Multi-sensors SLAM simulation for Planetary Rover Exploration

Tesi di Laurea Magistrale in
Space Engineering - Ingegneria Spaziale

Author: **Davide Viviani**

Student ID: 943486
Advisor: Prof. Mauro Massari
Academic Year: 2021-22

Abstract

In the world of planetary space exploration rovers have been used for many years now. We understood how to build, send and land them safely on other celestial bodies and their main goal is usually the exploration of the surface where they operate. Space exploration has brought rovers to the surface of the Moon and Mars, with now missions planning to go further and further. This push through further bodies brings an intrinsic and not trivial problem to the rover missions: the increasing difficulties in the communication with rover, pushing the research to more and more autonomous systems.
This is the main reason behind the development of a SLAM algorithm for planetary exploration, where the acronym SLAM stands for "Simultaneous Localization And Mapping", a self-explanatory name. This family of algorithms has the purpose of making the rover (or any other robotic system) able to navigate autonomously in a new environment, recognize features as landmarks, map them and orient itself in this map.

The following thesis analyzes the history and state of the art of the SLAM problem and its possible solutions while implementing a simulated digital environment into which generate and analyze data, proposing an innovative approach to planetary SLAM test and simulation. Then the SLAM algorithm is analyzed on the base of the digital data collected during the simulation and conclusions are drawn analyzing different parameters of the algorithm itself.
The algorithm tested aims at finding an innovative solution in the field of planetary SLAM implementation and testing, with the main goal of demonstrating how it can be analyzed from a virtual realistic planetary surface reconstruction.

The research inside this work is articulated in three sections: the simulation environment build up, the data elaboration and the actual SLAM implementation and test. The results are shown and analyzed from all of the three sections.

Keywords: SLAM, robotic exploration, Extended Kalman Filter, Mars, simulated environment.

Sommario

Nel mondo dell'esplorazione planetaria i rover sono strumenti utilizzati ormai da diversi anni. Abbiamo capito come costruirli, come inviarli, come farli atterrare con sicurezza su un altro corpo celeste. Il loro obbiettivo principale è l'esplorazione della superficie di altri corpi celesti. Ci siamo spinti con rover sulla Luna e su Marte, con missioni che pianificano di portarli sempre più lontano. Questa spinta verso altri corpi del Sistema Solare porta con sé un problema non semplice per le missioni future: le sempre maggiori difficoltà nelle comunicazioni tra il rover e la Terra. Perciò, i rover devono attrezzarsi per essere sempre più autonomi nel loro processo di esplorazione. Queste sono le ragioni che spingono allo sviluppo di un algoritmo SLAM atto all'esplorazione planetaria, in cui l'acronimo SLAM sta per: "Simoultaneous Localization And Mapping" (Localizzazione e mappatura simultanee), un nome che già spiega molto da sé. Questa famiglia di algoritmi ha l'obbiettivo di rendere il rover (o qualsiasi altro sistema robotico) capace di navigare autonomamente in un ambiente nuovo, riconoscerne particolarità come punti noti del paesaggio, mapparle ed orientarsi in questa mappa da sé costruita.

La seguente tesi analizza la storia e lo stato dell'arte dello SLAM e le sue possibili soluzioni. Contemporaneamente, si pone l'obbiettivo di ricreare un ambiente di simulazione digitale in cui generare ed analizzare dati utili. Successivamente, l'algoritmo SLAM viene testato e analizzato sulla base dei dati digitali così generati e alcune conclusioni vengono tratte, confrontando le variazioni nei diversi parameteri dell'algoritmo.
Gli algoritmi presentati in questa tesi hanno l'obbiettivo di trovare una soluzione innovativa al problema dello SLAM planetario, con le sue particolarità e utilizzando i tipici sensori disponibili a bordo di un rover.

La tesi qui presentata è articolata su tre parti: la costruzione dell'ambiente simulato, l'elaborazione dei dati ricevuti dai sensori e l'implementazione vera e propria dello SLAM. I risultati di tutte e tre le parti della simulazione saranno analizzati e presentati.

Parole chiave: SLAM, esplorazione robotica, filtro di Kalman esteso, Marte, ambiente simulato.

Contents

1 | Introduction

The first chapter here presented contains an introductory analysis on the SLAM problem, together with its intrinsic problems and the solutions proposed in this thesis work.

1.1. The need of a planetary SLAM

Robotic planetary space exploration is one of the main area of research for space agencies and several rovers have landed on the surfaces of Mars, Moon and other celestial bodies in last decades.
The technology available for rovers has improved substantially in this time period, exploiting more and more information from the surfaces explored. Rovers have proved themselves to be useful and robust tools, with long lifespans and mission duration. They capabilities drive agencies to push the limit further and further, planning to explore many more celestial bodies in the Solar System by means of robotic agents and rovers in particular.
However, a big problem still persists and affects how far a robotic mission can be pushed: the delays and difficulties in space communications create important issues when controlling the rover from Earth.
This problem have brought to the search of more and more autonomous exploration systems, capable of planning exploration missions while on the field by themselves and capable to autonomously survive the surrounding environment. In this area the Simultaneous Localization and Mapping problem allocates itself.
This family of algorithms has the purpose of making the rover (or any other robotic agent) able to navigate autonomously in a new environment, recognize features as landmarks, map them and orient itself in this map. In particular, it has to be tailored on the planetary surface exploration problem, counting on limited energetic and computational sources and on a limited number of sensors.

The work in this thesis in done in order to find a suitable way to reproduce a realistic SLAM problem on a planetary surface. This is achieved virtually, enabling cheaper and more flexible analysis tools and a easy to scale and modify simulation environment.

1.2. The SLAM problem

The Simultaneous Localization and Mapping problem, commonly known as SLAM or CML (Concurrent Mapping and Localization), is a relative well known problem in autonomous driving vehicles since early '90s. However, thirty years later, it still has a growing interest in the scientific community, because of all the different disciplines and areas involved and due to the many possible improvements still to be reached.
The SLAM problem presents itself when a robot does not have a knowledge both of its position and of the map of the environment in which it is moving. Instead, knowledge of the control input received (u_t) and of the sensors measurements (z_t) is available. The SLAM definition is well described by Thrun, Burgard and Fox [11]:

The term simultaneous localization and mapping describes the following problem: in SLAM, the robot acquires a map of its environment while simultaneously localizing itself relative to this map.

SLAM is intrinsically more difficult than localization alone, in that the map is unknown and has to be estimated along the way. It is also more difficult than mapping with known poses, since the poses are unknown and have to be estimated along the way.

The SLAM is a family of algorithms which involves several topics in the robotics and calculus fields: from sensor data exploitation to computational cost minimization, from data association to on-board computational capacity management.
A standard explanation and a first approach to SLAM is described in [5]. The SLAM problem was approached from a probabilistic point of view and solved by means of an Extended Kalman Filter.
A quick explanation of the basic principles and assumptions is going to be shown in section 1.2.1, together with the improvements which followed along the years.

From a probabilistic point of view, the SLAM can be divided into two main forms: the Online SLAM Problem and the Full SLAM Problem. The first one involves only the estimation of variables that persist at time t, like the pose (x) and the map (m) at time t. The second, instead, seeks to calculate a posterior over the entire path ($x_{1:t}$) along with the map. This difference has impact in the kind of algorithms that can be used for the SLAM solving. Another characteristic of the SLAM problem is related to the nature of the estimation problem: continuous and discrete components. The continuous estimation problem pertains to the location of the objects in the map (landmarks) and the robot's own pose variables. The discrete nature has to do with correspondence: when an object is detected the SLAM algorithm must reason about the relation of this object to previously detected objects.

1.2.1. History of SLAM

The history of SLAM, from the very first approaches to current developments is well described in the work *Past, present, and future of simultaneous localization and mapping*[3]. In SLAM the robot state is described by its pose, while the map is a representation of aspects of interest considered important for the kind of representation seek.
The importance of the map lays also in the fact that it can be exploited by the robot to correct the error in its estimation. By re-visiting places and/or re-observing already seen landmarks, the robot can correct its estimations and provide a more robust guess of both the map and its pose. Obviously, SLAM becomes fundamental when a map of the environment is not avaliable a priori.
The last thirty years of research in SLAM can be divided in three time periods:

- **Classical age** (1986-2004): introduction of SLAM as a probabilistic formulation;
- **Algorithmic-analysis age** (2004-2015): studying and exploiting of SLAM properties such as convergence, efficiency and observability;
- **Robust-perception age** (2015-now): characterized by robust performance, high-level perception, high-level understanding of the problem, etc...

The *Classical age* introduced the SLAM problem as a probabilistic formulation, including the Extended Kalman Filter approach that is going to be analyzed in this thesis. It so introduced Kalman filters solutions, Particle filters solutions and Maximum Likelihood Estimation solution for the data association problem, explained in detail in chapter 4. In this first period the importance of pivotal problems like the finding of an efficient data association were highlighted.

During the following age, the *Algorithmic-analysis age*, the SLAM properties were analyzed and exploited, highlighting the importance of phenomena as the sparsity in order to improve efficiency of the algorithms. In this period the intersection between SLAM and other disciplines became clear: on the lower level in SLAM, the front end, the problems intersect themselves with disciplines like computer vision and signal processing; on an higher level, SLAM founds correspondences in disciplines as geometry, graph-theory,

probabilistic estimation and many more.
In this context, the difficulties of the deployment of such a complex algorithm on real hardware should not be forgot and this problem claimed for the necessity of virtual simulation environments. This period evolved the importance on data association, coupling it with the Loop Closure problem, explained in section 1.3.3. With these concepts in mind we approached our age and the current state of the art of SLAM.

1.2.2. State of the Art

As mentioned before, we are currently in the *Robust-perception age* for SLAM algorithms. This translates in less error-prone algorithms, mapping of more extended environments, better feature extraction, better exploiting of computational resources and more efficient and tailored data association solutions.
In order to achieve this performances, way more advanced if compared to the first SLAM solutions, the architecture on the algorithm is divided into two main blocks, each of them with specific features and purposes:

- The **front-end**, is in charge of the data flow coming from the sensors, abstracting sensor model, filtering information and providing suitable data sets for the back-end;
- The **back-end**, performs the optimization of the data working on the probability distributions and the errors minimization.

Modern SLAM make use of several development in the fields of computer vision, sensor fusion, optimization algorithms, machine learning and many more subjects. Time by time a multi-disciplinary approach to SLAM, with team of experts from different areas, is becoming the standard.

1.2.3. Classical formulation of SLAM

The very first approach to SLAM was the probabilistic formulation of the problem, well illustrated in the work by White and Bailey [5] and in the book by Thrun [11].
These first approaches established a statistical basis for describing relationships between landmarks and uncertainty. It was shown that an high degree of correlations between the estimates of landmarks locations in the map is present and these correlations are growing with time as the number of observations increases. The correlation between the landmarks estimates is present because of the uncertainty in the estimate of the vehicle location, which propagates into the observations.

The probabilistic definition of SLAM requires some quantities to be defined:

- x_k: state vector containing location and orientation of the robot;
- u_k: vector containing the control inputs at different times;
- m_i: vector with the location of the landmark i;
- z_{ik}: observation of a landmark i at time k, written also as z_k.

Also the following sets have to be defined:

- $X_{0:k} = [x_0, x_1, ..., x_k]$: history of vehicle locations;
- $U_{0:k} = [u_1, u_2, ..., u_k]$: history of control inputs;
- $m = [m_1, m_2, ..., m_n]$: the set of all landmarks;
- $Z_{0:k} = [z_1, z_2, ..., z_k]$: the set of landmarks observations.

The probabilistic SLAM requires the following probabilistic distribution to be computed at all times k:

$$P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) \tag{1.1}$$

This probability distribution describes the joint posterior density of the landmark locations and vehicle state at time k given the observations and the control inputs up to and including time k, together with the initial state of the vehicle [5].
According to probabilistic rules, the Bayes theorem can be applied, with the definition of a state transition model and an observation model to be introduced. Respectively, they described the effects of the control input and of the observation.

Theorem 1.1 (Bayes theorem).

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

The *motion model*, describing the vehicle motion as a Markov process with dependence only on the previous pose x_{k-1} and the control input u_k applied, is represented by:

$$P(x_k \mid x_{k-1}, u_k) \tag{1.2}$$

And the *observation model*, describing the probability of making an observation z_k when vehicle location and landmark locations are known, is represented by:

$$P(z_k \mid x_k, m) \tag{1.3}$$

The recursion is shown to be a function of both the observation and motion models. According to this theorem, the SLAM problem can be reformulated in two steps, allowing a recursive prediction-correction implementation form:

- **Time-update**:

$$P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) = \int P(x_k \mid x_{k-1}, u_k) P(x_{k-1}, m \mid Z_{0:k-1}, U_{0:k-1}, x_0) dx_{k-1} \tag{1.4}$$

- **Measurement-update**:

$$P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k \mid x_k, m) P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0)}{P(z_k \mid Z_{0:k-1}, U_{0:k}} \tag{1.5}$$

Equations 3.3 and 3.4 provide a recursive procedure for calculating the joint posterior in equation 1.1. The recursion is a function of a vehicle model and an observation model. The observation model makes explicit the dependence of observations on both the vehicle and landmark locations. Solution to the probabilistic SLAM problem involve an appropriate representation for the observation model equation and the motion model equation which allows efficient and consistent computation of the prior and posterior distributions. The most common representation is in the form of a state-space model with additive Gaussian noise, leading to the use of the extended Kalman filter (EKF) to solve the SLAM problem. In this thesis the EKF solution has been adopted for solving the SLAM problem and it is going to be explained in detail in chapter 3.
In the last years other solution for the SLAM problem have been found and different modern strategies substituted the use of the EKF. However, the EKF is still a solid base to approach the simultaneous localization and mapping problem.

1.3. The Data Association problem

As stated before, the SLAM problem is a multi-disciplinary subject of research involving several areas. One of the main interests lays in translating the data from the front-end sensors into useful and exploitable information for the back-end. Particularly, the observation model has the aim to correct the current pose estimate by making use of the current landmark observations.
It is possible to state that the robot is moving in an environment while acquiring several sensor acquisitions. As it moves around, the data received from the sensors may observe the same landmark more than once, strongly affecting the results.
As explained in the work by Bailey and White [2]:

A major hurdle in the implementation of SLAM methods is to correctly associate observations of landmarks with landmarks held in the map.

The problem above is called the *Data Association problem.* Incorrect data association can lead to catastrophic failure of the SLAM. Data association is particularly important when a vehicle return to previously mapped region, problem which is called the *Loop Closure problem.* Both of the introduced problems are strictly related to the recognition of a landmark, in the so called *Data Recognition problem.* These problems are described in detail in the following sections and in chapter 4.

1.3.1. Data Recognition

A pivotal role in the data association problem is played by the recognition of an object itself. As the reader may notice, the line between the data recognition and the data association is very thin and, practically, they often merge themselves in real applications. Here data recognition is thought as the process of distinguishing a feature, an object or a particular pattern in the set of points coming from the sensor data. In this process the object is only recognized as distinguished from the whole cloud of points, while in the data association is marked as a specific landmark, which can be already seen previously or which can be marked as a new landmark in the map.
Data recognition is mainly a computer vision problem, seeing the computer vision as a general field in which to interpret signals from the sensors as maps with features. By doing so, several well known computer vision techniques can be applied.

Since it is not the main objective of this work, these techniques are not illustrated here. However, the data recognition is part of the SLAM algorithm implemented in this thesis and a strategy to apply it is shown in chapter 4.

1.3.2. Data Association

Since its origins, the Data Association problem has had a pivotal role in the solution of the SLAM problem and its implementation. The difficulties are contained in the fact that once some data are associated, it is not possible to remove the association anymore. This phenomena translates in the fact that if a wrong data association is taken into account, the problem acquires a degree of error which can not be revised. The incorrect data association can cause catastrophic failures of the localization algorithm.
At the current state of the art, several different implementation of the data association have been explored. However, it still remains an open problem, since it is usually dependent on the sensors and/or the type of environment in which the robot is deployed. For example, in the case under study in this thesis, the planetary SLAM must confront itself with a data association tailored on environments with similar landmarks and the absence of easy recognizable unique features, as explained in section 1.4.
One of the most common algorithm to solve the data association problem in the case of sensors generating data in the form of point clouds is the *Iterative Closest Point* (ICP) algorithm, along with its different versions and implementations. The ICP is used also in this work in order to obtain a basic data association algorithm to be used inside the main SLAM implementation. The detailed structure of the data association is illustrated in chapter 4.
Since the objective of this thesis is to demonstrate the solution of a SLAM algorithm in a simulated environment, the focus of the work is not on the data association which is the reason behind its basic version implementation; it so has a great margin of improvement.

1.3.3. Loop Closure

Loop closure is a special kind of data association which happens when an previously visited place is visited again by the robot. The loop closure problem consist in being able to recognize the already seen place and correct the whole estimation based on that. It can be easily understood that is a special case of data association. Loop closures are fundamental in SLAM and wrong loop closures bring the SLAM to more dramatic errors than a wrong normal data association error can bring.

In this thesis the loop closure problem is not analyzed for sake of simplicity, as it is a very complicated and advanced problem.

1.4. Planetary SLAM

The focus of this thesis is on developing an efficient way to simulate the process of a SLAM algorithm in the case of a planetary surface exploration. The following sections have the purpose of showing which are the typical characteristics of a planetary environment and how the SLAM problem applies to them.

1.4.1. Planetary environment

The SLAM algorithm of this work aims at solving the problem of localization and mapping for a rover on a planetary surface. In particular the simulated environment is built to recreate the surface of Mars. Except for a few of the characteristics, the same applies for other rocky celestial body surfaces; as an example it applies also to the Moon surface exploration. Differently from other well studied SLAM applications, i.e. the close environment exploration, the planetary surface SLAM lacks of several important and useful features for the algorithm:

1. ceiling;
2. corners;
3. squared easy-to-distinguish features;
4. constant illumination.

The first three elements are strictly geometrical useful elements: they are easy exploitable by computer vision algorithms and they are often useful for efficient data association solutions. On the other side, the planetary surface under investigation is an open surface, characterized by similar features represented by irregularly shaped rocks, without the presence of clear corners and very difficult to distinguish one from the other. Moreover, most of them are covered by dust, giving the same shade of color and making difficult the usage of a data association exploiting the color properties of landmarks.
Regarding the light, the reader must consider that on the surface of a celestial body different from Earth there is no artificial illumination during eclipse period, so it is important to keep under consideration also this phenomenon.

1.4.2. Available sensors

By enumerating some of the problems related to a planetary SLAM in the sections above, sensors are often mentioned. The whole structure of a SLAM algorithm relies on the sensors available on-board the vehicle. The sensors change from robot to robot and are usually selected tailoring their functioning to the special tasks of a certain robot.
In this thesis two main sensors from which to acquire the data necessary for the solving of the SLAM are considered:

- one *Stereo-Cam*, simulating the specifics of the Stereo-Cam on-board the rover Perseverance from the Mars 2020 mission [8];
- one *LIDAR* sensor, cooperating with the Stereo-Cam in generating useful point clouds of the surrounding environment.

The selected sensors have been simulated by means of the instruments available in Unreal Engine, Matlab and Simulink, as described in detail in chapter 2.
The reason behind the selection of this two sensors relies at recreating and testing a rover equipped with common sensors in space exploration, while exploiting the different properties of the two in order to make the measurements more reliable and robust. The different results generated by the sensor are discussed in chapter 5.

1.4.3. Computational resources

The computational resources required by the SLAM proposed in this thesis have not been taken as a strict requirement and they have not been analyzed in detail. However, several choices have been adopted in order to make the whole simulation applicable on a common laptop.
Among these choices there are some more important ones, which are going to be explained further in the text:

- Simulation splitting in three separate parts;
- StereoCam image dimensions reduction;
- Gray scale simulated environment inside Unreal Engine.

1.5. Thesis objective

As mentioned before, this thesis aims at developing an efficient SLAM algorithm for planetary surface exploration by means of a rover.
To tackle the various problems of hardware implementation and building, the thesis proposes an innovative approach by developing a realistic well detailed simulation environment in which to test various versions and implementations of a SLAM algorithm.
The simulation environment is build to be a realistic as possible, together with the intrinsic implementation of the rover sensors and their acquisitions. By means of this virtual representation, the test phase of the SLAM can be moved inside the digital environment, reducing the cost of physical implementation, testing and modification of a real test rover.
A second parallel objective is to pose the basis of a simulation architecture which can be exploited for various other robotic space exploration simulations.

1.6. Thesis structure

This work has been divided in different chapters, each of them explaining pivotal parts of the architecture of the thesis:

- Chapter 1, presents the problem under investigation, the state of the art and the solution proposed;
- Chapter 2, shows the softwares and the procedures followed to recreate the simulation environment;
- Chapter 3, shows the theory and the concepts behind the SLAM problem and the solution chosen;
- Chapter 4, presents the Data Association problem and the solution chosen;
- Chapter 5, presents the final results for the simulation environment and the actual SLAM implementation;
- Chapter 6, illustrates the possible future developments of this work and the final conclusions drawn.

2 | Simulation environment

In this chapter the procedures and setups to simulate the movement of the rover over the surface of Mars are illustrated. The simulation of the data acquisition works over four softwares: Matlab, Simulink, Blender and Unreal Engine. In the following pages they are going to be explained in details.

2.1. Simulating Mars

The planetary SLAM algorithm of this work is developed to operate on a planetary surface, in particular it works over a simulated Mars surface. The characteristics of the surface of Mars are several and, as other planetary surfaces, its surface is very different from area to area over the whole planet.
In this work a classical desert-rocky area is simulated as a dusty plane covered in volcanic rocks of different size and shapes. Different areas of the Mars surface can be simulated in the same way, recreating canyons, planes, craters or other martian environments.
Mars is by far the best known planet of the Solar System after the Earth: we know how its surface looks like with good detail thanks to the images coming from the various orbiters and the several rovers landed over the years. Between the main features of Mars we have the characteristic reddish soil, the dust covering the whole surface and the presence of geological features of interest such as volcanic or sedimentary rock formations. These features must be taken into account when modelling a rover, and a SLAM algorithm, in order to better understand and tackle the various problem that could arise. For example, the dusty and sandy terrain is one of the main reasons behind the need of a SLAM algorithm: even without errors in the odometry readings, the sand may cause some of the rover wheels to slip while travelling a selected route, causing the final path to be different from the nominal one. In this situation the SLAM algorithm acquires importance, being able to correct the pose estimation by observing the surrounding environment with its sensors.

The simulation environment has so been set to recreate a martian plane area, covered by several dusty volcanic rocks. In order to that, the softwares described in the following sections have been adopted.

2.2. Simulation softwares

As stated before, the whole simulation has been built merging the capabilities of several different softwares, being able to manage several issues such as the trajectory of the rover, the color and texture of the surfaces or the light of the scene.

2.2.1. Blender

The first step of the simulation consists in modelling a slice of the surface of Mars in which to move the rover and its sensors. The software selected to achieve this task is Blender. Blender is a well known software inside the Design community due to its strong capabilities to manage very detailed surfaces and textures and to produce high resolution renders. It also has a useful Python console embedded, several libraries of different textures and pre-made surfaces available online.

Why Blender?

Blender is a free software which can be easily downloaded online and brings very strong capabilities in rendering surfaces, together with an easy to understand interface.
Blender has the possibility to work in several modes, depending on what is the task to be accomplished, with a very versatile interface capable of switching from one mode the the others in a click, or also being able to manage several modes at the same time. In this work Blender has been used at a beginner level in order to achieve a simple reconstruction of the environment. However, it can be exploited at an advanced level in order to create more complex and detailed surfaces for the simulations.

One of the main aspects behind the choice of Blender is its Python console: the surface can be created by programming the commands needed in Python and allowing extra capabilities like moving the camera of the scene in order to acquire different renders. In the next sections Blender and its use are going to be shown in detail.

Recreating Mars surface

For sake of simplicity the surface recreated is a plane, without hilly sections or slopes, covered with rocks different in shape and dimensions. It has to be pointed out that Blender has the capability to create a more complex and realistic surfaces, but this goes over the aim of this thesis. However, what is going to be illustrated here can be used to extend the research over more complex shapes.
Exploiting Blender capabilities, the creation of the environment for the simulation has been divided in three subsections, merged together only at the very last step of the Blender procedure. The different areas are:

- Main surface shape and texture;
- Rocks;
- Dust.

The different sections are going to be explained in the following paragraphs.

Main surface and texture

As said before, the main surface is a plane geometry over which the rocks are then placed and covered with dust.
In Blender the plane surface has been realized with a plane mesh, in which the x and y dimensions were set not to have a too big environment to render, avoiding slowing down the simulation. In the following step the texture of the terrain has been applied. In order to do that the *UV editing* and *Shading* sections of Blender were involved.
In the *UV editing* section the texture is fitted over the mesh, allowing the correct scale between the texture features and the size of the element on which the texture is applied. For example, if the texture of a small stone is produced and then applied to a big rock, the features of the texture will probably not look natural. By means of the *UV editing* section, this scaling problem is solved and the surface is going to look more similar to a natural one.
The real creation of the texture emerges in the *Shading* section, where the possibility to actually work and modify the textures is allowed. The results are shown in figure 2.1.
The nodal representation in the *Shading* section allows to work on the merge of several model for a certain texture and it has a graphical interface useful to set the different parameters in an interactive manner. It allows to link and merge four main parameters and bases during the creation of a texture: color, roughness, normal map and displacement, which all can be distinguished by the light brown color of their respective nodes in figure.

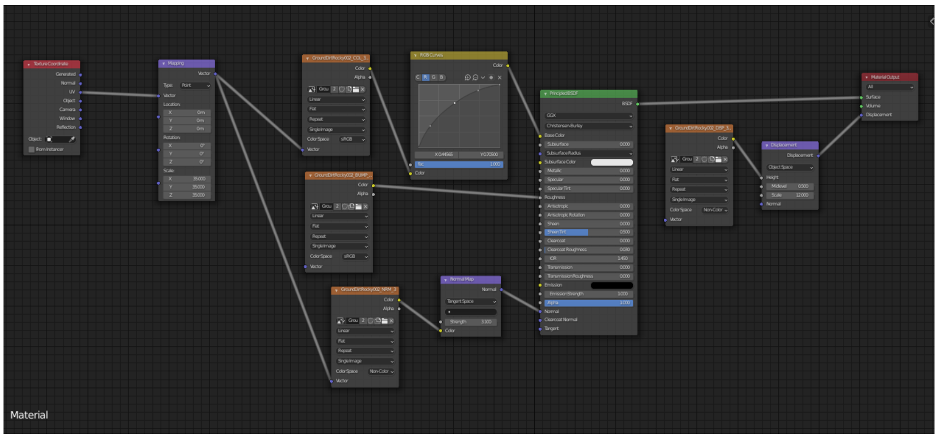


Figure 2.1: Nodes in Blender shading section

A deep and precise explanation of this parameters goes over the purpose of this work, however they can be explained as different maps that have to be overlapped and merged properly in order to obtain the desired texture. Playing with the parameters of these setups allows different textures, starting all from the same four files.
The files used in this work are taken from the Poligon website[1].
Once the UV editing and the Shading step are completed, the base ground is ready and the environment has to be filled with rocks and dust in order to recreate the martian surface needed. Figure 2.2 illustrates the final surface texture result.

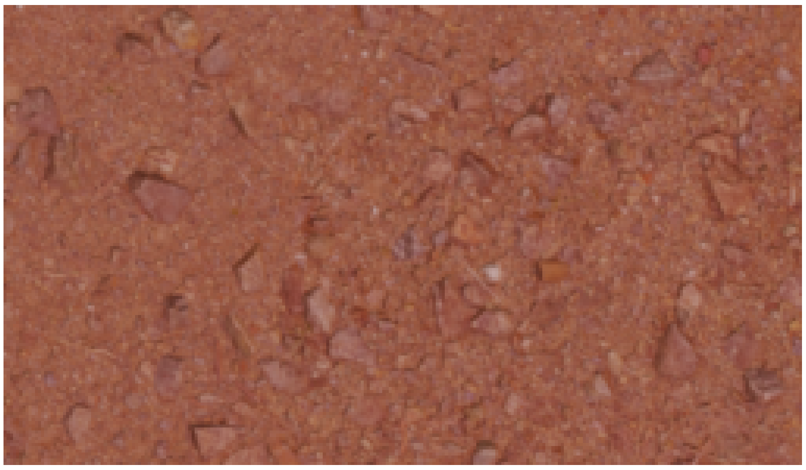


Figure 2.2: Mars ground reproduction

[1]https://www.poliigon.com/

Rocks

Once the surface is ready, the work focus has been shifted over the rocks to be built in order to fill the environment. Blender allows to create separate meshes in different *.blend* files and then merge them in a unique project. The rocks creation has so been divided in two steps: the creation of rock meshes and textures in a separate file and then the insertion of this file inside the already created plane surface.
As done for the plane surface, few meshes have been created in a Blender file: starting from the shape of a sphere and then modelling it with shapes similar the volcanic basaltic rocks inside the *Edit Mode*. Again, a basic shaping and modelling has been adopted, knowing that improvements in the shaping of each rock can be done.
Once the different objects with their shapes were enough detailed for their purpose, the work has been moved on texturing the rocks.
Again, the texture on the rocks has been applied with the same procedure shown in section 2.2.1, using the *Shading* mode and the nodes. All the rocks are modeled as black volcanic rocks, assuming to explore an area of the Mars surface rich in volcanic features. This is only an assumption used for this simulation and the whole scene can be modeled differently to resemble other martian areas. As in the step before, the surface textures is the result of the merging of a color map, a roughness map, a normal map and a displacement map. Again, these maps have been downloaded from the Poligon website[2].

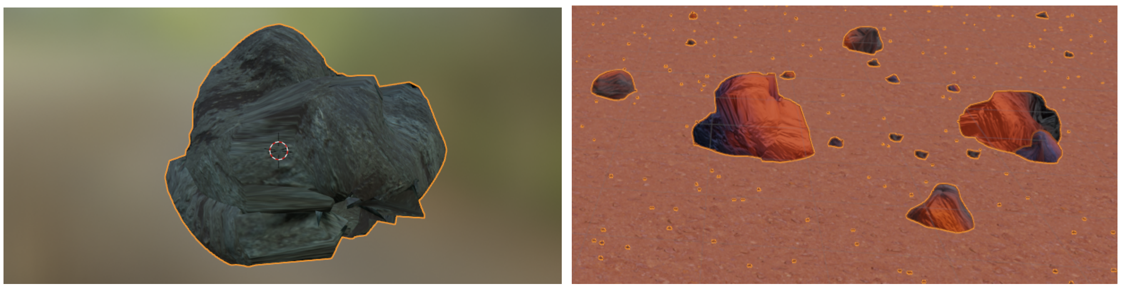


Figure 2.3: Examples of rocks created in Blender

Dust

A well known phenomenon on Mars is the presence of sand and dust which floats in the air and deposits over all the surfaces. As mentioned in section 2.1, the sand on the ground affects the odometry precision and efficiency, bringing out the need for a SLAM algorithm. Moreover, the dust on the rocks and features creates difficulties in the SLAM functioning,

[2]https://www.poliigon.com/

especially for sensors like the StereoCam.
The dust covering the rocks tends to equalize their external looking, shading out the particular textures and covering each of them with the same color shades. This creates difficulties in the recognition of landmarks and subsequently in the data association problem. This is the reason why also a Lidar sensor has been inserted in the work of this thesis, allowing the SLAM to run a geometrical data association, independent from the colors and shades of the rocks.
The dust has been recreated by means of the nodes in the *Shading* section of Blender, simulating a reddish thin surface covering part of the rocks.

Final result

Once the base surface and its texture were ready, together with the rocks and their textures, a final merge of the two elements was needed in order to achieve the final result. Blender has the useful function of treating a particular mesh, or a group of meshes, as particles to be spread over another mesh. By means of this function it was possible to treat rocks as particles to be spread along the base plane surface while randomizing their size, orientation and number. In this way it has been possible to recreate a large number of big and small stones starting from few rocks meshes, bringing out the final result in figure 2.4.

2.2.2. Unreal Engine

Blender proved itself as a useful and features rich software, with great capabilities for scene rendering. The Mars surface created through section 2.2.1 is detailed and the textures are enough detailed for the requirements needed in this work. However, the thesis focuses on a simulation of the rover movement and its acquisitions in this environment, so a way to realize it has to be seek.
A first idea was to operate through the Python console in Blender in order to coding the movement of the StereoCam inside the scene, being so able to acquire images and somehow simulate the Lidar beams and acquisitions. This procedure, however, proved itself to be very challenging and showed several technical problems which were far from a solution.
In this situation Unreal Engine, from Epic Games, came as a possible solution to link the static environment created in Blender to an active simulation environment. This is due to the fact that Unreal Engine can be easily linked to both Blender and Simulink, revealing

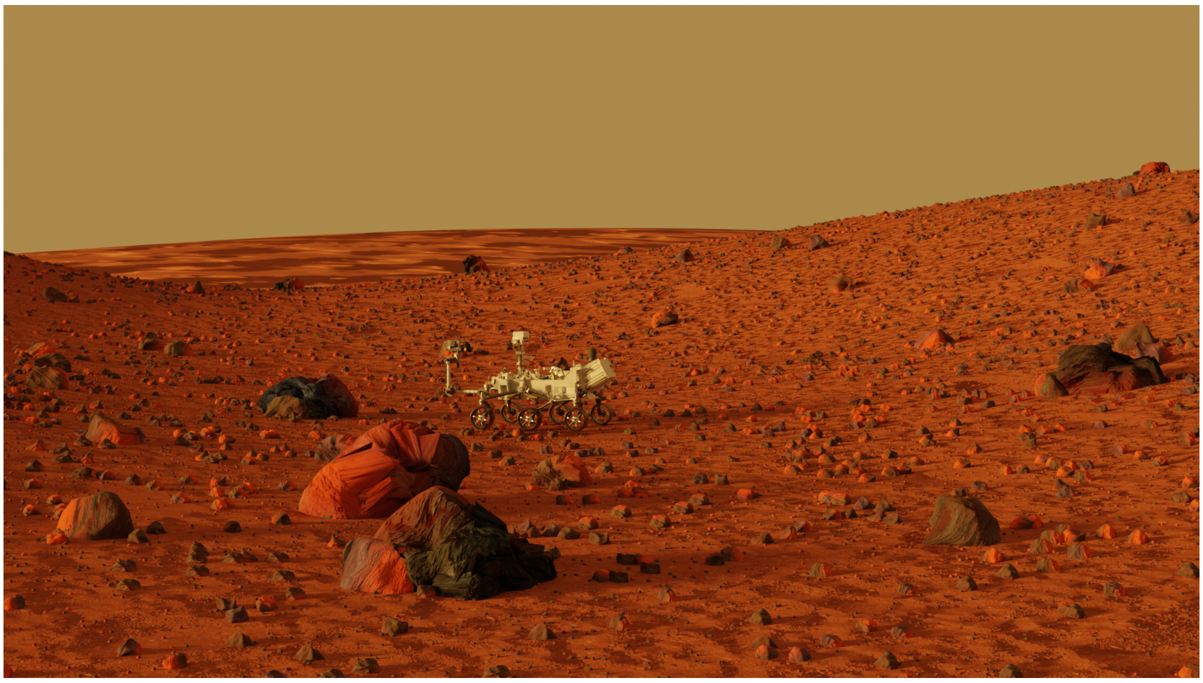


Figure 2.4: Final render of Mars surface (detailed version with hills and slopes)

itself as a useful simulation tool. In the following sections the properties of Unreal Engine are going to be explained in a deeper way.

Why Unreal Engine?

As stated above, Unreal Engine comes with the possibility of linking Blender and Simulink through it. Two are the main reasons why the decision of looking for such a property was taken: firstly, the environment in Blender was already created, so even if in Unreal Engine there is the possibility to create environments, the work already done was not going to be wasted; secondly, the capabilities of Blender in rendering are higher than in Unreal Engine. The reason behind the search of a link from Blender to Simulink is due to the fact that Simulink proves itself to be one of the most powerful simulation tools in commerce.

Unreal Engine capabilities

Unreal Engine is a software born for the creation and implementation of video games. It is provided by Epic Games, known as one of the major producers in the sector, but what is the reason behind the choice of a video game development software?
Unreal Engine can be upgraded with a library available on the Mathworks website, which links a simulation built in Simulink with an environment in Unreal Engine, moving an

actor through the digital environment while following the rules dictated by the Simulink model.
It so enables the possibility to move a rover on the surface created in Blender and exported to Unreal Engine, following the simulation parameters set in Simulink.

2.2.3. Simulink

As said before, the simulation requires not only a good simulaton of the surface to move on, but also a precise and efficient simulation of the rover movement and its sensor acquisitions. A powerful simulation tool is given by Simulink, from MathWorks.
Simulink can be linked with Unreal Engine in order to achieve really good performances.

Why Simulink?

The main reason behind the choice of Simulink is the possibility to connect it with Unreal Engine, allowing to simulate the dynamics and the sensors in the reconstructed environment. This can be done by means of the Automated driving toolbox, which is going to be presented in section 2.2.3.
In a similar way also the sensors, the Lidar and the StereoCam, can be simulated and the sensor acquisition can be stored in a Simulink variable.
The possibility to store the data of the simulation in Simulink allows then to exploit them as Matlab variables and work on them in a Matlab environment. The illustrated workflow allows a continue back and forth from the simulated Mars environment to a Matlab environment in which to study the data acquired.

Automated Driving Toolbox

As mentioned before, the link between Unreal Engine and Simulink is done by means of the Automated Driving Toolbox. This toolbox was originally made for simulations of self-driving cars in common urban environments. However, it is provided with some blocks and functions which allows simulations in different environments, either pre-made or customized. A similar function in also contained inside the *Aerospace Blockset*.
In the case under study, the different parameters for the link have been set to operate in a custom environment as it is going to be shown in the following section.

2.2.4. Simulink-Unreal Engine link configuration

In order to create a link between the Simulink and Unreal Engine simulations it is necessary to insert some configuration blocks inside the Simulink model which is going to be used for the actual simulation.
The blocks are at least three and are comprehensive of a block linking the two files, one block for the movement of the rover and one or more blocks for the sensors configuration. Respectively, these blocks are divided in three categories: configuration blocks, actor blocks and sensors blocks.
The simulation can be built in a less or more detailed way depending on the number of blocks inserted and the parameters to be configured in each of them. Even with the presence of the mentioned blocks, the simulation link between the two software is not trivial and several iterations were necessary before the correct setup was found.

Map

As explained in section 2.2.2, the map of the environment in which to simulate the rover path and acquisitions has been imported in Unreal Engine from a Blender file.
Once the file is imported, some features must be reorganized in order to keep the shape and the textures of the original Blender surface. After these few steps the file must be saved and from now on it will be ready for the simulation.

Actors

Fundamental in the structure of the simulation is the Actor: inside Unreal Engine the actor is the entity moving during the simulation and in the case under study it is the rover itself.
Unreal Engine requires a concrete object in the scene to be the set as the Actor, so a 3D model of the rover has been imported in the scene. The 3D model is the one of the rover Perseverance, currently working on the surface of Mars, and it was taken from the Jet Propulsion Laboratory free resources [3].
Once the object selected as the Actor is in the scene, the Actor has to be configured to move according to the Simulink simulation inputs. This task is achieved in Unreal Engine in the *Blueprint section*, while in Simulink by the *simulation 3D Actor Transform Set block*. An detailed description of the setup is beyond the purpose of this explanation. Figure 2.5 shows the Simulink block and the Unreal Engine section mentioned.

[3]https://mars.nasa.gov/resources/25042/mars-perseverance-rover-3d-model/

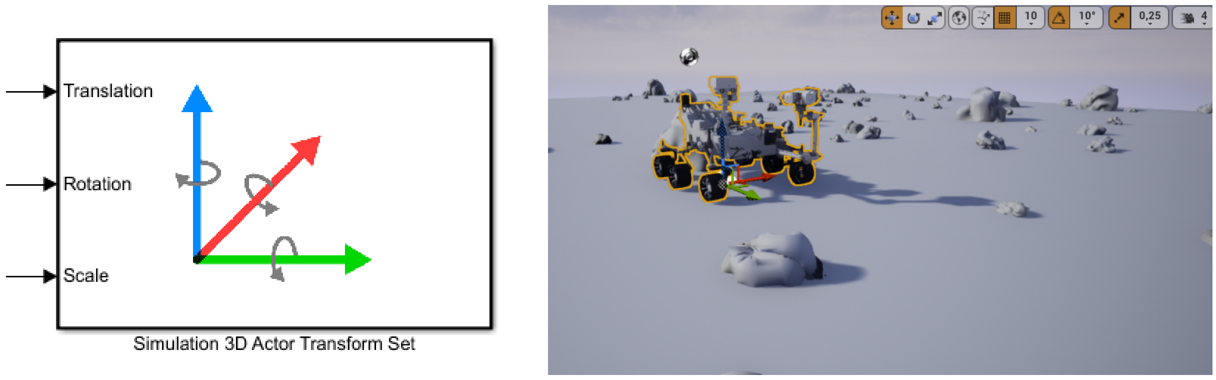


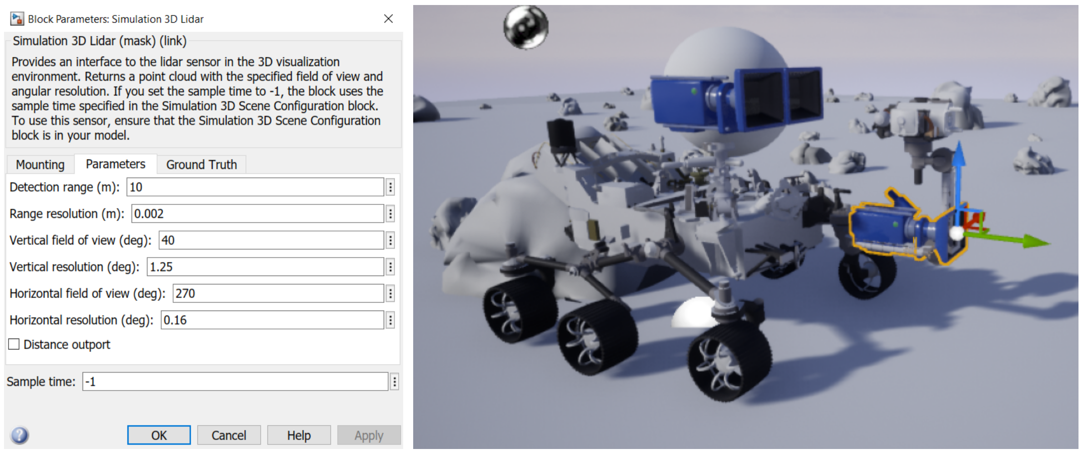
Figure 2.5: Simulation 3D Actor Transform Set block and Unreal Engine actor

Sensors

This thesis is built around the acquisitions from two main sensors mounted on the rover: one StereoCam and one Lidar scanner.
As already mentioned in section 2.2.2, the use of Unreal Engine comes from the need of modelling detailed and efficient sensors. The software, together with Simulink, provides the possibility to use already existing elements in it to built the sensors that are needed. It is so possible to obtain detailed and realistic data streams from the simulated sensors allowing an in depth analysis of the problem under study.
As for the Actor in the previous section, also the two sensors are built by pairing a block in the Simulink environment together with some elements in Unreal Engine.
The following sections are going to explain better the setup of the two sensors. The raw data collected by each of the sensors are collected in Simulink variables which are then stored in the Matlab workspace in order to be analyzed and elaborated.

Lidar

The Lidar is the simplest sensor between the two chosen to be reproduced, because of the presence of a Lidar block in the Automated Driving Toolbox which is linked directly to a Lidar entity in Unreal Engine. The parameters of the Lidar, together with the constraint to move with the rover are then set.
While the parts relating to the movement of the sensor during the simulation are set inside Unreal Engine, the Lidar intrinsic parameters are set inside the Simulink block, as shown in figure 2.6, where the Lidar Simulink setup is shown on the left, while the Lidar entity in Unreal Engine i shown on the right.


Block Parameters: Simulation 3D Lidar

Simulation 3D Lidar (mask) (link)

Provides an interface to the lidar sensor in the 3D visualization environment. Returns a point cloud with the specified field of view and angular resolution. 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.

Detection range (m): 10
Range resolution (m): 0.002
Vertical field of view (deg): 40
Vertical resolution (deg): 1.25
Horizontal field of view (deg): 270
Horizontal resolution (deg): 0.16
☐ Distance outport

Sample time: -1

Figure 2.6: Lidar sensor in Simulink and Unreal Engine

As in figure 2.6, the parameters set for the Lidar are summarized also in the table below:

horizontal f.o.v.	270°
vertical f.o.v.	40°
detection range	10 m
range resolution	0.002 m

Table 2.1: Lidar parameters

StereoCam

The second sensor chosen is a StereoCam. A StereoCam is a sensor made of two cameras coupled in order to reproduce the principles with which the human eye works and so obtain information about the depth of the scene at which the camera is looking at. Both Unreal Engine and Simulink do not have a StereoCam sensor ready, however they have the possibility to insert a monocular camera. By using two monocular cameras and working on their parameters setup and position setup in the scene, it has been possible to reproduce a StereoCam with good results. The StereoCam parameters and geometry has been set to reproduce the ones of the NavCams on-board the rover Perseverance [8]. To reproduce the StereoCam, two monocular cameras has been put inside the simulation, with same intrinsic parameters and a fixed distance in Unreal Engine.

It has to be highlighted that the StereoCam needs a particular calibration procedure, based on a calibration with a checkerboard. This procedure may sound unusual when in simulated environment, however it is necessary when dealing with the stream of data from the sensor. This calibration procedure has been done by means of a virtual checkerboard in Blender and Matlab and it is described in detail in the next section. Figure 2.7 shows the implementation of the StereoCam both in Simulink and Unreal Engine.

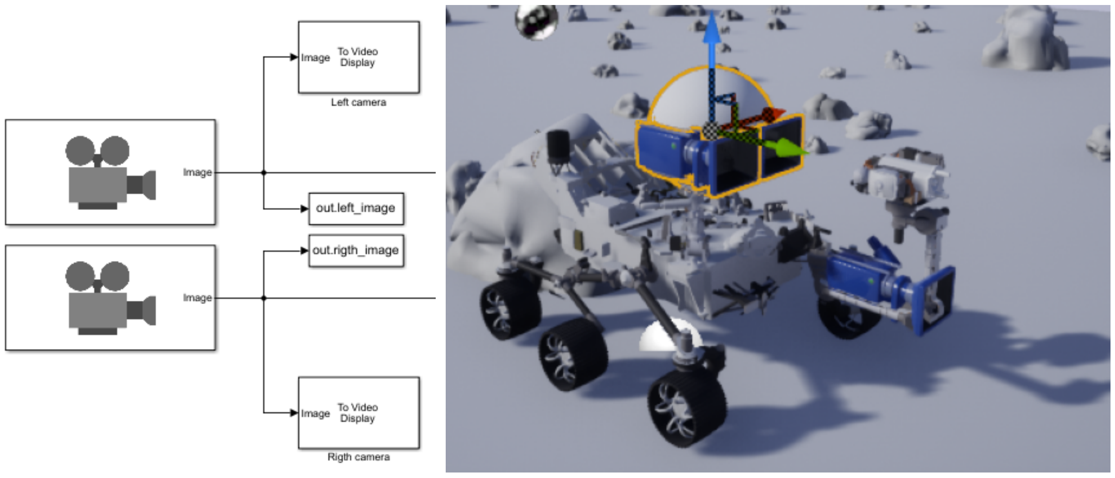


Figure 2.7: StereoCam sensor in Simulink and Unreal Engine

The parameters set for the StereoCam are:

horizontal f.o.v.	96°
vertical f.o.v.	73°
best focus	3.5 m
stereo baseline	42.4 cm
pixel format	600 x 400
mount height	1.5 m

Table 2.2: StereoCam parameters

As already said, the parameters are taken for the Perseverance rover specifications. However, the pixel format has been reduced to 600 x 400 in order to reduce the computational cost and duration on the simulation. It can be changed and incremented easily on the Simulink respective blocks once more computational resources are available.
As for the Lidar, the data stream of the two images from both the cameras composing the StereoCam is saved in a variable which then is exported to the Matlab workspace for further elaboration. It is also possible to visualize the StereoCam output live while the simulation of the rover movement and acquisition is going on. This is shown in chapter 5.

StereoCam calibration

The StereoCam needs a specific calibration in order to find the intrinsic parameters, which are going to be essential in the elaboration of the data stream and in the transformation of the images into point clouds.
The calibration procedure consists in acquiring a bunch of images from the StereoCam while observing a plane black and white checkerboard put in different orientations in front of the camera. By doing so, the calibration algorithm is able to find correspondences between the images seen from the left and right camera and so giving a value to intrinsic parameters such the stereo baseline or the lens distorsion. The calibration is needed because of the small imperfections a camera can accumulate during the building, so being with parameters slightly different from the nominal ones.
The calibration has been done by reproducing a StereoCam in Blender with the same specifics as the one in the simulation. Then several pictures of a checkerboard made in Blender and put in front of the camera in different orientations were acquired. The collection of pictures has been saved in two different folder, one with acquisitions from the left camera and one with acquisitions from the right camera.
Then by means of the *stereoCalibrator App* in Matlab the code for the parameters computation has been generated and the results computed, as in figure 2.8.

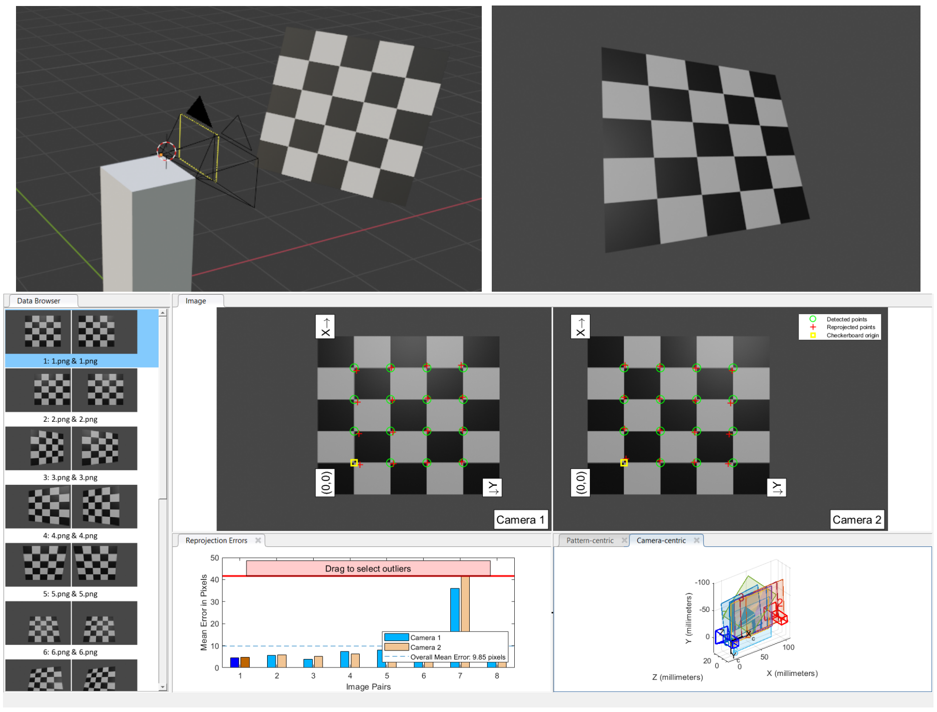


Figure 2.8: StereoCam calibration

Rover trajectory

Once the sensors setup is configured in the right way and the simulation link between Simulink and Unreal Engine is set, it is possible to start simulate the rover movement across the environment.

The rover trajectory is selected by the operator inside the Matlab workspace by manually drawing the desired trajectory itself over a bird-eye view map of the simulation environment. This approach aims at reproducing the rover following a previously decided path, which can be dictated both by a human operator or by some trajectory choice and optimization technique.

The approach is well illustrated on MathWorks website and some of the codes helping to do so are directly taken from the website. The functions taken help to create a user friendly interface for the trajectory drawing and also help at smoothing the path drawn while saving the poses coordinates. The trajectory drawing procedure follows the steps below:

Algorithm 2.1 Rover trajectory drawing

```
1: Take a bird-eye view of the simulation environment from Unreal Engine;
2: Import the bird-eye map in Matlab and set the map dimensions;
3: Launch the poses drawing command;
4: Draw the trajectory and save it to the workspace;
5: if no trajectory has been drawn then
6:     Import a default trajectory;
7: end if
8: Smooth the trajectory;
9: Set simulation time and compute the time vector;
10: Build the x,y,z poses as $nx2$ vectors with the time in the first column;
11: return poses X, poses Y, poses Rot
```

The following figure shows the bird-eye map and the trajectory drawn in Matlab:

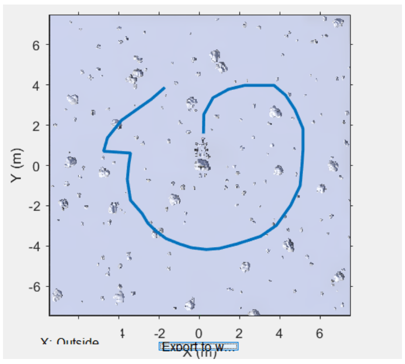


Figure 2.9: Rover trajectory drawing

Control inputs

Since the trajectory that has to be followed by the rover is selected and drawn manually while the poses are then saved in the workspace, the saved variables themselves are the starting point for the evaluation of the control input needed to move from one pose to the next one. In order to do that an *Odometry Motion Model* is adopted.
The odometry motion model is capable of computing the control input given at each time step in the form of a first rotation of the rover, a translation movement and a second rover rotation, respectively δ_{rot1}, δ_{trans} and δ_{rot2}.

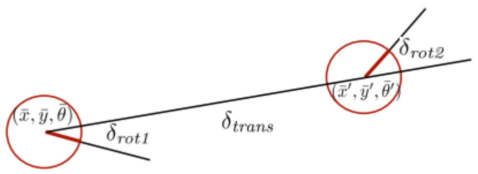

$(\bar{x}, \bar{y}, \bar{\theta})$
δ_{rot1}
δ_{trans}
$(\bar{x}', \bar{y}', \bar{\theta}')$
δ_{rot2}

Figure 2.10: Odometry motion model

The odometry motion model takes as inputs the poses drawn for the trajectory and then gives as a result the vectors containing the input commands for the whole trajectory.
Here the algorithm of the odometry motion model, as in [11], is shown:

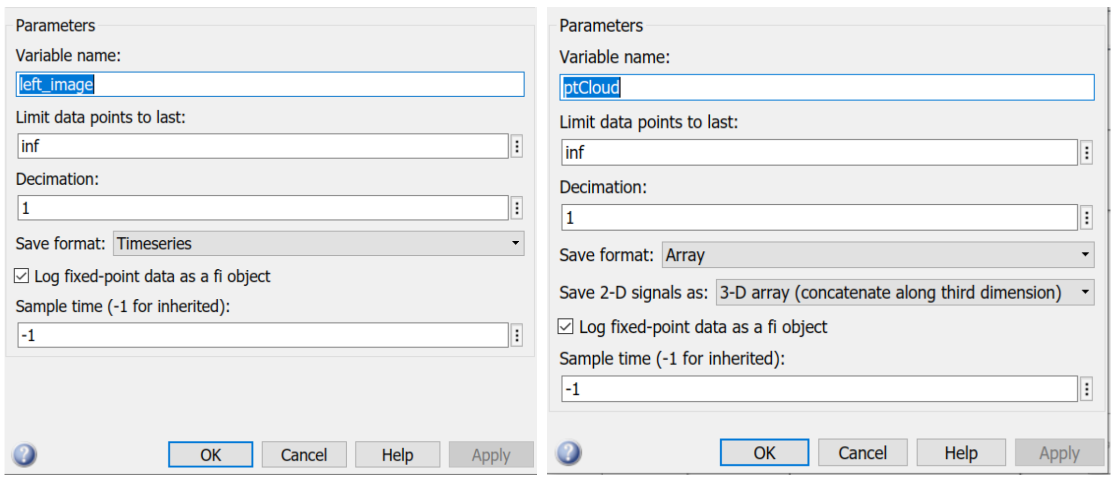
Algorithm 2.2 Odometry motion model

```
Require: poses X, poses Y, poses Rot
 1: $\delta_{rot1,1} = \arctan Y_2 - Y_1 X_2 - X_1 - Rot_1$
 2: $\delta_{trans,1} = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$
 3: $\delta_{rot2,1} = Rot_2 - Rot_1 - \delta_{rot1,1}$
 4: $n = length(X)$
 5: for i = 2:n do
 6:     $\delta_{trans,i-1} = \sqrt{(X_i - X_{i-1})^2 + (Y_i - Y_{i-1})^2}$
 7:     if $X_i > X_{i-1}$ then
 8:         $\delta_{trans,i-1} = \delta_{trans,i-1}$
 9:     else if $X_i < X_{i-1}$ then
10:         $\delta_{trans,i-1} = -\delta_{trans,i-1}$
11:     end if
12:     $\delta_{rot1,i-1} = \arctan Y_i - Y_{i-1} X_i - X_{i-1} - Rot_{i-1}$
13:     $\delta_{rot2,i-1} = Rot_i - Rot_{i-1} - \delta_{rot1,i-1}$
14: end for
15: return $u = \begin{bmatrix} \delta_{rot1,1:n-1} \\ \delta_{trans,1:n-1} \\ \delta_{rot2,1:n-1} \end{bmatrix}$
```

It must be noted that the dimension of the control input vector u is equal to $n-1$, where n is the length of the poses vector. This is justified by the fact that the controls start to activate from position one of the poses on.
Such generated control input u is going to be the control given to the rover while running the SLAM in order to make it follow the desired trajectory.

2.3. Sensor data elaboration

A brief digression on the type of data coming from the simulated sensor is worth at this point of the thesis in order to clarify what to expect and justify some of the steps which are going to be followed in the next sections.
The output type from the sensors can be set inside the relative Simulink blocks, as shown in figure 2.11:


Parameters
Variable name: left_image
Limit data points to last: inf
Decimation: 1
Save format: Timeseries
☑ Log fixed-point data as a fi object
Sample time (-1 for inherited): -1

Parameters
Variable name: ptCloud
Limit data points to last: inf
Decimation: 1
Save format: Array
Save 2-D signals as: 3-D array (concatenate along third dimension)
☑ Log fixed-point data as a fi object
Sample time (-1 for inherited): -1

Figure 2.11: StereoCam and Lidar output data types

As the figure shows, the data from the StereoCam is stored inside two *Timeseries* variables: one for the left camera and one from the right camera. At the same time, the data stream from the Lidar is stored inside a structure of point clouds, one for each of the acquisitions.
Regarding the StereoCam, in the next steps, a conversion of the data in point clouds is going to be needed together with the merging of the data coming from the two cameras. Meanwhile, since the Lidar data are already in the point cloud form, they are ready for the further elaborations in the second part of the simulation.

2.3.1. Sensor data to point clouds and landmarks

Algorithm 2.4 describes how the raw data from the sensors are transformed in useful data for the SLAM simulation.
To do so a division in the process between the Lidar data and the StereoCam data must be performed, since they require different elaboration techniques.

StereoCam data elaboration

As already mentioned in the previous section, the StereoCam data come in the form of two *Timeseries* variables, one for each of the two cameras.
The first step for the elaboration of the StereoCam data is to find the camera extrinsic and intrinsic parameters, step which is done with the calibration procedure described in section 2.2.4 and which is not going to be repeated here. The results from the calibration procedure are necessary in the pivotal following step: the *Disparity Map* computation.

The *Disparity Map* is a 2D map reduced from a 3D space, in which the brightness of the gray scaled pixels indicates the distance of the related points from the sensor. The description of how a disparity map is computed from a couple of stereo images is beyond the purpose of this thesis. However, it is important to note that it depends on the setup of a couple of parameters and it is computed in this work by means of a *Global Block Matching algorithm*. The parameters *Range* and *Uniqueness Threshold* are the ones to be set for finding the correct setup of the Global Block Matching algorithm. Several tries have been performed before finding the correct setup.
The disparity map computation is fundamental for evaluating the quality of the data generated and it is going to be necessary for the transformation of the images into useful point clouds.
By playing around with the *Range* and *Uniqueness Threshold* parameters, it has been possible to find a satisfying configuration for the Disparity Map computation, as shown in figure 2.12. The Disparity Maps computed at each step were then saved inside a structure variable.

After the Disparity Map computation, it is then transformed into a point cloud by a dedicated function. All the function and codes for this section are explained in Appendix A.2. Now both the Lidar and StereoCam data are organized in the form of structures of point clouds variables and their further elaboration can be done simultaneously.

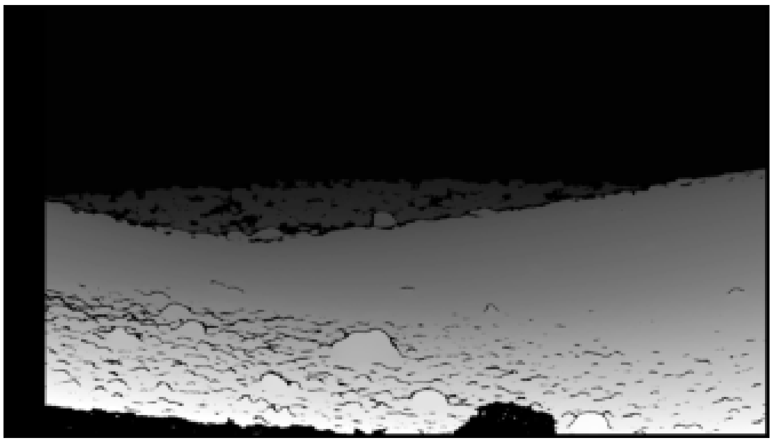


Figure 2.12: Disparity Map

Point clouds clustering in landmarks

Now that both the sensors data are in the form of points cloud, they can be elaborated in such a way to obtain cluster of points to be identified as landmarks. In order to that an approach based on the Euclidean distance between point, the angular distance between them and the number of points in each cluster has been adopted. In particular:

- *distThreshold* indicates the Euclidean distance under which the points of the same cluster must be;
- *angleThreshold* indicates the minimum angle of separation between two different clusters;
- *NumClusterPoints* indicates the minimum and maximum number of point accepted for a cluster.

Also in this case, the different tuning of the three parameters brings out different results in the clustering, with a number of landmark identified proportional to the strictness of the parameters chosen.
The outcomes from the different setups is illustrated in chapter 5, while figure 2.13 illustrates an example of clustering.
The figure shows another problem, which is going to be illustrated in detail in chapter 4: between the clusters it is possible to clearly distinguish parts of the rover as, for example, the wheels. These are the parts that need to be filtered out before proceeding with the Data Association.

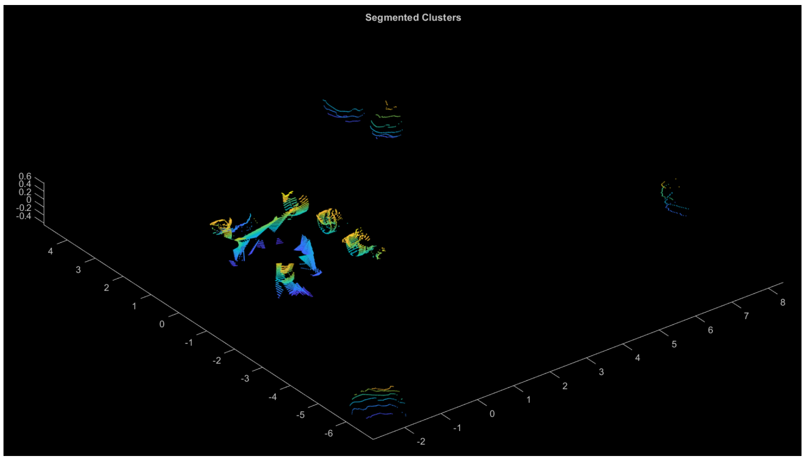


Figure 2.13: Clusters example

Once the clusters, or landmarks, are defined, it is possible to proceed with the identification of the landmark geometric centre and its coordinates.

Landmarks coordinates

The following step is a simple geometric computation of the center point coordinates of each of the clusters.
Each of the coordinates computed is then stored in the same structure where each point cloud is stored, in a dedicated field for the coordinates of each of the landmarks in every point cloud. At this point the set of the total number of acquired point clouds from the sensors is available, together with the coordinates in the rover local frame of the landmarks inside them.

2.4. Simulation Architecture

The previous sections illustrated in detail the softwares and the single task achieved by each of them in order to reach the final simulation architecture needed. Moreover, each of the softwares and the corresponding functions must be used in the correct order pending the correctness of the results.
For sake of simplicity, the simulation has been split into three different parts: the first part generates the data from the sensors moving along the trajectory with the rover, the second part manipulates the data in order to extract useful information and the third part is the actual Extended Kalman Filter SLAM implementation,
The first part of the simulation can be summarized as:

Algorithm 2.3 Simulation Part 1: sensor data generation

```
Require: Unreal Engine scene set, Simulink model set
 1: Open Matlab
 2: Run the main script and draw the trajectory
 3: Smooth the trajectory and generate control input $u$
 4: Open Simulink and set simulation time
 5: From Simulink open Unreal Engine
 6: In Unreal Engine open the simulation environment
 7: Run the simulation first in Simulink, then in Unreal Engine
 8: return $u$, StereoCam data, Lidar data and save them in a Matlab file
```

This procedure is the sequence of steps that brings from the virtual environment to a collection of useful data saved in a separated file, which can be easily uploaded and manipulated. The second part can be summarized as:

Algorithm 2.4 Simulation Part 2: sensor data to point cloud and landmarks

```
Require: StereoCam data, Lidar data
 1: Upload the Lidar and StereoCam data
 2: for Lidar data, StereoCam data do
 3:     Transform the data into one point cloud for each time step
 4:     for each point cloud do
 5:         Cluster the points inside the point clouds for distance and angular separation
 6:         Count the clusters (or landmarks)
 7:         Compute clusters geometric centre coordinates
 8:         Compute geometric centre orientation in the rover local frame
 9:     end for
10: end for
11: return Lidar Point Cloud and landmarks, StereoCam Point Cloud and landmarks
```

This part of the simulation operates manipulating the data collected in the section before, transforming the data from the sensors in 3D point clouds and identifying landmarks in them. The third part is the actual implementation of the SLAM and it is going to be described in detail is the following sections. Here a summary of its main steps is shown:

Algorithm 2.5 Simulation Part 3: Extended Kalman Filter SLAM

```
Require: *StereoCam point clouds*, *Lidar point clouds*, *u*
1: Upload the Lidar or StereoCam point clouds
2: for each time step do
3:     Actuate the control input
4:     Run the EKF Prediction step
5:     Run the Data Association
6:     Run the EKF Correction step
7: end for
8: return Results and plots
```

Algorithm 2.3 summarizes the whole simulation procedure illustrated in this section and it must be followed in order to obtain useful, meaningful and realistic data from the virtual sensors modelled.
Algorithm 2.4 and algorithm 2.5 are going to be explained in detail inside the next chapter as, especially for the second, they are the real core of the Extended Kalman Filter SLAM testing.
Figure 2.14 shows the Simulink file architecture, while figure 2.15 shows the whole simulation architecture.

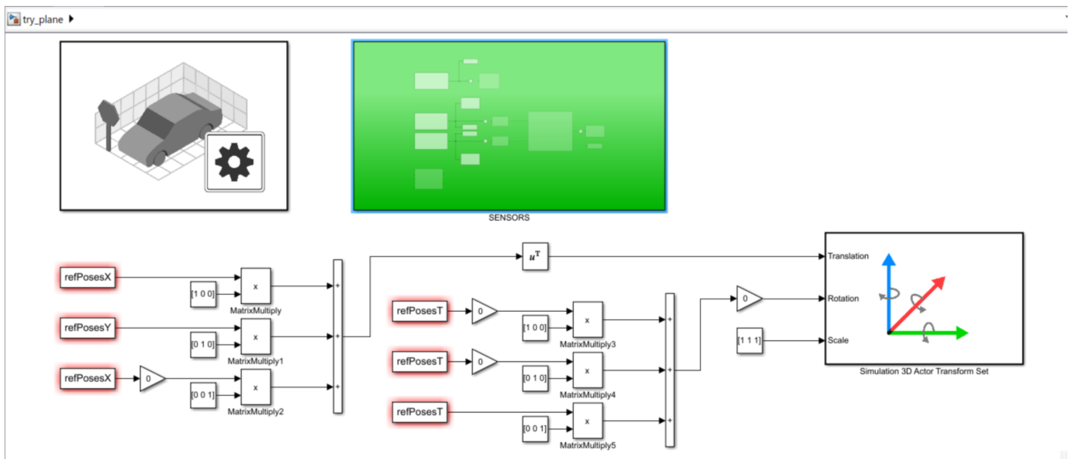


Figure 2.14: Simulink simulation model

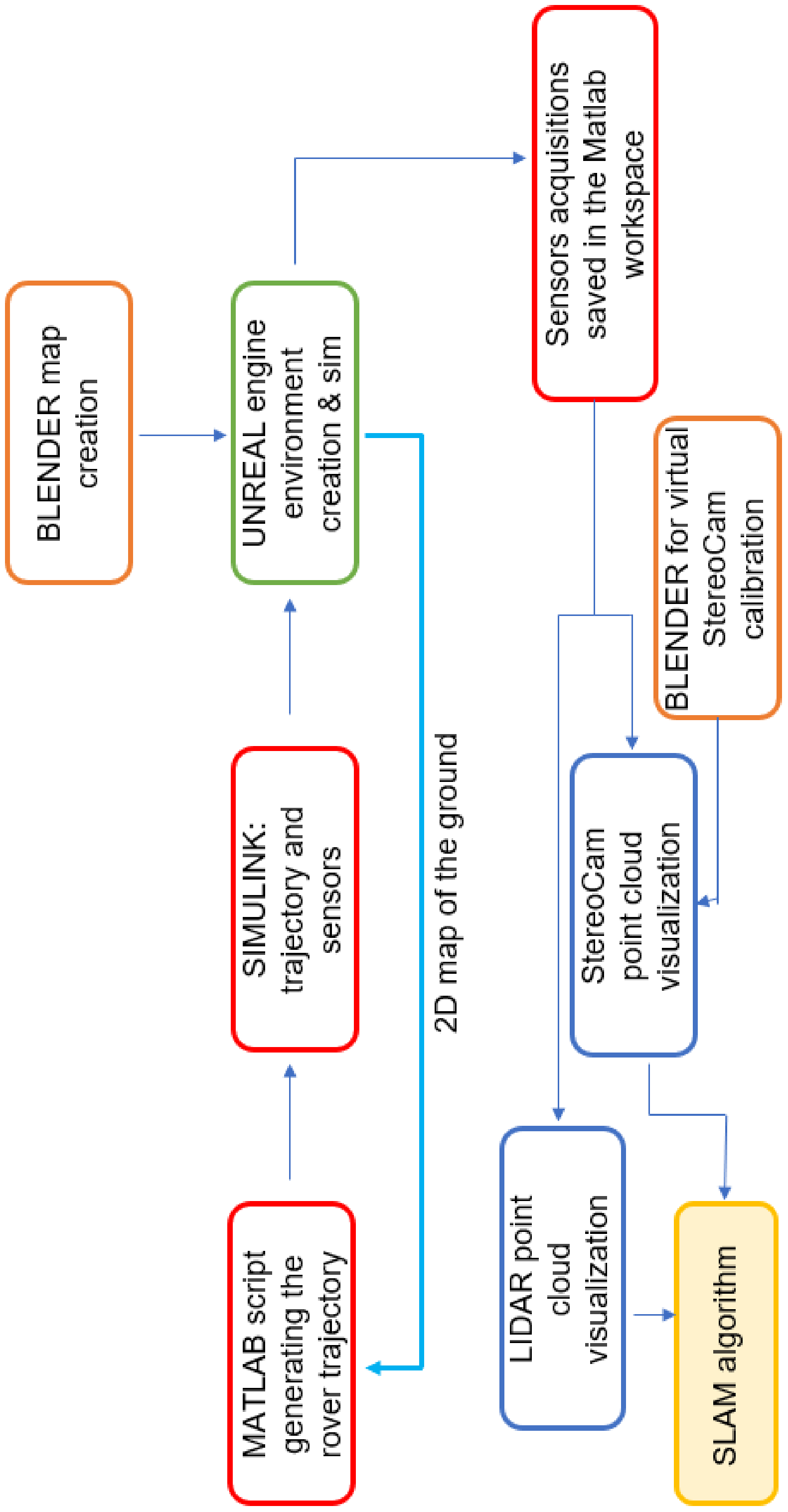

BLENDER map creation
UNREAL engine environment creation & sim
Sensors acquisitions saved in the Matlab workspace
SIMULINK: trajectory and sensors
MATLAB script generating the rover trajectory
2D map of the ground
BLENDER for virtual StereoCam calibration
StereoCam point cloud visualization
LIDAR point cloud visualization
SLAM algorithm

Figure 2.15: Simulation architecture

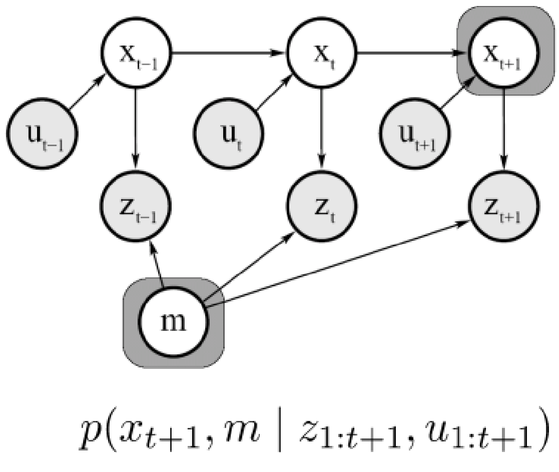
3 | Extended Kalman Filter SLAM

This chapter addresses the SLAM problem from a probabilistic point of view. Firstly, the probabilistic formulation of SLAM is illustrated, then the Kalman filters family is introduced in order to be used as a solution for SLAM in the core of the chapter.

3.1. Probabilistic SLAM

In chapter 1 the Simultaneous Localization and Mapping problem has been introduces as a problem where the robot does not have any previous knowledge of the environment, nor it knows its pose. All the available data are the control inputs $u_{1:t}$ and the sensor measurements $z_{1:t}$. In SLAM the robot acquires a map of its environment while simultaneously localizing itself relative to this map. As illustrated in chapter 1, SLAM can be formulated from a probabilistic point of view.

From a probabilistic perspective, the SLAM problem can be formulated in two different forms: *Online SLAM problem* and *Full SLAM problem*. The difference between the two is illustrated in section 1.2 and in [11]; graphically is represented in figure 3.1 and figure 3.2.



$$p(x_{t+1}, m \mid z_{1:t+1}, u_{1:t+1})$$

Figure 3.1: Online SLAM

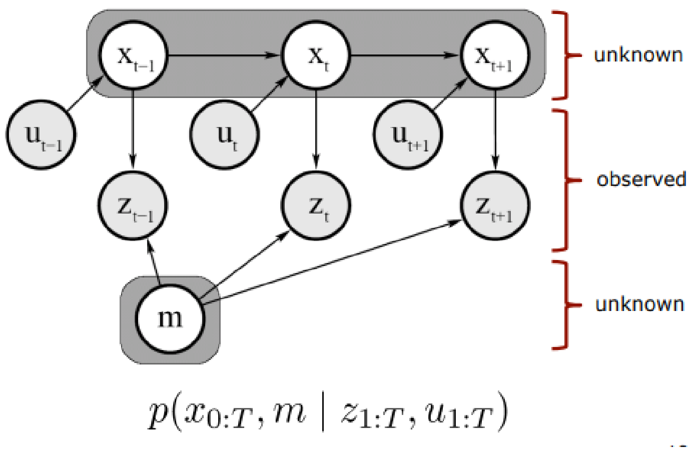


Figure 3.2: Full SLAM

The Online SLAM, figure 3.1, shows itself to be the result of integrating out past poses from the Full SLAM, figure 3.2. The Online SLAM involves estimating the posterior over the momentary pose along with the map:

$$P(x_t, m \mid z_{1:t}, u_{1:t}) \tag{3.1}$$

Here x_t is the pose at time t, m is the map and $z_{1:t}$, $u_{1:t}$ are respectively the measurements and the controls. The problem is called Online SLAM since it only computes the estimation of variables persisting at the current time t.
On the contrary, the Full SLAM computes an estimation over the entire path $x_{1:t}$ along with the map:

$$P(x_{1:t}, m \mid z_{1:t}, u_{1:t}) \tag{3.2}$$

Estimating the full posterior is the goal for SLAM in both cases, however calculating the full posterior is usually unfeasible due to the large number of discrete correspondences variables and high dimensionality of the continuous parameter space. This chapter is going to develop a solution algorithm for the Online SLAM problem.

A second key characteristic of the SLAM problem has to do with the nature of the estimation problem [11], since SLAM posses both a continuous and a discrete component. The continuous estimation problem is related to the robot own pose in the map and to the location of objects in it. Objects are called *Landmarks* in a feature-based SLAM representation. The discrete nature has to do with correspondence: when detecting a

landmark, the SLAM algorithm has to reason about the relation between this objects and the previously seen ones. This is typically a discrete operation: the object is either been seen before or not. The correspondence variable is going to be made explicit in the Data Association chapter.
Recalling section 1.2.3, the SLAM problem can be reformulated in two steps, allowing a recursive prediction-correction implementation form:

- Time-update:

$$P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) = \int P(x_k \mid x_{k-1}, u_k) P(x_{k-1}, m \mid Z_{0:k-1}, U_{0:k-1}, x_0) dx_{k-1} \tag{3.3}$$

- Measurement-update:

$$P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k \mid x_k, m) P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0)}{P(z_k \mid Z_{0:k-1}, U_{0:k}} \tag{3.4}$$

Equations 3.3 and 3.4 provide a recursive procedure for calculating the joint posterior in equation 1.1. This procedure is a function of a *Motion model* and of an *Observation model*, which can be linear or, in most real cases, non-linear.

The probably most influential solution of the probabilistic SLAM formulation is based on the Kalman filters family, especially on the Extended Kalman filter. The EKF SLAM algorithm illustrated applies the EKF to online SLAM introducing a *Known Data Association* assumption, which is going to be explained and relaxed in chapter 4.
The following sections are going to explain how a Kalman filter works, why we need its extended version and how they apply to the here illustrated SLAM formulation. In the next chapter the mathematical formulation of the Data Association is also explained.

3.2. The Kalman Filter

The Kalman filter is a recursive state estimator, member of the *Gaussian filters* family. It relies on the basic idea of that beliefs are represented by multivariate normal distributions [11]:

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} exp\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\} \tag{3.5}$$

This density over the variable x is characterized by two sets of parameters: the mean μ and the covariance Σ. The mean μ is a vector that posses the same dimensionality of the state x, while the covariance matrix Σ is a quadratic matrix that is symmetric and positive-semidefinite, with dimension of the state x squared. The number of elements of Σ depends quadratically on the number of elements in the state x. It is important to remember that Gaussians are uni-modal: they posses a single maximum.
The parameterization of a Gaussian by its mean and covariance is called *moments parameterization*. Other parameterizations do exist, with different algorithmic characteristics in the respective filters.

The Kalman filter was invented by Swerling and Kalman as a technique for filtering and prediction in linear Gaussians systems. It implements belief computation for continuous states and it is not applicable for discrete or hybrid state spaces.
The Kalman filter represent beliefs by the moments parameterization: at time t, the belief is represented by the mean μ_t and the covariance Σ_t. As already stated, they need Gaussians, which must fulfill the Markov assumption for Bayes filter and must have three other properties holding:

1. Markov assumption: the next state x_k depends on the previous state x_{k-1} and the applied control input u_k, and it is independent from both the observations and the map.

2. The state transition probability $p(x \mid u_t, x_{t-1})$ must be linear function in its arguments with added Gaussian noise. It is expressed by:

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t \tag{3.6}$$

   A_t is a square matrix of dimension n x n, while B_t is a matrix of dimension n x m, where n is the size of the state vector x_t and m the dimension of the control vector

u_t. Thus, the Kalman filter assumes linear system dynamics. The random variable ϵ_t is a Gaussian random vector that models the uncertainty introduced by the state transition, with zero mean and covariance notes as R_t.

3. The measurement probability $p_t(z_t \mid x_t)$ must also be linear in its arguments, with added Gaussian noise:

$$z_t = C_t x_t + \delta_t \tag{3.7}$$

   The matrix C_t has size k x n, where k is the dimension of the measurement vector z_t. The vector δ_t describes the measurement noise as a multivariate Gaussian with zero mean and covariance Q_t.

4. The initial belief $bel(x_0)$ must be normally distributed. The mean of this belief is denoted by μ_0 and its covariance by Σ_0:

$$bel(x_0) = p(x_0) = \det(2\pi\Sigma_0)^{-\frac{1}{2}} exp\{-\frac{1}{2}(x_0 - \mu_0)^T \Sigma_0^{-1}(x_0 - \mu_0)\} \tag{3.8}$$

These assumption ensure that the posterior belief $bel(x_t)$ is always Gaussian, for any point in time t. The mathematical proof of the properties above is beyond the purpose of this thesis and can be found in [11].

The Kalman filter algorithm can be represented as:

Algorithm 3.1 Kalman Filter

```
Require: $\mu_{t-1}$, $\Sigma_{t-1}$, $u_t$, $z_t$
1: $\bar{\mu}_t = A_t \mu_{t-1} + B_t \mu_t$
2: $\bar{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
3: $K_t = \bar{\Sigma}_t C_t^T (C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1}$
4: $\mu_t = \bar{\mu}_t + K_t(z_t - C_t \bar{\mu}_t)$
5: $\Sigma_t = (I - K_t C_t)\bar{\Sigma}_t$
6: return $\mu_t$, $\Sigma_t$
```

In steps 1 and 2, the predicted belief $\bar{\mu}$ and $\bar{\Sigma}$ is calculated representing the belief $\bar{bel}(x_t)$ one time step later, but before incorporating the measurement z_t. This belief incorporates the control input u_t following the rules dictated by linear systems dynamics.
In steps 3 to 5 this belief $\bar{bel}(x_t)$ is transformed into the desired belief $bel(x_t)$ through incorporating the measurement z_t.

The variable computed in step 3 is called *Kalman Gain*: it specifies the degree to which the measurement is incorporated into the new state estimate [11].

Step 4 manipulates the mean by adjusting it in proportion to the Kalman Gain, the measurement z_t and the measurement predicted. The important concept here is the *innovation*, or the difference between the actual measurement z_t and the expected measurement $C_t\bar{\mu}_t$. Then the covariance is updated in step 5.
Figure 3.3 shows the graphical illustration of the Kalman filter principles:

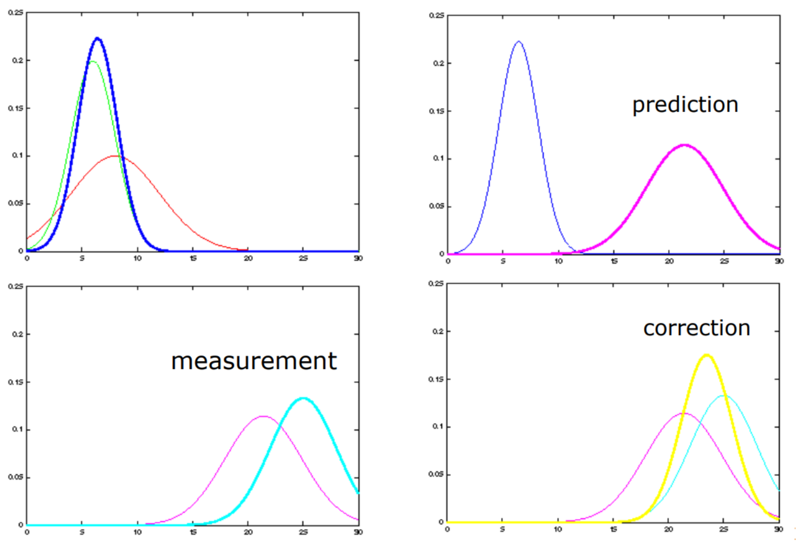


Figure 3.3: Kalman filter functioning

As mentioned, the Kalman filter alternates a prediction step and a correction step. By observing figure 3.3, it can be noted that the prediction step, incorporating the control input update, introduces uncertainty in the robot's belief. Meanwhile, the uncertainty in the robot's belief is decreased by the correction step, which introduces the measurement update.

A brief discussion over the meaning of the Kalman gain must be performed. The Kalman gain is computed in the form:

$$K_t = \bar{\Sigma}_t C_t^T (C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1} \tag{3.9}$$

Where $\bar{\Sigma}_t$ is the covariance after the control input incorporation, but before the measurement update, C_t is the matrix representing the linear dynamics including the measurement inside the update and Q_t is the matrix representing the measurement uncertainty.
Inside equation 3.9 is important to focus on the term:

$$(C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1} \tag{3.10}$$

This term is higher when the uncertainty over the measurement is lower, thus when Q_t is small, meaning that a precise and reliable sensor is present. This property translates in a negligible Q_t and so the Kalman gain becomes big as we have the inverse only of $(C_t \bar{\Sigma}_t C_t^T)^{-1}$, which usually brings high values. The consequence is that the contribution of the product for the Kalman gain in step 4 is very important and the final computation of the mean μ_t in step 4 is more driven by the *innovation* than from the $\bar{\mu}_t$ coming from the motion model. As an example, a perfect sensor would have a Q_t matrix with all null values, leading to:

$$\begin{aligned} Q_t &= 0 \\ K_t &= C_t^{-1} \\ \mu_t &= C_t^{-1} z_t \end{aligned} \tag{3.11}$$

On the contrary, higher uncertainty in the sensor brings higher values for Q_t, driving for a small and almost negligible K_t, with subsequently almost no contribution coming from the observation model to the mean μ_t in step 4. This results in a mean μ_t mainly based on the prediction model and only slightly corrected by the measurements. In the case of no trust in the information coming from the sensor:

$$\begin{aligned} Q_t &= \infty \\ K_t &= 0 \\ \mu_t &= \bar{\mu}_t \end{aligned} \tag{3.12}$$

The outcome of the algorithm can be very different pending on the values set on the measurement uncertainty matrix Q_t and so pending on how much trust can be placed on the sensors readings.

On the computational side, updating the Kalman filter with a control input, inside the prediction step, is relatively light. At the same time, the Kalman filter update based on the measurements is computationally more difficult and heavy.

3.3. The Extended Kalman Filter

The unavoidable assumption in the Kalman filter formulation is the linearity of both the motion model and the observation model. However, this is rarely the case when dealing with moving robots in real environments. More often, the two models are both non-linear in their formulation. As the Kalman Filter can not handle non-linearity, here enters the field the Extended Kalman Filter, which is an extension of the Kalman Filter itself.

Built on the base of the Kalman Filter, the Extended Kalman Filter (EKF) makes the assumption of *linearization* of the motion model and the measurement model functions. The key idea is that the non-linear function of the motion model $g(u_t, x_{t-1})$ is linearized by a tangent function to g at the mean of the Gaussian. The EKF also approximates the non linear function of the measurement update $h(x_t)$ by a linear function tangent to h, thereby retaining the Gaussian nature of the posterior belief. In order to achieve this linearization the EKF utilizes a *first order Taylor expansion*, as shown in figure 3.4. Once g and h are linearized, the mechanics of the EKF are equivalent to those of the Kalman filter.

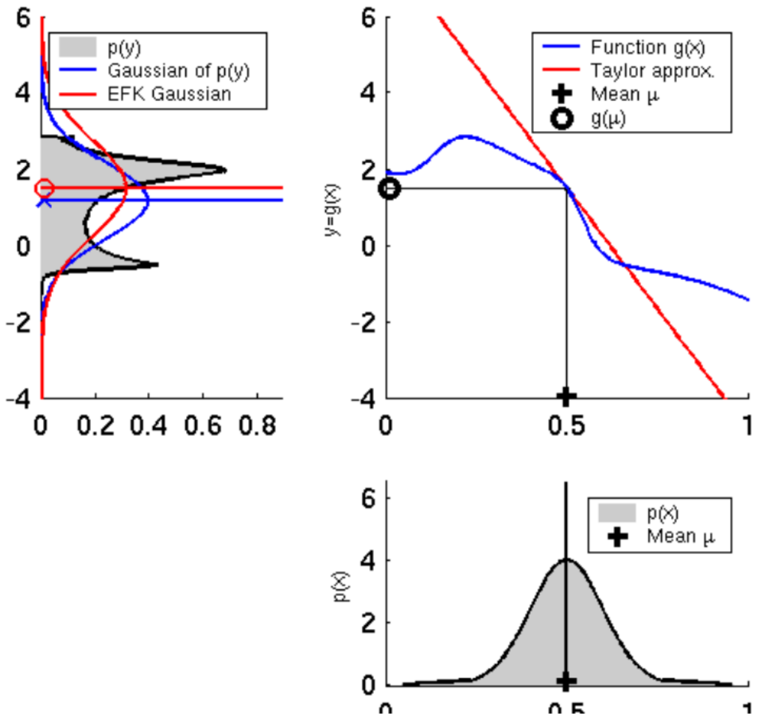


Figure 3.4: Graphical representation of linearization

The EKF algorithm is shown here, with G as the Jacobian of the motion model function g and H as the Jacobian of the measurement update function h:

Algorithm 3.2 Extended Kalman Filter

```
Require: $\mu_{t-1}$, $\Sigma_{t-1}$, $u_t$, $z_t$
1: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
2: $\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
3: $K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^{-1}$
4: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
5: $\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t$
6: return $\mu_t$, $\Sigma_t$
```

As can be noted the linear predictions in the Kalman filter are replaced by their non linear generalizations in the EKF. Moreover, the EKF uses Jacobians instead of the corresponding linear system matrices. The Extended Kalman filter inherits the basic belief from the Kalman filter, but it differs in that this belief is only approximate, not exact as in Kalman filters. A Gaussian approximation of the true belief is so computed by the Extended Kalman Filter [11].

	Kalman filter	Extended Kalman filter
state prediction	$A_t\mu_{t-1} + B_t u_t$	$g(u_t, \mu_{t-1})$
measurement prediction	$C_t\bar{\mu}_t$	$h(\bar{\mu}_t)$

Table 3.1: EKF and KF comparison

EKF Linearization

The main idea on which the EKF is based is the *linearization* and the key advantage lies in its efficiency. The top left graph of figure 3.4 represents a Monte-Carlo estimate of the Gaussian (blue line): it has been computed by passing a large number of points through g followed by the computation of their mean and covariance. On the other end, the EKF linearization only requires determination of the linear approximation, followed by the closed-form computation of the resulting Gaussian. Once g is linearized, the mechanics of the EKF are equivalent to the ones of the Kalman filter, as already stated.

The Extended Kalman filter applies a *First order Taylor expansion linearization*. It builds up a linear approximation of g starting from its value and slope. The slope is given by the partial derivative:

$$g^{'}(u_t, x_{t-1}) := \frac{\partial g(u_t, x_{t-1})}{\partial x_{t-1}} \tag{3.13}$$

As can be noted, both the value and the slope depends on the argument of g. A logical choice for selecting the argument is to choose the state deemed most likely at time of linearization [11]. For Gaussians, the argument is going to be the mean of the posterior μ_{t-1}. In other words, g is approximated by its values at μ_{t-1}, and at u_t, and the linear extrapolation is achieved by a term proportional to the gradient of g at μ_{t-1} and u_t:

$$\begin{aligned} g(u_t, x_{t-1}) &\approx g(u_t, \mu_{t-1}) + g^{'}(u_t, \mu_{t-1})(x_{t-1} - \mu_{t-1}) \\ &= g(u_t, \mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) \end{aligned} \tag{3.14}$$

Notice that G_t is a matrix of size n x n, with n denoting the dimension of the state. This is the Jacobian of g and its value depends on u_t and μ_{t-1}, hence it differs for different points in time. The exact same procedure is applied for the linearization of the measurement function h.
The EKF is one of the most popular tools for state estimation in robotics and its strength lies mainly in its computational efficiency. Each update requires time of the order $O(k^{2.4} + n^2)$, where k is the size of the measurement vector z_t and n is the dimension of the state vector x_t. Other known algorithms, as particle filters, may require time exponential in n. However, one important drawback of the EKF is that it approximates state transitions and measurements using linear Taylor expansion. The goodness of the linearization depends on two main factors: the degree of uncertainty and the degree of local non-linearity of the functions that are being approximated. The higher is the uncertainty, the more distorted is the Gaussian density computed after the Taylor linearization. Higher uncertainty typically results in less accurate estimates of the mean and the covariance of the resulting random variable. The second important factor is the local non-linearity of the function g: the more non-linear is the region of g in which the mean falls, the larger the approximation error results. The Extended Kalman filter Gaussian clearly underestimates the spread of the resulting density.

Now that the general derivation and structure of the Kalman filter and the Extended Kalman filter has been illustrated, their application to SLAM, in particular of the extended version, is explained in detail inside the next section.

3.4. Extended Kalman Filter SLAM

In this section a version of the Extended Kalman Filter which applies to the online SLAM problem is going to be illustrated. The EKF solution to the SLAM problem is probably the most relevant in the history of the subject.
The EKF SLAM has three intrinsic assumptions which bring to approximations:

1. *Feature-based SLAM*: the maps are going to be composed by point landmarks with a number usually minor than a thousand. The landmarks are going to need less ambiguity as possible, involving advanced computer vision tools, as better explained in chapter 4;

2. *Gaussian noise assumption*: this assumption is made for both motion and perception, with a relative small uncertainty in order to avoid linearization intolerable errors;

3. *Positive information*: EKF SLAM can only process positive sights of landmarks, meaning that it can not process the absence of a landmark as a consequence of the Gaussian belief representation.

In this section another assumption is introduced, which is the assumption of *Known Correspondeces*, which is going to be relaxed in chapter 4. This assumption implies that for each landmark observed, the algorithm exactly knows whether it is a new landmark or a re-observed one. By introducing the assumption of Known Correspondeces, the Data Association problem is currently set apart, allowing the focus to be only on the SLAM functioning itself. The assumption is going to be contained in the variable c_t^i, where c indicates the exact correspondence of an observed landmark i at time t. This assumption also allow to address only the continuous portion of the SLAM problem.

The Extended Kalman filter SLAM computes both the robot pose $x_t = (x, y, \theta)^T$ and the the estimate of the coordinates of all landmarks encountered. This makes it necessary to include the landmarks coordinates into the state vector. Let's call for one moment the state vector containing both the robot pose and the landmarks coordinates as *combined state vector*, written as y_t. It is composed in the form:

$$\begin{aligned} y_t &= \begin{bmatrix} x_t \\ m \end{bmatrix} \\ &= (x, y, \theta, m_{1,x}, m_{1,y}, s_1, ..., m_{N,x}, m_{N,y}, s_N)^T \end{aligned} \tag{3.15}$$

The variables x, y and θ represent the robot coordinates at time t, while $m_{i,x}$, $m_{i,y}$ are the coordinates of the i-th landmark and s_i its signature. The dimension of this state vector is $3N + 3$, where N denotes the number of landmarks in the map. Clearly, for any reasonable number N, this vector is significantly larger than the pose vector x_t itself.

The EKF SLAM algorithm is shown in algorithm 3.3. With respect to the EKF algorithm in the previous section, some new terms and elements have been introduced. In particular, steps from 1 to 4 belong to the so called *Prediction Step*, while steps from 5 to the end belong to the *Correction Step*. The two steps are going to be described in detail respectively in section 3.4.3 and section 3.4.4.
The same subdivision of the algorithm has been kept also in its Matlab implementation. Moreover, another section is going to be added inside the implementation in order to tackle the Data Association problem.

3.4.1. Motion model

As already mentioned, the SLAM algorithm needs to be fed with a *Motion model* describing how to update the rover state based on the control input received step by step.
For this simulation an *Odometry Motion model* has been implemented. Recalling that the control input is in the form $u_t = (\delta_{rot1}, \delta_{trans}, \delta_{rot2})^T$, the motion model is then derived from it:

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} \delta_{trans} \cos(\theta_{t-1} + \delta_{rot1}) \\ \delta_{trans} \sin(\theta_{t-1} + \delta_{rot1}) \\ \delta_{rot1} + \delta_{rot2} \end{bmatrix} \tag{3.16}$$

Which can be summarized as:

$$x_{t,1:3} = x_{t-1,1:3} + g(x, y, \theta, \delta_{trans}, \delta_{rot1}, \delta_{rot2}) \tag{3.17}$$

The EKF needs also the Jacobian matrix G of the odometry motion model to be computed, resulting in the following equation:

$$G = \begin{bmatrix} 1 & 0 & -\delta_{trans} \sin(\theta_{t-1} + \delta_{rot1}) \\ 0 & 1 & \delta_{trans} \cos(\theta_{t-1} + \delta_{rot1}) \\ 0 & 0 & 1 \end{bmatrix} \tag{3.18}$$

3.4.2. Measurement model

As for the motion model, also the sensor readings need to pass through a *Measurement model* and its Jacobian. The sensor measurement update function h corresponding to the i^{th} measurement at time t is:

$$h(\bar{\mu}_t, j) = z_t^i = \begin{bmatrix} r_t^i \\ \phi_t^i \end{bmatrix} = \begin{bmatrix} \sqrt{(\bar{\mu}_{j,x} - \bar{\mu}_{t,x})^2 + (\bar{\mu}_{j,y} - \bar{\mu}_{t,y})^2} \\ \arctan\left[(\bar{\mu}_{j,y} - \bar{\mu}_{t,y}), (\bar{\mu}_{j,x} - \bar{\mu}_{t,x})\right] - \bar{\mu}_{t,\theta} \end{bmatrix} \tag{3.19}$$

Where $(\bar{\mu}_{j,x}, \bar{\mu}_{j,y})^T$ is the pose of the j^{th} landmark and r_t^i and ϕ_t^i are respectively the range and the bearing of the landmark measured. The Jacobian H is computed by exploiting the properties of the various matrices introduced in algorithm 3.3, more precisely from step 11 to step 15. The mathematical derivation of the matrix H is here illustrated.
Since h depends only on the robot pose x_t and the location j^{th} of landmark m_j, the derivative factors into a low-dimensional Jacobian h_t^i and a matrix $F_{x,j}$, illustrated in step 14 of algorithm 3.3, which maps h_t^i into a matrix of the dimension of the full state vector:

$$H_t^i = h_t^i F_{x,j} \tag{3.20}$$

Here h_t^i is the Jacobian of the function $h(y_y, j)$ at $\bar{\mu}_t$, calculated with respect to the state variables x_t and m_j:

$$h_t^i = \frac{1}{q} \begin{bmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x & 0 \\ 0 & 0 & 0 & 0 & 0 & q \end{bmatrix} \tag{3.21}$$

Where:

$$q = \begin{bmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{bmatrix}^T \begin{bmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{bmatrix} = \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix}^T \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} \tag{3.22}$$

Here $j = c_t^i$ is the landmark that corresponds to the measurement z_t^i. The matrix $F_{x,j}$ is of dimension 6 x $(3N + 3)$ and maps the low-dimensional matrix h_t^i into a matrix of dimension 3 x $(3N + 3)$.

Thanks to this derivation of the motion and measurement model, together with their Jacobians, their computation can be easily performed inside the EKF SLAM algorithm.

3.4.3. Prediction step

Steps 1 to 4 compose the Prediction Step, in which the mean is updated in the three rover global coordinates representing the rover state:

$$x_{t,1:3} = (x, y, \theta)^T \tag{3.23}$$

This is an update only due to the effects of the control input received u_t, without considering the measurements vector z_t.
In step 1 a matrix F_x is built and introduced:

$$F_x = \begin{bmatrix} 1 & 0 & 0 & 0...0 \\ 0 & 1 & 0 & 0...0 \\ 0 & 0 & 1 & 0...0 \end{bmatrix} \tag{3.24}$$

This is a 3 x $(3N+3)$ matrix, where N is the number of landmarks, with all zero elements except for the 3 x 3 initial identity matrix. This matrix is introduced in order to exploit its structures inside later matrix multiplication with the aim to map only the rover state, avoiding the modification of the landmarks states when present.
Step 2 introduces the computation of an only *motion predicted mean* $\bar{\mu}_t$, adding the result of the motion equations on the three rover coordinates over the previous mean μ_{t-1}:

$$\bar{\mu}_t = \mu_{t-1} + F_x^T (g_x, g_y, g_\theta)^T \tag{3.25}$$

Step 3 computes the Jacobian G_t of the linearized motion function g, by means of matrix multiplication and the matrix F_x:

$$G_t = I + F_x^T \begin{bmatrix} 0 & 0 & \frac{\partial g}{\partial x} \\ 0 & 0 & \frac{\partial g}{\partial y} \\ 0 & 0 & \frac{\partial g}{\partial \theta} \end{bmatrix} F_x \tag{3.26}$$

At the end of the prediction, step 4 updates the *motion predicted covariance* $\bar{\Sigma}_t$ by the Jacobian G_t, the previous covariance Σ_{t-1} and the motion error matrix R_t:

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + F_x^T R_t F_x \tag{3.27}$$

At this point a motion-updated mean vector and a motion-update covariance matrix are available: in case of perfect motion model they should be the correct position of the rover in the global reference, without the need of a sensor update.

3.4.4. Correction step

Since the motion model can never be perfect and it is affected by errors, i.e. coming from the odometry measurements, the SLAM algorithm corrects the state of the rover by observing the surrounding landmarks in the Correction step.
This section of the algorithm articulates itself from step 5 to the end and, as said before, is here based on the assumption of perfect known Data Association in the variable c_t^i.
Step 5 introduces the sensor measurement error matrix Q_t, a diagonal 3 x 3 matrix in which σ_r^2 indicates the error over the range measurement, σ_ϕ^2 the error over the bearing measurement and σ_s^2 the error over the landmark specific signature.

Step 6 opens the core of the Correction step: a FOR loop in which for every landmark observed at step t the algorithm decides if it is whether an already observed landmark or a new one and updates the state and covariance according to it.
In step 7 the association of the landmark is explored and if the landmark under study has never been seen before, steps 8 to 10 initialize the landmark adding its measurements to the state vector:

$$\begin{bmatrix}\bar{\mu}_{j,x}\\ \bar{\mu}_{j,y}\\ \bar{\mu}_{j,s}\end{bmatrix} = \begin{bmatrix}\bar{\mu}_{t,x}\\ \bar{\mu}_{t,y}\\ s_t^i\end{bmatrix} + \begin{bmatrix}r_t^i\cos(\phi_t^i + \bar{\mu}_{t,\theta})\\ r_t^i\sin(\phi_t^i + \bar{\mu}_{t,\theta})\\ 0\end{bmatrix} \tag{3.28}$$

Steps 11 to 13 have the purpose of computing some geometrical quantities which are going to build up the expected measurement $\hat{z}_t^i$ of a landmark i.
Step 14 introduces the matrix $F_{x,j}$ as a 6 x $(3N+3)$ matrix in which the coordinates of the landmark j are set to 1, giving the matrix the shape in algorithm 3.3, according to what exposed in section 3.4.2.
Steps 15 and 16 compute the Kalman gain as function of the distance between the landmark and the rover, the matrix $F_{x,j}$, the motion-update covariance $\bar{\Sigma}_t$ and the sensor uncertainty matrix Q_t:

$$K_t^i = \bar{\Sigma}_t H_t^{iT}(H_t^i\bar{\Sigma}_t H_t^{iT} + Q_t)^{-1} \tag{3.29}$$

Steps 18,19 and then, 20,21 bring respectively to the final updates and computations of μ_t and Σ_t:

$$\begin{aligned}\mu_t &= \bar{\mu}_t = \bar{\mu}_t + K_t^i(z_t^i - \hat{z}_t^i)\\ \Sigma_t &= \bar{\Sigma}_t = (I - K_t^i H_t^i)\bar{\Sigma}_t\end{aligned} \tag{3.30}$$

Algorithm 3.3 Extended Kalman Filter SLAM

```
Require: $\mu_{t-1}$, $\Sigma_{t-1}$, $u_t$, $z_t$, $c_t$
 1: $F_x = \begin{bmatrix} 1 & 0 & 0 & 0...0 \\ 0 & 1 & 0 & 0...0 \\ 0 & 0 & 1 & 0...0 \end{bmatrix}$
 2: $\bar{\mu}_t = \mu_{t-1} + F_x^T \begin{bmatrix} g_x \\ g_y \\ g_\theta \end{bmatrix}$
 3: $G_t = I + F_x^T \begin{bmatrix} 0 & 0 & \frac{\partial g}{\partial x} \\ 0 & 0 & \frac{\partial g}{\partial y} \\ 0 & 0 & \frac{\partial g}{\partial \theta} \end{bmatrix} F_x$
 4: $\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + F_x^T R_t F_x$
 5: $Q_t = \begin{bmatrix} \sigma_r^2 & 0 & 0 \\ 0 & \sigma_\phi^2 & 0 \\ 0 & 0 & \sigma_s^2 \end{bmatrix}$
 6: for all observed features $z_t^i = (r_t^i, \phi_t^i, s_t^i)^T$ do
 7:     $j = c_t^i$
 8:     if landmark $j$ never seen before then
 9:         $\begin{bmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \\ \bar{\mu}_{j,s} \end{bmatrix} = \begin{bmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{t,y} \\ s_t^i \end{bmatrix} + \begin{bmatrix} r_t^i \cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i \sin(\phi_t^i + \bar{\mu}_{t,\theta}) \\ 0 \end{bmatrix}$
10:     end if
11:     $\delta = \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \begin{bmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{bmatrix}$
12:     $q = \delta^T \delta$
13:     $\hat{z}_t^i = \begin{bmatrix} \sqrt{q} \\ \arctan(\delta_x, \delta_y) - \bar{\mu}_{t,\theta} \\ \bar{\mu}_{j,s} \end{bmatrix}$
14:     $F_{x,j} = \begin{bmatrix} 1 & 0 & 0 & 0...0 & 0 & 0 & 0 & 0...0 \\ 0 & 1 & 0 & 0...0 & 0 & 0 & 0 & 0...0 \\ 0 & 0 & 1 & 0...0 & 0 & 0 & 0 & 0...0 \\ 0 & 0 & 0 & 0...0 & 1 & 0 & 0 & 0...0 \\ 0 & 0 & 0 & 0...0 & 0 & 1 & 0 & 0...0 \\ 0 & 0 & 0 & 0...0 & 0 & 0 & 1 & 0...0 \end{bmatrix}$
15:     $H_t^i = \frac{1}{q} \begin{bmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x & 0 \\ 0 & 0 & 0 & 0 & 0 & q \end{bmatrix} F_{x,j}$
16:     $K_t^i = \bar{\Sigma}_t H_t^{iT} (H_t^i \bar{\Sigma}_t H_t^{iT} + Q_t)^{-1}$
17:     $\bar{\mu}_t = \bar{\mu}_t + K_t^i (z_t^i - \hat{z}_t^i)$
18:     $\bar{\Sigma}_t = (I - K_t^i H_t^i) \bar{\Sigma}_t$
19: end for
20: $\mu_t = \bar{\mu}_t$
21: $\Sigma_t = \bar{\Sigma}_t$
22: return $\mu_t$, $\Sigma_t$
```

3.4.5. Unknown Correspondences

Algorithm 3.3 illustrates the Extended Kalman filter SLAM in the case of known correspondences, which means that at every time and in every observation the identity of a landmark in known and so whether if it has been seen already or not. This translates in assuming a perfect data-association.
In the real case this is almost impossible, so the data association problem must be taken in account and the algorithm has to be expanded with a dedicated section, as it is going to be done in the next chapter.

3.5. Extended Kalman Filter SLAM properties

The Extended Kalman filter formulation of the SLAM problem brings some important properties on the surface.

3.5.1. Fully populated Kalman gain

The first interesting property relates to the Kalman Gain K_t: the fact that the Kalman Gain is fully populated for all state variables, and not just the robot pose and the observed landmark, is very important. In SLAM, observing a landmark does not just improve the position estimate of this very landmark, but that of other landmarks as well. This effect is mediated by the robot pose: observing a landmark improves the robot pose estimate, and as a result it eliminates some of the uncertainty of landmarks previously seen by the same robot. The sensational effect here is that we do not have to model past poses explicitly, which would move the algorithm in the Full SLAM area of interest. Instead, this dependence is captured in the Gaussian posterior, specifically in the off-diagonal covariance elements of matrix Σ_t.
Figure 3.5 shows how the covariance matrix Σ_t evolves with time during the SLAM: the left column of images represents the evolution of the robot movement and observation, while the right column shows how the covariance matrix behaves. In the first top image the initialization of the SLAM is shown, with the robot indicated as a red dot and a big uncertainty ellipse over the first landmark observed; meanwhile the covariance matrix is initialized with only diagonal values with high uncertainty (values tending to ∞).
The second raw of images shows the path of the robot after a Δt from the first image: a number of landmarks has been observed and the related uncertainty ellipses are drawn. At the same time the covariance matrix has begun to be filled and the spread of values

off the merely diagonal terms has begun. The dependence between landmarks and poses has started to be represented by these off-diagonal terms.
The last raw of figures shows a further evolution in which some of the uncertainty ellipses have shrunk due to the more certain position obtained by the re-observing of landmarks and the spread of this information in the covariance matrix. This phenomena is represented by the light gray elements in the matrix: the lighter is the gray of an element, the less is the uncertainty associated. It can then be seem the evolution of the uncertainty in a number of the off-diagonal terms which changed their color from top to bottom, gaining a less uncertain state. In the limit, the landmarks estimates become fully correlated, as shown in figure 3.6.

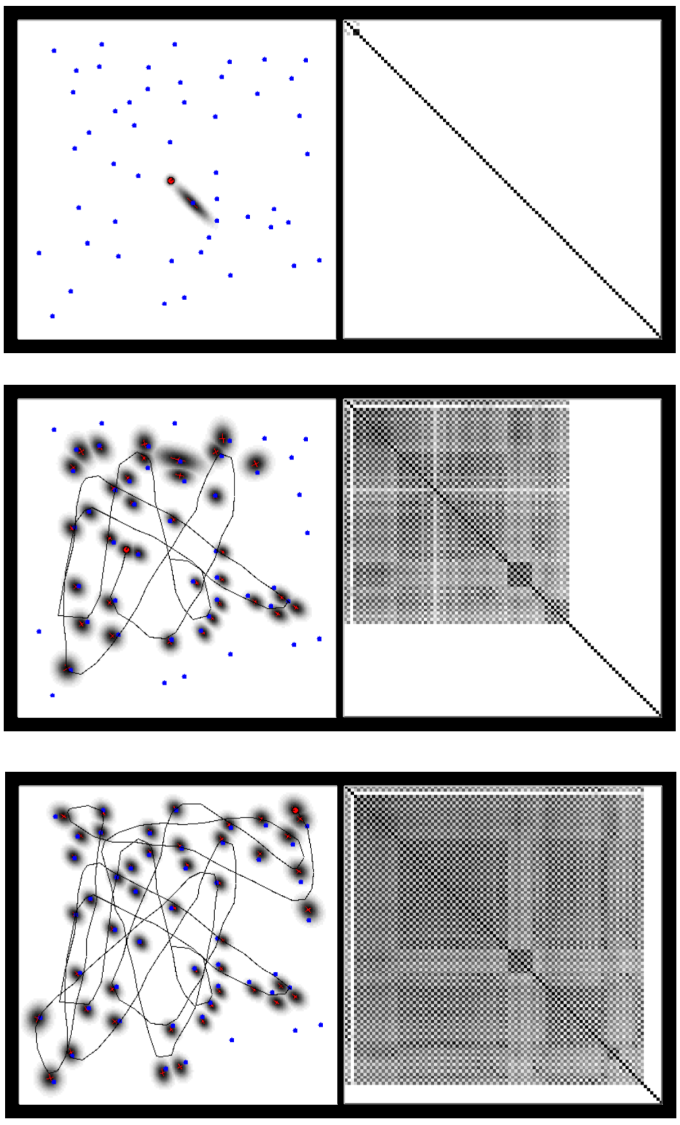


Figure 3.5: Covariance filling evolution

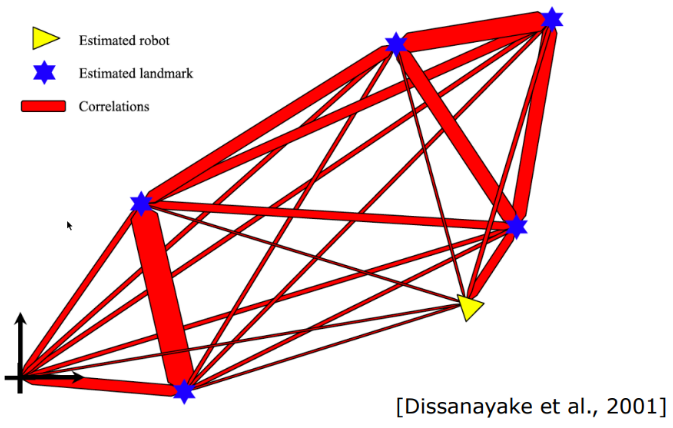


Figure 3.6: Landmarks correlation

The red lines in figure 3.6 can be seen as elastic bands of which the elasticity represents the less or more uncertain is the correlation between the landmarks linked by them: wider red lines stand for more rigid links, meaning a less uncertain correlation between the two linked landmarks estimates. Viceversa, thinner lines stand for more elastic link and a less certain correlation.

3.5.2. Uncertainties evolution

As observed in section 3.5.1, the uncertainty over the landmarks pose evolves with the robot movement and the observations history.
Figure 3.7 shows the uncertainty ellipses evolution of both the robot pose and the landmarks estimates during the EKF SLAM functioning. In the figure the robot path is a dashed line, while the uncertainties over its pose are represented at each step by the gray ellipses. The small dots are known distinguishable landmarks of unknown location and their location estimates are represented by white ellipses.
In the first three pictures the robot pose uncertainty in increasing and so the uncertainty about the landmarks it encounters (due to the above mentioned correlation between the pose and the landmarks estimates).
In the bottom right picture the robot encounters again the first landmark, making the uncertainty over all landmarks decrease as the uncertainty over its current pose. This is represented by the decreased size of the ellipses.
The underlined phenomena is due to the fact that reducing the uncertainty over the first landmark affects the whole covariance matrix. The uncertainty reduction for the first landmark estimate comes from the fact that the more measurements of a landmark are done, the more certain its estimate tends to be.

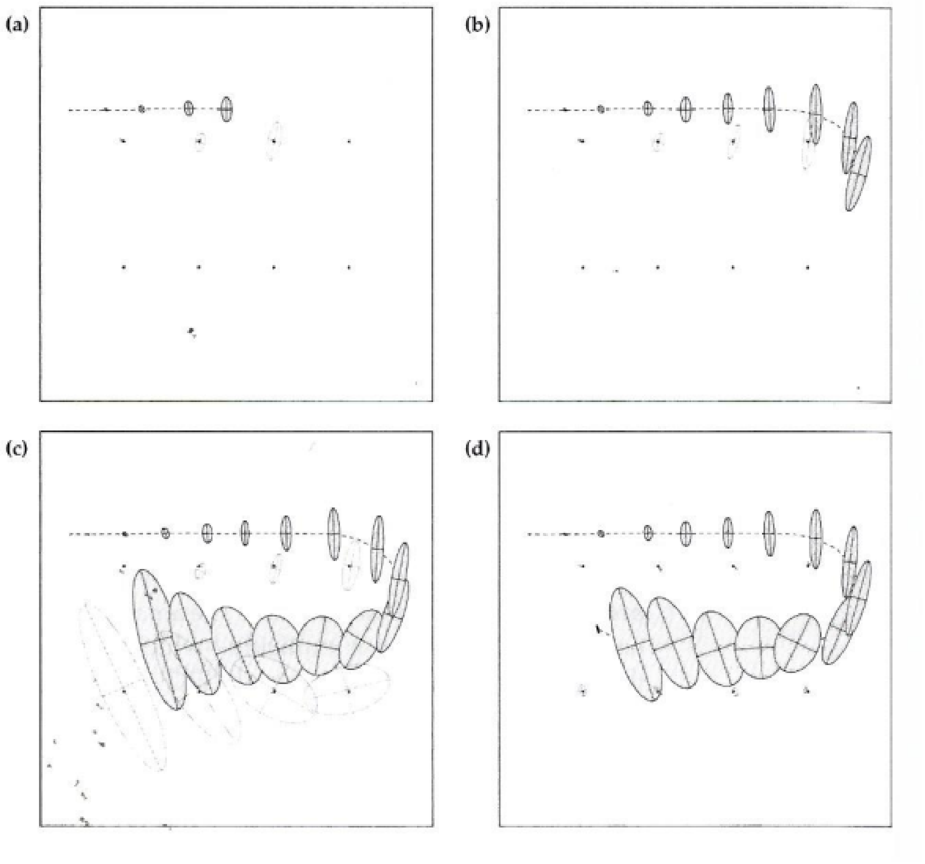


Figure 3.7: Uncertainties evolution in SLAM

It can be demonstrated that, in the limit, the covariance associated with any single landmark location estimate is determined only by the initial covariance in the vehicle location estimate, as shown here:

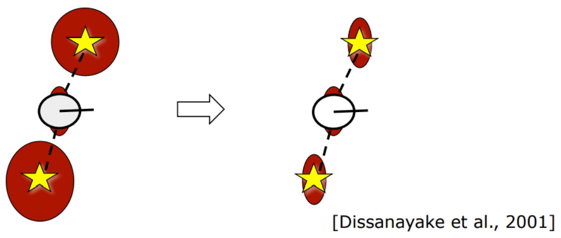
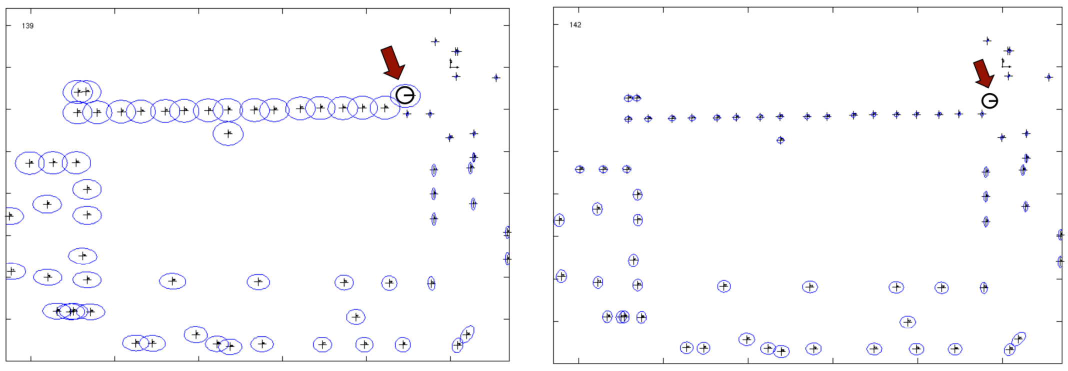


Figure 3.8: Landmark covariance, limit case

3.5.3. Loop Closure

Recognizing a previously mapped area is called *Loop Closing* and it is a sort of Data Association problem characterized by an high level of ambiguity and possible environmental symmetries. The Loop Closure makes the uncertainties collapse after its application (both in the case of a correct loop closure or in the case of a wrong one). Recalling figure 3.6: when a Loop Closure is executed, it freezes the red lines in their current form.



Courtesy of K. Arras

Courtesy of K. Arras

Figure 3.9: Loop Closure in SLAM

As said, it reduces the uncertainty over in robot and landmarks estimates, as shown in figure 3.9, in which on the left is represented the problem before the loop closure and on the right after the loop closure. However, wrong loop closures leads to filter divergences and irretrievable errors.
More about the Loop Closure problem can be found in [7], [6] and [10].

Usually, loop closures are exploited for better map building tasks. In this thesis the Loop Closure problem is not going to be explored, with consequential results.

4 | EKF with unknown Data Association

4.1. Unknown Data Association

In chapter 3 the Extended Kalman Filter algorithm has been shown with the fundamental assumption of known Data Association. This assumption guarantees that for each landmark observed, the algorithm knows exactly which landmark it is and if it has been already observed before or not. In order to reproduce this assumption it is necessary to insert a sort of a label over each observed feature, represented in chapter 3 by the variable c_t^i. However, when dealing with real sensors and environments, the landmarks are not observed with a specific label unequivocally marking them, leading to possible wrong associations. So, relaxing the assumption of perfectly known data association, the algorithm opens up to the *Unknown Data Association* problem.
The Data Association problem is mainly related to Computer Vision and Optimization techniques and it is one of the most challenging problem in autonomous systems SLAM. Several approaches have been explored during the past, exploiting sensors capabilities, computational cost analysis and SLAM algorithmic properties.

In this thesis a primitive approach to the Data Association problem is explored in order to test the algorithm and its outcomes. The approach used is based on the *Iterative Closest Point* algorithm for point clouds matching, in particular on its implementation proposed by Huang and Arun [1] and their *Least Squares* approach for optimization.
In the following section this approach to solve the Data Association problem is illustrated in detail. It must be pointed out for the reader that this approach is a very simple one, since the Data Association is not the main focus of this thesis, and big improvements on this area of the algorithm can be done. A great starting point for improvements in the Data Association part of the algorithm could be the work in [4].

The data association in this thesis is articulated over few steps, starting from the clustering of the point clouds coming from the sensors into useful landmarks, until the correct designation of the seen features. These steps are summarized as:

Algorithm 4.1 Data Association procedure

```
Require: point cloud at time $t$, point cloud at time $t-1$ (both with landmarks defined
    by algorithm 2.4)
1: Filter out the rover parts from the point clouds
2: Run Huang-Arun algorithm to find the movement between the point clouds
3: Match the features in the point clouds to select the observed landmarks
4: Associate to each observed landmark a range and a bearing measurement
5: Select only the best associations
6: return Point cloud at time $t$ with only its best associated landmark
```

As shown in algorithm 2.4, there is a part of the simulation procedure in which the point clouds from the sensors are scanned for landmarks. Landmarks are selected as clusters of point with an Euclidean distance between each other under a certain threshold and with a minimum angular distance. Then for each cluster its geometric centre coordinates in the robot local frame are computed. This is the structure of point cloud data entering algorithm 4.1.

4.1.1. Rover filtering

Step 1 of algorithm 4.1 aims at filtering out from the point cloud of the measurement under study all the spurious components of the rover that may have entered the sensor field during the acquisition. These parts can affect the SLAM algorithm introducing wrong associations and dangerous errors. For example, during the first simulations, a wheel of the rover was entering the acquired point clouds, affecting the whole result with unacceptable errors and making the SLAM strongly diverging.
In order to filter out the rover, step 1 operates by removing from the point cloud all the clusters inside a cuboid threshold centered on the rover. In this way it is possible to remove the wheels of the rover, its chassis or its robotic arm in the case it entered the sensor field of view during the acquisition.
After few iterations of the algorithm it was clear that also landmarks far from the rover created some wrong correspondences. It has so been decided to apply a similar method to filter out also the landmarks over a cuboid threshold centered over the rover. As a result, after step 1, only the landmarks inside a "squared donut" centered on the rover are kept for the Data Association further procedures.

4.1.2. Huang-Arun algorithm

Step 2 of algorithm 4.1 is fundamental for the subsequent development of the Data Association part. The aim of this step is finding the translation vector $\vec{t}$ and the rotation matrix R linking the point cloud at the current time t to the point cloud at the previous time $t-1$. In order to obtain these results, the algorithm proposed by Huang and Arun in [1] has been adopted. The algorithm is shown here below:

Algorithm 4.2 Least-Squares Fitting of Two 3-D Point Sets

```
Require: Two 3D point sets $p_i$ and $p_i^{'}$
 1: $p = \frac{1}{N}\sum_{i=1}^{N} p_i$
 2: $p^{'} = \frac{1}{N}\sum_{i=1}^{N} p_i^{'}$
 3: $\{q_i\} = p_i - p$
 4: $\{q_i^{'}\} = p_i^{'} - p^{'}$
 5: Calculate the 3x3 matrix: $H = \sum_{i=1}^{N} q_i q_i^{'T}$
 6: Singular Value Decomposition of H: $H = U\Delta V$
 7: Calculate: $X = VU^T$
 8: Calculate: $\det(X)$
 9: if $\det(X) = +1$ then
10:     $R = X$
11: else if $\det(X) = -1$ then
12:     The algorithm fails
13: end if
14: $\vec{t} = p^{'} - Rp$
15: return Rotation matrix $R$, translation vector $\vec{t}$
```

Algorithm 4.2 shows the approach to the least square fitting proposed by Huang-Arun and adopted in this thesis. It operates by finding the rotation matrix R and the translation vector $\vec{t}$ capable to rotate and translate the second point cloud over the first one.
Particular attention must be put on the IF cycle from step 9 to step 13: the algorithm gives a solution only if the determinant of matrix X, previously computed, is equal to $+1$, while the possible results for the determinant of matrix X are $+1$ and -1. However, the negative result stands for a reflection, while the positive stands for a rotation that is what is seek by the algorithm. The full proof can be checked in [1].

After the application of this step, the matrix R and the translation vector $\vec{t}$ are available and the two point clouds can be superposed in order to scan them and look for similar features to be associated.

4.1.3. Best features matching

With the two point clouds superposed it is possible to start scan both of them in order to evaluate possible matchings. The scan is done by selecting one cluster, or landmark, and comparing it to all the landmarks in the other point cloud. During this process the euclidean distances between each point are evaluated and a score is assigned to each of the associations.
The comparing between the cluster is done by means of a *Fast Approximate Nearest Neighbors* algorithm [9], or *Fast Global Registration* algorithm [13], thanks to the function:

```
[indexPairs, scores] = pcmatchfeatures(features1,features2);
```

Listing 4.1: Matching features function

At this stage, the variables *indexPairs* and *scores* obtained are in the form:

- *indexPairs* is a m x 2 matrix in which elements in the first column indicates the indexes of points in the current point cloud matching the indexes in the second column of points in the second point cloud;
- *scores* is a column vector containing the Euclidean distance between each of the couples of points contained in *indexPairs*.

Once a cluster has been compared to all the clusters in the other point cloud, the best score is selected as the association for the landmark under study. By doing so each of the landmarks is associated with one of the landmarks in the previous point cloud.
However, it is possible that wrong associations or low score associations entered the list of landmarks labels, so other criteria are applied to shrink the number of associations and select only the best ones and the more trustworthy:

- A first criteria is to select only the associations in which a number of points over a certain threshold has been associated. If the number of associated points is less than the threshold, the association is discarded;
- In the association remained, only the best scores are kept.

After running several time this algorithm for the Data Association, it was noted that the algorithm still kept some wrong data associations. The solution was to modify the last step in:

- In the associations remained, select only the best score.

By doing so only one best data association for each step is kept and fed to the correction step. This helped to filter out errors coming fro wrong associations and helped the algorithm to reach better convergence results.

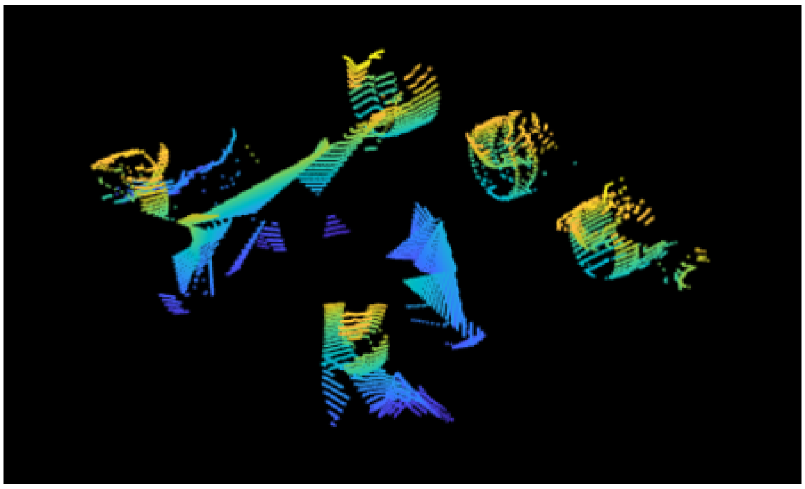


Figure 4.1: Examples of associated landmarks

Figure 4.1 shows an example of associated landmarks (without the filtering out of rover components): it can be clearly noted how the blue landmarks of the current point cloud superpose over the greenish landmarks of the previous acquisition, except for some non-associated clusters.

In appendix A this choices are explained in more detail together with the implemented solution.

5 | Results

This chapter is going to illustrate the results of this thesis work. It is divided in three different section, each of them showing the results in the three different parts of the simulation explained in chapter 2.

The first section illustrates the result in the sensors simulation area. The output shown are mainly of a qualitative kind, showing off the view from the StereoCam recreated or the raw point clouds visualization.
The second part shows the results in looking for the landmarks inside the raw data, again mainly in a qualitative way.
The core of the chapter is the third section in which the SLAM results are shown. Firstly, a graphical comparison between the nominal trajectory, the noisy one and the EKF SLAM trajectory is made, followed by a comparison between a SLAM with only the prediction step and a complete SLAM on their effects over the trajectory. The behaviour of the error from the nominal trajectory during the SLAM is then analyzed, followed by an analysis on the behaviour of the mean vector μ and the covariance matrix Σ.
Afterwards the results of the Extended Kalman filter SLAM are observed together with the variation of some of its parameters: the influence of the magnitude of the values of the sensor uncertainty matrix Q, the variation over the number of landmarks associated at each step and others more.
The final outputs analyzed are the results in case of bad or wrong Data Association and the possible effects of the insertion of a Loop Closure inside the algorithm developed.

When exposing the result some of the assumptions introduced during the thesis are revised and listed in order to recall how the path to the results has been followed. However, the reader is invited to confront the algorithms and functions written in the previous chapters to better understand the exposure of the results.

5.1. Simulation outputs

This section has the purpose of showing the outcomes from the sensor acquisitions inside the simulation environment. The results are going to be qualitative, showing the view from the stereo camera or the Lidar acquired point cloud.
As illustrated in chapter 2, both the sensors are acquiring their data inside the Mars environment simulated with the help of Blender and Unreal Engine. Once the sensor setup is completed and the simulation is run, the results are collected inside the Matlab workspace. From there it is possible to visualize all the following results.

5.1.1. Lidar acquisition

The Lidar sensor acquisition is set inside the Simulink environment directly in the form of a four dimensions variable. At each instant t of the simulation, the Lidar sensor generates a 3D unordered point cloud of the environment, which is saved in a dedicate variable. At each time t the acquisition has to be converted in the form of an ordered point cloud. Then it is possible to operate on it and to visualize it.

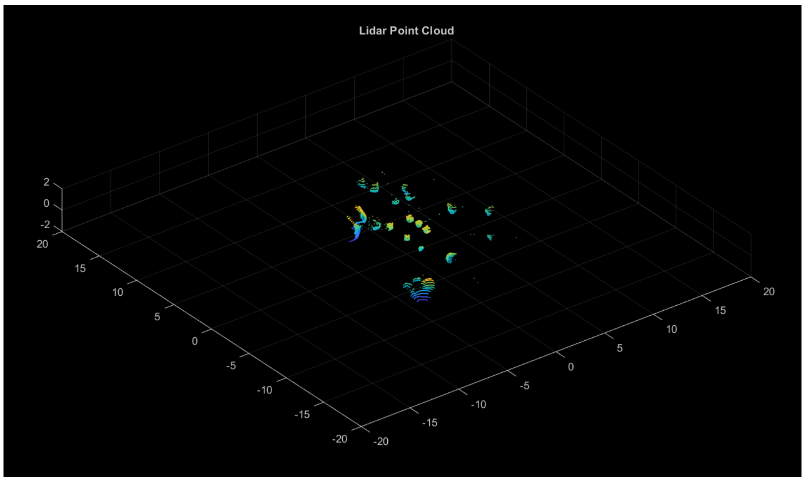


Figure 5.1: Lidar point cloud visualization

By looking at figure 5.1 it is possible to note that the ground is not revealed by the Lidar sensor: this happens because the ground has been set as "transparent" to the sensor inside Unreal Engine in order to avoid the insertion of a ground filtering inside the algorithm. It real cases, however, a filter capable of removing the spurious ground acquisition must be inserted since the transparent floor assumption would not be valid anymore.
Figure 5.2 shows a zoomed image of the Lidar point cloud acquisition. It can be noted

that a sort of clustering is already active: it is clearly possible to distinguish parts of the rover, such as the wheels, and different rocks. However, in this way it is not possible to have control over the clustering procedure, so the clustering algorithm has still to be used. The clustering results are going to be shown in section 5.2.

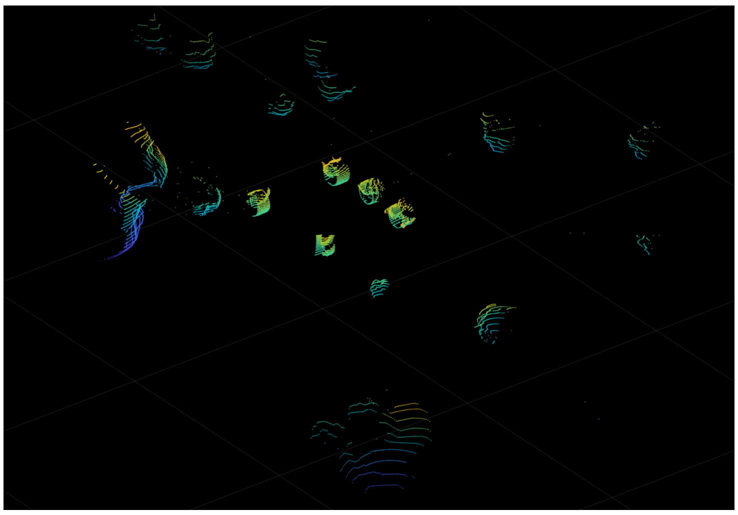


Figure 5.2: Lidar point cloud visualization zoomed

5.1.2. StereoCam acquisition

On the side of the Stereo Camera sensor, the procedure to follow in order to reach a correct from and visualization of the data is longer than in the previous case. In section 2.2.4 the setup of the StereoCam in the simulation environment has already been discussed. As already mentioned, in order to obtain a stereo image two cameras are needed, from now on called *left camera* and *right camera*. The view from the two cameras is shown in figure 5.3.

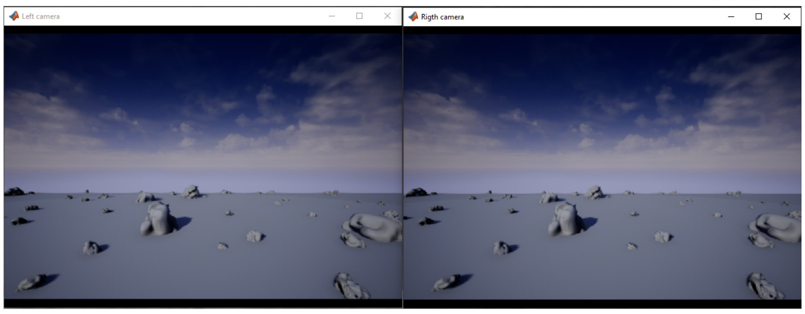


Figure 5.3: Stereo image from the left and right camera

The two images in figure 5.3 can seem almost similar: this is due to the fact that the stereo baseline has been set to 42 cm, so the prospective from the two cameras is quite close from one to the other. However, figure 5.4 shows the two images one on top of the other, highlighting the differences:

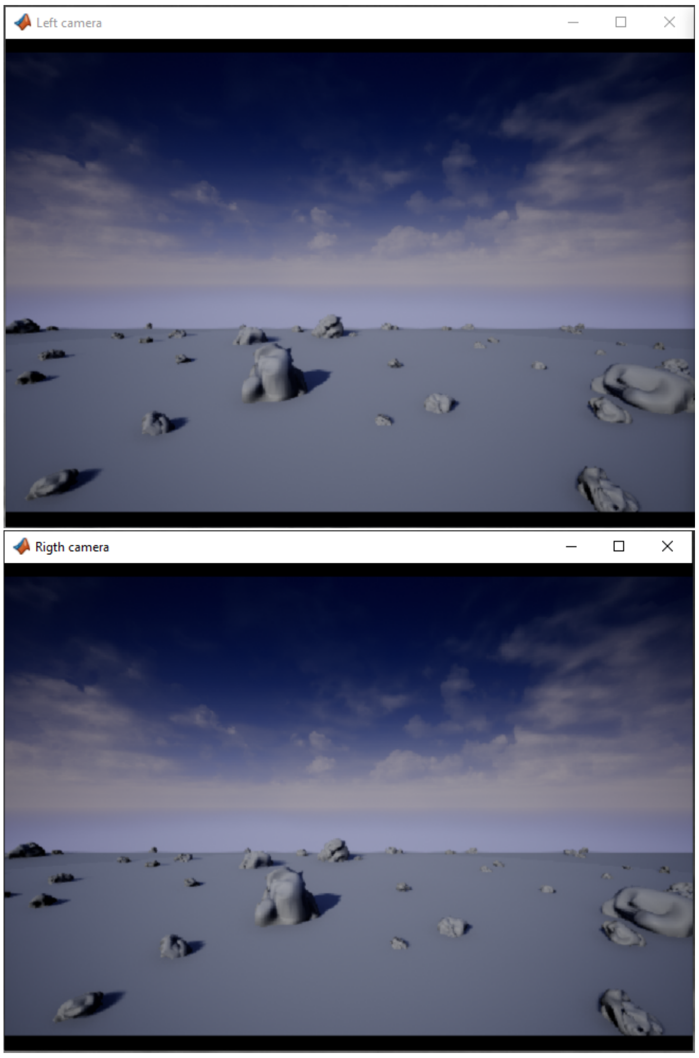


Figure 5.4: StereoCam left and right image comparison

From figure 5.4 the difference in the positions of the rock in the two images can be easily noted. This difference is the pivotal point on which the StereoCam works being so capable of generating suitable point clouds.
It must pointed out that the environment here is in gray-scale: this has been set to avoid enormous computational time for the simulation, due to the large weight of the red Mars surface textures. The textures can be easily recovered and inserted in a simulation computed with a more powerful hardware.

At each time t of the simulation, both the images are saved and stored inside apposite variables. This variables are fundamental for the next steps.

From the images of the two cameras, a first idea of the 3D reconstruction can be visualized by means of the Stereo Anaglyph Graph, shown here in figure 5.5.

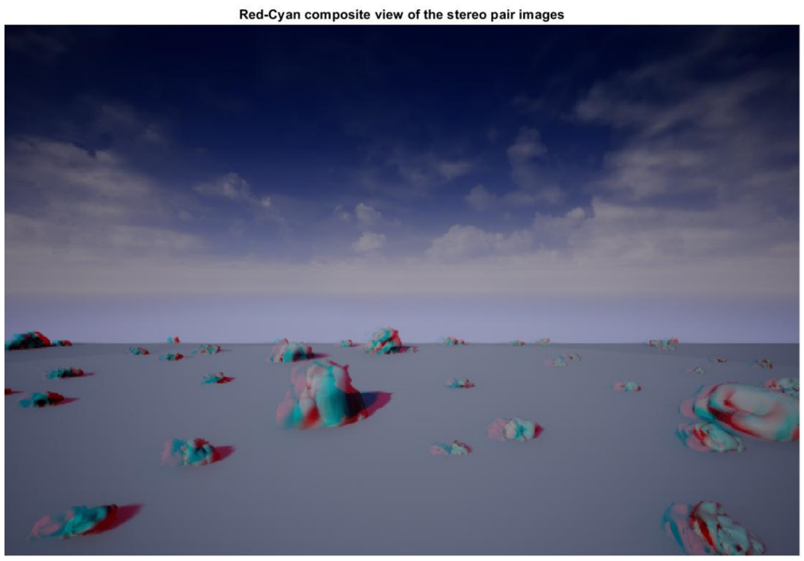


Figure 5.5: Stereo Anaglyph Graph

Figure 5.5 is *stereoscopic representation* of what the rover is seeing: the red color, associated with the left image, and the cyan color, associated with the right one, create a tridimensional illusion when observed under properly filtered lenses. The observable presence of the red and cyan shadows inside the image is a first proof that the StereoCam is effectively working.

The most important step for transforming the stereo acquisition in a useful 3D point cloud is the computation of the *Disparity Map*.

The *Disparity Map* is a 2D map reduced from a 3D space, in which the brightness of the gray scaled pixels indicates the distance of the related points from the sensor. The description of how a disparity map is computed from a couple of stereo images is beyond the purpose of this thesis. However, it must be pointed out that the disparity map computation is done by means of the *Global Block Matching algorithm*, in which two main parameters dictate the final result. The parameters under study are the *Range* and the *Uniqueness Threshold.*

Here in figure 5.6 some examples of how the disparity map of the simulation under study changes with the parameters listed before:

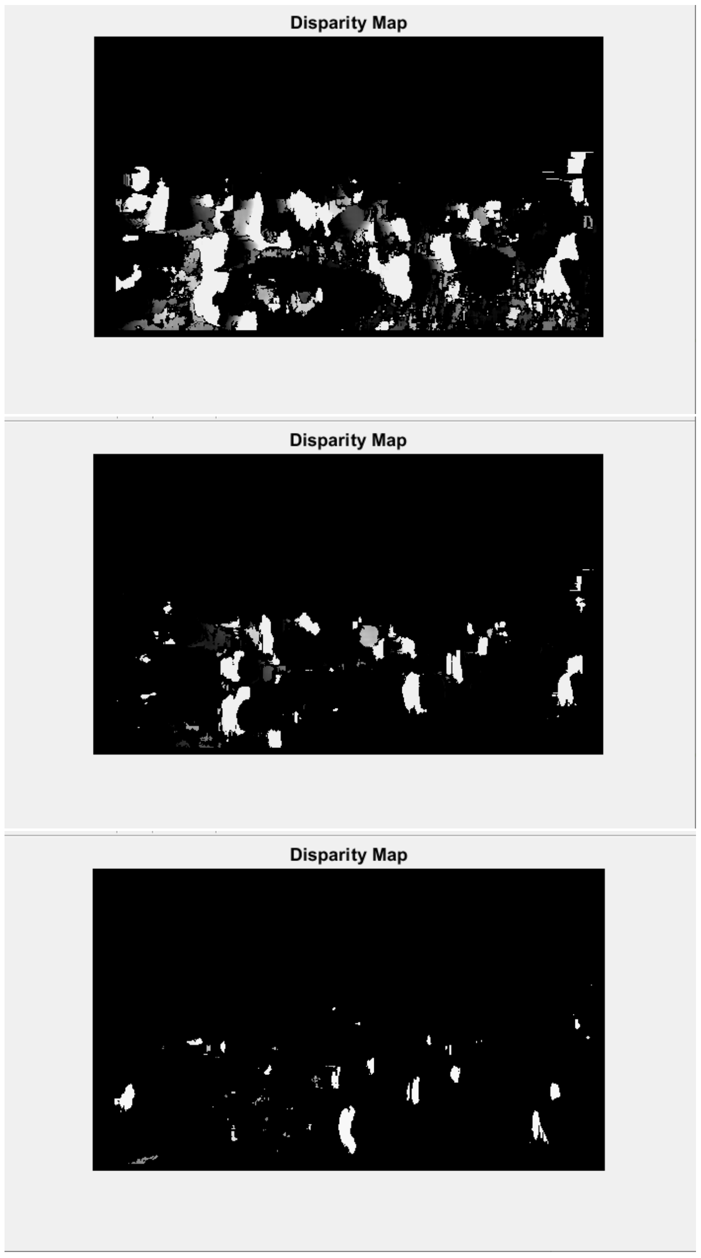


Figure 5.6: Disparity map changes with Range and Uniqueness Threshold

In the images of figure 5.6 the black color is associated with points without an assigned distance and which are not going to be inserted in the final point cloud. The gray and white colors represent respectively the further and closer point to the sensor. From the setup of the first two images, the shape of the rocks can be observed, while in the third can be imagined. However, the third image is affected by less noise.

The Disparity Map is the base on which the point cloud reconstruction from the Stereo-Cam is going to be composed.
By means of the disparity map and a matrix called Reprojection matrix, which maps a 2D point in a disparity map to a 3D point in the rectified camera coordinate system of the first camera, it is possible to reconstruct the 3D point cloud [12].
Here below are shown the results, both in OpenCV and Matlab environment:

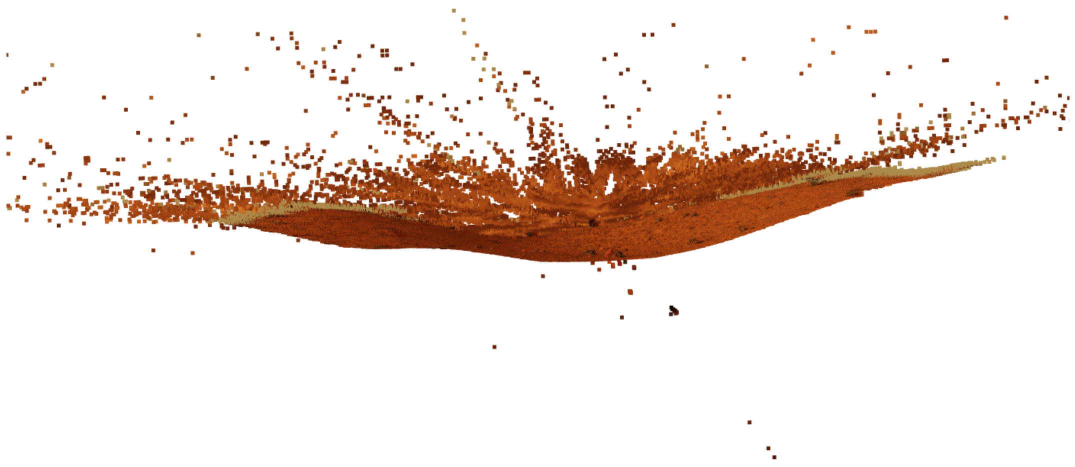


Figure 5.7: StereoCam point cloud result in OpenCV

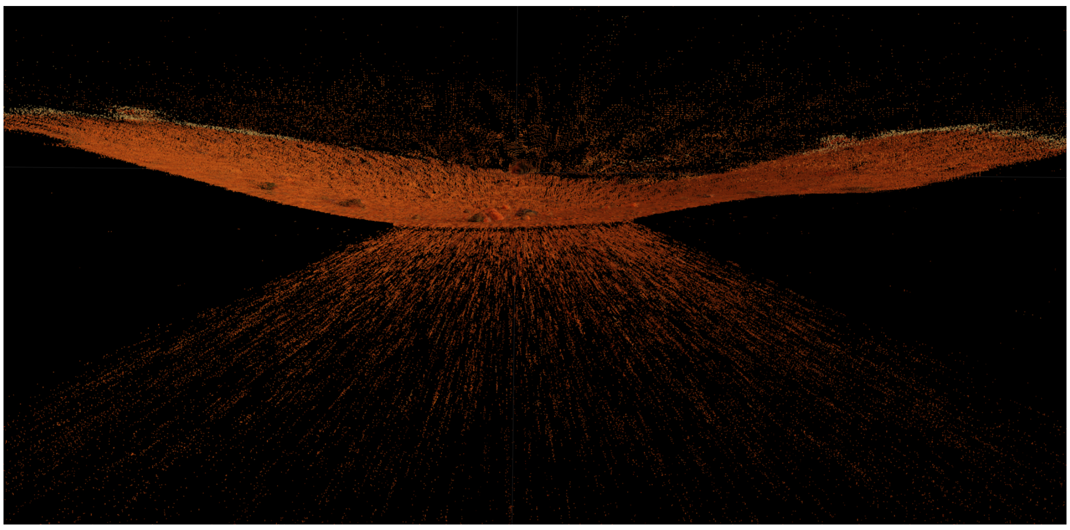


Figure 5.8: StereoCam point cloud result in Matlab

5.2. Landmarks identification outputs

Once the data coming from the sensors is organized in the form of point clouds, it is possible to proceed by applying algorithm 2.4 and extract landmarks form the point clouds, assigning a geometric centre to each of them. By doing so, it is then possible to assign also a range and bearing measurement to each of them and proceed with the Extended Kalman filter SLAM.
In this section the results from the landmarks search algorithm are shown when applied to the Lidar data, while keeping in mind that at this point also the StereoCam data are in the form of a point cloud and the same procedure and results apply to them.

As stated in chapter 4, the landmarks are isolated as clusters of points inside each point cloud. These clusters have to fulfill certain conditions on the angular distance between them and on the euclidean distance between the points composing each of them. As for the parameters of the disparity map, also in this case the values for the euclidean distance and the angular separation determine the number and shape of the clusters identified as landmarks. Figure 5.9 different shows possible results:

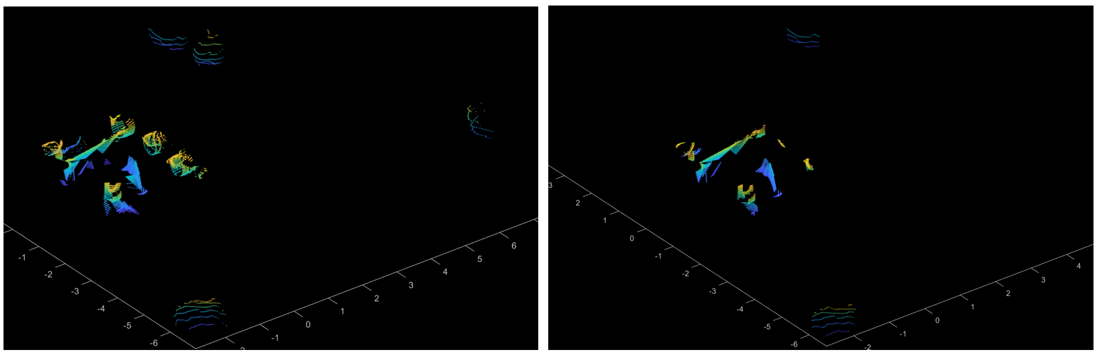


Figure 5.9: Cluster changes with the parameters setup

As can be noted by observing figure 5.9, the general shapes perceived and registered do not change much, except for a small amount of features. What really changes in the different setup for the parameters in the number of landmarks and their segmentation: choosing more strict values for the parameters translates in smaller landmarks. This means that the shape of one single landmark inside the more relaxed point cloud can correspond to a sum of different landmarks inside the more strict setup point cloud.
An example is shown in figure 5.10 where on the left is possible to observe the first landmark for the relaxed setup and on the right is possible to see the first landmark for the more strict one.

As can be easily understood, the first landmark inside the second point cloud in only a part of the first landmark in the relaxed one:

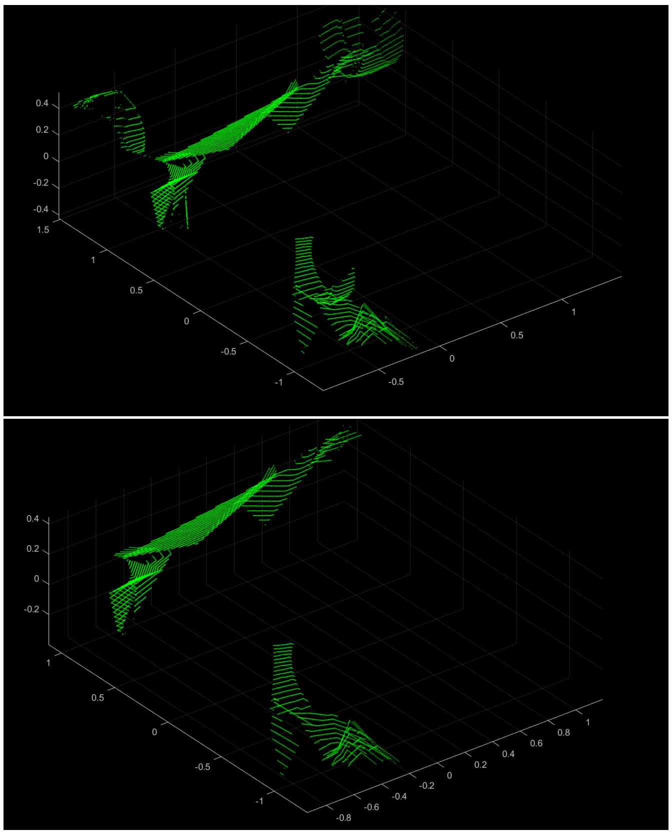


Figure 5.10: First cluster comparison

It must be kept in mind that the results shown in figures 5.9 and 5.10 show the clusters not only of the rocks selected as landmarks, but also the cluster corresponding to parts of the rover. These parts were kept in the figures mentioned because of their clear shape and so because they are more easy to confront inside the different images in order to understand the results. Instead, as mentioned in chapter 4, inside the algorithm these parts are filtered out.

After the landmarks have been identified inside the point clouds, their geometric centre is computed. Figure 5.11 shows a set of landmarks with a sphere identifying the location of the geometric centre of each of them.The spheres are set with a unitary radius and, again, the parts of the rover are kept in order to make the visual representation more clear.

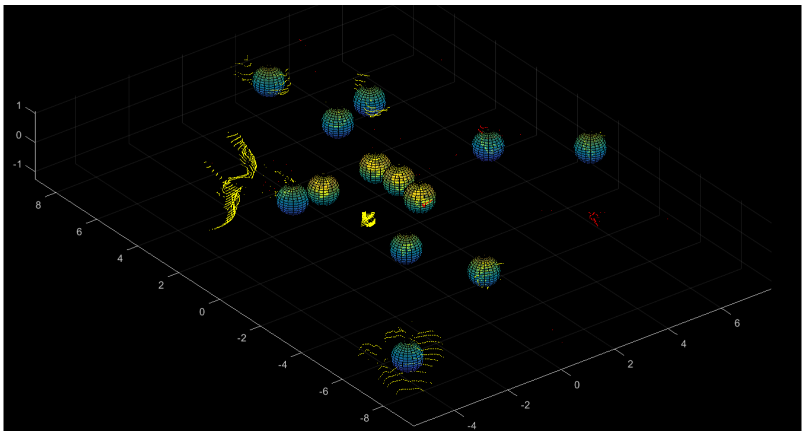


Figure 5.11: Landmarks geometric centre visualization

Figures 5.12 and 5.13 represent the resulting dependency of the number of landmarks on the different setup parameters. In particular, figure 5.12 shows the inverse dependency of the number of landmarks over the Euclidean distance threshold. Intuitively, the less is the value set for the threshold over the euclidean distance, the more the points of the cluster have to be close to each other. When the distance threshold becomes bigger, the larger is the number of points inside a single point cloud.

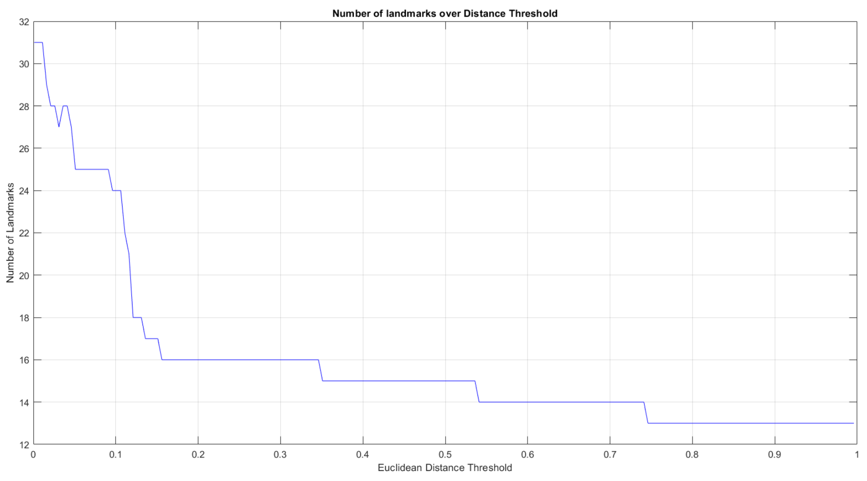


Figure 5.12: Landmarks number variation with Euclidean Distance Threshold

Figure 5.13 illustrates the dependency of the number of landmarks with respect to the angular threshold set during the clustering. The function groups adjacent points into the same cluster if the angle formed by the sensor and the points is greater than the angle threshold.

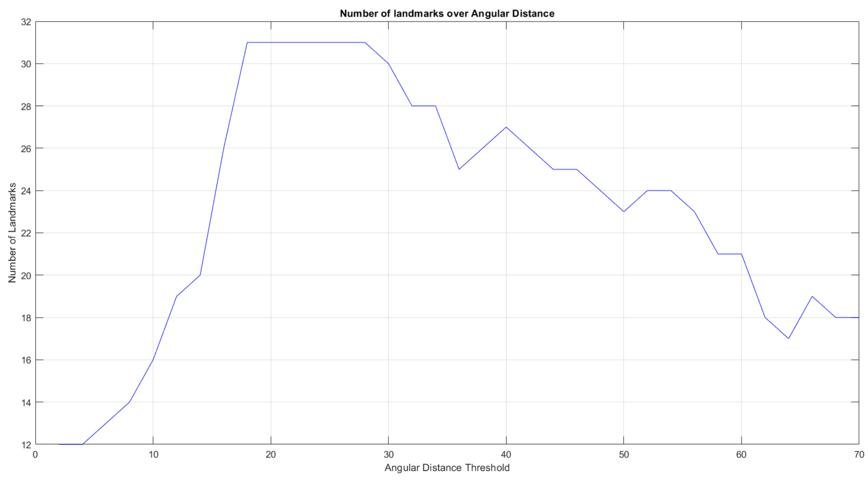


Figure 5.13: Landmarks number variation with Angular Threshold

At this point the results from the simulation environment and the landmarks clustering have been shown. These results are the base on which the Extended Kalman filter SLAM is going to operate, so the illustration of them was pivotal in order to understand the result that are going to be shown in the next section.

It must be pointed out for the reader that further deepening on the dependency between the simulation outputs and the parameters set in the various steps can be performed. For example, an analysis over the dependency between the point cloud generated from the StereoCam and the final form of the Disparity Map can be done.

5.3. EKF SLAM outputs

This section illustrates the final results of the Extended Kalman filter SLAM simulation. The results are shown firstly as outputs of a single simulation which is going to be analyzed in detail. Then an analysis on the errors behaviour is performed, followed by some case studies on the dependency of the EKF SLAM results with the changing of some of its parameters. The possible improvements are going to be then discussed in the next chapter.

5.3.1. Nominal trajectory of the rover

As illustrated in chapter 2 the desired trajectory is drawn inside the Matlab environment directly on the map of the 2D plane in which the rover is going to move. Figure **??** shows an example of the drawn trajectory, recalling what has been shown in chapter 2.

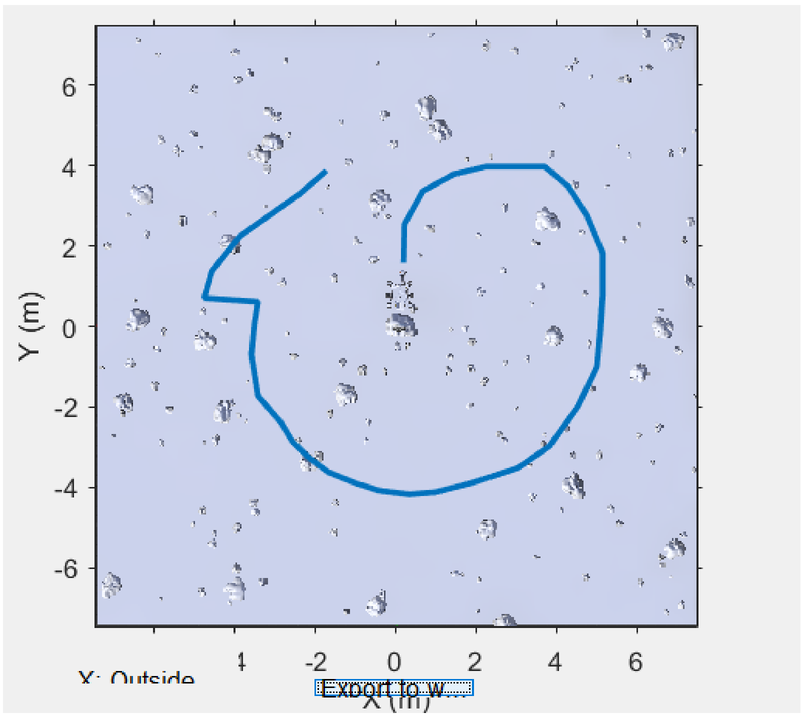


Figure 5.14: Nominal trajectory drawing

The trajectory drawn is then smoothed and the control input history to follow it is reconstructed. This drawn trajectory is going to be called *nominal trajectory* from now on and it is going to be compared with the same trajectory when affected by errors and to the trajectory after the Extended Kalman filter SLAM application.

5.3.2. Noisy trajectory

The nominal trajectory represents the path desired for the rover to follow. However, the rover movements, the controls or the wheel spinning are affected by unpredictable error when dealing with real situations.
The simulation of these errors is done by adding a random error, in the range of $[-0.1; 0.1]$ meters, to the translation control input δ_{trans}. Figure 5.15 represents the nominal trajectory in black and the one affected by errors in red, which from now on is going to be called *noisy trajectory*.

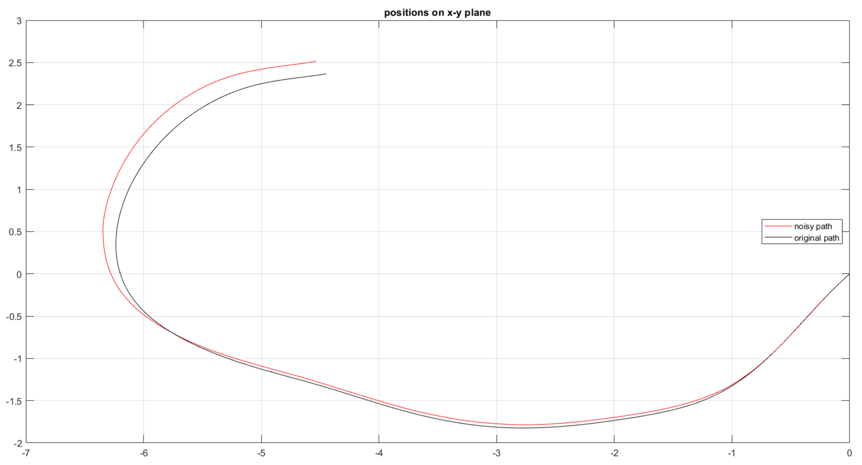


Figure 5.15: Nominal trajectory and Noisy trajectory

As can be noted, the red noisy trajectory differs from the desired one. The bigger the trajectory length is, the more the error between the two accumulates, driving to different poses from the desired one.
The behaviour of the error between the nominal trajectory and the noisy one is represented graphically in figure 5.16. By observing the figure the error respects what said about the trajectory, increasing with the distance run. It can be noted that in some places the error decreases before continuing to grow up again. These points represent the cases in which the noise on the noisy trajectory bring the trajectory itself closer to the nominal one, but without intentional movements or corrections: the random control δ_{trans} brings momentarily the noisy affected poses closer than before to the nominal trajectory, but still accumulating errors which start to increase again in the steps after.

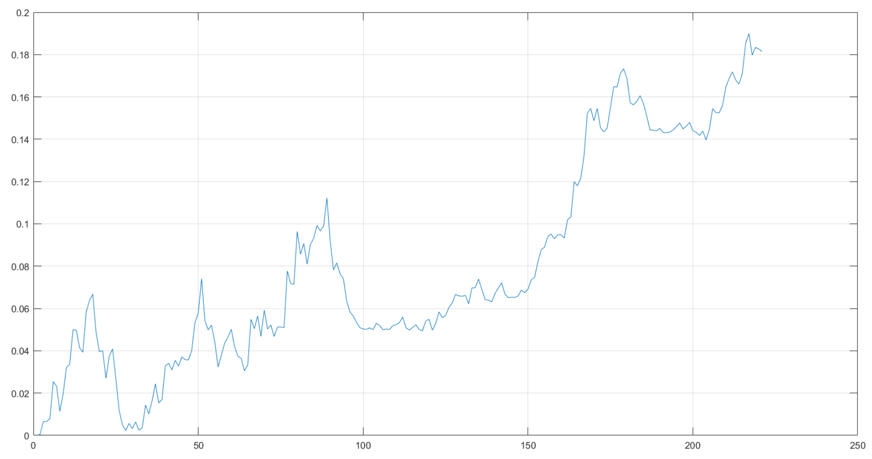


Figure 5.16: Error in time between nominal and noisy trajectory

5.3.3. Extended Kalman Filter SLAM introduction

The Extended Kalman Filter SLAM is here introduced in order to insert a correction method for the noisy trajectory, being so capable to correct itself and try to recollect the nominal pose desired.
The result of a SLAM simulation is shown here in figure 5.17:

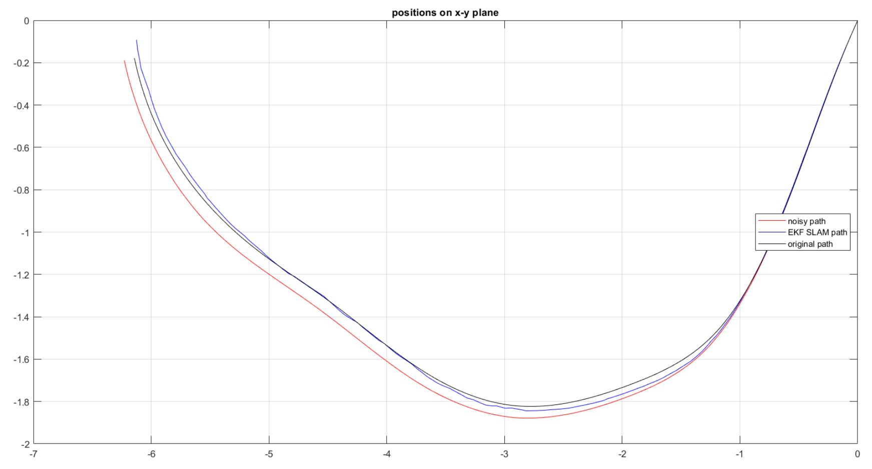


Figure 5.17: EKF SLAM trajectory

In the figure is possible to note the nominal trajectory (in black), the noisy trajectory (in red) and the Extended Kalman filter SLAM trajectory (in blue).
Figure 5.18 shows a detailed zoomed image of the trajectories represented in figure 5.17.

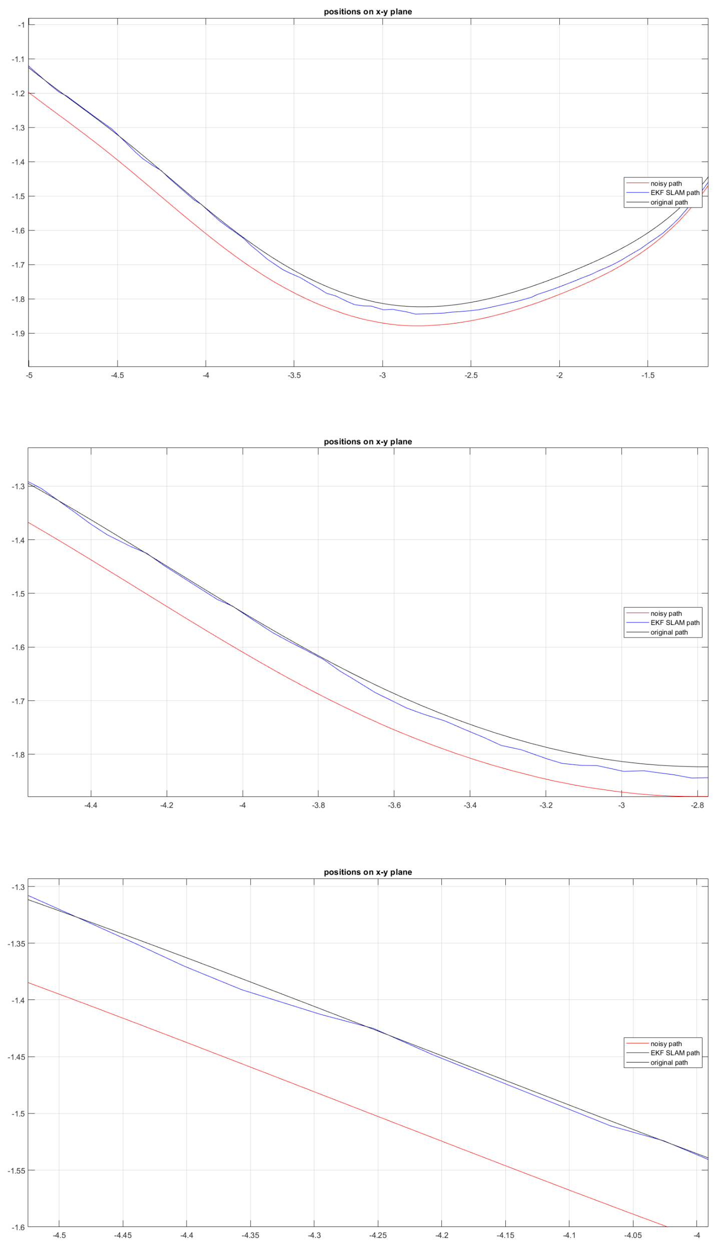


Figure 5.18: EKF SLAM trajectory in detail

As can be noted from figure 5.18, the EKF SLAM trajectory (blue) oscillates around the nominal trajectory (black): this behaviour is typical of the Extended Kalman Filter correction to the noisy trajectory (red). The difference between the behaviour of the noisy trajectory and the EKF SLAM trajectory lays in the fact the SLAM algorithm, by means of the sensor measurements inside the correction step, is capable of being aware of the the fact that its position is not the nominal one and correct it accordingly.

This happens because the sensor acquisition arrive from the rover at a certain nominal pose for each time instant t. The algorithm understands that the noisy forecasted pose in not correct and so computes a new pose based on the sensor inputs.

Figure 5.19 shows the behaviour of the error between the EKF SLAM trajectory and the nominal one during the simulation:

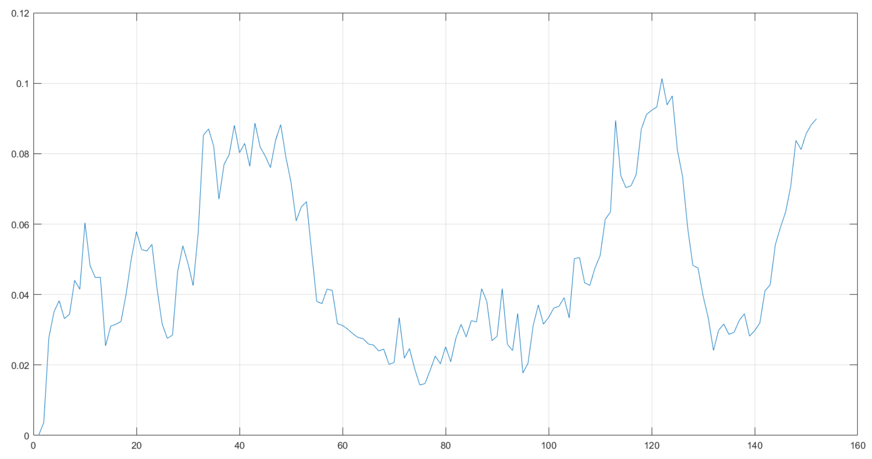


Figure 5.19: EKF SLAM error

It clearly has a behaviour similar to an harmonic one: these oscillations represents the pose going far from the nominal one and the consequent correction to a more close one. The error can be compared with the one of the noisy trajectory, as in figure 5.20. While the error in the noisy trajectory grows with a sort of linearity with distance, it can be observed that the error in the EKF SLAM increases with a slower rate and its behaviour is characterized by oscillations, as highlighted before.

It can be also noted that the magnitude of the error at the end of the trajectory is different in the two case: the EKF SLAM shows off an error which is almost half of the error in the noisy trajectory.

It must be pointed out that this relation between the error is not a certain law and when the SLAM fails or there are longer trajectories without loop closures, also the EKF SLAM tends to diverge. This happens because without the loop closure the errors, even if oscillating, sum themselves until a threshold from which the SLAM algorithm is not able anymore to correct its pose because of the data from the sensor loose useful meaning. A deeper analysis on this problem is going to be done in the next sections.

The results shown until now are related to a single Extended Kalman filter SLAM sim-

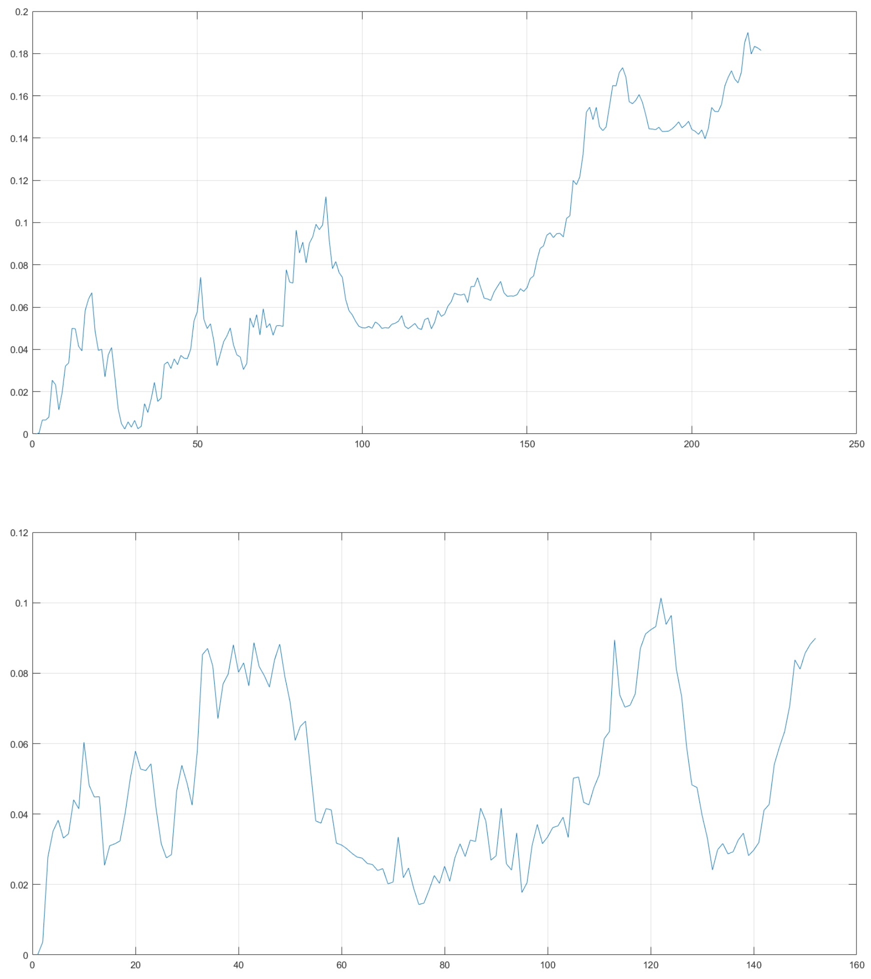


Figure 5.20: EKF SLAM error comparison

ulation, without loop closures.
In the following sections, the dependence of the results over some parameters are analyzed, together with the limitation of the EKF SLAM and its implementation.

5.3.4. EKF SLAM Sensor uncertainty matrix Q dependence

In chapter 3, particularly in algorithm 3.3, the sensor measurement uncertainty matrix Q has been introduced. As already specified, the matrix Q represents the uncertainty over the measurements, which can be due to imperfect sensors or noisy inside the measurements. Moreover, the matrix Q enters the computation of the Kalman Gain K_t, with the consequence of being a weighting factor for the correction of the rover and landmarks poses, as described in section 3.2 when the matrix Q was introduced for the first time.
As previously described, larger value of uncertainty in the sensors bring to larger values of Q and a almost negligible Kalman Gain, with no corrections to the prediction step as a result.

Here an analysis over the consequences of different values of the matrix Q on the Extended Kalman filter SLAM final result is performed.
The analysis is performed over small trajectories for sake of computational simplicity and trying different values for the measurements uncertainty. The analyzed trajectory are computed over the first hundred poses.
Figure 5.21 shows the trajectories resulting from different values of the matrix Q and the behaviour of their respective errors from the nominal trajectory.
The table below summarizes the values set for the measurement uncertainty matrix and the respective error from the final pose:

Q diagonal terms value	Final errors
10	0.35
100	0.28
300	0.17
500	0.10
1000	0.26

Table 5.1: Values of Q and respective errors

As can be observed both from the table above and figure 5.21, a fine tuning of the Q matrix brings to better results. In this work, since previous information about the sensor uncertainties were not available, the matrix Q has been tuned by a trial and error approach. The more high the values of Q are, the more smooth is the correction of the EKF SLAM from the noisy trajectory to the nominal one; however, the higher the values of Q are, the less power has the correction step in the algorithm, bringing the rover to an inability to correct itself.

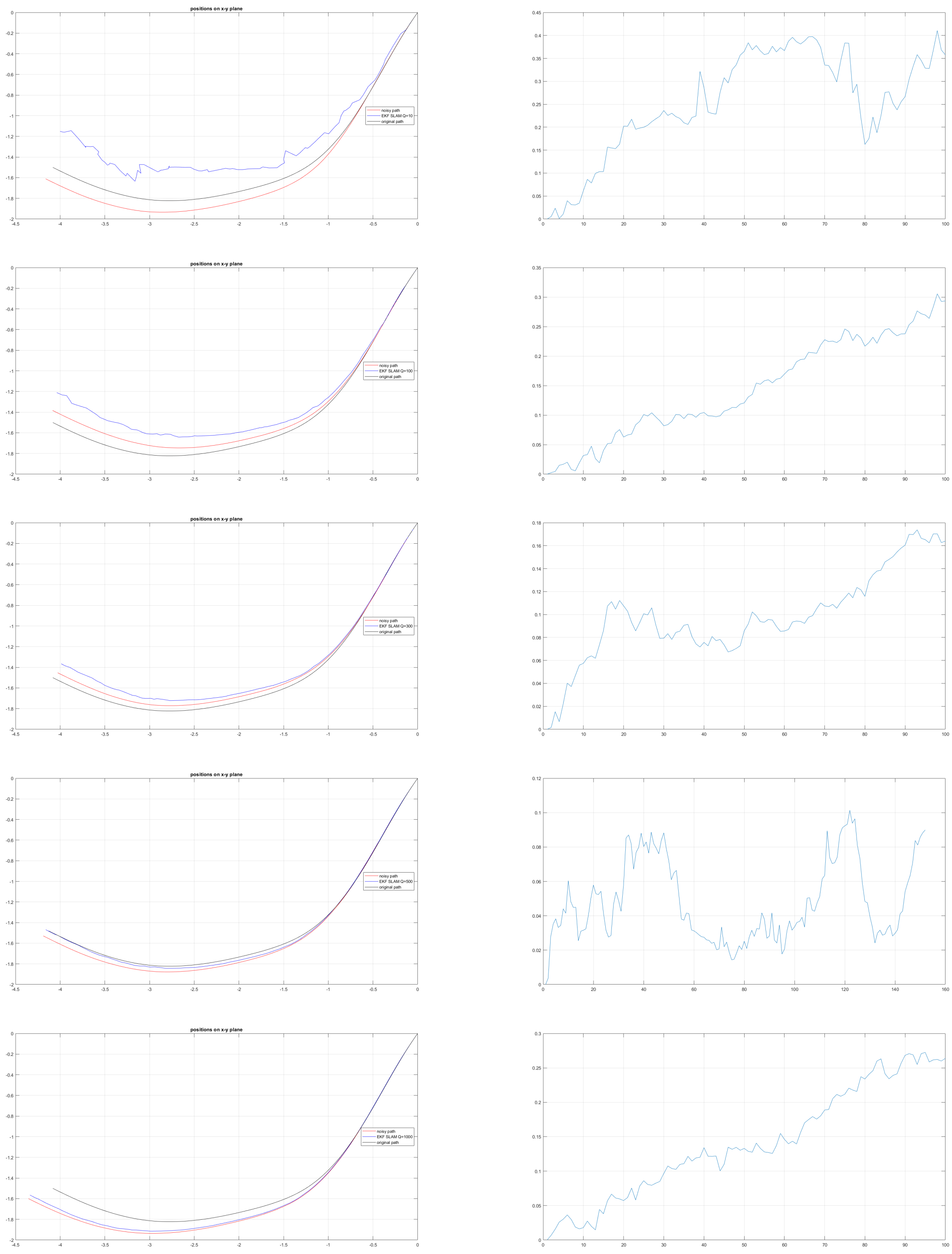


Figure 5.21: EKF SLAM, different Q comparison

It must be pointed out that a low value of the diagonal terms in the matrix Q was not feasible, due to an mount of uncertainty in the sensor measurements introduced with the assumptions done during the whole simulation:

- Data Association: the data association algorithm implemented in this work is a basic one. As said in chapter 4, the Data Association here uses different assumption and simplifications, which are taken into account by higher uncertainty values over the measurements;
- A priori data acquisition: both the Lidar and the StereoCam measurements are taken in the first part the simulation along the nominal trajectory. This translates in the fact that the measurements are not taken in the exact pose of the rover during the EKF SLAM algorithm, so they are affect by a certain level of uncertainty. Again, this is taken into account with higher values of the matrix Q.

Once the behaviour of the Extended Kalman filter SLAM when varying the measurement uncertainty matrix Q has been analyzed, it is possible to proceed analyzing other behaviours of the algorithm.
In particular, the work is going to focus itself on the relationship between the distance covered by the rover and the behaviour of the error from the nominal desired trajectory.

5.3.5. EKF SLAM error with distance

As observed before, during the simulation the error accumulates and slowly diverges from the nominal trajectory desired. As shown in figure 5.17, for the first part of the trajectory the Extended Kalman filter SLAM oscillates around the nominal trajectory, while, after a certain amount of distance covered by the rover, the EKF SLAM is not able anymore to correct itself and starts slowly to diverge.

An explanation of this phenomenon relies in the nature of the Extended Kalman filter SLAM formulation itself. In section 3.4, when exposing the EKF properties, it has been mentioned that the Extended Kalman filter works under the linearization assumption. This assumption, together with the structure of the algorithm, does not accept uncertainties that go over a certain threshold.
When the uncertainty grows over a certain value, the correction step is not able anymore to make a correct use of the measurements data. Moreover, the correction power at each step is limited by the Kalman Gain and the matrix Q, so when the error grows over a certain value, the correction done by the algorithm in not enough to reach the nominal trajectory anymore. The correction step still tries to adjust the trajectory, however the amount of error is not decreased enough and the error in not reduced, growing again at the next step.

Another possible cause of errors along the development of the simulation comes from the Data Association, as already analyzed in chapter 4. The more the error accumulates, due to the causes in the previous section, the more the Data Association algorithm is prone to fails or wrong associations. In particular, wrong associations are very powerful sources of errors and when they happen it is almost impossible to correct them.
When a wrong association happens and the mean vector μ and the covariance matrix Σ are updated according to it, the trajectory inevitably absorb a source of errors, which is going to affect the results until the end of the simulation.
This happens because at every association also the pose of the rover is updated: when a wrong association is incorporated, the pose of the robot is updated in a wrong new position. However, this updated position is in some ways worst than the not corrected one and the correct pose becomes very difficult to recover during the future steps.
Figure 5.22 shows the development of the trajectory, highlighting the possible errors sources with red circles:

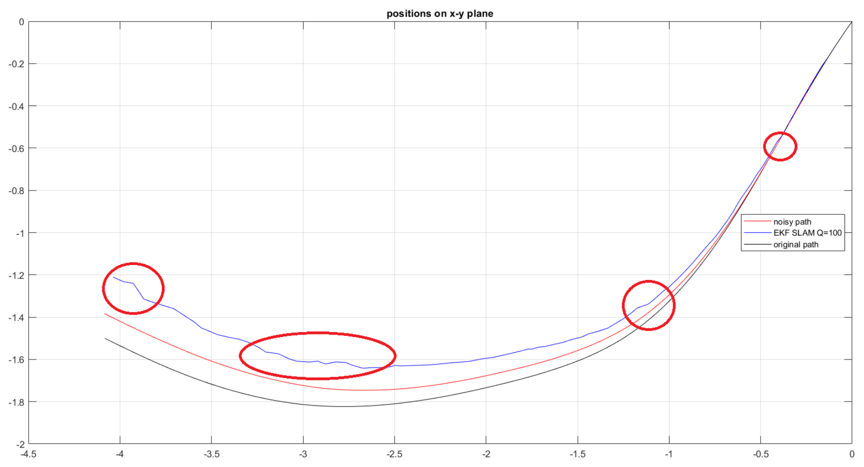


Figure 5.22: EKF SLAM errors with distance

The figure highlights the major error source points with red circles: as can be noted they progressively enlarge the error with cuspids in the trajectory. This behaviour can be referred to wrong or uncertain data association, as explained in section 5.3.5.

The growing of the error with the distance covered by the rover may sound counter intuitive while reasoning about the SLAM. The Extended Kalman filter SLAM has been introduced to reduce the odometry errors and make usage of the sensor readings in order to compute the correct rover pose. As one may think, the SLAM is supposed to reduce the error, with an oscillatory behaviour, along the entire trajectory.
However, as said above, the errors accumulates even with an efficient and performing SLAM and inevitably there is going to be a point in which the algorithm is not able to correct itself anymore. The only feature of SLAM which is able to correct the error even when it becomes too big for the correction step is the *Loop Closure*. The principles of the Loop Closure have been illustrated at the end of chapter 3: the drift of the wheels, and so the errors, are going to accumulate for autonomous vehicles without loop closures detection algorithms, especially in long term moving scenarios, as explained in [7].

5.3.6. EKF SLAM error with trajectory shape

The error accumulated during the SLAM without loop closures can be analysed also as function of the shape of the trajectory. Figure 5.23 shows a straight and a curved trajectory, together with the respective error behaviour:

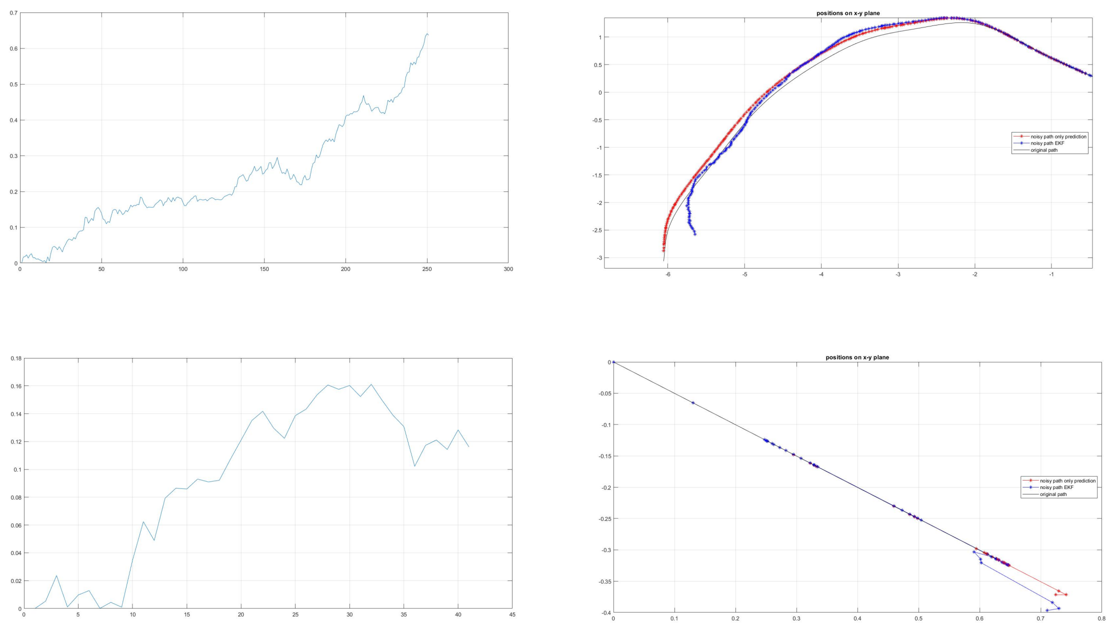


Figure 5.23: EKF SLAM error with trajectory shape

As can be clearly noted from the figure, the curved trajectory accumulates errors at a faster rate than what happens for a straight trajectory. This is a typical behaviour of SLAM, in which the error over straight trajectory grows almost linearly and the error along more complicated trajectory, as curved ones, grows with a faster rate.

Figure 5.24 shows this expected behaviour from the experiments of Nielsen[1]: it can be noted the linear growth of the error in the straight trajectory and the quadratic growth of the error inside the curved path. It can also be noted how the error at the end of the simulation over the straight trajectory stands at lower values than what happens for the curved one.

[1]https://www.youtube.com/watch?v=WV3ZiPqd2G4t=408s

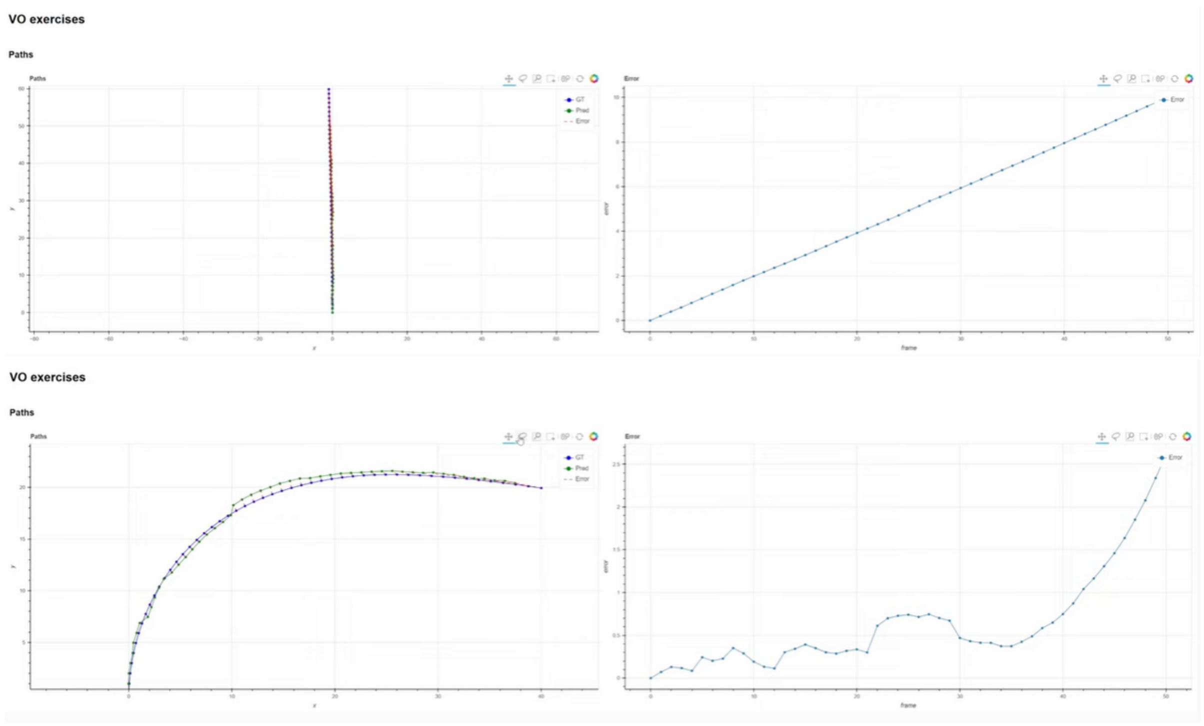


Figure 5.24: Expected errors for different path shapes

6 | Conclusions and future developments

The work presented so far had the aim of recreate a truthful planetary surface simulation environment, together with a rover and its sensors. The rover had then the objective to follow a desired trajectory inside the simulation and acquire data from its sensors; the data collected has been then used to test an Extended Kalman filter Simultaneous Localization and Mapping algorithm.
The desired outcomes from this thesis work have been achieved and can be divided in three main areas:

1. *Simulation environment*: by means of the cooperation between the software of Blender, Unreal Engine and Simulink, it has been possible to reproduce a detailed Mars surface and also to simulate realistic sensor acquisitions;
2. *Sensor data elaboration*: both in the case of the Lidar sensor and the StereoCam sensor it has been possible to recreate the process of elaboration of the sensor data from the raw acquisition to a more refined version to be fed to the SLAM algorithm;
3. *SLAM algorithm*: both the realistic simulation environment and the sensor data elaboration allowed to simulate an Extended Kalman filter SLAM with the problems of a real SLAM: the imperfections in the sensor acquisitions and in the data association brought the SLAM to a realistic level and implementation in which the final results were satisfying.

It is important to highlight what mentioned in the last point: the main outcome of this simulation is the possibility to recreate real problems of a planetary SLAM in a virtual environment. Thanks to this result, it is possible to study and analyzing the algorithm behaviour before its hardware implementation and so correct what needed before the rover construction, allowing a more economic, both in time and money, construction approach to planetary rovers.

6.1. Simulation problems and possible developments

Even if the thesis has brought to the desired results, it still is a basic level work in some of its features: the Data Association, the Loop Closure problem and many other areas can be analyzed in a deeper way in order to improve the quality of the results.
The work done can be developed both in the area of SLAM simulation and also in other robotics applications. The following sections illustrate some of the main problem of the current work and their possible improvements and developments.

6.1.1. Data Association problem

As mentioned in chapter 4, the Data Association implemented in this work is a simple, basic one. This has been done in order simply to test the functioning of the SLAM, but without focusing on the Data Association problem as the main objective.
More appropriate implementations of the Data Association can be analyzed and adopted in order to relax some of the assumption made through the work and obtain better and more robust results.
A possible solution can be the exploitation of other data from the sensor, different from merely the geometric ones: having both the StereoCam and the Lidar, it is possible to collect data regarding the reflectivity of the observed surfaces, the direction of the normal vector to those surfaces or their color. By fusing those information a better and more robust Data Association can be performed, as in the work by Bogoslavskyi [4]. Figure 6.1 shows different possible data exploitations for solving the Data Association problem:

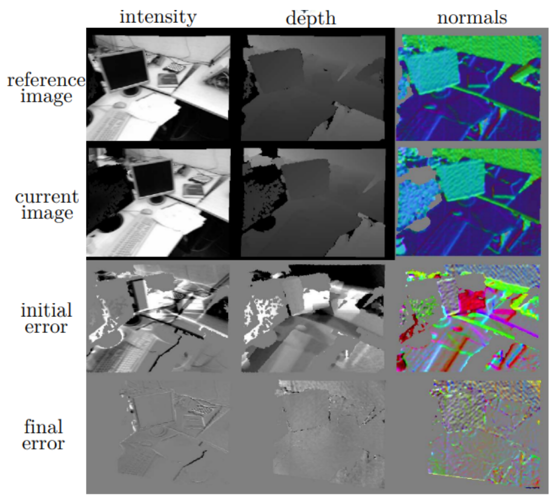


Figure 6.1: Different sensor data exploitations

6.1.2. Different SLAM implemetations

Chapter 1, introduced the SLAM problem and its probabilistic formulation. In chapter 3, the problem has been solved by means of the historically most common method for solving SLAM problems: the Extended Kalman filter method.
However, the SLAM problem can be solved in its probabilistic form by other methods, such as the Information filter method or the Unscented Kalman filter method. These solution exploit different properties of the formulation, bringing to different results, especially on the computational cost side.
The SLAM can be also formulated in other different ways and solved with methods such as Particle filters or Graphs Optimization techniques. Many techniques has been implemented and tested during the years and more advanced ones are explored nowadays.
The implementation of one of these techniques instead of the EKF SLAM could bring to better and more robust results for the entire simulation.

6.1.3. More realistic simulation environment

Once better techniques for the Data Association and for the SLAM solver are available, it is possible to improve the representation of the real environment inside the simulation. This can be achieved by modifying the simulation surface with ore detailed features, different light conditions or moving the simulation from a 2D plane to a 3D environment with different slopes and a rover moving also up and down.
It must be pointed out. however, that these improvements on the reality side of the simulation must be followed by a more powerful hardware on which to perform the simulation. This is due to the fact that the degrees of freedom are going to increase together with the more realistic features, increasing the simulation costs by non negligible factors.

6.1.4. Search for increased robustness

The algorithm in this thesis reached its purpose. However, it shows to be not robust against increased uncertainties or in longer trajectories covered by the rover.
Different trajectories can drive through larger errors depending on the different factors listed at the end of chapter 5. A more robust version of the algorithm could be achieved by performing some of the improvements listed in this chapter.

6.1.5. Sensor data elaboration

The two sensor explored in this work acquire data in different forms, but both the data flows are constructed to arrive in the form of point clouds. The point clouds are then elaborated in the same way in order to define clusters as landmarks and make the algorithm run. The algorithm has been tested separately for the StereoCam and the Lidar data. However, the two data can be elaborated in different ways, for example not transforming the StereoCam data into point clouds and instead exploiting features like the intensity of the light observed.
This approach can lead to a *sensor-fusion* method for the Data Association and the whole algorithm: in this way different data from the two sensor can be fused together to reach better and more robust final results.

6.1.6. In loop simulations

The simulation in this work has been divided in three sections, each of them separated from the others. The collection of data has so been done a priori with respect to the SLAM implementation. This has been taken into account inside the sensor uncertainty matrix Q.
A possible different approach could consider merging the three part together, obtaining the simulation of real time more truthful data acquisition for the sensors. Figure 6.2 shows a possible architecture for this kind of simulations: the simulation environment outputs feed directly the EKF, which in return feed back the new pose to the simulation environment and so on.

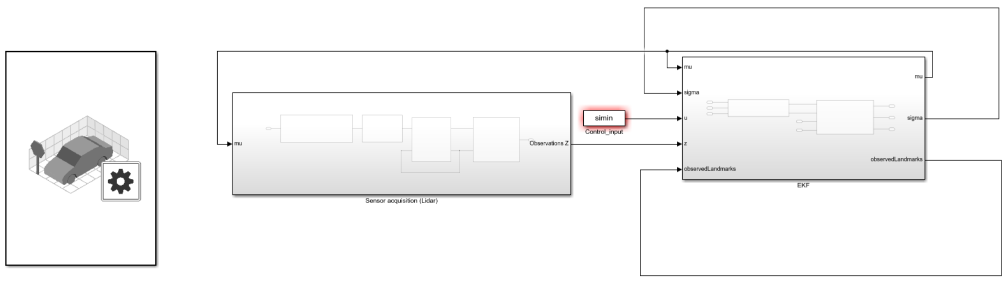


Figure 6.2: In loop simulation architecture

6.1.7. Other applications

The simulation approach proposed has been exploited for SLAM algorithms solutions. However, this approach can be adopted also for the simulation of other robotic and/or autonomous task, reconstructing the virtual environment and the dynamics of the robotic actor between the softwares listed.
For example, a possible application is to combine the planetary SLAM of this work together with a Path Planning algorithm and so studying the behaviour of a rover which has to decide both how to move inside an unknown environment, localize itself and map the environment in order to take the next steps.
Another possible approach is the simulation of SLAM for orbital maneuvers: for example a spacecraft orbiting a small celestial can be simulated and how its attitude can be determined by the use of a SLAM algorithm.
An example of another similar application is the simulation of the attitude control of satellite formations or tethered systems based again on a SLAM approach.

6.2. Conclusion

Exposing the possible improvements which can be done on this work in order to obtain more evolved results, this thesis has come to an end.
As illustrated in all the previous chapters, a realistic simulation environment has been built with a certain degree of detail. The simulated planetary surface has then been used to simulated a rover mission exploring it and the sensor acquisitions have been used for implementing an Extended Kalman filter algorithm.
The results obtained were in line from the results expected, keeping in mind that the whole SLAM was built with the assumption of not closing any loop. The behaviours of the algorithm parameters has been then analyzed in order to get a deeper knowledge of the results.

In the end, future developments have been suggested, acknowledging that the simulation architecture here presented is only a starting point for more advanced work and it is far for a final definitive form.

Bibliography

[1] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9, 1987.

[2] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. *IEEE Robotics and Automation Magazine*, 13, 2006.

[3] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, and John J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32, 2016.

[4] Bartolomeo Della Corte, Igor Bogoslavskyi, Cyrill Stachniss, and Giorgio Grisetti. A general framework for flexible multi-cue photometric point cloud registration. 2018.

[5] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping (slam): part i the essential algorithms. *Robotics and Automation Magazine*, 2, 2006.

[6] David Filliat. A visual bag of words method for interactive qualitative localization and mapping. 2007.

[7] Yang Li, Wanbiao Lin, Tianjun Zha, Zhenghong Jiang, Peiwen Li, and Lei Sun. Efficient loop closure detection method for lidar slam in challenging environment. 2021.

[8] J. N. Maki, D. Gruel, C. McKinney, M. A. Ravine, M. Morales, D. Lee, R. Willson, D. Copley-Woods, M. Valvo, T. Goodsall, J. McGuire, R. G. Sellar, J. A. Schaffner, M. A. Caplinger, J. M. Shamah, A. E. Johnson, H. Ansari, K. Singh, T. Litwin, R. Deen, A. Culver, N. Ruoff, D. Petrizzo, D. Kessler, C. Basset, T. Estlin, F. Alibay, A. Nelessen, and S. Algermissen. The mars 2020 engineering cameras and microphone on the perseverance rover: A next-generation imaging system for mars exploration. *Space Science Reviews*, 216, 2020.

[9] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. volume 1, 2009.

[10] Zhuli Ren, Liguan Wang, and Lin Bi. Robust gicp-based 3d lidar slam for underground mining environment. *Sensors (Switzerland)*, 19, 2019.

[11] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* MIT Press, 2005.

[12] Alex Zelinsky. Learning opencv—computer vision with the opencv library, 2009.

[13] Qian Yi Zhou, Jaesik Park, and Vladlen Koltun. Fast global registration. volume 9906 LNCS, 2016.

A | Implementing the SLAM

This chapter is going to describe how the codes were thought and developed from the simulation until the EKF SLAM testing. The focus of the chapter is mainly on the Matlab codes, since the work done with Blender, Unreal Engine and Simulink has already been well described in chapter 2.4. Here below an example on how the codes are going to be shown:

```
This is
an example
code
```

Listing A.1: Example code

In the next section the codes structure for each of the simulations part is going to be described and explained.

A.1. Sensor data generation

This section is going to illustrate the implementation of what already described in algorithm 2.3. As a brief recall, this part of the work has the purpose of simulating the rover movement inside the reconstructed environment and guarantee the storage of the sensors acquisitions. The main part of this task are achieved by the capabilities of the Simulink and Unreal Engine simulations, however a Matlab code has been written to facilitate and automatize this procedure.
The code mentioned has three main purposes:

1. Setup the map dimensions, which are going to be kept from here to the end of the SLAM simulation;
2. Generate the rover desired trajectory and so the control inputs;
3. Store the data streams from the sensors.

The first task is achieved by importing in the Matlab environment a bird-eye view of the simulation surface and set its dimensions to the one desired:

```matlab
% Generates a 2D projection of the scene with the limiting coordinates
    on x and y:
sceneImage = imread('HighresScreenshot00002.png');
imageSize = size(sceneImage);
xlims = [-7.5 7.5];      % in meters
ylims = [-7.5 7.5];      % in meters

sceneRef = imref2d(imageSize,xlims,ylims);

sceneRef.XWorldLimits   % in meters
sceneRef.YWorldLimits   % in meters

hScene = figure;
imshow(sceneImage,sceneRef)
title('2D map projection')
```

Listing A.2: Map dimensions setup

Once the dimensions of the map are set, it is possible to proceed to the second task and draw manually the trajectory desired. This is achieved by calling a function from MathWorks, which opens an interface over the bird-eye view of the map and enables the manual drawing:

```matlab
% Select waypoints to set the trajectory by calling the
    helperSelectSceneWaypoints function:
hFig = helperSelectSceneWaypoints(sceneImage, sceneRef);
```

Listing A.3: Drawing of the trajectory

The function is shown in appendix **??** and the result is shown in figure A.1.
After the trajectory waypoints have been saved in the Matlab workspace, the trajectory is smoothed in order to simulate in a better way a real movement. The smoothed poses are then transformed into time-series variables and fed to the Simulink and Unreal Engine models in order to start the data acquisition simulation. Here the simulation time is also set.
At the same time, the control inputs over each of the steps in the trajectory is computed by inverting an odometry motion model: from the poses the algorithm computes the different inputs given to the rover.

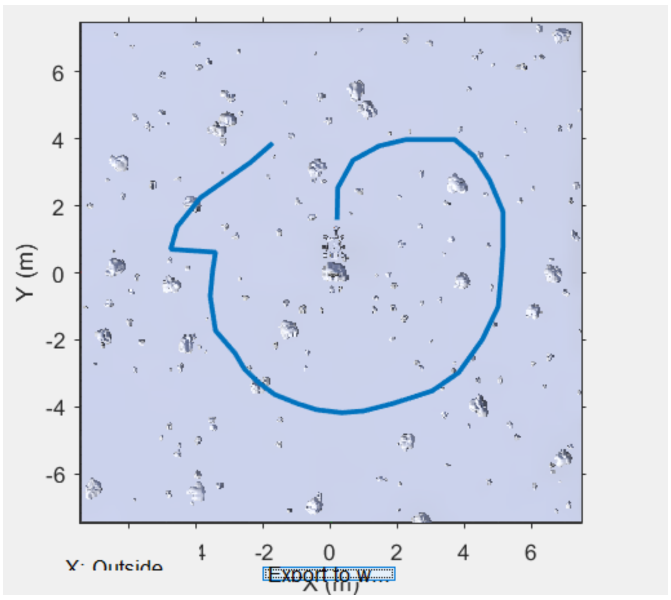


Figure A.1: Rover trajectory drawing

The inverse motion model is shown here:

```
for i=2:length(time)
    delta_trans_curr = sqrt((x(i)-x(i-1))^2 + (y(i)-y(i-1))^2);
    if x(i)>x(i-1)
        delta_trans(i-1) = delta_trans_curr;
    elseif x(i)<x(i-1)
        delta_trans(i-1) = - delta_trans_curr;
    end
    delta_rot1(i-1) = atan((y(i)-y(i-1))/(x(i)-x(i-1)))- theta(i-1);
    delta_rot2(i-1) = theta(i)-theta(i-1)-delta_rot1(i-1);
end
```

Listing A.4: Trajectory to Control Input

Where x, y and $theta$ are respectively the variables $refPosesX$, $refPosesY$ and $refPosesT$. The code above is called by the function:

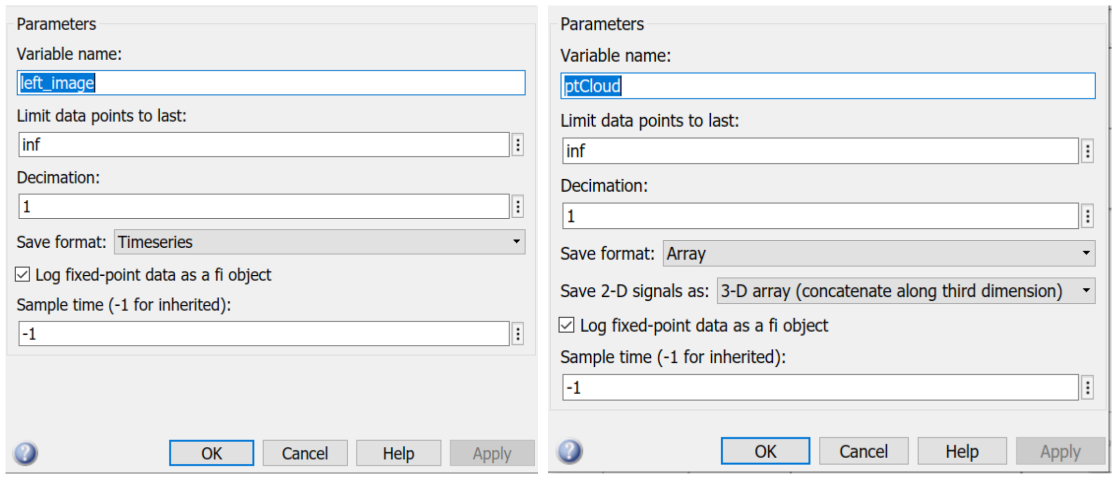
```
[control_input] = odometry_motion_model(refPosesX, refPosesY, refPosesT)
    ;
```

Listing A.5: Calling of trajectory to control input

At the end of this section the useful data from the StereoCam and the Lidar are stored inside the workspace and ready to be analyzed.

A.1.1. Sensor data storage

A brief digression on the type of data coming from the simulated sensor is worth at this point of the thesis in order to clarify what to expect and justify some of the steps which are going to be followed in the next sections.
The output type from the sensors can be set inside the relative Simulink blocks, as shown in figure A.2:


Parameters
Variable name: left_image
Limit data points to last: inf
Decimation: 1
Save format: Timeseries
☑ Log fixed-point data as a fi object
Sample time (-1 for inherited): -1

Parameters
Variable name: ptCloud
Limit data points to last: inf
Decimation: 1
Save format: Array
Save 2-D signals as: 3-D array (concatenate along third dimension)
☑ Log fixed-point data as a fi object
Sample time (-1 for inherited): -1

Figure A.2: StereoCam and Lidar output data types

As the figure shows, the data from the StereoCam is stored inside two *Timeseries* variables: one for the left camera and one from the right camera. At the same time, the data stream from the Lidar is stored inside a structure of point clouds, one for each of the acquisitions.
Regarding the StereoCam, in the next steps, a conversion of the data in point clouds is going to be needed together with the merging of the data coming from the two cameras. Meanwhile, since the Lidar data are already in the point cloud form, they are ready for the further elaborations in the second part of the simulation.

A.2. Sensor data to point clouds and landmarks

Algorithm 2.4 describes how the raw data from the sensors are transformed in useful data for the SLAM simulation.
To do so a division in the process between the Lidar data and the StereoCam data must be performed, since they require different elaboration techniques.

A.2.1. StereoCam data elaboration

As already mentioned in the previous section, the StereoCam data come in the form of two *Timeseries* variables, one for each of the two cameras.
The first step for the elaboration of the StereoCam data is to find the camera extrinsic and intrinsic parameters, step which is done with the calibration procedure described in section 2.2.4 and which is not going to be repeated here. The results from the calibration procedure are necessary in the pivotal following step: the *Disparity Map* computation.

The *Disparity Map* is a 2D map reduced from a 3D space, in which the brightness of the gray scaled pixels indicates the distance of the related points from the sensor. The description of how a disparity map is computed from a couple of stereo images is beyond the purpose of this thesis. However, it is important to note that it depends on the setup of a couple of parameters and on a function which is called by the following commands:

```
range = [-16 16];                       % First parameter
disparityMap_stored = cell(d, 1) ;   % Second parameter

for k = 1:d
    img1_struct.Data(1:200,:,:,k) = 0; %select the left image at time t
    img2_struct.Data(1:200,:,:,k) = 0; %select the right image at time t

    % transform the two image in gray scale and rectify them:
    J1 = im2gray(img1_struct.Data(:,:,:,k));
    J2 = im2gray(img2_struct.Data(:,:,:,k));
    [I1,I2]=rectifyStereoImages(J1,J2,stereoParams,'OutputView','valid')
   ;
    % compute the Disparity Map:
    disparityMap = disparityBM(I1,I2,'DisparityRange',range,'
   UniquenessThreshold',25);
    disparityMap_stored{k,1} = disparityMap; %store the result
end
```

Listing A.6: Disparity Map computation

The disparity map computation is fundamental for evaluting the quality of the data generated and it is going to be necessary for the transformation of the images into useful point clouds.

The Disparity Map is computed by means of the function:

```
disparityMap=disparityBM(I1,I2,'DisparityRange',range,'
   UniquenessThreshold',25);
```

Listing A.7: Disparity Map Block Matching function

The function used for the Disparity Map computation utilizes an *Block Matching algorithm* to which the parameters *Range* and *Uniqueness Threshold* have to be fed.
By playing around with the *Range* and *Uniqueness Threshold* parameters, it has been possible to find a satisfying configuration for the Disparity Map computation, as shown in figure A.3. The Disparity Maps computed at each step were then saved inside a structure variable.

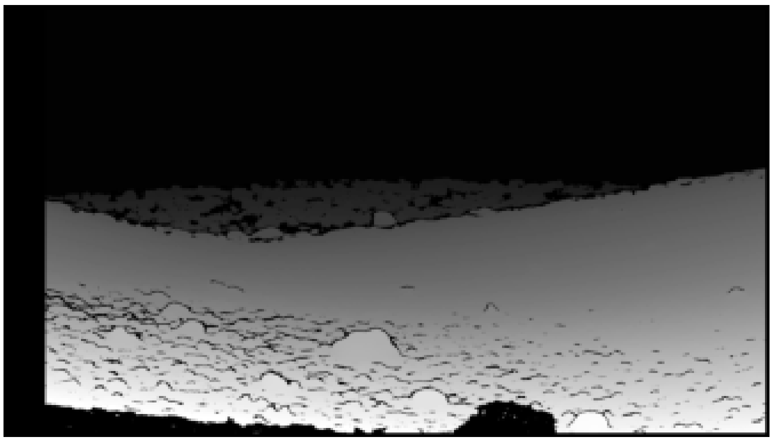


Figure A.3: Disparity Map

After the Disparity Map computation it is transformed into a point cloud by calling the function:

```
StereoPt_stored = stereo2pc(disparity_mat, stereoParams);
```

Listing A.8: Disparity Map to point cloud

Now both the Lidar and StereoCam data are organized in the form of structures of point clouds variables and their further elaboration can be done simultaneously.

A.2.2. Point clouds clustering in landmarks

Now that both the sensors data are in the form of points cloud, they can be elaborated in such a way to obtain cluster of points to be identified as landmarks. In order to that an approach based on the Euclidean distance between point, the angular distance between them and the number of points in each cluster has been adopted. In particular:

- *distThreshold* indicates the Euclidean distance under which the points of the same cluster must be;
- *angleThreshold* indicates the minimum angle of separation between two different clusters;
- *NumClusterPoints* indicates the minimum and maximum number of point accepted for a cluster.

These parameters must be set before calling the following function and cluster the points inside the sensors point clouds into landmarks:

```
% Parameters setup:
distThreshold = 0.005;
angleThreshold = 30;
NumClusterPoints=[600,Inf]; %only minimum number of points set

% Lidar point cloud clustering:
[ptCloud_Lidar_clustered] = landmarks_clustering_Lidar(ptCloud_Lidar,
    distThreshold,angleThreshold,NumClusterPoints);

% StereoCam point cloud clustering:
[ptCloud_Stereo_clustered] = landmarks_clustering_Stereo(StereoPt_stored
    ,distThreshold,angleThreshold,NumClusterPoints);
```

Listing A.9: Point cloud clustering

Here the detailed functioning of the function is not illustrated, however it can be found on MathWorks website []. Also in this case, the different tuning of the three parameters brings out different results in the clustering, with a number of landmark identified proportional to the strictness of the parameters chosen.
The outcomes from the different setups is illustrated in chapter **??**, while figure A.4 illustrates an example of clustering.

The figure shows also the problem illustrated in the chapter 4.1.3: between the clusters it is possible to clearly distinguish parts of the rover as, for example, the wheels. These are the parts that need to be filtered out before proceeding with the Data Association.
Once the clusters, or landmarks, are defined, it is possible to proceed with the identifica-

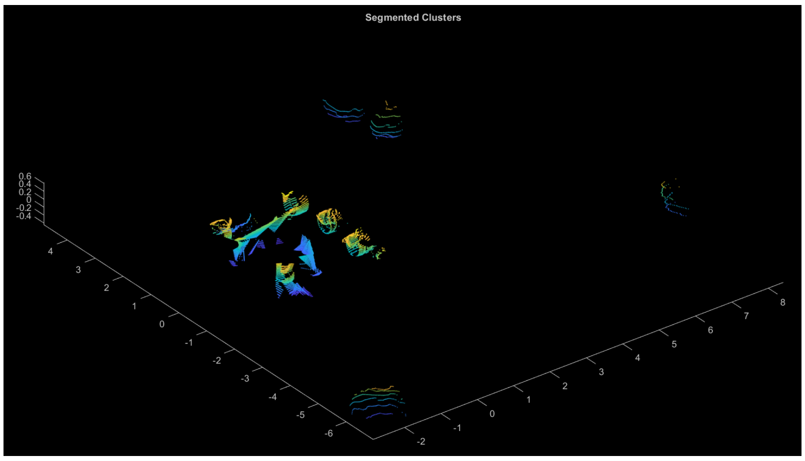


Figure A.4: Clusters example

tion of the landmark geometric centre and its coordinates.

A.2.3. Landmarks coordinates

The following step is a simple geometric computation of the center point coordinates of each of the clusters. This computation is achieved by the steps here below:

```
for i = 1:size(ptCloud_Lidar_clustered,2)
    for k=1:ptCloud_Lidar_clustered(i).numClusters
        pc_current = select(ptCloud_Lidar_clustered(i).pt_cloud,find(
    ptCloud_Lidar_clustered(i).labels == k));
        s = pc_current.Count;
        x_centre = sum(pc_current.Location(:,1)/size(pc_current.Location
    (:,1),1));
        y_centre = sum(pc_current.Location(:,2)/size(pc_current.Location
    (:,2),1));
        z_centre = sum(pc_current.Location(:,3)/size(pc_current.Location
    (:,3),1));
        theta = atan(y_centre/x_centre)*(180/pi); %[deg]
    end
end
```

Listing A.10: Landmark center coordinates

Each of the coordinates computed is then stored in the same structure where each point cloud is stored, in a dedicated field for the coordinates of each of the landmarks in every point cloud. At this point the set of the total number of acquired point clouds from the

sensors is available, together with the coordinates in the rover local frame of the landmarks inside them.
It is now possible to move on in the actual EKF SLAM algorithm simulation.

A.3. Extended Kalman Filter SLAM implementation

Now that the control inputs and the clustered point clouds re available, the SLAM algorithm can be implemented and tested.
First of all, a variable initialization is required before the start of the actual algorithm:

```
% Sensor acquisition:
load('clustered_landmarks.mat')

% Control Input:
load simulation_data.mat control_input

% Initialize mean and covariance:
N = 50;
INF = 1000;

observedLandmarks = repmat(false,1,N); % Observed landmarks vector
mu = repmat([0.0], (2*N+3), 1);        % Mean

robSigma = zeros(3);
robMapSigma = zeros(3,2*N);
mapSigma = INF*eye(2*N);
sigma = [[robSigma robMapSigma];[robMapSigma' mapSigma]]; %Covariance
```

Listing A.11: EKF SLAM initialization

As can be noted, the dimension of the mean vector and the square covariance matrix is set to $(2N+3)$, with a sufficiently big number N of forecasted landmarks in order to avoid landmarks labelled with a number bigger the maximum number accepted N.
The diagonal elements of the covariance matrix from the fourth row on are set to an high value in order to simulate the total uncertainty about them at the starting point of the simulation (theoretically they must be $+\infty$ at this point).

After this preliminary step, it is possible to proceed with the Extended Kalman filter algorithm itself. As shown in algorithm 3.3, the EKF SLAM is a recursive procedure, repeating the prediction step and the correction step at each iteration. This recursion is represented by a FOR loop in which the steps alternates themselves:

```
for i = 2:%(number of point clouds)
    u.d_rot1 = control_input.drot1(i-1);
    u.d_trans = control_input.dtrans(i-1);
    u.d_rot2 = control_input.drot2(i-1);

    % error introduction:
    amplitude = 0.01;
    u.d_trans = control_input.dtrans(i-1) + amplitude*randn(1,1);

    % Prediction step:
    [mu, sigma] = prediction_step(mu, sigma, u);

    % Data association:
    pc_first = ptCloud_Lidar_clustered_landmarks(i-1);
    pc_current = ptCloud_Lidar_clustered_landmarks(i);

    [pc_first] = filter_out_rover (pc_first);
    [pc_current] = filter_out_rover (pc_current);

    [R, T] = ls_arun_huang(pc_first, pc_current);
    [id_def, score] = observed_landmarks(pc_first, pc_current, R, T);
    [z] = observation_association(pc_current,id_def);
    [z, associations] = best_association (id_def,score, pc_current);

    % Correction step:
    [mu, sigma, observedLandmarks] = correction_step(mu_n, sigma_n, z,
   observedLandmarks);

    sprintf('iteration number: %f',i)
end
```

Listing A.12: EKF SLAM loop

The first steps introduce the control input in a more useful structure array form, followed by the introduction of an error over the control. The error is reproduced as a random variation over the translation control input. This variation is constrained between $[-0.01, 0.01]$ meters.

A.3.1. Prediction step

The prediction step procedure is introduced in line 11 of the previous code by calling a dedicated function. The function operates in the following manner:

```
function [mu, sigma] = prediction_step(mu, sigma, u)

nn = length(mu);
F_x = [eye(3), zeros(3,nn-3)];

d_rot1 = u.d_rot1;
d_trans = u.d_trans;
d_rot2 = u.d_rot2;

theta_prev = mu(3);

% Reconstruct the motion from the control inputs:
motion = [d_trans * cos(mu(3)+d_rot1); d_trans * sin(mu(3)+d_rot1);
    d_rot1 + d_rot2];
S = F_x';
mu = mu + (F_x')* motion;
[theta] = normalize_angle(mu(3));
mu(3) = theta;

% Compute the 3x3 Jacobian Gx of the motion model
a = -d_trans * sin(theta_prev + d_rot1);
b =  d_trans * cos(theta_prev + d_rot1);
Gx_t = [0 0 a; 0 0 b; 0 0 0];
G = eye(nn) + (F_x')*Gx_t*F_x;

% Motion noise
motionNoise = 0.01;
R3 = [motionNoise, 0, 0;
     0, motionNoise, 0;
     0, 0, motionNoise/10];
R = zeros(size(sigma,1));
R(1:3,1:3) = R3;

% Compute the predicted sigma after incorporating the motion
sigma = G .* sigma .* G' + R;
end
```

Listing A.13: Prediction step

As already mentioned in section 3.4.3, a matrix F_x is introduced at the beginning of the algorithm. The structure of the matrix in shown in algorithm 3.3 and here it can be noted how its structure is exploited to allow easier updates of the mean vector μ by simple matrix multiplications.
The motion model described in section 3.4.1 is then applied to the mean vector, updating only the first three elements (the robot pose) thanks to the multiplication by matrix F_x. The code lines from 19 to 23 build up the Jacobian matrix G of the motion model and, after the Gaussian motion noise matrix R, the covariance matrix Σ can be updated following the rules again of algorithm 3.3.

A.3.2. Data Association

Lines 13 to 23 of the SLAM loop code tackle the Data Association problem with the procedure explained in chapter 4. The first action is to select the currently seen point cloud and the one seen before in order to confront them, subsequently the rover parts and the landmarks too far from it are filtered out from both the point clouds in lines 17 and 18. The filtering of the rover parts is described here:

```matlab
a = -2; % [m]
b = +2; % [m]

for i = 1:pc_current.numClusters
    xl = pc_current.landmarks(i).xc;
    yl = pc_current.landmarks(i).yc;
    zl = pc_current.landmarks(i).zc;

    if xl>a & xl<b
        if  yl>a & yl<b
            wrong = [wrong, i];
        end
    end
end
```

Listing A.14: Filtering out the rover

The procedure to remove also distant obstacles is identical to this, except for the distance threshold values. Once the rover is filtered out, the Huang-Arun algorithm is applied in order to obtain the rotation matrix R and the translation vector $\vec{t}$ between the two point clouds. The computation of these two elements is done following the exact procedure illustrated in algorithm 4.2, so it is not going to be shown here in its code form.

After that the core of the Data Association is presented: from line 21 to 23 the list of observed landmarks is computed together with each score, then the associations are done and the final best association is selected. Line 21 introduces the function:

```matlab
[id_def, score] = observed_landmarks(pc_first, pc_current, R, T);
```

Listing A.15: Observed landmarks function

This function has the aim of firstly superposing the two point clouds and secondly, for each cluster in the current point cloud, scans the clusters of the previous point cloud in order to assign a score to each possible association. This is achieved by:

```matlab
for i = 1: pc_first.numClusters
    for k = 1: pc_current.numClusters
        land1 = select(pc_first.pt_cloud,find(pc_first.labels == i));
        l1 = land1.Location;
        land2 =select(pc_current.pt_cloud,find(pc_current.labels == k));
        l2 = land2.Location;
        [m,n] = size(l2);

        l_back = [];
        for l = 1:m
            l_back(l,:) = ( -T + R\l2(l,:)')';
        end
        l1 = pointCloud(l1);
        l2 = pointCloud(l_back);

        B = squeeze(l1.Location);
        features1 = double(B);
        features2 = l2.Location;

        [indexPairs, scores] = pcmatchfeatures(features1,features2);

        [z,m] = size(indexPairs);
        if z <= 300
            scores = NaN;
        end

        score_now = mean(scores);
        score(i,k) = score_now;
    end
end
```

Listing A.16: Cluster association scoring

At the beginning a FOR loop in initialized where each of the clusters in the current point cloud is selected and iteratively compared with all the clusters observed in the previous point cloud. A particular attention on the form of the data must be paid since the point clouds and the clusters are now in the form of *ordered point clouds* variables. This is going to be important in the next steps.
Lines 9 to 14 apply the rotation matrix R and translation vector $\vec{t}$ to the current point cloud in order to move it back over the previous point cloud. In order to do so the structure of the data must be changed and then put again in the form of ordered point clouds variables. This is done and refined with the help of lines 16 to 18.
At this point the comparing between the cluster is done by means of a *Fast Approximate Nearest Neighbors* algorithm [9], or *Fast Global Registration* algorithm [13], thanks to the function:

```
[indexPairs, scores] = pcmatchfeatures(features1,features2);
```

Listing A.17: Matching features function

At this stage, the variables *indexPairs* and *scores* obtained are in the form:

- *indexPairs* is a m x 2 matrix in which elements in the first column indicates the indexes of points in the current point cloud matching the indexes in the second column of points in the second point cloud;
- *scores* is a column vector containing the Euclidean distance between each of the couples of points contained in *indexPairs*.

The following IF cycle throws away the scores of the matchings in which the number of matching points is less than a certain threshold, setting the score to be a *Not a Number* value. For each of the matchings compared a mean value of the scores is then computed and stored in the variable *scorenow*. The value found is the score considered afterwards to evaluate the association between the landmark i in the previous point cloud and the landmark k in the current point cloud observed. It is stored in the variable *score*.
The second part of the function is then run: the column of the variable *score* are scanned and for each column the minimum value between the scores is found together with its index. So for each landmark in the current point cloud the minimum score is selected and a first association with a landmark in the previous point cloud is found. After this step a check is performed to match the dimensions of the matrix containing the associations found at each step since the number of landmarks changes from an acquisition to the other.
This part of the function is shown here:

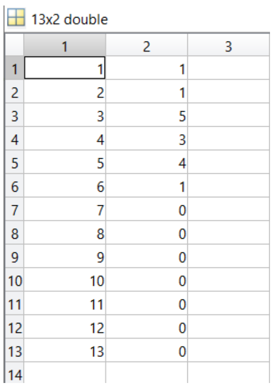
```
[dd,ff] = size(score);
id = [];
for h = 1:ff
    c = min(score(:,h));
    id_c = find(score(:,h)==c);
    id = [id, id_c'];
end

[s,t] = size(id_def);
if s < length(id)
    gg = length(id) - s;
    for f = 1:gg
        id_def(end+1,:) = 0;
    end
elseif s > length(id)
    gg = s - length(id);
    for u = 1:gg
        id(end+1) = 0;
    end
end

id_def = [id_def, id'];
```

Listing A.18: Cluster scoring evaluation

The variable *id def* now contains the associations found as shown in figure A.5. The first column contains the landmarks of the previous point cloud and the second column contains the number of the landmark in the current point cloud associated with each of them. The null values in the second column indicate that no association has been found for those landmarks.


13x2 double

	1	2	3
1	1	1	
2	2	1	
3	3	5	
4	4	3	
5	5	4	
6	6	1	
7	7	0	
8	8	0	
9	9	0	
10	10	0	
11	11	0	
12	12	0	
13	13	0	
14			

Figure A.5: First associations storage variable

As can be noted, some values in the second column repeat themselves more than one time: this means that different landmarks in the current data set have been associated to the same landmark in the previous data set because it is the best associated landmark for all of them. This phenomenon can not be accepted, since it is going to afflict in a bad manner the correction step: the state of a landmark would be corrected more than one time for a single observation and it is not acceptable.
A strategy to avoid the repeating of the association to the same landmark has been adopted is not necessary here since in the following lines the Data Association is completed by assigning the associations to the landmarks and then select only the best association based on the score.
For each point cloud observed only one landmark is so used as a Data Association indicator to be inserted inside the correction step.

A.3.3. Correction step

After the Data Association is solved, the algorithm tackles the correction step of the Extended Kalman filter SLAM. This step in solved by the function:

```
[mu, sigma, observedLandmarks] = correction_step(mu, sigma, z,
   observedLandmarks);
```

Listing A.19: Correction step function

The function receives as inputs the mean vector $\bar{\mu}$ and the covariance matrix $\bar{\Sigma}$ both containing the prediction step updates, together with the observation vector z containing the only associated landmark and the variable *observedLandmark* which serves as memory for saving the identity of the landmark seen.
The internal structure of the function follow the second part of algorithm 3.3 in chapter 3, from step 5 to the end as shown in code A.21.
In line 53 of A.21 the matrix of measurement uncertainty is computed:

```
Q = 200*eye(2*m+3);
```

Listing A.20: Measurements uncertainty matrix

The value of the diagonal terms in matrix Q as been set high: this is done to simulate the fact that we are really uncertain about the measurements updates, considering the low level data association, and we want to weight the correction step importance inside the algorithm less than the one of the prediction step.

```
% Number of measurements in this time step
[nn, ll] = size(mu);
m = size(z.id, 2);

Z = zeros(m*2 +3, 1);
expectedZ = zeros(m*2 +3, 1);
H = [];

for i = 1:m
  % Get the id of the landmark corresponding to the i-th observation
  landmarkId = z.id(i);
    z.bearing(i) = normalize_angle(z.bearing(i));
  % If the landmark is obeserved for the first time:
  if(observedLandmarks(landmarkId)==false)
    % Initialize its pose in mu based on the measurement and the current
    robot pose:
    mu(2*landmarkId+2) = mu(1) + z.range(i) * cos(z.bearing(i) + mu(3));
        mu(2*landmarkId+3) = mu(2) + z.range(i) * sin(z.bearing(i) + mu
   (3));
    % Indicate in the observedLandmarks vector that this landmark has
   been observed
    observedLandmarks(landmarkId) = true;
    end

  % Add the landmark measurement to the Z vector
    Z(2*i+2) = z.range(i);
    Z(2*i+3) = z.bearing(i);

  % Use the current estimate of the landmark pose
  % to compute the corresponding expected measurement in expectedZ:
    delta = [mu(2*landmarkId+2) - mu(1); mu(2*landmarkId+3) - mu(2)];
    q = delta'*delta;
    expectedZ(2*i+2) = sqrt(q);
    angle = atan(delta(2)/delta(1)) - mu(3);
    expectedZ(2*i+3) = angle;

    low_Hi = (1/q)*[-sqrt(q)*delta(1) -sqrt(q)*delta(2) 0 sqrt(q)*delta
   (1) sqrt(q)*delta(2);
                     delta(2) -delta(1) -q -delta(2) delta(1)];
    k = nn;
    L = zeros([2 k]);
    L(1, 2*landmarkId+2) = 1;
    L(2, 2*landmarkId+3) = 1;
  Fx_j = [eye(3), zeros(3,nn-3); zeros(2,3), L(:,4:end)];
```

```
    Hi = low_Hi*Fx_j;
  % Augment H with the new Hi
  H = [H;Hi];
end

H_up = zeros(3,nn);
H = [H_up; H];
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;

% Construct the sensor noise matrix Q
Q = 200*eye(2*m+3);

% Compute the Kalman gain
K = sigma*H'*inv(H*sigma*(H') + Q);

mu = mu + K*(Z - expectedZ);
sigma = (eye(103) - K*H)*sigma;
mu(3) = normalize_angle(mu(3));
```

Listing A.21: Correction step

After the correction step function only few steps remain in the algorithm. They are dedicated to the storage of the several variables, for the preparation of the variables for the plots and for the computation of the error between the EKF SLAM trajectory and the nominal non-noisy trajectory desired.

List of Algorithms

Listings

List of Figures

List of Tables

List of Symbols

Variable	**Description**	**SI unit**
EKF	Extended Kalman Filter	-
KF	Kalman Filter	-
SLAM	Simultaneous Localization and Mapping	-
CML	Concurrent Mapping and Localization	-
μ	Mean vector	-
Σ	Covariance matrix	-
$\bar{\mu}$	Predicted Mean vector	-
$\bar{\Sigma}$	Predicted Covariance matrix	-
K	Kalman Gain	-
R	Motion uncertainty matrix	-
Q	Measurement uncertainty matrix	-
z	Sensors measurements	-
u	Control input	-
x	Rover state vector	-
m	Map	-
g	Motion model function	-
G	Jacobian of the motion model	-
h	Measurement model function	-
H	Jacobian of the measurement model	-
$P()$	Probability of...	-
b	Stereo Baseline	m
f.o.v.	Field of View	°
t	Time	s

Acknowledgements

First of all, I would like to thank my advisor, professor Mauro Massari, for the opportunity he gave me with this thesis and for all the help he gave me during these months. This work has opened my eyes on several interesting engineering topics and space research areas I would have never explored by myself.

I would like to thank my family: my sister, my mother and my father, for all the support given to me in these years. A special thought goes to my grandparents Anna and Benito, they have always been my lighthouse in the world of knowledge since I was a kid.
A big thank you goes to my colleagues at university: Filippo, Lorenzo, Davide, Alessandro and Marcello. They have always been there for help with university subjects and I have plenty of good memories with them.
Incredible amount of appreciation goes to my friends who have been with me for more than a decade: Tex, Diego, Rave, Nando and Jonny, your contribution has been priceless. A special thank you goes to Rudolf, for sure my Kalman Filter master. Impossible to forget to thank my friend Anita for the many hours spent at listening my complaints and thanks also to my surf buddy Elisa. Thanks to all friends who helped me in these years. Incredible thanks to Niccolò, Giulia and Berse: my guides in the adulthood matters.
Thanks to Ksenia, Giorgio, Toma, Gabriel and Giorgio, the best duckies in the world.

It is impossible to quantify the amount of thanks I owe to a special person who I have the fortune to have in my life: for every up and down of this master degree, since that afternoon during the pandemic and even before, thank you Gloria for standing by me.

The work presented in this thesis is the conclusion of a long path. However, I prefer to see it as the starting point for something bigger and just the beginning of my learning process: I would so like to thank professors Marco Quadrelli and Krisptopher Wehage to have accepted me as a colleague in their research team at Jet Propulsion Laboratory for the next months. *Dare mighty things*

Ringraziamenti

Prima di tutto, vorrei ringraziare il mio relatore, professore Mauro Massari, per l'opportunità datami con questa tesi e per tutto l'aiuto datomi in questi mesi. Questo lavoro mi ha aperto gli occhi su numerosi argomenti di ricerca che mai avrei approfondito da solo e che mi hanno affascinato.

Vorrei ringraziare la mia famiglia: mia sorella, mia madre e mio padre, per tutto il supporto datomi in questi anni. Un pensiero speciale va ai miei nonni, Anna e Benito, che da sempre sono stati il mio faro nel mondo della conoscenza.
Un grande grazie va ai miei compagni di università: Filippo, Lorenzo, Davide, Alessandro e Marcello. Sono sempre stati presenti quando avevo bisogno in università e ho un sacco di bei ricordi con loro.
Un incredibile grazie va ai miei amici da sempre: Tex, Diego, Rave, Nando e Jonny, il vostro aiuto è stato incommensurabile. Un grazie speciale a Rudolf, il mio mastro del Kalman Filter. Non posso non ringraziare Anita, per le mille ore spese ad ascoltare le mie lamentele, e un grazie va anche ad Elisa, la mia surf buddy preferita. Grazie a tutti gli amici che mi hanno aiutato in questi anni. Un grazie infinito a Niccolò, Giulia e Berse: le mie guide nel mondo degli adulti. Grazie mille anche a Ksenia, Giorgio, Toma, Gabriel e Giorgio, le migliori paperelle del mondo.

Non posso quantificare quanti grazie devo ad una persona speciale che ho la fortuna di avere nella mia vita: per ogni alto e basso di questa laurea magistrale, da quel pomeriggio durante la pandemia e fin da prima, grazie Gloria per stare al mio fianco ogni giorno.

Il lavoro di questa tesi rappresenta la conclusione di un lungo percorso. Tuttavia, mi piace vederlo come un punto di partenza per qualcosa di più grande e come l'inizio del mio percorso di conoscenza. Voglio così ringraziare i professori Marco Quadrelli e Kristopher Wehage per avermi accettato come collega per i prossimi mesi nel loro gruppo al Jet Propulsion Laboratory.