UNIVERSITÀ DEGLI STUDI DI PADOVA

UNIVERSITA' DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria Industriale DII

Corso di Laurea Magistrale in Ingegneria Aerospaziale

Analysis of a Stereo Visual SLAM System in a Virtual and Real Environment

Relatore: Prof. Marco Pertile

Laureando: Davide Sabbadin

Matricola: 1222670

Anno Accademico 2022/2023

Abstract

Simultaneous Localization And Mapping (SLAM) is a system that allows a robot, or generic mobile device, to construct a map of its surroundings and at the same time define its pose (position and rotation), without a priori information. In particular, when this process is performed solely through the use of cameras, it is called Visual-SLAM. In this thesis, the Visual-SLAM ORB-SLAM2 system is implemented with the ultimate goal of demonstrating the validity of employing virtual simulations as a testing environment. Two constituent blocks can be identified in the paper. The first part focuses first on the introduction of the SLAM problem, from its probabilistic formulation to the general characteristics of a method based on vision systems, and then moves on to a detailed description of the used implementation of the ORB-SLAM2 system. The intention is to provide a complete and in-depth map of the algorithm: from input image processing to loop closure recognition and subsequent optimization. The second part shows how the framework was prepared to carry out the simulations and what results were obtained. A series of tests demonstrate the usefulness of this tool for quantifying the effect of some typical conditions to which a Visual-SLAM system is subjected, such as the presence of dynamic objects, adverse lighting conditions, and high speed of the camera motion. A comparison is then presented between the outcome of the test performed using the KITTI public dataset and the outcome of the tests done in the virtual environments.

Sommario

La localizzazione e mappatura simultanea (Simultaneous Localization And Mapping, SLAM) è un sistema che permette ad un robot, o generico dispositivo mobile, di costruire una mappa dell'ambiente circostante e allo stesso tempo di definire la propria posa (posizione e rotazione), senza informazioni a priori. In particolare, quando questo processo viene realizzato solamente mediante l'uso di telecamere, esso prende il nome di Visual-SLAM. In questa tesi viene implementato il sistema Visual-SLAM ORB-SLAM2 con lo scopo ultimo di dimostrare la validità di impiego di simulazioni virtuali come ambiente di testing. Possono essere individuati due blocchi costituitivi nell'elaborato. La prima parte si concentra prima sull'introdurre il problema SLAM, dalla sua formulazione probabilistica alle caratteristiche generali di un metodo basato su sistemi di visione, per poi spostarsi ad una descrizione dettagliata della implementazione utilizzata del sistema ORB-SLAM2. L'intenzione è di fornire una completa e approfondita mappa dell'algoritmo: dalla elaborazione delle immagini in input fino al riconoscimento della chiusura del loop e successiva ottimizzazione. La seconda parte illustra come è stato preparato il framework per effettuare le simulazioni e quali sono i risultati ottenuti. Una serie di test dimostra l'utilità di questo strumento per quantificare l'effetto di alcune condizioni tipiche a cui è soggetto un sistema Visual-SLAM, quali la presenza di oggetti dinamici, condizioni di illuminazione ed elevata velocità del moto delle telecamere. Viene poi presentato un confronto tra l'esito del test sostenuto utilizzando il dataset pubblico KITTI e l'esito dei test fatti negli ambienti virtuali.

Contents

Chapter 1

Introduction

1.1 What is SLAM?

Simultaneous Localization and Mapping (SLAM) allows a robot, or a generic moving rigid body, to build a model of the surrounding environment while at the same time define within that environment its pose, without a *priori* information. Different types of sensors can be relied upon to do this, each with its own advantages and disadvantages. SLAM therefore aspires to simultaneously solve two complex and heavily interrelated aspects, resulting in a chicken-and-egg situation: a map of the surrounding environment is needed for accurate localization, however a pose estimate is needed for accurate mapping.
Conceptually, this problem is not at all difficult for a human being to understand and perform; it is something we learn to do from the moment of birth. Take for example the case when we enter a room unfamiliar to us for the first time. To orient and move around in this new environment we are unconsciously performing some kind of SLAM, where our eyes are the sensors and the information they provide is interpreted by the brain in such a way that allows us to know our position, what direction we are facing and even at what speed we are moving. Thanks to these information, we can not only know our pose at any given instant, but we can also safely navigate the room without bumping into something as well as interact with objects around us. At the same time, depending on our position, we are also understanding the surroundings and creating a mental image, a *map*, of the place that is then stored in our memory. A blind person cannot rely on his own eyes, yet has other senses available to perform the same function. A white cane can be used to probe the surroundings and provide the person with information somewhat comparable to that delivered by the eyes. In some even more astonishing cases, some

people are able to effectively orient themselves through sounds, and even precisely reconstruct an environment by exploiting the reflection of acoustic waves emitted by themselves, just like a bat echolocation or the SONAR of a submarine [37].
These abilities are what SLAM is about. The great challenge that an ever-growing number of scientists and researchers around the world have been trying to overcome for almost four decades is to be able to transport these capabilities to inanimate objects, robots and vehicles. In other words, how can we equip and program a machine in such a way that we can get it to ultimately perform these tasks? Just as for a human being, even in the case of man-made devices there are a variety of ways to do it. However, the problem turned out to be of great difficulty. Its resolution requires the use of a wide range of algorithms and computations, as well as good understanding and processing of signals provided by sensors. Only in more recent times, with the development of increasingly high-performance, low-cost compact hardware combined with the development of new and efficient approaches, this technology is becoming of extreme practical interest.

While there is still great room for improvement, constant development has made SLAM now mature enough to find many real-world applications. Mobile robots, self-driving cars, Unmanned Aerial Vehicles (UAVs) and rovers for planetary exploration ([9]) are some examples of application on vehicles where SLAM can play a crucial role. There are actually even more peculiar uses. Many of the most recent Augmented Reality (AR) and Virtual Reality (VR) implementations, such as the *HP Reverb* ([22]), rely on SLAM to allow the user to conduct his movements in the virtual environment. Applications in the medical field are also very promising. Minimally invasive surgery can be effectively performed with the aid of a real-time SLAM, so as to provide the surgeon with accurate understanding of the hidden area to be operated on ([32]).

1.2 A bit of history

Cesar Cadena et al. [1] identifies three ages in the development of the SLAM problem:

1. *Classical Age* (1986-2004);
2. *Algorithmic-analysis Age* (2004-2015);
3. *Robust-perception Age* (2015-present).

According to *Durrant-Whyte et al.* [5], the true genesis of the probabilistic SLAM problem occurred at the 1986 IEEE Robotics and Automation Conference held in San Francisco. This event marks the beginning of the *Classical Age*, in which was recognized the need to address the issue of consistent and probabilistic mapping, both from a conceptual and computational point of view.
Within a few years, a major paper from *R. Smith et al.* [16] addressed the scenario of a vehicle moving in an unknown environment while acquiring relative observations of landmarks (a landmark is a feature of the environment that is taken as a reference to estimate the vehicle pose). From this work emerged that the estimates in the location of these landmarks necessarily all had to be correlated with each other because of the common error in estimating vehicle location. Consequently they concluded that to simultaneously solve the problem of localization and mapping it was necessary to repeatedly update a joint state, composed of the robot pose and every landmark position, following each landmark observation. This resulted in the need for a prohibitively large state vector, which increases the computational cost with the square of the number of landmarks. What led to difficulties in research was that the convergence problem was misjudged: at the time it was a widespread opinion that the uncertainty of the estimated state (state of the map plus location of the vehicle) obtained from a increasing number of observations would not converge. As a result, given the the excessive computational complexity of the mapping problem, the researchers' efforts shifted to minimize the number of correlations between landmarks, even coming down to address the problems of localization and mapping as separate.
The breakthrough came later, when it was realized that localization and mapping, when formulated as a single estimation problem, have actually a convergent solution. In particular, the importance of correlation between landmarks was reevaluated: contrary to what was believed, the solution improves as the number of correlations increases. Following these findings, great collective interest arose in solving the two problems simultaneously, so much so that in 1995 the acronym SLAM was coined. The later years of this first period led to the development of the main probabilistic formulations for SLAM, based first on Extended Kalman Filters and subsequently on Rao-Blackwellised Particle Filters and Maximum Likelihood estimation.
Between the late 1990s and the beginning of the new millennium, research in SLAM became increasingly popular and widespread, growing from conventions involving a few dozen researchers to full-fledged summer schools such as those held in Stockholm, Toulouse and Oxford. It is in this climate of driven research that the *Algorithmic-analysis Age* takes place. Some of the most important topics of discussion and

development that have been prominent as of these years (such as observability, convergence and consistency), are covered in the paper by *Dissanayake G et al.* [4].
The third and current era of SLAM development is the *Robust-perception Age*, and aims at solving four key requirements needed to push SLAM technology to more complex and larger-scale applications. (1) Robust performance: low failure rate for extended period of time and in a wide set of environments; (2) High-level understanding: understanding of the environment beyond basic geometric reconstruction (high-level geometry, semantics, physics and affordances); (3) Resource awareness: have systems tailored to use -and even adjust to- the available sensing and computational resources; (4) Task driven perception: have systems capable of selecting which information are relevant for the task it has to achieve.
In order to accomplish these goals, modern SLAM is now confirmed to be a multidisciplinary subject that combines many areas of research, such as sensor fusion, optimization, computer vision, machine learning and much more.

1.3 The SLAM Problem

A good introduction to the topic can be provided by answering two important questions that Cesar Cadena et al. [1] offered an answer to:

1. Do autonomous robots need SLAM?

2. Is SLAM solved as an academic research endeavor?

To answer the first question it is necessary to better understand what are the distinguishing features of a SLAM system and why odometry methods are not sufficient to meet certain operational requirements. Odometry can be defined as the use of data obtained from a motion sensor to estimate the change in position over time, and dates back to ancient times with the earliest forms of wheel odometers [35]. Although the technology to address odometry has been refined considerably over nearly two millennia, it has inherent restrictions that can not be fully resolved. All sensors, no matter how expensive, are subject to errors in performing measurement due to technological limitation in the manufacturing process, but also due to changes in the characteristics of their surroundings (temperature, magnetic fields, *etc.*). This causes drifts in the pose estimate that accumulates over time, hence a satisfying estimate becomes unachievable for long paths. It is possible to rely on apposite external landmarks or auxiliary systems (such as the GPS) that update the pose periodically to reduce drift, however, it is clear how in these cases the system is no longer autonomous in determining its state. State-of-the-art odometry exploits

both visual and inertial information to obtain very small errors, in the order of $< 0.5\%$ the trajectory length, but it is still unable to perform some required crucial tasks that can instead be tackled with SLAM.
A robot that relies solely on odometry sees the world as an "endless corridor" in which subjects and places never recur even if they have in fact been encountered before ((Figure 1.1)). This inability to recognize two successive passages through the same place is resolved in SLAM with the introduction of a new module called *loop closure*, which is specially dedicated to periodically checking whether the trajectory intersects or closes itself. In addition, the problem of minimizing the accumulated error is addressed with a dedicated back-end optimization module: odometry data are provided to the back-end to be optimized with methods that are relevant to the *state estimation* research area.

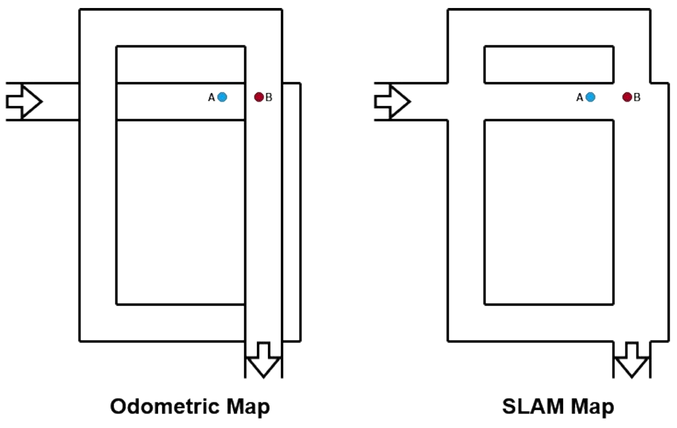


Figure 1.1: Through simple odometry the robot does not realize that points A and B are actually close to each other. Loop Closure allows to reconstruct the real topology of the environment.

The main three improvements that SLAM brings to an odometry algorithm are therefore:

- Error minimization techniques can be exploited to reduce the overall deviation in trajectory reconstruction, thereby achieving much greater accuracy than those provided by odometry alone;
- The capability to create a map that correctly describes the navigated space and the relative position between its features, and that furthermore can be exploited to predict and validate future measurements. This aspect makes the map of much greater practical utility;

- The combination of metric information and place recognition makes SLAM much simpler and more robust, avoiding wrong data association and perceptual aliasing.

Another aspect that makes SLAM of interest is indeed its contribution to the improvement of the odometry algorithms themselves. In fact, odometry constitutes an important block (front-end) in the framework of a generic SLAM method, and for this reason occupies a substantial share in the research environment. Not surprisingly, the previously mentioned methods that combine visual and inertial information ([14] [10]) arose in the very contest of SLAM development and can be treated as reduced SLAM systems in which the loop closure module is disabled.
So, is SLAM necessary? The reader should have guessed that the answer depends on the context, but in general it is clear how it brings very important capabilities that otherwise would not be obtainable. All those contexts that require a system that can locate itself accurately and create a descriptive map of the environment must rely on SLAM.
A final remark should be made about the definition of map. Depending on the applications for which the method is being developed in fact, different types of maps can be generated.

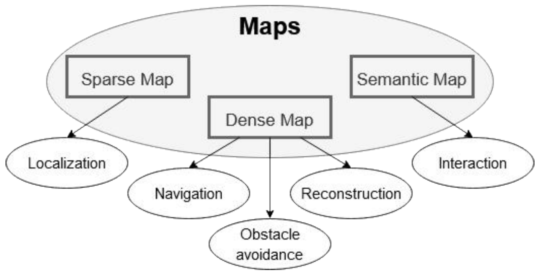


Figure 1.2: Different types of maps.

A *topological map* that only consider the connectivity between nodes might be enough. Or, we may only be interested in satisfying a basic function of a map, such that it allows spatial location of the vehicle. In this case, a *sparse metric map* that stores only some landmarks position and does not express all the objects is sufficient to accomplish the purpose. In other cases, however, we may be interested in carrying out navigation, obstacle avoidance or even scene reconstruction, for which is necessary to create a *dense metric map.* Dense maps are way more complex and expansive to produce since they aim to model all the objects that appear in the scene. Finally, in order to fulfill more advanced purposes such as interaction between people/robots and objects in the environment, it is necessary to generate a *semantic map* where subjects are entities that are defined and recognized.

Let's now address the second question: is SLAM solved as an academic research endeavor?
Again, it is not appropriate to give a sharp answer. In order to assess the maturity of the SLAM technology, one must consider the scope of application, particularly with reference to the following three aspects:

- *Vehicle/Robot.*
  - Sensors: what sensors are being used and what are their characteristics (sampling rate, resolution, accuracy, *etc.*)?
  - Type of motion: what are the vehicle dynamics and its maximum linear/angular velocity?
  - Available computational resources: is the algorithm able to run in the hardware that is at disposal?
- *Environment.* What characteristics does the environment in which SLAM is performed have? Presence of dynamic objects, presence of natural or artificial landmarks, scale, illumination conditions, *etc.*;
- *Performance.* What is the required accuracy in the estimate of the state of the robot and in the representation of the environment? What is the success rate, map typology, maximum operation time, estimation latency *etc.* that we want to achieve?

Relatively simple problems such as accurately (errors $< 10cm$) map a two-dimensional indoor environment by means of a robot equipped with a laser scanner and a wheel encoder, or perform visual SLAM in a controlled environment with slowly moving robots can be considered largely solved. On the other hand, many scenarios that result from the combination of Vehicle/Environment/Performance still have to cope with the four key requirements that define the *Robust-perception Age.* As of today, it is for example still difficult to have methods capable of operating at high speeds or in highly dynamic environments, or to face some of the strict performance specifications mentioned above.
Although the past twenty years have defined this technology and elevated it from the research environment alone, for many real-world applications SLAM is not solved yet. To achieve truly robust methods that can meet the requirements sought, more work is needed.

1.4 Probabilistic SLAM Formulation

This section aims to introduce the meaning of the two fundamental equations that summarize the SLAM process and tackles what considerations must be made to arrive at their resolution.

1.4.1 Fundamental Equations

Consider a generic robot equipped with sensors to perform SLAM. These sensors make a series of measurements in time steps denoted with $1, \dots, k$, therefore the problem must be addressed as discrete and is concerned only with the locations and the map at these moments. Regarding the locations, denote the pose (position plus orientation) of the robot with $\mathbf{x}$ so that the trajectory can be written as $\mathbf{x}_1, \dots, \mathbf{x}_k$. As already seen, the concept of map is very broad in SLAM and depends on the context of application. Sticking to its most general definition, which sees it as a description of the environment within which the robot moves, we can express the map with the set of landmarks $\mathbf{y}_1, \dots, \mathbf{y}_{j-1}, \mathbf{y}_j, \mathbf{y}_{j+1}, \dots, \mathbf{y}_N$ that are observed by the sensors during their operation.
The process seeks on the one side to describe how $\mathbf{x}$ varies from time step $k-1$ to k, while on the other wants to correctly associate a specific landmark $\mathbf{y}_j$ with the pose $\mathbf{x}_k$ in which the sensors detected it.

The robot's pose $\mathbf{x}_k$ at time step k depends on three terms: the previous pose value x_{k-1}, the input commands $\mathbf{u}_k$ provided by its control systems or by the pilot and finally the noise $\mathbf{w}_k$, which forces the introduction of a stochastic model. A mathematical expression, to which we refer as *motion equation*, that establishes the motion of the system can then be of the form

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k), \tag{1.1}$$

where $f(\cdot)$ is a general function that allows the problem to be expanded to any motion input.

Similarly, a second equation called *observation equation* is defined to describe the process in which the robot sees a landmark $\mathbf{y}_j$ at $\mathbf{x}_k$ and generates an observation data $\mathbf{z}_{k,j}$

$$\mathbf{z}_{k,j} = h(\mathbf{y}_j, \mathbf{x}_k, \mathbf{v}_{k,j}), \tag{1.2}$$

where again, $\mathbf{v}_{k,j}$ is the noise in this observation and $h(\cdot)$ is a general abstract function since there are many different sensors available to perform SLAM and their characteristics define the shape of $\mathbf{z}$ and $h(\cdot)$.

The set of these two equations:

$$\begin{cases} \mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k), & k = 1, \dots K \\ \mathbf{z}_{k,j} = h(\mathbf{y}_j, \mathbf{x}_k, \mathbf{v}_{k,j}), & (k, j) \in \mathcal{O} \end{cases} \tag{1.3}$$

is thus representative of the entire SLAM problem [18]. Notice that generally only a very small part of all landmarks can be seen in one location and an *observation equation* is formulated only when $\mathbf{x}_k$ sees $\mathbf{y}_j$. Moreover, we will see that in a Visual SLAM method there can be several thousand landmarks (or *features*), therefore the number of observation equation will be much larger than that of *motion equations.* Indeed, there may not even be *motion equations* if the robot is not equipped with motion measuring instruments. In such a situation, the problem is still solvable by making certain assumptions, such as imposing that the sensor is not moving or that it is moving at a constant speed, or again simply ignoring the presence of that set of equations. In the latter case, the entire optimization consists of only *observation equations* and it becomes similar to the Structure from Motion (SfM) problem, where a set of data (for example images) is employed to reconstruct the motion and structure. The difference between SLAM and SfM lies in the fact that in the first case the images are acquired in chronological order, while in the second one they may also be unrelated to each other and in no particular order.

Once it is recognized how to express the pose and how to parameterize the *observation equation* (these aspects will be dealt with in the next chapter when discussing Visual SLAM), the state estimation problem can be addressed.
As said, every measurement is affected by noise, so $\mathbf{x}$ and $\mathbf{y}$ are here regarded as random variables that obey a certain probability distribution. Solving the SLAM problem then involves answering the questions: how do we estimate $\mathbf{x}$ and $\mathbf{y}$ distribution by means of the control input $\mathbf{u}$ and the sensor reading $\mathbf{z}$ data? How do we update the estimation if new data is acquired?
The answers to these questions vary according to the nature of the two above equations, the precision required and the computational capabilities of the system. Generally, there are two ways to deal with the state estimation problem: the *incremental* method and the *batch* estimation method.

In the *incremental* method, also referred to as *filtering*, the current state is only determined by the previous one and the estimate is progressively constructed by ignoring observations and poses prior to the last one acquired. In the *batch* estimation method instead, the collected data is stored and consulted for every iteration to construct the best trajectory and map. Adding to this, during back-end optimization such nonlinear optimization approach can be used to use future information to update past values for $\mathbf{x}$ and $\mathbf{y}$.
In the most simplistic case the solution for a linear system subject to Gaussian noise can be nimbly obtained with the Kalman Filter (KF). However, in real-world applications we are more often dealing with nonlinear non-Gaussian (NLNG) systems, for which it is necessary to resort to other more complex means, such as the Extended Kalman Filter (EKF) if we assume the noise to be Gaussian, or more advanced particle filters (such as the *Rao-Blackwellised* filter) and *batch* estimation techniques like *graph optimization.*
It is appropriate to point out that EKF-based methods have been the mainstream choice in SLAM applications up until the first decade of the 2000s, however their theoretical and practical limitations have made them of less and less interest for state-of-the-art applications [18]:

1. The noise is assumed to always have a Gaussian distribution thanks to a linearization of the *motion* and *observation* equations around the working point;

2. The *k-th* state is only related to $k - 1$ (*Markov property*). This makes it difficult to recognize a return to the same place and to perform loop closure;

3. EKF SLAM is generally not suitable for large scale scenarios since the space required to accommodate the numerous state variable's mean and variance increases squarely;

4. It has no outlier detection mechanism, causing the system to diverge in their presence.

Because of these shortcoming, nonlinear *batch* optimization is deemed able to offer better results in terms of accuracy and robustness. In addition, the availability of more performing and compact hardware have made this more recent technique the most employed.

1.4.2 Batch State Estimation

For the reasons just mentioned, the SLAM implementation used in the simulations of this thesis makes use of *batch* estimation during optimization, both within bundle adjustment and pose graph for the back-end.
Considering all the instances $1, \ldots, N$ and assuming M map points it is possible to regroup all poses and map coordinates as

$$\mathbf{x} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}, \quad \mathbf{y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_M\}. \tag{1.4}$$

Similarly, all input and observation values are collected in $\mathbf{u}$ and $\mathbf{z}$.
The conditional probability distribution can then be written as

$$P(\mathbf{x}, \mathbf{y} | \mathbf{z}, \mathbf{u}), \tag{1.5}$$

and expresses the problem from a probabilistic point of view: finding the most probable estimate for the $\mathbf{x}$, $\mathbf{y}$ state of the robot when it is subject to $\mathbf{u}$ and observes $\mathbf{z}$. Using Bayes' theorem the variables are rearranged as follows

$$P(\mathbf{x}, \mathbf{y} | \mathbf{z}, \mathbf{u}) = \frac{P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y}) P(\mathbf{x}, \mathbf{y})}{P(\mathbf{z}, \mathbf{u})} \propto P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y}) P(\mathbf{x}, \mathbf{y}), \tag{1.6}$$

where $P(\mathbf{x}, \mathbf{y} | \mathbf{z}, \mathbf{u})$ is the *posterior probability*, $P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y})$ is the *likelihood* and $P(\mathbf{x}, \mathbf{y})$ the *prior*. Generally *posterior probability* can not be easily found directly in a nonlinear system, but from this expression we can formulate the search for an optimal point that maximizes the *posterior probability*, also known as Maximum a Posteriori (MAP) estimation:

$$(\mathbf{x}, \mathbf{y})^*_{MAP} = arg\, max\, P(\mathbf{x}, \mathbf{y} | \mathbf{z}, \mathbf{u}) = arg\, max\, P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y}) P(\mathbf{x}, \mathbf{y}). \tag{1.7}$$

Moreover, in case the *priori* information is not known -*i.e.* it is not known $\mathbf{x}$ and $\mathbf{y}$- the problem to be solved is that of Maximum Likelihood Estimation (MLE):

$$(\mathbf{x}, \mathbf{y})^*_{MLE} = arg\, max\, P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y}). \tag{1.8}$$

The meaning of this expression is then the search for the state $(\mathbf{z}, \mathbf{u})$ that is most likely to produce the data $(\mathbf{x}, \mathbf{y})$. To address this matter and express it in a form that allows its resolution, it is formulated in terms of least-square problem.

1.4.3 Least Square Formulation

To introduce this topic, assume that the two noise terms $\mathbf{w}_k$, $\mathbf{v}_{k,j}$ affecting the data satisfy a Gaussian distribution with zero mean:

$$\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k), \quad \mathbf{v}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{k,j}) \tag{1.9}$$

where $\mathbf{R}_k$ and $\mathbf{Q}_{k,j}$ are the covariance matrices.
Referring again to the fundamental equations 1.3, the conditional probability of the motion and observation data for a single time step are:

$$\begin{cases} P(\mathbf{x}_k|\mathbf{u}_k, \mathbf{x}_{k-1}) = \mathcal{N}(f(\mathbf{x}_{k-1}, \mathbf{u}_k), \mathbf{R}_{k,j}) \\ P(\mathbf{z}_{j,k}|\mathbf{x}_k, \mathbf{y}_j) = \mathcal{N}(h(\mathbf{y}_j, \mathbf{x}_k), \mathbf{Q}_{k,j}) \end{cases} . \tag{1.10}$$

The MLE problem associated with each of them can be solved by converting it into a minimum of negative logarithm problem. This is done for convenience since the logarithm function is monotonically increasing: maximizing the original function is equivalent to minimizing its negative logarithm. It can be shown, by considering the negative logarithm of the probability density function expansion form and inserting the observation model, that the MLE form can be rewritten like this:

$$\begin{aligned} (\mathbf{x}, \mathbf{y})^* &= arg\,max\, \mathcal{N}(h(\mathbf{y}_j, \mathbf{x}_k), \mathbf{Q}_{k,j})) \\ &= arg\,min\Big((\mathbf{z}_{k,j} - h(\mathbf{x}_k, \mathbf{y}_j))^T\, \mathbf{Q}_{k,j}^{-1}\, (\mathbf{z}_{k,j} - h(\mathbf{x}_k, \mathbf{y}_j)) \Big). \end{aligned} \tag{1.11}$$

As long as the quadratic term (the square root of which is named *Mahalanobis distance*) inside the parenthesis is minimized, the state's maximum likelihood is obtained. Solving this equation allows minimizing the noise term inherently present in the observations. For brevity's sake, the expression for the motion model is not given, but it can be derived in a completely analogous way.
Equation 1.11 is limited to a single instance, but there is obviously interest in minimizing noise for the whole set of observations and poses. Assuming that the inputs $\mathbf{u}$ and observations $\mathbf{z}$ are all independent from each other, it is possible to factorize the joint distribution as:

$$P(\mathbf{z},\mathbf{u}|\mathbf{x},\mathbf{y}) = \prod_{k} P(\mathbf{u}_k|\mathbf{x}_{k-1},\mathbf{x}_k) \prod_{k,j} P(\mathbf{z}_{k,j}|\mathbf{x}_k,\mathbf{y}_j), \tag{1.12}$$

in which the contribution of both fundamental equations is included.
Rewriting now the noise as the error between the model and the real data,

$$\begin{aligned} \mathbf{e}_{u,k} &= \mathbf{x}_k - f(\mathbf{x}_{k-1},\mathbf{u}_k), \\ \mathbf{e}_{z,j,k} &= \mathbf{z}_{k,j} - h(\mathbf{x}_k,\mathbf{y}_j), \end{aligned} \tag{1.13}$$

using the *Mahalanobis distance* and turning the product into a summation thanks to the property of the logarithm, Equation 1.12 becomes the objective function of the least-squares:

$$min\, J(\mathbf{x},\mathbf{y}) = \sum_{k} \mathbf{e}_{u,k}^T\, \mathbf{R}_k^{-1}\, \mathbf{e}_{u,k} + \sum_{k}\sum_{j} \mathbf{e}_{z,k,j}^T\, \mathbf{Q}_{k,j}^{-1}\, \mathbf{e}_{z,k,j}. \tag{1.14}$$

Notice that each error term is weighted by the inverse of the Gaussian covariance matrix $\mathbf{R}_k^{-1}$ and $\mathbf{Q}_{k,j}^{-1}$ (also called *information matrix*). This implies that if an observation is accurate then the covariance matrix will be "small" and the information matrix will be "large": the error associated with this observation will weight more than the others in solving the least-squares problem.
In addition, despite the potentially very large number of state variables (which can easily exceed several thousand), each error quadratic form is only related to a few of them. This *sparse* structure is crucial for making the system of incremental linear equations solvable in practice since the inversion of a high-dimensional *dense* coefficients matrix is prohibitively expensive for most systems.
One final question now remains to be answered regarding the least-squares problem: how to solve it?

The *Levenberg-Marquardt method*

The resolution of nonlinear least-square problems is a well covered topic in many numerical analysis textbooks, and a detailed description of it is beyond the scope of this thesis. For an in-depth discussion it is recommended the consultation of such works, however, it was still deemed important to quickly refresh here the *Levenberg-Marquardt (L-M)* method since it is the most widely used method in Visual SLAM and has been applied recurrently in the implementation described in Chapter 2.
The L-M method, also called *Damped Newton (DN)* method, is an advanced algo-

rithm used to solve nonlinear least-squares problems by interpolating between the *Gauss-Newton (G-N)* approach and the *steepest descent* method. This makes it more robust and accurate than the simple *G-N*, albeit generally slower.
Consider a generic least-square problem expressed by

$$\min_{\Delta x} F(\mathbf{x}) = \frac{1}{2}\|f(\mathbf{x})\|_2^2 \quad , \quad \mathbf{x} \in \Re^n \tag{1.15}$$

with $f(\mathbf{x})\colon \Re^n \mapsto \Re$, a generic scalar nonlinear function and its first-order Taylor expansion:

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \mathbf{J}(\mathbf{x}^T \Delta\mathbf{x}), \tag{1.16}$$

where $\mathbf{J}(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ is the Jacobian. The objective is to find the increment $\Delta\mathbf{x}$ such that $\|f(\mathbf{x} + \Delta\mathbf{x})\|^2$ reaches the minimum, or, in other words, to solve the linear least-square problem:

$$\Delta\mathbf{x}^* = \arg\min_{\Delta\mathbf{x}} \frac{1}{2}\|f(\mathbf{x}) + \mathbf{J}(\mathbf{x}^T)\Delta\mathbf{x}\|^2. \tag{1.17}$$

By expanding the square term and setting its derivative with respect to $\Delta\mathbf{x}$ equal to zero, the important *normal equation* is obtained:

$$\begin{gathered} \mathbf{J}(\mathbf{x})\mathbf{J}^T(\mathbf{x})\Delta\mathbf{x} = -\mathbf{J}(\mathbf{x})f(\mathbf{x}), \\ \mathbf{H}\Delta\mathbf{x} = \mathbf{g} \end{gathered} \tag{1.18}$$

where $\mathbf{H} = \mathbf{J}(\mathbf{x})\mathbf{J}^T(\mathbf{x})$ approximates the *Hessian* matrix, which is the second-order derivative of $f(\mathbf{x})$ with respect to $\mathbf{x}$.
Equation 1.18 is the core of the *Gauss-Newton* method and its resolution returns the increment $\Delta\mathbf{x}$ to be used for the next iteration, until it is minor than a predetermined threshold value.
The convergence of *G-N* may encounter difficulties in some cases, for example if the positive semi-definite matrix $\mathbf{H}$ is singular or ill-conditioned, or if $\Delta\mathbf{x}$ is so large that the linear approximation becomes inaccurate. To address these shortcomings we then consider the *Levenberg-Marquardt* method.
We introduce a *trust-region* for $\Delta\mathbf{x}$ around the expansion point that defines where the second-order approximation is valid. To quantify the scope of this region it is necessary to understand the degree of approximation by means of the indicator ρ:

$$\rho = \frac{f(\mathbf{x} + \Delta\mathbf{x}) - f(\mathbf{x})}{\mathbf{J}(\mathbf{x})^T \Delta\mathbf{x}} \tag{1.19}$$

where the numerator is the decreasing value of the real object function, and the denominator is the decreasing value of the approximation. The closer ρ is to 1, the better is the approximation and is appropriate to enlarge the *trust-region* radius μ. The solving procedure for *L-M* can be summarized as follows.

1. Set the initial value $\mathbf{x}_0$ and the initial trust-region radius μ;
2. For the k-th iteration, calculate $\mathbf{J}(\mathbf{x}_k)$ and the residual $f(\mathbf{x}_k)$;
3. Solve the linear equation:

   $$(\mathbf{H} + \lambda \mathbf{D}^T \mathbf{D}) \Delta \mathbf{x}_k = \mathbf{g} \tag{1.20}$$

   to calculate the increment;
4. Compute ρ with equation 1.19;
5. If $\rho > a$, set higher value for μ. Otherwise, if $\rho < a$, set lower value for μ;
6. If $\rho > b$, where $b > a$, set $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$;
7. Repeat from step 2 if solution did not converge, *i.e.* if $\Delta \mathbf{x}_k$ is not below the chosen threshold.

Here, a, b and the factor for increasing/decreasing μ are empirical values chosen by the user.
Let's now focus on how Equation 1.20 was obtained. Consider a linear problem based on the *G-N* method added with the *trust-region*, Equation 1.17 then becomes:

$$\Delta \mathbf{x}_k^* = \arg \min_{\Delta x_k} \frac{1}{2} \| f(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)^T \Delta \mathbf{x}_k \|^2, \quad s.t. \quad \|\mathbf{D} \Delta \mathbf{x}_k\|^2 \leq \mu; \tag{1.21}$$

so that the increment is limited within a region with radius μ. The coefficient matrix $\mathbf{D}$ is in the simplest case the identity matrix $\mathbf{I}$, hence the region in a sphere. In more general cases, $\mathbf{D}$ is instead a non-negative diagonal matrix (typically the square root of the diagonal elements of $\mathbf{H} = \mathbf{J}(\mathbf{x})\mathbf{J}^T(\mathbf{x})$), so that the sphere becomes an ellipsoid and the constraint range is larger on the dimensions with small gradient.
In brief, to solve the optimization problem expressed by Equation 1.21 it is necessary to form a Lagrangian function that involves the radius constrain into the objective

function:

$$\mathcal{L}(\Delta\mathbf{x}_k, \lambda) = \frac{1}{2}\|f(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)^T \Delta\mathbf{x}_k\|^2 + \frac{\lambda}{2}(\|\mathbf{D}\Delta\mathbf{x}_k\|^2 - \mu), \tag{1.22}$$

where λ is the Lagrange multiplier and serves as a damping factor that can be adjusted for each iteration.
Similarly to what was done with *G-N*, by posing the derivative made with respect to $\Delta\mathbf{x}_k$ of this expression equal to 0 we finally obtain Equation 1.20, which is analogous to Equation 1.18 but with the additional term $\lambda\mathbf{D}^T\mathbf{D}$. Small values of λ cause $\mathbf{H}$ to dominate the equation and make the quadratic approximation model predominant (*i.e.* the method becomes similar to *G-N*). Higher values of λ, on the other hand, make the search for the solution expand beyond the quadratic model and the method becomes therefore closer to the *steepest descent* approach.

1.5 Factor Graph and Landmarks Correlation

The problem so far described can be better understood in terms of inference over a factor graph [18]. This representation makes it possible to visualize at the same time all the main actors that characterize a SLAM system and to see what relationships are established among them. The variables $\mathbf{x}_k$, $\mathbf{y}_j$ and $\mathbf{K}$ are the *nodes* in the graph. In particular, $\mathbf{K}$ is the variable associated with the sensor intrinsic calibration parameters and in the case of a camera describes the camera to image / image to pixel transformations. The *likelihood* $P(\mathbf{z}, \mathbf{u}|\mathbf{x}, \mathbf{y})$ and the *prior* $P(\mathbf{x}, \mathbf{y})$ are here called *factors*, and they encode probabilistic constraints over a subset of nodes.
A factor graph is then a graphical model that explicates the dependence between the *factors* and the corresponding *nodes* and expresses the nonlinear least-squares optimization problem.

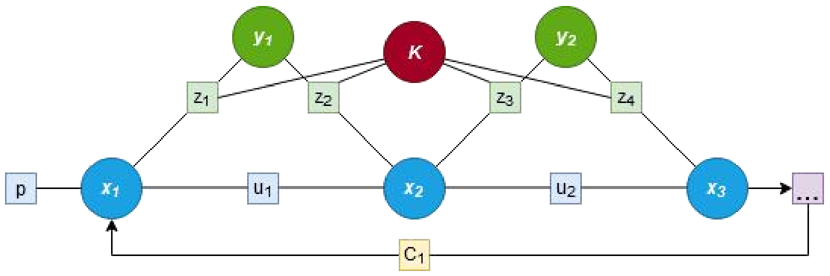


Figure 1.3: Example of a factor graph.

Consider Figure 1.3. The circles represent variables: pose (blue), landmark positions (green), intrinsic calibration parameters (red). The squares indicate instead the *factors*: denoted with u are the *factors* that correspond to odometry constraints while z correspond instead to camera observations. p denotes prior factors and c_1 denotes one instance of loop closure. Although this is a simple example, it does not differ conceptually from the graph that is established for a real case. The sparse nature of the problem can also be appreciated, whereby variables generally interconnect only in smaller groups.

As understood, simultaneous estimation of the robot pose and landmark locations is being pursued, but their true values are never known or measured directly. Errors in knowledge of the pose when the landmarks observations are made cause much of the error between true and estimated landmarks location. This aspect entails a high correlation among the errors in landmarks location estimate: the relative location between any two landmarks $\mathbf{y}_i$ and $\mathbf{y}_j$ can be obtained with high accuracy, even when the absolute location of $\mathbf{y}_i$ and $\mathbf{y}_j$ is not well known. In probabilistic terms it means that the joint probability density $P(\mathbf{y}_i, \mathbf{y}_j)$ is highly peaked even if the marginal densities $P(\mathbf{y}_i)$ and $P(\mathbf{y}_j)$ are quite disperse. The crucial turning point mentioned in Section 1.2 that made the SLAM problem solvable as we know it today concerns this very aspect: it has been proven that correlations between landmark estimates increase monotonically with the number of observations made, namely the joint probability density on all landmarks $P(\mathbf{y})$ becomes more peaked as more observations are made. This characteristic is justified by the fact that the observations can be regarded as ”nearly independent” readings of the relative location between landmarks (”nearly” in the sense that, as we will see in an instant, the observation errors will actually be correlated through successive robot motions).

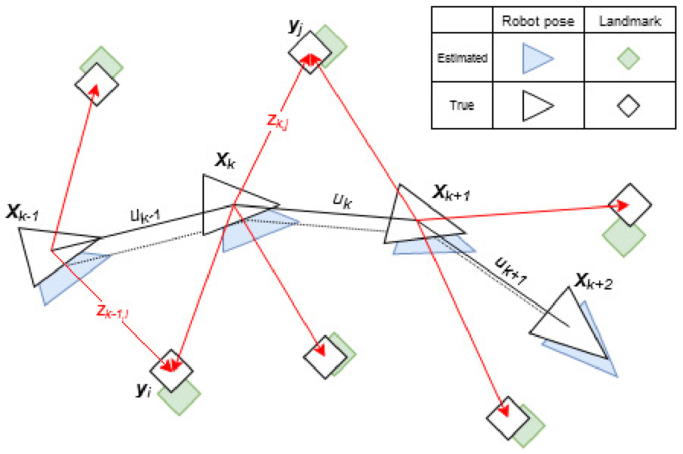


Figure 1.4: The essential SLAM problem.

Refer now to the representation of the essential SLAM problem shown in Figure 1.4. The relative location of the observed landmarks $\mathbf{y}_i$ and $\mathbf{y}_j$ is measured from a certain location with pose $\mathbf{x}_k$. As the vehicle moves to $\mathbf{x}_{k+1}$, it sees again $\mathbf{y}_j$ and updates the estimated location -both of the landmark and its own- with respect to its previous pose $\mathbf{x}_k$. Now, since the relative location of $\mathbf{y}_i$ and $\mathbf{y}_j$ is well known, the position of $\mathbf{y}_i$ is also updated, even though this landmark is not observed form $\mathbf{x}_{k+1}$. In addition, the correlation between these two landmarks increases even more because the same observation is used to update their position. While the robot is in $\mathbf{x}_{k+1}$, then, it also sees two new landmarks whose position is determined relatively to $\mathbf{y}_j$ and therefore are immediately linked to the rest of the map. As the vehicle advances, it is clear how this process repeats itself creating a network of relative correlations whose precision increases with every new step.
Figure 1.5 helps to visualize this fact by means of the spring network analogy.

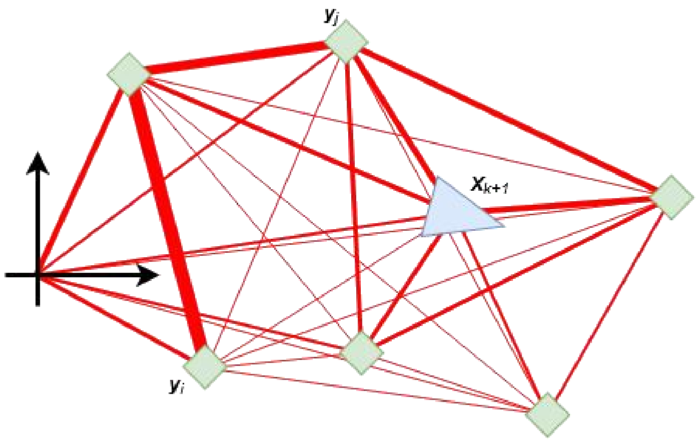


Figure 1.5: Spring network analogy. The thickness of the red lines represents the stiffness.

All landmarks encountered up to a certain time-step can be imagined forming a network with connections running between them, the pose of the robot and a global reference system (which can be taken for example from an assigned initial pose value). These connections act like springs whose stiffness depends on the level of correlation achieved: as the vehicle moves within the environment and performs new observations, the springs become increasingly stiffer due to the phenomenon of correlation propagation previously described. The greater the stiffness, the better is the estimate found and subsequent observations will make these acquired positions vary less and less toward convergence to the true value. A lower stiffness, on the other hand, indicates that the estimate has low confidence and the position can be readjusted “with little effort” by subsequent observations.

1.6 Visual SLAM

As mentioned, there is a variety of ways to perform SLAM, depending on the scope of application. LIDARs (Light Detection and Ranging), SONAR (Sound Navigation and Ranging) and cameras, have all been applied successfully in this field. Moreover, systems that combine two or more different sensors are very often considered to increase the robustness and accuracy. A widely used coupling is that between cameras and IMUs (Inertial Measurement Units) as these two technologies are in many ways complementary and when combined offer significantly superior performance and stability. Interesting discussions on the topic of Visual-Inertial Odometry can be found in [3] [15]. On top of these more popular sensors, there is also interest in making use of less conventional means, such as particular types of cameras (Light-Field Camera and Event-Based Camera) or even magnetic, olfaction, and thermal sensors [1].

This thesis focuses solely on Visual-SLAM (V-SLAM), which makes exclusive use of cameras as sensors to solve localization and mapping. V-SLAM has been a hot topic since the early 2000s and it is proving to be an effective and reasonably inexpensive approach for many scenarios. It is, however, also showing to be one of the most sophisticated embedded vision technologies to date.

1.6.1 Cameras

The cameras that are mounted on board of vehicles and robots to perform SLAM can be of the most diverse nature, each with its own advantages and disadvantages. The three most widely used configurations are:

- **Monocular Camera**: a single conventional camera is the most simple, compact and inexpensive solution. It generates as output a video captured at a certain number of frames per second that is provided as input to the SLAM algorithm. However, since a single image is just a 2D projection of a 3D space, the depth information is lost. This quantity is essential to perform SLAM, and its relative value can only be recovered here with the pixel disparity calculated from translational movements of the camera. Consequently, in monocular SLAM there is also the problem of scale ambiguity: the map can drift in one additional DoF other than the six pose values, namely the scale.

- **Stereo Camera**: two synchronized monocular cameras disposed with a certain distance from each other (baseline) constitute a stereo configuration.

Stereo cameras can estimate the depth solely from the difference between the image pair, thus overcoming the limitations of a single monocular camera. A larger baseline generates a larger parallax and allows to measure further distances. Depending on the application, a lot of space may be required to accommodate an effective configuration. Another disadvantage is that the set-up and calibration process is more complicated.

- **RGB-D Camera**: also known as depth cameras, RGB-D cameras became popular in more recent years, especially following the launch of *Microsoft Kinect* [33] in 2010. They can simultaneously collect color images and depth images, and directly gain depth maps mainly by actively emitting infrared structured light or calculating time-of-flight of the pulses. Because of this direct measurement, they require less computational resources than a stereo camera, but are subject to restrictions that make them suitable only for particular environments. As of today, depth cameras still suffer from small field of view, noisy data, narrow measurement range and high sensitivity to light sources. For these reasons they are essentially only used in indoor environments.

It must then be considered that, in any case, it is not just the technical specifications that dictate the choice, but weight/size and cost can become the main driver parameters. A small mass-produced UAV will not be able to accommodate a payload consisting of a large stereo configuration or expensive sensors and quality lenses. This can result in having a sub-optimal set-up, that relies on cameras that generate distorted images of modest quality, with low resolution and plagued by a good amount of noise. Fortunately, the capacity of modern V-SLAM algorithms is such that they can handle even this type of data, so it is indeed possible to use low-cost equipment with satisfactory results.

In light of the considerations just made, a stereo set-up was chosen to be used in this thesis, the characteristics of which were directly taken from the reference KITTI dataset that will be introduced in Section 3.2.

1.6.2 Pinhole Camera Model

The geometric model that most simply describes the operation of projecting 3D space points (in meters) into a 2D image plane (in pixels) is that of the pinhole camera. This representation will be employed to model the cameras, therefore is here briefly introduced [18].

Consider Figure 1.6, the camera coordinate system is given by $O-XYZ$, where O is the optical center and coincides with the “hole” in the camera plane of the pinhole model. The 3D world point P with camera coordinates $[X, Y, Z]^T$ is projected through O and reaches the physical imaging plane $O' - X'Y'Z'$ to define the image point P' defined by $[X', Y', Z']^T$. Considering that the focal length f sets the physical distance $O - O'$ between the imaging plane and the camera plane, the following relationship applies:

$$\frac{Z}{f} = \frac{X}{X'} = \frac{Y}{Y'}, \tag{1.23}$$

which describes the spatial relationship between P and its image. It is necessary to point out that, to be precise, in Eq. 1.23 the second and third term should have a “$-$” sign in front because the image is projected inverted. If the imaging plane is arbitrarily moved to the front however, this expression results correct.

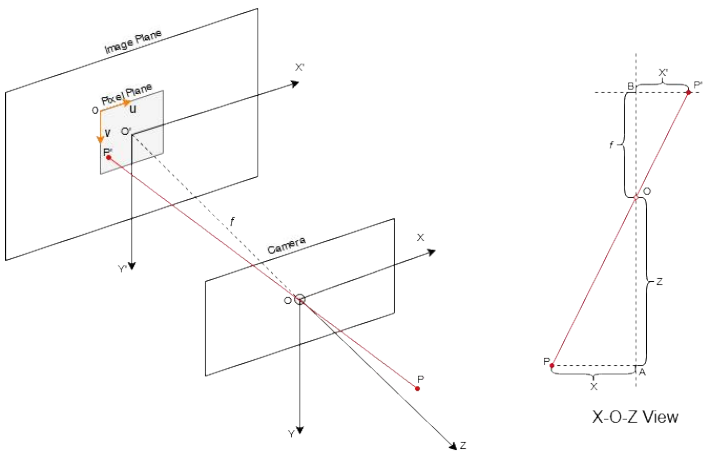


Figure 1.6: Pinhole Camera Model.

Since the camera sensor consists of pixels, within the imaging plane is also defined the pixel plane $o - uv$ in relation to which P' has coordinates $\mathbf{p}_{uv} = [u, v]^T$. By referring again to Equation 1.23, the relationship between the coordinates of P and the pixel coordinates are:

$$\begin{cases} u = f_x \frac{X}{Z} + c_x \\ v = f_y \frac{Y}{Z} + c_y \end{cases}, \tag{1.24}$$

where $[c_x, c_y]^T$ are the principal point offset and $f_x = \alpha\, f$, $f_y = \beta\, f$ are the scaled focal lengths, all expressed in pixels.

Putting Z to the left side and using homogeneous coordinates for $\mathbf{p}_{uv}$, the matrix form becomes:

$$Z\mathbf{p}_{uv} = Z\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \triangleq \mathbf{KP}, \tag{1.25}$$

with $\mathbf{K}$ the intrinsics matrix. Eq. 1.25 describes the projective 3D to 2D transformation from the camera coordinates to the pixel coordinates. To express the relationship between pixel coordinates $\mathbf{p}_{uv}$ and world coordinates $\mathbf{P}_w$ it is necessary to introduce the rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$ that define the camera pose:

$$Z\mathbf{p}_{uv} = Z\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \mathbf{K}(\mathbf{RP}_w + \mathbf{t}) = \mathbf{KTP}_w. \tag{1.26}$$

where the third equation implies a conversion from homogeneous to non-homogeneous coordinates, with $\mathbf{T} \in SE(3)$.

1.6.3 Image Distortion

In order to describe the entire projection process, it is necessary to add to the pinhole camera model the effect of distortion. Real cameras are equipped with a number of lenses for the purpose of controlling the image characteristics. In SLAM, it may be of interest to make use of lenses that widen the Field-of-View so that there are more subjects in the scene and more features can be detected. Their presence, however, can also negatively affect the way light propagates toward the Imaging Plane in essentially two ways: (1) the shape of the lens itself affects how the light is conveyed, (2) due to dimensional tolerances of the components and their mechanical assembly, the Imaging Plane and the lens are not perfectly aligned and parallel. As a result, two typical kinds of distortion arise:

- **Radial distortion**. Since typically the lenses are center-symmetrical, the distortion they cause will be radially symmetrical. In particular, this class is divided into two main categories: *Barrel distortion*, where the radius of pixels decreases as the optical axis-s distance increases; *Pincushion distortion*, which is the opposite of the previous one. In both cases, the line that intersects the center of the image and the optical axis remains the same.

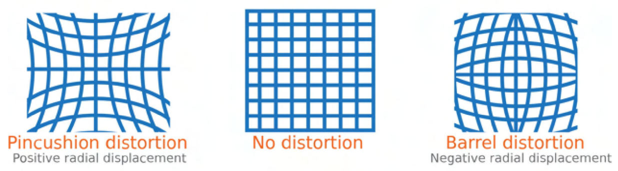


Figure 1.7: Radial Distortion (image from *Mathworks* documentation).

These effects can be modeled by a polynomial expression:

$$x_{Distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{1.27}$$

$$y_{Distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{1.28}$$

Where $x_{Distorted}$ and $y_{Distorted}$ are the normalized image coordinates of the point after distortion, x and y are the normalized coordinates before distortion, k_1, k_2, k_3 are the radial distortion coefficients of the lens and r is the distance between a point $\mathbf{p}$ on the normalized plane and the origin of the coordinate system in the plane, therefore $r^2 = x^2 + y^2$.

- **Tangential distortion**. Because of the technological limitation in the construction and assembly of the optical devices, the lens and the imaging surface cannot be perfectly parallel, thus introducing a distortion that tilts and stretches the image.

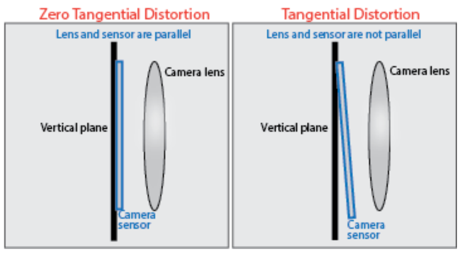


Figure 1.8: Tangential Distortion (image from *Mathworks* documentation).

Similarly to above, the expressions for tangential distortion are of the type:

$$x_{Distorted} = x + 2p_1 xy + p_2(r^2 + 2x^2) \tag{1.29}$$

$$y_{Distorted} = y + p_1(r^2 + 2y^2) + 2p_2 xy \tag{1.30}$$

where p_1, p_2 are the tangential distortion coefficients of the lens.

The number of coefficients can be chosen depending on the severity of the effect, and their value can be provided by the manufacturer or be estimated through the calibration process.

1.7 Purpose of the Thesis and Outline

The following thesis aspires to accomplish three goals:

1. Present a detailed analysis and description of the *Matlab* implementation of the Visual-SLAM ORB-SLAM2 system;
2. Implement a comprehensive and self-sufficient framework for performing SLAM testing in a virtual environment, with possibilities for great customization of the scene, sensors and trajectory;
3. Demonstrate the validity of a virtual environment as an alternative to a dataset for testing purposes.

Specifically, the structure of the work is as follows:

Chapter 2. Describes the ORB-SLAM2 system as it has been implemented in *Matlab*, showing for each thread a detailed block diagram of the operations that take place and focusing on certain aspects considered crucial for this and other similar SLAM systems.

Chapter 3. After justifying the interest in using a virtual environment, the framework built for performing the tests is described, paying particular attention to describing how the different software was used.

Chapter 4. Presents how the tests have been prepared and performed, both on images taken from a real scene and on images generated in a virtual environment. A discussion of the results obtained follows.

Chapter 5. Encapsulates final considerations on the results and how the work can be expanded or improved in future developments.

Chapter 2

ORB-SLAM2 *Matlab* Implementation

2.1 ORB-SLAM Series Systems

ORB-SLAM [11] is a feature-based monocular SLAM system presented in 2015 that is capable of operating in real time, both in small and large outdoor/indoor environments. It became very popular for its qualities as a complete and easy-to-use system (at least, simpler than other methods), so much so that it is still often the preferred choice in the world of open-source feature-based methods. In 2017 was introduced ORB-SLAM2 as a result of the work carried out by Raùl Mur-Artal and Juan D. Tardòs [12], improving on the capabilities of the predecessor and, most importantly, allowing the method to be extended to include binocular and RGB-D applications. Some of the characteristics that make these systems of great interest are:

- Highly versatile and compatible with a wide array of sensors;
- The exclusive use of ORB features, which offer an excellent trade-off between computational speed and accuracy;
- An innovative three tread structure consisting of Tracking, Local Mapping and Loop Closure;
- An highly effective loop detection algorithm to mitigate accumulated drift;
- High use of optimizations around feature points that makes the system particularly robust.

The map that is generated consists of sparse feature points, therefore can only be employed to satisfy the localization needs, but not navigation, obstacle avoidance, interaction *etc.* Given its characteristics, ORB-SLAM (ORB-SLAM2) is still a very valid method today and is among those that constitute the state of the art in Visual-SLAM.

It must be noted that in 2021 has also been made public ORB-SLAM3 [2], which introduced further improvements and expansions for the system. The two most impactful new features are: (1) the capability of performing visual-inertial SLAM that relies on Maximum-a-Posteriori estimation. This significantly increases the versatility and the robustness of the system, with an increase in accuracy of up to 10 times; (2) the introduction of a multiple map system that relies on a new place recognition method with improved recall. In case tracking is lost, a new map is initialized and will then be seamlessly merged with previous maps when revisiting mapped areas.
Unfortunately, there are no *Matlab* implementations yet for ORB-SLAM3, but this does not prevent the achievement of the goals set for this thesis.

2.2 *Matlab* System Implementation Overview

This Chapter provides a comprehensive insight on the entire ORB-SLAM2 system as implemented by *MathWorks* in *Matlab*. This is a closely related version to the one originally proposed by Raùl Mur-Artal and Juan D. Tardòs, to which some variations were applied. It is a very recent release (it makes use of functions introduced only in *Matlab R2022b*), and it has been confirmed that an updated version will be made available in the future. The full original code can be downloaded from the *MathWorks* documentation [25], while the changes that have been made to the parameters of interest so that it could satisfy the purposes of this thesis can be found in Chapter 4.
A complete and detailed description of the entire method in all its facets would certainly be prohibitively long given its complexity, therefore the following focuses on what have been considered the crucial and characteristic aspects, while covering all sub-routines of the system. In particular, the intent of this work is to provide a novel introspective on a V-SLAM system, in which its structure and logical process that led to its formation are well defined.
The entire pipeline can be condensed into four macro blocks (Fig.2.1):

1. **Map Initialization**: The first pair of stereo images is used to initialize the map of 3D points using the disparity map. The left image is stored as first key frame;

2. **Tracking**: For each stereo pair, the pose of the camera is estimated by matching features between subsequent frames. The estimate is improved by tracking the local map and each frame is evaluated to establish if it is a key frame to forward to Local Mapping;

3. **Local Mapping**: New 3D map points are computed from the disparity of the stereo pair and by triangulating feature points in the current Key Frame and its connected Key Frames. A local bundle adjustment minimizes reprojection errors by adjusting simultaneously the camera pose and the 3D points;

4. **Loop Closure**: Each Key Frame is evaluated to find a possible loop closure candidate by comparing it against all previous Key Frames using a bag-of-features approach. Once the loop is detected, pose graph optimization refines the camera pose for the entire trajectory.

Each of these threads will be expanded and described in the subsequent sections with the help of block diagrams built directly by consulting the code.

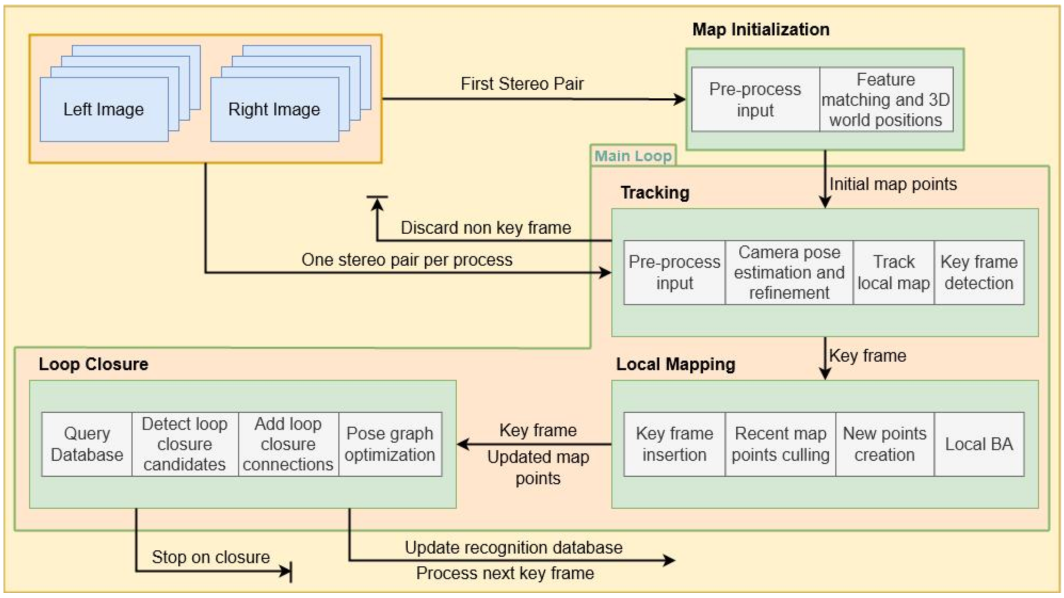


Figure 2.1: System Threads and Modules.

2.3 Map Initialization

The process that initialize the map is the first to take place and has important implications on the subsequent operations of the method. It uses only the first stereo pair to find the ORB feature points in each image and reconstruct the initial 3D world points form the disparity map. The feature points of the two images are then associated with each other and identified in the 3D world map. All the crucial steps that occur in this thread are summarized in the flow chart in Figure 2.2.

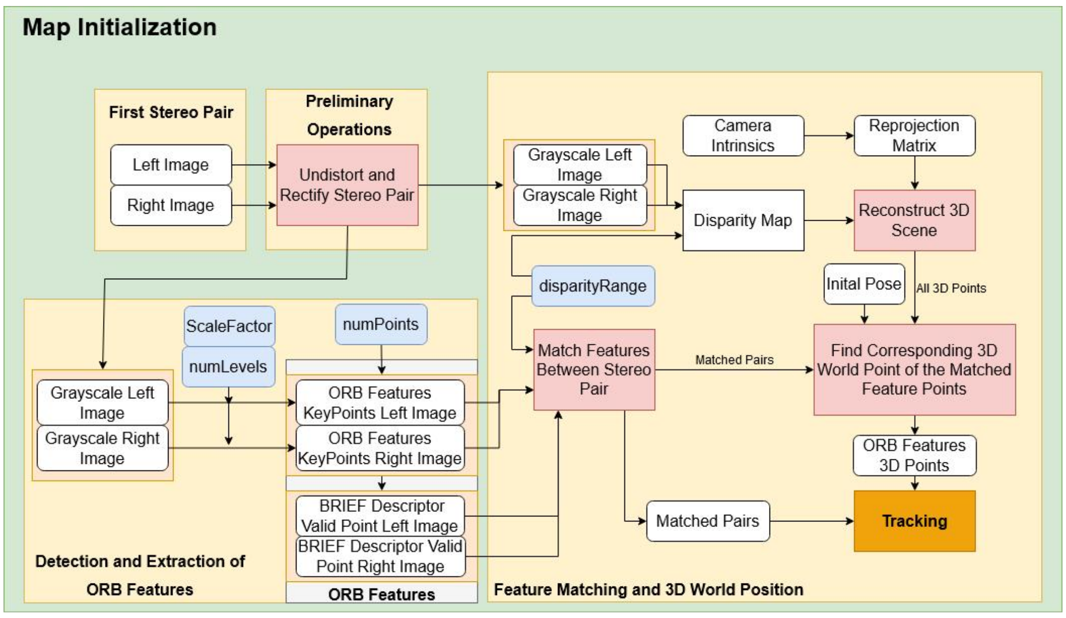


Figure 2.2: Map Initialization can be subdivided in three sections: Preliminary Operations, Detection and Extraction of ORB Features, Feature Matching and 3D World Position.

2.3.1 Preliminary Operations

Before the actual SLAM can begin, it is necessary to prepare the images properly so that they can then be used by the system's algorithms.

Image Distortion

Section 1.6.3 introduced distortion and how it can affect the shape of the image. V-SLAM algorithms require images provided free of distortion to interpret correctly the three-dimensional space they are depicting. The images from the dataset that will be used in this work are for this reason given to the users already distortion-free. Accordingly, to maintain the same framework, the virtual cameras will also be set up to generate distortion-free images. In the more general case however, it is possible to remove distortion through the `undistortImage` function from the *computer vision Toolbox* in *Matlab*. Once the intrinsic parameters of the camera and the values of the distortion coefficients are provided, it returns the undistorted images.

Image Rectification

A second operation that generally needs to be performed when dealing with stereo images is that of rectification, which is a transformation used to project images into a common image plane. To match a pixel of one image of a stereo pair with the other it is necessary to move along the epipolar line, which is generally slanted since there is a relative rotation between the two cameras or there is an offset in the direction of the z axis. Rectification is performed to simplify the resolution of the correspondence problem: it warps both images so that they appear coplanar, that is, as if they had been captured with two cameras having only a horizontal offset. As a result, the obtained images satisfy these two properties:

1. All epipolar lines are parallel and horizontal;
2. Corresponding points have identical vertical coordinates.

Ideally, the cameras in a stereo SLAM system are installed coplanar, however this condition is not guaranteed as it is impossible to maintain in a practical implementation, much less if the vehicle is in motion and subjected to vibrations. Consequently it is always a good practice to perform the rectification operation when dealing with real cameras. As in the case of distortion, this has already been performed in the dataset images (in fact, the two actions are typically performed one after the other, sometimes enclosed in a single action) and therefore they are provided to the users ready for use, together with the Reprojection Matrix $\mathbf{Q}$.

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & 1/b & 0 \end{bmatrix} \tag{2.1}$$

This is a 4×4 matrix containing the focal length f, the reciprocal of the baseline b of the fictitious rectified stereo camera, and the coordinates $[c_x, c_y]$ of the Principal Point of the rectified left camera. It will be used later to reconstruct, starting from the image-diparity coordinates, the coordinates of the three-dimensional point expressed in the left camera rectified reference system.
If the images are not already rectified, this operation can also be easily performed

within *Matlab* with the `rectifyStereoImages` function by applying two projective transformations. The output consists in the two adjusted images and also the newly obtained Reprojection Matrix $\mathbf{Q}$.

What about the images obtained by the virtual stereo configuration? The problems inherent the relative positioning of cameras that afflict the real world do not arise here if the cameras are placed with only an horizontal offset and their relative position remains constant throughout the simulation. Figure 2.3 shows the two image plane of the virtual stereo pair, onto which the point M of the scene is projected in P and P', respectively. O and O' are the optical centers and in yellow is the epipolar line. For the epipolar line to be horizontal, *i.e.* to establish the coplanarity of the image planes, these two relationships must be satisfied:

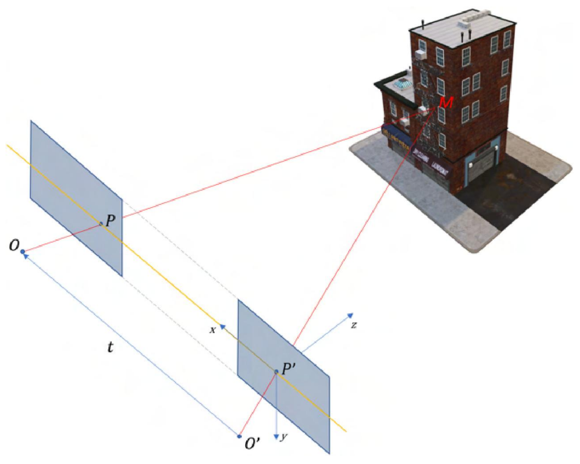


Figure 2.3: Epipolar line on two coplanar image planes.

$$\mathbf{R} = \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

$$\mathbf{t} = (T, 0, 0) \tag{2.3}$$

Where $\mathbf{R}$ is the 3×3 rotation matrix and $\mathbf{t}$ is the translation vector that defines the relative position between the cameras. The cross product between these two objects define the Essential Matrix $\mathbf{E}$, that relates corresponding points in stereo images and here is

$$\mathbf{E} = \mathbf{t}^{\wedge}\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -T \\ 0 & T & 0 \end{bmatrix} \tag{2.4}$$

Considering now the normalized coordinates for the matched points P and P', which are $\mathbf{x} = [u, v, 1]^T$ and $\mathbf{x}' = [u', v', 1]^T$, the *Longuet-Higgins* equation imposes

$$(u', v', 1) \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -T \\ 0 & T & 0 \end{bmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 0 \tag{2.5}$$

and by developing the product

$$Tv = Tv' \tag{2.6}$$

which means that the y coordinate does not change, therefore the image of the 3D point M in the two planes belongs to the same horizontal line.

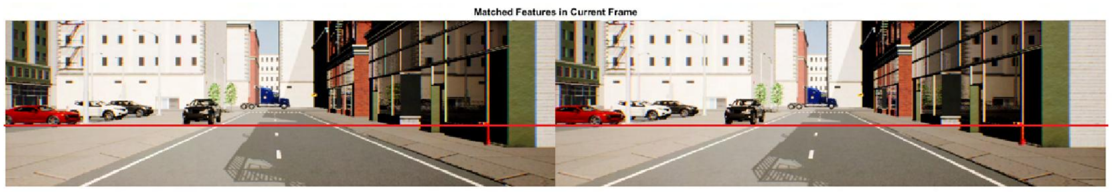


Figure 2.4: Undistorted and rectified stereo images obtained from the first frame of a simulation. The red epipolar line is horizontal and passes through the same points in both images.

The stereo pair is now ready to be used in the detection and extraction of the features.

2.3.2 Detection and extraction of ORB features

ORB-SLAM2 is a system of the feature method type. In this category the goal is to identify some representative points from the images in such a way that they are tracked even after small changes in camera pose. These unambiguous points -also called image features in VO- can therefore be found in a sequence of frames and will be used later to identify their 3D position in space and to address the problem of camera pose estimation. The choice of these points is not trivial, since they should be able to meet the following goals [18]:

- **Repeatability**: Different images will present the same feature;
- **Distinctiveness**: Different features have different expressions;
- **Efficiency**: The number of feature points in the image is much smaller than the total number of pixels;
- **Locality**: The feature is only related to a small image area.

Consequently, individual pixels and even corner and edges are typically not suitable for SLAM applications since a variation of the pose or of the illumination conditions could instantly change the appearance of the feature. Over the years there has been significant research to find points that could satisfy the previous list, and among the best achievements is the ORB feature.

ORB Feature

The typology of image feature used by ORB-SLAM2 is called ORB (Oriented FAST and Rotated BRIEF). Today it is a favorite when there is the need to perform real-time image feature extraction since it is a good trade off between quality and performance: from a comparison [13] with other popular and more precise feature types such as SURF and SIFT, extracting 1000 points in the same image takes about $15.3\,ms$ for ORB, about 14 times more for SURF and about 342 times more for SIFT. It uses an improved version of the FAST (Features from Accelerated Segment Test) key point and of the BRIEF (Binary Robust Independent Elementary Feature) descriptor, and it is because of these components that it can achieve impressive computational speeds while maintaining scale and rotation invariance during image transformation.

Oriented FAST Key Point

FAST is a type of corner point that is achieved by evaluating on a grayscale image the brightness distribution in a cluster of pixels. The sequence of operations is as follows:

1. Takes the brightness I_p of a pixel p in the image;
2. Sets a threshold T that can be chosen by the user (here 20%);
3. Selects 16 pixels on a Bresenham circle of radius 3 around the pixel p;
4. Evaluates if the central pixel p is a feature point by checking if there are enough consecutive points of the circle that have a brightness greater than $I_p + T$ or lower than $I_p - T$;
5. Repeats the previous four steps for each pixel in the image.

Since it only compares the pixels' brightness it is extremely fast, but has suboptimal repeatability and uneven distribution. Also, the FAST key point only provides the 2D position of the point. To address these limitations, ORB uses the improved Oriented FAST key point, which adds the description of both scale and rotation. The scale invariance is obtained by means of an image pyramid, as shown in Figure 2.5. The image is downsampled `numLevels` times with a certain `scaleFactor` so that different resolutions are achieved. This operation allows the same feature point to be identified even when the camera moves away from or closer to it.

The scale value at each level of decomposition is $\texttt{scaleFactor}^{(level-1)}$, where `scaleFactor` must be within the range $[0, \texttt{numLevels} - 1]$. Given an input image with raw resolution of $M \times N$, the size at each level or decomposition becomes

$$\frac{M}{\texttt{scaleFactor}^{(\texttt{level}-1)}} \times \frac{N}{\texttt{scaleFactor}^{(level-1)}}. \tag{2.7}$$

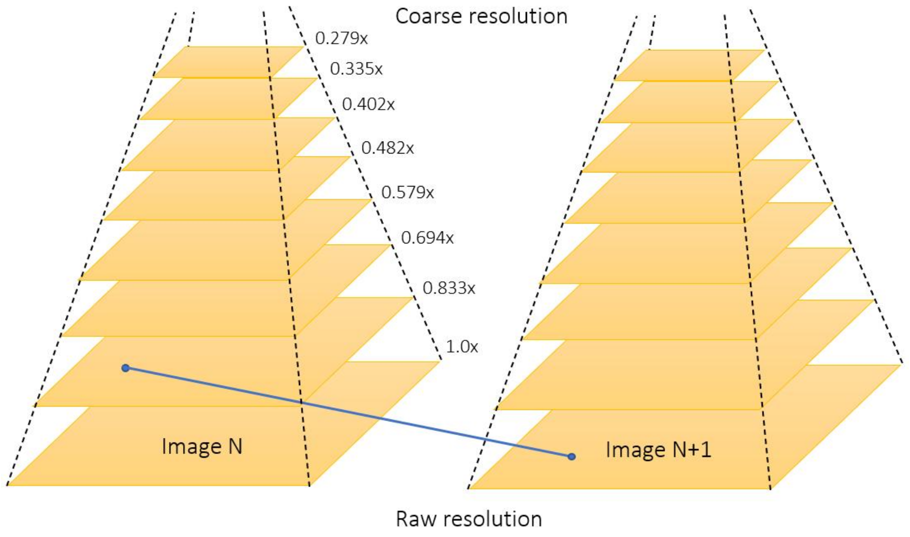


Figure 2.5: Pyramids for two subsequent images where $\texttt{numLevels} = 8$ and $\texttt{scaleFactor} = 1.2$. During the feature matching, images on different layers can be matched to obtain scale invariance.

The rotation of features is instead determined by the intensity centroid method. The gray centroid is the equivalent to the concept of center of mass, but instead of mass, pixel intensity is used.
The moment of a small image block B is defined as

$$m_{pq} = \sum_{x,y \in B} x^p y^q I(x,y), \quad p,q = \{0,1\} \tag{2.8}$$

from which it is possible to calculate the centroid of the image block

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right). \tag{2.9}$$

The vector $\vec{OC}$, where O is the geometric center of the image, has therefore direction

$$\theta = arctan\left(\frac{m_{01}}{m_{10}}\right). \tag{2.10}$$

BRIEF Descriptor

To completely characterize the feature it is now needed the descriptor component, which is derived from pixels surrounding a key point. Descriptors -also known as feature vectors- are necessary to describe and match features that are specified by a single point location.
The binary BRIEF descriptor is calculated for all key points and gathered in a $m \times n$ matrix, where m is the number of descriptors and n is the number of elements in the binary vector. This vector encloses the size relationship between two random pixels p and q near the key point: if $p > q$ then returns the value 1, otherwise, if $p < q$ it returns 0. The use of random pixels and binary encoding greatly increases the speed of this operation. Since the original BRIEF does not have rotation invariance, ORB uses an improved version where the direction information is retrieved from the Oriented FAST key point to calculate the Steer BRIEF feature after the rotation.

Thanks to the combination of the FAST and BRIEF components, ORB features are very efficient and well behaved under translation, rotation and scaling.
Matlab allows to detect ORB features through the function `detectORBFeatures`, while the descriptors are extracted through `extractFeatures`. A final expedient that is made is to use `selectUniform` to take a number of feature points `numPoints` - typically between 500 and 2500, depending on the resolution of the image - with the strongest metrics approximately distributed throughout the image. This prevents all points from being assembled in a limited region.

2.3.3 Feature Matching and 3D World Positions

Now that the feature points for the first couple of stereo images have been obtained, they are matched to find their 3D world locations. This operation can be divided into three steps:

1. Reconstruction of the 3D scene from the disparity map;
2. Feature matching between the stereo pair;
3. Localization of the ORB feature points in the 3D world map.

Reconstruction of the 3D scene from the disparity map

The disparity map is a greyscale image representing the apparent motion of the pixels between a pair of rectified stereo images due to the different position of the cameras in space (Fig. 2.7). Before generating this map, it is appropriate to first estimate the maximum disparity, which can be obtained from a red-cyan anaglyph, as shown in Figure 2.6. The rectified stereo images are superimposed with each other to compute the distance between the same pixels. The closer the object is to the camera, the greater the disparity will be. The measured value is used to define the disparity range `disparityRange` $= [MinDisparity, MaxDisparity]$.

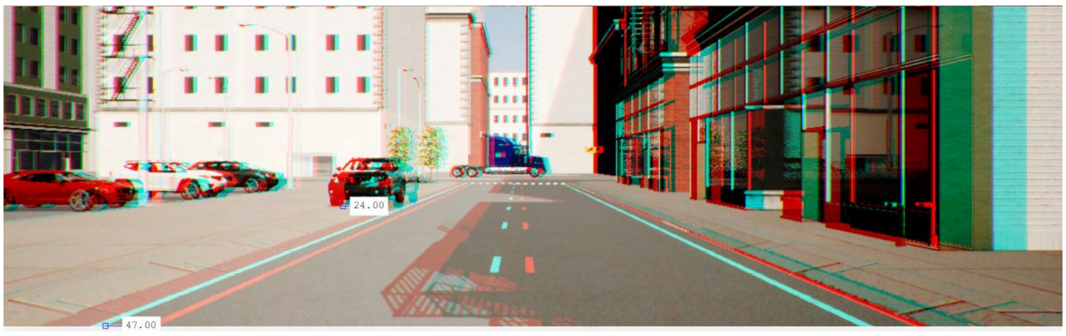


Figure 2.6: Red-cyan anaglyph of the stereo pair in figure 2.4 with the measure of the disparity for two generic points. The road line in the lower left corner was chosen to measure the maximum value, where the distance of coincident points corresponds to 47 pixels.

Next, the census transform (CT) of the image pair is performed and the binary strings associated with each pixel are used to evaluate the Hamming distance. This quantity refers to the number of different digits in two binary vectors and measures the minimum number of substitutions required to change one string into the other. The Matching Cost Matrix that stores the Hamming distance for each pixel pair is therefore obtained as a result. Finally, the pixel-wise disparity is computed from this matrix using the Semi-Global Matching (SGM) method in `disparitySGM` and the map is derived.

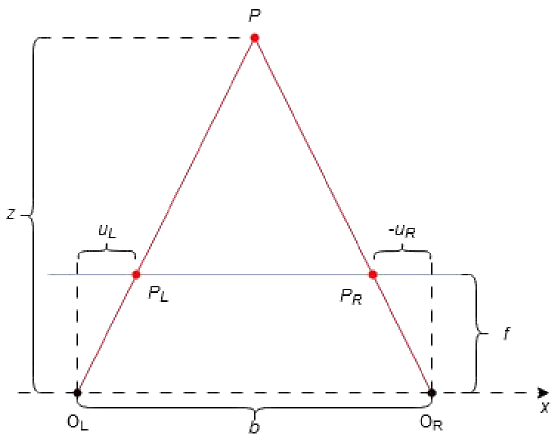


Figure 2.7: Geometric model for a stereo configuration, where $d = u_L - u_R$ is the disparity and the depth $z = f\,b/d$.

The newly obtained disparity map (Fig 2.8), together with the reprojection matrix, is employed in the function `reconstructScene` to reconstruct the 3D scene in terms of $[X, Y, Z]$ point coordinates corresponding to all the pixels of the image. These coordinates are relative to the optical center of the left camera and are stored in a $M \times N \times 3$ matrix, where $M \times N$ is the resolution of the rectified images.

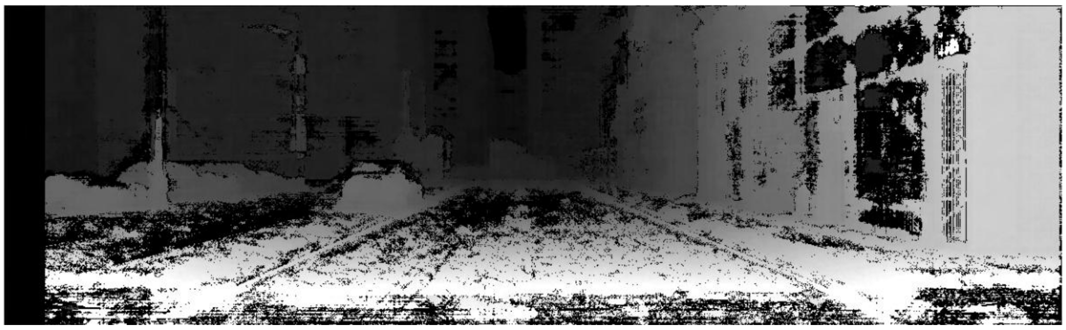


Figure 2.8: Disparity map obtained from the stereo pair and scaled for a value of `disparityRange` $= [0, 48]$ (it must be a multiple of 8).

Feature matching between the stereo pair

The ORB features in the stereo pair are matched so that they can be associated with the corresponding world points. The two sets of features are compared by means of an exhaustive method that computes the pair-wise distance between the BRIEF descriptors and it was set so that a pair of features is matched only if the distance between them is less than 40% from a perfect correspondence. Candidate matches are then evaluated to isolate only those pairs that are sufficiently close to the same epipolar line, meet the disparity range requirements and have nearly identical scale.

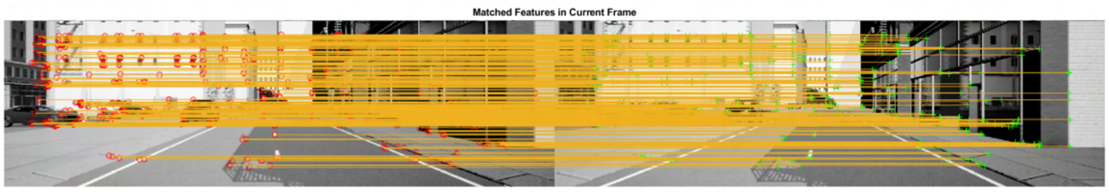


Figure 2.9: Matched features in the first grayscale stereo pair.

Localization of the ORB feature points in the 3D world map

Since there is only interest in the 3D points that correspond to the ORB features, two filtering operation extract from the map only the points that have the same pixel locations as the features and keep only those within a $200 \times baseline$ range in the z direction: this second passage excludes points that can not be valid features since their disparity is zero and their associated distance is set to infinity.
These points are finally projected on the initial camera pose so that they are placed correctly in the global map that will be updated to host also the points for the subsequent frames:

$$\mathbf{p}_{global,in} = \mathbf{p} \cdot \mathbf{R}_{in} + \mathbf{t}_{in} \tag{2.11}$$

where $\mathbf{p}$ is the vector with the $[X, Y, Z]$ point coordinates of the ORB features, $\mathbf{R}_{in}$ and $\mathbf{t}_{in}$ are respectively the rotation matrix and the translation values for the initial pose, and $\mathbf{p}_{global,in}$ is the point coordinates vector transformed for the global map.

2.4 Data Management and Visualization

Once the map has been initialized, it is necessary to manage data and allow visualization of the 3D map points and pose of the camera in the global map. *Matlab* makes use of two objects, `imageviewset` and `worldpointset`, to perform these tasks. They serve as a database to store all information associated with current and previous observations and they set the SLAM problem as factor graph. Understanding these two objects is critical to knowing how the data stream is managed and how subsequent observations are connected to improve pose estimation. Some concepts that will be explored in more detail as we progress with the description of the SLAM system are mentioned here, therefore please continue reading the following sections to have an appropriate insight.

`imageviewset` handles view attributes and pairwise connections between views of data. It encloses four properties:

– A scalar counter that keeps track of the number of views, *i.e.* number of key frames that have been identified;

– A scalar counter that keeps track of the number of pairwise connections between views;

– A three column table with the absolute pose of the camera, the feature vector with `numPoints` rows, and the feature points properties (location, metric, count, scale and orientation) for each key frame;

– A five column table where each row is descriptive of a connection. Column 1 and 2 identify which views are connected with each other, column 3 gives the relative pose of the second view with respect to the first one, column 4 encloses the information matrix that expresses the uncertainty of the measurement error, and finally column 5 holds a `numPoints` $\times$ 2 matrix with the indices of matched feature points between two views.

`worldpointset` stores correspondences between 3-D world points and 2-D image points across camera views. It encloses eight properties:

– A $M \times 3$ matrix with the coordinates for all 3D world points, where M is the number of world points;

– A row vector that identifies the views associated with the world points;

– A three column table that specifies the 3D to 2D point correspondences. For each world point are specified which views see that point and, within a view, what is the corresponding feature point;

– A $M \times 2$ matrix with the minimum and maximum distances from which a world point is observed;

– A $M \times 3$ matrix with the mean viewing direction of each world point (Fig. 2.10). Each row defines the vector that provides an estimate of the viewing angle from which that specific 3D point can be observed, mediated for all views that see that point. When a new view is introduced, the 3D points that can potentially be observed can be predicted based on this mean viewing direction and the distance range limits (see below);

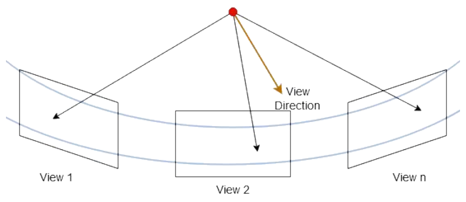


Figure 2.10: Mean viewing direction for n views. The two blue circles represent the distance range limits.

– A M column vector with the ID of the representative feature (given by the medoid of all the feature descriptors associated with the world point) descriptor index for each point;

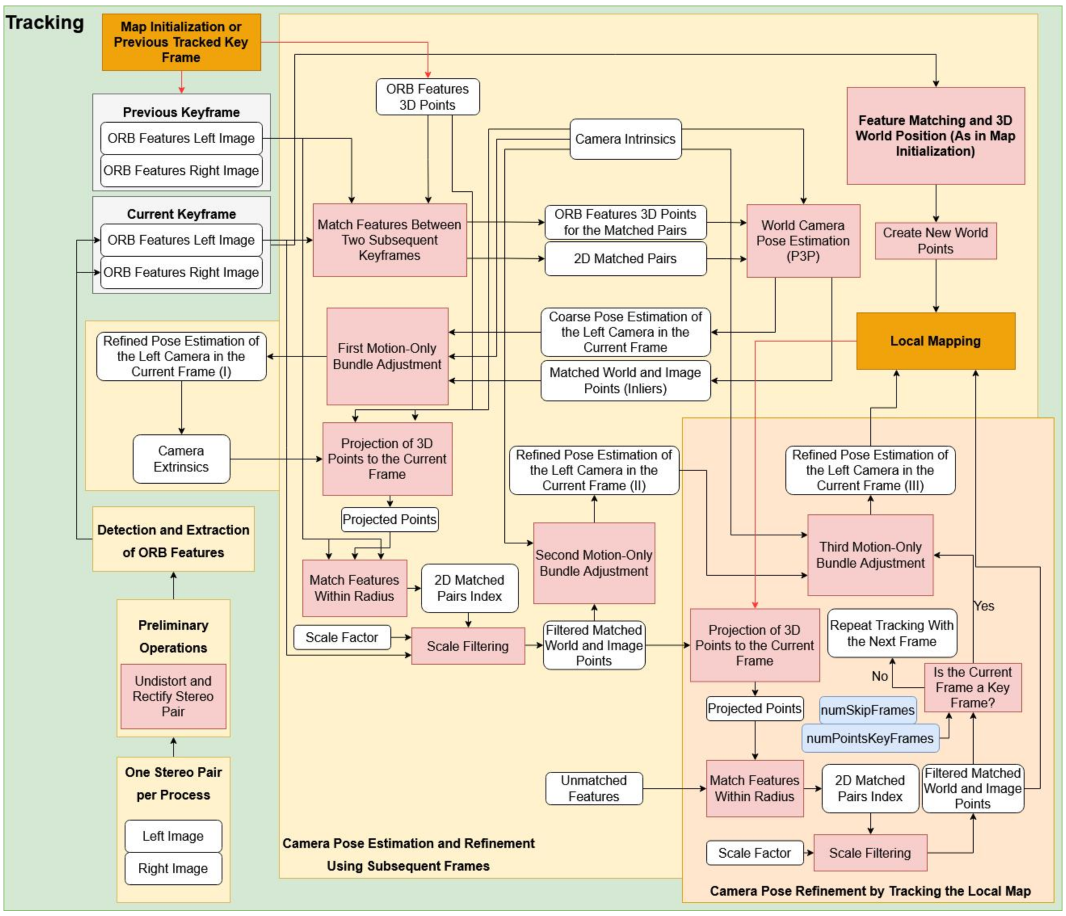
– A M column vector with the ID of the representative views for each point. A representative view is a view that contains the representative feature for each world point;

– A scalar that counts the number of world points.

Both of these objects grow in dimensions as the cameras move within the environment by adding new key frames. The previously existing values are also updated to improve the poses and map thanks to the numerous bundle adjustments that will be performed during Tracking and Local Mapping and during pose graph optimization.

2.5 Tracking

Tracking is at the core, together with the subsequent Local Mapping and Loop Closure, of the main loop of the system. It is an iterative process that is performed on each stereo pair provided, therefore takes place from when the map is initialized until when the frame that allows loop closure has been found. Figure 2.11 shows a summary diagram that encapsulates all the major steps necessary to reach its final goal, which is to identify if a frame is a key frame to be forwarded to the Local Mapping thread.



Tracking
Map Initialization or Previous Tracked Key Frame
Previous Keyframe
ORB Features Left Image
ORB Features Right Image
Current Keyframe
ORB Features Left Image
ORB Features Right Image
Refined Pose Estimation of the Left Camera in the Current Frame (I)
Camera Extrinsics
Detection and Extraction of ORB Features
Preliminary Operations
Undistort and Rectify Stereo Pair
One Stereo Pair per Process
Left Image
Right Image
ORB Features 3D Points
Camera Intrinsics
Feature Matching and 3D World Position (As in Map Initialization)
Match Features Between Two Subsequent Keyframes
ORB Features 3D Points for the Matched Pairs
2D Matched Pairs
World Camera Pose Estimation (P3P)
Create New World Points
First Motion-Only Bundle Adjustment
Coarse Pose Estimation of the Left Camera in the Current Frame
Matched World and Image Points (Inliers)
Local Mapping
Projection of 3D Points to the Current Frame
Projected Points
Match Features Within Radius
2D Matched Pairs Index
Scale Factor
Scale Filtering
Refined Pose Estimation of the Left Camera in the Current Frame (II)
Second Motion-Only Bundle Adjustment
Filtered Matched World and Image Points
Unmatched Features
Camera Pose Estimation and Refinement Using Subsequent Frames
Refined Pose Estimation of the Left Camera in the Current Frame (III)
Third Motion-Only Bundle Adjustment
Yes
Projection of 3D Points to the Current Frame
Repeat Tracking With the Next Frame
No
Is the Current Frame a Key Frame?
Projected Points
numSkipFrames
numPointsKeyFrames
Match Features Within Radius
2D Matched Pairs Index
Filtered Matched World and Image Points
Scale Factor
Scale Filtering
Camera Pose Refinement by Tracking the Local Map

Figure 2.11: Summary diagram of all the main operations that take place during Tracking.

The first step of Tracking is similar to what was done during Map Initialization, but with a new pair of stereo frames at each iteration. Therefore, please refer to the previous section for details concerning the Preliminary Operations and the Detection and Extraction of ORB features.

The following is a description of the steps required to perform Camera Pose Estimation and Refinement, and Key Frame Detection.

2.5.1 Camera Pose Estimation and Refinement

In order to obtain a first estimation of the camera pose at a given instant, it is first necessary to match corresponding features from two consecutive frames. Feature matching is therefore now performed between two successive frames of the left camera instead of between frames of the stereo pair. The `findWorldPointsInView` function is used to return the indices of world points observed in the previous view and of the associated features. These features indices are then matched with the corresponding indices in the current frame, so that the same feature, associated with the same 3D point, is recognized. As expected, only some of the features that were found in the first frame will find a match with those in the second view because of the pose variation that occurred. The criteria by which matching takes place are the same as those seen during Map Initialization, with a 40% match threshold and an high ratio threshold for rejecting ambiguous matches. Now that the matched image points and matched world points are known, there is everything needed to estimate the camera pose.

3D-2D Perspective-n-Point (PnP)

The estimation of camera motion and pose is a classic problem in visual odometry that can be solved with many different approaches, depending on the set of information that are available. In this implementation, we tracked a set of 3D points in world coordinates and their corresponding 2D projection on the image plane, therefore we are dealing with a Perspective-n-Point (PnP) problem.
The goal that wants to be achieved can be better understood by considering the perspective projection model of the camera:

$$s_i \, \mathbf{p}_{uv,\,i} = [\mathbf{R}|\mathbf{t}] \, \mathbf{P}_{w,\,i}, \tag{2.12}$$

$$s_i \begin{pmatrix} u_i \\ v_i \\ 1 \end{pmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{pmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{pmatrix} \tag{2.13}$$

where $\mathbf{P}_{w,\,i}$ is the point in homogeneous world coordinates, $\mathbf{p}_{uv,\,i}$ is the corresponding homogeneous image point, $\mathbf{R} \in SO(3)$ and $\mathbf{t} \in \mathbb{R}^3$ compose the projection matrix $\mathbf{C}$ and are the unknowns rotation matrix and translation vector (camera extrinsics). Finally, s is the scale factor for the image point. In this expression $\mathbf{p}_{uw,\,i}$ uses normalized plane coordinates and neglects the influence of the known intrinsic matrix

K.
The system can be solved with the Direct Linear Transformation (DLT) method by solving the linear equations as:

$$\begin{bmatrix} \mathbf{P}_{w,1}^T & 0 & -u_1\mathbf{P}_{w,1}^T \\ 0 & \mathbf{P}_{w,1}^T & -v_1\mathbf{P}_{w,1}^T \\ \vdots & \vdots & \vdots \\ \mathbf{P}_{w,n}^T & 0 & -u_n\mathbf{P}_{w,n}^T \\ 0 & \mathbf{P}_{w,n}^T & -v_n\mathbf{P}_{w,n}^T \end{bmatrix} \begin{pmatrix} r_{11} \\ r_{12} \\ \vdots \\ r_{33} \\ t_3 \end{pmatrix} = 0\,, \tag{2.14}$$

where $[R|\mathbf{t}]$ has been expresses as a vector of dimension 12 and a total of n feature points are being considered. Explicating u_i and v_i it can be easily shown that each feature point provides two linear constraints on the vector in Equation 2.14, therefore the linear solution can be achieved by at least six pairs of matching points. The precision of this method can therefore be increased increasing the number of points and searching for a lest-square solution of the overdetermined equation. Interestingly, DLT also allows to estimate the matrix **K** if it is not know but this comes at the expense of reduced accuracy. On the downside, the elements in $[R|\mathbf{t}]$ are treated as 12 unrelated unknowns, therefore this solution may not satisfy the $SO(3)$ constrain on the rotation matrix and it becomes necessary to look for the best approximation by means of QR decomposition.

Perspective-three-Point (P3P)

If $n = 3$, a popular alternative method to the DLT for solving the PnP problem is given by the Perspective-three-Point (P3P) algorithm [7]. It exploits only three pairs of 3D-2D matching points and establishes their geometric relationship. This algorithm is often chosen in V-SLAM to pair with a subsequent least-square optimization to improve the pose. Since it uses less input data and offers a more efficient solving procedure, the P3P followed by a bundle adjustment has been used in this implementation.

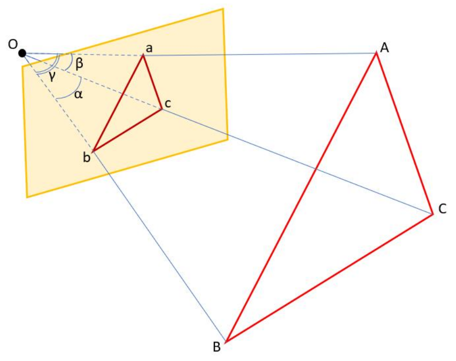


Figure 2.12: The P3P Problem.

Consider Figure 2.12: let O be the principal camera point; A, B, C the 3D points in world coordinates while a, b, c the projections of those points on the camera image plane.

Using the law of cosines on the triangles AOB, AOC, BOC, the P3P equation system is:

$$\begin{cases} |\bar{OB}|^2 + |\bar{OC}|^2 - |\bar{OB}||\bar{OC}|\,2cos(\alpha) - |\bar{BC}| = 0 \\ |\bar{OC}|^2 + |\bar{OA}|^2 - |\bar{OA}||\bar{OC}|\,2cos(\beta) - |\bar{AC}| = 0 \\ |\bar{OA}|^2 + |\bar{OB}|^2 - |\bar{OA}||\bar{OB}|\,2cos(\gamma) - |\bar{AB}| = 0 \end{cases} \tag{2.15}$$

where $\bar{OA}$, $\bar{OB}$, $\bar{OC}$ are the distances of the 3D world points from the principal camera point -*i.e.* the coordinates of the 3D points in the camera coordinate system- and are the unknowns to be found.
These equations can be easily reformulated to make explicit the quadratic nature of the problem as shown in [7], obtaining the following

$$\begin{cases} (1-u)\,y^2 - ux^2 - 2cos(\alpha)\,y + 2uxy\,cos(\gamma) + 1 = 0 \\ (1-w)\,x^2 - wy^2 - 2cos(\beta)\,x + 2wxy\,cos(\gamma) + 1 = 0 \end{cases} \tag{2.16}$$

Since the position of the 2D points in the image are known, the three cosine angles can be calculated by exploiting the relationship between the triangles *aOb*-*AOB*, *bOc*-*BOC* and *aOc*-*AOC*. $u = |\bar{BC}|^2/|\bar{AB}|^2$ and $w = |\bar{AC}|^2/|\bar{AB}|^2$ can instead be calculated by the known world coordinates of A, B, C. The two unknown values are therefore $x = |\bar{OA}|/|\bar{OC}|$ and $y = |\bar{OB}|/|\bar{OC}|$ and the resolution of these equations allows to derive the position of the 3D world points in the camera reference system. It should be noted that the analytical solution of these equations is not straightforward and can be obtained with Wu's elimination method. Also, the P3P problem as presented here has at most four physical solutions and the one that estimates the correct position of the points with higher probability can be extracted by means of a fourth verification point.
Once x and y are known, the problem becomes a 3D-3D pose estimation problem with matching information, which can be easily solved with the Iterative Closest Point (ICP) approach to find the desired value of the pose in terms of $\mathbf{R}$ and $\mathbf{t}$.

Since there are many more ORB features than those needed for a single iteration of the P3P method, the robustness of the solution is increased by eliminating outlier correspondences using the M-estimator Sample Consensus (MSAC) algorithm [17]. MSAC uses the same sampling strategy as the popular RANSAC algorithm, but chooses the solution that maximize the likelihood function rather than just the number of inliers, allowing to obtain equal or superior results to those of RANSAC.

By setting the total number of trials (dependent on the number of 2D and 3D points, confidence for finding the maximum number of inliers and the reprojection error threshold) it eliminates outlier correspondences and use only the inliers points to compute the camera pose. What has been described in this section is implemented in the subroutine `estworldpose`.
As a result of this operation, the first estimate of the matrix $[\mathbf{R}|\mathbf{t}]$ is obtained.

Motion-only Bundle Adjustment

Bundle adjustment is a popular iterative method used to solve batch state estimation problems with a non linear least-squares approach. In general, it refines simultaneously both the camera poses and the 3D world points positions by minimizing the reprojection errors so that the projected 2D features match the detected results. In contrast to the more general case of bundle adjustment that will be addressed in Section 2.6.2, however, it is accomplished in this Tracking thread using a variation of the *Levenberg-Marquardt* optimization algorithm in which the 3D world points are fixed, therefore only the camera pose is improved and an initial improvement in the values of $\mathbf{R}$ and $\mathbf{t}$ is returned. The estimate for the pose obtained from the P3P algorithm (which serves now as optimized initial value), together with the matched world points $\mathbf{P}_{w\,,i}$, the corresponding matched image points $\mathbf{p}_{uv\,i}$ and the intrinsics matrix $\mathbf{K}$ are all considered together to perform this pose refinement.
According to the pinhole camera model described in Section 1.6.2, the relationship between the 2D pixel position and the 3D world position is here:

$$s_i\,\mathbf{p}_{uv,\,i} = \mathbf{K}(\mathbf{R}\mathbf{P}_{w,\,i} + \mathbf{t}) = \mathbf{K}\mathbf{T}\mathbf{P}_{w,\,i}, \tag{2.17}$$

Due to the unrefined value of $\mathbf{T}$ and the noise of the observation points, there is a residual in each $i - th$ equation which can be sum up to construct a nonlinear least-square problem to minimize to find the most possible camera pose:

$$\mathbf{T}^* = \arg\,\min_{\mathbf{T}} \sum_{i=1}^{n} \rho\left(\|\mathbf{p}_{uv\,i} - \pi_s(\mathbf{T}\mathbf{P}_{w\,,i})\|^2_{\mathbf{Q}}\right) \tag{2.18}$$

where ρ is the robust Huber cost function, $\mathbf{Q}$ is the covariance matrix associated to the scale of the point and π_s is the projection function for the rectified stereo pair, defined as:

$$\pi_s\left(\begin{bmatrix}X\\Y\\Z\end{bmatrix}\right)=\begin{bmatrix}f_x\frac{X}{Z}+c_x\\f_y\frac{Y}{Z}+c_y\\f_x\frac{X-b}{Z}+c_x\end{bmatrix} \tag{2.19}$$

The residual term in Equation 2.18 is the reprojection error $\mathbf{e}(\mathbf{x})$, defined as the distance between the detected and the reprojected point.
The presence of the robust Huber kernel ρ is aimed at mitigating an important limitation with the minimization of the sole $\mathcal{L}_2$ loss function given by the square of the reprojection error.

Despite the efforts in performing a correct matching between feature points, mismatches are possible and are associated with large errors that are treated no differently from any other error term. In graph terms, it is as if wrong edges were added. Because of these large errors, the optimization algorithm tends to adjust the estimated values of the nodes connected by this edge to make them comply with the incorrect match. This detrimental behavior is heavily accentuated by the fact that the $\mathcal{L}_2$ norm grows quadratically. The Huber kernel ρ is therefore introduced to guarantee that the error of each edge can not affect too severely the other edges.

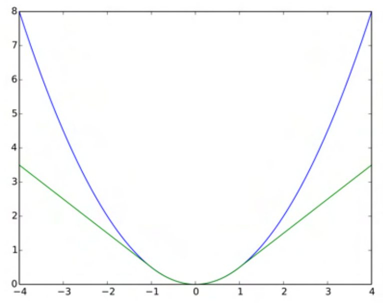


Figure 2.13: $\mathcal{L}_2$ loss in blue and Huber kernel in green, with $\delta = 1$ (*Image courtesy of Wikipedia*).

$$\rho(e)=\begin{cases}\frac{1}{2}e^2 & \text{when } |e|\leq\delta,\\ \delta(|e|-\frac{1}{2}\delta) & \text{otherwise}\end{cases} \tag{2.20}$$

where e is the residual term. As illustrated in Fig 2.13, if the error is grater than the threshold δ, then the function grows linearly instead of quadratically, thereby limiting its influence.

Returning to Equation 2.18, in order to apply the *Levenber-Marquardt* method, it is necessary to calculate the derivative of each error term with respect to the optimization variable knowing that the first-order Taylor expansion is:

$$\mathbf{e}(\mathbf{x}+\Delta\mathbf{x})\approx\mathbf{e}(\mathbf{x})+\mathbf{J}^T\Delta\mathbf{x}, \tag{2.21}$$

where the Jacobian $\mathbf{J}^T$ is a 2×6 matrix since the reprojection error $\mathbf{e}(\mathbf{x})$ has only two dimensions in non-homogeneous coordinates and the camera pose $\mathbf{x}$ has six dimensions.
Since this is a motion-only bundle adjustment, only the pose variable derivative within $\mathbf{J}^T$ must be obtained. It can be found with the perturbation model considering the derivative of the change of $\mathbf{e}$ with respect to $\delta\boldsymbol{\xi}$. For the $i-th$ equation, by using the chain rule:

$$\frac{\partial \mathbf{e}}{\partial \delta\boldsymbol{\xi}} = \lim_{\delta\boldsymbol{\xi}\to 0} \frac{\mathbf{e}(\delta\boldsymbol{\xi} \oplus \boldsymbol{\xi}) - \mathbf{e}(\boldsymbol{\xi})}{\delta\boldsymbol{\xi}} = \frac{\partial \mathbf{e}}{\partial \mathbf{P}'_i}\frac{\partial \mathbf{P}'_i}{\partial \delta\boldsymbol{\xi}}, \tag{2.22}$$

where the first term on the right side is the derivative of the error with respect to the projection point and the second term is the derivative of the transformed point with respect to the perturbation. It can be shown that by developing the two terms and multiplying them together, the Jacobian matrix $\mathbf{J}^T$ becomes:

$$\frac{\partial \mathbf{e}}{\partial \delta\boldsymbol{\xi}} = -\begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} & -\frac{f_x X' Y'}{Z'^2} & f_x + \frac{f_x X'^2}{Z'^2} & -\frac{f_x Y'}{Z'} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} & -f_y - \frac{f_y Y'^2}{Z'^2} & \frac{f_y X' Y'}{Z'^2} & \frac{f_x X'}{Z'} \end{bmatrix}. \tag{2.23}$$

This expression describes the first-order derivative of $\mathbf{e}$ with respect to the left perturbation model and the negative sign in front is because it is defined as the observed value minus the predicted value.

Further Refinement Using Subsequent Frames and Stereo Pairs

The camera pose estimation process continues with a series of optimizations and filters aimed at further reducing the error in $\mathbf{T}$.
The newly obtained camera extrinsics values are used to project again the world points on the current frame with `world2img` and search for additional matches with the map points in the previous key frame. Note that in the first iteration of Tracking the keyframe considered is simply the first one, being the one used for Map Initialization, while later this operation is done for only the frames identified as key frames in the previous iteration. The additional matches are found by performing feature matching within a scaled radius centered on the projected points. In this way the matching is more efficient because not all descriptors have to be compared with each other, but only those of the points close to the projections found are taken into account. These new matches are then filtered again so that only those that have the correct scale are added to the list.

In much the same way as before, now takes place the second motion-only bundle adjustment that further refines the pose and outputs a new, improved value for **R** and **t**.

So far only the ORB Features of the current left frame have been employed during Tracking. Since a stereo configuration is being used, the right frame is also available and can be exploited to identify additional new 3D points. This is done with a feature matching between the current stereo pair similar to the one seen during Map Initialization. The 3D points that correspond to the ORB features here matched are then compared with those previously extracted with the match between two consecutive frames. If they are indeed new points they are added to the 3D points set to be forwarded to Local Mapping. Although this operation is not necessary (consider the case of mono SLAM), it helps to increase the robustness of the Tracking process by providing more points to be used during bundle adjustment.

Camera Pose Refinement by Tracking the Local Map

A third refinement of the camera pose in the current view is performed only if the frame has been identified as a key frame. This operation employs the `imageviewset` and `worldpointset` objects that have been updated during Local Mapping for the previous iteration, as well as the matched world and image points from the current view. First of all, all 3D points are projected to the current frame and are discarded if: (1) they fall outside the view; (2) the angle between the mean viewing direction and the current viewing direction for the 3D point is above a certain threshold; (3) the distance from the map point to the camera center is out of its scale invariance region. If the point passes these checks, its scale in the view is computed and the descriptor is compared with the still unmatched ORB features, at the predicted scale and within a certain radius from the projection on the pixel plane. The best correspondence is thus associated with the map point. The resulting matched world and image points are employed to evaluate if the current frame is a key frame. In the event of a negative outcome, the tracking operation starts again with the next frame, while if the key frame is recognized the third motion-only bundle adjustment takes place.

2.5.2 Key Frame Detection

There is now to determine under which conditions the currently analyzed frame can become a key frame. This step is very important and can be considered the

ultimate goal of the entire Tracking process since only the key frames will take part in the Local Mapping.
For a frame to become a key frame, the following two conditions must be met:

1. `numSkipFrames` frames have passed since the last key frame or the current frame tracks fewer than `numPointsKeyFrame` map points;

2. The map points tracked by the current frame are fewer than 90% of points tracked by the reference key frame.

The correct choice of the parameters `numSkipFrames` and `numPointsKeyFrame` in the first condition is critical to the proper functioning of the entire method: if too many frames are skipped there is a risk that important frames will be lost and not enough matches will be found during the subsequent tracking process; on the other hand, a too low value will significantly reduce the real-time capabilities of the SLAM method. In addition, there is also the possibility that even if a very low value of `numSkipFrames` is chosen, the accuracy in recreating the camera trajectory does not increase or even decreases. Figure 2.14 is intended to intuitively show this fact: notice how with `numSkipFrames` = 2 more key frames are selected, but they are not the same as those taken for `numSkipFrames` = 3. Frame 5 and 9 may have features more relevant to the SLAM algorithm than those present in frames 4, 7 or 10, therefore, although counterintuitively, accuracy can be adversely affected by too low a value for `numSkipFrames`.

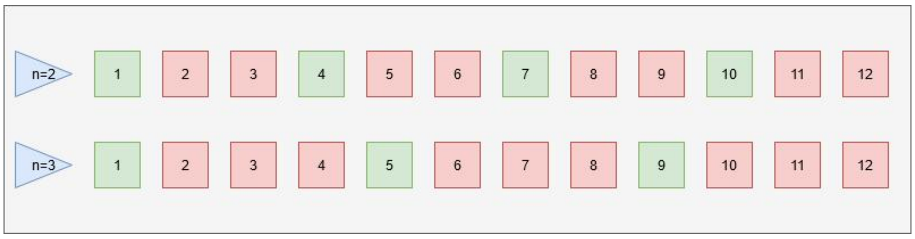


Figure 2.14: Evaluation of `numSkipFrames` for the choice of key frames.

In my experience with the simulations done, the condition on `numSkipFrames` is met most of the time and more rarely the procedure resorts to `numPointsKeyFrame`. A first criterion for choosing `numSkipFrames` lies in knowing the frame rate at which the footage is captured: in the case of the KITTI dataset, and thus also the virtual simulations that I conducted, the videos are shot at the only 10 fps, therefore the optimal values for `numSkipFrames` found were all low, between 2 and 5. Higher values result in the loss of crucial key frames.

2.6 Local Mapping

The iterative process that constitutes the main loop continues with Local Mapping (Fig. 2.15). Local Mapping is responsible for updating the two `imageviewset` and `worldpointset` objects with the views and map points that are refined starting from the key frames. The pose and map that result form this operation will define the estimated trajectory for the entire route taken by the cameras before loop closure and the final optimization.
When a new key frame has been recognized during Tracking, it is forwarded to the Local Mapping thread where it is immediately added to the two `imageviewset` and `worldpointset` objects to update the properties within. The indices of the map points observed in this new key frame are then compared with those of the previous key frames during a culling operation to ensure that `worldpointset` contains as few outliers as possible due to spurious data association: if a map point is not observed in at least three key frames it is discarded. Also, the map points that had been created by matching the features in the stereo pair during the final stages of Tracking are now added to the list and create new 3D to 2D correspondences. Notice that to minimize outliers a check is made with all map points available obtained from sequence tracking, stereo disparity and triangulation from the previous key frames. The connection property within `imageviewset` returns all connections that have been identified up to the current key frame, *i.e.* all key frames that have at least one pair of matched feature points. This covisibility information can be expressed as a *covisibility graph* where each node is a key frame and an edge between two key frames exists if they share observation of the same map point. The weight of an edge is the number of shared map points. A table with these connected views is generated by specifying the parameter `MinNumMatches` that defines the minimum number of matched feature points for a connection to be valid. If `MinNumMatches` $= 0$ then all connections are valid, while for higher values the weakest connections are not considered. By comparing the current view with those connected with it, new map points are extracted. To do this, an iterative process for each connected view is initiated to triangulate the features in two views (yellow box in Figure 2.15).

In order to keep contained the number of key frames, a local culling operation is performed to detect and eliminate redundant key frames. All key frames in the *covisibility graph* whose 90% of the map points have been in at least other three key frames in the same or finer scale are discarded. This expedient has a double advantage on the functioning of the algorithm: firstly, bundle adjustment is lightened since there are less key frames it has to process; secondly, lifelong operation in the same environment can be achieved since the number of key frames will not grow unbounded, unless the visual content of the scene changes.

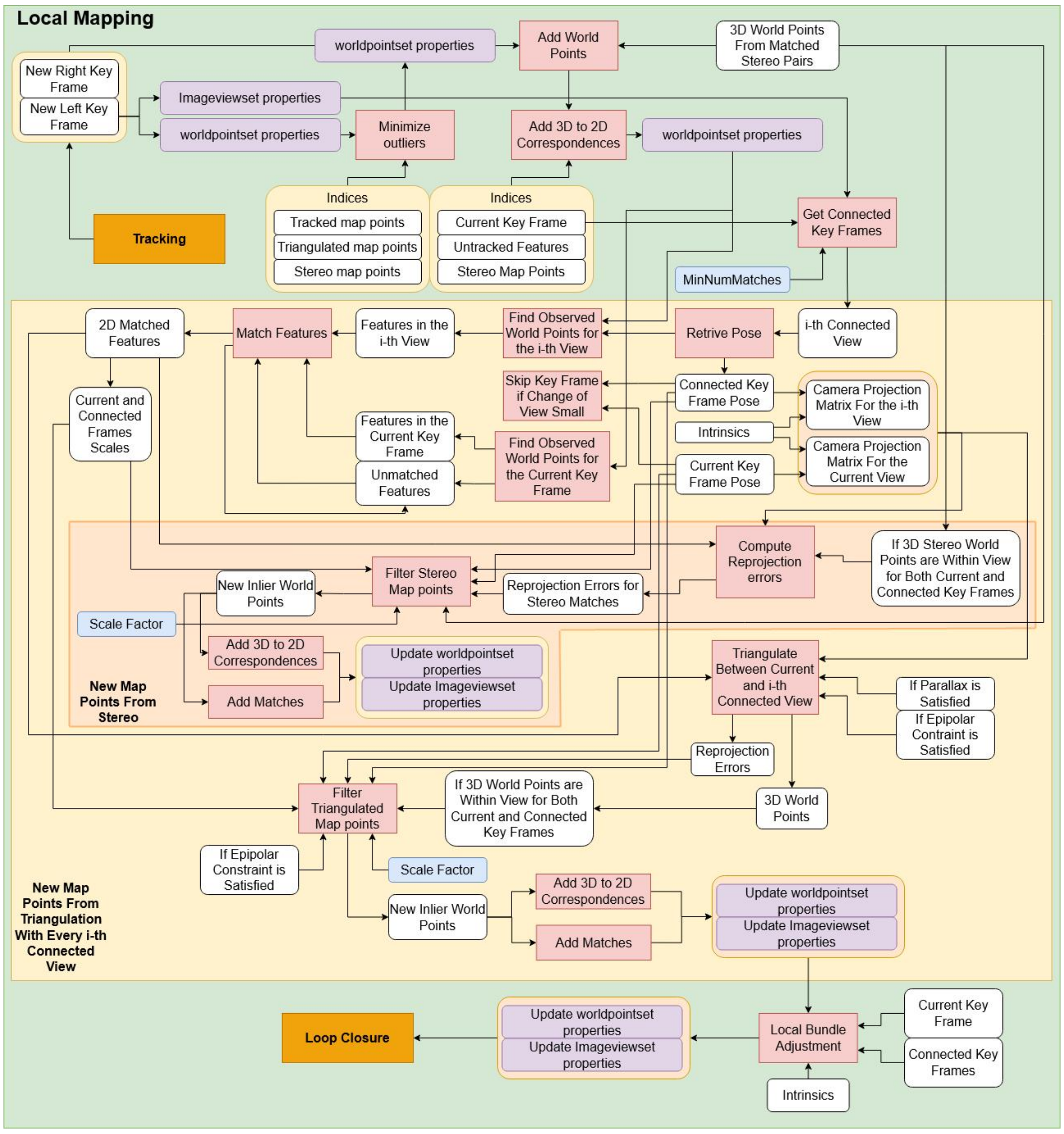

Local Mapping
New Right Key Frame
New Left Key Frame
Tracking
worldpointset properties
Imageviewset properties
worldpointset properties
Minimize outliers
Add World Points
3D World Points From Matched Stereo Pairs
Add 3D to 2D Correspondences
worldpointset properties
Indices
Tracked map points
Triangulated map points
Stereo map points
Indices
Current Key Frame
Untracked Features
Stereo Map Points
Get Connected Key Frames
MinNumMatches
2D Matched Features
Match Features
Features in the i-th View
Find Observed World Points for the i-th View
Retrive Pose
i-th Connected View
Current and Connected Frames Scales
Skip Key Frame if Change of View Small
Connected Key Frame Pose
Intrinsics
Current Key Frame Pose
Camera Projection Matrix For the i-th View
Camera Projection Matrix For the Current View
Features in the Current Key Frame
Unmatched Features
Find Observed World Points for the Current Key Frame
Compute Reprojection errors
If 3D Stereo World Points are Within View for Both Current and Connected Key Frames
Filter Stereo Map points
New Inlier World Points
Reprojection Errors for Stereo Matches
Scale Factor
Add 3D to 2D Correspondences
Add Matches
Update worldpointset properties
Update Imageviewset properties
New Map Points From Stereo
Triangulate Between Current and i-th Connected View
If Parallax is Satisfied
If Epipolar Contraint is Satisfied
Reprojection Errors
3D World Points
If 3D World Points are Within View for Both Current and Connected Key Frames
Filter Triangulated Map points
If Epipolar Constraint is Satisfied
Scale Factor
New Inlier World Points
Add 3D to 2D Correspondences
Add Matches
Update worldpointset properties
Update Imageviewset properties
New Map Points From Triangulation With Every i-th Connected View
Loop Closure
Update worldpointset properties
Update Imageviewset properties
Local Bundle Adjustment
Current Key Frame
Connected Key Frames
Intrinsics

Figure 2.15: Summary diagram of all the main operations that take place during Local Mapping.

2.6.1 New Map Points From Triangulation

To estimate the spatial position of the feature points it is possible to use the camera motion: the same point observed from two different views can provide the distance from the optical center of the cameras. Consider Figure 2.16, referring to the camera motion $[\mathbf{R}|\mathbf{t}]$ from the connected frame (left) to the current key frame (right), the relationship between the normalized coordinates of two feature points $\mathbf{p}_{c,1}$ and $\mathbf{p}_{c,2}$ is made explicit:

$$s_2\mathbf{p}_{c,2} = s_1\mathbf{R}\mathbf{p}_{c,1} + \mathbf{t}. \tag{2.24}$$

$[\mathbf{R}|\mathbf{t}]$ and the normalized coordinates are known form Tracking, therefore the unknowns are the depth s_1 and s_2 of the feature points which yields the 3D location of the respective world point P.

`Triangulate` finds the position of the 3D points referred to the world coordinate system given by the the projection matrices $\mathbf{C}_1$ and $\mathbf{C}_2$ derived from the poses of the two views, which map P in homogeneous coordinates onto the corresponding image points. The obtained world points are deemed valid if the resulting scale factors are positive. *Matlab* documentation does not specify which method is used here to accomplish triangulation, however a generic method to perform triangulation can be described with Equation 2.25, where $\sim$ specifies that the result must be satisfied minus a multiplicative constant since homogeneous coordinates are being used, and $\mathbf{P}$ is the homogeneous representation of the 3D point.

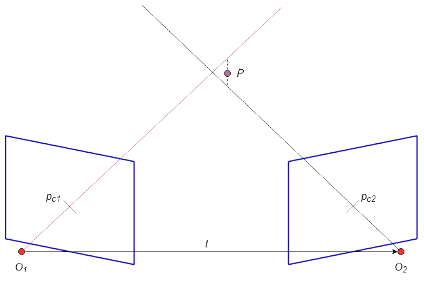


Figure 2.16: Use of triangulation to estimate the depth of point P.

$$\mathbf{P} \sim f(\mathbf{p}_{c,1}, \mathbf{p}_{c,2}, \mathbf{C}_1, \mathbf{C}_2). \tag{2.25}$$

In the ORB-SLAM2 system, a map point can be classified as "close" or "far" depending on its depth with respect to the baseline of the configuration. Close points can be effectively triangulated as depth is accurately estimated and and provide scale, translation and rotation information. Far points, on the other hand, can provide good rotation information but not so good scale and translation information. For this reason, far points are triangulated when they are supported by multiple views.

Another aspect that must be paid attention to when performing triangulation is the so-called triangulation contradiction (Fig. 2.17). Triangulation is only possible because of relative translation between two views. When translation is small, the uncertainty in depth estimation is higher because of the higher sight angle variation associated with the change in position of the world point. For this reason, by comparing the pose for the two views it

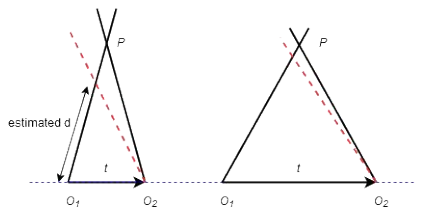


Figure 2.17: The contradiction of triangulation.

is established if the change of view is too small (< `baseline`) and if the frames are to skip before they take part in triangulation. On the other hand, very large translations could cause significant variations in the image's appearance, which will in turn make feature matching more difficult. A check on parallax and the epipolar constraint allow for better control on the matched pairs. The epipolar constraint can be expressed in a compact form as:

$$\mathbf{p}_{c2}^T \mathbf{E} \mathbf{p}_{c1} = \mathbf{p}_2^T \mathbf{F} \mathbf{p}_1 = 0, \tag{2.26}$$

where $\mathbf{E} = \mathbf{t}^\wedge \mathbf{R}$ is the essential matrix, $\mathbf{F} = \mathbf{K}^{-T} \mathbf{E} \mathbf{K}^{-1}$ is the fundamental matrix, $\mathbf{p}_1 = \mathbf{K}\mathbf{p}_{c1}$ and $\mathbf{p}_2 = \mathbf{K}\mathbf{p}_{c2}$. This equation states that O_1, O_2, P all belong to the epipolar plane. Due to noise this condition is not perfectly met, therefore it is checked that the image points in the connected frame are sufficiently close to the epipolar line and far from the current key frame epipole.
After triangulation, there is a further check on the newly obtained 3D points to verify that they are within view for both frames and, if they are, they take part in a filtering operation to extract the final inlier world points resulting from local mapping. A point is recognized as inlier if belongs to both views, the reprojection errors are below a certain threshold proportional to the scale and if the scale is valid.

In Figure 2.15 can also be seen an orange box within the yellow one. It encloses a process similar to the one just described but instead of using the points obtained from triangulation it is dedicated to the 3D stereo world points extracted from disparity of the stereo images. After computing the reprojection errors for these points they are filtered with the same subroutine used for triangulated points and they are also used to update the `imageviewset` and `worldpointset` objects.

2.6.2 Local Bundle Adjustment

Bundle Adjustment has already been introduced in Section 2.5.1 for the particular case in which only the camera pose is improved. During Local Mapping however, it is used to refine both the pose and the map to derive the final estimate of these values before loop closure is detected.
Global (or Standard) Bundle Adjustment (GBA) uses all map points and poses that have been obtained up to the current key frame to form the objective function. This allows for greater accuracy in the reconstruction of the scene, but it can quickly become unsuitable for real-time applications because of the computational cost associated with the fast growing number of variables typical of a real-world SLAM

problem.
For this reason, a Local Bundle Adjustment (LBA) is performed as an alternative, where the estimated parameters are the poses of only a limited number of frames and the world points that have at least one detected projection into these frames (Fig. 2.18). As a result, the computational load is significantly lightened, at the slight expense of estimation accuracy.

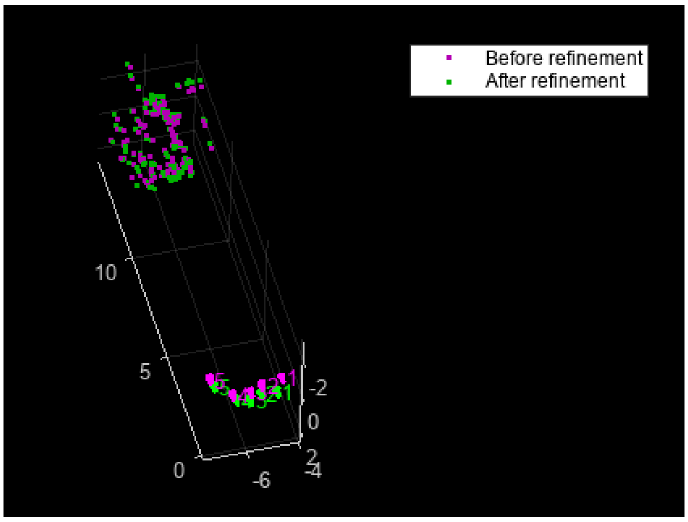


Figure 2.18: Example of BA. Both the camera poses and points location are optimized. (Image courtesy of *Mathworks*.)

Referring to the probabilistic SLAM problem described in Chapter 1, the observation equation can be written here as a non linear function:

$$\mathbf{z}_{ij} = h(\mathbf{T}_i, \mathbf{P}_{w,j}), \tag{2.27}$$

and states that the observation data given by the pixel coordinate $\mathbf{z}_{ij} \triangleq [u_s, v_s]^T$ is generated by observing the space point $\mathbf{P}_{w,j}$ at the pose $\mathbf{T}_i$. In terms of least-squares, the error of this observation:

$$\mathbf{e}_{ij} = \mathbf{z}_{ij} - h(\mathbf{T}_i, \mathbf{P}_{w,j}), \tag{2.28}$$

expresses the overall cost function:

$$\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n} \|\mathbf{e}_{ij}\|^2 = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n} \|\mathbf{z}_{ij} - h(\mathbf{T}_i, \mathbf{P}_{w,j})\|^2. \tag{2.29}$$

The solution of this least-squares problem, again with the *Levenberg-Marquardt* algorithm, corresponds to the realization of bundle adjustment.

The LBA in ORB-SLAM2 optimizes a set of covisible key frames $\mathcal{K}_L$ and all points $\mathcal{P}_L$ seen within them. All other key frames $\mathcal{K}_F$ that also see those points but are not connected to the current key frame are included in the optimization but remain fixed. If $\mathcal{X}_K$ is the set of matches between points in $\mathcal{P}_L$ and key points in the *k-th* key frame, then the nonlinear least-squares problem to minimize is given by:

$$\{\mathbf{P}_i, \mathbf{R}_l,\, \mathbf{t}_l | i \in \mathcal{P}_L,\, l \in \mathcal{K}_L\} = \arg \min_{\mathbf{P}_i, \mathbf{R}_l, \mathbf{t}_l} \sum_{k \in \mathcal{K}_L \cup \mathcal{K}_F} \sum_{j \in \mathcal{X}_k} \rho \left(\| \mathbf{p}_{c,j} - \pi_s(\mathbf{R}_k \mathbf{P}_{w,j} + \mathbf{t}_k) \|^2_{\mathbf{Q}} \right) \tag{2.30}$$

It can be seen that the problem becomes significantly more complex than the one described by Equation 2.18 for the motion-only BA. For convenience of expression, here the pose is formulated with $\{\mathbf{R} \in SO(3),\, \mathbf{t} \in \mathbb{R}^3\}$ rather than $\{\mathbf{T} \in \mathbb{R}^{4\times4}\}$, but it expresses the same quantity.

The specific form for the partial derivative of the entire cost function for the *i-th* pose has been shown in Equation 2.23, however the partial derivative of the function to the *j-th* point was not derived since the spatial position of the feature points was not to be optimized before. For the *j-th* space point, the derivative of $\mathbf{e}$ with respect to $\mathbf{P}$ can be decomposed with the chain rule and can easily be demonstrated that:

$$\frac{\partial \mathbf{e}}{\partial \mathbf{P}} = \frac{\partial \mathbf{e}}{\partial \mathbf{P}'} \frac{\partial \mathbf{P}'}{\partial \mathbf{P}} = - \begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} \end{bmatrix} \mathbf{R}. \tag{2.31}$$

Known these two derivatives, it is possible to solve the normal equation Eq. 1.20 of the *L-M* method, reported here for convenience:

$$(\mathbf{H} + \lambda \mathbf{D}^T \mathbf{D}) \Delta \mathbf{x} = \mathbf{g} \tag{2.32}$$

where $\Delta \mathbf{x}$ is the increment of $\mathbf{x} = [\mathbf{T}_1, \ldots, \mathbf{T}_m, \mathbf{P}_{w,1}, \ldots, \mathbf{P}_{w,n}]^T$, which encloses all variables together, and $\mathbf{H} = \mathbf{J}^T \mathbf{J} + \lambda \mathbf{I}$.

Sparsity of the Hessian matrix

Given the shape of the Hessian matrix $\mathbf{H}$, the dimension of the linear equation system to solve within the *L-M* algorithm can become extremely high considering the number of poses and especially points that are involved. This characteristic would preclude the real-time use of the V-SLAM system, however, one of the major

breakthroughs of this field was the very realization that $\mathbf{H}$ has a sparse structure, and it can explicitly be represented by a graph (Figure 2.19).
Physically this means that each camera view only observes a small part of 3D points. The $\mathbf{e}_{ij}$ error term within $\mathbf{J}(\mathbf{x})$ describes only the residual about the *j-th* point $\mathbf{P}_j$ in the *i-th* pose $\mathbf{T}_i$, that is, the corresponding Jacobian is zero except for two blocks:

$$\mathbf{J}_{ij}(\mathbf{x}) = \left(\mathbf{0}_{2\times6},\ \ldots,\ \mathbf{0}_{2\times6},\ \frac{\partial \mathbf{e}_{ij}}{\partial \mathbf{T}_i},\ \mathbf{0}_{2\times6},\ \ldots,\ \mathbf{0}_{2\times3},\ \ldots,\ \mathbf{0}_{2\times3},\ \frac{\partial \mathbf{e}_{ij}}{\partial \mathbf{P}_{w,j}},\ \mathbf{0}_{2\times3},\ \ldots,\ \mathbf{0}_{2\times3}\right), \tag{2.33}$$

where the derivatives have been made explicit in Equations 2.23 and 2.31. Each error term is therefore dependent only on the variables directly involved: in terms of graph optimization, this observation edge is only related to two vertices. When considering the summation of all variables, the Hessian matrix can be expressed as:

$$\mathbf{H} = \sum_{i,j} \mathbf{J}_{ij}^T \mathbf{J}_{ij} = \begin{bmatrix} \mathbf{H}_{11} & \mathbf{H}_{12} \\ \mathbf{H}_{21} & \mathbf{H}_{22} \end{bmatrix}. \tag{2.34}$$

$\mathbf{H}_{11}$ is only concerned with camera poses and $\mathbf{H}_{22}$ only with feature points, and they are both always block-diagonal matrices. $\mathbf{H}_{12}$ and $\mathbf{H}_{21}$ instead may have a sparse or dense structure, depending on the data being observed and are in general unpredictable.

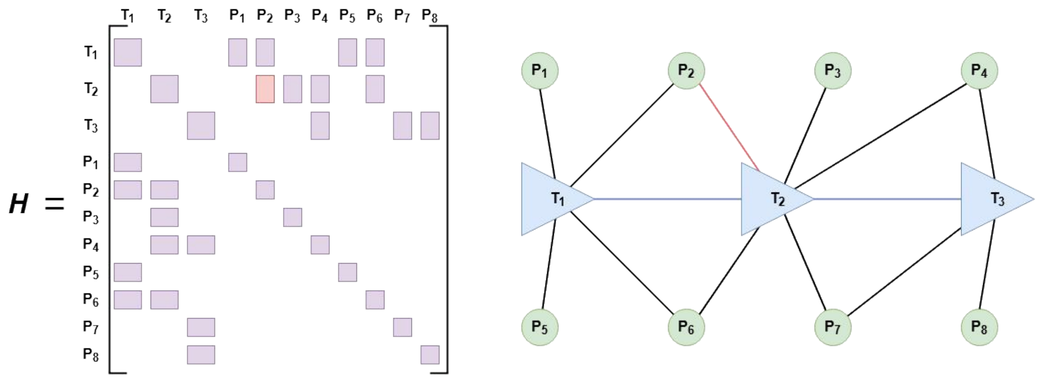

H =
T1 T2 T3 P1 P2 P3 P4 P5 P6 P7 P8
T1 T2 T3 P1 P2 P3 P4 P5 P6 P7 P8
P1 P2 P3 P4
T1 T2 T3
P5 P6 P7 P8

Figure 2.19: Example depicting the factor graph and the corresponding Hessian matrix, where $m = 3$ and $n = 8$. In real V-SLAM applications this matrix has several thousands of elements, most of which are due to the points variables.

This structure can be exploited to accelerate the calculations, in particular, *Matlab*'s `bundleAdjustment` function uses as solver for the systems of linear equations characterized by high sparsity the *preconditioned-conjugate-gradient* method, as it better fits the resolution of systems that are too large for direct resolution.

With the implementation of the LBA and subsequent updates of the `imageviewset` and `worldpointset` objects, the Local Mapping thread is completed.

2.7 Loop Closure and Pose Graph Optimization

This last section covers the concluding part of the system, where the loop closure is identified and the *a posteriori* optimization operation is performed to correct the drift. Before dissecting the Loop Closure thread, the bag-of-words approach used for image retrieval is briefly introduced.

2.7.1 Place Recognition Database Initialization

In the workflow presented so far, the module devoted to the Place Recognition Database Initialization was deliberately skipped. It was chosen to discuss it here as it is closely related to Loop Closure, although it is actually necessary to insert it before the main loop begins since the database will be updated for every key frame. Loop Closure is achieved when the system realizes that two frames, acquired at a sufficient distance from each other, are similar enough to establish that the camera is framing a scene that was previously seen. Direct feature matching is not a viable strategy since it becomes very time intensive if performed for every iteration with every previous key frame to detect the closure. Also, since some time has passed between the two potential similar frames, it is possible that conditions in the scene have changed such that the match is not recognized. To address this problem, ORB-SLAM2 utilizes a content-based image retrieval (CBIR) system to retrieve similar images from a database. In particular, this system uses a bag-of-words, which is a collection of image descriptors that represent the data set of images. The workflow is as follows:

1. An object to manage the image data is first created. In this object are uploaded the images to train the bag-of-words dictionary. In the case of the KITTI dataset have been used the 1101 frames from the left camera of Sequence 07, while in the case of the virtual simulation, have been used the same number of frames but taken from a random route within the scenario. This allowed to generate a visual vocabulary that is tailored to the search set;

2. The bag-of-words is generated offline using a custom ORB feature extractor function. The features are thus of the same type of those used during the SLAM algorithm, with the only difference that here only 1000 points for each

image are extracted (in the tests that will be presented, 2000 features are instead extracted during the operation of the system).
The dictionary is expressed with a tree structure (Fig. 2.20) created by clustering, where `numLevels` = 3 is the number of levels (or depth) of the tree, while `branchingFactor` = 10 defines the amount the vocabulary can grow at successive levels. The maximum number of words that are generated is therefore $\texttt{branchingFactor}^{\texttt{numLevels}} = 1000$. This is a relatively small number of words, but was deemed sufficient for the characteristics of the images;

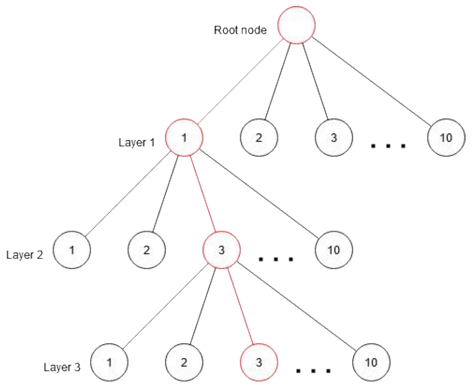


Figure 2.20: Tree structure of the dictionary.

3. A search index that maps the visual words to the images is established. This index is incrementally updated with new features during the Loop Closure thread (see step 4);

4. The database is searched for similar images to the query one (*i.e.* the current key frame being evaluated). If no suitable matches are found (see 2.7.2), then the features are added to the database, otherwise a connection between the query image and the most similar is created to establish the loop closure.

2.7.2 Loop Closure

The last thread of the main loop is Loop Closure. Loop Closure tries to detect and close the loop by comparing the current key frame processed by the Local Mapping process with images in the place recognition database that are visually similar to it. To optimize the detection process, a scalar value determines the threshold below which the current key frame is not considered as a possible candidate to close the loop, and its features are directly added to the database. This value is chosen based on how many key frames are extracted during the entire run: if taken too low the system could be better optimized to save computational time, but if taken too high there is a risk of skipping the frame that properly closes the loop. When a key frame is forwarded through the thread, it undergoes a sequence of operations to assess whether it is a possible candidate for loop closure. First, all key frames visually similar to the current one are retrieved with the bag-of-words approach. Their similarity score is extracted, which ranks the image retrieval results in a scale from 0 to 1 (where 1 is a perfect match) based on the metric parameter of the ORB features.

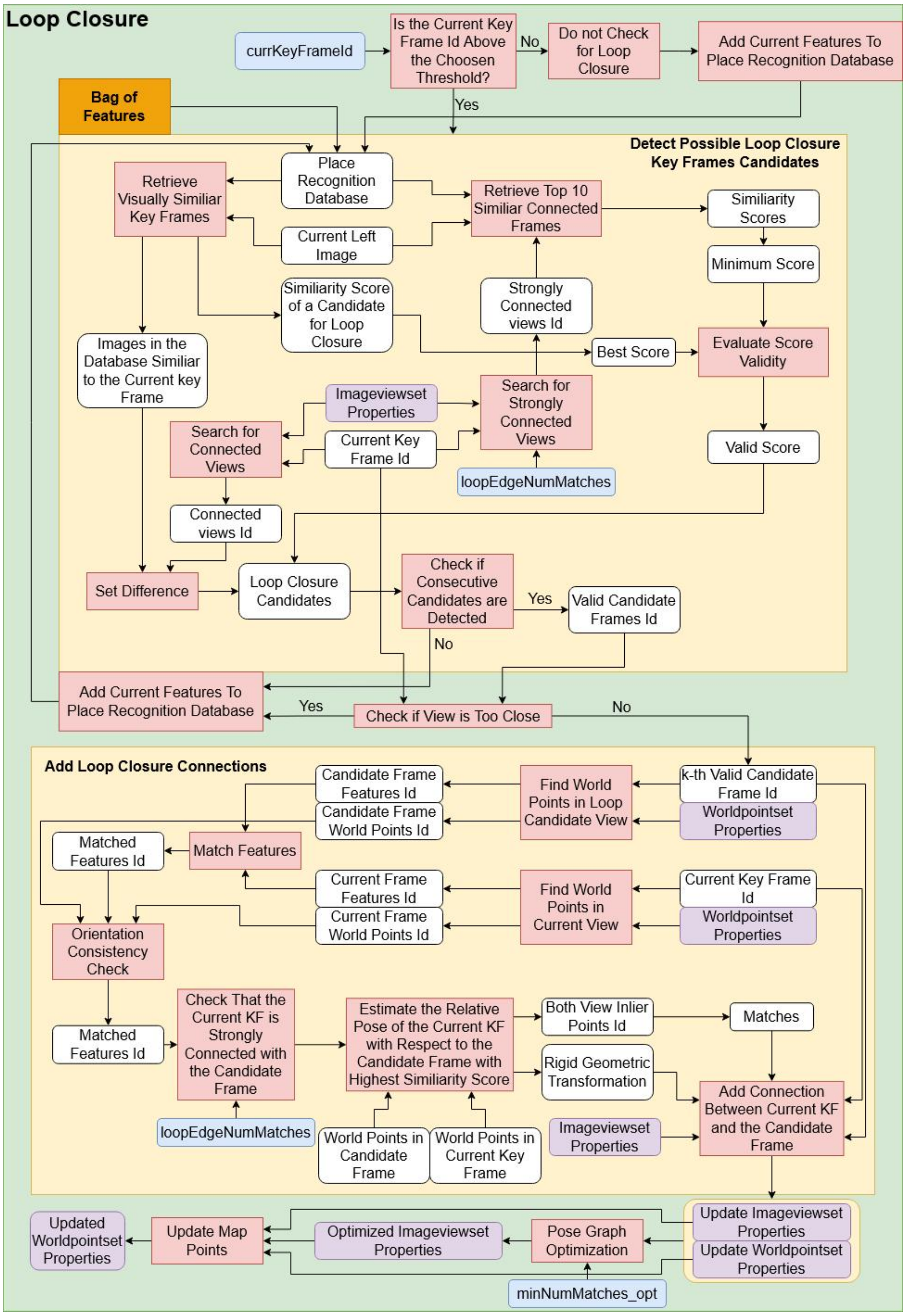

Loop Closure
currKeyFrameId
Is the Current Key Frame Id Above the Choosen Threshold?
No
Do not Check for Loop Closure
Add Current Features To Place Recognition Database
Bag of Features
Yes
Detect Possible Loop Closure Key Frames Candidates
Retrieve Visually Similiar Key Frames
Place Recognition Database
Retrieve Top 10 Similiar Connected Frames
Similiarity Scores
Current Left Image
Minimum Score
Similiarity Score of a Candidate for Loop Closure
Strongly Connected views Id
Images in the Database Similiar to the Current key Frame
Best Score
Evaluate Score Validity
Imageviewset Properties
Search for Strongly Connected Views
Search for Connected Views
Current Key Frame Id
Valid Score
loopEdgeNumMatches
Connected views Id
Check if Consecutive Candidates are Detected
Set Difference
Loop Closure Candidates
Yes
Valid Candidate Frames Id
No
Add Current Features To Place Recognition Database
Yes
Check if View is Too Close
No
Add Loop Closure Connections
Candidate Frame Features Id
Find World Points in Loop Candidate View
k-th Valid Candidate Frame Id
Candidate Frame World Points Id
Worldpointset Properties
Matched Features Id
Match Features
Current Frame Features Id
Find World Points in Current View
Current Key Frame Id
Current Frame World Points Id
Worldpointset Properties
Orientation Consistency Check
Check That the Current KF is Strongly Connected with the Candidate Frame
Estimate the Relative Pose of the Current KF with Respect to the Candidate Frame with Highest Similiarity Score
Both View Inlier Points Id
Matches
Matched Features Id
Rigid Geometric Transformation
Add Connection Between Current KF and the Candidate Frame
loopEdgeNumMatches
World Points in Candidate Frame
World Points in Current Key Frame
Imageviewset Properties
Update Imageviewset Properties
Updated Worldpointset Properties
Update Map Points
Optimized Imageviewset Properties
Pose Graph Optimization
Update Worldpointset Properties
minNumMatches_opt

Figure 2.21: Summary diagram of all the main operations that take place during Loop Closure.

To guarantee that the candidates are not connected to the current key frame, a filtering operation takes place that excludes from the list of visually similar key frames those who are also connected. Then, the similarity is computed also between the current key frame and its strongly connected key frames, that are connected frames with at least `loopEdgeNumMatches` matched feature points. The resultant minimum similarity score from the top 10 similar connected frames is used as a baseline to find loop closure candidates. In particular, the conditions on the similarity scores from these two set of images are imposed as follows: the score of the candidate frame must be higher than both the baseline from the connected key frames and a value equal to 75% of the best score found from the non connected key frames.
Before using these frames to add a loop closure connection, it must be verified that at least three loop candidates are consecutively detected to avoid false associations. If this condition is not met the thread is interrupted and the current features are added to the place recognition database, otherwise a second check follows. Indeed, a condition on the distance between two views must also be satisfied: if the current view and the candidate view are less then 100 key frames apart, then the thread is interrupted also in this case. This is aimed at avoiding the possibility of incorrect closure being recognized if after many key frames the camera sees the same features. At this point a set of valid candidates has been found and the process for creating the connections is initialized.

From the current key frame and the *k-th* valid candidate are extracted the world points and associated features obtained from the LBA in Local Mapping. These ORB features are matched in an analogous way to what was often done in previous threads and, after a check on the consistency of the orientation property, it is evaluated if there are enough matches to satisfy the strong connection condition imposed by `loopEdgeNumMatches`. Finally, it is performed an estimation of the relative pose between the current key frame with respect to the candidate frame with the highest similarity score, which will be used to add the loop connection. The `imageviewset` and `worldpointset` objects are updated as consequence to store this information.

In this implementation of the ORB-SLAM2 method, the main loop ends when the available video frames are terminated or when the loop closure is recognized. In both cases the final pose graph optimization is initiated to correct for the accumulated drift, but only when the loop is closed there is a significant improvement of the estimate.

2.7.3 Pose Graph Optimization

It is possible to exploit the new connections from the passage of the camera through the same location to perform an overall optimization for the entire trajectory. As seen in Section 2.6.2, the world points are by far the most demanding variables to optimize as they are much larger in number than the poses, and it can easily become prohibitive to perform GBA when seeking for real-time SLAM. A strategy that allows to solve the problem of global optimization is given by the use of a pose graph. Pose graph is a widely used optimization approach that is particularly valuable in problems with large scale given its speed and better convergence: only the connections between the camera poses are considered and the world points are only regarded as constraints of pose estimation. Indeed, it must be considered that the position of the world points have already been gradually refined during LBA, and a further refinement would yield not so significant improvements.

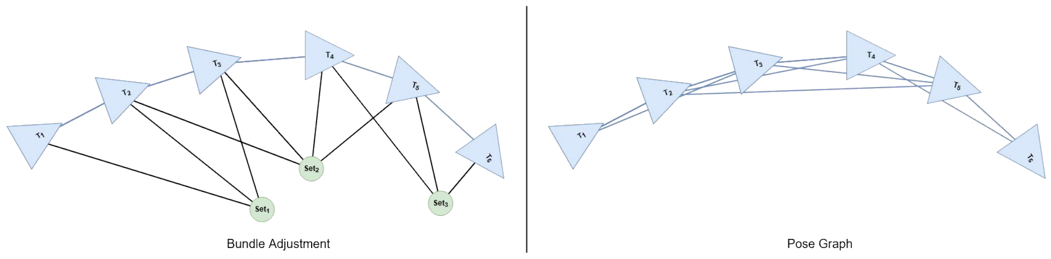


Figure 2.22: Pose Graph requires a much reduced calculation scale since the world points are no longer optimized. The set of co-visible points define the edges between poses.

Moreover, ORB-SLAM2 does not use the previously built *Covisibility Graph* for the global optimization as in many cases it can be very dense because of all the shared feature points that are observed. An *Essential Graph* that retains all the nodes (key frames) but that discards the edges with lower weight is instead employed. This graph is created internally within the `imageviewset` object by removing connections with fewer than `minNumMatches_opt` matches in the *Covisibility Graph*. Despite this reduction of the optimization problem, as demonstrated for ORB-SLAM in [11], pose graph optimization can obtain results accurate enough that an additional full BA would improve the solution only slightly. To be precise, the improved ORB-SLAM2 presented in [12], incorporates a full BA after pose graph optimization that runs in a different thread and allows to achieve the optimal solution. This expedient, however, was not included in the *Matlab* implementation of the method.
Given the pose graph of binary edges (each edge is between two poses and it estimates the relative motion between them), the least-squares error is then of the

following type:

$$\mathbf{e}_{i,j} = \log(\mathbf{T}_{i,j}^{-1}\mathbf{T}_i^{-1}\mathbf{T}_j)^{\vee}, \tag{2.35}$$

where $\mathbf{T}_i$ and $\mathbf{T}_j$ represent the camera pose for a node and $\mathbf{T}_{i,j} = \mathbf{T}_i^{-1}\mathbf{T}_j$ represents their relative motion in $SE(3)$. $^{\vee}$ is the operator that turns a skew-symmetric matrix in a vector. Once the Jacobians have been derived, the least-squares problem where the optimization variable is the pose from each key frame is defined by the objective function:

$$\min \sum_{i,j \in \varepsilon} \left(\mathbf{e}_{i,j}^T \mathbf{\Lambda}_{i,j} \mathbf{e}_{i,j} \right). \tag{2.36}$$

ε is the set of all edges and $\mathbf{\Lambda}_{ij}$ is the information matrix of the edge, which has been set to the identity. Pose graph optimization is thus based on rigid body transformations, and it uses the *Levenberg-Marquardt* algorithm with sparse *Cholesky* factorization from the *g2o* library.

After optimization, the new poses are used to update the locations of the 3D world points so that the definitive values for both the entire trajectory and the map are finally obtained. To verify the accuracy of the estimate just obtained, a comparison can be made with the trajectory obtained by GPS/IMU in the case of a dataset or with the exact trajectory in the case of virtual simulation.

Chapter 3

Simulation Set Up

This chapter discusses the steps followed to set up the virtual environment simulations that will make use of the ORB-SLAM2 method just described. The goal is to prepare a versatile framework that can be employed to perform simulations faithful to a real-world scenario and that returns comparable accuracy in pose determination. For this purpose, the KITTI dataset was taken as a reference to define characteristics such as camera intrinsics, vehicle dynamics and environment features, as well as to quantify the order of magnitude of the errors obtained with this particular ORB-SLAM2 implementation.

3.1 Why a Virtual Environment?

Suppose, for example, that we want to evaluate the quality of the V-SLAM method being developed. There are basically three approaches that can be followed to supply the images to be given as input: we can use personally acquired images, a dataset or a virtual environment. The first option allows for more freedom and control of the variables involved, thus making it possible to build a tailored set up. The problem, however, is that this path can be very cumbersome and expansive to put into practice. Personally acquiring the images involves setting up a vehicle with the chosen calibrated sensors to make it navigate a suitable environment. Moreover, to verify the correctness in location estimation, it is desirable to have the actual pose -or a precise approximation- of the vehicle during the entire duration of image acquisition, for example by using a GPS receiver or other sensors to perform odometry. Preparing all of this can be a non-trivial challenge for small research groups due to the costs and time required, and for this reason it is mostly used in laboratories for tests of limited range, for example with small UAVs or ground vehicles moving in confined spaces or even by moving the camera by hand. Besides, what if we want to evaluate the results with a different set of cameras? Or change the testing scenario from an indoor environment to a park full of vegetation, rather than a road traveled by other vehicles? It is entirely reasonable that one would want to make these changes, in fact, lighting conditions, scale, orientation and texture of the scene subjects, as well as the motion of the cameras and their intrinsics are some of the determining conditions affecting the proper functioning of the method. Extensive evaluation therefore requires numerous tests that could involve substantial changes to the set up.

One widely used alternative is thus that given by the datasets. Datasets are intended to provide users with everything they need to be able to effectively test the algorithm. Teams of researchers devoted themselves to capture video sequences and describe in detail the process they followed: what sensors are used and what their characteristics are, how they are mounted on the vehicle, what is the ground truth trajectory and the precision with which it was estimated, and so forth. This operation is usually repeated for different environments or conditions so that a wide range of situations of interest in applying one's method is covered. It is clear, then, how this second approach is much easier and faster for the users to follow than the first one, and can provide scenarios that otherwise would not be possible to experience. Another important advantage of datasets is the fact that they are the same for everyone. This means that they can be employed to make comparisons between different algorithms and see which one best behaves in those particular conditions.

Finally, the third alternative is presented. Simulations consist of reproducing in a virtual environment a scenario that can be exploited to generate video sequences to be used to carry out, in our case, SLAM. By means of dedicated software, the 3D environment is modeled and rendered, inside which is inserted a virtual camera that replicates the characteristics of the real one that is to be simulated. This camera is then moved following the desired trajectory while recording the footage.

The traits of such an approach are the following:

- Potentially any scenery can be recreated: a urban setting, the interior of a building, the surface of Mars can be all replicated with a high level of detail and fidelity. Today's hardware and software make it possible to achieve photo-realistic graphics, with high resolution textures and advanced illumination. It should be noted, however, that creating a scenario form scratch can be very complex and time-consuming, depending on the level of detail desired. In addition, the computational resources required can be very high. For these reasons, a trade-off should be made between realism and performance;

- Many sensors, including conventional cameras, can be reproduced and if necessary easily modified. This allows to evaluate different configurations very quickly by simply changing parameters such as intrinsic, resolution, frame rate, *etc.*;

- Vehicle dynamics can also be faithfully simulated if necessary, and any desired trajectory can be plotted. Unlike the previous cases, here the exact pose of the camera is known frame by frame. This simplifies and improves the accuracy of comparing estimated and actual data;

- As with datasets, the set up can be shared with the research community so that everyone can perform test under the same conditions;

From these characteristics it is clear that there are significant advantages in making use of a virtual environment and that it has the potential to be a viable alternative to the more conventional methods previously described.

Before moving on to the description of the simulation set up, the KITTI dataset is introduced.

3.2 KITTI Vision Benchmark Suite

The *KITTI Vision Benchmark Suite* [23] is a project developed by the Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago that offers a wide array of real-world computer vision benchmarks. A station wagon equipped with two pairs of high-resolution cameras, one capturing grayscale images and the other RGB images, was driven around the German city of Karlsruhe and its surroundings to capture numerous datasets in different settings. An accurate estimation of the ground truth is provided by a laser scanner and an inertial navigation system (GPS/IMU), also mounted on the vehicle. Sequences of various lengths set in the city, residential areas, highways, and campus were thus obtained, with both static and dynamic subjects. The entire suite provides data and images that can be used in several tasks of interest, such as stereo vision, optical flow, visual odometry, 3D object detection and 3D tracking.
Given the variety of scenarios it offers and the rigor with which the data were acquired, KITTI is very popular in these areas of research and is often chosen to test new algorithms and implementations or to compare the validity of other benchmarks. For these reasons, it was also chosen by me to run the ORB-SLAM2 method

and use the results as a reference to test the validity of the subsequent simulations in the virtual environment.

Visual Odometry / SLAM Evaluation Dataset

Sensor Setup

Figure 3.1 shows the configuration of the fully equipped vehicle employed to record the dataset. The sensors used are:

- ☐ 1 Inertial Navigation System (GPS/IMU): *OXTS RT 3003*;
- ☐ 1 Laser scanner: *Velodyne HDL-64E*;
- ☐ 2 Grayscale cameras, 1.4 Megapixels: *Point Grey Flea 2 (FL2-14S3M-C)*;
- ☐ 2 Color cameras, 1.4 Megapixels: *Point Grey Flea 2 (FL2-14S3C-C)*;
- ☐ 4 Varifocal lenses, 4-8 mm: *Edmund Optics NT59-917*.

In particular, the cameras are mounted with a stereo baseline of $0.54\,m$, approximately level with the ground plane at a height of $1.65\,m$ and the captured raw (distorted and unrectified) images have a size of 1392×512 pixels. The cameras operate at a frame rate of 10 fps as they have been synchronized with the laserscanner, which also spins at 10 fps. Users are further provided with the images already undistorted and rectified, that have been cropped to 1242×375 pixels and which can be used directly as input. Additional information about the sensor setup and how the dataset were obtained can be found in the presentation document of the KITTI Dataset [8].

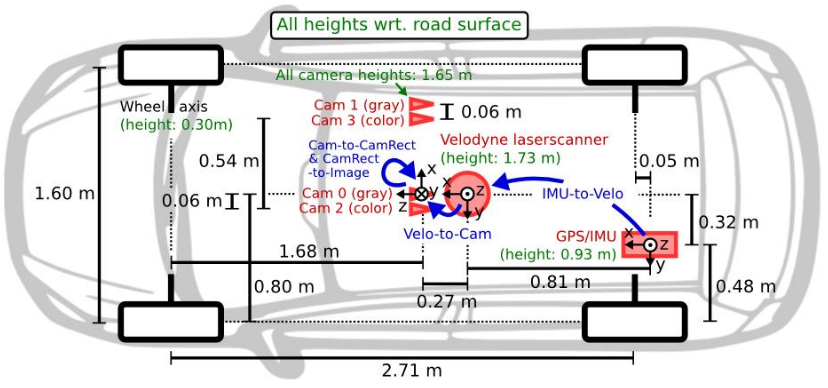


Figure 3.1: KITTI's full sensor setup (courtesy of the KITTI website [23]).

The dataset that was downloaded consists of 22 stereo sequences shot with the stereo color cameras and saved in a lossless *.png* format. Of these 22 sequences, only the first 11 are intended for method training and provided with the ground truth trajectories. The test here conducted makes use of the sequence 07 (Figure 3.2), which was obtained in a sunny day in a suburban neighborhood by following at modest speed a closed route of slightly less than 700 meters in length.

Recurrent features are: the presence of numerous cars parked along the road and some sporadic ones moving in directions parallel or normal to the direction the camera is facing, the almost constant presence of two-story buildings on both the right and left sides of the street, the presence of some vegetation elements.
In addition to the stereo image sequence, two files were also downloaded: one containing the pose of the GPS/IMU unit synchronized for each frame, and the other containing the projection matrices of the cameras. Their description and use will be discussed in Chapter 4.

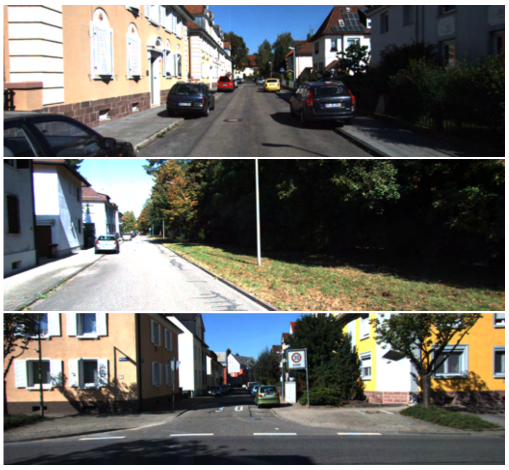


Figure 3.2: Some significant frames captured by the left camera during Sequence 07.

3.3 Simulation Software and Hardware

3.3.1 Software

The next paragraphs will introduce the software that were used for the realization of this thesis, specifying the role they played.

Matlab (Ver. R2022b)

Matlab [26] is a programming platform based on the matrix-based Matlab language, that allows to easily express computational mathematics expressions and elaborate them. It debuted in a commercial version in 1984, and has since been confirmed over the years as a reference tool for engineers and scientists interested in developing algorithms, analyzing data, and creating models and applications. Some of its strengths are the ease of use, at least compared to other programming languages, the possibility to expand its capabilities with packages and toolboxes specific to certain fields of interest (see 3.3.1), and the support of a community that is particularly active in forums and other platforms. These aspects have also made

it of great interest for academic use; in fact, the *University of Padua* provides its students with a Campus-Wide license that can be installed on one's personal computer.
The choice to use this software stems from the advantages just mentioned, as well as the fact that it is a suitable tool for dealing with Simultaneous Localization And Mapping. The set of algorithms needed to solve the SLAM problem are therefore implemented in *Matlab*, to which the other software will interface.

Simulink (Ver. 10.6)

Simulink [27] is a graphical programming environment also developed by *Mathworks* and employed for modeling, simulating and analyzing multidomain dynamical systems. It is widely used in both the academic and the industrial environment in that it allows complex models to be simulated with a scaling level of detail, at user's discretion. Depending on the scope of the project or the stage of development that is being addressed, a system can be modelled accordingly, from a preliminary study case to the detailed definition and construction of a particular subsystem. Since *Simulink* is strictly integrated within the *Matlab* suite, it was natural to use it to set-up a simulation where the SLAM algorithm can be tested. Thanks to this direct interface, it has been possible to build an integrated system describing the problem as a whole with greater ease.

Unreal Engine (Ver. 4.26)

Unreal Engine (UE) [38] is a popular open source 3D computer graphics engine developed by *Epic Games* that made its first appearance in 1998 with the video game *Unreal*. Since then it has found wide applications not only in the video game industry but also in film, architecture, automotive and all those contexts that require major applications of 3D creation tools. *UE* can be used to generate large and intricate static or dynamic environments with exceptional level of detail, as well as interactive scenarios with accurate simulation of physical interaction between the various actors within the world. These characteristics have made it in more recent times also of great interest in the field of simulations when paired with other software. *Mathworks* gives the possibility to interface *Matlab/Simulink* with *Unreal Engine* in a fairly simple way, thus creating a set of three software programs that are well interconnected and that manage to cover almost completely the needs required by the development of this thesis.

This program is used here to load, customize and setup an arbitrary scene within which the simulation is performed.

Matlab/Simulink/Unreal Engine Add-ons

As mentioned, the capabilities of *Matlab* and *Simulink* can be extended by downloading packages and add-ons that integrate new features specific to a certain area of interest. Below is a brief description of those that were used for this work:

- ***UAV Toolbox***: provides tools and reference applications for designing, simulating, testing, and deploying unmanned aerial vehicle (UAV) and drone applications.
- ***UAV Toolbox Interface for Unreal Engine Projects***: framework that allows to use custom scenes to co-simulate in both *Simulink* and *Unreal Engine*.
- ***Computer Vision Toolbox***: provides algorithms, functions, and apps for designing and testing computer vision, 3D vision, and video processing systems. Contains the set of functions that were used to implement the V-SLAM method.
- ***Image Processing Toolbox***: provides a comprehensive set of reference-standard algorithms and workflow apps for image processing, analysis, visualization, and algorithm development.
- ***Automated Driving Toolbox***: provides algorithms and tools for designing, simulating, and testing ADAS and autonomous driving systems. Contains the set of functions that were used to obtain a 2D map of the scene and track on it the path that the drone has to follow.
- ***Unreal Engine plug-ins***: it is necessary to install and enable them to complete the interface operation between *Matlab/Simulink* and *Unreal Engine*.
  - ***MathWorks Interface***: enables connectivity between the two software;
  - ***MathWorks Automotive Content***: adds automotive vehicle meshes and materials;
  - ***MathWorks UAV Content***: adds UAV vehicle meshes and materials.

Microsoft Excel

Microsoft Excel [34] is a spreadsheet part of Microsoft Office, to date probably the most common productivity software suite in use around the world. Its popularity makes further introduction unnecessary. It was used here to properly format the pose data exported from *UE* so that it could be processed by *Matlab/Simulink*.

FFmpeg

FFmpeg [20] is a free open-source command-line program to record, convert and stream audio and video. It allowed to manage the videos obtained as output from *Simulink*, specifically to break them down into frames and eventually reduce their frame rate to $10fps$ (same value as the *KITTI* dataset). In addition, it allowed the reverse operation to be performed with the images in the KITTI dataset: from the frames, the video was reconstructed. This was useful to verify the correctness in the orientation of the vehicle and to identify at what instant some significant events occur during the recording.

3.3.2 Hardware

The simulations were carried out on my personal computer, the relevant characteristics of which are given in the table below.

Central Processing Unit (CPU)	AMD Ryzen 5 5600X
Random Access Memory (RAM)	Crucial Ballistix DRAM DDR4 3600MHz, 32GB
Graphic Processing Unit (GPU)	GIGABYTE RTX 3060 VISION OC R2, 12GB
Solid State Drive (SSD)	Samsung 980 PRO, 1TB

Table 3.1: PC's Hardware specifications.

It proved to be a suitable configuration, given the high computational requirements for *UE* to work with a large high-resolution scenario and detailed lighting. As for V-SLAM algorithm, it definitely exceeds what are the capabilities of a typical hardware embedded on a commercial robot, drone or generic vehicle. In this work, there has been no particular concern about the management of computational resources, however, there has always been attention in seeking a compromise between accuracy and speed by not overloading the algorithm with excessive parameters values.

3.4 Virtual Simulation

The framework that was built to perform the simulations is summarized in Figure 3.3. As will be described later, a different alternative to the one shown here was explored and implemented, however this one was found to be the most effective and the one that allowed for greater customization and control of variables.

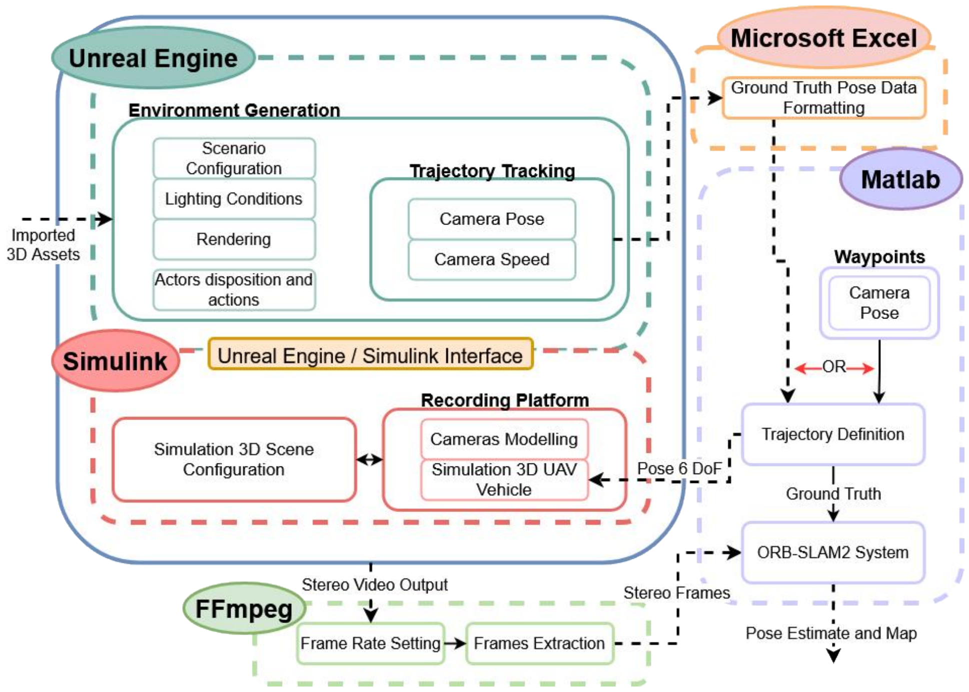


Figure 3.3: System Diagram for the Simulation Framework.

First, the desired virtual environment is generated in *Unreal Engine*. Next, *Simulink* is interfaced with *UE* and the recording platform is modelled. A step-by-step guide of the process for interfacing these two software is available in the *Mathworks* documentation [29], in particular, the procedure making use of the *UAV Toolbox Interface for Unreal Engine Projects* was followed. As a result of this operation, it becomes possible to initialize a co-simulation in custom scenes by adding sensors and vehicles in the *Simulink* model and making them appear in the virtual environment managed by *UE*. To plot the trajectory for the vehicle to follow, two alternative methods are presented: the first uses waypoints set in *Matlab* on a top-down map of the scenario, the second extracts the pose, frame-by-frame, of a generic three-dimensional path chosen by the user within *UE*. The output of the simulation are the two videos recorded by the stereo cameras, and are then processed by *FFmpeg*. ORB-SLAM2 is implemented entirely within *Matlab* and uses the freshly generated images and ground truth as input.

3.4.1 Unreal Engine Scenario

A great deal of time and expertise is required to create a complex virtual scenario from scratch considering that each object has to be drawn, modeled, textured and eventually animated. The task can be sped up dramatically by resorting to downloadable assets, made by professionals or amateurs, that are suitable for the scene one wants to represent. For this project, I prepared a scenario that makes use of the *AutoVrtlEnv* [28] *UE* project made available by *Mathworks*. The *AutoVrtlEnv* file is part of the *Vehicle Dynamics Blockset Interface for Unreal Engine 4 Projects* and includes editable versions of prebuilt 3D scenes. In particular, the *US City Block 3D Environment* was taken as reference to then be customized. It reproduces an urban environment spanning about $450\,m \times 300\,m$ meters that consists of a dense grid of streets surrounded by multi-story buildings. It was deemed a good starting point but lacking several features found in the KITTI sequence, which is why the following changes have been brought:

- Numerous static objects of various kinds enrich the scenery. Parked vehicles, vegetation, street signs and more provide new elements from which ORB Features can be extracted;
- Dynamic vehicles have been introduced that move according to a predetermined path and are encountered at certain intersections and road sections;
- The location of some buildings has been changed;
- Lighting conditions were also imported from the *AutoVrtlEnv* project, but then modified to better simulate those sought.

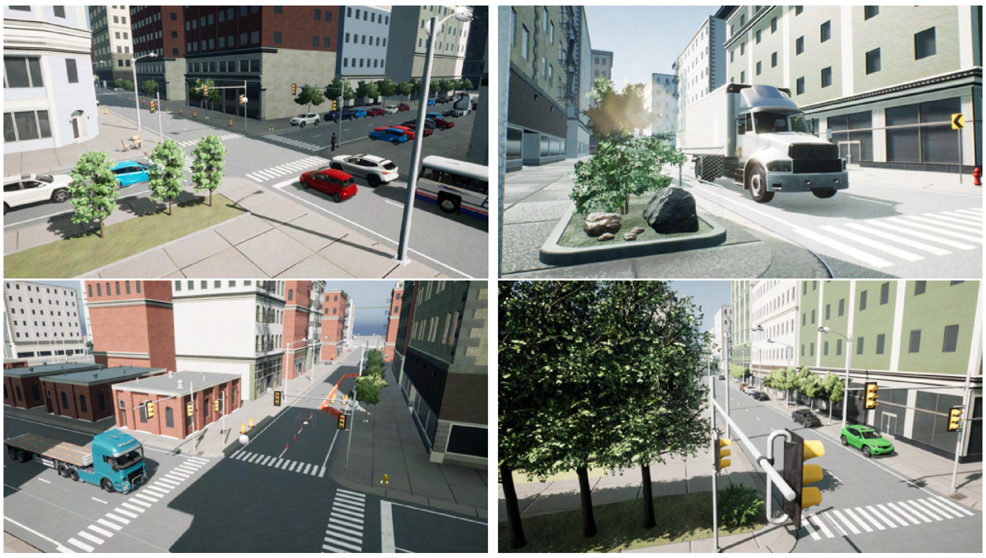


Figure 3.4: Some screenshots taken from the virtual scenario.

Being able to faithfully reproduce the conditions under which the images in the dataset were recorded is complicated. An optimal comparison would have required the creation of a scenario with dedicated assets that reproduce the road and surroundings and then provide as input the IMU/GPS pose to be followed frame by frame. It is clear how this approach would become extremely complex and time-consuming, also going against what should be one of the main advantages in adopting a virtual environment: speed and flexibility of use. For this reason, the modifications made are considered good for representing the characteristics of a real urban area reminiscent of the KITTI sequence, but are not intended to perfectly replicate it.

Dynamic Vehicles

To introduce moving vehicles in these scenarios it was necessary to start from two new actors, one that represents the vehicle on which the desired static mesh can be applied, and one that represents the path to be followed. A Timeline associated with the *Move Car01* Event (see Fig. 3.5) allows to define a profile for the motion. Here was used a float curve that increases linearly from 0 to 1 over a certain amount of time to set the speed, so that the length of the path is entirely covered within that time interval. In order to do this, the variable *Alpha* is introduced and set to update with the value of the float curve. *Alpha* then linearly interpolates the location of the car along the length of the spline through a float Lerp.

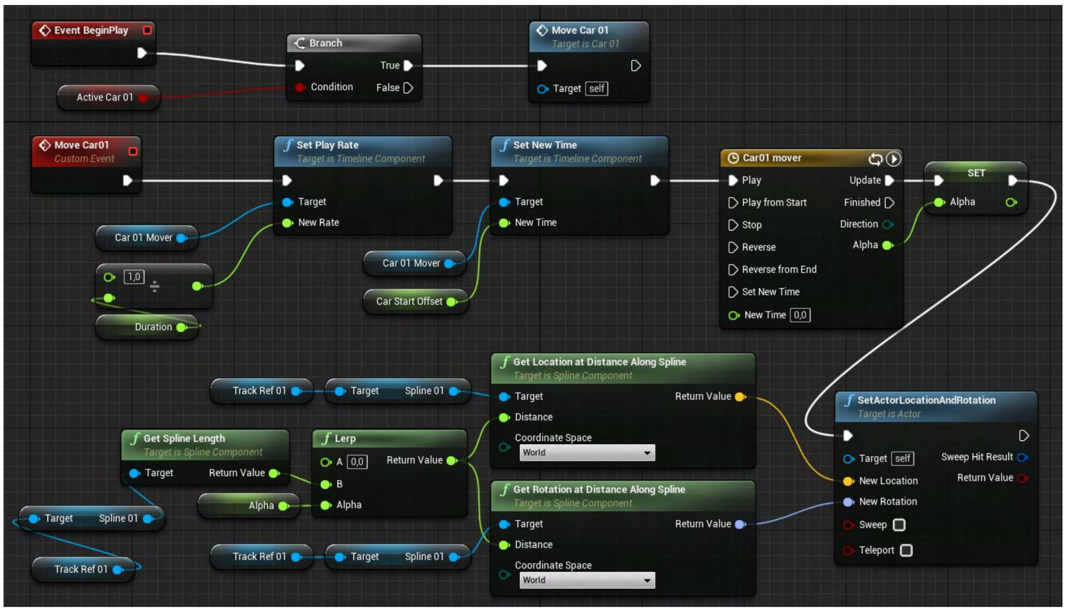


Figure 3.5: Unreal Engine Event Graph built for the dynamic vehicles.

The result is a car moving at the chosen constant speed along the spline that is drawn on the scenery. Once the end of the path is reached, the process starts over again. For sequences in which the cameras are stationary at an intersection and several vehicles pass in quick succession, it is sufficient to add actors to the same spline and set an offset that distances them from each other.

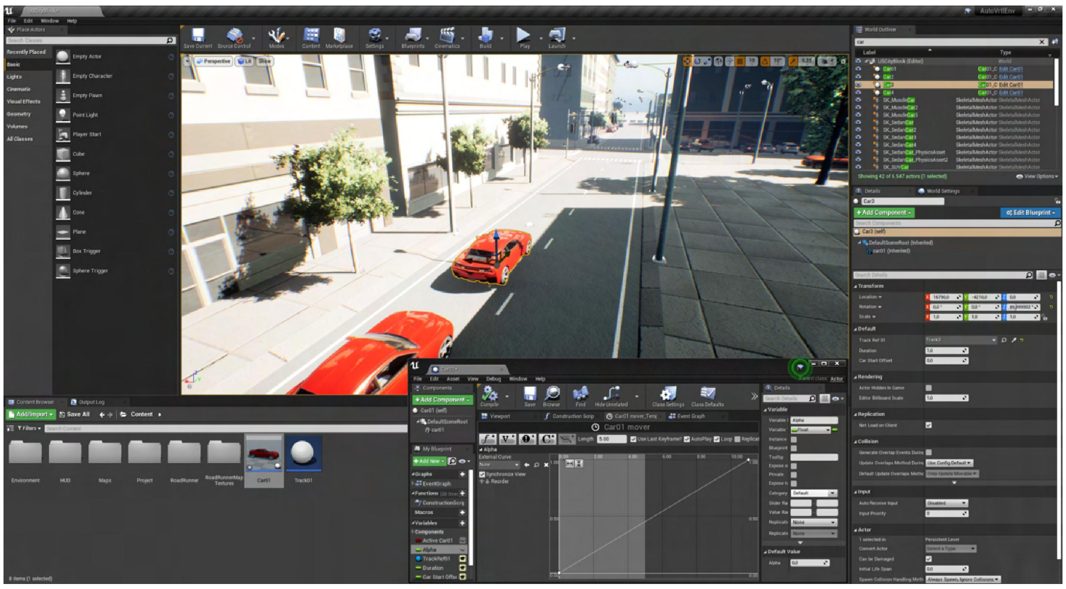


Figure 3.6: Unreal Engine interface during the process of creating dynamic vehicles. The bottom window shows the timeline editor to adjust the speed profile.

In Figure 3.6, the two red cars will proceed in a straight line one after the other until the end of the road. The recording platform will arrive from the road on the right and turn to flank oncoming traffic. In this way, the cars will be seen first with motion orthogonal and then parallel to the direction toward which the cameras are pointing.

Lighting Conditions

Lighting conditions can significantly affect the performance of a SLAM method. Shaded subjects may not be clearly visible, while subjects directly lit by the sun may be overexposed and completely lose detail or cause important reflections. Moreover, the interaction of light with the lens is cause of unwanted artifacts on the image. These features have been reproduced by tweaking three different actors in *Unreal Engine* so that the conditions were more similar to those in the dataset sequence:

1. A *Directional Light* actor simulates sunlight and allows to control the characteristic of this light source. In particular, by editing the *Light Intensity* and

Shadow Amount parameters, a bright scenery with dark shadows was obtained. The temperature was also adjusted to make it more like the one desired;

2. The actor *Sky Light* was used to regulate the sky's appearance and its lighting/reflections in the world;

3. The actor *Post Process Volume* allowed to introduce some lighting artifacts on the image.

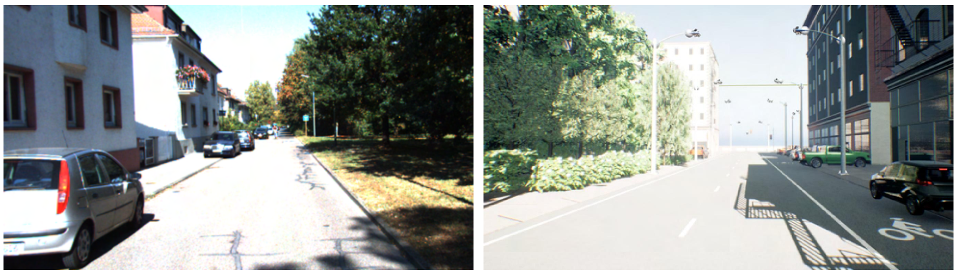
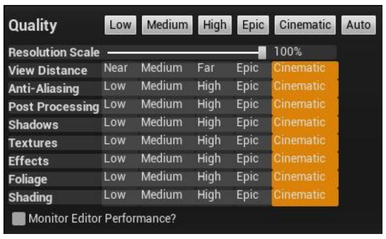


Figure 3.7: Environment and lighting similarities between KITTI and the virtual scenario.

Figure 3.7 shows two similar stretches of road, on the left taken from the KITTI sequence and on the right from the virtual scenario. In both cases one can recognize chromatic aberration, bloom effect from reflective surfaces, high amount of shadows, and overexposure resulting in loss of detail. Lens flare have also been replicated and is slightly visible when the camera moves toward the sun or very reflective surfaces, such as cars with metallic bodywork. Motion blur was not introduced as it is negligible in the sequence of real images.
Light have been rendered at production quality, which is the highest achievable but requires more time to compute.

Graphics Quality

It is important to provide sufficient graphics quality to give the scenario a certain level of realism. In general it is possible to have extremely high resolutions even for smaller objects, however it must be considered that here the cameras are rarely found in close range with the assets, and when this happens such assets are not considered to track the ORB features. Furthermore, the resolution of the images captured here to perform SLAM is not high enough to appreciate



Quality	Low	Medium	High	Epic	Cinematic	Auto
Resolution Scale					100%	
View Distance	Near	Medium	Far	Epic	[✓ Cinematic]	
Anti-Aliasing	Low	Medium	High	Epic	[✓ Cinematic]	
Post Processing	Low	Medium	High	Epic	[✓ Cinematic]	
Shadows	Low	Medium	High	Epic	[✓ Cinematic]	
Textures	Low	Medium	High	Epic	[✓ Cinematic]	
Effects	Low	Medium	High	Epic	[✓ Cinematic]	
Foliage	Low	Medium	High	Epic	[✓ Cinematic]	
Shading	Low	Medium	High	Epic	[✓ Cinematic]	

☐ Monitor Editor Performance?

Figure 3.8: Engine Scalability Settings.

the use of more detailed textures. The textures packages that I used go up to 4K resolution for the larger objects, and, given the previous considerations, were found to be suitable for the purpose of this project.
The graphics quality and Material Quality Level has instead been set to the maximum within Unreal Engine (Figure 3.8).

3.5 Coordinate Systems

There are four reference systems that must be taken into account when using this framework:

1. **Simulink Earth-Fixed Coordinate System**:

   The Earth-fixed coordinate system (X_E, Y_E, Z_E) defined within the *UAV Toolbox* has its axes fixed in an inertial reference frame and has no linear or angular acceleration and no angular velocity. It follows the Right Hand rule and its axes are oriented as follows:

   - X_E: points in the initial forward direction of the UAV and parallel to the ground plane;
   - Y_E: orthogonal to the X_E-axis and parallel to the ground plane, to complete the tern;
   - Z_E: points upward, normal to the ground plane.

2. **Unreal Engine World Coordinate System**:

   The World Coordinate System (X, Y, Z) defined in Unreal Engine is also an inertial reference frame, but follows the Left Hand rule. Its origin is located in the center of the scenario and the axes are defined similarly to those in Simulink, except for the difference in convention.

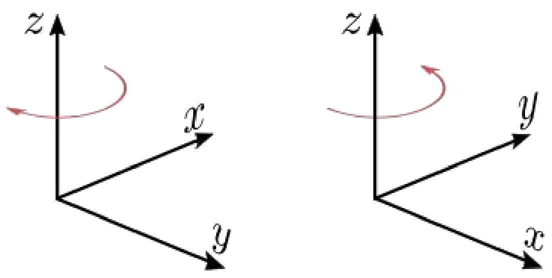
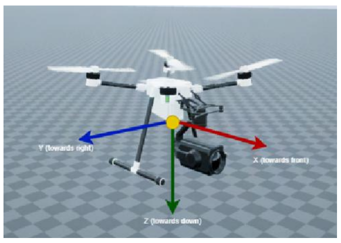


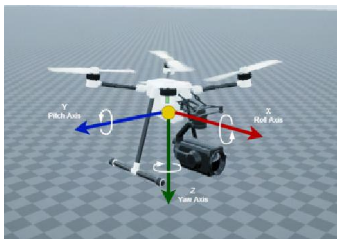
Figure 3.9: Differences between the Unreal Engine (left) and Simulink (right) Inertial Coordiante Systems.

3. **Body Reference System**:

   The Body Reference system, also defined in the *UAV Toolbox*, is non-inertial and fixed in both origin and orientation to the rigid body of the UAV.



(a) Translational DoFs



(b) Rotational DoFs

Figure 3.10: Body Coordinates (Courtesy of *Mathworks* documentation).

   - **x-axis**: points through the nose of the vehicle;
   - **y-axis**: points towards the right (with respect to the camera's direction of view), perpendicular to the x-axis;
   - **z-axis**: points down, perpendicular to the x-y plane and completing the Right Hand rule.

   The rotational degrees of freedom available to the rigid body are defined as follows:

   - ψ: Yaw angle about the z-axis;
   - θ: Pitch angle about the y-axis;
   - ϕ: Roll angle about the x-axis;

4. **Camera(s) Reference System**:

   In all simulations, the stereo cameras are attached to the vehicle body, and share its reference system but they are translated along the y-axis by $+\,baseline/2$ and $-\,baseline/2$ respectively.

The opposite convention used by the reference systems must be taken into account to avoid discrepancies between the desired and actual motion of the UAV within the scenario. In the transition from one software to another, it is therefore necessary to change the sign of the y-coordinate and ψ angle.

3.6 Simulink Model for UAV Stereo Visual SLAM

The model built in Simulink to describe a vehicle equipped with a stereo camera to realize V-SLAM is depicted in Figure 3.11.

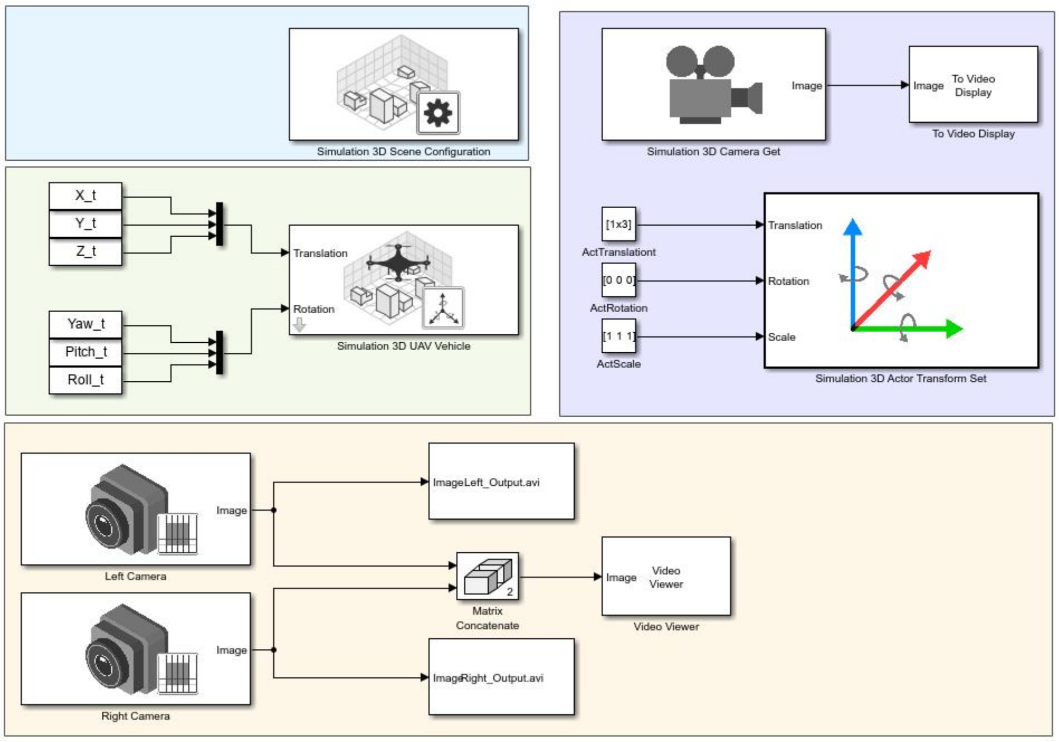


Figure 3.11: UAV Stereo Visual SLAM Simulink Model.

It has been organized into four macro blocks separated by different colors:

- **Cyan Block**: contains the *Simulation 3D Scene Configuration* block, which configures the 3D simulation environment that was rendered in *Unreal Engine*. It allows to select the directory path of the *UE* project in which we want the simulation to take place and to set two scene parameters: the Scene View, here set to the vehicle, and the Sample Time that will also be adopted by the other blocks. The Sample Time sets the fps of the cameras. It is also possible to quickly override the scene weather with a series of sliders that change the sun altitude and azimuth, as well as cloud, fog and rain properties. *Unreal Engine* must be initiated from here to enable it to interface with *Simulink*.

- **Green Block**: segment used to represent a quadcopter UAV within the scene and define its pose for each instance of the simulation. It was chosen to use a UAV instead of an automobile because it enables a six-degree-of-freedom motion for the cameras, thus covering the most general case for the definition of the pose. The *Simulation 3D UAV Vehicle* block has in fact as input the (X_E, Y_E, Z_E) position and (ψ, θ, ϕ) attitude values, and is paired with a *Sim3dQuadRotor* actor in Unreal Engine. The vectors can be obtained with one of the two methods that will be described in section 3.8. This segment could be further expanded by adding a drone dynamic model and a drone flight control model to appropriately represent the flight of the UAV. Since the goal here is to evaluate the SLAM method, this step was not deemed necessary for our purposes.

- **Yellow Block**: here are enclosed the models for the cameras. The left and right cameras are associated with two *Sim3dCamera* actors created in Unreal Engine and attached to the UAV with a given offset. A *Video Viewer* opens a window showing real-time images captured by both cameras, useful for viewing the actual images that will be processed by the SLAM method. At the end of the simulation the two videos in *.avi* format are generated by means of two *To Multimedia File* blocks, and will then be processed using *FFmpeg*.
- **Violet Block**: introduces a camera associated with a *Sim3dCamera* actor that follows the vehicle during its motion, and whose output is displayed in a dedicated video display. The images captured have a resolution of 1920×1080 pixels and a horizontal field of view of 60 degrees. The *Simulation 3D Actor Transform Set* block defines its pose.

An alternative to the approach shown here was also implemented but then abandoned. The yellow block can be configured to accommodate *Matlab*'s SLAM algorithm within *Simulink* to run in real time as images are generated in the virtual environment (Fig. 3.12).

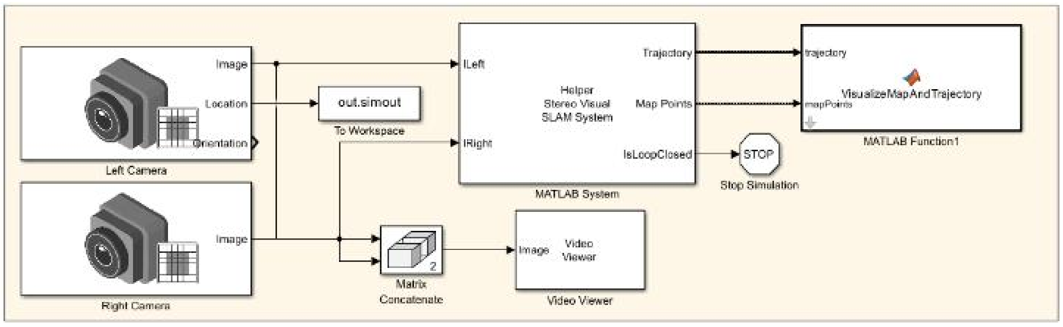


Figure 3.12: Alternative framework to implement the SLAM algorithm within Simulink.

The images captured by the cameras are in this case directly provided as input to the ORB-SLAM2 system, which estimates the trajectory and creates the map to

be visualized in a 3D graph while the simulation is still running. The simulation stops when it reaches the end of the predetermined path, the stop time, or a loop is closed.
This integrated implementation of the method is more streamlined and immediate to set up, however, it has some limitations that have been found to be not negligible. First, it is more complicated to access and modify the SLAM algorithm. While using the same algorithm described in Chapter 2, the code has been written to call directly on *Simulink*'s model and therefore cannot be separated from it. Second, the user has no direct access to the recorded videos before they are input to the *Matlab System* block. The possibility of processing video with *FFmpeg* has been found to be very valuable. For example, the frame rate can be decreased and the resolution changed (downscales are feasible but upscales can be problematic since new pixels are obtained by interpolation), so that it is not necessary to repeat the simulation to experiment with new video parameters. Finally, I wanted it to be possible to use the exact same approach in both the dataset and simulation cases. Following the scheme in Figure 3.3 resulted in a universal method in which it is convenient to change the parameters as needed and accommodates any type of input: in the case of images from a dataset it is sufficient to use only the final step, in which the stereo frames and ground truth data are given to the ORB-SLAM2 system that has been set accordingly.

3.7 Recording Platform and Virtual Cameras

The recording platform is co-simulated through *Simulink* and *Unreal Engine*. Thanks to the plugins previously installed in *UE*, it is possible to introduce into the scenario new actors managed by the *Simulink* model. In this case, three actors have been added: one *Sim3dQuadRotor* to simulate the vehicle and two *Sim3dCamera* to simulate the sensors. Figure 3.13 shows a depiction of the recording platform used in the virtual scenario. The word "depiction" was used because it is immaterial where the actors are placed within the scene, their location is always overridden by the parameters in the *Simulink* model. The *Simulation 3D UAV Vehicle* is in fact responsible for determining the initial pose of the vehicle, which is in turn taken from the first position vector $[X, Y, Z]$ and rotation vector $[\psi, \theta, \phi]$ defined during the trajectory tracking process. Another peculiarity is the fact that the mesh of a sedan car was used instead of a UAV. Again, it is irrelevant for the purpose of simulation which mesh is used: it was deemed more appropriate to show here a set up similar to that seen in the case of *KITTI* since it is the one we want to compare.

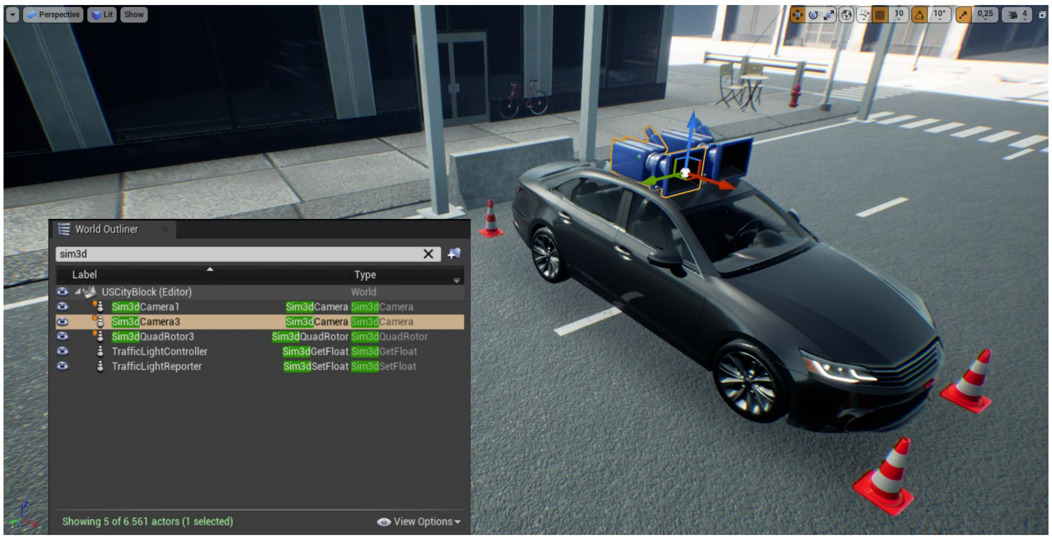
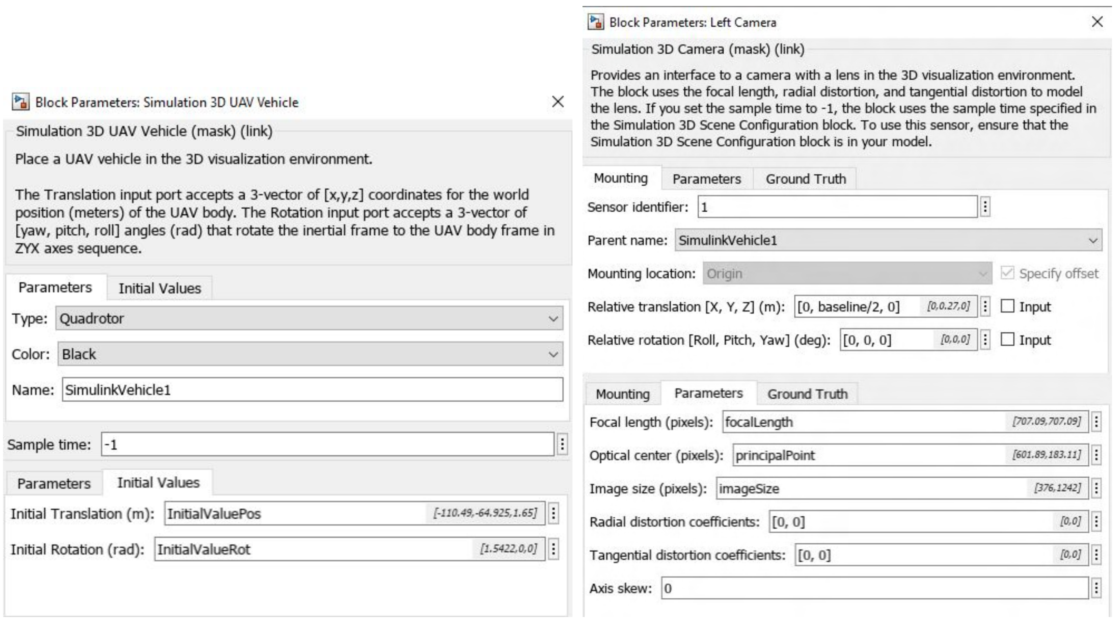

Perspective
Lit
Show
World Outliner
sim3d
Label
Type
USCityBlock (Editor) World
Sim3dCamera1 Sim3dCamera Sim3dCamera
Sim3dCamera3 Sim3dCamera Sim3dCamera
Sim3dQuadRotor3 Sim3dQuadRotor Sim3dQuadRotor
TrafficLightController Sim3dGetFloat Sim3dGetFloat
TrafficLightReporter Sim3dSetFloat Sim3dSetFloat
Showing 5 of 6.561 actors (1 selected)
View Options

Figure 3.13: Recording Platform used in the virtual scenario.

The characteristics of the sensors, where they are mounted and their intrinsics parameters, are managed by the block *Simulation 3D Camera*. This block models a conventional camera through the pinhole camera model plus a lens in front of it. They are thus both attached to the UAV actor and arranged with only an offset along the y-axis equal to the baseline of $0.54\,m$. The intrinsics parameters and image size are exactly the same from the *KITTI* dataset, and the distortion coefficients have been set to zero so even in this case there is no need to undistort the images and rectify them.



Block Parameters: Simulation 3D UAV Vehicle

Simulation 3D UAV Vehicle (mask) (link)

Place a UAV vehicle in the 3D visualization environment.

The Translation input port accepts a 3-vector of [x,y,z] coordinates for the world position (meters) of the UAV body. The Rotation input port accepts a 3-vector of [yaw, pitch, roll] angles (rad) that rotate the inertial frame to the UAV body frame in ZYX axes sequence.

Parameters | Initial Values

Type: Quadrotor
Color: Black
Name: SimulinkVehicle1

Sample time: -1

Parameters | Initial Values

Initial Translation (m): InitialValuePos [-110.49,-64.925,1.65]
Initial Rotation (rad): InitialValueRot [1.5422,0,0]

Block Parameters: Left Camera

Simulation 3D Camera (mask) (link)

Provides an interface to a camera with a lens in the 3D visualization environment. The block uses the focal length, radial distortion, and tangential distortion to model the lens. If you set the sample time to -1, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, ensure that the Simulation 3D Scene Configuration block is in your model.

Mounting | Parameters | Ground Truth

Sensor identifier: 1
Parent name: SimulinkVehicle1
Mounting location: Origin ☑ Specify offset
Relative translation [X, Y, Z] (m): [0, baseline/2, 0] [0,0.27,0] ☐ Input
Relative rotation [Roll, Pitch, Yaw] (deg): [0, 0, 0] [0,0,0] ☐ Input

Mounting | Parameters | Ground Truth

Focal length (pixels): focalLength [707.09,707.09]
Optical center (pixels): principalPoint [601.89,183.11]
Image size (pixels): imageSize [376,1242]
Radial distortion coefficients: [0, 0] [0,0]
Tangential distortion coefficients: [0, 0] [0,0]
Axis skew: 0

Figure 3.14: Parameters for *Simulink*'s Simulation 3D UAV Vehicle block (left) and Simulation 3D Camera block (right).

Figure 3.14 shows the parameters imposed for the *Simulation 3D UAV Vehicle* block and the *Simulation 3D Camera* block for the left camera. The only difference from the right one is in the relative translation with respect to the vehicle.

3.8 Trajectory and Orientation Definition

To make the framework suitable for use with a generic vehicle it is necessary to be able to control its 6 DoF. There is no interest here in correctly modeling the flight dynamics of a quadcopter, and it is sufficient to define the position and attitude values we want the vehicle to follow during the course of the simulation. Since the reference systems of the cameras are fixed with respect to that of the vehicle, it is immediate to switch from one to the other.

3.8.1 Waypoints Selection on a Top-Down Map of the Scene

Matlab provides an algorithm [30] that allows you to select a sequence of waypoints from a Top-Down map of the scene and visualize the path of the vehicle following these waypoints in the 3D simulation environment. Since this mapping method was designed for a ground-moving vehicle, some modifications had to be made to adapt it to a flying object.

A high resolution screenshot of an orthographic top view (Figure 3.15) of the scene is converted to a map thanks to the `imref2d` object, that associates the pixels of the image to the world coordinates. The distance along the X and Y coordinates from the center of the scene to the extremes of the map must be carefully measured and expressed in meters. In this way the figure will be centered and the trajectory that is tracked will physically coincide with the coordinates on the scenario in *Unreal Engine*.
Through a series of clicks, the path is imposed and a polyline is drawn on the map. Waypoints and Yaw angles are then exported to the workspace as an M-by-3 matrix of the poses $(\mathbf{X}, \mathbf{Y}, \boldsymbol{\psi})$, where $\boldsymbol{\psi}$ must be defined within the range $[-\pi, +\pi]$. A cubic spline function is then used to transform the sequence of poses in a continuous path, as shown in figure 3.15.

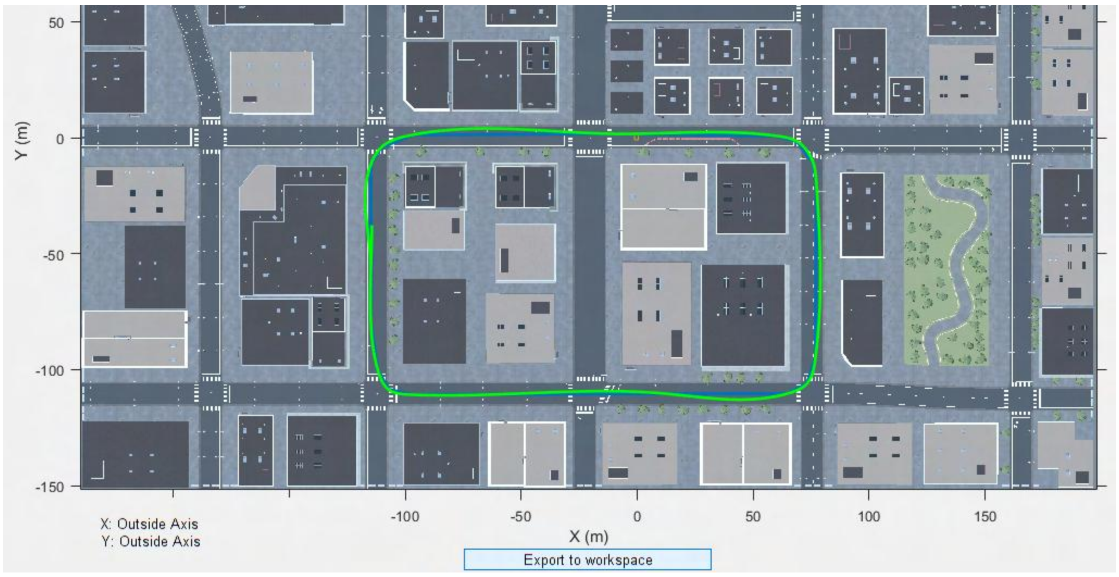


Figure 3.15: Top-Down map of the *US City Block* scene. In blue is plotted the segmented path defined by the waypoints, and in green is the path smoothed using the spline function.

Finally, in order to introduce the poses in the *Simulation 3D UAV Vehicle* block, it is necessary to decompose the matrix into column vectors and associate them with a time vector proportional to the cumulative path length.

This strategy, while very convenient to use once set up, has obvious limitations:

- Since it uses a spline function, it is not possible to precisely define the path point by point; potentially causing a collision with the assets in the environment;

- The use of a screenshot as a map prevents its application in enclosed locations (technically it is still possible, but you cannot see where the route is being plotted without having to hide assets that obstruct the view), and significantly limits the accuracy with which coordinates are chosen. Passages in tight areas are therefore difficult to obtain;

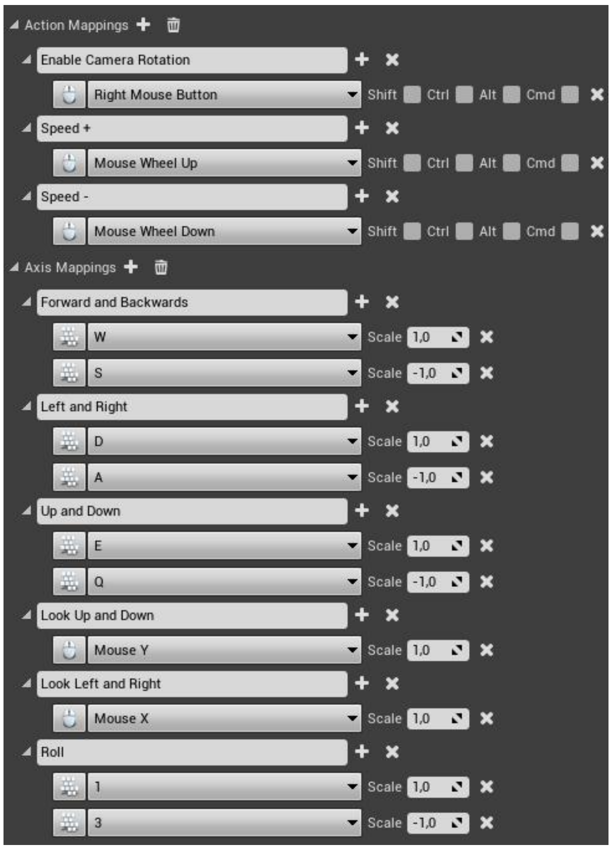
- Only 3 of the 6 DoF are imposed, leaving the *Z* coordinate and the *Pitch* and *Roll* angles undefined. This means that they must be added later as a constant or a function, limiting the possible poses and lengthening the path preparation process;

In order to address these issues, it was decided to develop the novel approach described in the next section.

3.8.2 Unreal Engine frame-by-frame pose

A method that easily allows an arbitrary path to be plotted in *Unreal Engine* and whose position and orientation values can be imported into *Matlab* would be ideal for quickly tailoring the simulation. No description of such a method was found in the literature, and the approach implemented to solve this problem is described here.

A camera that can be moved at the user's will is placed within the scene, and controlled with a series of inputs similarly to what is done in many videogames. To do so, the camera actor must be programmed to perform the desired set of tasks, so that there is total freedom of movement.
First of all, Action Mappings are used to bind keys to input behaviour, defining the commands to enable camera rotation and to increase or decrease the speed of motion. Axis Mappings set instead the inputs to move along the axes and to rotate the camera (Fig. 3.16). *Unreal Engine* allows you to choose from a variety of peripheral devices and for convenience mouse and keyboard were employed, however for better control it is suggested to use a joystick.



```
Action Mappings
  Enable Camera Rotation
    Right Mouse Button   Shift  Ctrl  Alt  Cmd
  Speed +
    Mouse Wheel Up       Shift  Ctrl  Alt  Cmd
  Speed -
    Mouse Wheel Down     Shift  Ctrl  Alt  Cmd
Axis Mappings
  Forward and Backwards
    W         Scale 1,0
    S         Scale -1,0
  Left and Right
    D         Scale 1,0
    A         Scale -1,0
  Up and Down
    E         Scale 1,0
    Q         Scale -1,0
  Look Up and Down
    Mouse Y   Scale 1,0
  Look Left and Right
    Mouse X   Scale 1,0
  Roll
    1         Scale 1,0
    3         Scale -1,0
```

Figure 3.16: Mappings to set keys and axis to input behavior.

Next, the blueprint for the camera actor is compiled. An event graph is built around each input so that it responds as desired during path tracking. Figure 3.17 shows the construct for the Forward and Backwards, Left and Right, Up and Down translations as well as for the three rotations: a Boolean variable causes the command to be activated only when the right mouse button is pressed, thus allowing to look up, down, left and right. The roll angle can instead be modified using the keyboard. Each rectangle is a variable, an information storage container that stores a value or reference to an object. Those in red are *Events* that provide the current value of the associated input axis once per frame when input is enabled for the containing actor. In green are the Float variables, a floating-point data type, while in blue are object variables that add a delta to the location of the component in world space.

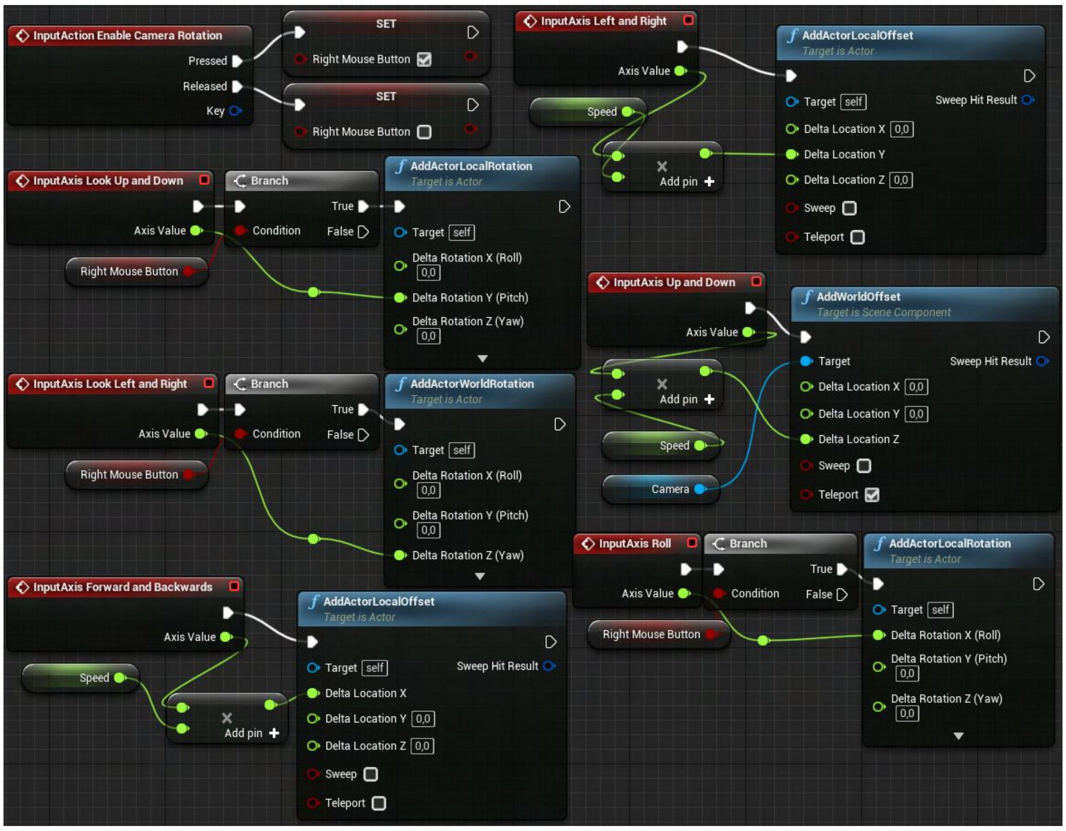


Figure 3.17: Blueprint with the commands desired to move and rotate the camera.

The Actions that allow to adjust within a defined interval the speed at which the camera moves are instead shown in Figure 3.18.

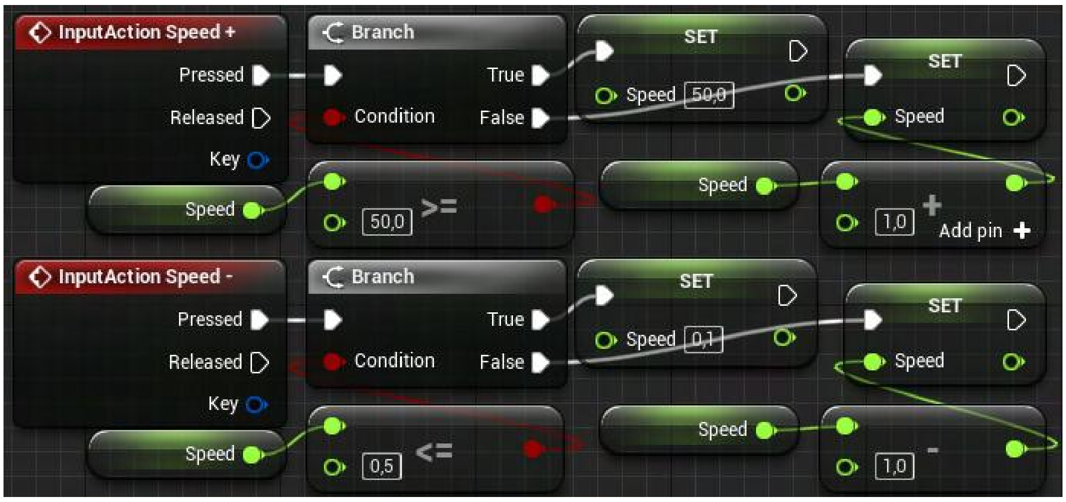


Figure 3.18: Blueprint with the commands necessary to change the speed of the camera.

Finally, an Event Tick is called every frame to print two strings, both to the log and on the screen. These strings contain the location and the rotation of the component in world space (Figure 3.19).

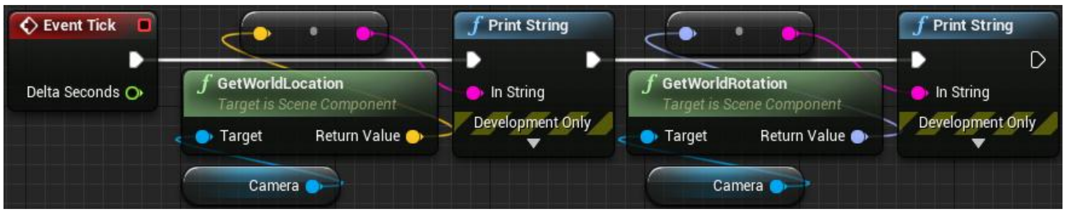
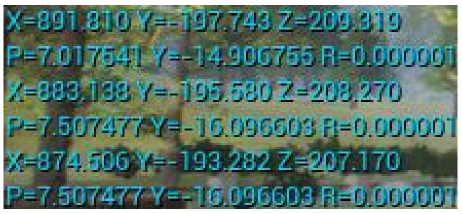


Figure 3.19: Blueprint with the commands necessary to print the pose.

At this point everything is properly set up to generate the data that must then be entered into the *Simulink* model to define the pose, during the simulation. The moment the level is started, it is possible to move and plot the path that we want the vehicle to follow. The log is compiled in real time frame-by-frame with the coordinates (X, Y, Z) in centimeters and the $(Pitch, Yaw, Roll)$ attitude values in degrees within the range $[-180, +180]$. These same values are also shown in screen, as depicted in Figure 3.20 where the 6 DoF are listed for three frames.



```
X=891.810 Y=-197.743 Z=209.319
P=7.017641 Y=-14.906755 R=0.000001
X=883.138 Y=-195.580 Z=208.270
P=7.507477 Y=-16.096603 R=0.000001
X=874.506 Y=-193.282 Z=207.170
P=7.507477 Y=-16.096603 R=0.000001
```

Figure 3.20: Pose values printed during the definition of the camera movement.

Once the motion has been fully outlined, the log output is copied into *Excel* to format it in such a way that can be used in *Matlab*. Proper formatting was achieved by a dedicated set of sub-routines written in *Visual Basic* (see Appendix A). The Excel *.xlsx* file obtained is then imported in *Matlab* and converted in a table, which is in turn decomposed into its columns, finally obtaining the desired pose vectors. Once the appropriate dimensional conversions have been made, the vectors are paired with a time vector, as in the case of paragraph 3.8.1, that progresses by a certain Δt for each frame. The resulting M-by-2 matrix associated with each DoF is the input required by the *Simulation 3D UAV Vehicle* block in *Simulink*.

It is important to point out that the operation just described, in which the level in *UE* is started to draw the path, must be undertaken in a second project copy of the one that is interfaced to *Simulink*, but without the *Mathworks* plug-ins. This is because once a project has those plug-ins installed, it can only be started from the

Simulation 3D Scene Configuration block. As a result, it is not possible to command actors other than those defined within *Simulink*.

Chapter 4

Tests Description and Results

This chapter presents by what criteria the tests were conducted and what the respective results are. Once the parameters used are defined, five tests are shown: one with the KITTI dataset Sequence 07, and four within the virtual environment. Specifically, of the four simulations, three are devoted to evaluating the effect of some features encountered in the KITTI scenario, while the last one aims to bring together all reproducible traits to compare whether the results are similar to those derived from real images.

4.1 KITTI dataset Test Setup

Resuming from Section 3.2, it is now presented in detail how the ORB-SLAM2 algorithm was set up, how the KITTI dataset was used and what errors were obtained in the pose estimation.

4.1.1 ORB-SLAM2 parameters

The role of most of the parameters given here has already been discussed during the description of the ORB-SLAM2 system. Therefore, please refer to Chapter 2 for the appropriate insights. Excerpts are given directly from the *Matlab* code [25], which has been modified to suit the needs of this test.

```
% Intrinsics parameters, stereo pair distance (baseline) and image
   size.
focalLength    = [707.0912 707.0912];    % spec in pixels
```

```
principalPoint = [601.8873 183.1104];     % spec in pixels [x, y]
baseline       = 0.54;                    % spec in meters
imageSize      = size(currILeft,[1,2]);   % [1242,375] pixels
```

```
% Number of downsamples (numLevels) and scale factor to achieve
   scale invariance of the ORB features from the rectified stereo
   pair.
scaleFactor = 1.2;
numLevels   = 8;

% Disparity range for 3D scene reconstruction
disparityRange = [0 48];   % spec in pixels

% Number of ORB features points uniformly distributed throughout
   each image
numPoints = 2000;
```

`scaleFactor` and `numLevels` are the default values of the algorithm, and they have been proven to offer the best performance compromise. The disparity range was taken by following the procedure in Section 2.3.3. `numPoints` is appropriate for the resolution of the images, and corresponds to the number also used by *Raùl Mur-Artal & Juan D. Tardòs* [12] for the same sequence.

```
 % Offline creation of the bag-of-features
imds = imageDatastore("C:\Path","FileExtensions",[".png"])
bag = bagOfFeatures(imds,CustomExtractor=
   @helperORBFeatureExtractorFunction,TreeProperties=[3, 10],
   StrongestFeatures=1);
% Load the bag of features
bofData  = load("BoF_KITTI_07.mat");
```

The bag of features specific for the scenario was created offline and uploaded in the system as a *.mat* file.

```
% Number of frames skipped n and minimum number of map points m
   tracked to identify a key frame
numSkipFrames     = 4;
numPointsKeyFrame = 120;
```

`numSkipFrames` heavily affects the speed and result of the method. A rather low value was chosen because of the small number of frames per second. The optimal value was found by trial and error, making 10 runs, each time increasing by 1 the parameter. The considerations made in section 2.5.2 apply.

```
% Creation of new map points by triangulation
```

```
minNumMatches = 30;  % Minimum number of matched feature points for
    a connection between views to be valid.
```

This `minNumMatches` value was deemed appropriate with the number of feature points being matched.

```
AbsoluteTolerance=1e-7  % termination tolerance of mean squared
   reprojection error in pixels.
RelativeTolerance=1e-16 % termination tolerance of relative
   reduction in reprojection error between iterations.
Solver='preconditioned-conjugate-gradient'  % Because of high
   sparsity
MaxIteration=150  % Default value =50
```

During refinement of local key frames and map points through bundle adjustment, the maximum number of iterations before Levenberg-Marquardt algorithm stops has been incremented.

```
if currKeyFrameId > 250   % Start looking for closure after key
   frame 250

    loopEdgeNumMatches = 50;  % Minimum number of feature matches
   of loop edges
    [...]
end
```

This value for `currKeyFrameId` allows the search to begin just a few key frames before the vehicle returns to the starting position. In any case, it was verified that there were no erroneous detection before the same position was reached by reducing the value of this parameter to 10.

```
minNumMatches_opt = 15  % minimum number of matched feature points
Tolerance = 1e-16  % tolerance of the optimization cost function
MaxIterations = 300 % Levenberg Marquardt opt algorithm maximum
   number of iterations
```

For pose graph optimization, it was set a new variable for the minimum number of matched feature points for a connection between views to be valid. The starting algorithm employed the same value used to create new points by triangulation, but it proved to be not optimal.

```
    gpsData     = load("GPS_KITTI_07.mat");
    timestamps  = load("Times07.mat");
```

Pose from the KITTI dataset and the timestamps for each frame are finally loaded in order to evaluate the error.

At this point it is necessary to make an important observation. Tuning the parameters for a visual SLAM system can be hard and requires a lot of heuristics. As has been noted by the developers themselves, the *Matlab* implementation of the ORB-SLAM2 method is undergoing improvements to increase its robustness and predictability. As an example, a small variation of `numPoints`, in the order of just tens of points, can significantly degrade the pose estimate. The same can be said for other parameters, such as `numSkipFrames`, `numPointsKeyFrame` and `minNumMatches`. It is therefore clear how balancing the algorithm as best as possible can become an extremely tedious process. This made it particularly difficult to find a set of values that returned acceptable errors and, most importantly, could be used with different input images without significant modification. The values previously listed result from a long series of trials and are the ones I considered best. In particular, the search for greater precision was interrupted when levels of accuracy comparable with those obtained from the demonstration test provided by *Mathworks* were achieved.

4.1.2 Ground Truth

As mentioned, the ground truth is given by a laser scanner and an inertial navigation system with high accuracy. These values were assumed to be exact and taken as an absolute reference to quantify the estimation error generated by the SLAM method. The pose is provided in a text file in which each row shows the first 3 rows of a 4x4 homogeneous pose matrix flattened into one line, so that

```
    R11 R12 R13 Tx R21 R22 R23 Ty R31 R32 R33 Tz
```

represents the matrix

```
    R11 R12 R13 Tx
    R21 R22 R23 Ty
    R31 R32 R33 Tz
    0   0   0   1
```

where the 3×3 sub-matrix $\mathbf{R}$ expresses the rotation and the column vector $\mathbf{T}$ the position, both with respect to the initial pose:

```
    1.000000e+00 1.197625e-11 1.704638e-10 5.551115e-17 1.197625e
   -11 1.000000e+00 3.562503e-10 0.000000e+00 1.704638e-10 3.562503
   e-10 1.000000e+00 2.220446e-16
```

that, approximating with sufficient accuracy and putting into matrix form, is

```
    1  0  0  0
    0  1  0  0
    0  0  1  0
```

There are 1101 of these matrices, one for each frame of the sequence.
In order to refer these data to the global reference system used in *Matlab* and make a comparison with the estimation from the SLAM algorithm, $\mathbf{R}$ is right multiplied by the matrices $[0\quad 0\quad 1;\quad 0\quad 1\quad 0;\quad -1\quad 0\quad 0]$ and $[0\quad -1\quad 0;\quad 1\quad 0\quad 0;\quad 0\quad 0\quad 1]$, so that two elemental rotations around the y and z axes are performed. Subsequently, the rotated matrix is converted to Euler angles according to the *ZYX* sequence, thus deriving the values in radians for the roll, yaw, and pitch angles which were then converted to degrees in the graphs below to simplify their interpretation. Referring again to the sensor setup (Figure 3.2), it can be seen that there is a position offset between the GPS/IMU unit and the left camera *Cam 2*. To avoid having a constant bias error when comparing the two trajectories, the two sensors' initial pose were made to coincide at the center of the global reference system. This assumption was made because the sensors are rigidly mounted to the car, and never vary their relative position.

4.1.3 Camera Projection Matrix

The calibrated camera parameters are also made available to the users through a text file containing 5 matrices expressed as 12 elements rows. The first four of these matrices are the 3×4 projection matrices $\mathbf{P}$ for each camera, describing the mapping of a pinhole camera from homogeneous 3D world points to homogeneous 2D image points. The fifth row is instead the concatenation of all camera positions. $\mathbf{P}$ is in the block-form:

$$\mathbf{P} = [\mathbf{M}| - \mathbf{MC}] \tag{4.1}$$

where $\mathbf{M}$ is an invertible 3×3 matrix and $\mathbf{C}$ is the camera center in world coordinates. In particular, for the sequence 07 and the color camera *Cam 2*, the projection matrix is:

```
 707.9012 0 601.8873 46.8878 0 707.0912 183.1104 0.1179 0 0 1
0.0062
```

that is

```
 707.9012  0         601.8873  46.8878
 0         707.0912  183.1104  0.1179
 0         0         1         0.0062
```

while for the other camera of the stereo pair *Cam 3*:

```
 707.9012 0 601.8873 -333.4597 0 707.0912 183.1104 1.9301 0 0 1
0.0033
```

that is

```
    707.9012   0          601.8873   -333.4597
    0          707.0912   183.1104   1.9301
    0          0          1          0.0033
```

This matrix by itself, however, does not explicitly express the camera pose or its internal geometry. It was then necessary to decompose $\mathbf{P}$ into the intrinsics matrix $\mathbf{K}$ and extrinsics matrix $\mathbf{E}$. It can be demonstrated that $\mathbf{P}$ can also be written as:

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] = \mathbf{K}\mathbf{E} \tag{4.2}$$

where $\mathbf{K}$ is a 3×3 upper triangular matrix, $\mathbf{R}$ is a orthogonal 3×3 rotation matrix whose columns are the directions of the world axes in the camera's reference frame, and $\mathbf{t} = -\mathbf{RC}$ is the vector containing the position of the world origin in camera coordinates.
$\mathbf{C}$ is simply obtained by left-multiplying the last column of $\mathbf{P}$ by $-\mathbf{M}^{-1}$. While, given the properties of $\mathbf{K}$ and $\mathbf{R}$, it is possible to recover them by using RQ decomposition. RQ decomposition in not a function available in *Matlab*, so an external file [31] was downloaded from *Matlab File Exchange* and implemented.
As a result, the $\mathbf{K}$ and $\mathbf{E}$ matrices were finally obtained. For *Cam 2*:

$$\mathbf{K} = \begin{bmatrix} 707.0912 & 0 & 601.8873 \\ 0 & 707.0912 & 183.1104 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.3}$$

$$\mathbf{E} = \begin{bmatrix} 1 & 0 & 0 & 0.0610 \\ 0 & 1 & 0 & -0.0014 \\ 0 & 0 & 1 & 0.0062 \end{bmatrix}. \tag{4.4}$$

And, for *Cam 3*:

$$\mathbf{K} = \begin{bmatrix} 707.0912 & 0 & 601.8873 \\ 0 & 707.0912 & 183.1104 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.5}$$

$$\mathbf{E} = \begin{bmatrix} 1 & 0 & 0 & -0.4744 \\ 0 & 1 & 0 & 0.0019 \\ 0 & 0 & 1 & 0.0033 \end{bmatrix}. \tag{4.6}$$

As expected, the inrtinsic matrix $\mathbf{K}$ is the same for both cameras and it will be used to define the values of the reprojection matrix. Looking at the extrinsic matrix instead, the rotation sub-matrix $\mathbf{R}$ is an identity since the images are already rectified, while $\mathbf{t}$ differs because of the different position of the cameras on the vehicle. In particular, along the x-axis the baseline distance is confirmed: $|-0.4744| + 0.0610 = 0.5354 \approx 0.54m$.

4.1.4 Results

Figure 4.1 shows the full trajectory in the XY plane plotted by means of *Matlab*'s Point Cloud Player. Three curves of three different colors can be distinguished: in green is the trajectory provided by the GPS/IMU system and thus serving as a reference; in red is the estimated trajectory, obtained in real time during frame analysis; and in magenta is the optimized trajectory, created as a result of loop closure recognition and pose graph optimization. The thousands of points surrounding the trajectories are instead the 3D map points. The color gradient indicates the elevation of the points, from dark blue to yellow as Z increases. Although this is a sparse map, it is interesting to notice that the number of points extracted is large enough that some characteristics of the environment can be recognized. The walls of the buildings lining the street are well defined, as well as in some sections the sidewalks, some parked vehicles, trees and signposts.

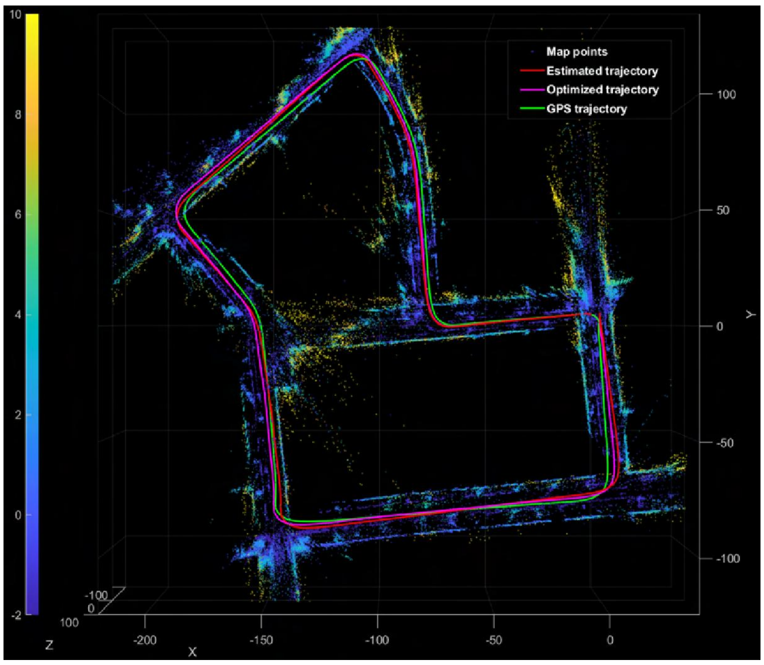


Figure 4.1: KITTI's Sequence 07 Trajectory.

The route starts from coordinates $(X, Y, Z) = (0, 0, 0)$, with almost immediately a 90 degrees left turn, and then closes again after traveling about 686 meters in 110.5 seconds.

This test performed using the sequence from the KITTI dataset is used here to illustrate how the errors were derived. These same criteria will also be used later for all other tests so that a meaningful comparison can be made. For this case only, in addition, the errors between the estimated and optimized trajectory are also compared to demonstrate the importance of loop closure recognition and subsequent optimization. Since the purpose here is not to evaluate the goodness of the optimization process of the ORB-SLAM2 system, only the optimized result will be considered for the later simulations in the virtual environment, knowing that it has an overall lower error than the estimated one.

The timestamps and the associated $[X, Y, Z]$ positions have been used to derive the distance covered over time and the instantaneous speed, as shown in Figure 4.2. The top speed is $11.6\,m/s$ while the average speed is $6.2\,m/s$.

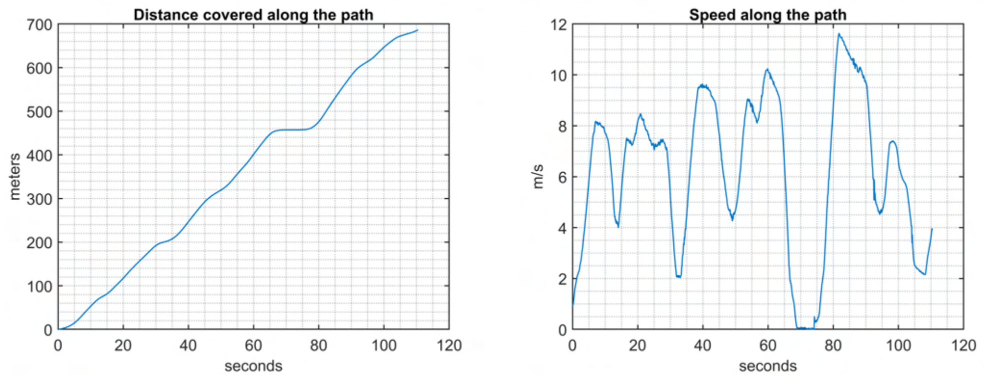


Figure 4.2: Distance covered and speed along the path.

To evaluate the error there are two alternatives generally used in SLAM, it can either refer to the distance covered or to the elapsed time. In this sequence, the vehicle moves at a highly variable speed, even coming to a complete stop in the interval $67s - 74s$. This fact is particularly problematic when plotting the errors because different key frames are captured in the same position, thus providing different error values associated with the same distance covered. For this reason, all the following graphs will refer to the elapsed time.

Fig. 4.3 shows for all six d.o.f. the ground truth and the estimated and optimized values.

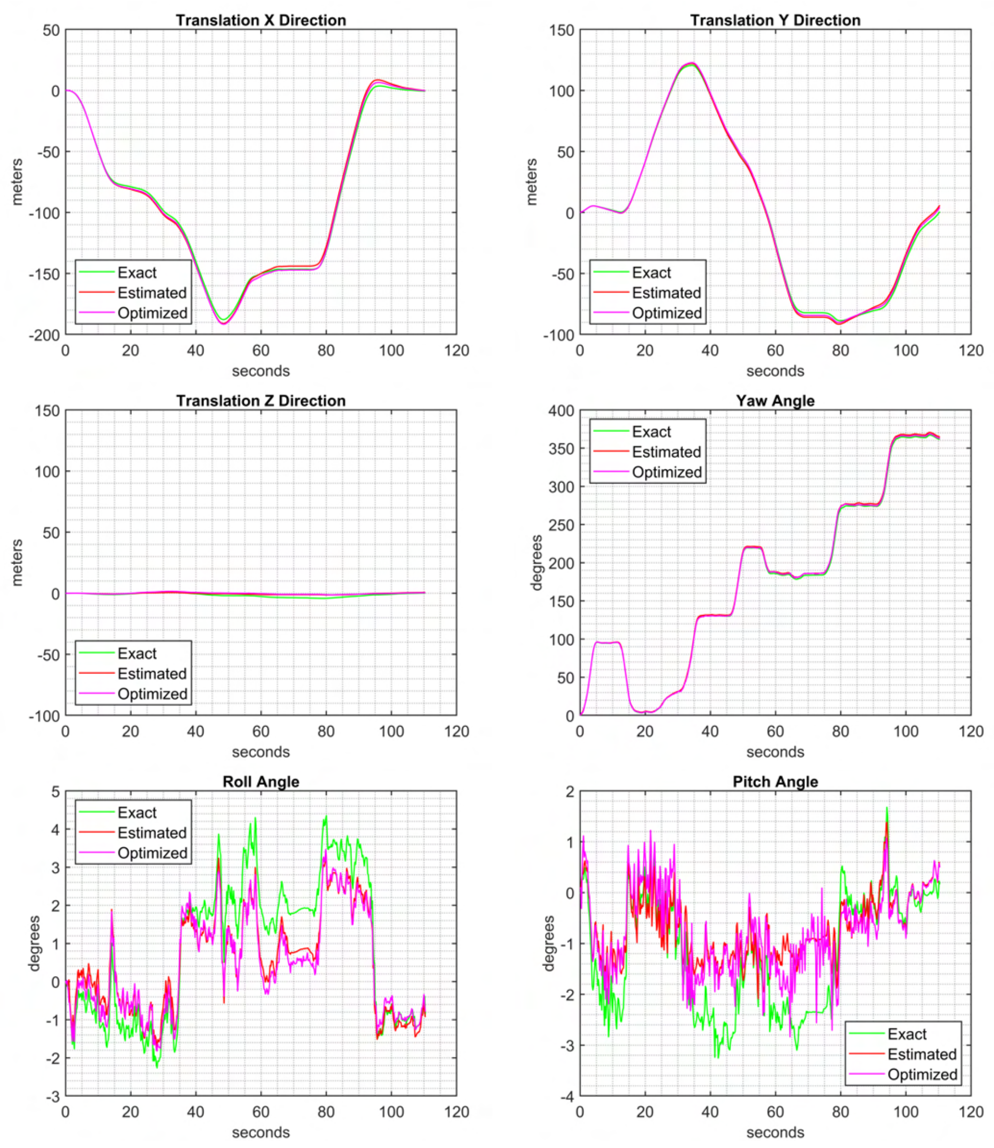


Figure 4.3: X, Y, Z translations and Yaw, Roll, Pitch angles.

For further proof of the correctness of the physical meaning of the plotted rotations, reference was made to the sequence's video. The frames from the dataset were used to reconstruct the 10 fps video through the following *FFmpeg* code line:

```
@ECHO OFF
ffmpeg -r 10 -i %%06d.png -c:v libx264 -vf scale=1242:376 -pix_fmt
   yuv420p -crf 20 -an output.mp4
pause
```

Note that it was necessary to add a row of pixels (from 375 to 376) because this operation is not supported for odd numbers in the resolution. The obtained video provided visual confirmation for the the yaw angle. Confirmation for the pitch angle, on the other hand, comes from the fact that it is associated with the change

in position along the Z direction.
The errors for the *i-th* pose were obtained as

$$(Error)_i = (GT\ Value)_i - (Predicted\ Value)_i \quad for\ i = 1, \dots, N \qquad (4.7)$$

where N is the total number of key frames found. They are given here for all six d.o.f. (Fig. 4.4).

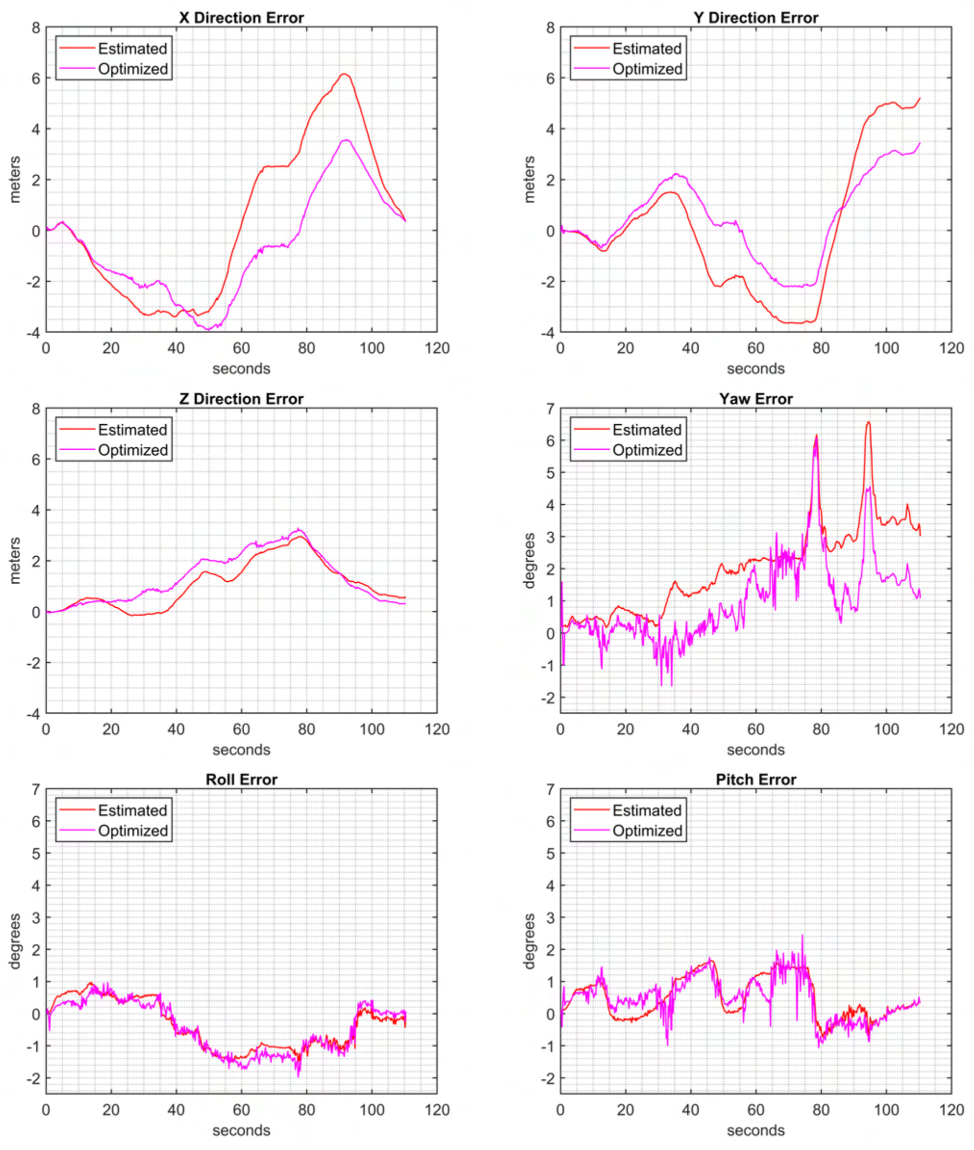


Figure 4.4: Translation and Rotation Errors.

Another aspect that requires attention when plotting the errors concerns the loop closure and at which frame it occurs. As mentioned, there are a total of 1101 frames, however closure is recognized in frame 1064 and the run is stopped. To make the three trajectories correspond at every instant it is therefore necessary to take the exact number of frames used, and not the entire sequence. This would lead to a slight shift in values with a consequent increase in the error since poses in different frames would be compared.

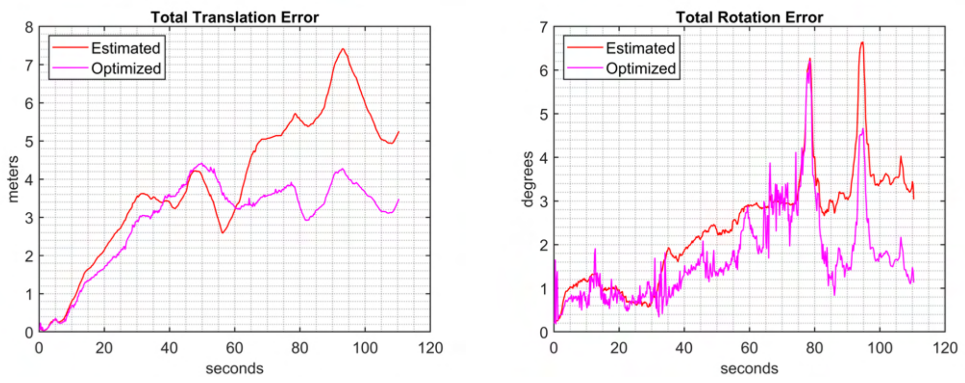


Figure 4.5: Total Translation and Rotation Errors.

The absolute total translation and rotation errors (Fig 4.5) for the *i-th* key frame are given by

$$\begin{aligned} Tr\_Error_{tot} &= \sqrt{\sum_{j=1}^{3}(\mathbf{t}_{GT,j} - \mathbf{t}_{Predicted,j})^2}, \\ Rot\_Error_{tot} &= \sqrt{\sum_{j=1}^{3}(\boldsymbol{\alpha}_{GT,j} - \boldsymbol{\alpha}_{Predicted,j})^2} \end{aligned} \tag{4.8}$$

where $\mathbf{t} = [X, Y, Z]$ and $\boldsymbol{\alpha} = (\psi, \theta, \phi)$. As expected, the error for the optimized trajectory is overall significantly reduced with respect to the the estimated one. Some conclusions can also be made about how the error varies at certain significant events. In general, the most important variations in the translation error occur at sections where the vehicle is moving at higher velocity, as can be seen from the left graph in Fig. 4.6, where the instantaneous speed have been superimposed to the total translation error. When the car stops at the intersection at second 67, six vehicles pass in quick succession in front of it at an estimated speed of about $11\,m/s$. During this period the translation error is almost stabilized in the XY plane even though it continues to increase slightly along the Z direction. Similarly, the error on rotations also remains rather stable here. This demonstrates a fairly good behaviour of the algorithm in situations where there is no motion of the cameras and the environment has dynamic objects. The left graph in Fig. 4.6 shows instead the total rotation error and the speed, which is here useful to see when sharp turns occurs (that is, when the speed plummets). The rotation error shows significant increases precisely at these turns. In particular, the Yaw error has very important peaks at the 90-degree corners at $74s$ and $94s$. In the first case, the vehicle starts from a standstill for a fast turn just as a car is passing in front. After following said car in a straight stretch, both turn left at sustained speed, producing a situation similar to the previous one where a moving object is framed for a decent amount of time.

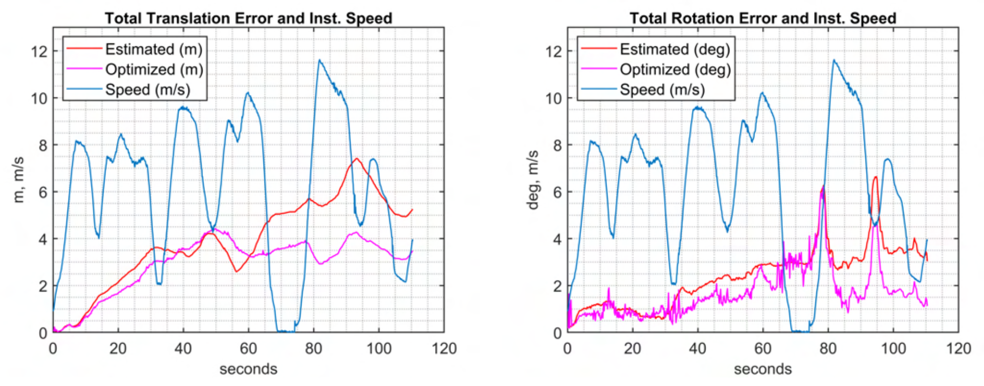


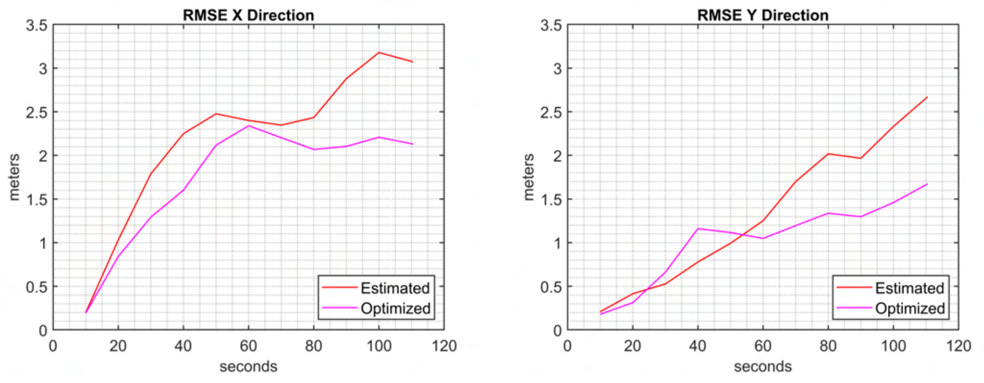
Figure 4.6: Translation and Rotation Errors shown with the instantaneous speed.

Certainly not insignificant is the presence throughout the sequence of dark shadows alternating with areas highly illuminated by sunlight. However, no particular events associated with the lighting conditions have been recognized as the cause of a significant increase in error.

The *Root Mean Square Error* is then given in Figure 4.7 as

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Error_i)^2} \tag{4.9}$$

where the average value is calculated for key frames distributed every ten seconds, so that it is obtained in the intervals $0s - 10s$, $0s - 20s$, $0s - 30s$, and so on, up to the overall *RMSE* value for the whole course.



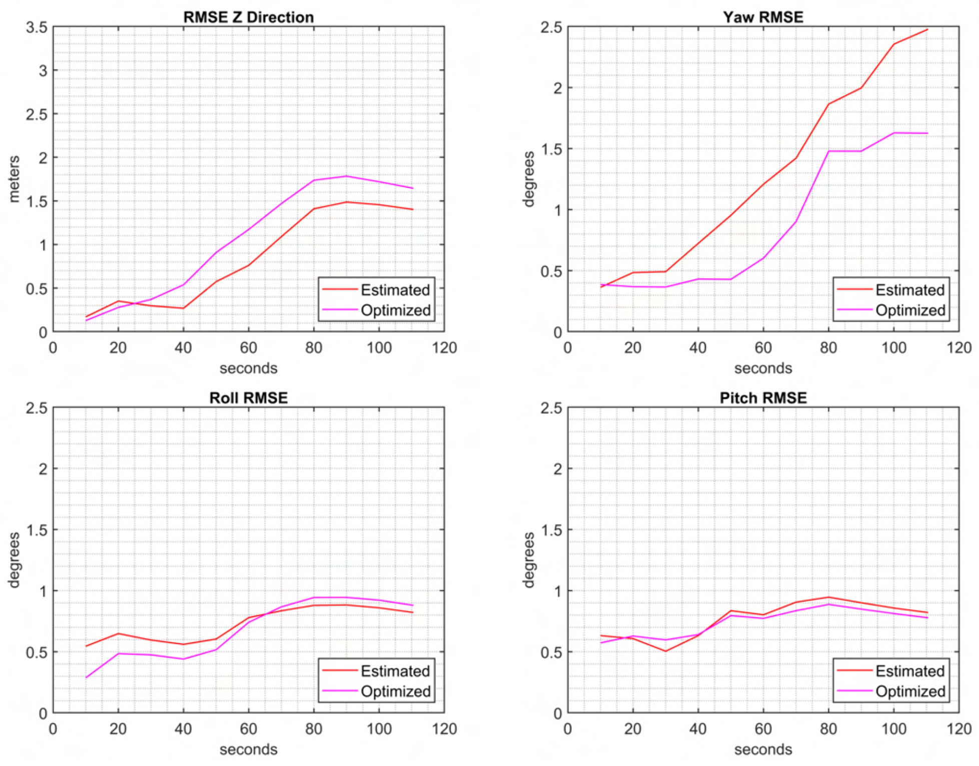
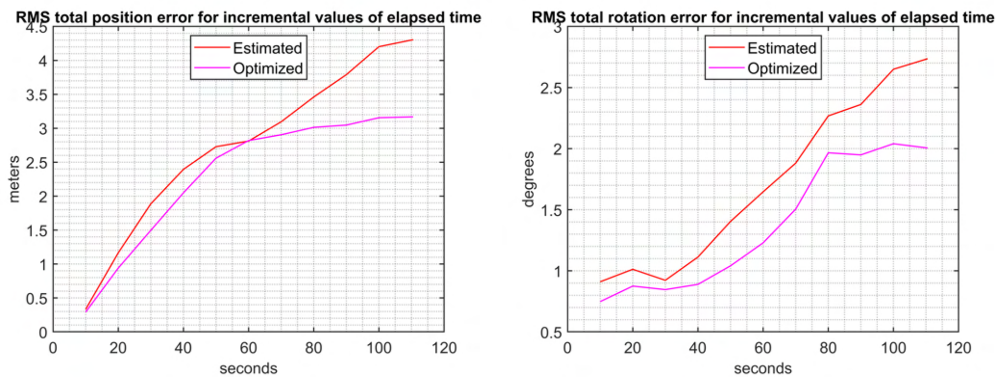


Figure 4.7: Translation and Rotation RMS Errors.

The overall RMS errors are presented in Fig. 4.8 and the respective values are summarized in Table 4.1. The RMS error can also be represented with respect to the distance covered instead of the elapsed time given that it is averaged over a certain length. The average value is here taken for cumulative distance values of 50 meters, up to the total error mediated over the entire length. As expected the trend is similar but not identical because of the variable speed of the vehicle. Referring to the distance covered gives a more immediate understanding of which sections increase the error the most.



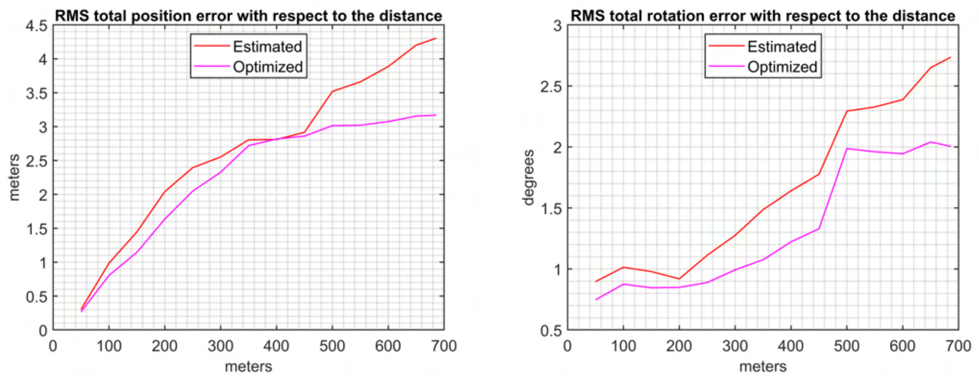


Figure 4.8: Overall RMS Errors with respect to the elapsed time and the covered distance.

Table 4.1: RMSE Errors in the KITTI Sequence 07.

Time interval	RMSE Pos. Est.	RMSE Pos. Opt.	RMSE Rot. Est.	RMSE Rot. Opt.
$0\,s - 10\,s$	$0.3333\,m$	$0.2920\,m$	$0.9110\,deg$	$0.7506\,deg$
$0\,s - 20\,s$	$1.1649\,m$	$0.9403\,m$	$1.0141\,deg$	$0.8766\,deg$
$0\,s - 30\,s$	$1.8876\,m$	$1.4973\,m$	$0.9225\,deg$	$0.8480\,deg$
$0\,s - 40\,s$	$2.3946\,m$	$2.0492\,m$	$1.1115\,deg$	$0.8881\,deg$
$0\,s - 50\,s$	$2.7286\,m$	$2.5597\,m$	$1.4037\,deg$	$1.0370\,deg$
$0\,s - 60\,s$	$2.8191\,m$	$2.8191\,m$	$1.6444\,deg$	$1.2261\,deg$
$0\,s - 70\,s$	$3.0958\,m$	$2.9051\,m$	$1.8793\,deg$	$1.5069\,deg$
$0\,s - 80\,s$	$3.4609\,m$	$3.0139\,m$	$2.2689\,deg$	$1.9652\,deg$
$0\,s - 90\,s$	$3.7901\,m$	$3.0470\,m$	$2.3606\,deg$	$1.9480\,deg$
$0\,s - 100\,s$	$4.2024\,m$	$3.1555\,m$	$2.6471\,deg$	$2.0397\,deg$
$0\,s - 110.5\,s$	$4.3045\,m$	$3.1684\,m$	$2.7387\,deg$	$2.0053\,deg$
Distance interval	RMSE Pos. Est.	RMSE Pos. Opt.	RMSE Rot. Est.	RMSE Rot. Opt.
$0\,m - 50\,m$	$0.3040\,m$	$0.2672\,m$	$0.8966\,deg$	$0.7473\,deg$
$0\,m - 100\,m$	$0.9845\,m$	$0.8043\,m$	$1.0140\,deg$	$0.8753\,deg$
$0\,m - 150\,m$	$1.4469\,m$	$1.1460\,m$	$0.9787\,deg$	$0.8460\,deg$
$0\,m - 200\,m$	$2.0413\,m$	$1.6361\,m$	$0.9193\,deg$	$0.8489\,deg$
$0\,m - 250\,m$	$2.3946\,m$	$2.0492\,m$	$1.1127\,deg$	$0.8893\,deg$
$0\,m - 300\,m$	$2.5521\,m$	$2.3282\,m$	$1.2758\,deg$	$0.9927\,deg$
$0\,m - 350\,m$	$2.8049\,m$	$2.7191\,m$	$1.4861\,deg$	$1.0758\,deg$
$0\,m - 400\,m$	$2.8104\,m$	$2.8174\,m$	$1.6404\,deg$	$1.2225\,deg$
$0\,m - 450\,m$	$2.9158\,m$	$2.8597\,m$	$1.7761\,deg$	$1.3293\,deg$
$0\,m - 500\,m$	$3.5188\,m$	$3.0140\,m$	$2.2938\,deg$	$1.9863\,deg$
$0\,m - 550\,m$	$3.6599\,m$	$3.0204\,m$	$2.3276\,deg$	$1.9606\,deg$
$0\,m - 600\,m$	$3.8858\,m$	$3.0734\,m$	$2.3878\,deg$	$1.9440\,deg$
$0\,m - 650\,m$	$4.2024\,m$	$3.1555\,m$	$2.6471\,deg$	$2.0397\,deg$
$0\,m - 686\,m$	$4.3045\,m$	$3.1684\,m$	$2.7378\,deg$	$2.0053\,deg$

To conclude, in Table 4.20 are summarized some significant quantities that will later be useful for making direct comparisons with the simulations in the virtual environment. In fact, since the scenarios and paths will still be different despite the similarities, a direct point-by-point comparison is meaningless. However, it remains of interest to compare magnitudes such as the maximum or the final error achieved.

Table 4.2: Quantities of interest.

Distance covered	686 m
Duration	110.5 s
Average speed	6.2 m/s
Maximum speed	11.6 m/s
Dynamic objects	7
Maximum translation error (opt)	4.4261 m
Maximum rotation error (opt)	6.1951 deg
Final translation error (opt)	3.4797 m
Final rotation error (opt)	1.1306 deg
RMS final translation error (opt)	3.1684 m
RMS final translation error in % of the distance covered (opt)	0.462%
RMS final rotation error (opt)	2.0053 deg

4.2 Virtual Scenario Performance Evaluation Tests

In conducting the first tests in the virtual environment described in Section 3.4.1, the need to develop some case studies aimed at quantifying the effect of some features encountered in the KITTI scenario was soon realized. In particular, three aspects have been evaluated: (1) the speed of the vehicle, (2) the presence of dynamic objects, (3) the lighting conditions.

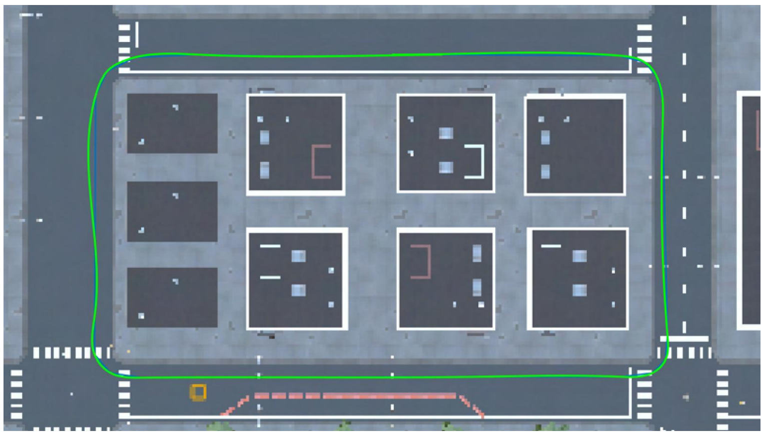


Figure 4.9: Top-down map of the path followed by the vehicle.

The following series of tests also highlights the strength of a virtual system as a testing platform, as it demonstrates the simplicity with which it is possible to change only targeted features of the environment, which can be very complex if not impossible to do in a real environment. Each evaluation is carried out following the same chosen path (Fig. 4.9) and the desired changes are applied from case to case. The path followed by the vehicle has been imposed by selecting waypoints on the top-down map of the scene and a distance of 270 meters is covered while proceeding at constant speed with the cameras always raised 1.65 m above the ground. Approximating the path with a rectangle, the longest side has a length of about 90 m while the shortest about 45 m. The initial pose is:

```
Starting_Position = [21.3924, 53.8970, 1.65];
Starting_Rotation = [0.0038, 0, 0];
```

The stereo videos that are obtained in *Simulink* are broken into frames using *FFmpeg* through the code:

```
E:\Video>ffmpeg -i Right_Output.avi %06d.png
```

where `Right_Output.avi` is the video generated by the right camera. The same operation is then performed for the left video. As in the case of the dataset, all sequences are recorded at 10 fps.

ORB-SLAM2 Parameters

The parameters are overall the same as those used previously. It was necessary to adjust from which `currKeyFrameId` to start the loop closure search and load the appropriate bag of features.
These values will not be altered in any of the tests that follow, so that changes in the results will be attributable only to modifications made to the scenario.

```
% Intrinsics parameters, stereo pair distance (baseline) and image
   size.
focalLength    = [707.0912 707.0912];    % spec in pixels
principalPoint = [601.8873 183.1104];    % spec in pixels [x, y]
baseline       = 0.54;                   % spec in meters
imageSize      = size(currILeft,[1,2]);  % [1242,375] pixels

%ORB features
scaleFactor = 1.2;
numLevels   = 8;
disparityRange = [0 48];   % spec in pixels
numPoints = 2000;   % ORB Feature points

% Bag of features
bofData   = load("BoF_Virtual_City.mat");
```

```
% Keyframes identification
numSkipFrames     = 4;
numPointsKeyFrame = 120;

% Map points by triangulation
minNumMatches = 30;

% Levenberg-Marquardt
AbsoluteTolerance=1e-7
RelativeTolerance=1e-16
MaxIteration=150

%Loop closure
if currKeyFrameId > 140   % Start looking for closure after key
    frame 250
     loopEdgeNumMatches = 120;  % Minimum number of feature matches
    of loop edges
     [...]
end

% Optimization
minNumMatches_opt = 15  % minimum number of matched feature points
Tolerance = 1e-16  % tolerance of the optimization cost function
MaxIterations = 300 % Levenberg Marquardt opt algorithm maximum
    number of iterations
```

4.2.1 Standard Condition

All tests will refer to a “standard condition”, which was chosen as the starting point from which to implement the modifications. Each time a new typology of changes is address, the simulation will restart from here so that it is possible to make an equal comparison.
In this scene there are no dynamic objects and the vehicle completes the course in 90 seconds, so that it keeps a constant speed of $3\,m/s$.

Results

The results for the simulation under standard condition are given below, always referring to the optimized pose only.

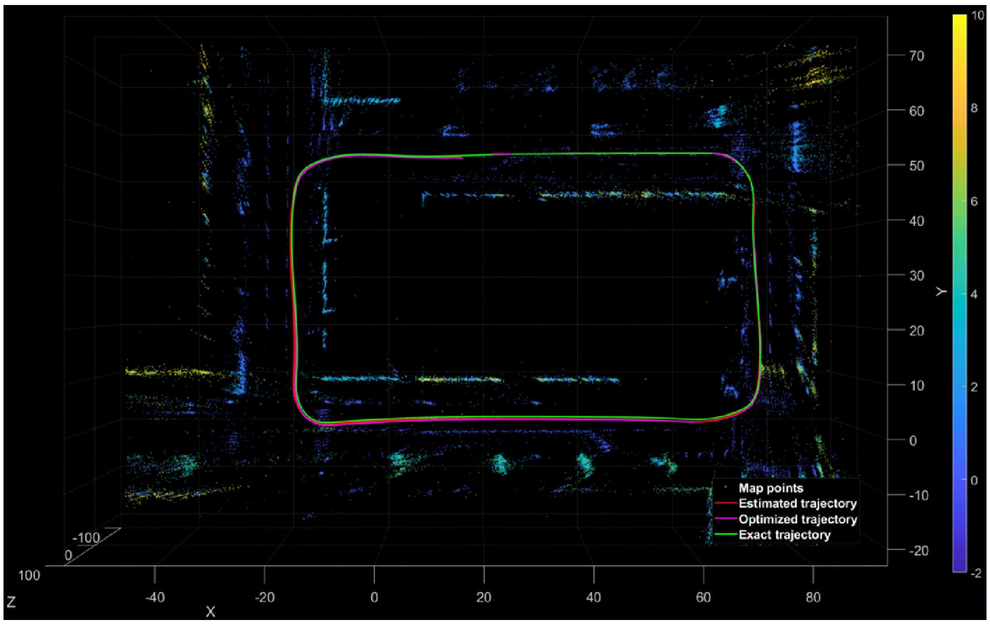


Figure 4.10: Plot of the run under standard conditions.

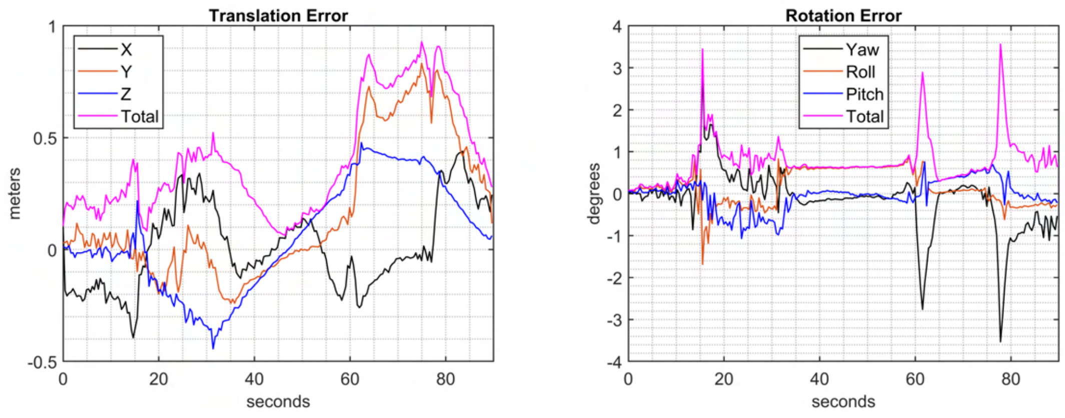


Figure 4.11: Translation and rotation errors in standard conditions.

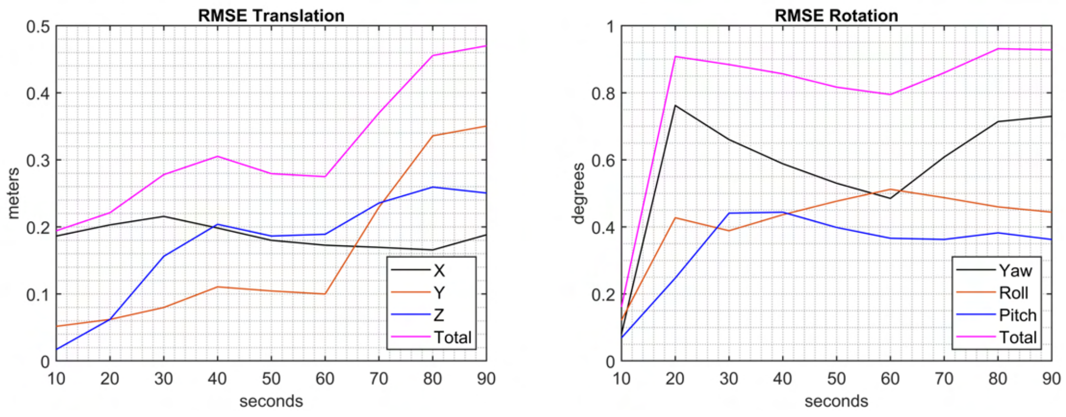


Figure 4.12: Translation and rotation RMS errors in standard conditions.

Table 4.3: Quantities of interest, Standard Conditions.

Distance covered	$270\,m$
Duration	$90\,s$
Constant speed	$3\,m/s$
Dynamic objects	No
Maximum translation error	$0.9286\,m$
Maximum rotation error	$3.5624\,deg$
Final translation error	$0.2789\,m$
Final rotation error	$0.6319\,deg$
RMS final translation error	$0.4701\,m$
RMS final translation error in % of the distance covered	0.1741%
RMS final rotation error	$0.9280\,deg$

4.2.2 Impact of the Speed of Motion

Constant Speed: 5 m/s

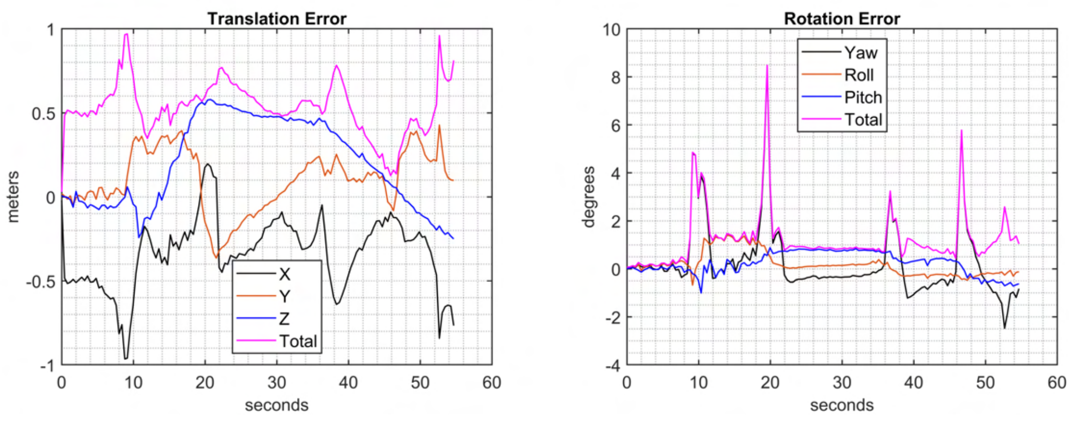


Figure 4.13: Translation and rotation errors with a constant speed of $5\,m/s$.

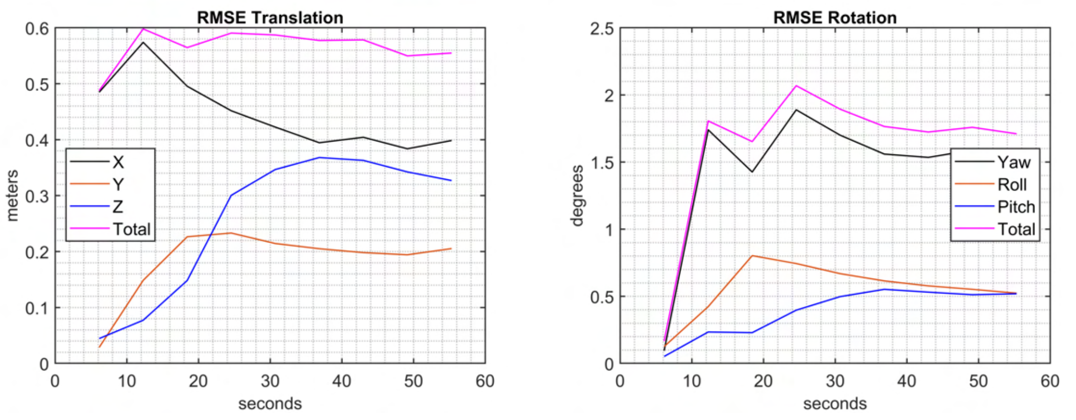


Figure 4.14: Translation and rotation RMS errors with a constant speed of $5\,m/s$.

Table 4.4: Quantities of interest, $5\,m/s$.

Distance covered	$270\,m$
Duration	$54\,s$
Constant speed	$5\,m/s$
Dynamic objects	No
Maximum translation error	$0.9696\,m$
Maximum rotation error	$8.4820\,deg$
Final translation error	$0.8117\,m$
Final rotation error	$1.0408\,deg$
RMS final translation error	$0.5548\,m$
RMS final translation error in % of the distance covered	0.2055%
RMS final rotation error	$1.7105\,deg$

Constant Speed: 7 m/s

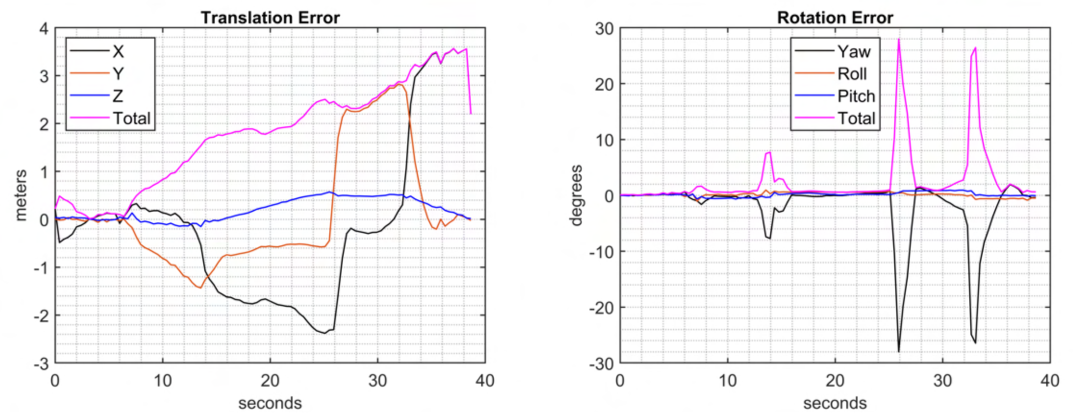


Figure 4.15: Translation and rotation errors with a constant speed of $7\,m/s$.

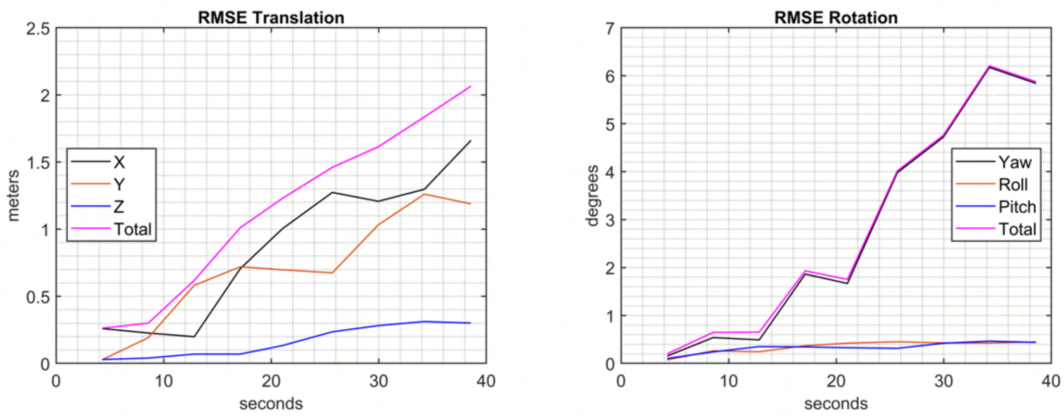


Figure 4.16: Translation and rotation RMS errors with a constant speed of $7\,m/s$.

Table 4.5: Quantities of interest, $7\,m/s$.

Distance covered	$270\,m$
Duration	$38.57\,s$
Constant speed	$7\,m/s$
Dynamic objects	No
Maximum translation error	$3.5597\,m$
Maximum rotation error	$28.0002\,deg$
Final translation error	$2.1974\,m$
Final rotation error	$0.6257\,deg$
RMS final translation error	$2.0648\,m$
RMS final translation error in % of the distance covered	0.7647%
RMS final rotation error	$5.8742\,deg$

Constant Speed: 9 m/s

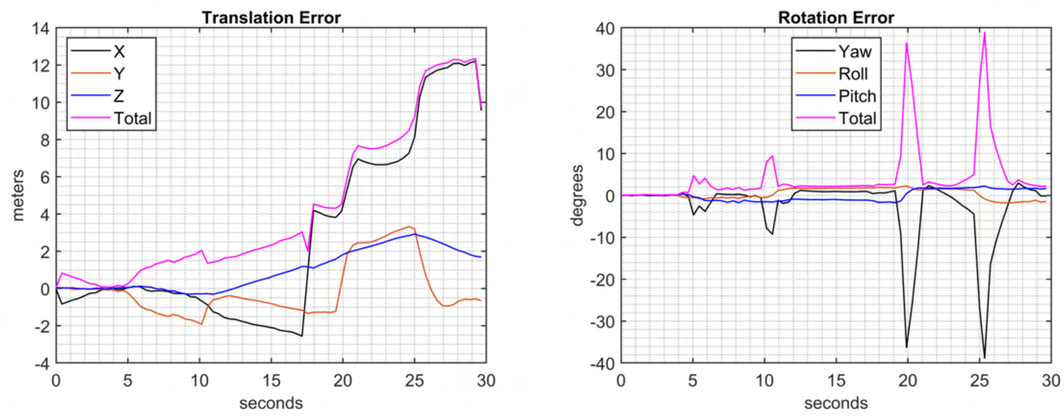


Figure 4.17: Translation and rotation errors with a constant speed of $9\,m/s$.

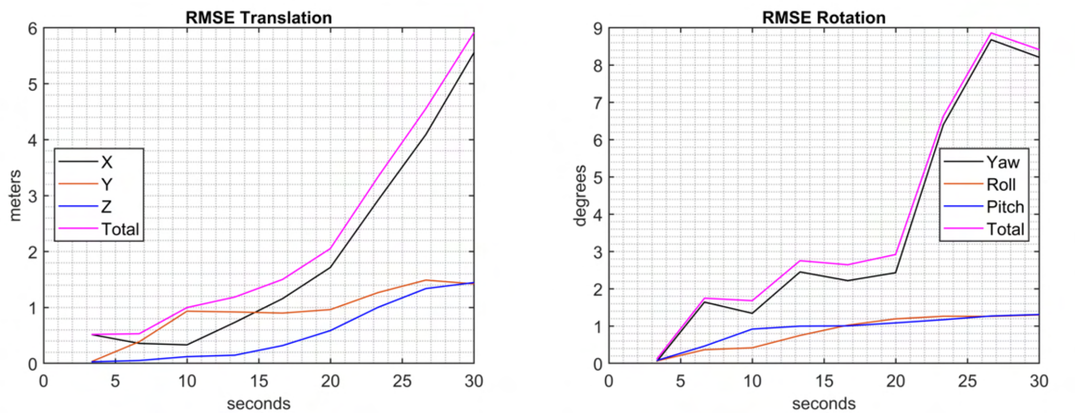


Figure 4.18: Translation and rotation RMS errors with a constant speed of $9\,m/s$.

Table 4.6: Quantities of interest, $9\,m/s$.

Distance covered	$270\,m$
Duration	$30\,s$
Constant speed	$9\,m/s$
Dynamic objects	No
Maximum translation error	$12.3367\,m$
Maximum rotation error	$38.8966\,deg$
Final translation error	$9.7278\,m$
Final rotation error	$2.1894\,deg$
RMS final translation error	$5.9160\,m$
RMS final translation error in % of the distance covered	2.1911%
RMS final rotation error	$8.4132\,deg$

Considerations on the velocity tests

Table 4.7 and Figures 4.19, 4.20 group the results for the four velocities at which the tests were conducted. When the vehicle faces a turn, there is a spike in the overall rotation error due to the rapid change in the yaw angle. Up to $5\,m/s$ this effect is fairly contained, although confirming the trend of increasing error as the speed increases. Making a comparison with the KITTI sequence (Fig.4.6), in which the sharp turns are taken at a speed between $4\,m/s$ and $5\,m/s$, it is appreciable how the results are similar. On the other hand, if the speed is increased further, the turns are estimated poorly by the algorithm as the available connected frames get smaller and smaller to track the scenery. In any case, it always recognizes with good accuracy how much the vehicle rotates, making it so that past the curve there is an error of only a few degrees. The estimate on translation also worsens

in straight sections as speed increases, with an increment in error as the distance covered increases (for example in the $50\,m - 100\,m$ and $100\,m - 150\,m$ stretches). Even here, however, it is the rotations that make tracking significantly worse. The moment there is a large error in orientation estimation, an offset is created between the exact and optimized values that is carried along the rest of the path. This offset fails to be corrected even with loop closure, although there is an improvement in the last meters.
To improve results at higher speeds it is possible to act directly on the algorithm by reducing `numSkipFrames` or by acting on the camera, using one that allows to record at a higher number of frames per second (in the case of real cameras, the presence of motion blur must also be considered).

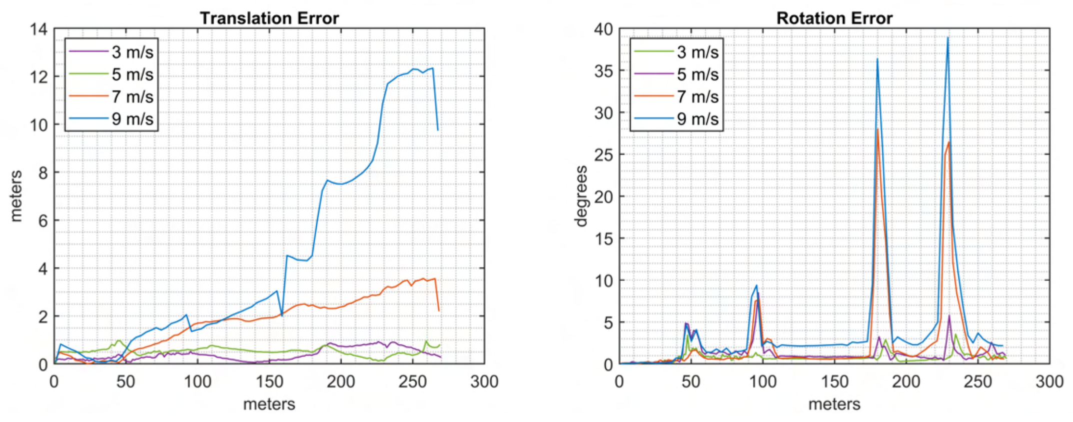


Figure 4.19: Translation and rotation errors comparison.

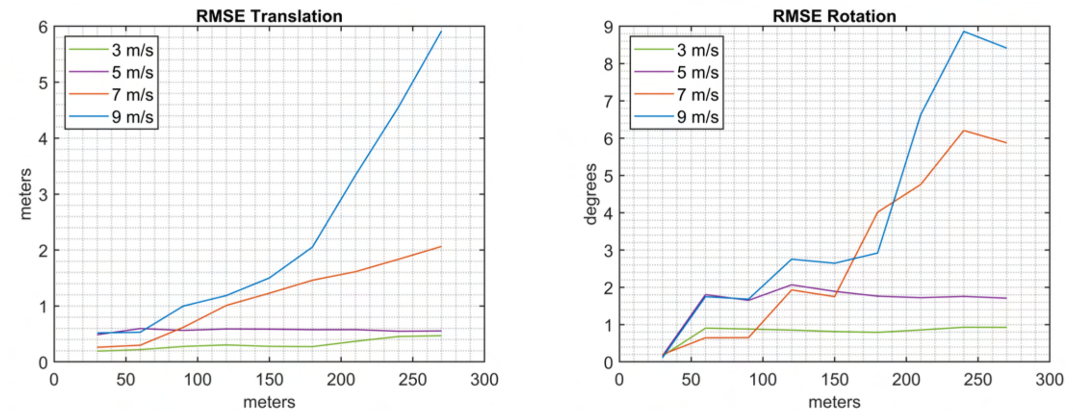


Figure 4.20: Translation and rotation RMS errors comparison.

Table 4.7: Comparison of the results.

	$3\,m/s$	$5\,m/s$	$7\,m/s$	$9\,m/s$
Distance covered	$270\,m$	$270\,m$	$270\,m$	$270\,m$
Duration	$90\,s$	$54\,s$	$38.57\,s$	$30\,s$
Maximum translation error	$0.9286\,m$	$0.9696\,m$	$3.5597\,m$	$12.3367\,m$
Maximum rotation error	$3.5624\,deg$	$8.4820\,deg$	$28.0002\,deg$	$38.8966\,deg$
Final translation error	$0.2789\,m$	$0.8117\,m$	$2.1974\,m$	$9.7278\,m$
Final rotation error	$0.6319\,deg$	$1.0408\,deg$	$0.6257\,deg$	$2.1894\,deg$
RMS final translation error	$0.4701\,m$	$0.5548\,m$	$2.0648\,m$	$5.9160\,m$
RMS final translation error in % of the distance covered	$0.1741\,\%$	$0.2055\,\%$	$0.7647\,\%$	$2.1911\,\%$
RMS final rotation error	$0.9280\,deg$	$1.7105\,deg$	$5.8720\,deg$	$8.4132\,deg$

4.2.3 Impact of dynamic objects on SLAM system performance

Six tests were conducted to assess the impact of moving vehicles within the scene. The first four tests form a general study of these elements, while the subsequent two are designed to reproduce situations similar to those found in the dataset.

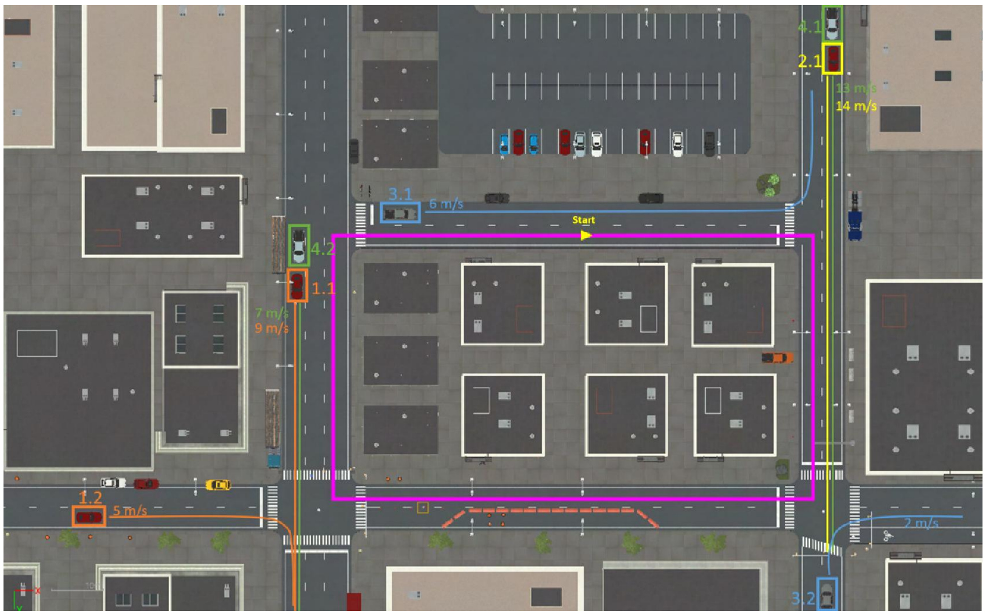


Figure 4.21: Top-down map showing the trajectories of the moving cars. In magenta is an approximation of the trajectory followed by the recording platform.

The set-up for the first four tests is illustrated in Figure 4.21: in each run is added a group of vehicles (Table 4.8), distinguished by color and numbered. A total of 7 vehicles were therefore progressively added to the scene, each one with a different speed and traveling different paths. Since their motion is in a loop, some of them appear in the frame several times (Table 4.9).
Vehicle trajectories and speeds are designed to present different situations that may typically occur in an urban scenario. There are vehicles overtaking the recording

platform, others going against it in the opposite lane, and others moving orthogonally to its direction. Moreover, they are repeatedly encountered in both straight and curved sections.

Table 4.8: Simulations set-up summary.

	Groups	N. of cars	N. of cars seen
Run 1	1	2	4
Run 2	1+2	3	7
Run 3	1+2+3	5	10
Run 4	1+2+3+4	7	14

Table 4.9: Speeds and number of appearances in the frame.

	Speed	Appearances
Car 1.1	$9\,m/s$	2
Car 1.2	$5\,m/s$	2
Car 2.1	$14\,m/s$	3
Car 3.1	$6\,m/s$	1
Car 3.2	$2\,m/s$	2
Car 4.1	$13\,m/s$	2
Car 4.2	$7\,m/s$	2

1) Run 1

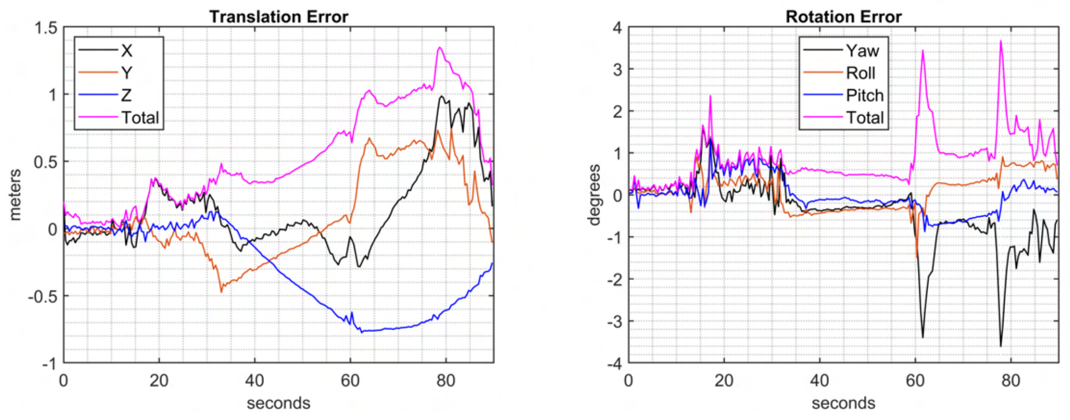


Figure 4.22: Translation and rotation errors, 4 moving cars.

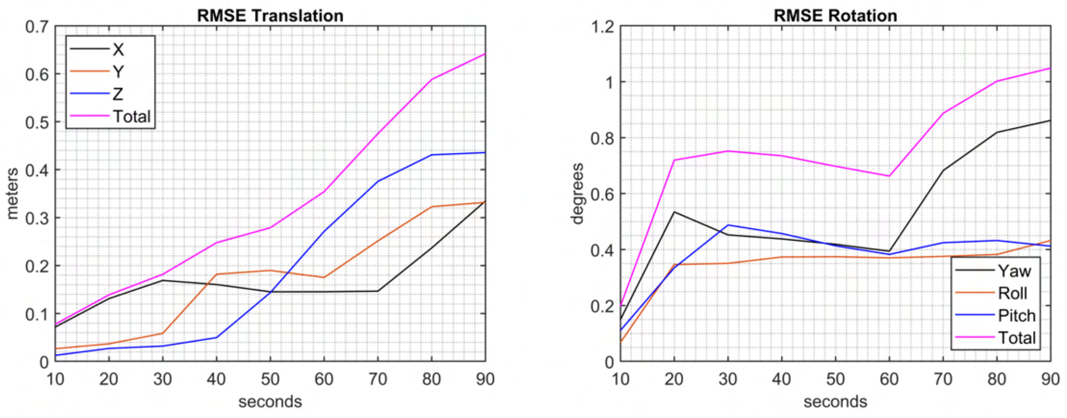


Figure 4.23: Translation and rotation RMS errors comparison, 4 moving cars.

Table 4.10: Quantities of interest, 4 moving cars.

Distance covered	$270\,m$
Duration	$90\,s$
Constant speed	$3\,m/s$
Dynamic objects	4
Maximum translation error	$1.3480\,m$
Maximum rotation error	$3.6683\,deg$
Final translation error	$0.3205\,m$
Final rotation error	$0.6987\,deg$
RMS final translation error	$0.6419\,m$
RMS final translation error in % of the distance covered	0.2377%
RMS final rotation error	$1.0487\,deg$

2) Run 2

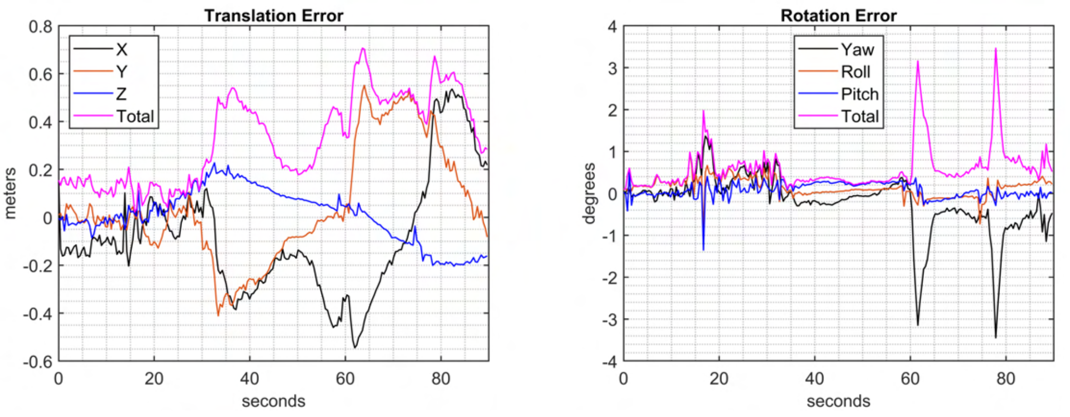


Figure 4.24: Translation and rotation errors, 7 moving cars.

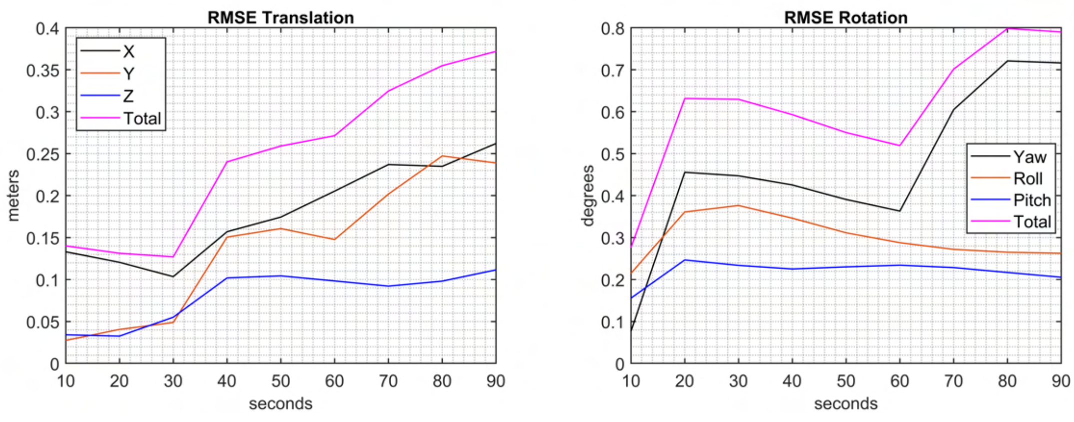


Figure 4.25: Translation and rotation RMS errors comparison, 7 moving cars.

Table 4.11: Quantities of interest, 7 moving cars.

Distance covered	270 m
Duration	90 s
Constant speed	3 m/s
Dynamic objects	7
Maximum translation error	0.7064 m
Maximum rotation error	3.4638 deg
Final translation error	0.2837 m
Final rotation error	0.5174 deg
RMS final translation error	0.3718 m
RMS final translation error in % of the distance covered	0.1377%
RMS final rotation error	0.7900 deg

3) Run 3

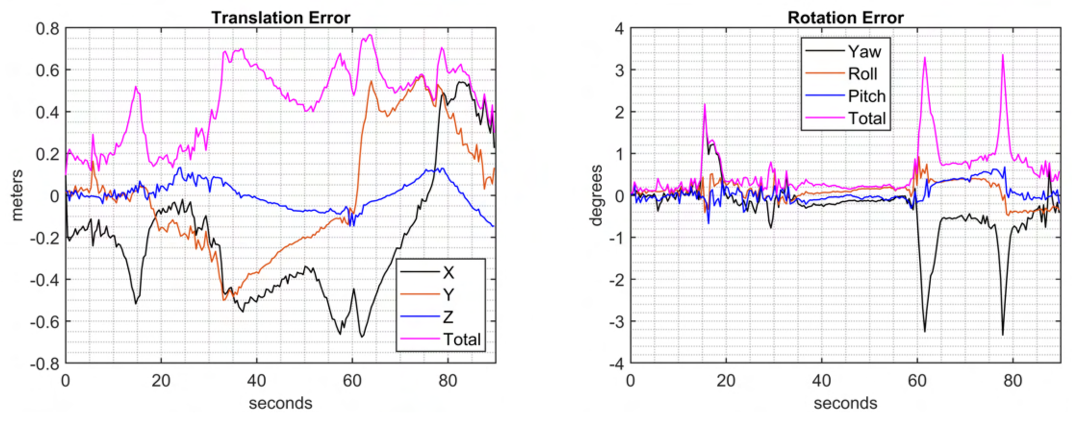


Figure 4.26: Translation and rotation errors, 10 moving cars.

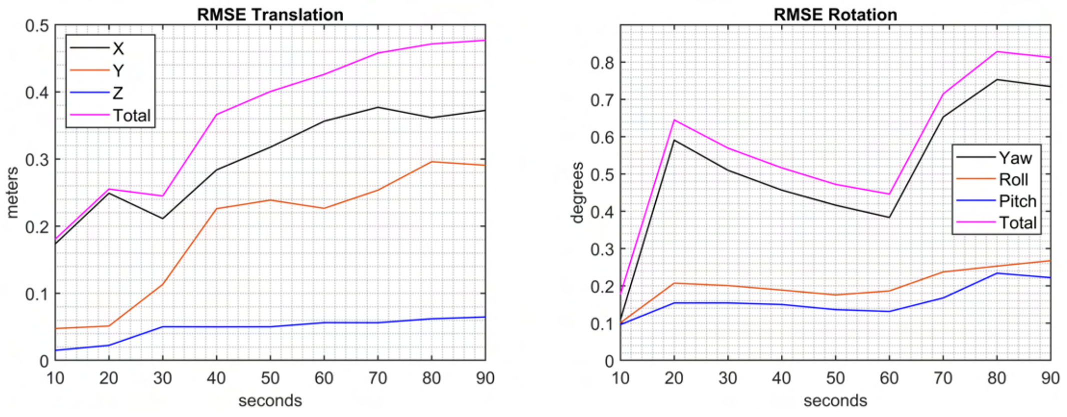


Figure 4.27: Translation and rotation RMS errors comparison, 10 moving cars.

Table 4.12: Quantities of interest, 10 moving cars.

Distance covered	$270\,m$
Duration	$90\,s$
Constant speed	$3\,m/s$
Dynamic objects	10
Maximum translation error	$0.7671\,m$
Maximum rotation error	$3.3592\,deg$
Final translation error	$0.2996\,m$
Final rotation error	$0.5746\,deg$
RMS final translation error	$0.4769\,m$
RMS final translation error in % of the distance covered	0.1766%
RMS final rotation error	$0.8125\,deg$

4) Run 4

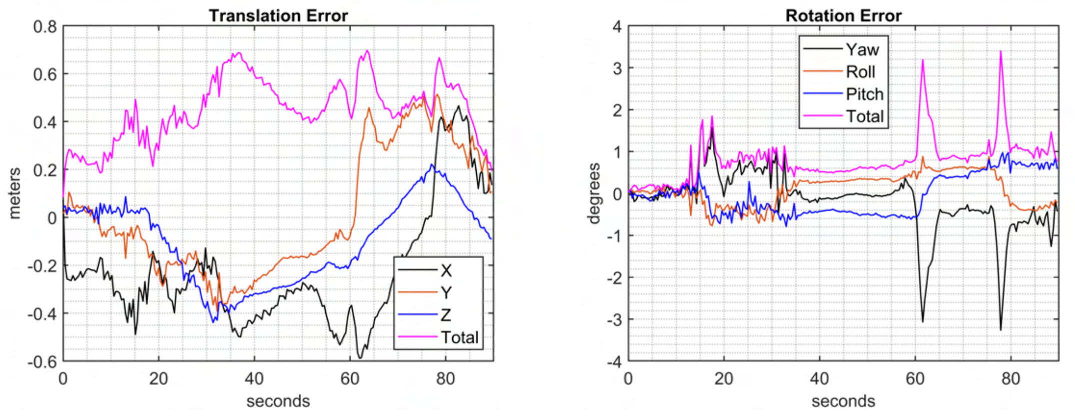


Figure 4.28: Translation and rotation errors, 14 moving cars.

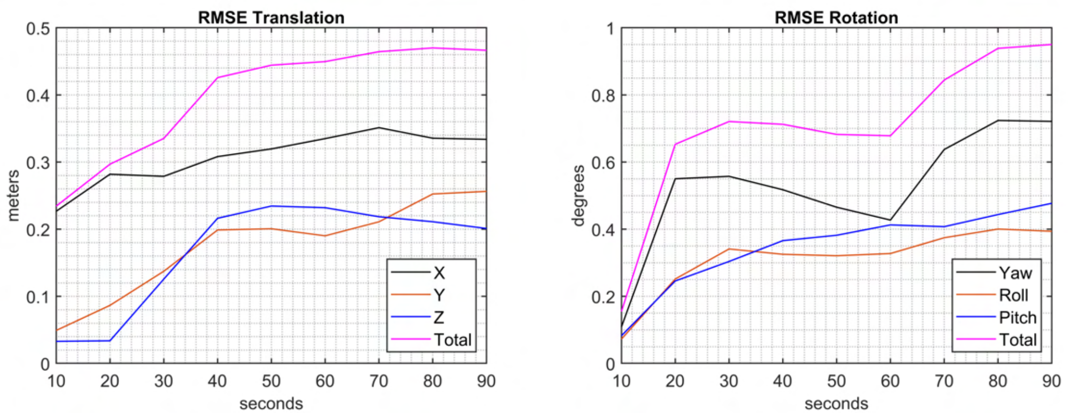


Figure 4.29: Translation and rotation RMS errors comparison, 14 moving cars.

Table 4.13: Quantities of interest, 14 moving cars.

Distance covered	270 m
Duration	90 s
Constant speed	3 m/s
Dynamic objects	14
Maximum translation error	0.6973 m
Maximum rotation error	3.4007 deg
Final translation error	0.1933 m
Final rotation error	0.7578 deg
RMS final translation error	0.4664 m
RMS final translation error in % of the distance covered	0.1727%
RMS final rotation error	0.9502 deg

Considerations on the previous tests

The results obtained from these tests were partly surprising. As can be appreciated from Figure 4.30 and 4.31 and Table 4.14, the introduction of moving objects had an unpredictable impact on the overall pose estimation. As expected, with 4 moving vehicles the estimation worsens from standard condition, however the introduction of other cars in the subsequent runs significantly improve the effectiveness of the algorithm. Even changing the order with which the cars are added, or changing their trajectories, the results are similar (for example, substituting Car 2.1 with Car 1.1 in the first run still leads to poor results, with a translation error up to 1.5 m). A justification for this behavior can be found by considering how key frames are recognized. The number and IDs of the key frames may slightly change depending on the objects that are seen. This leads to a ripple effect due to bundle adjustments and pose graph optimization that also involves frames distant from those that have changed. The improvement in the estimation of some path sections can thus be associated with more key frames being captured there, which would not be taken without moving vehicles. These considerations are supported in particular by two facts: 1) for different tests, different frames were recognized for loop closure, despite the fact that no changes were made in the initial (and thus final) stretch in any of the cases; 2) the rotation error is overall worse in the standard condition, presumably because the added key frames are in correspondence with the turns, where more dynamic objects are seen. In light of this outcome, it becomes difficult if not impossible to predict whether, when and how the presence of dynamic objects affect pose estimation. It transpires from this test that significantly different, but not necessarily bigger errors are obtained as the number of dynamic objects is increased.

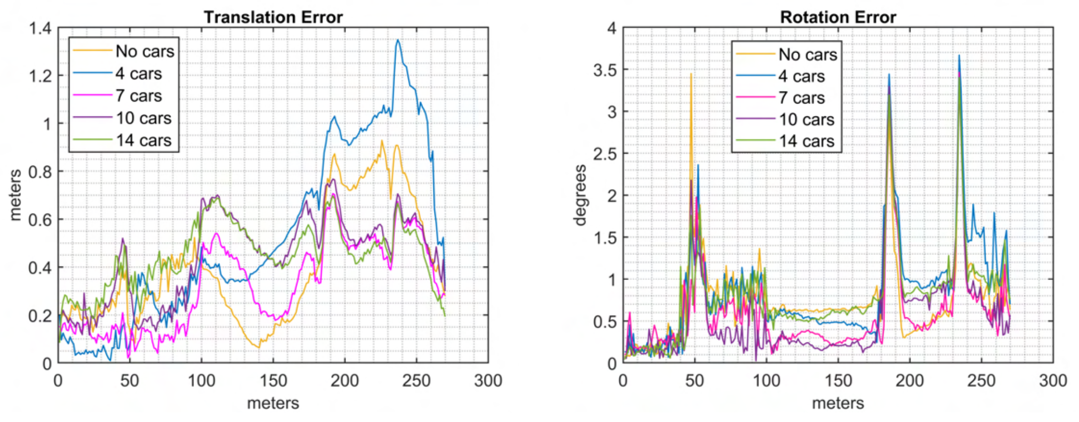


Figure 4.30: Translation and rotation errors comparison.

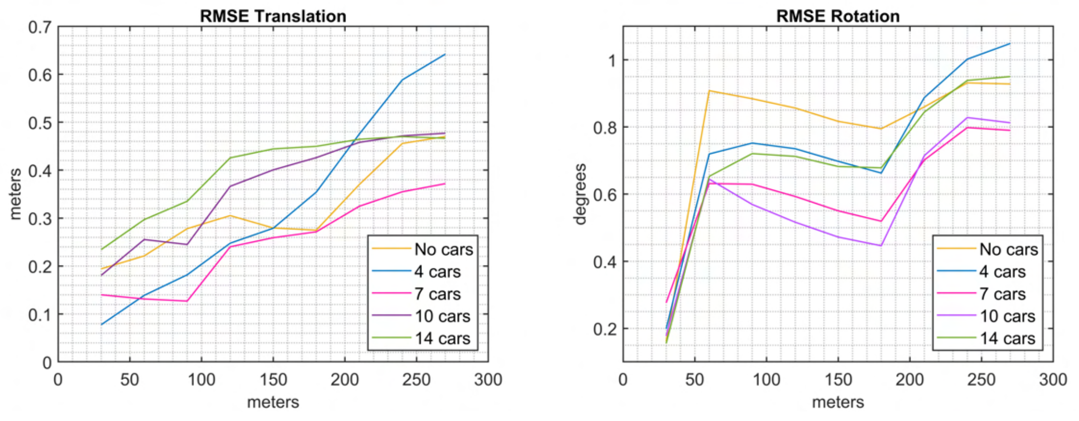


Figure 4.31: Translation and rotation RMS errors comparison.

Table 4.14: Comparison of the results.

	0 cars	4 cars	7 cars	10 cars	14 cars
Distance covered	$270\,m$	$270\,m$	$270\,m$	$270\,m$	$270\,m$
Duration	$90\,s$	$90\,s$	$90\,s$	$90\,s$	$90\,s$
Speed	$3\,m/s$	$3\,m/s$	$3\,m/s$	$3\,m/s$	$3\,m/s$
Maximum translation error	$0.9286\,m$	$1.3480\,m$	$0.7064\,m$	$0.7671\,m$	$0.6973\,m$
Maximum rotation error	$3.5624\,deg$	$3.6683\,deg$	$3.4678\,deg$	$3.3592\,deg$	$3.4007\,deg$
Final translation error	$0.2789\,m$	$0.3205\,m$	$0.2837\,m$	$0.2996\,m$	$0.1933\,m$
Final rotation error	$0.6319\,deg$	$0.6988\,deg$	$0.5175\,deg$	$0.5746\,deg$	$0.7578\,deg$
RMS final translation error	$0.4701\,m$	$0.6419\,m$	$0.3718\,m$	$0.4769\,m$	$0.4701\,m$
RMS final translation error in % of the distance covered	$0.1741\,\%$	$0.2377\,\%$	$0.1377\,\%$	$0.1766\,\%$	$0.1741\,\%$
RMS final rotation error	$0.9280\,deg$	$1.0487\,deg$	$0.7840\,deg$	$0.8125\,deg$	$0.9502\,deg$
Loop Closure frames	$4, 5, 7 - 221$	$4, 6, 7 - 223$	$3, 4, 5 - 221$	$5, 6, 7 - 224$	$4, 6, 7 - 224$

5) Vehicle tracking

During the *KITTI* sequence there is a section of road in which the recording platform follows at close range a car ahead of it. This situation was reproduced in the virtual environment by synchronizing the motion of the recording vehicle with that of an upcoming car. At the second turn, the cameras start to follow a car that is proceeding in a straight line at a constant speed of $3\,m/s$ and stay behind it for about 30 seconds, up until the next turn. The results of this test are presented in the graphs below.

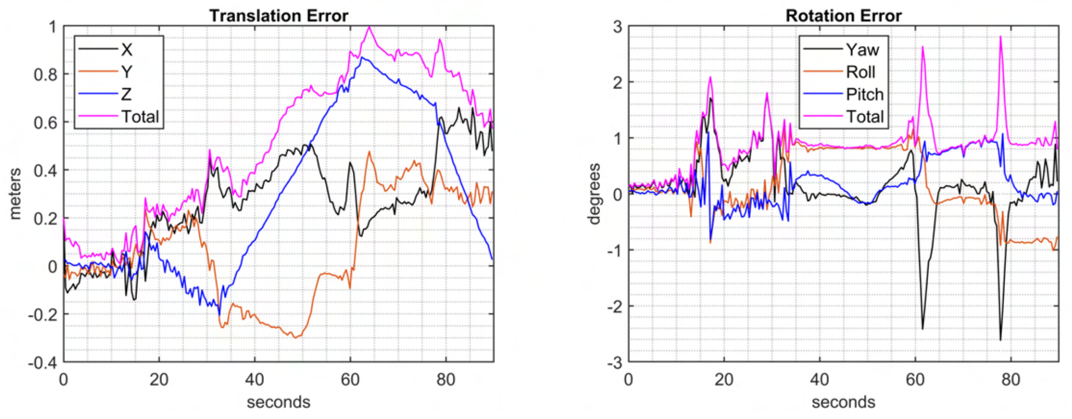


Figure 4.32: Translation and rotation errors, vehicle tracking.

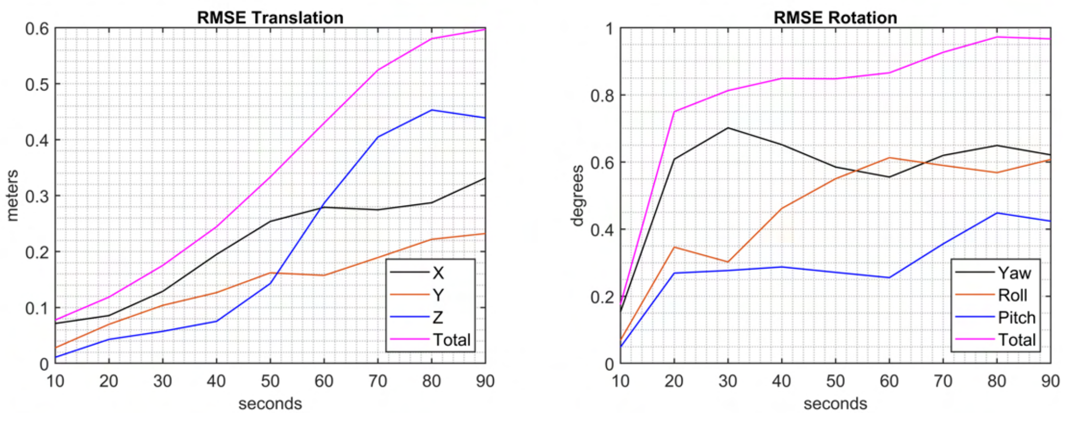


Figure 4.33: Translation and rotation RMS errors comparison, vehicle tracking.

The comparison with the Standard Scenario is instead shown in Figures 4.34 and 4.35 and Table 4.15. It is evident how from the moment in which the vehicle is framed the translation error becomes significantly larger. Here, too, optimization operations on connected key frames make the overall error trend unpredictable: in the first stretch, while there is no difference between the two cases, the optimized trajectory gives here a better estimate.
As for rotations, overall the error here is greater, but the peaks are less pronounced. Again, at the first turn where no moving vehicle is seen in either case, the estimate is much better from this simulation.

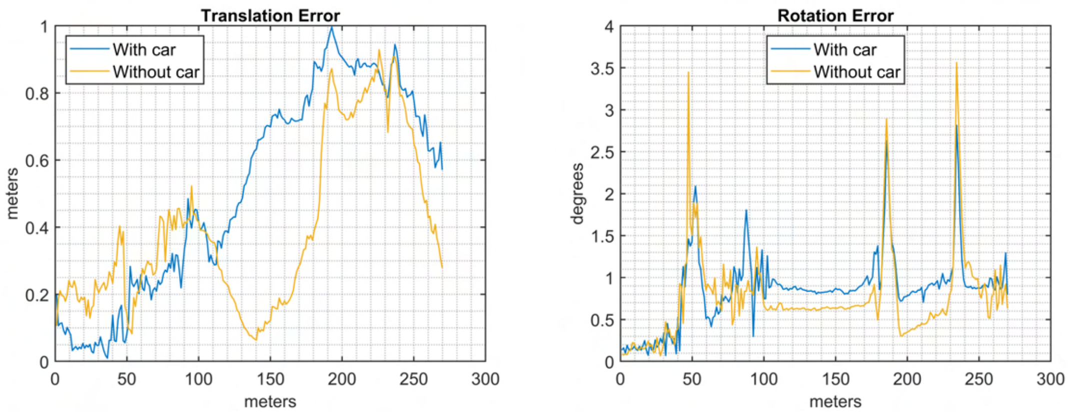


Figure 4.34: Translation and rotation errors comparison.

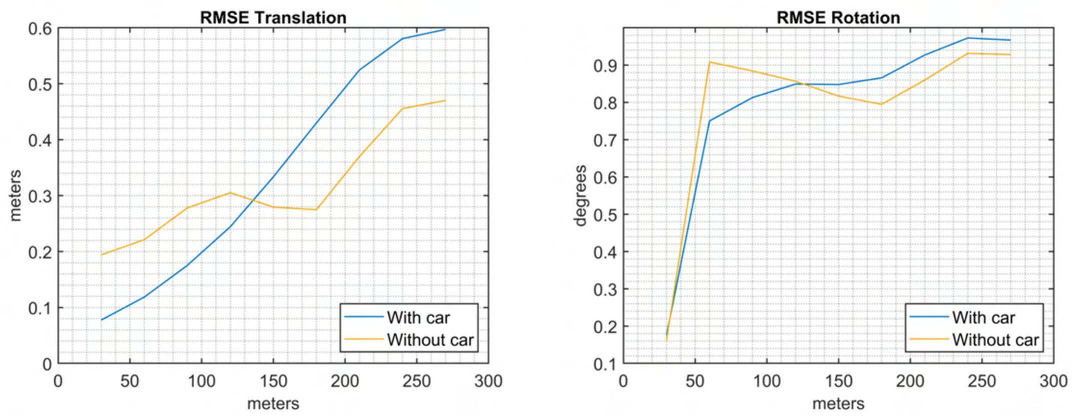


Figure 4.35: Translation and rotation RMS errors comparison.

Table 4.15: Comparison of the results, with and without tracked car.

	Without car	With car
Distance covered	$270\,m$	$270\,m$
Duration	$90\,s$	$90\,s$
Dynamic objects	0	1
Maximum translation error	$0.9286\,m$	$0.9961\,m$
Maximum rotation error	$0.5624\,deg$	$2.8143\,deg$
Final translation error	$0.2789\,m$	$0.5706\,m$
Final rotation error	$0.6319\,deg$	$0.7940\,deg$
RMS final translation error	$0.4701\,m$	$0.5971\,m$
RMS final translation error in % of the distance covered	$0.1741\,\%$	$0.2211\,\%$
RMS final rotation error	$0.9280\,deg$	$0.9669\,deg$

6) Vehicles at an Intersection

Another feature seen during the *KITTI* sequence concern the scenario in which a car stops at an intersection and sees a series of vehicles passing in front of it. To evaluate the impact of these vehicles on the pose drift, a test was prepared by keeping the cameras fixed while framing the intersection for a duration of 10 seconds. In the first case there are no dynamic objects, while in the second case 5 vehicles (4 pickups of different colors and 1 white box truck) moving in both directions and at variable speed (from $3\,m/s$ to $10\,m/s$) are added. In this case, the loop closure search is not performed since the vehicle remains stationary, so only the estimated pose was considered. Nevertheless, the results are indicative of what happens also within a closed pathway (as in KITTI) since the optimized trajectory is created starting from the estimated one. Another peculiar aspect of this test is that the operation of local key frame culling is highlighted. In fact, for the sequence in which there are no moving objects, only 4 key frames in total are retained, while in the other sequence 19 key frames are kept. The results are summarized in Fig. 4.36 and Table 4.16.

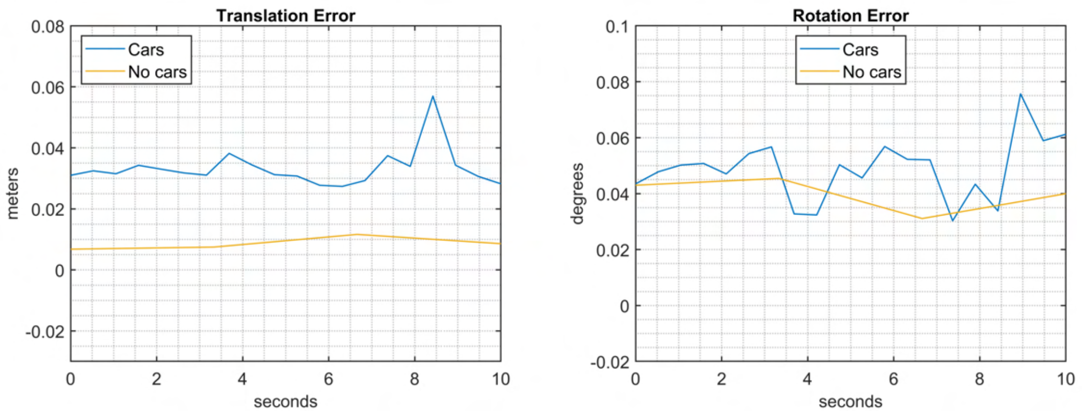


Figure 4.36: Translation and rotation errors comparison, vehicles at an intersection.

Table 4.16: Comparison of the results.

	Without cars	With cars
Distance covered	$0\,m$	$0\,m$
Duration	$10\,s$	$10\,s$
Dynamic objects	0	5
Maximum translation error	$0.0116\,m$	$0.0569\,m$
Maximum rotation error	$0.0454\,deg$	$0.0757\,deg$
Final translation error	$0.0086\,m$	$0.0282\,m$
Final rotation error	$0.0400\,deg$	$0.0612\,deg$

Overall, the drift remained rather low, despite the high volume of dynamic objects within the scene. In particular, the major contribution in the translational drift is given by Y-axis component. The peak between seconds 8 and 9 is due to a highly congested moment, as seen in Figure 4.37. These results are in agreement with those of the dataset sequence, in which was assessed the good behavior of the algorithm in these circumstances.

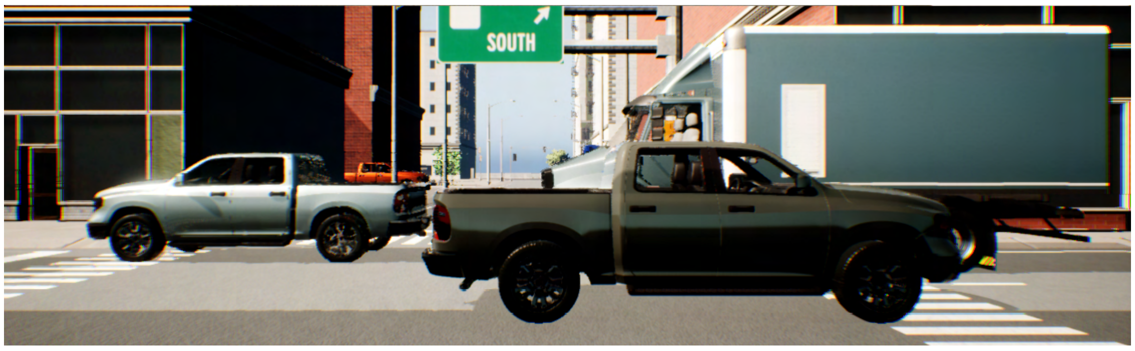


Figure 4.37: Left frame captured during the simulation.

4.2.4 Impact of lighting conditions on SLAM system performance

As previously seen in Section 3.4.1, appropriately modeling lighting conditions in *Unreal Engine* is a complex operation that relies on numerous parameters. To study the behavior of the system under different conditions, it was chosen to act only on the intensity level of the *directional light* that simulates the Sun. In this way is affected only how much light is fed into the environment (Figure 4.38). Five runs explore an incremental value of light intensity, from 0 to 100 lux (the Standard Condition have been obtained with 50 lux). Observe how even at 0 lux, there is still light illuminating the scene. This comes from the *SkyLight* component that controls a diffuse light coming from the sky. As will be seen on the other hand, the performance of the algorithm degrades significantly even without further reducing visibility, so it was not deemed necessary to further darken the scene.

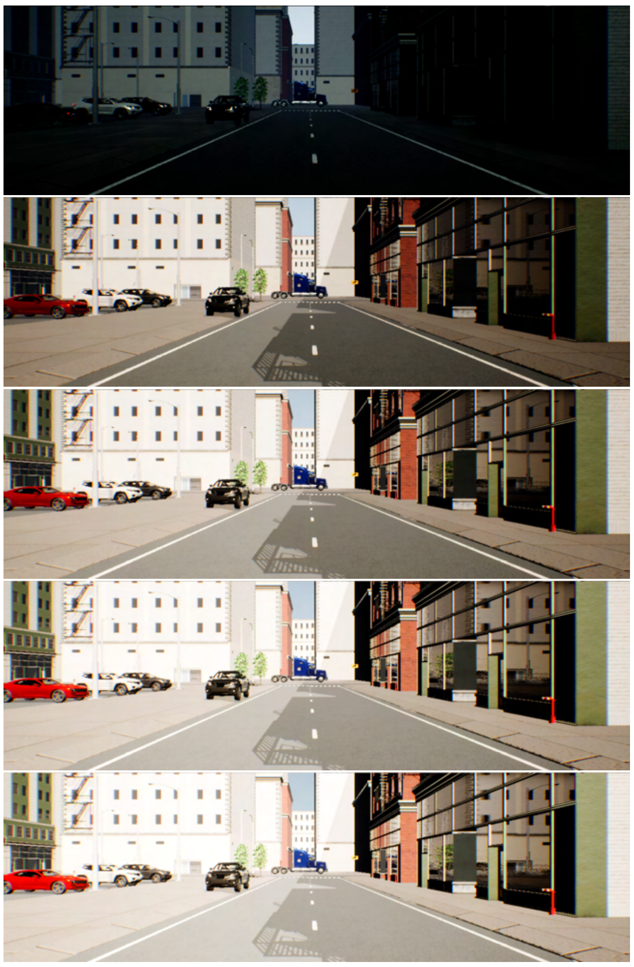


Figure 4.38: Comparison of the lighting, from the top left: 0, 25, 50, 75, 100 lux.

1) 0 lux

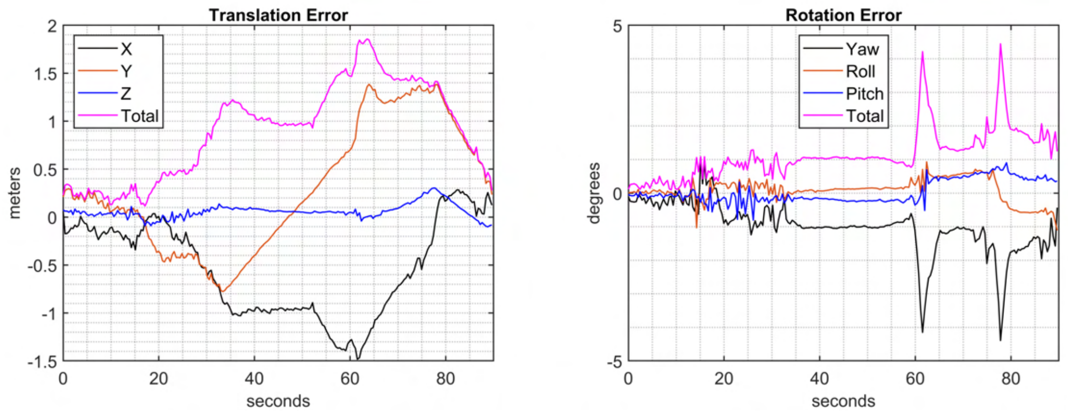


Figure 4.39: Translation and rotation errors, 0 lux.

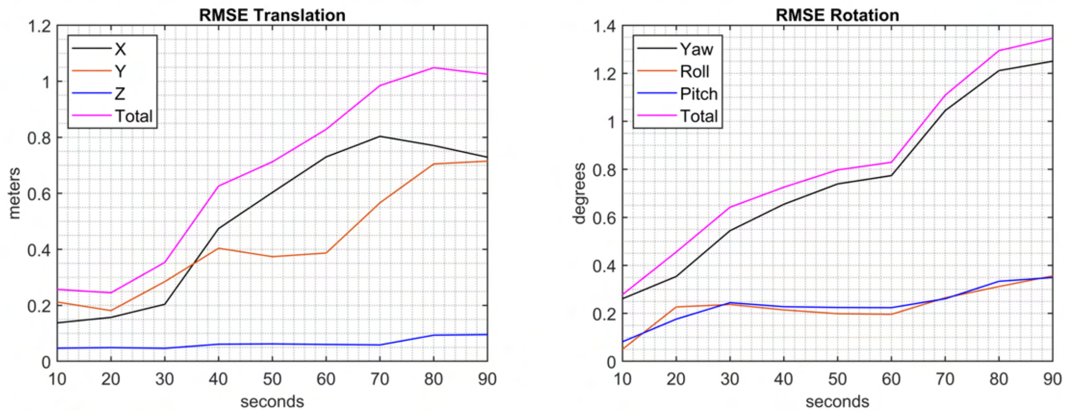


Figure 4.40: Translation and rotation RMS errors, 0 lux.

Table 4.17: Quantities of interest, 0 lux.

Maximum translation error	1.8569 m
Maximum rotation error	4.4471 deg
Final translation error	0.2777 m
Final rotation error	1.2442 deg
RMS final translation error	1.0254 m
RMS final translation error in % of the distance covered	0.3798%
RMS final rotation error	1.3463 deg

2) 25 **lux**

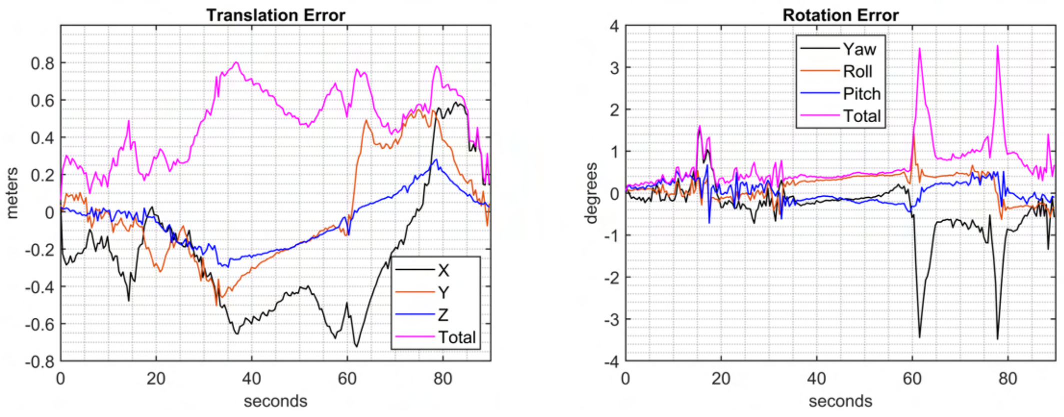


Figure 4.41: Translation and rotation errors, 25 lux.

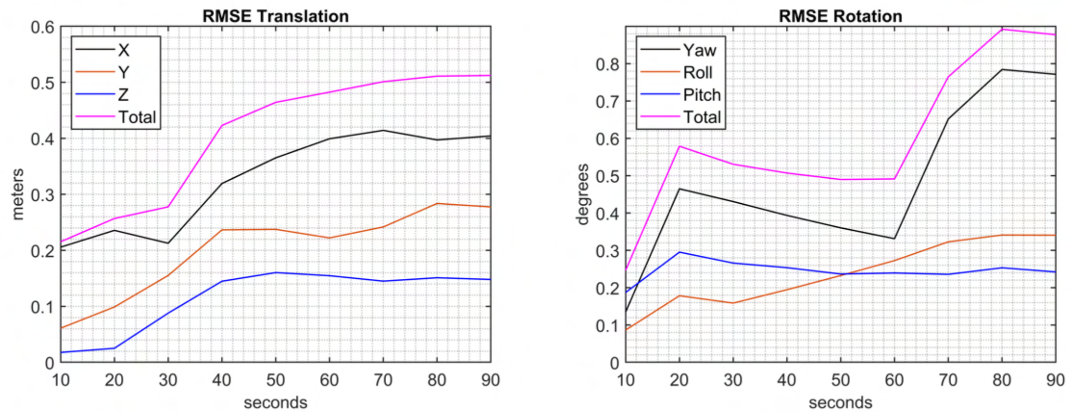


Figure 4.42: Translation and rotation RMS errors, 25 lux.

Table 4.18: Quantities of interest, 25 lux.

Maximum translation error	0.8026 m
Maximum rotation error	3.5182 deg
Final translation error	0.1509 m
Final rotation error	0.6711 deg
RMS final translation error	0.5122 m
RMS final translation error in % of the distance covered	0.1897%
RMS final rotation error	0.8774 deg

3) 75 **lux**

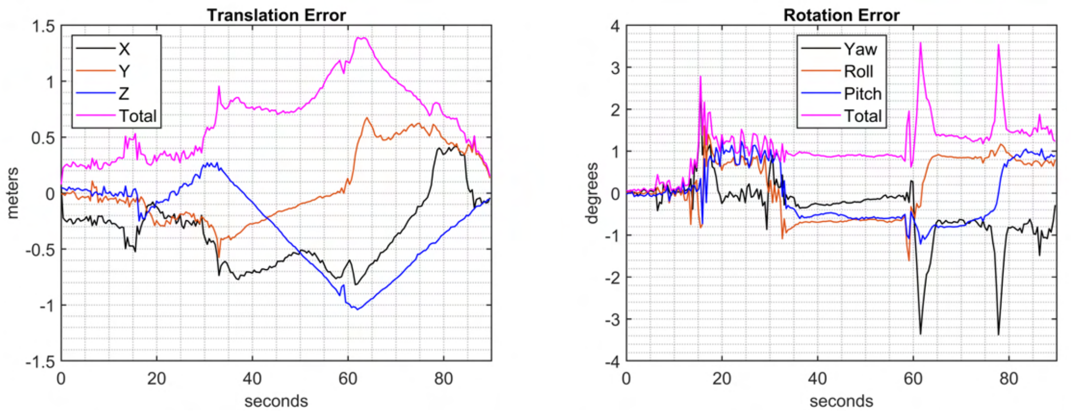


Figure 4.43: Translation and rotation errors, 75 lux.

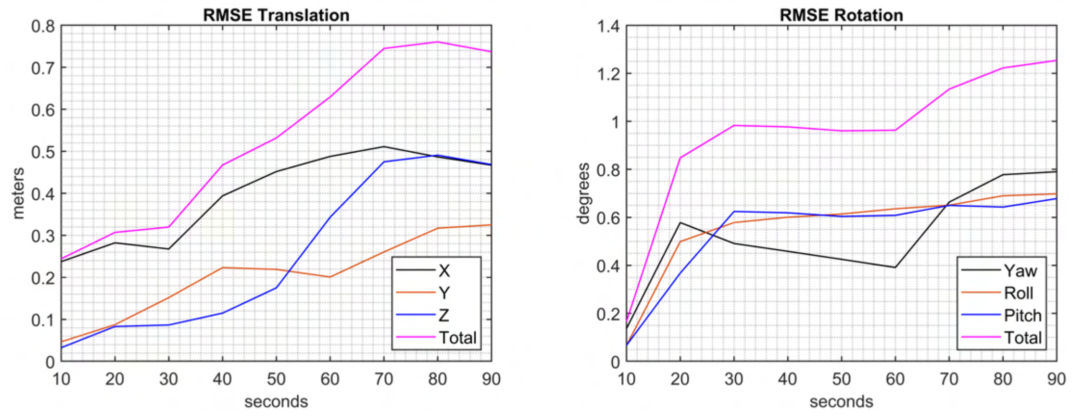


Figure 4.44: Translation and rotation RMS errors, 75 lux.

Table 4.19: Quantities of interest, 75 lux.

Maximum translation error	1.3900 m
Maximum rotation error	3.5860 deg
Final translation error	0.1515 m
Final rotation error	1.2315 deg
RMS final translation error	0.7371 m
RMS final translation error in % of the distance covered	0.2730%
RMS final rotation error	1.2537 deg

4) 100 lux

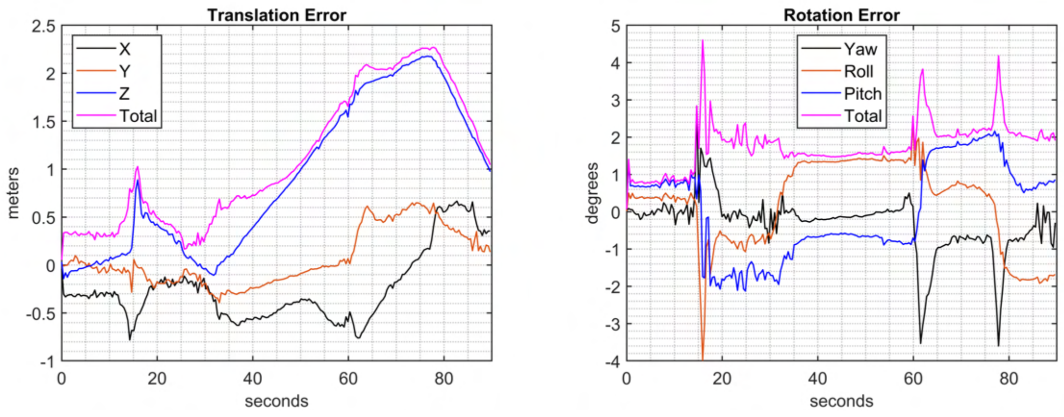


Figure 4.45: Translation and rotation errors, 100 lux.

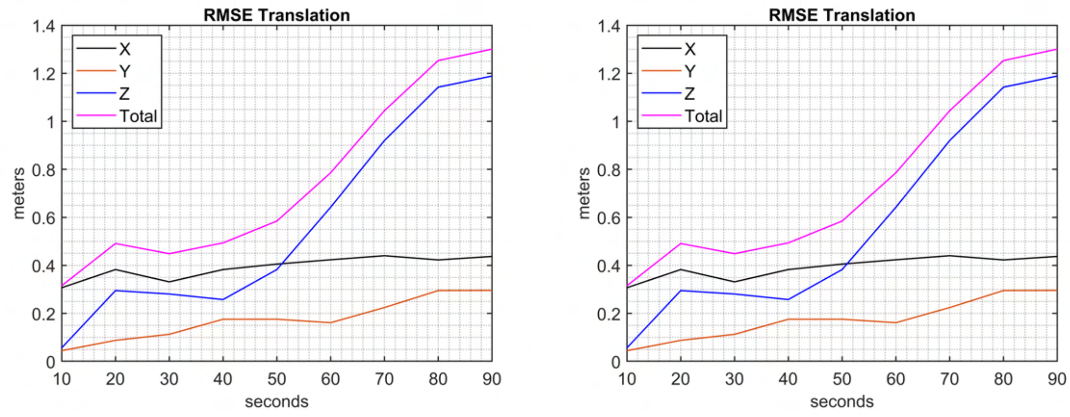


Figure 4.46: Translation and rotation RMS errors, 100 lux.

Table 4.20: Quantities of interest, 100 lux.

Maximum translation error	2.2702 m
Maximum rotation error	4.6038 deg
Final translation error	1.0456 m
Final rotation error	1.9098 deg
RMS final translation error	1.3007 m
RMS final translation error in % of the distance covered	0.4817%
RMS final rotation error	1.8921 deg

Considerations on the previous tests

The amount of light illuminating the scene has an important impact on the detection of valid ORB features. The considerations made in the case of dynamic vehicles regarding the key frames that are chosen in performing loop closure remain valid. Indeed, it can be seen from Table 4.21 that different correspondences are recognized for each run here as well. When the environment is dimly lit, the shadowy areas do not allow reference points to be identified, on the other hand, an excess of light overexposes the image. In both cases, there is then a loss of detail in the image, which affects the functioning of the system. As expected, the worst cases are for the two extremes at 100 and 0 lux, while the sweet spot appears to be between 25 and 50 lux.

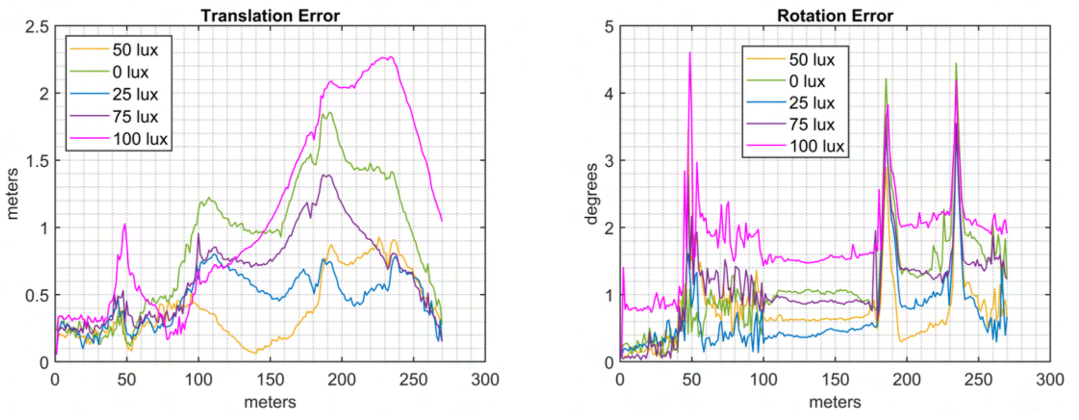


Figure 4.47: Translation and rotation errors comparison.

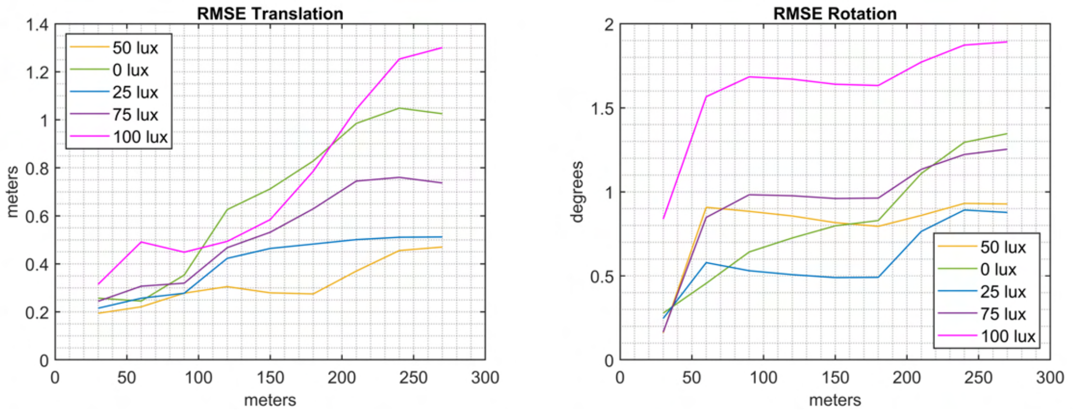


Figure 4.48: Translation and rotation RMS errors comparison.

Table 4.21: Comparison of the results.

	0 lux	25 lux	50 lux	75 lux	100 lux
Distance covered	$270\,m$	$270\,m$	$270\,m$	$270\,m$	$270\,m$
Duration	$90\,s$	$90\,s$	$90\,s$	$90\,s$	$90\,s$
Speed	$3\,m/s$	$3\,m/s$	$3\,m/s$	$3\,m/s$	$3\,m/s$
Maximum translation error	$1.8569\,m$	$0.8026\,m$	$0.9286\,m$	$1.3900\,m$	$2.2702\,m$
Maximum rotation error	$4.4471\,deg$	$3.5182\,deg$	$3.5624\,deg$	$3.5860\,deg$	$4.6038\,deg$
Final translation error	$0.2777\,m$	$0.1515\,m$	$0.2789\,m$	$0.1515\,m$	$1.0456\,m$
Final rotation error	$1.2442\,deg$	$0.6711\,deg$	$0.6319\,deg$	$1.2315\,deg$	$1.9098\,deg$
RMS final translation error	$1.0254\,m$	$0.5122\,m$	$0.4701\,m$	$0.7371\,m$	$1.3007\,m$
RMS final translation error in % of the distance covered	$0.3800\,\%$	$0.1900\,\%$	$0.1741\,\%$	$0.2730\,\%$	$0.4817\,\%$
RMS final rotation error	$1.3463\,deg$	$0.8774\,deg$	$0.9280\,deg$	$1.2537\,deg$	$1.8921\,deg$
Loop Closure frames	$4, 6, 7 - 224$	$1, 2, 3, -221$	$4, 5, 7 - 221$	$2, 4, 5 - 223$	$1, 2, 4 - 223$

4.3 Scenario Modeled after KITTI's Sequence 07

The last test presented is aimed at repeating what occurs during KITTI's Sequence 07. As anticipated, compromises had to be made in reproducing a scenario comparable to that seen in the dataset. Due to the conformation of the source *US City Block 3D Environment* map, a slightly shorter route was plotted than the reference one. A distance of 600 meters is covered in 95 seconds by following the path shown in Figure 4.50. The height of the cameras has been set at $1.65\,m$ above the ground and is kept constant throughout the route. Similarly, the pitch and roll angles were set to zero. The car therefore only moves in the XY plane by varying the yaw angle. Dynamic vehicles were also arranged in numbers and modes comparable to those in the dataset. 5 cars and 1 box truck pass at a speed of $11\,m/s$ in front of the recording platform while it is stationary at an intersection. Another car is followed up close in a straight stretch for about $90\,m$ before it moves out of the way. A light intensity of 50 lux was deemed appropriate (please go back to Section 3.4.1 for more details about how the lighting condition were modeled).

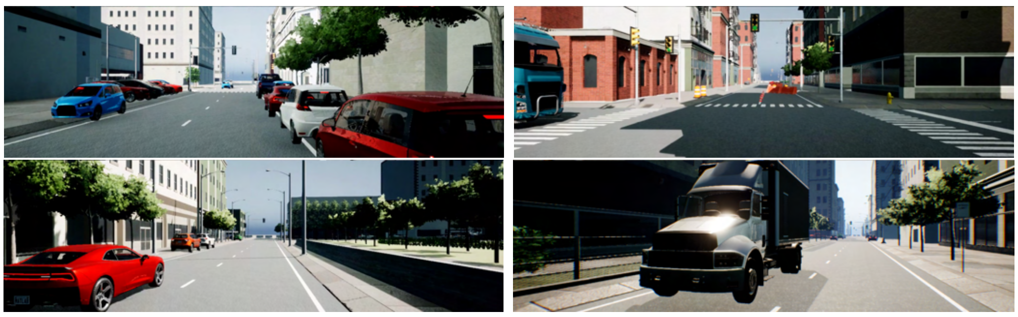


Figure 4.49: Some images captured by the right camera in the scenario.

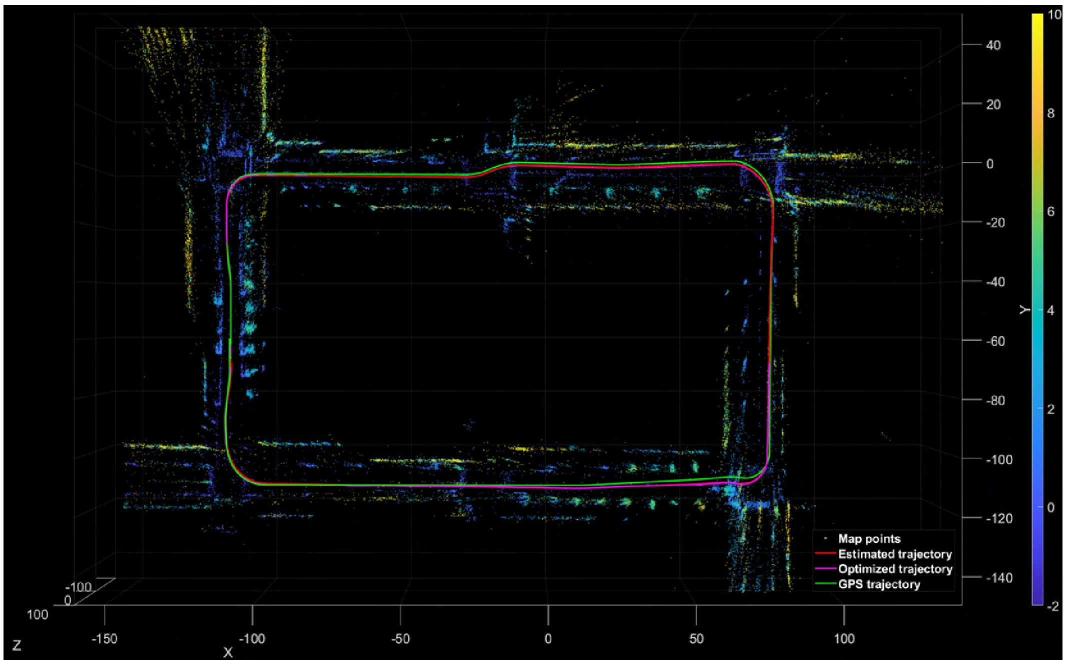


Figure 4.50: Virtual scene path as seen in the Point Cloud Player.

The initial pose is defined by the *Player Start* actor, which can be arbitrary set anywhere within the scenario. In this case:

```
Starting_Position = [-112.2138, -72.6462, 1.65];
Starting_Rotation = [1.5708, 0, 0];
```

and the route is traveled in a clockwise direction. The ORB-SLAM2 parameters are the same as those listed in Section 4.2. Some frames captured during the simulation can be seen in Figure 4.49.

Table 4.22: Scenario characteristics comparison.

	Virtual Scene	Sequence 07
Distance covered	600 m	686 m
Duration	95 s	110.5 s
Average speed	6.3 m/s	6.2 m/s
Maximum speed	10 m/s	11.6 m/s
Dynamic objects	7	7

The trajectory was plotted within *UE* using the method described in Section 3.8.2, and was applied a variable speed from 0 m/s up to 10 m/s for the vehicle (Fig. 4.51). The maximum speed is 1.6 m/s lower than that in KITTI, however, it was deemed of little influence since it is reached for a very short amount of time to then drop again around 10 m/s. The average speed of 6.3 m/s is instead very close to the one sought. Notice how the speed variation is stepped because of the choice made in the input command. In fact, the mouse wheel is used to accelerate/decelarate, and each step is due to the repositioning of the finger during the operation.

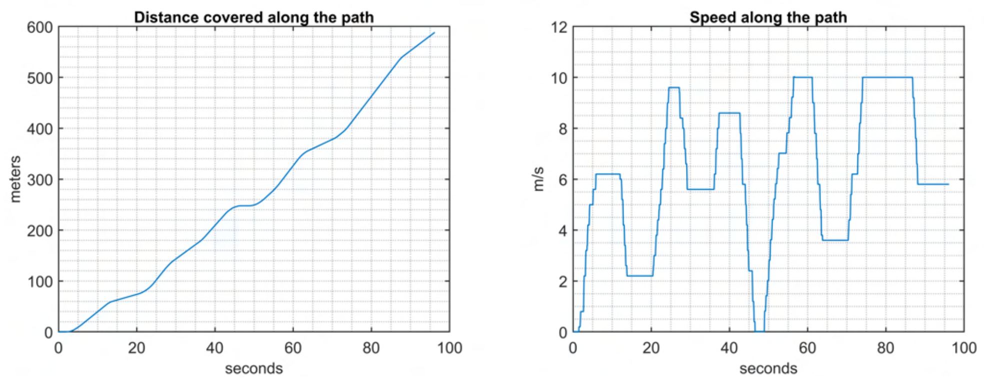


Figure 4.51: Distance covered and speed during the simulation.

The results of this simulation are presented in Fig. 4.52, Fig. 4.53 and Table 4.23.

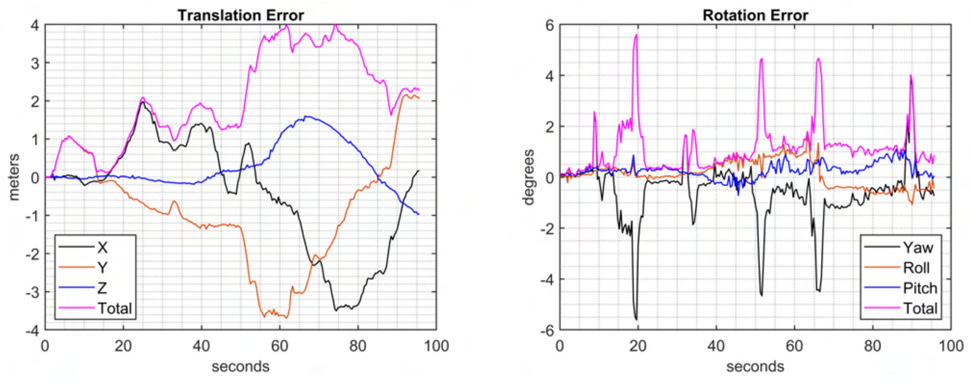


Figure 4.52: Translation and rotation errors.

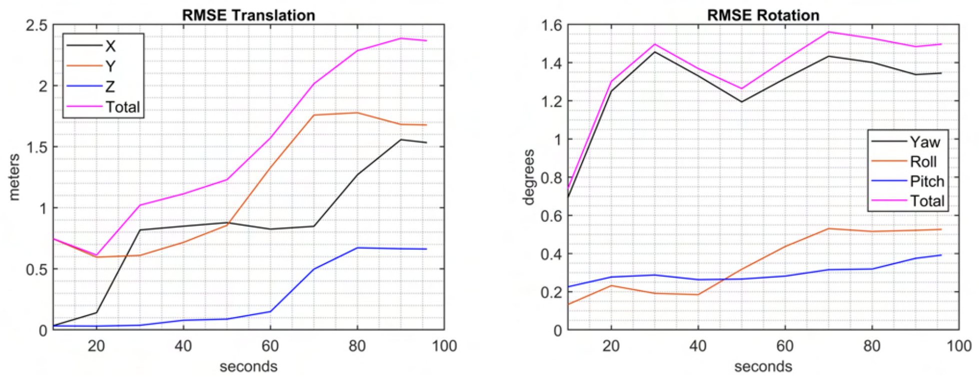


Figure 4.53: Translation and rotation RMS errors.

The variation of the error with respect to the speed is as expected. In general, when the speed increase the translation error also grows, while the turns cause peaks in the rotation error (Fig. 4.54). Between seconds 46 and 49, when the car is not moving, the estimation has minimum drift.

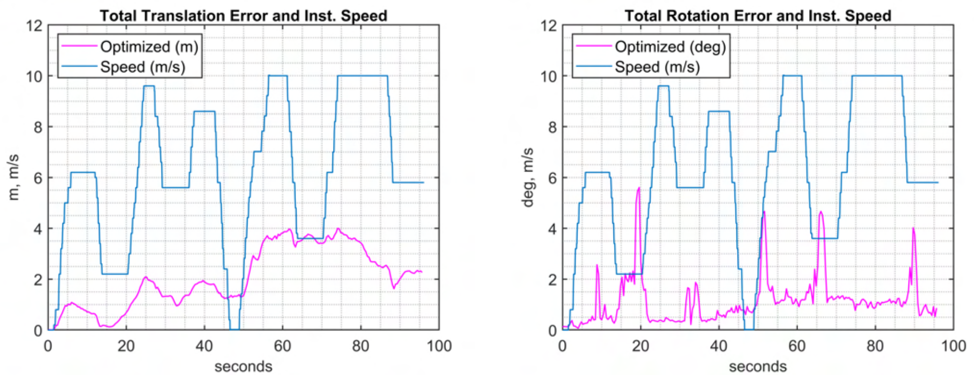


Figure 4.54: Translation and rotation errors shown with the instantaneous speed.

Results comparison with the *KITTI* sequence

The virtual simulation provided results partly in agreement with those found in the dataset sequence (Table 4.23).

Table 4.23: Results for the Virtual Scene and Sequence 07.

	Virtual Scene	Sequence 07
Maximum translation error	3.9997 m	4.4261 m
Maximum rotation error	5.6095 deg	6.1951 deg
Final translation error	2.2679 m	3.4797 m
Final rotation error	0.8636 deg	1.1306 deg
RMS final translation error	2.3670 m	3.1684 m
RMS final translation error in % of the distance covered	0.3945 %	0.4619 %
RMS final rotation error	1.4966 deg	2.0053 deg

The differences in the errors can be attributed to some feature that has not been reproduced faithfully enough, but also to the simple fact that the environments still have discrepancies that are impossible to fill (assets, trajectory, lighting conditions, *etc.*). In addition, the difference in path length may also have contributed to a slight underestimate of the error. As has been shown in previous tests, even small variations can affect the entire estimate, so it was unlikely to find precise matches between the two cases. The most relevant difference concerns the final error on the translation. Comparing the graphs it can be seen that in the dataset the error fails to effectively recover thanks to the loop closure. In the virtual simulation instead, despite having the same increasing trend (notice that the maximum translation errors are not that far apart), at a certain point the error starts to drop as the repetition of the ORB features between the key frames at the beginning and at the end of the sequence is recognized more accurately. It is important to note that this is not a peculiarity of the virtual environment because other tests in similar scenes (but with characteristics judged more distant from those sought) have not closed the trajectory so well.

The behaviour on the rotations is considered well replicated to some extent since the magnitude of the maximum and final error is in line with expectations. This is assumed to be due to the fact that they are shorter phenomena in which the rotational speed, correctly reproduced, is the most decisive variable. In the sequence in the virtual environment however, all the sharp turns see a peak that increases the error to at least 4 degrees, while in the dataset sequence this phenomena is seen only in the last two turns. On the other hand, the KITTI sequence has an overall higher mean error. The precise reason for this behavior has not been identified, but it is presumed to be related to the fact that in the real environment there are much more diverse characteristics from one turn to another in comparison to those simulated. What's more, KITTI's car makes multiple small changes in direction, so this can actually cause drift buildup in stretches that I modeled as straights.
The sensors model in *Simulink* is obviously also the cause of mismatches between the two cases. Moreover, despite the efforts in introducing optical artifacts in the simulation images, there are certainly unresolved differences in this respect.
Finally, even the functioning of the algorithm itself influences the outcome of the tests. It has been said that there has been a lot of research and experimentation on my part to find parameter values that could offer good results in both environments. This does not exclude that the virtual scene gives overall a better estimation also because this particular configuration is here more suitable.

Chapter 5

Conclusions and Future Developments

This final chapter draws conclusions about the work presented and evaluates possible future developments and improvements.
Chapter 1 stated the three goals that this thesis set out:

1. Present a detailed analysis and description of the *Matlab* implementation of the ORB-SLAM2 system;
2. Implement a comprehensive and self-sufficient framework for performing SLAM testing in a virtual environment, with possibilities for great customization of the scene, sensors and trajectory;
3. Demonstrate the validity of a virtual environment as an alternative to a dataset for testing purposes.

Each of these points is taken up below to make the appropriate considerations.

5.1 *Matlab* implementation of the ORB-SLAM2 system

To my knowledge, the ORB-SLAM2 implementation described and used in this thesis is to date the only one available in *Matlab*. It has some differences from the one originally proposed by Raùl Mur-Artal and Juan D. Tardòs [12], for this reason it was not possible to use their results to verify that I had correctly applied the system. For example, the global bundle adjustment that takes place at the end of the

original implementation certainly improves the estimate over the one I obtained. The verification of the correct application comes from the fact that the behavior of the system and the order of magnitude of errors is in line with those obtained by running the examples provided by *Mathworks*. In addition, a short conversation with one of the authors of the *Matlab* version confirmed the correctness with which I chose the parameters to change in order to adapt the algorithm to new case studies. As disclosed by the authors and experienced in person during the course of this work, there are still optimization and stability issues that they are actively working on. It is then of primary interest to see how the trajectory estimation improves with future versions of this implementation, and also how much more stable the system becomes as parameter values change. Moreover, it would be interesting to make a comparison with the more recent ORB-SLAM3, if it ever gets implemented in *Matlab*.
While it is therefore expected that ORB-SLAM2 is capable of providing a overall better estimate than that shown, the purpose of this thesis was not affected by this fact. Indeed, the same implementation and parameters are used with both dataset images and those derived from the virtual environment.
The documentation consulted to write this thesis was found to lack a complete description, from beginning to end, of a SLAM system that could also be fully understood by those who are less experienced in this area of research. The description spanning the entirety of Chapter 2 has been designed to offer readers with an introduction to this particular system, with some insights aimed at aspects considered more important. Therefore, it is believed that in addition to providing a detailed understanding of how *Matlab* code is structured, this paper may be useful to those who are new to this topic.

5.2 Simulation Framework

The framework that was built started from a fairly solid position, with the interface between *Matlab/Simulink* and *Unreal Engine*, however, it had limitations that were overcome by making changes and additions. The implementation provided integrated the SLAM system within *Simulink*, but it was deemed unsuitable to meet the levels of customization sought and was then remodeled as seen in the diagram of Figure 3.3.
The applications of *FFmpeg* varies according to need. In the simplest cases it is used only to break down into frames the videos obtained as output from *Simulink*, but it can also be used as an alternative way to reduce the resolution or the frame

rate if other set ups were to be evaluated.
Unreal Engine has proven to be a powerful tool for generating and manage virtual environments. Only the scenario provided by *Mathworks* was used in this thesis, however, alternatives were explored to see if any *UE* project could be used. A second scenario was actually prepared for use (Fig. 5.1). It reproduced a small residential neighborhood through the use of two different free packs downloaded from the *UE Marketplace*: the *Downtown West Modular Pack* developed by *PurePolygons* [36] and the *Vehicle Variety Pack* [39], developed by *Switchboard Studios*.

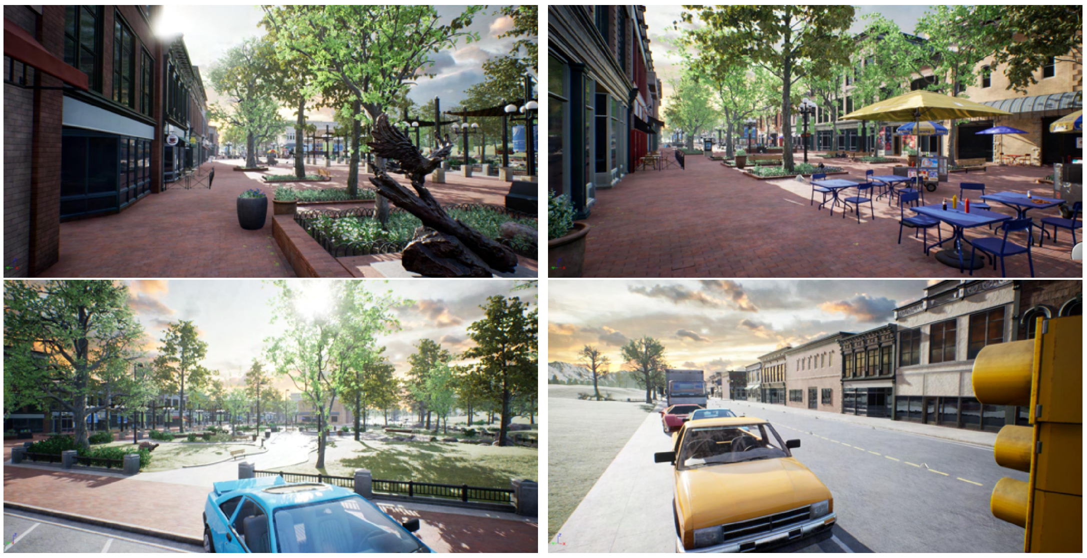


Figure 5.1: Some screenshots taken from the discarded virtual scenario.

This was a very valid environment with high resolution textures and very high levels of detail, which had been modified and adapted to function like the one used in the thesis. Unfortunately, it was later realized that there was a scale problem that could not be solved. The assets were created by the authors with an unrealistic scale, so the size of the objects differs from their real-world counterpart (for example, to have the cars in the same scale as the rest of the environment, they would have to be about 9 meters long). Maintaining these dimensions would have caused an incorrect estimate in the trajectory reading once interfaced with *Simulink*. Due to the complexity of the scenario, it was not possible to change the scale of every asset, so the work had to be discarded. No other free projects were found that could meet the needs of this work. Despite this setback that did not allow the assessments of errors to proceed, it was still possible to confirm the framework's compatibility with this (and therefore any other) project.
Given the configuration that was intended to be reproduced, here were plotted only trajectories in the horizontal plane by varying the yaw angle. The system, however,

is set up to be able to vary all six degrees of freedom. This increases its versatility, allowing it to reproduce with precision the motion of any recording platform, whether in indoor or outdoor environments.
There certainly remains much work to be done on the realism of the scene, especially if the goal is to copy from a sequence of real images. The aspect in which I realize there are more deficiencies is that of lighting. This is very complex to model properly and requires skills in the use of *UE* that are beyond my capabilities. In general, should it not be possible to model from scratch this or other aspects of the environment, one can use projects behind a pay wall that are designed by professionals or amateur artist as a good starting point. It is important to note that the fidelity with which the scenario is created does not affect the validity of the framework that has been illustrated.

In light of these facts, the goal of presenting a universal and self-sufficient framework for virtual images generation and processing is considered to have been achieved. A future development of interest is to integrate a *Simulink* model for vehicle dynamics into this framework, so as to complete the whole simulation given by environment, sensors and vehicle. While it is believed that this addition is not necessary to reproduce the correct operation of the SLAM system, it can expand the scope of the simulations.

5.3 Simulation results

Is performing tests with images captured in a virtual environment equivalent to using those taken from a dataset? While it is difficult to give a definitive answer to this question, the results obtained are promising and they suggest that a sufficiently well modeled scenario can come very close to the real case. An underestimation of the error was expected from the beginning, as some disturbance phenomena could not be modeled.
Overall, several hundred tests were carried out, both to understand the influence of the parameters of ORB-SLAM2 and to evaluate the effect of different trajectories and characteristics of the environment. No significantly different response was observed between the real and simulated environments when trying to replicate the scene. To be fair, given the way the system itself works, it would have been necessary to reproduce an exact copy of the path and scene seen in KITTI to confirm the validity of the virtual alternative. A future study might be to prepare a room so that it is easy to recreate it in *Unreal Engine*, and then trace the same short trajectory in

the two environments. The two sets of images, one from the real environment and one from the simulation, can then be fed into the SLAM algorithm to make a more accurate comparison. Given the expected underestimation of the error, it is up to the user to consider whether to follow this more flexible and rapid testing method or to seek more accurate results with a dataset or personally acquired images. Personally, from what emerged in this thesis, I believe that virtual environments can be a valuable tool for conducting all relevant preliminary tests, but a final confirmation with images captured in the real world is still necessary to definitively validate the worthiness of the system being tested.

Another study of interest that may be conducted at a later time, concern the impact of textures resolution (and in general, graphic quality) in the SLAM system performance. Being able to figure out what minimum level of graphics quality, depending on the resolution of the sensor being used, can be used without deteriorating trajectory estimation allows the computational load to be minimized. This would make it possible to use this configuration even with less performing hardware.

Bibliography

[1] *Cadena, C. et al. "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age." IEEE Transactions on Robotics 32.6: 1309–1332, 2016*

[2] *Campos, C., & Elvira, R., & Rodriguez, J.J.G., & Montiel, J.M.M., & Tardòs, J.D., ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM. IEEE Transactions on Robotics, 23 Apr 2021*

[3] *Delmerico, J., & Scaramuzza, D., A Benchmark Comparison of Monocular Visual-Inertial Odometry Algorithms for Flying Robots. In Proceedings of the IEEE Internetional Conference on Robotics and Automation (ICRA), pages 2502-2509, 2018*

[4] *Dissanayake, G., & Huang,S., & Wang,Z., & Ranasinghe, R., A review of recent developments in Simultaneous Localization and Mapping. In International Conference on Industrial and Information Systems, pages 477–482. IEEE, 2011.*

[5] *Durrant-Whyte, H., & Bailey, T., Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms. In Robotics and Automation Magazine, 2006*

[6] *Endres, F., Hess, J., & Sturm, J., & Cremers, D., & Burgard, W., 3D Mapping With an RGB-D Camera. IEEE TRANSACTIONS ON ROBOTICS, VOL. 30, NO. 1, February 2014*

[7] *Gao, X.-S., & Hou, X.-R., & Tang, J., & Cheng, H.F., "Complete Solution Classification for the Perspective-Three-Point Problem." IEEE Transactions on Pattern Analysis and Machine Intelligence. Volume 25,Issue 8, pp. 930–943, August 2003.*

[8] *Geiger, A., & Lenz, P., & Stiller, C., & Urtasun, R., Vision Meets Robotics: The KITTI Dataset*

[9] *Geromichalos, D., & Azkarate, M., & Tsardoulias, E., & Gerdes, L., & Petrou, L., & Perez Del Pulgar, C., SLAM for autonomous planetary rovers with global localization, 2020*

[10] *Mourikis, A.I., & Roumeliotis, S.I., A Multi-State Constraint Kalman Filter for Vision-aided Inertial Navigation. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pages 3565–3572. IEEE, 2007.*

[11] *Raùl Mur-Artal, & Montiel, J.M.M., & Tardòs, J.D., ORB-SLAM: A Versatile and Accurate Monocular SLAM System. IEEE Transactions on Robotics, Vol.31, No.5, October 2015*

[12] *Raùl Mur-Artal, & Tardòs, J.D., ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. IEEE Transactions on Robotics, 19 Jun 2017*

[13] *Ruble, E., & Rabaud, V., & Konolige, K., & Bradski, G., ORB: an effective alternative to SIFT or SURF. IEEE International Conference on Computer Vision (ICCV), pp.2564-2571, 2011*

[14] *Lynen, S., & Sattler, T., & Bosse, M., & Hesch, J., & Pollefeys, M., & Siegwart, R., Get out of my lab: Large-scale, real-time visual-inertial localization. In Proceedings of Robotics: Science and Systems Conference (RSS), pages 338–347, 2015.*

[15] *Scaramuzza, D., & Zhang, Z., Visual-Inertial Odometry of Aerial Robots. In Springer Encyclopedia of Robotics, 2019*

[16] *Smith, R., & Self, M., & Cheeseman, P. Estimating uncertain spatial relationships in robotics. In I.J. Cox and G.T. Wilfon, editors, Autonomous Robot Vehicles, pages 167-193. Springer-Verlag, 1990.*

[17] *Torr, P.H.S., Zisserman, A., "MLESAC: A New Robust Estimator with Application to Estimating Image Geometry." Computer Vision and Image Understanding, Volume 78, Issue 1, Pages 138-156, ISSN 1077-3142, 2000*

[18] Xiang Gao, & Tao Zhang
"Introduction to Visual SLAM. From Theory to Practice", Singapore, Publishing House of Electronics Industry, 2021

[19] https://www.blender.org/, retrieved on 12/02/2023,
Blender Website

[20] https://ffmpeg.org/, retrieved on 12/02/2023,
FFmpeg Website

[21] https://artsandculture.google.com/asset/hilare-autonomous-mobile-robot-laboratory-of-analysis-and-architecture-of-systems/UwFV2X6hPn0NjA, retrieved on 12/02/2023,
Google Arts & Culture, Hilare autonomous mobile robot, Laboratory of Analysis and Architecture of Systems, 1977

[22] https://www.hp.com/it-it/shop/offer.aspx?p=b-hp-reverb-vr-headset, retrieved on 12/02/2023,
HP Website

[23] https://www.cvlibs.net/datasets/kitti/, retrieved on 12/02/2023,
KITTI Vision Benchmark Suite Website

[24] https://it.mathworks.com/help/vision/ug/stereo-visual-slam-for-uav-navigation-in-3d-simulation.html, retrieved on 12/02/2023,
MathWorks Website

[25] https://it.mathworks.com/help/vision/ug/stereo-visual-simultaneous-localization-mapping.html, retrieved on 12/02/2023,
MathWorks Website

[26] https://it.mathworks.com/products/matlab.html, retrieved on 12/02/2023,
MathWorks Website

[27] https://it.mathworks.com/products/simulink.html, retrieved on 12/02/2023,
MathWorks Website

[28] https://www.mathworks.com/help/vdynblks/ug/customize-scenes-using-simulink-and-unreal-editor.html, retrieved on 12/02/2023,
MathWorks Website

[29] https://it.mathworks.com/help/vdynblks/ug/install-support-package-and-configure-environment.html, retrieved on 12/02/2023,
MathWorks Website

[30] https://it.mathworks.com/help/driving/ug/select-waypoints-for-3d-simulation.html, retrieved on 12/02/2023,
MathWorks Website

[31] https://it.mathworks.com/matlabcentral/fileexchange/52065-rq-decomposition-using-givens-rotations?focused=3884550&s_tid=gn_loc_drop&tab=function, retrieved on 12/02/2023, *MathWorks Website*

[32] https://medrobotics.ri.cmu.edu/node/128458, retrieved on 12/02/2023,
Medical Robotics, Carnegie Mellon University

[33] https://it.wikipedia.org/wiki/Microsoft_Kinect,
Microsoft Kinect, Wikipedia

[34] https://www.microsoft.com/it-it/microsoft-365/excel, retrieved on 12/02/2023,
Microsoft Website

[35] https://en.wikipedia.org/wiki/Odometer, retrieved on 12/02/2023,
Odometer, Wikipedia

[36] http://www.purepolygons.com/, retrieved on 12/02/2023,
PurePolygons Website

[37] https://www.smithsonianmag.com/innovation/how-does-human-echolocation-work-180965063/, retrieved on 12/02/2023,
Smithsonian Magazine

[38] https://www.unrealengine.com/en-US, retrieved on 12/02/2023,
Unreal Engine Website

[39] https://www.unrealengine.com/marketplace/en-US/product/bbcb90a03f844edbb20c8b89ee16ea32, retrieved on 12/02/2023,
Vehicle Variety Pack

Appendices

Appendix A

Visual Basic Sub-Routines

The operations performed by the code below are as follows:

1. Imports the text file containing the poses into an Excel sheet;
2. Rearranges the rows and columns of the sheet so that there are six columns each containing one DoF;
3. Swaps columns so that they are reordered as desired: (**X**, **Pitch**, **Y**, **Yaw**, **Z**, **Roll**);
4. Deletes columns containing unnecessary data that had been imported;
5. Remove first two characters in each cell, so that within the cells there are only numerical values.

```
'----------------------------------------------------------------
Sub ImportTextFileToExcel()
    Dim textFileNum, rowNum, colNum As Integer
    Dim textFileLocation, textDelimiter, textData As String
    Dim tArray() As String
    Dim sArray() As String
    textFileLocation = "C:\Users\Davide\Desktop\Pos_Rot.txt"
    textDelimiter = " "
    textFileNum = FreeFile
    Open textFileLocation For Input As textFileNum
    textData = Input(LOF(textFileNum), textFileNum)
    Close textFileNum
    tArray() = Split(textData, vbLf)
    For rowNum = LBound(tArray) To UBound(tArray) - 1
        If Len(Trim(tArray(rowNum))) <> 0 Then
            sArray = Split(tArray(rowNum), textDelimiter)
            For colNum = LBound(sArray) To UBound(sArray)
               ActiveSheet.Cells(rowNum + 1, colNum + 1) = sArray(
   colNum)
            Next colNum
        End If
    Next rowNum
    MsgBox "Data Imported Successfully", vbInformation
End Sub
'----------------------------------------------------------------
Sub SplitEveryOther()
Dim Rng As Range
Dim InputRng As Range, OutRng As Range
Dim index As Integer
xTitleId = "SplitEveryOther"
Set InputRng = Application.Selection
Set InputRng = Application.InputBox("Range :", xTitleId, InputRng.
   Address, Type:=8)
Set OutRng = Application.InputBox("Out put to (single cell):",
   xTitleId, Type:=8)
Set OutRng = OutRng.Range("A1")
num1 = 1
num2 = 1
For index = 1 To InputRng.Rows.Count
    If index Mod 2 = 1 Then
        OutRng.Cells(num1, 1).Value = InputRng.Cells(index, 1)
        num1 = num1 + 1
    Else
        OutRng.Cells(num2, 2).Value = InputRng.Cells(index, 1)
        num2 = num2 + 1
    End If
Next
End Sub
'----------------------------------------------------------------
Sub Swap_Columns()

    Sheets("Home").Columns("A:A").Cut Destination:=Columns("O:O")
    Sheets("Home").Columns("G:G").Cut Destination:=Columns("A:A")
    Sheets("Home").Columns("O:O").Cut Destination:=Columns("G:G")

    Sheets("Home").Columns("B:B").Cut Destination:=Columns("O:O")
    Sheets("Home").Columns("H:H").Cut Destination:=Columns("B:B")
    Sheets("Home").Columns("O:O").Cut Destination:=Columns("H:H")

    Sheets("Home").Columns("C:C").Cut Destination:=Columns("O:O")
    Sheets("Home").Columns("I:I").Cut Destination:=Columns("C:C")
    Sheets("Home").Columns("O:O").Cut Destination:=Columns("I:I")

    Sheets("Home").Columns("D:D").Cut Destination:=Columns("O:O")
    Sheets("Home").Columns("J:J").Cut Destination:=Columns("D:D")
    Sheets("Home").Columns("O:O").Cut Destination:=Columns("J:J")

    Sheets("Home").Columns("E:E").Cut Destination:=Columns("O:O")
    Sheets("Home").Columns("K:K").Cut Destination:=Columns("E:E")
    Sheets("Home").Columns("O:O").Cut Destination:=Columns("K:K")

    Sheets("Home").Columns("F:F").Cut Destination:=Columns("O:O")
    Sheets("Home").Columns("L:L").Cut Destination:=Columns("F:F")
    Sheets("Home").Columns("O:O").Cut Destination:=Columns("L:L")

End Sub
'---------------------------------------------------------------
Sub Delete_Example1()

  Columns("G:K").Delete

End Sub
'---------------------------------------------------------------
Sub RemoveFirstTwoCharactersInEachCell()
For Each cell In Range("A1:F1", Range("A999999:F999999").End(xlUp))
If Not IsEmpty(cell) Then
cell.Value = Right(cell, Len(cell) - 2)
End If
Next cell
End Sub
'---------------------------------------------------------------
```