query image and its distribution of BoW.

To discard candidates that are too close in time to the actual observation, all the keyframes in the covisibility graph that are connected to the last keyframe inserted (the one to match) are discarded. In addition, to augment the robustness of the retrieved candidates, each one of them must be consistent with previously detected candidates. More precisely, a candidate is considered consistent with another one if they share at least a keyframe in their covisibility neighbor. After all these steps, finally, a candidate is marked as good if at least three previous consecutive consistent candidates have been detected.

Naturally, more than one candidate could be retrieved for a single keyframe (due to places with similar appearance). So it's mandatory to find the one that best matches with the current keyframe, and to discard the false positive.

4.2.2 Geometric consistency

Once that the candidates are retrieved, additional tests must be performed to discard the bad ones. First of all a direct check on the ORB features observed by them and the current keyframe is performed, by using the direct index of the database to compute the matches. Only if a sufficient amount of correspondence is found the candidate keyframe can be kept and continue the tests.
After the BoW matching of the previous step, a set of map point correspondences is available between each successful candidate and the current keyframe. By having therefore 3D-to-3D correspondences for each candidate, relative poses between the current keyframe and each one of them can be computed, and it can be used to further refine the matching search. The relative pose calculus is carried on in various steps, in order to cancel out the outliers and the wrong matches.
First, a series of RANSAC iterations is performed between the current keyframe and each candidate, to find out a suitable $SE(3)$ pose (if it exists) among them. This first search is performed using the Horn method [17], which requires a minimal number of points in order to compute a solution. If the search is successful, further correspondence searching between the current keyframe and the proposed candidate is performed, taking advantage of the new estimated relative position. If enough matches are found, a final optimization procedure is performed (using all the matched map points) to find the optimal $SE(3)$ pose connecting the two keyframes. If this one is supported by enough inliers then the candidate is marked as good and a loop closure procedure, as reported in 3.3.3, can be performed. This search is carried out among all the loop candidates, till one is successful or all fail.
Now an exhaustive explanation of the RANSAC iterations scheme, the Horn method and the $SE(3)$ optimization is presented.

RANSAC Random sample consensus (RANSAC) is an iterative method, introduced in [18] by Fischer and Bolles, with the goal of estimating the parameters of a mathematical model given a set of measured data containing outliers.
Standard fitting techniques, such as for example linear least squares, are heavily influenced by outliers in the model. Some expedients as loss functions can help to reduce their impact, but nevertheless the solution obtained will be degraded with respect to the optimal one. RANSAC, however, was designed precisely to overcome this issue at the cost of resulting in a nondeterministic algorithm, producing a good result only with a certain probability, being dependent on the number of iterations performed.
The RANSAC algorithm is based on two main assumptions: there are few outliers with respect to the good data (otherwise it would be impossible to set a

criterion for distinction) and that enough inliers to produce a good model are present. The main difference of the RANSAC algorithm with respect to other fitting techniques is that it doesn't resort to the use of all the data available, but it tries to find a model with the minimal set of data that is possible to use. Having stated that, the RANSAC algorithm can be seen as an iterative execution of the following steps:

- A minimal subset from the set of data is randomly selected. This set must contain the minimal number of data needed in order to produce a good estimate of the desired model and can be seen as a guess of the inliers of the problem.
- The set of data extracted is used to fit a model. The specific calculus performed on this step depends of course on the type of model to fit.
- All the data in the original dataset are tested against the obtained model, using some criteria or loss function that varies with the type of model itself. This check will result in a subdivision of the original dataset into inliers and outliers, depending on the model found.
- If the number of inliers found is sufficiently high, then the model is marked as good and the algorithm can stop, otherwise it will start all over again, by still keeping the best model found till the actual iteration in memory.

The most crucial part of a correct RANSAC procedure is the proper choice for the algorithm's three parameters: the threshold to classify a point as an outlier or inlier given the model, the minimal number of good points to be found in order to mark the model found as correct and the number of iteration performed. By leaving the decision of the first two to the specific model involved in the procedure, it can be proven that the latter one (the number of iterations) is directly dependent on the goodness of the result achieved.
The key to proving that is to see the successful model as the one obtained when only inliers are selected during the first step. Having stated this, the probability ε of choosing an inlier every time a point is selected can be formulated as:

$$\varepsilon = \frac{n_{\text{min}}}{N}, \tag{4.5}$$

where n_{min} is the minimal number of data required to properly estimate the model and N is the total number of data available. Actually, the numerator should match the number of actual inliers present in the set of data, but since this quantity is not available, a solution is to put this value equal to the minimal number of inliers required by the algorithm to be successful, so using one of the parameters provided.

After this introduction, another assumption is made in order to obtain the result: each point must be selected independently from the others. This assumption in reality is not verified, since each time a data is selected it must be removed from the set of available ones for the next extraction (this must be done in order to not select the same data twice). However, this approximation becomes less impacting the bigger the number of data available. Having stated that, it's easy to state that the probability that all the n_{min} data chosen are inliers is $\varepsilon^{n_{\text{min}}}$, while $1 - \varepsilon^{n_{\text{min}}}$ is the probability that at least one of this chosen data is an outlier (resulting so in an incorrect model estimation). By naming p the probability that the algorithm produces a good result, which became the parameter to choose instead of the number of iterations k, it's easy to obtain the following relationship:

$$1 - p = (1 - \varepsilon^{n_{\text{min}}})^k, \tag{4.6}$$

which expresses the probability that the algorithm would never produce a good result (by never selecting n_{min} good points) during all the k iteration. Now, from this relationship, the minimal number of iterations required to obtain a good result with a probability equal to ε can finally be calculated as:

$$k = \frac{\log(1 - p)}{\log(1 - \varepsilon^{n_{\text{min}}})}. \tag{4.7}$$

Absolute orientation The problem of estimating the relative pose between two systems that observe a common set of 3D points is referred to in the literature as *absolute orientation*, and a closed-form solution to this problem was provided by Horn in [17].
The original method was designed for points with monocular observations, the most general case, so the scale was considered an unknown variable. Having stated that, the original problem has therefore a total of 7 DoF, so at least 3 points, called *control points*, are needed (providing 9 independent constraints) in order to find the solution.
Given the noisy nature of the observations, the map points would never match perfectly the captured features in the two images, so the solution is found by optimizing the sum of squares of residual errors, given by:

$$\hat{R}, \hat{\boldsymbol{t}} = \arg\min_{R,\boldsymbol{t}} \sum_{i=1}^{N} ||\boldsymbol{r}_{1,i} - R\boldsymbol{r}_{2,i} - \boldsymbol{t}||^2 \tag{4.8}$$

where the vectors $\boldsymbol{r}_{1,i}$ and $\boldsymbol{r}_{2,i}$ are the coordinates of the N observed points seen by the two keyframes reference systems. As previously stated, in the original formulation also a scale λ was involved in the procedure (multiplying the matrix R), but since in the treated case the scale is observable (given the stereo constraints), λ can be fixed to one and removed from the treatment.

Since the mutual orientation doesn't depend on the translation, the problem can be decoupled by first finding the optimal rotation $\hat{R}$ and then using it to compute the optimal translation $\hat{\boldsymbol{t}}$.
The ideas proposed by Horn are actually three: to use relative coordinate (referred to two virtual points called *centroids*) in the (4.8), to split the rotation solution from the translation one (since orientation doesn't depend on translation) and to resort using quaternions as a rotation representation, to reduce the number of variables and constraint involved in the optimization. So by defining, as done in the paper, the two centroids (calculated in the two coordinate system) as:

$$\overline{\boldsymbol{r}}_1 = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{r}_{1,i}, \quad \overline{\boldsymbol{r}}_2 = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{r}_{2,i}. \tag{4.9}$$

By then expressing the points coordinates with respect to the two centroids as $\boldsymbol{r}'_{j,i} = \boldsymbol{r}_{j,i} - \overline{\boldsymbol{r}}_j$, the error term in the (4.8) can be rewritten as:

$$\boldsymbol{e}_i = \boldsymbol{r}'_{2,i} - R(\boldsymbol{r}_{1,i}) - \boldsymbol{t}' \tag{4.10}$$

where $R(\cdot)$ is a notation to express any type of rotation, by making so the formulation more general, and

$$\boldsymbol{t}' = \boldsymbol{t} - \overline{\boldsymbol{r}}_2 + R(\overline{\boldsymbol{r}}_1). \tag{4.11}$$

By then putting the new error term inside the (4.8), the problem can be rewritten, performing the calculus (since the norm of a vector can be seen as the dot product of the vector with itself), as:

$$\min_{R,\boldsymbol{t}} \sum_{i=1}^{N} \left|\left|\boldsymbol{r}'_{2,i} - R(\boldsymbol{r}'_{1,i})\right|\right|^2 - 2\boldsymbol{t}' \cdot \sum_{i=1}^{N} l\left[\boldsymbol{r}'_{2,i} - R(\boldsymbol{r}'_{1,i})\right] + N\left|\left|\boldsymbol{t}'\right|\right|. \tag{4.12}$$

By recalling that $\sum_{i=1}^{N} \boldsymbol{r}_{j,i} = 0$ for construction, the second term is obviously equal to zero. The first and last terms show instead the independence of the orientation part from the translation.
By carrying the calculus on the first term, it can be found that it is the only one term depending on the rotation, and so the problem is reduced to

$$\min_{R} - \sum_{i=1}^{N} \boldsymbol{r}'_{2,i} \cdot R(\boldsymbol{r}'_{1,i}) \tag{4.13}$$

while the difference of the two sums depending only on $||\boldsymbol{r}'_{j,i}||$ is as close to zero as the less is the mismatch between the two frame's measurements (so is constant in the optimization).
Now, by using the quaternion q and its property to express the rotation $R(\cdot)$, the problem (4.13) translates in:

$$\hat{q} = \arg\max_q \sum_{i=1}^{N} \left(q\boldsymbol{r}'_{1,i}q^*\right) \cdot \boldsymbol{r}'_{2,i} = \arg\max_q \sum_{i=1}^{N} \left(q\boldsymbol{r}'_{1,i}\right) \cdot \left(\boldsymbol{r}'_{2,i}q\right). \tag{4.14}$$

It can be proven, by using the property of quaternion multiplication, that the last expression can be written as:

$$\hat{q} = \arg\max_q \sum_{i=1}^{N} q\mathcal{N}_i q = \arg\max_q q\left(\sum_{i=1}^{N}\mathcal{N}_i\right) q = \arg\max_q q\mathcal{N}q \tag{4.15}$$

where $\mathcal{N}$ is a symmetric matrix constructed as

$$\mathcal{N} = \begin{bmatrix} M_{11}+M_{22}+M_{33} & M_{23}-M_{32} & M_{31}-M_{13} & M_{12}-M_{21} \\ M_{23}-M_{32} & M_{11}-M_{22}-M_{33} & M_{12}+M_{21} & M_{31}+M_{13} \\ M_{31}-M_{13} & M_{12}+M_{21} & -M_{11}+M_{22}-M_{33} & M_{23}+M_{32} \\ M_{12}-M_{21} & M_{31}+M_{13} & M_{23}+M_{32} & -M_{11}-M_{22}+M_{33} \end{bmatrix} \tag{4.16}$$

where the terms M_{ij} are the terms of the M matrix

$$M = \sum_{i=1}^{N} \boldsymbol{r}'_{1,i}\boldsymbol{r}'^{T}_{2,i}. \tag{4.17}$$

Having stated that, the optimal rotation can be found using linear algebra: the quaternion that maximizes the (4.15) is the one having the same direction of the eigenvector of the matrix (4.16) associated with its highest eigenvalue. Finally, once the rotation is found, the optimal translation can be found by minimizing the last term of the (4.12), which happens obviously when $\boldsymbol{t}'$ is zero. Remembering the (4.11), the optimal translation can then be easily found as:

$$\hat{\boldsymbol{t}} = \overline{\boldsymbol{r}}_2 - \hat{R}\overline{\boldsymbol{r}}_1, \tag{4.18}$$

where $\hat{R}$ is the optimal rotation matrix obtained by the optimal quaternion $\hat{q}$. In conclusion, the Horn's method for solving the absolute orientation problem is fast and efficient, providing in a single step both the optimal translation and rotation, allowing its implementation on a RANSAC procedure, explained before, to augment its robustness to outliers and errors.

SE(3) optimization As reported earlier, after having an available initial estimate and enough matches between the two observing keyframes, an optimization procedure is performed. The enforce the constraint on the two keyframes, the problem is structured so that double linking constraints are present between the two keyframes.

Given the set $\mathcal{X}$ of N map points to match, in common for both the keyframes involved, the following quantities can be introduced: $\boldsymbol{X}_{i,j}$ are the coordinates of the map point i in the keyframe j; $\boldsymbol{x}_{i,j}$ are instead the coordinates of the observed keypoint, in the image plane belonging to the keyframe i, associated to the map point j; finally, T_{ij} is the homogeneous transformation that transform points expressed in the reference system j into the i.
Having stated that, two types of errors are defined:

$$\begin{aligned} \boldsymbol{e}_{1,i} &= \boldsymbol{x}_{1,i} - \pi_1\left(T_{12}\boldsymbol{X}_{2,i}\right) \\ \boldsymbol{e}_{2,i} &= \boldsymbol{x}_{2,i} - \pi_2\left(T_{12}^{-1}\boldsymbol{X}_{1,i}\right) \end{aligned} \tag{4.19}$$

where $\pi(\cdot)$, in this case, represents only the projection law (2.2), being a monocular observation with a fixed scale sufficient for finding an optimal solution. The general optimization problem can then be written as:

$$\hat{T}_{12} = \underset{T_{12}}{\arg\min} \sum_{i=1}^{N} \left(\rho(\boldsymbol{e}_{1,i}^T \Sigma_{1,i} \boldsymbol{e}_{1,i}) + \rho(\boldsymbol{e}_{2,i}^T \Sigma_{2,i} \boldsymbol{e}_{2,i})\right) \tag{4.20}$$

where the matrix $\Sigma_{j,i}$ is the inverse of the covariance matrix of the measurements of the points i in the keyframe j, calculated in the same way as in (3.5), and ρ is as always the robust Huber loss function (3.6).
To further increase the robustness of the optimization, even this one is performed twice, by eliminating outliers with a χ^2-test between the two ones.

4.2.3 Loop correction

After having finally confirmed a loop, the new information must be used to correct the accumulated drift error and propagate it through the whole graph. This is done by using the relative pose obtained by solving the (4.20), and by taking as granted the pose of the keyframe detected as a loop closure. The correction procedure is now illustrated.
First of all, the current frame pose is updated, using:

$$T_{cw} = T_{cl} T_{lw} \tag{4.21}$$

where the l subscript is referred to the loop keyframe and so, naturally, T_{cl} is simply the solution obtained by the (4.20). After that, the new information is propagated through the whole covisibility neighbor of the current keyframe. Being then T_{wc}^{old} the uncorrected transformation that links the current keyframe to the world, T_{cw}^{new} the corrected one and T_{iw}^{old} the transformation between the world and the i-neighbor keyframe, the new corrected pose T_{iw}^{new} is computed as:

$$T_{iw}^{new} = (T_{iw}^{old} T_{wc}^{old}) T_{cw}^{new} = T_{ic}^{old} T_{cw}^{new} \tag{4.22}$$

where T_{ic}^{old} is the non-corrected relative pose between the current keyframe and the neighbor considered.
After that, also the poses of all the map points observed by this group are corrected by first projecting the point in the observing keyframe with the non-corrected pose (since this position never changes, depending only on the detected feature), and then re-projecting the point back in the world using the corrected pose:

$$\boldsymbol{p}_i^{new} = T_{wi}^{new} T_{iw}^{old} \boldsymbol{p}_i^{old} \tag{4.23}$$

where $\boldsymbol{p}_i^{old}$ and $\boldsymbol{p}_i^{new}$ are the old and the new 3D pose of the map point in the world.
Then the correction is performed on the map points that are observed by the current keyframe and the loop one, by fusing the duplicates and by adding the new ones, in order to align both sides of the loop.
The same fusion procedure is finally performed for all the keyframes in the neighbors of the current keyframe and the loop one, by finally closing the loop and connecting in an exhaustive way both sides of the graph (in fact the map points fusion will lead to the creation of new connections among all the keyframes involved).
After this correction, the graph is now ready to perform the essential graph optimization as explained in the section 3.3.3, to propagate the correction of the pose along all the keyframes.

4.3 Relocalization

The other main process that uses place recognition is, of course, the global relocalization module. This module is practically mandatory, as stated in 1, if a robust tracking is desired. In fact, vSLAM algorithms are particularly sensitive to fast camera motion or image blur, which could cause the loss of position information (since each pose estimate relies on the previous one). For this reason, the ability to correct relocalize the camera pose is mandatory.
The general idea is to take advantage of the BoW database to retrieve possible candidates, and then compute RANSAC iterations (as explained in the previous section) using the EPnP algorithm [19].
The main difference with respect to loop detection is that, for relocalization, no time consistency is required since no information is available on the actual position. For this reason the geometrical check is performed right after the candidates' retrieval, by computing correspondences of the ORB features (using the database direct index for matching).
All the candidates that pass this test go then under a RANSAC procedure that tries to compute an initial estimate of the vehicle pose.

If a candidate is marked as good by the RANSAC procedure, the outliers detected are discarded and the pose is refined: essentially, the optimization 3.3.1 and an ORB search using the optimal pose are alternatively computed, until enough correspondences are found. If the optimization retrieves directly a pose with enough inliers, then the estimated pose is marked as good and the tracking can continue; otherwise two additional subsequent searches and optimizations are performed to try to enlarge the quantity of correspondences supporting the pose. If this is successful then the candidate's pose is marked as good, otherwise the procedure continues elaborating on the next candidate.
This is done for every image captured until a new pose estimation is available. The assumption of this procedure is that, in order to correct relocalize the vehicle, the observed area must be already mapped, otherwise it would be impossible to estimate a pose for the algorithm.
Now a detailed explanation of the technique used for computing the robot pose in the RANSAC procedure is provided.

4.3.1 EPnP

The problem of finding the 3D pose of a camera given n 3D points and their 2D reprojection is known as *Perspective-n-Point* problem in the literature. The minimal number of points required to correctly compute the pose is 3, since 7 DoF are present in the problem (3 for translation, 3 for orientation and one for scale). One of the most famous algorithms is in fact P3P, which uses exactly 3 points to compute the camera pose. This however could easily result in solution ambiguity, since the basic approach of P3P leads produces up to four feasible solutions for the camera pose. For this reason, PnP problems (with $n \geq 4$) are of deep interest to the research community.
This problem is mainly tackled in two different ways: with iterative approaches, usually trying to minimize a cost function in order to find the better solution, or by closed-form ones, which seek to arrange a system of equations (often non-linear) that can be solved in a single step.
However, if not correctly treated, the resolution of a general PnP problem could require a non-negligible computational complexity. This was especially true until the introduction of EPnP in 2009 by Lepetit et. al in their article [19]. Other PnP algorithms (at the time), required for example $\mathcal{O}(n^5)$ or $\mathcal{O}(n^8)$ as computationally complexity, while EPnP is simply $\mathcal{O}(n)$.
The EPnP algorithm is closed-form based, so theoretically it doesn't require iterations to be performed. Since the authors of the algorithm found that its accuracy was less precise than other iterative approaches under proper initialization, they decided to enrich the algorithm with an additional Gauss-Newton method to increase its accuracy (without significant impact on the

computation) and so achieve state-of-the-art performances.
The key idea of the EPnP algorithm is to solve the problem using only 4 virtual points, called *control points* calculated from the whole set of n points available, bounding in this way the computational complexity of the problem.
Given $\boldsymbol{p}_i^w$ as a 3D point in the world coordinates system, $\boldsymbol{p}_i^c$ as a 3D point in the camera coordinates system, $\boldsymbol{c}_j^w$ and $\boldsymbol{c}_j^c$ the coordinates of the chosen control points in the two frame's system, the following relationships can be written:

$$\boldsymbol{p}_i^w = \sum_{j=1}^{4} \alpha_{ij}\boldsymbol{c}_j^w, \quad \boldsymbol{p}_i^c = \sum_{j=1}^{4} \alpha_{ij}\boldsymbol{c}_j^c, \quad \text{with} \quad \sum_{j=1}^{4} \alpha_{ij} = 1, \quad \forall i = 1, \dots, n. \tag{4.24}$$

Basically, the (4.24) states that every map point can be seen as a linear combination of the 4 control points, using the same coefficients in both frames. These control points can be chosen arbitrarily, but the method's stability increases if the first one is taken as the centroid of the points, while the others 3 along the principal directions of the data.
The general approach of the EPnP algorithm is to retrieve the coordinates of all the control points $\boldsymbol{c}_i^c$, so that then the relative transformation between them and $\boldsymbol{c}_i^w$ can be computed, returning the desired camera pose.
The first step is to compute the 4 $\boldsymbol{c}_i^w$ in order to find the corresponding α_{ij} coefficients. After that, the general projection law can be written:

$$w_i\boldsymbol{u}_i = K\boldsymbol{p}_i^c = K\sum_{j=1}^{4} \alpha_{ij}\boldsymbol{c}_j^c, \quad \forall i, \tag{4.25}$$

where $\boldsymbol{u}_i = [u_i, v_i, 1]^T$ are the coordinates of the corresponding keypoints in the image, K is the intrinsic camera matrix and w_i are the scalar projective parameters. This expression can now be expanded, as:

$$w_i \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \sum_{j=1}^{4} \alpha_{ij} \begin{bmatrix} x_j^c \\ y_j^c \\ z_j^c \end{bmatrix}, \quad \forall i. \tag{4.26}$$

The unknown parameters of this linear system, are the 12 control points' coordinates and the n projective parameters. However, the last row of the (4.26) shows that $w_i = \sum_{j=1}^{4} \alpha_{ij} z_j^c$, and this can be substituted in the remaining expressions, leading to two linear equations for each reference point and canceling out the projective parameters themselves:

$$\begin{cases} \sum_{j=1}^{4} \left[\alpha_{ij} f_x x_j^c + \alpha_{ij}(c_x - u_i) z_j^c\right] = 0 \\ \sum_{j=1}^{4} \left[\alpha_{ij} f_y y_j^c + \alpha_{ij}(c_y - v_i) z_j^c\right] = 0 \end{cases}. \tag{4.27}$$

Writing these equations for all the n correspondences, the general system

$$Mx = 0 \tag{4.28}$$

can be obtained, with $\boldsymbol{x} = [\boldsymbol{c}_1^{cT}, \boldsymbol{c}_2^{cT}, \boldsymbol{c}_3^{cT}, \boldsymbol{c}_4^{cT}]^T \in \mathbb{R}^{12}$ and $M \in \mathbb{R}^{2n \times 12}$ matrix. The desired solution belongs therefore to the kernel of M and can be expressed as:

$$\boldsymbol{x} = \sum_{i=1}^{N} \beta_i \boldsymbol{v}_i \tag{4.29}$$

where $\boldsymbol{v}_i$ are the right-singular eigenvectors of M, corresponding so to its N singular eigenvalues. Given the structure of the matrix M, they can be directly found as the null eigenvectors of the matrix $M^T M$, which will always be a constant 12×12 matrix. Its construction is essentially the most time-consuming step of the whole procedure (especially for large n), and it's in fact the only step whose computational demands increase with the number of correspondences (in a linear fashion). All the other operations performed (before and later) are bounded and almost constant in time.
The peculiarity of this approach is that the eigenvectors decomposition can be applied in constant time, independently from the number n of correspondences (if at least greater than 4), and also if the rank of the system is less than the number of unknowns (12). This happens because the matrix on which the decomposition is applied, $M^T M$, doesn't depend anymore on n.
The dimension N of the null space of $M^T M$ can vary from 1 to 4, depending on the camera model: in the ideal case, with perfect correspondence matches, it should be one, but affine camera models (first-order approximation of the general camera model) could bring this dimensionality up to 4, given the scale uncertainties on all the 4 control points. By also considering that noise presence and non-ideality of the matches would never assure perfectly zero eigenvalues, the number N cannot be known in advance.
This problem is tackled by computing the solutions (the estimated pose) for all four values of N (by using the four eigenvectors associated with the four smallest eigenvalues), and then keeping the one that better minimizes the reprojection error:

$$\sum_{i=1}^{N} ||\boldsymbol{u}_i - \pi(T_{cw}\boldsymbol{x}_i^w)||^2 . \tag{4.30}$$

The last problem to tackle is so how to compute the β_i coefficients in the (4.29), depending on the number N of eigenvectors involved. The basic idea, used for all four possible values of N, is to compute B imposing as a constraint that the distances between the control points in the camera frame, obtained with the (4.29), must be equal to the ones between the actual control points obtained by the real 3D points coordinates:

$$\left|\left|\sum_{k=1}^{N} \beta_k \boldsymbol{v}_k^{[i]} - \beta_k \boldsymbol{v}_k^{[j]}\right|\right|^2 = ||\boldsymbol{c}_i^w - \boldsymbol{c}_j^w||^2, \tag{4.31}$$

where $\boldsymbol{v}^{[i]}$ is the sub-vector of the eigenvector $\boldsymbol{v}$ corresponding to the coordinates of the control point $\boldsymbol{c}_i^c$.
Basically, all the quantities involved (the distance between the various $\boldsymbol{v}_k^{[i]}$ and between the control points $\boldsymbol{c}_i^w$) can be computed in advance after the decomposition of $M^T M$, allowing to speeding up the computation.

- *N=1*: In this case the (4.29) is simply equivalent to $\boldsymbol{x} = \beta \boldsymbol{v}$. So, by imposing the equality (4.31), an easy computation prove that:

$$\beta = \frac{\sum_{i,j \in [1,4]} ||\boldsymbol{v}^{[i]} - \boldsymbol{v}^{[j]}|| \cdot ||\boldsymbol{c}_i^w - \boldsymbol{c}_j^w||}{\sum_{i,j \in [1,4]} ||\boldsymbol{v}^{[i]} - \boldsymbol{v}^{[j]}||^2}. \tag{4.32}$$

- *N=2*: From the (4.31), it can be shown that the unknown variables are basically $\beta_{11} = \beta_1^2$, $\beta_{12} = \beta_1\beta_2$ and $\beta_{22} = \beta_2^2$ (this approach in cryptography is known as *linearization*). Therefore a linear system of six equations in the three variables can be constructed and easily solved:

$$L\boldsymbol{\beta} = \boldsymbol{\rho}, \qquad L \in \mathbb{R}^{6\times3}, \quad \boldsymbol{\beta} \in \mathbb{R}^3, \quad \boldsymbol{\rho} \in \mathbb{R}^6, \tag{4.33}$$

  where the elements of L and $\boldsymbol{\rho}$ are respectively derived from $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ and from the distance between the control points $\boldsymbol{c}_i^w$.

- *N=3*: The procedure is basically the same as for the previous case, except for the fact that in this case $L \in \mathbb{R}^{6\times5}$ and $\boldsymbol{\beta} \in \mathbb{R}^5$, since not all the β_{ab} are necessary for the solution.

- *N=4*: In this case it is not possible to use the same linearization used previously, because the number of unknowns would increase too much, (all the products $\beta_a\beta_b$ are threatened as independent variables). However, *relinearizing* again (and employing the commutativity of the multiplication) a similar linear system to the previous cases can be obtained, with $L \in \mathbb{R}^{6\times4}$ and $\boldsymbol{\beta} = [\beta_{11}, \beta_{12}, \beta_{13}, \beta_{14}]^T$.

To further increase the accuracy, the candidate values for $\boldsymbol{\beta}$ (for all four cases) are refined by also adding an optimization procedure. More precisely, using as always the constraints of distances between the control points, the following optimization is performed:

$$\boldsymbol{\beta} = \arg\min_{\boldsymbol{\beta}} \sum_{i,j \in [1,4] \text{s.t} i<j} \left(||\boldsymbol{c}_i^c - \boldsymbol{c}_j^c||^2 - ||\boldsymbol{c}_i^w - \boldsymbol{c}_j^w||^2\right) \tag{4.34}$$

where c_i^c is the estimated control points calculated with the (4.29).
Since this optimization depends only on $\boldsymbol{\beta}$, it requires practically constant time, not depending neither on the number of correspondences used neither on N, so the computational requirement is basically negligible, bringing nevertheless a pretty boost in the quality of the solution.
Finally, once the coordinates of the control points in the camera frame are available, the desired position and orientation can be computed. This operation is performed in a similar way to the previously treated *absolute orientation* problem, having basically the same structure to optimize:

$$\hat{R}, \hat{\boldsymbol{t}} = \arg\min_{R,\boldsymbol{t}} \sum_{i=1}^{N} |\boldsymbol{c}_i^c - (R\boldsymbol{c}_i^w + \boldsymbol{t})\,||^2 \tag{4.35}$$

The problem is tackled in a similar way as done for the Horn method: decoupling the rotation and the translation, estimating the optimal orientation by exploiting the quaternions representation (by so finding the one associated with the higher eigenvalues), and then computing the optimal translation. As previously stated, this procedure is actually performed for all four values of N, by finding therefore four valid transformations linking the camera to the world: the best one is the one that best minimizes the (4.30).
Since the EPnP algorithm is performed in a RANSAC procedure, its robustness to outliers is increased and, in addition, a double execution is performed: the first one is done as in the classic RANSAC scheme with only the minimal number of correspondences required (4), randomly extracted from the available set; then, after having classified outliers and inliers, if the obtained pose has enough inliers to pass the RANSAC test is further refined by performing again the EPnP using all the inliers found as correspondences. This ensures good quality in the obtained pose, along with additional robustness to outliers, that will be further refined by the following optimizations as stated before.

Chapter 5

Future developments

Since its release, ORB-SLAM2 has been one of the state-of-the-art algorithms, given its accuracy and versatility. However, its goodness was surpassed by its heir, ORB-SLAM3, which by adding IMU integration and a multi-map system brought its accuracy to a higher level.
Of course, this type of addition increases the computational demand of the algorithm, so a study on how to improve these features in order to implement them in an embedded system should be performed.
Additionally, a way to introduce the possibility of building a human-readable 3D map of the environment (or expanding an already existing one) would be a great addition, allowing also to reduce computational time by avoiding optimization of map points whose location is known in advance.
In this chapter, an explanation of how IMU measurements could be integrated, following the same idea of ORB-SLAM3 [3], will be illustrated, along with an illustration of the Octomap algorithm, one of the most efficient techniques to build and memorize a dense map of an environment.

5.1 Pre-integrated IMU measurements

The introduction of IMU measurements proposed in the ORB-SLAM3 algorithm is based on the article [20], which presents a preintegration theory based on the rotation group $SO(3)$[1], allowing to introduce IMU measurements as additional constraints between the graph keyframes themselves.
The basic idea is to capture all IMU data as long as they are available and then to build a single constraint (imposing relative position and orientation) between two consecutive keyframes, by putting together and integrating all these measurements. This allows of course the creation of additional edges besides the ones illustrated in the section 3.2.1.

[1]Manifold theory of rotation group is treated in the appendix A

5.1.1 IMU and kinematic model

The model used for the IMU is a simple double integrator, that takes the measurements of the gyroscope and the accelerometer, affected however by noise:

$$\begin{aligned} \tilde{\boldsymbol{\omega}}^b_{wb} &= \boldsymbol{\omega}^b_{wb} + \boldsymbol{b}^g + \boldsymbol{\eta}^g \\ \tilde{\boldsymbol{a}}^b &= R_{bw}(\boldsymbol{a}^w - \boldsymbol{g}^w) + \boldsymbol{b}^a + \boldsymbol{\eta}^a \end{aligned} \tag{5.1}$$

where the superscript indicates the frame in which those quantities are expressed, and $\boldsymbol{\eta}$ and $\boldsymbol{b}$ are the two types of disturbances affecting the measurements, respectively a white noise and a bias (modeled as a random walk process). It is then natural to understand the meaning of all the other symbols in the (5.1). Of course, all the values presented in these equations are time-dependent, but to simplify the notation this dependency has not been highlighted.
Then, by assuming that both a^w and ω^w_{wb} remain constant in the interval of time Δt between two IMU measurements, and by using the standard kinematic model for a rigid body motion in space

$$\dot{R}_{wb} = R_{wb} S_{\omega^b_{wb}}, \quad \dot{\boldsymbol{v}}^w = \boldsymbol{a}^w, \quad \dot{\boldsymbol{p}}^w = \boldsymbol{v}^w \tag{5.2}$$

where $S_{\omega^b_{wb}}$ is the skew-symmetric matrix associated with the angular velocity of the body, the general dynamic model of the vehicle can be written as:

$$\begin{aligned} R_{wb}(t+\Delta t) &= R_{wb}(t) \exp_{SO(3)}\left(\boldsymbol{\omega}^b_{wb}(t)\Delta t\right) \\ \boldsymbol{v}^w(t+\Delta t) &= \boldsymbol{v}^w(t) + \boldsymbol{a}^w(t)\Delta t \\ \boldsymbol{p}^w(t+\Delta t) &= \boldsymbol{p}^w(t) + \boldsymbol{v}^w(t)\Delta t + \frac{1}{2}\boldsymbol{a}^w(t)\Delta t^2 \end{aligned} . \tag{5.3}$$

The quantity $\boldsymbol{\omega}^b_{wb}$ and $\boldsymbol{a}^w$ can be easily found by using the (5.1). Of course, this approximation of the integration can be considered valid only if the IMU measurements are provided at sufficiently high frequency, otherwise higher order methods should be used. In the (5.3), the $\exp_{SO(3)}$ symbol represents the exponential map that links the rotational manifold $SO(3)$ to the tangent vector space (more precisely, the exponential map brings quantities from the tangent space to the manifold).

5.1.2 Pre-integrated measurements

Then, by putting together the (5.1) and the (5.3), and by iterating the integration for all the IMU measurements at time k between two consecutive keyframes i and j (as illustrated in the figure 5.1), it's possible to find:

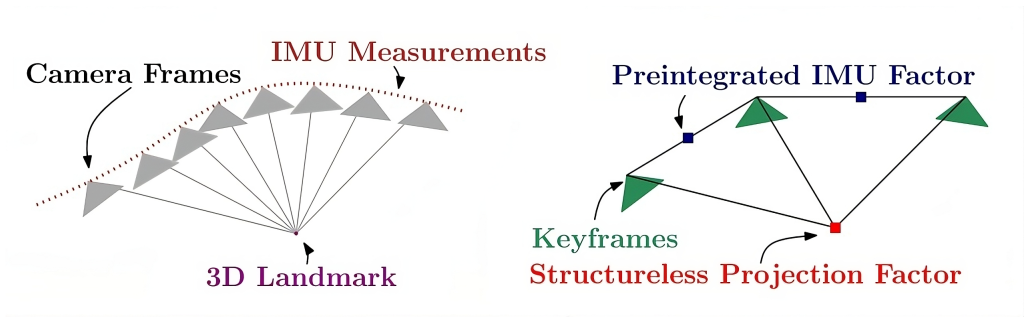


Figure 5.1: *Illustration, taken from the article, on how several IMU measurements are preintegrated in a single component, imposing a constraint between two keyframes.*

$$
\begin{aligned}
R_j =& R_i \prod_{k=i}^{j-1} \exp SO(3) \left((\tilde{\boldsymbol{\omega}}_k - \boldsymbol{b}_k^g - \boldsymbol{\eta}_k^g) \varDelta t \right) \\
\boldsymbol{v}_j =& \boldsymbol{v}_i + \boldsymbol{g}\varDelta t_{ij} + \sum_{k=i}^{j-1} R_k \left(\tilde{\boldsymbol{a}}_k - \boldsymbol{b}_k^a - \boldsymbol{\eta}_k^a \right) \varDelta t \\
\boldsymbol{p}_j =& \boldsymbol{p}_i + \sum_{k=i}^{j-1} \left[\boldsymbol{v}_k \varDelta t + \frac{1}{2}\boldsymbol{g}\varDelta t^2 + \frac{1}{2} R_k \left(\tilde{\boldsymbol{a}}_k - \boldsymbol{b}_k^a - \boldsymbol{\eta}_k^a \right) \varDelta t^2 \right]
\end{aligned}
\tag{5.4}
$$

where $\varDelta t_{ij}$ is the time distant between the two keyframes. From this model, it is then pretty fast to obtain the so-called *pre-integrated* IMU measurements between the two keyframes, as:

$$
\begin{aligned}
\varDelta R_{ij} &= \prod_{k=i}^{j-1} \exp_{SO(3)} \left((\tilde{\boldsymbol{\omega}}_k - \boldsymbol{b}_k^g - \boldsymbol{\eta}_k^g) \varDelta t \right) \\
\varDelta v_{ij} &= \sum_{k=i}^{j-1} \varDelta R_{ik} \left(\tilde{\boldsymbol{a}}_k - \boldsymbol{b}_k^a - \boldsymbol{\eta}_k^a \right) \varDelta t \qquad . \\
\varDelta p_{ij} &= \sum_{k=i}^{j-1} \left[\varDelta \boldsymbol{v}_{ik} \varDelta t + \frac{1}{2} \varDelta R_{ik} \left(\tilde{\boldsymbol{a}}_k - \boldsymbol{b}_k^a - \boldsymbol{\eta}_k^a \right) \varDelta t^2 \right]
\end{aligned}
\tag{5.5}
$$

The greatest advantage of the measurements written in this way is that they are independent of the actual poses of the keyframes, while the (5.4) should be recalculated every time that the position of the keyframe i changes.

5.1.3 IMU residuals

The quantities in the (5.5) can now be used to enforce an additional edge between two keyframes, allowing to further increase the power of the graph-

based optimization.
However, in order to properly add these constraints to the graph-based optimization reported in the chapter 3, additional variables to optimize must be added for each keyframe, namely the velocity $\boldsymbol{v}_i$ and the two biases estimate $\boldsymbol{b}_i^g$ and $\boldsymbol{b}_i^a$. In addition to that, the keyframe poses must be elaborated to be referred to the same body frame in which the IMU measurements are referred. This of course doesn't add any additional variables but requires an additional relationship between the camera pose, used for example in the (3.4), and the one used by the inertial residuals.
Having said that a general form of the inertial residual between two consecutive keyframes i and j can be written as:

$$\begin{aligned}\boldsymbol{r}_{ij} &= \begin{bmatrix}\boldsymbol{r}_{\Delta R_{ij}}^T & \boldsymbol{r}_{\Delta v_{ij}}^T & \boldsymbol{r}_{\Delta p_{ij}}^T\end{bmatrix}^T \\ \boldsymbol{r}_{\Delta R_{ij}} &= \log_{SO(3)}\left(\Delta R_{ij}^T R_i^T R_j\right) \\ \boldsymbol{r}_{\Delta v_{ij}} &= R_i^T\left(\boldsymbol{v}_j - \boldsymbol{v}_i - \boldsymbol{g}\Delta t_{ij}\right) - \Delta v_{ij} \\ \boldsymbol{r}_{\Delta p_{ij}} &= R_i^T\left(\boldsymbol{p}_j - \boldsymbol{p}_i - \boldsymbol{v}_i\Delta t_{ij} - \frac{1}{2}\boldsymbol{g}\Delta t_{ij}^2\right) - \Delta p_{ij}\end{aligned} \tag{5.6}$$

where the quantities ΔR_{ij}, Δv_{ij} and Δp_{ij} are calculated as in the (5.5). Then, a general cost term can be added to all the optimization involving multiple keyframes (as for example in the local BA (3.7)):

$$\sum_{i,j\in\mathcal{G}|j=i+1} ||\boldsymbol{r}_{ij}||^2_{\Sigma_{\mathcal{I}_{ij}}^{-1}} . \tag{5.7}$$

The covariance matrix $\Sigma_{\mathcal{I}_{ij}}$ expresses the uncertainties related to the measurements and the procedure to obtain it, along with a formal proof on how to obtain the (5.6) as a MAP estimate residual, is reported in the original article [20].
The addition of IMU measurements into the SLAM algorithm could lead to a great increase in estimation accuracy, allowing the enrichment of the information provided by pure camera systems. One of the main benefits would be an increase in tracking robustness since having IMU measurements at disposal would overcome the weakness of the camera to localization under fast motions.

5.2 Octomap

Since the dense representation of the environment is of crucial importance for various map-based tasks, such as obstacle avoidance and path planning, an efficient way for performing this environment reconstruction is mandatory.

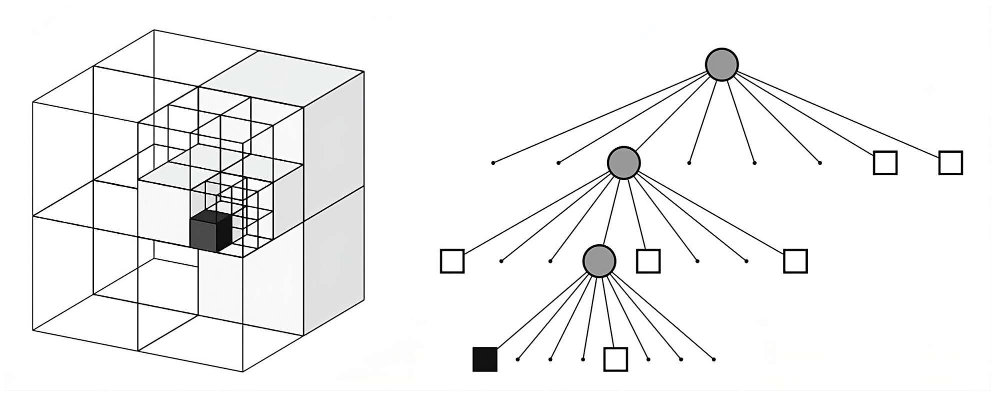


Figure 5.2: *Illustration, taken from the article, on how an octree construction works. Each full voxel is marked as black, while all the children of a node are pruned if they store the same value.*

Octomap was introduced in [21] precisely for this reason, bringing a new on-the-edge way to construct a dense map. More precisely, the Octomap algorithm builds an environment representation fusing an octree structure with a probabilistic occupancy grid mapping.

5.2.1 Octrees

Octrees are a special type of hierarchical data structure used for spatial subdivision of 3D spaces, whose use for 3D computer graphics was first introduced in [22].
An octree is basically a graph where each node has eight children: each node represents the space contained in a cubic volume, usually called *voxel* in computer vision, and each one of the eight children is associated with one of the eight sub-volumes in which the parent voxel is divided. By recursively applying this division on a 3D space (by starting from one node representing all the available space), e hierarchical subdivision of the space can be obtained, whose resolution is defined by the minimum voxel size (directly related to the tree depth).
Originally they were thought for usage in binary space, as for example an occupancy grid, whose cell can be empty or full. The ambiguity of an unknown cell (that could be treated as empty or uninitialized), in the octomap implementation is tackled by explicitly representing free volumes in the tree, by so leaving uninitialized all the nodes belonging to unmapped area.
The main advantage of octrees is that by using binary values for the voxels, all the children of a node can be pruned if they share all the same values. This allows a very compact representation of the environment, discarding all redundant data regarding for example voxels inside objects and voxels in free

space. An example of octree construction can be seen in 5.2.
With respect to a full-size binary grid, that would require a voxel for every little portion of space, the access to the elements is of course slower (the one of a grid is $\mathcal{O}(1)$), but its still pretty fast, being $\mathcal{O}(d)$ with d being the depth of the octree considered, adding however a substantial decrease in memory usage.

5.2.2 Probabilistic occupancy grid

Instead of using binary values for voxels, in a probabilistic occupancy grid they are instead filled with their probability of being occupied given some sensor measurements: more precisely, given z_t the current measurement of the sensor, the occupancy probability of a voxel n can be estimated as:

$$P(n|z_{1:t}) = \left(1 + \frac{1 - P(n|z_t)}{P(n|z_t)} \frac{1 - P(n|z_{1:t-1})}{P(n|z_{1:t-1})} \frac{P(n)}{1 - P(n)}\right)^{-1}, \tag{5.8}$$

where $P(n|z_t)$ is the probability of the observed voxel to be occupied given the measurement z_t (depending so on the type os sensor used) and $P(n)$ is a prior probability. The assumption on an uniform prior probability (so all the voxel with the same probability of being occupied) leads to $P(n) = 0.5$.
Using logarithmic functions the (5.8) can be rewritten as

$$L(n|z_{1:t}) = L(n|z_{1:t-1}) + L(n|z_t), \quad L(\cdot) = \ln \frac{P(\cdot)}{1 - P(\cdot)}, \tag{5.9}$$

that allows faster computation, using just additions.
The main advantage of using this type of grid is that probabilistic values can be updated basically without limits and so this approach can also take into account dynamic environments and sensor disturbances.
To efficiently use an occupancy probability grid a threshold is usually applied, such that a voxel is considered free or occupied only when its occupancy probability is below or upper a certain value. So, while the (5.9) is valid for static environments, dynamic ones require higher responsiveness in order to quickly detect map changes. For this reason the (5.9) is modified into:

$$L(n|z_{1:t}) = \max(\min(L(n|z_{1:t-1}) + L(n|z_t)), l_{\text{max}}, l_{\text{min}}) \tag{5.10}$$

leading basically to a truncated update (values are updated up to their limit threshold), speeding up the detection of changes in the map and assuring a bounded confidence in the map.

5.2.3 Octomap fusion

The octomap algorithm basically takes and puts together the two idea above illustrated: each node in the octree memorize its occupancy probability as (5.10), taking so both the benefits of the compact size of the octree and the fast map update of probabilistic occupancy grid. The clamped update defined by the (5.10) allow also for pruning techniques since brings with itself the possibility to mark a voxel as empty or full if the threshold is reached.
Since the probability update is applied only on the leaf nodes, a method to extend the occupancy probability to the parent nodes must be provided: several methods are available, from which the most common are a mean value or a maximum criterion:

$$L_p(n) = \frac{1}{8}\sum_{i=1}^{8} L(n_i) \qquad L_p(n) = \max_i L(n_i). \tag{5.11}$$

The choice of this criteria depends on the conservative's level desired, by assigning or not a full value to an actually empty cell (by making the path planning more collision-free).

5.2.4 Sparse outlier removal

One of the main problems of building an octomap from a raw points cloud (obtained for example from an RGBD camera), is that point cloud data with whom build the map are still available but they are affected by outliers (due for example to errors in the measurements and incorrect triangulations): if an octomap is built directly with these point clouds, the map would be inaccurate and dangerous to use for navigation tasks.
A methodology to deal with this problem has been introduced in [23], which proposed the usage of K-Nearest Neighbor and Gaussian distribution to perform outliers removal.
The K-Nearest Neighbor is a standard technique for outlier removal, that performs an organization of the measured raw point cloud using a K-dimensional tree: it is basically a binary tree, in which every node is a K-dimensional point, and each non-leaf node splits the space in half, from which then the tree construction proceeds.
To filter out the outliers, the approach is based on evaluating how much a point is closer to its neighbor, and can be summarized in the following steps:

- For each point $\boldsymbol{p}_n$ in the points cloud, take the closest K points $\boldsymbol{p}_i$.
- Compute the average distance of each point with respect to its K-neighbor:

$$d_n = \frac{1}{k}\sum_{i=1}^{k} ||\boldsymbol{p}_n - \boldsymbol{p}_i|| \,. \tag{5.12}$$

- A point is considered an outlier and discarded if its average distance d_n is above a certain threshold d_t, meaning basically that the target point is too far with respect to the other points.

The main drawback of this technique is that the value d_t must be set manually and it strongly affects the outlier filtering mechanism. For this reason, in the article cited, a change in the classic K-Nearest Neighbor approach is made: a Gaussian distribution of the average distances computed for each point is assumed and then used to overcome this flaw.
More precisely, after having calculated d_n for all the points in the point clouds, the following quantities are calculated:

$$\mu = \frac{1}{N}\sum_{n=1}^{N} d_n, \quad \sigma = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(d_n - \mu)^2}, \tag{5.13}$$

being naturally the mean and the standard deviation of the Gaussian distribution considered. Then a confidence level can be set, determined by:

$$\begin{cases} c_1 = \mu - \alpha\sigma \\ c_2 = \mu + \alpha\sigma \end{cases} \tag{5.14}$$

where α is an optional scale factor used to adjust the size of the range (in the article was set equal to 1). Then a point is preserved if its K-neighbor average distance d_n falls inside the range $[c_1, c_2]$, and marked as outlier and discarded otherwise.
This approach allows to improve the classic K-Nearest Neighbor by dynamically adapting the filter intensity to the specific points cloud measured.

5.2.5 Octomap from vSLAM

The original technique for building octomap takes into account mainly laser sensor measurements, from which points clouds can be directly constructed. The advantage of these sensors, in which also RGB-D cameras can be included, is that they provide a direct and reliable depth measurement. However, in stereo cameras for example, this depth must be estimated from observations and its accuracy depends on the distance of the observed point from the image plane (recalling the (2.10)).
This problem is tackled in [24], where the uncertainties due to stereo camera observations are taken into account by introducing a new formulation for

the probabilities in (5.9). This is done by defining a range $[d_{\text{min}}, d_{\text{max}}]$ in which depths are considered valid; then the voxel probability is computed taking into account the decreasing accuracy of the measurements. More precisely, given the sigmoid function

$$P = \frac{1}{1+\exp(-\alpha)}, \quad \alpha = \ln\left(\frac{P}{1-P}\right) \tag{5.15}$$

obtained by inverting the logarithmic map of the (5.9), the new probability can be formulated as a function of the depth d:

$$P(d) = 1 - \frac{1}{1+\exp[-k(d-d_0)]}, \qquad k = 2\frac{\ln\left(P_{\text{max}}^{-1}-1\right)}{d_{\text{min}}-d_{\text{max}}}, \quad d_0 = \frac{d_{\text{min}}+d_{\text{max}}}{2} \tag{5.16}$$

where P_{max} is the probability assigned at an observation at the distance d_{min}. This formulation allows so to define a distributed varying probability along all the feasible range bringing, due to the symmetry of the sigmoid function, $P(d_{\text{max}}) = P_{\text{min}} = 1 - P_{\text{max}}$ and $P(d_0) = 0.5$. The (5.16) takes basically the role of $P(n|z_t)$ in the standard octomap probability formulation (5.8).
Given the uncertainties of observations, however, the same map point could still be added repeatedly times at slightly different positions. To avoid this phenomenon a probabilistic merging is applied: when a new point c_1 is added to the map, the distance from the closest point c_2 is calculated, and if below a certain threshold (directly depending on the map resolution) a new map point c_3 is created by fusing together the two previous observations. The position of the new points is calculated as:

$$\boldsymbol{S}(c_3) = \frac{P(d_1)}{P(d_1)+P(d_2)}\boldsymbol{S}(c_1) + \frac{P(d_2)}{P(d_1)+P(d_2)}\boldsymbol{S}(c_2), \quad \boldsymbol{S}(c_i) = \begin{bmatrix} x_i \\ y_i \\ z_i \\ d_i \end{bmatrix}. \tag{5.17}$$

Then the occupancy probability can be calculated directly using the(5.16). The procedure is then performed recursively until no more points whose distance is below the threshold are present around the new point.

Chapter 6

Simulations and results

The main purpose of this dissertation was not only to provide an exhaustive analysis of the ORB-SLAM2 algorithm, but also to try to implement it in an embedded system, so where computational power is limited with respect to a modern laptop.
For doing this, the main thing to do was to speed up the original ORB-SLAM2 code, in order to reduce its computational burden. After having done that, a series of tests using the EuRoC dataset for stereo camera [25] were performed, comparing the performance of the custom version against the original one. Finally, a real test, by implementing the code on a quadcopter was performed, in order to see how well the new version of the ORB-SLAM2 could perform.

6.1 Implementation details

Given the age of the original code, built on top of the original ORB-SLAM code, the first changes performed aimed to modernize the code.
The major change in this optic was to substitute basically every raw pointer used inside the code with a smart one. The main difference between smart pointers with respect to raw ones is that their resource access is automatically handled by the pointer itself: this avoids the explicit deletion of the pointed object, mandatory instead for the raw ones associated with the heap memory. Smart pointers ensure that every resource is assigned to the object at the creation time and, more importantly, that is released as soon as the pointer is canceled. More precisely, two main types of smart pointers exist: *shared* and *unique*. The latter, as the name says, points to an object that must have a unique owner: the ownership can of course change, but the object would never have two owners at the same time, raising exceptions if this occurs. This type of smart pointer frees the resources taken by the object as soon as it is dereferenced. Shared ones, instead, allow multi-ownership and the re-

sources are freed as soon as the last pointer referring to the object is canceled. The use of smart pointers permits a better use of resources, avoids conflicts and assures a deterministic usage of the memory (since the creation and the destruction of the object is fully controllable by the programmer).
The second main change done, resulting in the first computational reduction, was to use the Eigen library for all the algebraic operations involved while leaving OpenCV (that in the original code was used for everything) only for features and keypoints managing. This was done because the Eigen library has more optimized functions for matrix and vector calculations, allowing also the use of both dynamic and fixed size matrices, so taking into account all the information already available at compile time to further optimize the code.
In addition, Eigen is fully compatible with the optimization library used, G2O [26], while OpenCV matrices all required an additional conversion step in order to be used. G2O is an open-source C++ framework, specifically designed for optimizing graph-based nonlinear error functions. Given this peculiarity, it provides a direct API for interfacing with SLAM and BA problems. It is worth noticing that also the Ceres library was considered as a candidate substitute for optimization, given its API that is directly related to the mathematical formulation of the problem while G2O, being graph-based, requires an explicit construction of the nodes and the edges between them. However, in the end, the use of G2O was preserved, given its higher compatibility with the algorithm structure.
The choice of using the Eigen library has made it possible to use more specific functions taking advantage of the matrix structure involved: for example, in the case of SVD of symmetric matrices (that happens for examples in the Horn method and also in the EPnP algorithm), Eigen makes available specific function in order to take advantage of it. Also when dealing with $SE(3)$ transformations, Eigen allows the use of special matrix containers that exploit the specific structure of the homogeneous transformations both for multiplication and inverse calculation.
An additional improvement were also made by directly acting on the code, by for example performing in more efficient ways the same computation (for example, by avoiding repeating the same operation multiple times if possible).
One of the biggest changes was acting directly on the function responsible for the creation of stereo matches. The original implementation was pretty accurate by performing first a grid search to find possible candidates in the right image, and then by performing an interpolation based on the multiscale observations of the identified keypoint, to find the best one. Finally, the right matches were filtered by discarding the ones with a disparity too high with respect to the others.
Despite being very accurate, this implementation was pretty demanding from

the computational point of view, especially for a function executed on the tracking module, the most critical from the time point of view.
For this reason, a lightweight implementation of the stereo matching has been performed. The basic idea is to order the keypoints of the right image by the increasing order of the v_R coordinate. After that, for each left keypoint a search is performed by checking all the right keypoints whose v_R is similar to v_L. To take into account the multi-scale observations, only keypoints observed in a similar scale (so at a distance of a maximum of one level) are maintained. Finally, the same filtering criteria of the original code on disparity is applied. This procedure, despite being more approximated, has led to a decrease in the computational time, by however keeping almost the same accuracy of the original keypoints.
Since the algorithm structure is divided into three main threads, performing respectively tracking, local mapping and loop closing, a better explanation of the structure of the three threads is now provided. All the actions taken by the following threads that could cause race conditions (for example, two threads modifying the same keyframe or the same map point) and all not-safe operations are handled with the use of mutex locks. Given the not ideality of mutex locks from a computational viewpoint, their use is maintained for the smallest amount of time possible.

6.1.1 Tracking thread

The tracking thread is the module that, taking as input a stereo image, calculates the estimated camera pose. For performing this procedure, the first step is the creation of a frame (that is basically a structure associated with each camera pose). Each frame is created by extracting the ORB features from the two images (left and right) and performing a search to find stereo matches. The keypoints without correspondences are marked as monocular. Every keypoint is then associated with a grid structure, in order to perform a fast retrieval of the coordinates of each one of them.
After the frame creation, which is also the most time-consuming step given the ORB features extraction, the pose estimation procedure is performed. The whole procedure can be summarized in two steps: a pose initial prediction and a pose refining, both based on the optimization (3.4). The first step can be performed in three different ways, according to the camera situation.

- The standard procedure is to track map points by projecting in the current frame the ones seen in the last one. The correspondences are found by exploiting the ORB descriptors' distance and by taking benefits of the grid organization of the keypoints. The search is performed by also taking advantage of the estimated relative pose between the last and the actual frame. If enough correspondences are found, the pose optimization can then be exploited. Given the optimization problem (3.4), an initial pose estimation must be available: this estimate is calculated using the last frame pose, assuming a constant motion law (so the displacement of the current frame with respect to the last one is assumed equal to the displacement between the last frame and the previous one).

- If the precedent step fails for too few correspondences (or if a relocalization has happened recently) the search is performed on the points tracked by the reference keyframe of the last frame, exploiting the BoW direct index for fast matching (so the BoW of the current frame must be computed). In this case, the initial pose of the optimization is obtained by setting the last frame pose as a candidate.

- If even the previous step fails, then this means that the tracking has lost information on the current frame position and a relocalization procedure is launched. The relocalization, performed from the next frame, has the job of re-estimating a feasible position of the camera. Only if a solution is found then the tracking thread can proceed.

To take advantage of the local mapping module running in parallel (that refines keyframes and map points poses), the last frame pose is always updated with respect to its relative position to its reference keyframe, as well as the points tracked.
After the first pose prediction, a second optimization is performed, with a wider search of map point correspondences to enhance the accuracy. Starting from the matches tracked, all the keyframes observing the matched map points are retrieved (thanks to the information stored in the map points themselves this is done in a pretty fast manner). From this set of local keyframes, the one with the higher correspondence of point with the current frame is targeted as its *reference keyframe*. Moreover, also some keyframes that are neighbors to the already included ones are taken, to further extend the search area.
Then, all the map points observed by these keyframes are projected into the current frame, searching for correspondences with the extracted keypoints. Finally, the optimization (3.4) is performed again on this bigger set of matches, refining the estimation accuracy. Also in this case, if the number of matched drops below a certain threshold, the tracking is considered lost and a global relocalization is launched.
After having produced a pose estimate, the current frame is analyzed to see if the requirements for a new keyframe creation are met and, in the last case, the keyframe and its associated map points (obtained by triangulating close stereo points) are created.

6.1.2 Local mapping thread

The local mapping thread is composed of a main cycle, running periodically, that checks if a new keyframe has been inserted. If the search is successful, then the following activities are performed, one after the other.

- First, the newly created keyframe is processed, in order to create the proper covisibility connections, and is so added to the graph.
- Right after the previous step, the map point culling procedure is executed, canceling out the one with too few observations.
- Then, a procedure to create and triangulate new map points is performed, exploiting the different observations available from more keyframes.
- Then a fusing procedure is performed on the same points observed by different keyframes, but associated with different keypoints, in order to construct a more compact map. This procedure is however performed only if there aren't any additional keyframes waiting to be processed.
- Then a local BA is performed, following the procedure highlighted in the section 3.3.2.
- Finally a culling step on the covisibility neighbor of the current keyframe is performed.

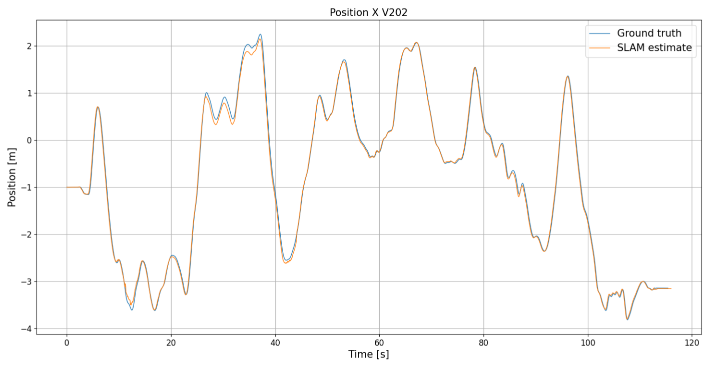
Once all the listed steps are performed, the new keyframe is passed to the loop closing module and the local mapping thread will wait for the next keyframe to be inserted (or will start to process the pending ones).

6.1.3 Loop closing thread

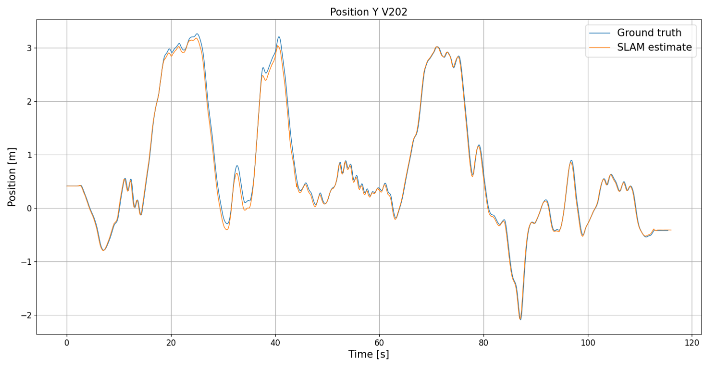
The last thread running is the loop closing one, which tries to detect a loop for every keyframe passed to it. The general functioning is the same as the local mapping, consisting of a loop execution: when a new keyframe is passed to this thread the database is queried, in order to detect the loop, as explained in the section 4.2; if it is successful, then the loop closing procedure reported in the section 3.3.3 is performed. Finally, a global BA optimization is performed, launched however on another thread to reduce its impact on the other SLAM modules. To avoid conflicts the local mapping thread is stopped and any running global BA thread is also aborted. This is done during the correction phase to avoid the creation or destruction of connections and keyframes by one thread, while they are processed by the other one.

6.2 Simulations

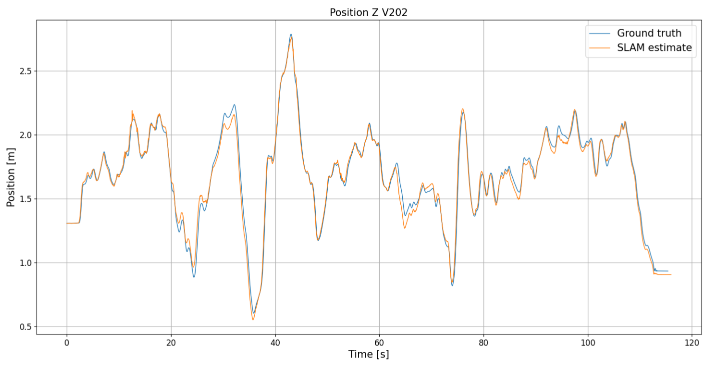
To test the algorithm, as said before, the EuRoC dataset was used, since it provided both stereo camera images and ground truth data along with a wide variety of scenarios captured, at different levels of difficulty.
Given the multi-thread approach, bringing so a non-deterministic outcome of the algorithm under the same dataset, multiple tests were performed on the same set of images so that an average behavior would be available.
More precisely, four simulations for every set present in the EuRoC dataset were performed, both for the original and the custom algorithm, on a laptop equipped with an Intel Core i7-8550U, a platform definitively less powerful than the one used for testing in the original work of ORB-SLAM2.
The outcomes were satisfactory: the new version of the algorithm has around the same accuracy as the original one, despite being faster and so more reliable in devices with small computational power available.
More precisely, in the figure 6.1 the comparison between the ground truth and the estimate position (respectively for the coordinate X, Y and Z) obtained in the set V202 (a Vicon room exploration classified with a medium difficulty) is shown. Analyzing the plots, it can be seen how the estimation and the ground truths follow the same identical behavior, despite a slightly increasing drift of the estimate: however, as it can be seen, this drift is reduced in several parts of the plot, and this is due to the loop closing mechanism, that has the job of providing exactly this result.
In the original article, in order to measure the accuracy of the algorithm, all the predictions were corrected offline using the relative position of the frames to their reference keyframes, in order to fully exploit the graph structure of the problem (by so correcting also poses acquired before a loop closure). Despite being an optimal index for evaluating the whole algorithm, this approach doesn't provide useful results in a real-time scenario, where the SLAM algorithm must provide an immediately available pose. The errors obtained with these two different approaches are visible in the figure 6.2, both for the original algorithm and the custom one.
As it can be seen in the plots, the error with the offline prediction is of course smaller, but also the online one performs quite well, resulting even in a slight improvement with respect to the original algorithm.
To conclude the analysis of this dataset, the measured times for both the original and the custom one are reported in the table 6.1. The times were measured by splitting each module into its most fundamental parts, in order to have an overall view of the performance impacts. The *response time* is instead a measure of the time that is needed by the algorithm to return a pose prediction after having received a pair of images (it is basically a measure of the responsiveness of the algorithm).
As far as concerns the loop closing mechanism, the distinction between the



(a) *Simulation result of coordinate X.*

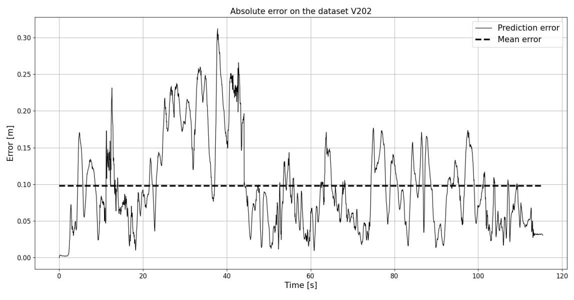


(b) *Simulation result of coordinate Y.*

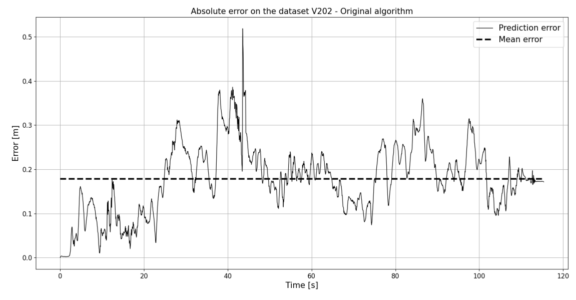


(c) *Simulation result of coordinate Z.*

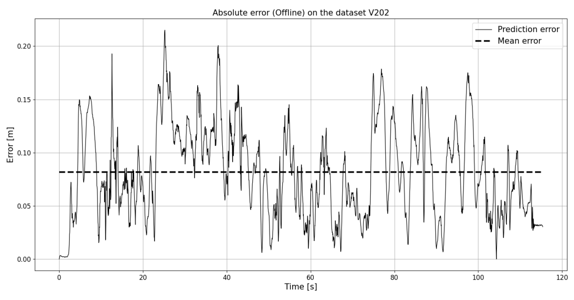
Figure 6.1: *Simulation result on the dataset V202 (medium).*



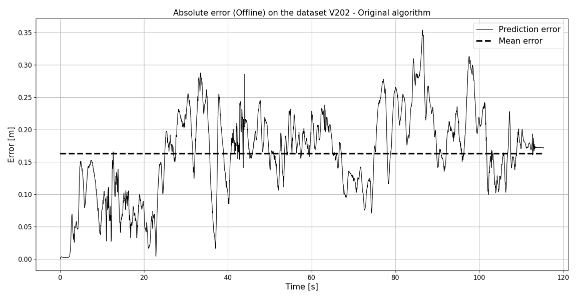
(a) *Online prediction error of the custom version of the algorithm.*



(b) *Online prediction error of the original version of the algorithm.*



(c) *Offline prediction error of the custom version of the algorithm.*



(d) *Offline prediction error of the original version of the algorithm.*

Figure 6.2: *Comparison between the online and offline errors obtained in the dataset V202.*

SE3 detection and the *SE3 computation* highlights, respectively, the time needed for the whole $SE(3)$ detection and computation (ending with the loop confirmation) and the time needed by the RANSAC iteration to find a feasible candidate using the correspondences. The *loop detection* is instead the measure of time required to retrieve one or more candidates from the database query.

As it can be seen, the new version of the algorithm performs better in basically all the tasks, with some exceptions. All the tasks that involved the OpenCV functionality, such as stereo rectification and ORB extraction are basically the same and, as far as concern the tracking module, they are the most time-consuming part. For all the other parts, the new version performs quite better, with an average time reduction of almost 40%. In some cases, as for example the *stereo matching* and the *SE3 computation*, the new algorithm is about ten times faster. The only tasks that are degraded are the culling procedure and the database querying, but the overhead is highly negligible with respect to all the other times reduction.

In the figures 6.2, 6.4 and 6.5 the comparisons of the errors obtained in the other simulations are reported. For the reasons specified earlier, only the errors obtained with the online prediction have been reported.

The correspondent times have been reported in the tables 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9 and 6.10. As it can be seen, the performances obtained in the MH datasets are worse than the ones obtained in the Vicon rooms 1 and 2: this is probably due to the fact that in the MH dataset, the area in which the exploration is performed is very wide, forcing the SLAM to use more far points than close ones to localize the drone, resulting in both an increasing in time needed and also in a worse performance obtained. However, the precision of the new algorithms, on average, is similar to the original one, by also gaining a significant time reduction in all the simulations exploited, proving the success of the applied approach.

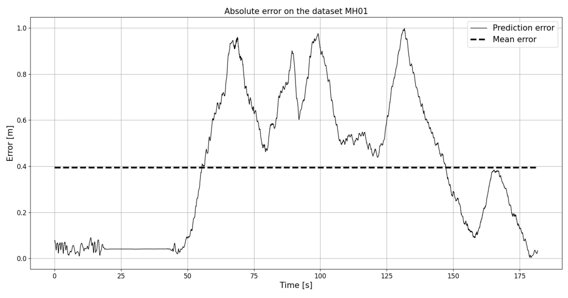
An additional test on the dataset V203 was also performed but, as for the original ORB-SLAM2 algorithm, the performances were really degraded with respect to all the other datasets, given its high difficulty.

However, for completeness and also to prove the algorithm's robustness to tracking loss, the simulation results are reported in the figure 6.6, along with the time measurements in the table 6.11 (in the time table, given its presence with respect to the other datasets, also the relocalization time measurement is reported).

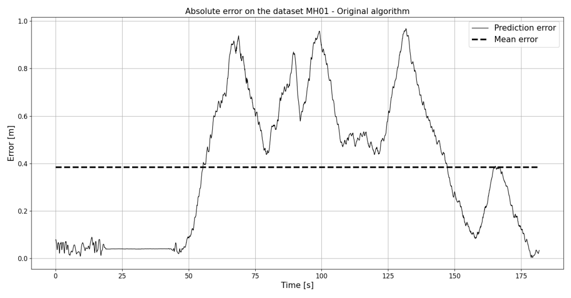
As it can be seen, despite the tracking loss and the increasing accumulating error, the algorithm is still able to correctly relocalize itself if the right conditions are met, as well as to fully detect loop closure and use them to correct the accumulated error.

Vicon room 2 - medium		
-	Original	Custom
Rectification	1.91 ± 0.59	1.93 ± 0.50
ORB extraction	26.62 ± 4.39	25.85 ± 4.07
Stereo matching	17.85 ± 5.12	1.76 ± 0.39
Pose prediction	2.83 ± 1.07	2.85 ± 1.02
Local map tracking	13.83 ± 8.43	7.52 ± 3.62
New KF decision	0.070 ± 0.086	0.082 ± 0.121
New KF creation	1.99 ± 0.86	0.72 ± 0.25
Response time	63.36 ± 15.73	38.83 ± 6.51
KF insertion	17.19 ± 6.79	11.44 ± 4.82
Map points culling	0.51 ± 0.29	0.70 ± 0.43
Map points triangulation	9.21 ± 4.94	6.77 ± 3.68
Map points fusion	72.96 ± 29.12	24.72 ± 12.44
Local BA	214.64 ± 187.46	165.33 ± 154.46
KF culling	7.73 ± 7.75	8.44 ± 8.07
Loop detection	6.34 ± 5.45	6.59 ± 4.51
SE3 detection	33.66	14.46
SE3 computation	1.17	0.034
Loop correction	208.35	77.16
Essential graph optimization	129.41	78.37
Global BA	648.27	604.97
Graph update	176.44	71.51

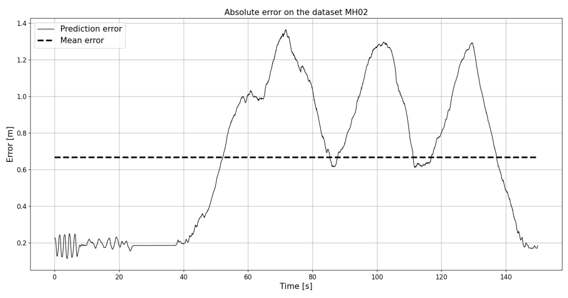
Table 6.1: *Table representing the time comparison obtained on the set V202. All the times are measured in ms (mean $\pm$ std. deviation).*



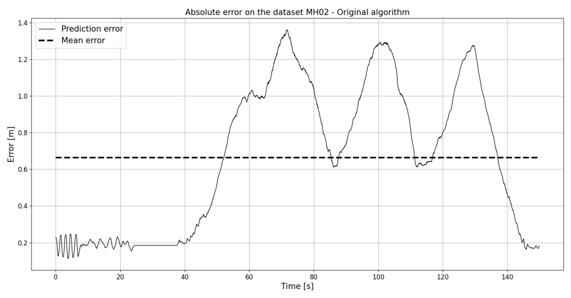
(a) *Online prediction error of the custom version of the algorithm on MH01.*



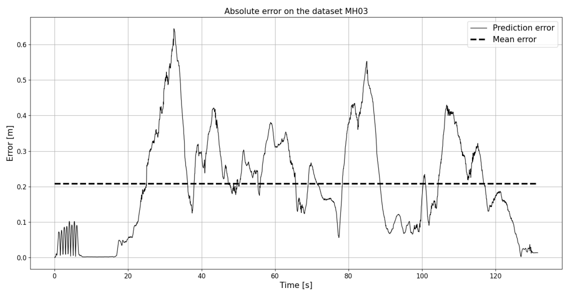
(b) *Online prediction error of the original version of the algorithm on MH01.*



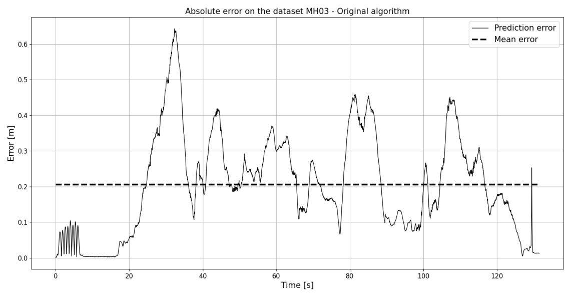
(c) *Online prediction error of the custom version of the algorithm on MH02.*



(d) *Online prediction error of the original version of the algorithm on MH02.*

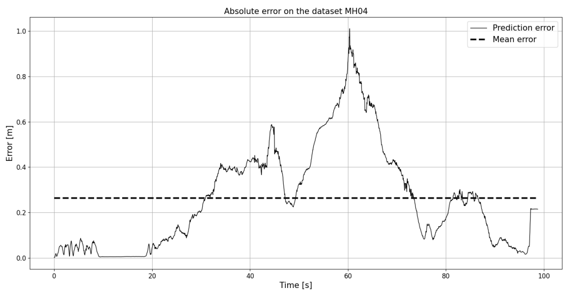


(e) *Online prediction error of the custom version of the algorithm on MH03.*

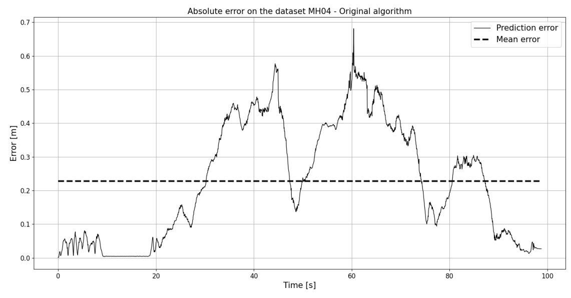


(f) *Online prediction error of the original version of the algorithm on MH03.*

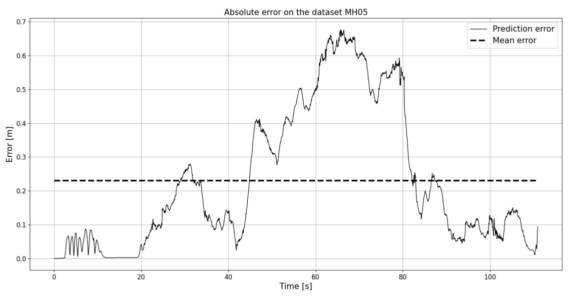
Figure 6.3: *Errors comparison of the simulation in the datasets MH01 (easy), MH02 (easy), MH03 (medium).*



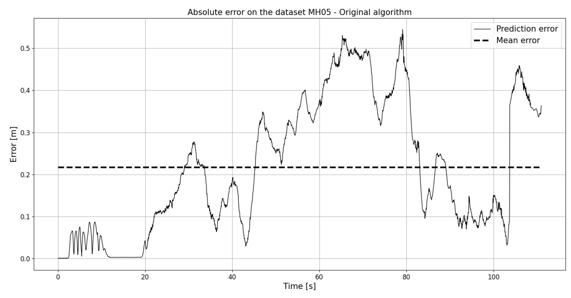
(a) *Online prediction error of the custom version of the algorithm on MH04.*



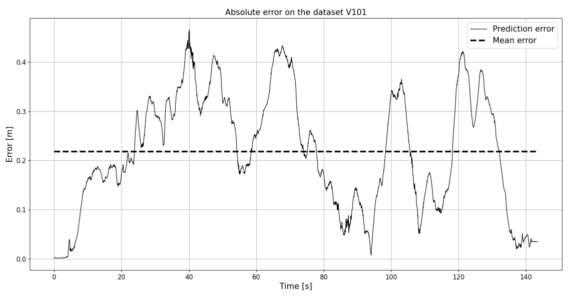
(b) *Online prediction error of the original version of the algorithm on MH04.*



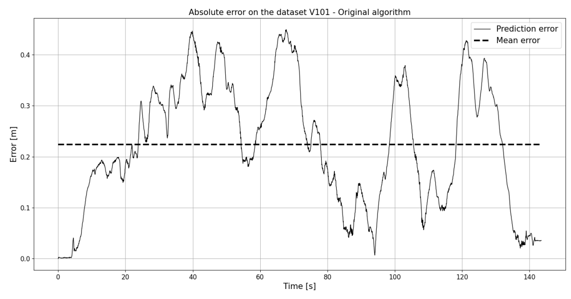
(c) *Online prediction error of the custom version of the algorithm on MH05.*



(d) *Online prediction error of the original version of the algorithm on MH05.*

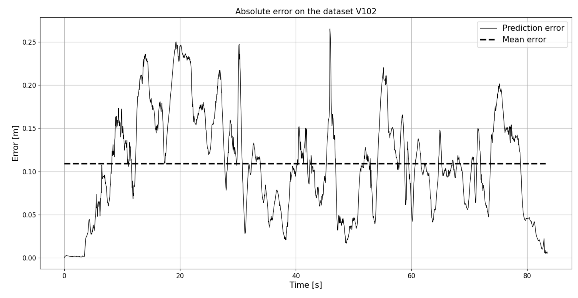


(e) *Online prediction error of the custom version of the algorithm on V101.*

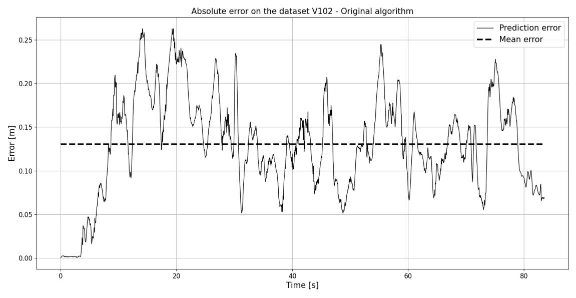


(f) *Online prediction error of the original version of the algorithm on V101.*

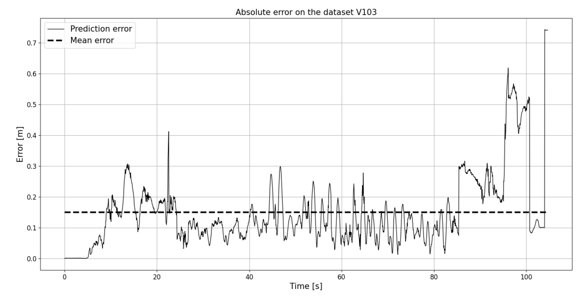
Figure 6.4: *Errors comparison of the simulation in the datasets MH04 (difficult), MH05 (difficult), V101 (easy).*



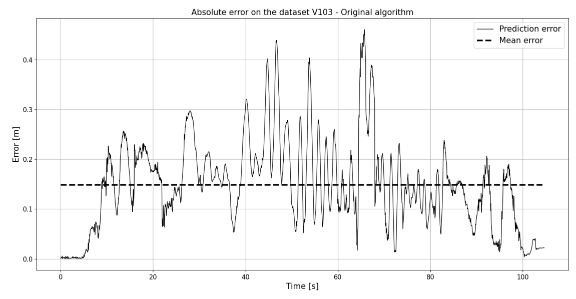
(a) *Online prediction error of the custom version of the algorithm on V102.*



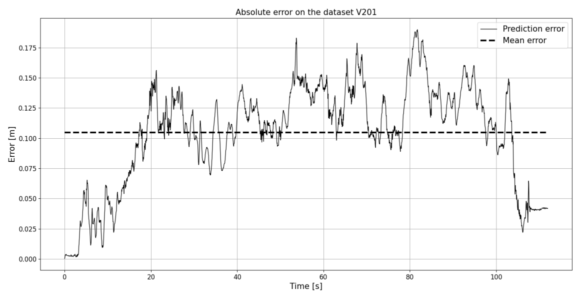
(b) *Online prediction error of the original version of the algorithm on V102.*



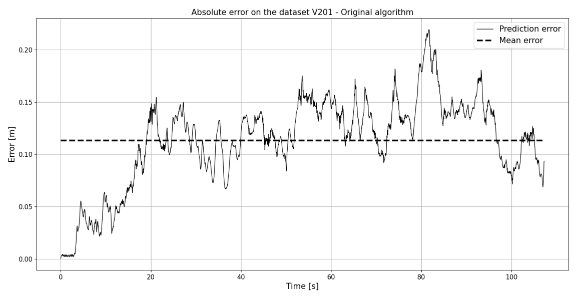
(c) *Online prediction error of the custom version of the algorithm on V103.*



(d) *Online prediction error of the original version of the algorithm on V103.*



(e) *Online prediction error of the custom version of the algorithm on V201.*



(f) *Online prediction error of the original version of the algorithm on V201.*

Figure 6.5: *Errors comparison of the simulation in the datasets V102 (medium), V103 (difficult), V201 (easy).*

Machine Hall 01 - easy		
-	Original	Custom
Rectification	1.90 ± 0.55	1.96 ± 1.03
ORB extraction	30.63 ± 4.41	30.82 ± 4.40
Stereo matching	15.81 ± 5.60	1.67 ± 0.37
Pose prediction	3.32 ± 1.21	3.37 ± 0.78
Local map tracking	16.11 ± 9.50	11.62 ± 5.84
New KF decision	0.073 ± 0.17	0.078 ± 0.033
New KF creation	0.90 ± 0.35	0.40 ± 0.085
Response time	68.58 ± 18.74	47.16 ± 11.4
KF insertion	26.85 ± 10.81	22.38 ± 8.39
Map points culling	0.23 ± 0.12	0.34 ± 0.26
Map points triangulation	9.73 ± 4.67	7.75 ± 4.19
Map points fusion	79.6 ± 23.5	42.7 ± 15.46
Local BA	246.85 ± 201.88	202.46 ± 193.3
KF culling	12.41 ± 10.18	16.94 ± 12.35
Loop detection	13.12 ± 9.84	15.13 ± 11.48
SE3 detection	-	-
SE3 computation	-	-
Loop correction	-	-
Essential graph optimization	-	-
Global BA	-	-
Graph update	-	-

Table 6.2: *Table representing the time comparison obtained on the set MH01. All the times are measured in ms (mean $\pm$ std. deviation).*

Machine Hall 02 - easy		
-	Original	Custom
Rectification	1.88 ± 0.59	1.93 ± 0.54
ORB extraction	30.90 ± 4.85	30.49 ± 4.35
Stereo matching	16.01 ± 5.60	1.68 ± 0.33
Pose prediction	3.27 ± 1.13	3.2 ± 1.03
Local map tracking	14.85 ± 9.50	10.26 ± 6.45
New KF decision	0.072 ± 0.12	0.083 ± 0.074
New KF creation	0.94 ± 0.39	0.43 ± 0.13
Response time	67.94 ± 18.12	46.62 ± 8.66
KF insertion	26.30 ± 12.71	22.84 ± 13.02
Map points culling	0.25 ± 0.20	0.32 ± 0.17
Map points triangulation	10.03 ± 5.03	7.34 ± 4.43
Map points fusion	76.13 ± 24.5	39.04 ± 20.34
Local BA	254.96 ± 208.36	170.19 ± 158.81
KF culling	11.17 ± 10.48	15.401 ± 16.61
Loop detection	12.99 ± 8.98	15.26 ± 11.90
SE3 detection	-	-
SE3 computation	-	-
Loop correction	-	-
Essential graph optimization	-	-
Global BA	-	-
Graph update	-	-

Table 6.3: *Table representing the time comparison obtained on the set MH02. All the times are measured in ms (mean $\pm$ std. deviation).*

Machine Hall 03 - medium		
-	Original	Custom
Rectification	1.91 ± 0.55	2.03 ± 0.71
ORB extraction	30.9 ± 5.32	29.95 ± 5.45
Stereo matching	17.35 ± 5.74	1.73 ± 0.34
Pose prediction	3.21 ± 1.04	3.2 ± 1.02
Local map tracking	16.32 ± 8.43	11.32 ± 5.85
New KF decision	0.076 ± 0.13	0.086 ± 0.082
New KF creation	0.91 ± 0.38	0.45 ± 0.14
Response time	69.56 ± 18.41	47.41 ± 9.5
KF insertion	24.45 ± 10.53	20.10 ± 10.52
Map points culling	0.24 ± 0.13	0.35 ± 0.19
Map points triangulation	10.01 ± 7.72	4.85 ± 4.43
Map points fusion	81.12 ± 28.51	38.31 ± 17.51
Local BA	281.37 ± 305.51	190.31 ± 230.45
KF culling	12.11 ± 12.35	14.91 ± 15.11
Loop detection	12.54 ± 7.5	14.61 ± 9.41
SE3 detection	-	-
SE3 computation	-	-
Loop correction	-	-
Essential graph optimization	-	-
Global BA	-	-
Graph update	-	-

Table 6.4: *Table representing the time comparison obtained on the set MH03. All the times are measured in ms (mean ± std. deviation).*

Machine Hall 04 - difficult		
-	Original	Custom
Rectification	1.88 ± 0.51	1.96 ± 0.62
ORB extraction	27.51 ± 4.35	29.95 ± 5.45
Stereo matching	17.93 ± 5.74	1.73 ± 0.34
Pose prediction	2.78 ± 1.04	2.84 ± 0.98
Local map tracking	11.58 ± 7.78	7.29 ± 5.85
New KF decision	0.055 ± 0.28	0.070 ± 0.029
New KF creation	0.8 ± 0.37	0.39 ± 0.16
Response time	61.43 ± 18.41	41.2 ± 6.51
KF insertion	19.11 ± 9.15	13.77 ± 6.15
Map points culling	0.26 ± 0.15	0.32 ± 0.27
Map points triangulation	12.82 ± 6.87	7.82 ± 6.57
Map points fusion	63.52 ± 22.31	26.32 ± 11.43
Local BA	211.21 ± 153.11	156.89 ± 150.39
KF culling	4.56 ± 4.67	7.23 ± 6.94
Loop detection	5.62 ± 3.45	8.49 ± 5.35
SE3 detection	-	26.18
SE3 computation	-	0.120
Loop correction	-	145.54
Essential graph optimization	-	511.66
Global BA	-	$5.99 \cdot 10^3$
Graph update	-	120.55

Table 6.5: *Table representing the time comparison obtained on the set MH04. All the times are measured in ms (mean ± std. deviation).*

Machine Hall 05 - difficult		
-	Original	Custom
Rectification	1.89 ± 0.59	1.99 ± 0.59
ORB extraction	28.1 ± 4.55	27.80 ± 4.32
Stereo matching	18.32 ± 5.47	1.81 ± 0.35
Pose prediction	3.09 ± 0.98	2.9 ± 0.98
Local map tracking	11.13 ± 6.06	8.11 ± 4.35
New KF decision	0.073 ± 0.081	0.073 ± 0.041
New KF creation	0.79 ± 0.33	0.41 ± 0.14
Response time	63.53 ± 15.42	41.83 ± 8.37
KF insertion	21.01 ± 9.07	18.92 ± 12.95
Map points culling	0.25 ± 0.13	0.33 ± 0.19
Map points triangulation	11.98 ± 6.72	7.51 ± 6.2
Map points fusion	69.68 ± 25.65	33.21 ± 16.52
Local BA	226.50 ± 187.48	185.45 ± 220.21
KF culling	6.37 ± 7.20	9.85 ± 9.34
Loop detection	7.34 ± 4.34	10.56 ± 6.37
SE3 detection	17.46	10.43
SE3 computation	1.11	0.038
Loop correction	825.3	630.32
Essential graph optimization	653.6	677.9
Global BA	$14.417 \cdot 10^3$	$7.17 \cdot 10^3$
Graph update	202.65	82.66

Table 6.6: *Table representing the time comparison obtained on the set MH05. All the times are measured in ms (mean $\pm$ std. deviation).*

Vicon room 1 - easy		
-	Original	Custom
Rectification	1.82 ± 0.45	1.89 ± 0.54
ORB extraction	26.65 ± 3.63	26.78 ± 4.07
Stereo matching	15.66 ± 3.55	1.68 ± 0.35
Pose prediction	2.65 ± 0.92	2.82 ± 1.01
Local map tracking	8.87 ± 4.03	5.63 ± 1.91
New KF decision	0.059 ± 0.024	0.076 ± 0.048
New KF creation	2.02 ± 0.78	0.78 ± 0.26
Response time	54.69 ± 8.11	37.7 ± 5.28
KF insertion	13.54 ± 3.46	9.41 ± 1.95
Map points culling	0.44 ± 0.24	0.75 ± 0.41
Map points triangulation	6.97 ± 4.57	6.75 ± 3.18
Map points fusion	61.71 ± 26.37	20.58 ± 8.15
Local BA	147.13 ± 100.87	122.82 ± 79.91
KF culling	4.72 ± 4.58	6.3 ± 4.93
Loop detection	3.58 ± 2.18	4.01 ± 1.55
SE3 detection	-	-
SE3 computation	-	-
Loop correction	-	-
Essential graph optimization	-	–
Global BA	-	-
Graph update	-	-

Table 6.7: *Table representing the time comparison obtained on the set V101. All the times are measured in ms (mean ± std. deviation).*

Vicon room 1 - medium		
-	Original	Custom
Rectification	1.89 ± 0.51	1.97 ± 0.55
ORB extraction	26.98 ± 4.48	26.24 ± 4.31
Stereo matching	16.88 ± 5.82	1.74 ± 0.36
Pose prediction	2.88 ± 1.27	2.84 ± 1.24
Local map tracking	10.17 ± 5.71	5.68 ± 2.65
New KF decision	0.068 ± 0.036	0.081 ± 0.037
New KF creation	2.19 ± 0.99	0.782 ± 0.28
Response time	58.31 ± 12.16	37.33 ± 6.08
KF insertion	10.86 ± 4.34	9.44 ± 2.49
Map points culling	0.51 ± 0.36	0.83 ± 0.70
Map points triangulation	7.56 ± 3.91	6.72 ± 3.25
Map points fusion	67.17 ± 31.00	20.58 ± 10.51
Local BA	181.57 ± 137.24	123.11 ± 98.89
KF culling	4.77 ± 4.50	5.00 ± 5.34
Loop detection	5.45 ± 2.18	5.34 ± 3.98
SE3 detection	18.48	15.81
SE3 computation	1.325	0.11
Loop correction	85.62	60.69
Essential graph optimization	138.20	68.78
Global BA	741.41	321.45
Graph update	27.51	43.71

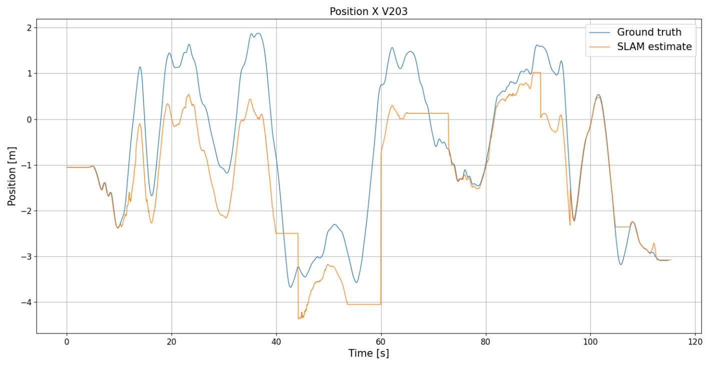
Table 6.8: *Table representing the time comparison obtained on the set V102. All the times are measured in ms (mean $\pm$ std. deviation).*

Vicon room 1 - difficult		
-	Original	Custom
Rectification	1.91 ± 0.54	2.04 ± 0.63
ORB extraction	25.38 ± 4.40	26.32 ± 5.01
Stereo matching	16.25 ± 5.71	1.75 ± 0.45
Pose prediction	2.65 ± 1.17	2.72 ± 1.12
Local map tracking	9.84 ± 5.05	5.49 ± 2.64
New KF decision	0.064 ± 0.037	0.083 ± 0.039
New KF creation	2.29 ± 1.22	0.780 ± 0.31
Response time	55.52 ± 12.77	37.14 ± 6.63
KF insertion	11.93 ± 4.32	7.61 ± 2.40
Map points culling	0.53 ± 0.28	0.69 ± 0.45
Map points triangulation	7.04 ± 4.87	5.04 ± 4.00
Map points fusion	61.04 ± 31.55	18.82 ± 12.72
Local BA	156.34 ± 131.37	108.54 ± 106.65
KF culling	5.09 ± 5.26	4.98 ± 4.88
Loop detection	3.94 ± 2.42	4.05 ± 3.12
SE3 detection	20.06	13.16
SE3 computation	0.98	0.026
Loop correction	92.84	64.06
Essential graph optimization	108.47	90.49
Global BA	978.32	$1.25 \cdot 10^3$
Graph update	30.19	27.95

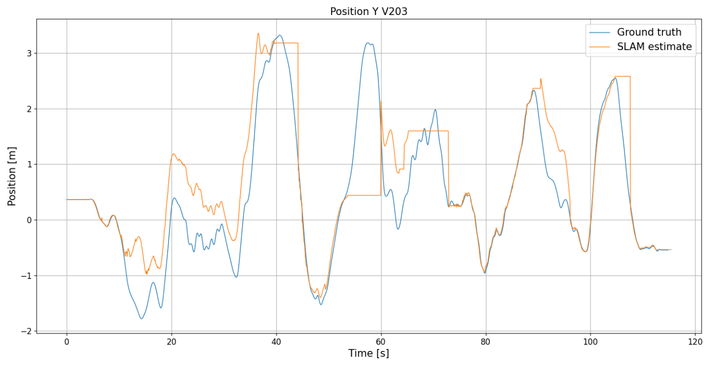
Table 6.9: *Table representing the time comparison obtained on the set V103. All the times are measured in ms (mean $\pm$ std. deviation).*

Vicon room 2 - easy		
-	Original	Custom
Rectification	1.81 ± 0.45	2.04 ± 0.63
ORB extraction	25.92 ± 3.69	27.20 ± 5.01
Stereo matching	16.24 ± 4.33	1.77 ± 0.41
Pose prediction	2.75 ± 1.05	3.12 ± 1.20
Local map tracking	6.19 ± 2.34	4.95 ± 1.62
New KF decision	0.055 ± 0.021	0.079 ± 0.025
New KF creation	1.96 ± 0.72	0.73 ± 0.23
Response time	51.89 ± 7.51	37.77 ± 5.75
KF insertion	12.87 ± 3.97	8.53 ± 2.32
Map points culling	0.55 ± 0.34	0.57 ± 0.31
Map points triangulation	8.22 ± 4.01	5.90 ± 3.27
Map points fusion	55.043 ± 24.02	18.24 ± 8.32
Local BA	86.51 ± 55.48	76.26 ± 52.05
KF culling	2.05 ± 1.85	3.00 ± 2.57
Loop detection	2.21 ± 1.30	3.09 ± 1.58
SE3 detection	-	-
SE3 computation	-	-
Loop correction	-	-
Essential graph optimization	-	-
Global BA	-	-
Graph update	-	-

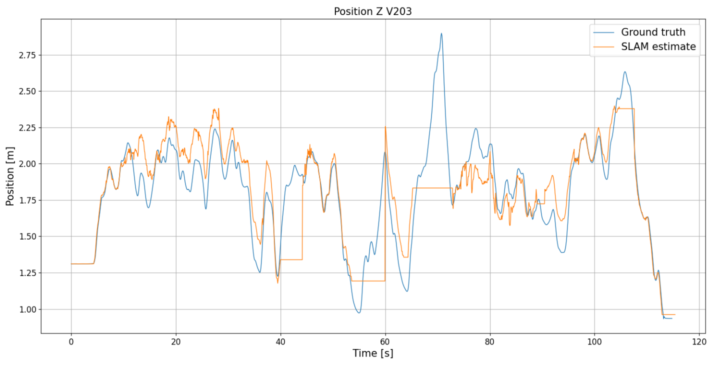
Table 6.10: *Table representing the time comparison obtained on the set V201. All the times are measured in ms (mean ± std. deviation).*



(a) *Simulation result of coordinate X.*



(b) *Simulation result of coordinate Y.*



(c) *Simulation result of coordinate Z.*

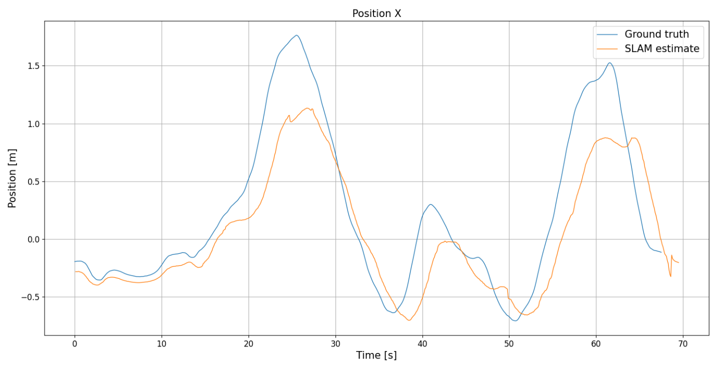
Figure 6.6: *Simulation result on the dataset V203 (difficult).*

Vicon room 2 - difficult		
-	Original	Custom
Rectification	1.92 ± 0.58	2.02 ± 0.66
ORB extraction	25.90 ± 4.74	24.72 ± 4.46
Stereo matching	17.56 ± 5.72	1.67 ± 0.44
Pose prediction	2.72 ± 1.19	2.68 ± 1.20
Local map tracking	7.88 ± 4.06	4.95 ± 2.06
New KF decision	0.062 ± 0.071	0.071 ± 0.035
New KF creation	2.29 ± 1.16	0.75 ± 0.23
Relocalization time	7.21 ± 9.89	6.12 ± 1.76
Response time	55.05 ± 12.78	34.65 ± 6.34
KF insertion	9.31 ± 3.26	8.67 ± 2.50
Map points culling	0.53 ± 0.43	0.75 ± 0.31
Map points triangulation	7.99 ± 5.01	6.51 ± 3.82
Map points fusion	53.22 ± 27.34	18.40 ± 10.98
Local BA	87.98 ± 88.27	86.72 ± 77.78
KF culling	2.51 ± 4.50	4.07 ± 4.54
Loop detection	2.55 ± 1.11	3.19 ± 1.89
SE3 detection	24.66	10.21
SE3 computation	1.19	0.032
Loop correction	119.98	28.63
Essential graph optimization	192.80	63.02
Global BA	$1.37 \cdot 10^3$	618.76
Graph update	63.84	47.21

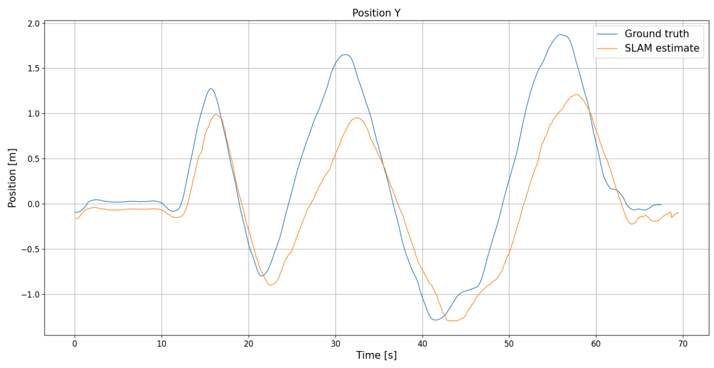
Table 6.11: *Table representing the time comparison obtained on the set V203. All the times are measured in ms (mean $\pm$ std. deviation).*

6.3 Real setup

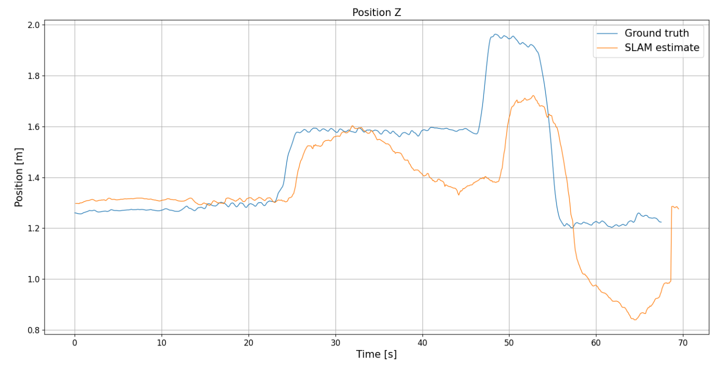
In order to fully test the applicability of the algorithm in a real platform, a test on a real quadcopter has been performed. The vehicle considered is equipped with a Jetson Xavier NX. The test consisted of a simple exploration in a small environment, while the ground-truth position was provided by a Vicon system.
The main problem in the area explored was the lack of a rich feature environment, which negatively impacted the algorithm behavior, especially in certain areas. The result of the simulation can be seen in the figure 6.7, with the time reported in the table 6.12. As it can be seen, the algorithm has a worse behavior with respect to the simulation performed on the EuRoC dataset. The main problem, as it can be seen by the computation time, is the lag between the image captured by the camera and the position returned by the drone: given a camera frame rate of $30fps$, the average response time is about three times longer. This is highlighted by the horizontal drift in the plots of all three coordinates. However, despite the performance degradation, the algorithm runs without ever losing the state and always following the general drone behavior. More importantly, as it can be seen from the last time instants, when the vehicle returned to the initial position, it detected a loop closure and performed the correction, resulting so in a substantial decrease of the accumulated drift.



(a) *Result of coordinate X.*



(b) *Result of coordinate Y.*



(c) *Result of coordinate Z.*

Figure 6.7: *Outcome of the algorithms executed on a real embedded device.*

Real setup	
ORB extraction	56.14 ± 10.87
Stereo matching	1.10 ± 0.12
Pose prediction	5.24 ± 1.79
Local map tracking	18.38 ± 12.66
New KF decision	0.17 ± 0.077
New KF creation	1.02 ± 0.23
Response time	89.89 ± 23.94
KF insertion	19.58 ± 11.30
Map points culling	0.58 ± 0.29
Map points triangulation	10.45 ± 5.91
Map points fusion	48.23 ± 22.12
Local BA	137.64 ± 370.0
KF culling	21.60 ± 43.81
Loop detection	13.84 ± 13.70
SE3 detection	86.83
SE3 computation	0.31
Loop correction	$1.42 \cdot 10^3$
Essential graph optimization	$1.30 \cdot 10^3$
Global BA	-
Graph update	-

Table 6.12: *Table representing the computation time obtained during the real experiment. All the times are measured in ms (mean $\pm$ std. deviation).*

Conclusion

The work of this thesis was to undertake a comprehensive analysis of the ORB-SLAM2 algorithm, by suggesting ways to obtain a more optimized code, requiring less computational power to achieve the same accuracy as the original one.
The proposed optimization resulted in an effective and substantial time reduction concerning the original algorithm while keeping an almost equal accuracy. Despite the non-optimal result obtained with the quadcopter, the more general goal of achieving an algorithm with the same accuracy as the original one, but with a lower computational demand, has been accomplished.
In conclusion, this study highlighted the possibilities of optimization of the ORB-SLAM2 code, by pushing forward its applicability in real-time scenarios with low computational power at its disposal.
To ensure reproducibility and transparency, the new version of the code has been made available on GitHub[1], with the hope that this work will push forward the research on the implementation of embedded real-time SLAM algorithm.

[1]Available at the link https://github.com/Luigi940260/orb-slam2-optimized.git

Appendix A

Rotation manifold

In this appendix a brief explanation about rotation manifold theory will be presented, bearing in mind that the purpose of this dissertation is not to provide an exhaustive coverage of this field.

A.1 SO(3) group

To better explain the argument, the definition of *group* from algebra is first recalled: a group is a non-empty set $\mathcal{G}$ with an associated binary operation, satisfying the following properties:

- The set is closed under the associated operation. In other words, if A and B belong to the group, also $A \cdot B$, where $\cdot$ is a symbol for the operation involved, must be inside the group.
- The operation is associative: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$, for any $A, B, C \in \mathcal{G}$.
- The identity element associated with the operation exists inside the set: $\exists I \in \mathcal{G}$ such that $A \cdot I = A, \quad \forall A \in \mathcal{G}$.
- For every element of the set, an inverse element with respect to the operation exists inside the set itself: $\forall A \in \mathcal{G}, \quad \exists A^{-1} : A \cdot A^{-1} = I$.

By stating that, is then obvious to see the set of all 3D rotations about the origin, belonging to the Euclidean space $\mathbb{R}^3$, as a group under the operation of composition. More precisely, given the *orthogonal group* $O(n)$ (whose elements are linear transformations that preserve orthogonality between two vectors), the rotations are defined as *special orthogonal group* $SO(3)$, whose elements are all direct isometries, transformations preserving both euclidean distances and orientation of the space.
More in detail, the rotation group is in fact a *Lie-group*, being both the composition operation and its inverse differentiable. From this property, $SO(3)$

has a three-dimensional manifold structure, that so cannot be represented directly into Euclidean space. This leads, as it could be expected, to several problems for their representations.
More precisely, despite $SO(3) \in \mathbb{R}^3$, a direct representation of rotation in an Euclidean vector space in $\mathbb{R}^3$ doesn't exist. More precisely, recalling the Euler's rotation theorem, it can be proven that every three-dimensional rotation can be described as a single rotation of a certain angle around a fixed axis, so at least an $\mathbb{R}^4$ quantity must be used (another example of this are quaternions).
This type of representation, however, is of course redundant with respect to the problem (in both the angle-axis and the quaternion representation only three variables are actually independent), and it leads to some singularity. For example, in the angle-axis representation, every rotation about a direction leads to the exact same result of making a rotation of the inverse angle around an axis in the opposite direction. Also, any rotation of $\theta + 2k\pi$, with k an integer number, is equivalent to the rotation θ.
This type of redundancy also happens with quaternions since every quaternion q represents the same rotation as its opposite $-q$. Other representations are often used, like for example roll, pitch and yaw, and rotation matrices.
Roll, pitch and yaw representation, despite requiring a minimal number of parameters, suffers from severe singularities problems, like for example gimbal lock (happening when two of the three rotation axes are aligned). Rotation matrices have instead nice analytical properties (a rotation composition is reduced to a simple matrix multiplication) but are more redundant than quaternions (living in $\mathbb{R}^{3\times3}$).
To better understand the property of $SO(3)$, the topological space of the rotations can be visualized as a 3D sphere centered in the origin, where each point inside and on it represents a different rotation. More precisely, by taking a sphere of radius π, every point can be seen as a rotation around the axes connecting it with the sphere center, whose amplitude is defined by its distance from the origin. This visualization brings however the ambiguity of two antipodal points representing the same rotation: these points must then be virtually "connected" and considered exactly as a unique point. This is basically the reason why $SO(3)$ is a connected space, but not simply connected (because every arc that connects two antipodal points is in fact a loop, but it cannot be shrunk or deformed into a single point).

A.2 Lie algebra

Given the previous explanation, it's then clear why $SO(3)$ isn't indeed an Euclidean space. Fortunately, $SO(3)$ has been proven to be a Lie-group, and so a 3-dimensional manifold. As every manifold, each point of $SO(3)$ has a neigh-

borhood that is homeomorphic (so a bijective and continuous map between the two exits) to an open subset of the 3-dimensional Euclidean space. This tangent space is called *Lie algebra* and, as far as concerns the $SO(3)$ group, is denoted $\mathfrak{so}(3)$. By choosing to represent rotation with matrices (any rotation representation can be easily converted into another) the $\mathfrak{so}(3)$ coincides with the space of the 3×3 skew-symmetric matrices.
Since a linear space $\mathfrak{s}$ (over a field F) can be defined as a Lie algebra only if a binary operation, called Lie bracket $[\cdot, \cdot]$, exists and has the following property:

- Bilinearity:

$$\begin{matrix} [ax + by, z] = a[x, z] + b[y, z] \\ [x, az + by] = a[x, z] + b[x, y] \end{matrix}, \quad \forall x, y, z \in \mathfrak{s}, \quad \forall a, b \in F. \tag{A.1}$$

- Alternativity: $[x, x] = 0, \quad \forall x \in \mathfrak{s}$.
- Jacobi identity: $[x, [y, z]] + [y, [z, x]] + [z, [x, y]] = 0, \quad \forall x, y, x \in \mathfrak{s}$.

The first and second conditions together implied also the anticommutativity property that $[x, y] = -[y, x], \quad \forall x, y \in \mathfrak{s}$.
As far as concerns the $SO(3)$ group and its associated $\mathfrak{so}(3)$, the above properties can all be verified defining, as done for every matrix group, the Lie bracket as:

$$[R_1, R_2] = R_1 R_2 - R_2 R_1. \tag{A.2}$$

whose result is, of course, a skew-symmetric matrix itself, guaranteeing also the closeness of the operation.
This space of skew-symmetric matrices is homeomorphic to the Euclidean vector space $\mathbb{R}^3$. This can be easily proven because every skew-symmetric matrix can be associated with a vector $\boldsymbol{v} \in \mathbb{R}^3$ as:

$$\boldsymbol{v}^\wedge = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}^\wedge = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_2 & 0 \end{bmatrix} \tag{A.3}$$

.
The mathematical relationship that maps elements from the Euclidean tangent space to its associated subset in the manifold is called *exponential map*. As far as concerns rotation, the map associating each element of the Lie algebra to a rotation can be defined as:

$$\exp(\phi^\wedge) = I + \frac{\sin(||\phi||)}{||\phi||}\phi^\wedge + \frac{1 - \cos(||\phi||)}{||\phi||^2}\left(\phi^\wedge\right)^2, \quad \exp : \mathcal{R}^3 \to SO(3) \tag{A.4}$$

where $\phi^\wedge$ is the skew-symmetric matrix associated to the vector ϕ. The (A.4) is basically equivalent to the standard Rodrigues' formula. Also the opposite *logarithm map* can be defined, associating a rotation matrix to a skew-symmetric matrix (and so, by the (A.3), to a vector in $\mathbb{R}^3$):

$$\ln R = \frac{\varphi(R - R^T)}{2\sin\varphi}, \quad \varphi = \cos^{-1}\left(\frac{\text{tr}(R) - 1}{2}\right). \tag{A.5}$$

The result of this logarithmic map is a skew-symmetric matrix that can be directly related to a rotation, by noting that:

$$\ln R = (\boldsymbol{v}\phi)^\wedge, \tag{A.6}$$

where $\boldsymbol{v}$ and ϕ are the axis and the angle associated with the rotation. Of course, for the reasons stated before, the exponential map is a bijection only if rotations are restricted to have $||\phi|| < \pi$, while in every other case multiple $\phi^\wedge$ would be mapped into the same rotation.

A.3 SE(3) group

A more general group whose existence is fundamental for a great variety of tasks is the Euclidean group $E(n)$: this group is composed of all the Euclidean isometries happening in the Euclidean space (transformation preserving Euclidean distance). By restricting it by eliminating all the reflections (that are indeed Euclidean isometries), the special Euclidean group $SE(n)$ is obtained, whose elements are all the type of rigid motions happening in an n-dimensional space.
Fortunately, $E(n)$ is also a Lie-group and, moreover, it is a subgroup of the affine group (all transformations that transform a line into another line). In fact, every element of $E(n)$ can be expressed in two different ways:

- A pair of elements $R, \boldsymbol{r}$ where $R \in \mathbb{R}^{n\times n}$ is an orthogonal matrix and $\boldsymbol{t} \in \mathbb{R}^n$.
- A single matrix $T \in \mathbb{R}^{(n+1\times(n+1))}$, that is basically of the form:

$$T = \begin{bmatrix} R & \boldsymbol{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{A.7}$$

  where of course $\mathbf{0} \in \mathbb{R}^3$ is a row vector and 1 is a scalar.

As far as concerning the $SE(n)$ group, the same representations are valid, by restricting the type of orthogonal matrices R to the ones having determinant strictly equal to 1 (so by eliminating reflections).

For what concerns the Lie algebra of $SE(3)$, the group of rigid body motions in the standard 3D space, the same reasoning applied for the $SO(3)$ is valid. This is due to the possibility of decoupling the rotational description to the translational one, by adopting a similar mapping as in the (A.4) for the rotation part and by taking into account that translations are already living in a Euclidean space.

Bibliography

[1] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM: A versatile and accurate monocular SLAM system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[2] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[3] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM3: An accurate open-source library for visual, visual–inertial, and multimap SLAM," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[4] H. Choset and J. Burdick, "Sensor based planning. I. the generalized Voronoi graph," *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, vol. 2, pp. 1649–1655, 1995.

[5] H. U. Takafumi Taketomi and S. Ikeda, "Visual SLAM algorithms: a survey from 2010 to 2016," *IPSJ Transactions on ComputerVision and Applications*, 2017.

[6] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse, "MonoSLAM: Real-time single camera SLAM," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1052–1067, 2007.

[7] G. Klein and D. Murray, "Parallel tracking and mapping for small AR workspaces," *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pp. 225–234, 2007.

[8] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison, "DTAM: Dense tracking and mapping in real-time," *2011 International Conference on Computer Vision*, pp. 2320–2327, 2011.

[9] "LSD-SLAM: Large-scale direct monocular SLAM," *Computer Vision – ECCV 2014*, pp. 834–849, 2014.

[10] J. Kannala and S. Brandt, "A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, pp. 1335–1340, 2006.

[11] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," *2011 International Conference on Computer Vision*, pp. 2564–2571, 2011.

[12] F. Lu and E. Milios, "Globally consistent range scan alignment for environment mapping," *Autonom. Robots*, vol. 4, pp. 333–349, 1997.

[13] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A Tutorial on Graph-Based SLAM," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[14] D. Galvez-Lopez and J. D. Tardos, "Bags of binary words for fast place recognition in image sequences," *IEEE Transactions on Robotics*, vol. 28, no. 5, 2012.

[15] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2, pp. 2161–2168, 2006.

[16] Sivic and Zisserman, "Video google: a text retrieval approach to object matching in videos," *Proceedings Ninth IEEE International Conference on Computer Vision*, vol. 2, pp. 1470–1477, 2003.

[17] B. K. P. Horn, "Closed-form solution of absolute orientation using unit quaternions," *Journal of the Optical Society of America*, vol. 4, 1987.

[18] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, p. 381–395, 1981.

[19] V. Lepetit, F. Moreno-Noguer, and P. Fua, "EPnP: An accurate O(n) solution to the PnP problem," *International Journal of Computer Vision*, vol. 81, 2009.

[20] F. D. Christian Forster, Luca Carlone and D. Scaramuzza, "On-manifold preintegration for real-time visual–inertial odometry," *IEEE Transactions on Robotics*, vol. 33, no. 1, 2017.

[21] M. B. C. S. Armin Hornung, Kai M. Wurm and W. Burgard, "OctoMap: an efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, 2013.

[22] D. Meagher, “Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-D objects by computer,” 1980.

[23] J. Zhang, S. Liu, B. Gao, and C. Zhong, “An improvement algorithm for octomap based on RGB-D SLAM,” *2018 Chinese Control And Decision Conference (CCDC)*, pp. 5006–5011, 2018.

[24] H. Zhang, C. Xu, and J. Gu, “Dense reconstruction from visual SLAM with probabilistic multi-sequence merging,” *2022 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 33–40, 2022.

[25] G. P. S. T. R. J. O. S. A. M. S. R. Burri M., Nikolic J., “The EuRoC micro aerial vehicle datasets,” *The International Journal of Robotics Research*, vol. 35, pp. 1157–1163, 2016.

[26] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, “G2o: A general framework for graph optimization,” *2011 IEEE International Conference on Robotics and Automation*, pp. 3607–3613, 2011.