“A first approach to a Simultaneous Localisation and Mapping (SLAM) solution implementing the Extended Kalman Filter for visual odometry data”

Juan Andrés Carvajal

August 29, 2014

Abstract

This project was a done as a final grade project to obtain the degree in communications electronics engineering from TECNUN. It was done at Vicomtech under the supervision of Leonardo de Maeztu , Marcos Nieto and Ainhoa Cortes. This project has consisted on the study and a fist approach implementation of a simultaneous localisation and mapping algorithm. The algorithm of choice was the EKF SLAM(Extended Kalman Filter Simultaneous Localisation and Mapping).To properly understand the concepts behind SLAM and its implementation various papers and project reports have been read, but the theoretical implementation and the practical implementation have been mostly based on the report by Jose-Luis Blanco,"Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM"[9]. Also for the implementation , the software C++ has been used with the help of the OpenCV library for the handling and processing of images and matrices. Matlab was also used when complicated math operations needed to be done. The implementations was made with the help of previous code provided by Vicomtech [1] . The code provided contained a successful implementation of a visual odometry problem and it included most of the image processing needed for the next steps of this project.

Chapter 1

Introduction

1.1 The SLAM Problem

When we talk about SLAM (Simultaneous Localisation and Mapping) we should not think of SLAM as only an algorithm but we should think of it as more of a "concept"[3].In SLAM we imagine a robot, or any type of mobile agent for that matter (e.g., a car), placed in an environment unknown to him. To properly solve the SLAM problem this robot should be able to incrementally build a map of the environment it is in (mapping) and at the same time it should locate himself within this map(localisation)[2]with the help of sensors incorporated to it. We could say we call it more a concept than just an algorithm because it is a specific situation in which we have a problem and it can be solved in many different ways, and depending on the environment the best solution possible could vary.

The birth of SLAM could be tracked to the 1986 IEEE Robotics and Automation Conference held in San Francisco, California. There many researchers including Peter Cheeseman, Jim Crowley, and Hugh Durrant- Whyte recognised that consistent probabilistic mapping was a fundamental problem in robotics [2]. Over the next years a lot of research was done in this subject and many interesting findings and papers were published. Among the most important ones were the ones by Smith et al which showed that as a robot moves through the environment taking observations of landmarks relative to the robot in the map, these estimated landmarks will all be correlated with each other because of the common error in the estimated vehicle position[4][5]. This implied that to solve the full SLAM problem we needed a state vector formed of both the position of the robot and all of the observations, and both of these would have to be updated every time we had a new observation. This state vector could be huge and the proposed solution could be very complex and costly, computationally speaking. Because of this and because of the believe that the estimated map errors would not converge and that the estimated error would grow in an unbounded manner researchers tried to minimise or eliminate the correlation between landmarks,

and research focused on solving the problem treating mapping and localisation separately [2].

The key to SLAM as we know it today was when it was realised that the problem was actually convergent and the correlations between landmarks were vital to the solution of the problem, and the bigger the correlation the better the solution would be[2]. At the International Symposium on Robotics Research (ISRR'99) in 1999 the Kalman filter based SLAM and its achievements by Sebastian Thrun were presented and discussed[6].

1.2 Structure of the SLAM Problem

To understand the proposed solution we need to identify the agents involved in SLAM, their notations and their actions.

We, of course, have the mobile agent which moves around in the environment. We will denote the position of the robot at time k as x_k. Consequently the position of the robot at time *k-1* will be denoted x_{k-1} ,and so on. The control vector u_k takes the agent from position x_k to x_{k+1}. We will call this transition the *motion model*.

As the agent moves it will discover interesting features through its sensors. The information of the position of the feature the sensors gives us which is relative to the position at time k of the agent will be called observation. We will denote the different observations as $z_{i,k}$. Meaning that it is the observation of feature i at time k.When we have this two options can follow:

The first option is that the feature i has never been seen before.When this happens we will obtain the position of this feature relative to the point (0,0) of our coordinate system(absolute position) using a model called *inverse observation model*. When we have it described as an absolute position we will call it a landmark, which we will denote as m_i.Landmarks are the ones which will describe our environment or as we called it earlier, our map. The location of these landmarks and the position of the robot mentioned previously will all have a level of uncertainty because of errors in the sensors and noise. This leads us to our second option.

The other option is that the observed feature has already been seen before, and that means it is already a landmark in our map. We use this to correct both the position of the moving agent and the position of the landmarks. The way of doing this is by trying to predict the value of the observation using the position of the corresponding landmark and the position of the robot. This process of predicting the value will be the *direct observation model*.

The *motion model*,the *inverse observation model*,and the *direct observation model* and how we apply them specifically in our system will all be explained later on this report.

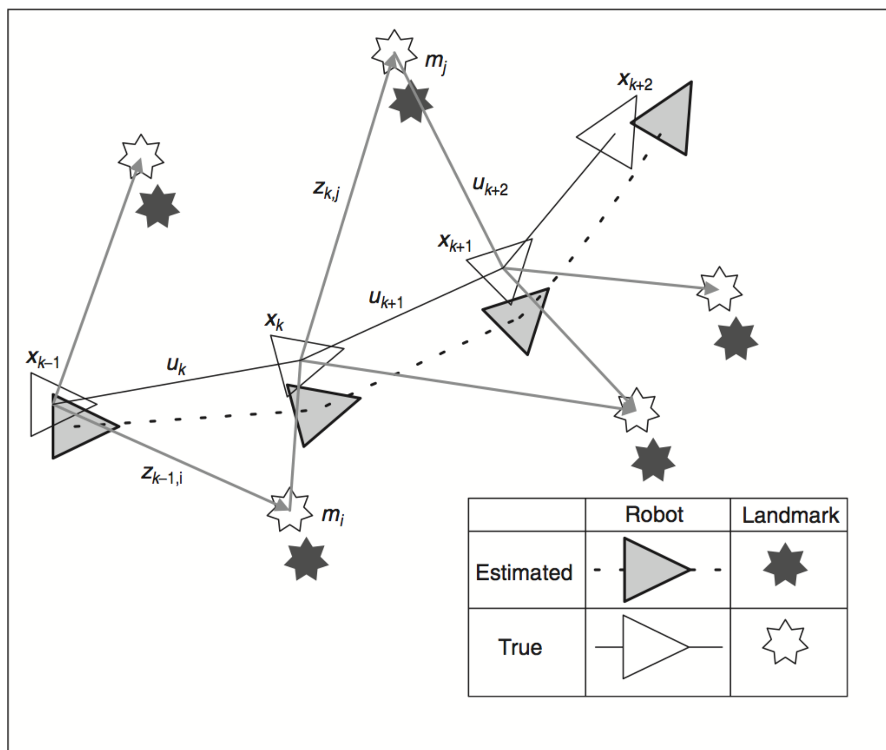

Figure 1.1: Diagram of evolution of a SLAM system [1]

Chapter 2

Objective of the Project

The objective of this project was to understand what exactly the SLAM problem was and to propose a simple solution that could serve as a first approach to solving the problem.

This solution should be able to obtain an estimate of the position of the moving agent and the features of the map at each time step by using data obtained from sensors to calculate a more accurate estimate or in other words to decrease the uncertainty in the position of the agent and the features. In our case the sensors that will be used are a stereo camera system. The data obtained from processing the images has to be all we need to apply the algorithm.

The project also included that this proposed solution could be programmed in C++ using OpenCV library ,with the additional usage of the code and video data set from Vicomtech's stereo odometry system [1] which obtained odometry data from the stereo camera system.

2.1 The System

To be able to apply a SLAM algorithm we first need to define the coordinate system in which our features and our agent will be in. There are many ways to define a coordinate system but in the end they must all represent the same movement and position . In a coordinate system there should always be a coordinate (0,0) or as we may also call it the world starting point. For us this point will be the where we obtain the first frames from our camera .

In our coordinate system the localisation or position of the robot will be 6 dimensional. The 6 dimensions will be the following :

$$x \ y \ z \ \phi \ \chi \ \psi$$

In this system x, y, and z represent the classical elements of a 3 dimensional cartesian coordinate system. Because we also care about the rotation that the agent has we introduce ϕ (the yaw), χ (the pitch), and ψ (the roll). These coordinates are easier to understand if we observe Figure 2.

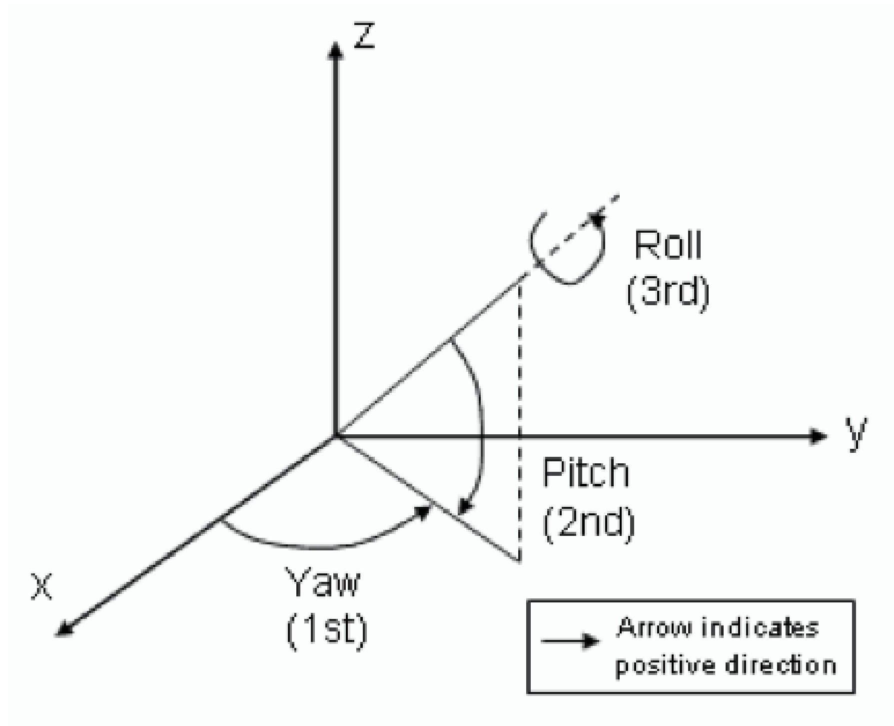

Figure 2.1: The coordinate system that will be used in this project for the position of the moving agent.

For our landmarks we will only use a 3D representation, because we do not care about their rotation, in the same coordinate system. Landmark m_i would be represented the following way:

$$x \; y \; z$$

2.2 The Camera and Visual Odometry

The camera system plays a huge role in how we will implement our algorithm. In this project we will use a stereo camera system. That means we have two cameras, or to say it in a more technical manner, two image sensors. This is useful because from this stereo vision we can get 3D information from the images and this will be key to obtain and associate data needed in our algorithm. More of this will be explained on Chapter 5.

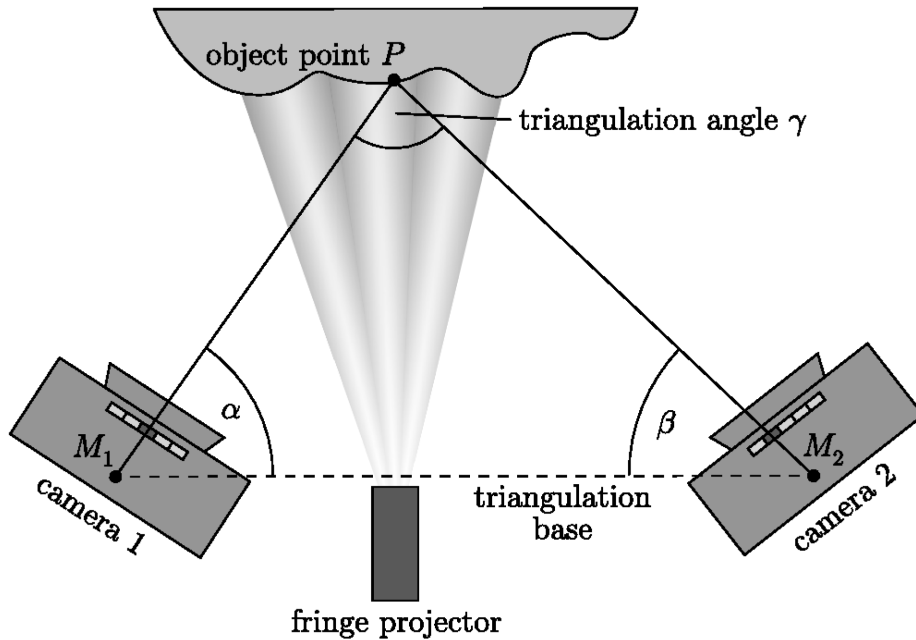

Figure 2.2: An example of a stereo camera system.

As we mentioned previously, code has been inherited from a previous implementation [1]. We will briefly try to explain what it did and how our algorithm will differ from it.

The code implemented a visual odometry algorithm. Odometry is the use of data to obtain change of position over time. In the code the data was obtained by the cameras. What the code did was from a couple of images taken at the same time it matched them with OpenCV functions and then triangulated the matched points. At the next time step we would have obtained a couple of new images. At this new time step we would use a new matching algorithm that would make a "circular" match. That means that the only points considered good matches would complete a cycle between the old images and the new images. With this we obtain 2D matches. Before we had triangulated some points from the first couple of images. Because we know which points from 2D of the original images were the ones triangulated we can observe if any of these triangulated points have matches in the new set of 2D images. If they do, then what we have are 3D-2D correspondences. This data will be the input to a function of OpenCV which obtains a pose or movement of the camera from the original set of images to the new set of images.The algorithm does this every time step. We have obtained the odometry only from the data from the cameras.

How is this different from our algorithm? The visual odometry will only calculate the odometry or pose from a time step to the next one. It does not take into account information from previous time steps. That is a key to any SLAM algorithm. Also, a very important and key difference is that the visual odometry

believes that the calculated pose is correct. It does not take into account any possible source of noise or uncertainty. In SLAM, this is a fundamental step. Every camera system and every computational process will have sources of noise leading to uncertainty in the results. In our implementation of EKF-SLAM we use the odometry data not only to estimate the position of the agent but also to correct the uncertainty of the agent and of the features position. More of this will be explained in Chapter 5.

Because this is only a first approach, the implemented algorithm was intended to be simple and not to computationally costly , and that is why our implementation does not take into account all of the time steps for the steps that will later be explained , but it will be able to keep track of the position of the agent through the whole movement in the map, or to say in in other words it will keep an absolute position of the robot with respect to the starting point.

2.2.1 Applications of the EKF-SLAM Algorithm

It is easy to see that a successful implementation of the algorithm would do great things in the world of navigation. If robots, cars , etc. would be able to locate themselves in a map (which would also be known) with a small level of uncertainty these things could become autonomous and they could navigate themselves.

In present day SLAM is used for both autonomous vehicles but also for assistance to the driver. Practical applications can be found everywhere. From vacuum cleaners that navigate the house on their own to exploration of underground mines by autonomous robots.Perhaps the most popular application in the present day is the self driving cars. Self driving cars find their basic principles in SLAM, although they are much more complicated that what we do in this project because they use several more sensor and they sometimes include other type of navigation assistance, for example GPS.

One of the most successful and famous implementations of SLAM is the Stanley vehicle created by Stanford University's Stanford Racing Team in cooperation with the Volkswagen Electronics Research Laboratory (ERL). It competed and won the 2005 DARPA challenge, which was a competition funded by the Defense Advanced Research Projects Agency for autonomous cars in the desert.

The interest in SLAM is greater each day, and applications, both military and civilian, will surely be a big factor in the near future.

Chapter 3

Different Approaches to SLAM

In this section we will discuss the most popular and successful approaches of SLAM over the years.

3.1 The Extended Kalman Filter SLAM

The EKF SLAM(Extended Kalman Filter SLAM) has been the classical approach to solve the SLAM problem. Its basis is in the Kalman filter but it has a slight difference. To explain it we first need to know what the Kalman filter is.

3.1.1 The Kalman Filter

The Kalman Filter is a technique invented by Rudolph Emil Kalman with the purpose of filtering and predicting in linear systems. It can only be applied in linear. This filter represents **beliefs** at time k by a **mean** and a **covariance** of a state.

The basic idea behind the filter is that we introduce into the filter some noisy data and our algorithm will give us a less noise or more exact data. Its applications can be found in many areas, from economics to computer vision and tracking applications.

We will now show and explain the Kalman filter algorithm.First we are going to explain the notation we are going to use briefly. For now the notation of the mean or estimate of our state vector will be $\hat{\mathbf{x}}$ (not to be confused with the position of the robot x_k mentioned previously) and the covariance will be expressed as $\mathbf{P}$. The notation $\hat{\mathbf{x}}_{n|m}$ will mean that it is the estimate at time n with observations up to, and including time m. It is analog with $\mathbf{P}_{n|m}$ The algorithm can be described as follows:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k \mathbf{u}_k \tag{3.1}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^{\mathrm{T}} + \mathbf{Q}_k \tag{3.2}$$

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1} \tag{3.3}$$

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} + \mathbf{R}_k \tag{3.4}$$

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} \mathbf{S}_k^{-1} \tag{3.5}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k \tag{3.6}$$

$$\mathbf{P}_{k|k} = (I - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1} \tag{3.7}$$

To compute the Kalman Filter algorithm we will need as inputs the estimates and covariances at time *k-1*,the control vector u_k and the measurements or observations z_k. The output of the Kalman filter is the estimate and covariance at time k.

We have (3.1) and (3.2) as the Predict Stage and (3.3) to (3.7) as the Update or Correction stage. In (3.1) and (3.2) we calculate the predicted mean and covariance at time k but without taking into account the observations z_k. The matrix $\mathbf{F}_k$ and $\mathbf{B}_k$ are linear and relate current states with the previous states through the control vector. Or in other words they are linear matrices to used for the prediction of the next state. It is multiplied two times to the covariance because the covariance is a quadratic matrix.

In the next steps we will incorporate the observations to compute the output of the filter. Briefly what we do is calculate in (3.5) what we call the Kalman Gain, which is the degree to which the observation or measurement is incorporated into a new state estimate. In other words, the Kalman Gain will tell us how much we trust the measurement to change the estimate. To calculate this Kalman Gain first we must have calculated in (3.4) what we call the Kalman innovation. This tells us how much the measurements vary and that will lead us to how much confidence we have in our measurements. In (3.3) we find the deviation between the actual measurement z_k and the measurement predicted by using the matrix $\mathbf{H}_k$ and $\hat{\mathbf{x}}_{k|k-1}$. The matrix $\mathbf{H}_k$ is linear and analog to the measurement to what the $\mathbf{F}_k$ and $\mathbf{B}_k$ matrices are to the state transition. That is, it is used to predict the measurement.

Once we have all these elements we can proceed to calculate the new estimated mean $\hat{\mathbf{x}}_{k|k}$ in (3.6) by adjusting it in proportion to the Kalman gain and the deviation mentioned previously. In (3.7) we calculate the new covariance adjusting for the information gain resulting from the measurement [7].

3.1.2 The Extended Kalman Filter

Applying the Kalman filter in real life problems is very complicated due to the fact that linear state transitions and linear measurements with added Gaussian noise are not always found in real life problems[7]. Lots of this problems have a non-linear behaviour. So to be able to apply the Kalman filter to these problems we need to do something differently.

The solution to this problem is the extended Kalman filter. What we do here is that instead of the algorithm being governed by linear matrices we introduce non-linear functions. We will call these functions f and h. The prediction of the next state and of the measurement will be made by the two nonlinear functions. As a consequence of this we will introduce the Jacobians F_k, which will replace $\mathbf{F}_k$ and $\mathbf{B}_k$, and H_k which will replace $\mathbf{H}_k$. Even though we use the same letters in a couple of matrices we should not confuse them.

The EKF, as it is called, is indeed a solution. But that does not mean that it does not have its problems. By using the nonlinear function the real belief of the state is no longer a Gaussian, but the EKF approximates it as a Gaussian, with a mean and a covariance. That means the belief is no longer exact, as it was in the Kalman Filter, but it is only an approximation.

In the next subsection we will see the EKF algorithm and the differences between both algorithms will be easier to see and understand.

3.1.3 The EKF Algorithm

The EKF Algorithm is the following:

$$\hat{\boldsymbol{x}}_{k|k-1} = f(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k) \tag{3.8}$$

$$\boldsymbol{P}_{k|k-1} = F_k \boldsymbol{P}_{k-1|k-1} F_k^\top + \boldsymbol{Q}_k \tag{3.9}$$

$$\tilde{\boldsymbol{y}}_k = \boldsymbol{z}_k - h(\hat{\boldsymbol{x}}_{k|k-1}) \tag{3.10}$$

$$\boldsymbol{S}_k = H_k \boldsymbol{P}_{k|k-1} H_k^\top + \boldsymbol{R}_k \tag{3.11}$$

$$\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1} H_k^\top \boldsymbol{S}_k^{-1} \tag{3.12}$$

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \tilde{\boldsymbol{y}}_k \tag{3.13}$$

$$\boldsymbol{P}_{k|k} = (\boldsymbol{I} - \boldsymbol{K}_k H_k) \boldsymbol{P}_{k|k-1} \tag{3.14}$$

We can see how in (3.8) and (3.10) the next state and measurement prediction has been replaced with the nonlinear functions, and the inputs those functions need. Also we can see the Jacobians F_k and H_k have replaced the linear matrices in the other steps of the algorithm where needed.

Below, the mathematical expression of the Jacobians can be found.

$$F_k = \left.\frac{\partial f}{\partial \hat{\boldsymbol{x}}}\right|_{\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_{k-1}} \quad H_k = \left.\frac{\partial h}{\partial \hat{\boldsymbol{x}}}\right|_{\hat{\boldsymbol{x}}_{k-1|k-1}} \tag{3.15}$$

3.2 Fast-SLAM

In this section we will only introduce another popular approach to SLAM to prove that there can be different solutions to the same problem. FastSLAM will not be described as thoroughly as EKF SLAM because it is not the solution used in this project.

FastSLAM can be practical to use when we have an environment with a big set of features(in the order of millions). Also when there are many random patterns, shapes, and textures[8].

In FastSLAM we can say that we handle each feature separately because of conditional independence according to a Bayesian Network. This will make the probability distribution to focus on the trajectory of the robot instead of just a single pose of the robot at a certain time. This is the reason for its speed , as it makes the map representation a set of independent Gaussians instead of a joint map covariance like we did in the EFK SLAM. FastSLAM is also called Rao-Blackwellized Filter SLAM because it has the structure of a Rao-Blackwellized state. For the pose states recursive estimation is performed with the technique of particle filtering and we use EKF for the map states we use the EKF by processing each landmark individually as an EKF measurement assuming we have a known pose[2].

To properly and thoroughly describe FastSLAM we would need to make a whole new project. But what is described previously shows that different techniques can be used to solve the problem. This is mentioned in this project to emphasise the notion that SLAM is more of a concept than just an algorithm, and how it can be solved in different ways and depending on the situation a solution could be better than other.

Chapter 4

Methodology

In this Chapter we will briefly explain some of the tools we used to implement our algorithm. Basically the most important tools used in our project were C++, Matlab and the library for C++ OpenCV.

We will not explain the C++ tools used the code used for C++ was very basic mostly because if we wanted to use a special function, class or feature we used them from the OpenCV library.

4.1 OpenCV Library

This library is a great tool to use if images are involved. Also, because images can be represented as matrices the library is very useful when handling matrices.

The most important functions and classes used in the code were:

GridAdaptedFeatureDetector: According to the OpenCv website this class adapts a detector to partition the source image into a grid and detect points in each cell. With an object of this class and the detect function of this class we can detect points in an image.

BriefDescriptorExtractor: An object of this class will compute BRIEF descriptors. Descriptors are like the characteristics of each point detected in the image.With the compute function of this class the descriptors of each point will be created.

class DescriptorMatcher: Class for matching descriptors. With the match function we can match the descriptors to see with point has a correspondence in other image.

triangulatePoints: This function reconstructs 3D points by triangulation of correspondences.

Point3d: This class allows us to create variables of this class and for them to be 3 dimensional. Another example would be Point3f, which would create a 3D point of floats.

class Mat: This is probably our most used class. All of our matrices are objects of the Mat class. This class allows us to directly do operations on

matrices and one of the most interesting and used functions is the rect() function which lets us copy or insert just a ”rectangle” or a portion of the matrix. The importance of this function will be seen later on this project when the operations for the implementation appear.

These are the most important and most used functions and classes we have used in our implementation and that make the handling of images possible and easier in C++.

4.2 Matlab

Later on this project we will find that we need to compute some Jacobians. The Jacobians are basically derivatives and with the type of matrices that we have computing this by hand would be impossible. And a feasible implementation of derivatives in C++ or OpenCV was not an option. That is why we used the Matlab symbolic toolbox. In this toolbox we can define some variables and all of the results from the operations done regarding that variable will be in function of the variable. This was helpful because we could just insert the general expression in matlab and the result would always be in function of the variables we wanted. The usage of this will be easily seen when the operations for the Jacobians are reached and also the Matlab code can be found in Appendix A.

Chapter 5

Implementation of the EKF SLAM

5.1 System Specific Matrices of EKF SLAM

In this section and the ones that follow we will describe the implementation of the EKF SLAM applied to our specific system .We will have a very similar system and theoretical implementation as the one used by Jose-Luis Blanco in his report "Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM" [9].

We already defined in Chapter 2 what the coordinates for our system would be. We now represent the position of the agent at time k with the previously defined coordinates.

$$x_k = [x\ y\ z\ \phi\ \chi\ \psi] \tag{5.1}$$

We do the same for our landmarks , representing landmark m_i the following way:

$$m_i = [x\ y\ z] \tag{5.2}$$

5.1.1 Mean State Vector

Now that we have defined the coordinate system of both the moving agent and the landmarks we can define our state vector which in our probabilistic approach will be considered as the mean of the belief. It is defined as the concatenation of the moving agent pose or location and all of the known landmarks at this time step.

$$\mathbf{x} = [x\ m_1\ m_2\ m_3 ...\ m_L]^\top \tag{5.3}$$

We are using the same notation as before with $\mathbf{x}$ being the mean of the belief, x the agent pose, and m_i as the position of the landmarks.

When implementing the EKF SLAM in C++ we will initialise the robot pose to all zeros, as we belief this is the starting point.

As for the landmarks, we will briefly explain how we obtain them using the code provided by Vicomtech [1] and some code added by myself to adjust it into the EKF algorithm.

When we have our first camera pair, that is at time $k = 0$ we detect features in our image using both cameras and using a procedure(which explanation is out of the boundaries of this project) we try to keep only the features that we believe have a correspondence in the other image, in other words we match the features in one image to the other image as best as we can. There will always be some outliers and these will contribute to the noise in our system. Those points will be considered our first set of landmarks. To properly introduce them into our mean state vector we need to know their absolute coordinates referred to the world starting point.

As mentioned previously, we have decided that we are currently at the world starting point. Because of this and because of our stereo camera system we only need to triangulate the matching points in our images and once we convert the distance obtained from the triangulation to world coordinates instead of pixel coordinates we already have the distance from the world starting point to all of the landmarks. They are ready to be introduced to our mean state vector.

We use the functions provided by OpenCV and explained in Chapter 4 to match and triangulate these points.

5.1.2 Covariance Matrix

The other element of the state belief is the covariance matrix. The covariance will have a structure with sub-matrices as the one seen below

$$\begin{bmatrix} P_{xx} & P_{xm1} & P_{xm2} & ... & P_{xmL} \\ P_{m1x} & P_{m1m1} & P_{m1m2} & ... & P_{m1mL} \\ ... & ... & ... & ... & ... \\ P_{mLx} & P_{mLm1} & P_{mLm2} & ... & P_{mLmL} \end{bmatrix} \tag{5.4}$$

In our system the first element of the matrix P_{xx} is a 6×6 matrix and it contains the variances of the robot pose. The first column ,except P_{xx}, P_{mix} contains the covariances between the landmark i and the robot pose and will be of dimensions 3×6 . The first row excluding P_{xx} is the transpose of the column just explained and its dimension will obviously be 6×3 . The other elements P_{mimj} are the covariances between the landmarks with dimension 3×3. That will give a dimension to our covariance matrix of $6 + 3L \times 6 + 3L$.

We will initialise the covariance matrix to zeros, because this uncertainty will determine the best the robot can localise itself from now on as stated in [9] and [10].

We have the mean state vector and the covariance defined for time $k = 0$. In the next sections we will see what happens when time passes and the robot moves.

5.2 First Step: Prediction

In this section we will discuss the specific elements and the implementation of first two steps of the EKF algorithm which are stated in (3.8) and (3.9).

5.2.1 The Motion Model

The only variables that are affected by time and therefore movement are those belonging to the pose of the agent. The landmarks we consider static with time.

Lets remember the first step of the EKF filter :

$$\hat{\boldsymbol{x}}_{k|k-1} = f(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k)$$

with f being a nonlinear function. This function is they key to predict $\hat{\boldsymbol{x}}_{k|k-1}$.In our system . It's important to realise that we need the inputs $\hat{\boldsymbol{x}}_{k-1|k-1}$ and $\boldsymbol{u}_k$ being the previous mean state vector and the control vector. With this inputs we will construct the motion model to make the prediction of the mean.

The control vector u_k can be expressed as

$$u = \{x_u \ y_u \ z_u \ \ \phi_u \ \chi_u \ \psi_u\} \tag{5.5}$$

We know from (5.1) how our state vector mean is expressed but because of rotations in movements we can't just add $x + x_u$ and so on to obtain the motion model. What we need to do is use the pose composition operator $\oplus$ to obtain the new pose. The composition operator is equivalent to multiplying their equivalent homogeneous matrices.

$$f(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k) = \boldsymbol{x}_{k-1|k-1} \oplus \boldsymbol{u}_k \tag{5.6}$$

To do this we need to be able to express the control and the pose in homogeneous matrix form. Both of this matrices can be obtained in homogeneous form from their elements x_u , y_u , z_u , ϕ_u , χ_u and ψ_u using the following formula

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & x \\ R_{21} & R_{22} & R_{23} & y \\ R_{31} & R_{32} & R_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} cos\phi cos\chi & cos\phi sin\chi sin\psi - sin\phi cos\psi & cos\phi sin\chi cos\psi + sin\phi sin\psi & x \\ sin\phi cos\chi & sin\phi sin\chi sin\psi + cos\phi cos\psi & sin\phi sin\chi cos\psi - cos\phi sin\psi & y \\ -sin\chi & cos\chi sin\psi & cos\chi cos\psi & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

We can consider the homogeneous matrix can be divided into c 3×3 rotation sub matrix and a 3×1 translation sub matrix not taking into account the last row.

The result of the composition operator would be another homogeneous matrix. We are interested in obtaining the the elements x , y , z , ϕ , χ *and* ψ of the resulting homogenous matrix. If we write individually for each element what the composition operator yielded we obtain the following.

$$x_k = x_{k-1} + R_{11}x_u + R_{12}y_u + R_{13}z_u \tag{5.8}$$
$$y_k = y_{k-1} + R_{21}x_u + R_{22}y_u + R_{23}z_u \tag{5.9}$$
$$z_k = z_{k-1} + R_{31}x_u + R_{32}y_u + R_{33}z_u \tag{5.10}$$
$$\phi_k = \phi_{k-1} + \phi_u \tag{5.11}$$
$$\chi_k = \chi_{k-1} + \chi_u \tag{5.12}$$
$$\psi_k = \psi_{k-1} + \psi_u \tag{5.13}$$
$$\tag{5.14}$$

Note that for the yaw, pitch and roll we have not really put the operations made by the composition operator, but we just add the previous angle with the angle added by the control. We do this because in the case of the angles it is possible to add them and we can obtain them directly from the rotation matrix of the resulting homogeneous matrix (from the composition operator).

In the implementation in C++ instead of obtaining the correspondent yaw, pitch, and roll two times, one for the control matrix and one for the previous pose matrix, we can directly obtain the values of the yaw, pitch, and roll from the resulting matrix of the composition operator to be more efficient. We can do it using the following algorithm from [11].

$$yaw = atan2(R_{32}, R_{33}) \tag{5.15}$$
$$c2 = \sqrt{R_{32}^2 + R_{33}^2} \tag{5.16}$$
$$pitch = atan2(-R_{31}, c2) \tag{5.17}$$
$$roll = atan2(R_{21}, R_{11}) \tag{5.18}$$

What we have in (5.8) to (5.14) is also our motion model which shows us how the our mean pose will change with time. This seems to work fine, but to apply it we need to obtain the inputs correctly.

5.2.2 Obtaining control from visual odometry

As we saw in the previous section once we have the inputs it is possible to build a motion model. In this subsection it will be briefly and shallowly explained how we obtain the controls from Vicomtech's stereo odometry system [1]. Most of this has already been explained in subsection 2.2.1 but here we will try to explain it again and explaining in a deeper way the "circular" match.

We have explained earlier that at time $k = 0$ we have detected features using both cameras and we have found matches between the two sets of features.To make this process more accurate when we match a pair of images we do it "both ways". That means we match the image from the right camera with the image from the left camera and we also match the image from the left with the image from the right. We only keep as matches the features that are considered matches in both of the steps mentioned. We are again using the functions explained in Chapter 4. At time $k = 1$ we do the same process with the new pair of cameras but having saved the information of the images at time $k = 0$. Saving the information of time k=0 allows us to make a new match (using the same steps as before) between the image obtained by the right camera at time $k = 0$ with the image obtained by the right camera at time $k = 1$. We then do the match again with the left camera at time $k = 0$ with the left camera at time $k = 1$. We then have the information of the matches between the "old" cameras , the "new cameras ", between the left "old" and "new" cameras ,and between the "old" and "new" right cameras. If we imagine this we can see it is a cycle .We will only consider matches between times $k = 0$ and $k = 1$ the features that " go around" ,or that "complete" this cycle. This process will give us a smaller number of outliers than just matching one of the images with the image at the next time frame. In Figure 5.1 we can see that at each time step several features can be detected, represented in the image by all of the dots. The red dots represent features that have been detected, but only the green dot complies with the circular matching explained before (represented by the arrows), and therefore the green dot is the only one we will consider a match.

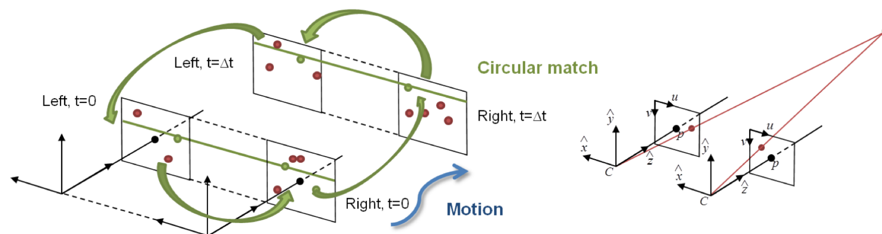

Figure 5.1: Only the green dot representing a feature will comply with the circular match [1].

Just like we said in subsection 2.2.1 we can associate the matches in 2D to the points in 3D that we triangulated before at time $k = 0$, and including the previous position of the moving agent (at time $k = 0$ the position would be (0,0,0,0,0,0) as we mentioned earlier), we can introduce this into the Perspective-N-Point algorithm utilising the RANSAC scheme which will give us the "pose"

of the agent from 3D-2D point correspondences. We can implement it directly using the the PnPRansac function from OpenCV library for C++. In reality the "pose" that the function gives us is the "pose" from $k = 0$ to $k = 1$, which is what we call the control.

Next we triangulate the matches from the new left and right images. In the next time frame these new images and all of its information will become the old images and repeats itself iteratively. This is how we obtain the control at every time step from k to $k + 1$.

5.2.3 Predicted State Vector $\hat{\mathbf{x}}_{k|k-1}$

We can directly apply equations (5.8) to (5.14) using the control obtained from the odometry process explained before and the previous location of the moving agent to obtain

$$\hat{\mathbf{x}}_{k|k-1}$$

, which is the predicted mean state vector without having taken into account the observations. It may seem we have taken into account the observations because the process to obtain the control uses the matches. But this process uses the matches as an input to the RANSAC algorithm which selects only a number of all the matches, and this process has its own source of noise when estimating the pose.

As we said before only the position of the agent is affected by time, as landmarks are considered static. That is why we only observe the position of the robot being affected in the motion model.

5.2.4 Predicted Covariance Matrix $\mathbf{P}_{k|k-1}$

To complete the prediction stage the next step is to predict the covariance matrix not taking into account the observations yet. For this we need the covariance matrix at the previous time step $\boldsymbol{P}_{k-1|k-1}$,the Jacobian F_k , and a matrix which is the uncertainty associated to the movement of the agent, $\boldsymbol{Q}_k$.

Jacobian F_k

The mathematical expression to obtain the Jacobian F_k was the following :

$$F_k = \left.\frac{\partial f}{\partial \hat{\boldsymbol{x}}}\right|_{\hat{\boldsymbol{x}}_{k|k-1}, \boldsymbol{u}_{k-1}}.$$

We know the operations function f makes ,so if we apply the definition of Jacobian, that is the matrix of all first-order partial derivatives of a vector-valued function we obtain the following:

$$F_k = \begin{bmatrix} \left.\frac{\partial f}{\partial x}\right|_{\boldsymbol{6x6}} & 0|_{\boldsymbol{6x3}} & 0|_{\boldsymbol{6x3}} & \cdots & 0|_{\boldsymbol{6x3}} \\ 0|_{\boldsymbol{3x6}} & I|_{\boldsymbol{3x3}} & 0|_{\boldsymbol{3x3}} & \cdots & 0|_{\boldsymbol{3x3}} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0|_{\boldsymbol{3x6}} & 0|_{\boldsymbol{3x3}} & 0|_{\boldsymbol{3x3}} & \cdots & I|_{\boldsymbol{3x3}} \end{bmatrix} \tag{5.19}$$

With $\frac{\partial f}{\partial x}$ being a Jacobian sub matrix according only to the robot position.

In some way the matrix we have obtained was kind of intuitive to see because the function f referring to the motion model only affected the parameters of the localisation of the robot because the landmarks are static.

The result of $\frac{\partial f}{\partial x}$ will be the following

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 1 & 0 & 0 & F14 & F15 & F16 \\ 0 & 1 & 0 & F24 & F25 & F26 \\ 0 & 0 & 1 & 0 & F35 & F36 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.20}$$

where:

$$F14 = -sin\phi_{k-1} * cos\chi_{k-1} * x_u + (-sin\phi_{k-1} * sin\chi_{k-1} * sin\psi_{k-1} - cos\phi_{k-1} * cos\psi_{k-1}) * y_u$$
$$+ (-sin\phi_{k-1} * sin\chi_{k-1} * cos\psi_{k-1} + cos\phi_{k-1} * sin\psi_{k-1}) * z_u$$
$$F15 = -cos\phi_{k-1} * sin\chi_{k-1} * x_u + cos\phi_{k-1} * cos\chi_{k-1} * sin\psi_{k-1} * y_u + cos\phi_{k-1} * cos\chi_{k-1} * cos\psi_{k-1} * z_u$$
$$F16 = (cos\phi_{k-1} * sin\chi_{k-1} * cos\psi_{k-1} + sin\phi_{k-1} * sin\psi_{k-1}) * y_u + (-cos\phi_{k-1} * sin\chi_{k-1} * sin\psi_{k-1}$$
$$+ sin\phi_{k-1} * cos\psi_{k-1}) * z_u$$
$$F24 = cos\phi_{k-1} * cos\chi_{k-1} * x_u + (cos\phi_{k-1} * sin\chi_{k-1} * sin\psi_{k-1} - sin\phi_{k-1} * cos\psi_{k-1}) * y_u$$
$$+ (cos\phi_{k-1} * sin\chi_{k-1} * cos\psi_{k-1} + sin\phi_{k-1} * sin\psi_{k-1}) * z_u$$
$$F25 = -sin\phi_{k-1} * sin\chi_{k-1} * x_u + sin\phi_{k-1} * cos\chi_{k-1} * sin\psi_{k-1} * y_u + sin\phi_{k-1} * cos\chi_{k-1} * cos\psi_{k-1} * z_u$$
$$F26 = (sin\phi_{k-1} * sin\chi_{k-1} * cos\psi_{k-1} - cos\phi_{k-1} * sin\psi_{k-1}) * y_u + (-sin\phi_{k-1} * sin\chi_{k-1} * sin\psi_{k-1}$$
$$- cos\phi_{k-1} * cos\psi_{k-1}) * z_u$$
$$F35 = -cos\chi_{k-1} * x_u - sin\chi_{k-1} * sin\psi_{k-1} * y_u - sin\chi_{k-1} * cos\psi_{k-1} * z_u$$
$$F36 = cos\chi_{k-1} * cos\psi_{k-1} * y_u - cos\chi_{k-1} * sin\psi_{k-1} * z_u$$

Uncertainty in the movement $\boldsymbol{Q}_k$

The $\boldsymbol{Q}_k$ matrix will be the uncertainty in the new state of the agent because of the noise in the control and it will be associated to a Jacobian with respect to the control vector u.

$$\frac{\partial f}{\partial u}$$

This Jacobian will also be zero for the landmarks so we only need to compute the Jacobian for the position, which we will take from [9].

$$\frac{\partial f}{\partial u} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 & 0 & 0 \\ R_{21} & R_{22} & R_{23} & 0 & 0 & 0 \\ R_{31} & R_{32} & R_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.21}$$

With all the Rxx elements defined in equation (5.7). Then $\boldsymbol{Q}_k$ will be

$$\boldsymbol{Q}_k = \frac{\partial f}{\partial u} * U * \frac{\partial f}{\partial u}^T \tag{5.22}$$

With U being a matrix with of established uncertainty from the beginning .

Resulting $\mathbf{P}_{k|k-1}$

Applying

$$\boldsymbol{P}_{k|k-1} = F_k \boldsymbol{P}_{k-1|k-1} F_k^\top + \boldsymbol{Q}_k$$

we obtain the following matrix:

$$\begin{bmatrix} \frac{\partial f}{\partial x} P_{xx} \frac{\partial f}{\partial x}^T & \frac{\partial f}{\partial x} P_{xm1} & \frac{\partial f}{\partial x} P_{xm2} & ... & \frac{\partial f}{\partial x} P_{xmL} \\ P_{m1x} \frac{\partial f}{\partial x}^T & P_{m1m1} & P_{m1m2} & ... & P_{m1mL} \\ ... & ... & ... & ... & ... \\ P_{mLx} \frac{\partial f}{\partial x}^T & P_{mLm1} & P_{mLm2} & ... & P_{mLmL} \end{bmatrix} \tag{5.23}$$

As we can see only the first row and column of sub matrices have changed. This has logic because as we have mentioned several times before, landmarks are static.

5.3 Second Step : Correction or Update

In this step the moving agent will move and it will observe features that have been mapped before, meaning that they are in our state vector already. Incorporating theses re-observed features will allow us to make the correction .

5.3.1 The Observation Model

In the observation model $h(\hat{\boldsymbol{x}}_{k|k-1})$ we will try to predict the location of a landmark relative to the pose of the moving agent. We will try to predict using $\hat{\boldsymbol{x}}_{k|k-1}$, which gives us the predicted position of the moving agent at time k and the absolute coordinates of the landmarks. In other words, what we will do is for each landmark m_i , knowing where our agent supposedly is, we will calculate the position of this landmark relative to where the agent is (the distance from the agent to that landmark). We will denotate the predicted relative position of the landmark as $\hat{m_i} = [\hat{x_i}\ \hat{y_i}\ \hat{z_i}]$ This is important because later we will compare this prediction to the actual observation we make through the sensors.

We will again use homogeneous coordinates. In this case we have an absolute landmark m_i, which will be expressed in the following way:

$$\begin{bmatrix} 0 & 0 & 0 & x_i \\ 0 & 0 & 0 & y_i \\ 0 & 0 & 0 & z_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.24}$$

And as before we have the position of the agent in a homogeneous matrix the same as the first element of the equation in (5.7). Once we have this we do the following operation:

$$\begin{bmatrix} & & & \hat{x}_i \\ & & & \hat{y}_i \\ & & & \hat{z}_i \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & x \\ R_{21} & R_{22} & R_{23} & y \\ R_{31} & R_{32} & R_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix}^{(-1)} \begin{bmatrix} 0 & 0 & 0 & x_i \\ 0 & 0 & 0 & y_i \\ 0 & 0 & 0 & z_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.25}$$

We are multiplying the inverse of the location of the agent with the absolute landmark. Notice how we do not take into account the rotation sub matrix of the resulting matrix. This is so because we consider landmarks as only points, with no rotation.

5.3.2 The Observation Noise

Just as we had noise in the previous step we also have a variable that measures or represents noise in the observation model. This will be determined by the variable R.This only depends on the sensor, and it is represented by the matrix

$$R = \begin{bmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_y \end{bmatrix} \tag{5.26}$$

Each value represent the variance for each element of the observation. They are set at the beginning of the algorithm. It is a diagonal matrix which tells us that the variance in the elements of the measurement are independent from each other.

5.3.3 Determining re-observed landmarks from visual odometry

Everything explained before is possible in our implementation but first we need to know which landmarks observed at time step *k+1* are mapped. In Section 5.2.2 we talked about how we obtained the control from visual odometry. Obtaining the re-observed landmarks from the same process is relatively easy, we just need to keep a little bit of bookkeeping.

Let's remember that we have matched features from the process just mentioned. To know which landmark has been re-observed we just need to create a mask vector that will tell us, of all the features observed which ones have a match in the previous time step.Only these features (which have been re-observed)will be used in the following steps of the correction stage. We only take two consequent time steps at a time for our implementation so we only really check for re-observed landmarks from one previous time step.

Let's give an example. Let's suppose we have a 3D feature in our map called m_1. We can obviously trace this 3D point to its original 2D point. Now we

have the 2D point that originated the 3D point which is in our map. Let's call this 2D point $m_{1_{2D}}$. As we said previously when we have a new time step we match the old images (the images that contain $m_{1_{2D}}$) with the new images. So after we have matched we have several correspondences between the old images and the new images. If $m_{1_{2D}}$ is among these matches then it means that the 3D point m_1 has been observed again and we will use it for the correction step. The mas vector that we talk about is just a logical vector(1 or 0) that tells us if a point like $m_{1_{2D}}$ has a match in the new time step. We do this with all the 3D points from the old time step.

This is extremely important because these re-observed features will allow us to apply the correction step in the Extended Kalman Filter.

5.3.4 Kalman Innovation and Gain

In (3.5) we have the formula to obtain the Kalman gain. To obtain it we need Jacobian $\mathbf{H}_k$, our predicted covariance matrix, and innovation $\mathbf{S}_k$.

Full Matrix S

If we directly apply (3.11) we would obtain a full matrix of size $6+3L \times 6+3L$. According to [9] a way to speed up the implementation is to take advantage of the special structure of $\mathbf{H}_k$ (which will be explained later) and to calculate $S_{i,j}$ as a scalar of the jth component of the ith observed landmark.

Scalar $S_{i,j}$

To do this we need to take the elements of the observation one at a time. According to [9] the resulting scalar will be the following

$$\frac{\partial h_{i,j}}{\partial x} P_{xx} \frac{\partial h_{i,j}}{\partial x}^T + 2 \frac{\partial h_{i,j}}{\partial m_i} P_{mix} \frac{\partial h_{i,j}}{\partial xi}^T + \frac{\partial h_{i,j}}{\partial m_i} P_{mimi} \frac{\partial h_{i,j}}{\partial mi}^T + R_{jj} \tag{5.27}$$

where the partial Jacobians are the j'th row of $\frac{\partial h_{i,j}}{\partial \hat{x}}$ and R_{jj} is the j'th diagonal element of R.

Partial Jacobians of $\frac{\partial h_{i,j}}{\partial \hat{x}}$

As we have just seen the Jacobian of the observation model, also called $\mathbf{H}_k$ can be partitioned into partial Jacobians to make operations easier. Because the observation of each landmark i depends on the vehicle pose and its corresponding landmark the full Jacobian will be sparsely populated.

$$\frac{\partial h}{\partial \hat{x}} = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_1}{\partial m_1} & 0 & ... \\ \frac{\partial h_2}{\partial x} & 0 & \frac{\partial h_2}{\partial m_2} & ... \\ \frac{\partial h_3}{\partial x} & 0 & 0 & ... \\ ... & ... & ... & ... \end{bmatrix} \tag{5.28}$$

This mean that in reality we only need to computer the partial Jacobians $\frac{\partial h_i}{\partial x}$ and $\frac{\partial h_i}{\partial m_i}$ for each landmark. And each row:

$$\frac{\partial h_{i,j}}{\partial \hat{x}} = \begin{bmatrix} \frac{\partial h_{i,j}}{\partial x} & ... & \frac{\partial h_{i,j}}{\partial y_i} & ... \end{bmatrix} \tag{5.29}$$

with $j = 1, 2, 3$ corresponds to each of the 3 components of the observation.

To obtain the partial Jacobians we have used the symbolic toolbox by matlab. This equations are too big and complicated to put into the project so the code to obtain them is in appendix A.

Kalman Gain Vector $K_{i,j}$

Just as we converted S into $S_{i,j}$ we can also convert the Kalman gain to $K_{i,j}$ for a more efficient algorithm. Just as [9] we obtain the following:

$$K_{i,j} = \left(\begin{bmatrix} P_{xx} \\ P_{y1x} \\ P_{y2x} \\ ... \end{bmatrix} \frac{\partial h_{i,j}}{\partial x}^T + \begin{bmatrix} P_{xyi} \\ P_{y1yi} \\ P_{y2yi} \\ ... \end{bmatrix} \frac{\partial h_{i,j}}{\partial y_i}^T \right) 1/S_{i,j} \tag{5.30}$$

5.3.5 Correcting State Vector and Covariance Matrix

We could directly apply (3.13) to correct the state vector. It is very important to notice the importance of (3.10) which we have not explicitly explained before. It is the subtraction between the actual value of the observation made through the sensors and the predicted value of the observation made through the observation model.

But because we have made some changes to be more efficient, the operation we will make is

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_{i,j}(z_{i,j} - \hat{m}_{i,j}) \tag{5.31}$$

With $(z_{i,j} - \hat{m}_{i,j})$ being the difference between the actual observation and the predicted observation.

As for the covariance matrix we will apply the following formula:

$$\boldsymbol{P}_{k|k} = \boldsymbol{P}_{k|k-1} - S_{i,j} K_{i,j} K_{i,j}^T \tag{5.32}$$

We will do this for each dimension of each observed landmark.

5.4 Third Step : Creating New Landmarks

We previously mentioned that we would only take two consequent time steps for each operation. That means that our implemented algorithm behaves as if only the images and information at time step *k* and time step *k+1* existed. At time step *k+2* everything is reset except for the agent localisation, so *k+1*, and its related information, is treated as if it were the beginning of the whole process. This was made to make it a more doable project. Because of this we really do

not add any more landmarks at each new time step, because the landmarks of the map will only be the ones at the previous time step and at the next time step the landmarks of the map will be only the landmarks detected and triangulated at that time step.

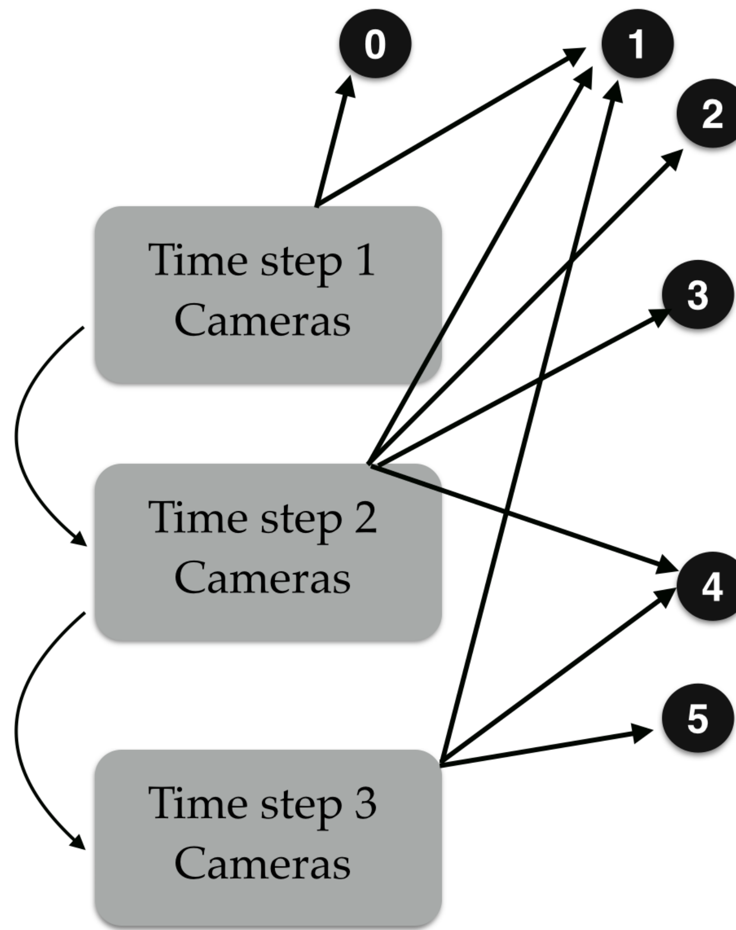

Figure 5.2: Cameras at different time steps observing different features

If we take a look at the picture above we will see various pair of cameras at different time steps, and therefore at different places observing features. At time step 1 the cameras observe only feature 1. After triangulation this becomes our 3D map. After we move we re-observe feature 1 . This, as explained before will help us make the correction step. Now, if we were trying to implement a full SLAM points 2, 3, and 4 would have to be triangulated and added to the map as new observed landmarks. But because our implementation only takes 2

time steps at a time we will triangulate the points seen at time step 2 (1,2,3,4) and now this will be our new 3D map. Notice that point 0 has been seen only by the cameras at time step 1, so this point 0 would not be in our new 3D map because we have forgotten about the 3D map created at time step 1. Now after we move to time step 3 we re-observe point 4. Again this will help us with the correction step. Now imagine that at this time step we would have also observed point 0. This point would have not helped in our implementation because we have forgotten about the 3D map from time step 1. A full SLAM implementation would have not forgotten about it and used it as another point for the correction step. After this we would only triangulate points 4, 5, and 1 to be our new 3D map.

Having said this, this step is very important to functional SLAM algorithm and that is why we will briefly explain it very theoretically.

After step 2, when we observe landmarks at a new time step *k+1* that are already mapped , and we make all the correct stage operations thank to these re-observed landmarks, we would like to introduce the landmarks observed at *k+1* that were **not previously mapped.** This will make our map richer as we move through the environment. To do this we will utilise the inverse observation model. In this model we have a observed landmark relative to the agent and we want to make this landmark absolute to the world coordinates. Again utilising homogeneous coordinates what we do is apply the composition operator between the localisation of the moving agent and the observation z at that time step to obtain the absolute coordinates.

$$\boldsymbol{m_{new}} = x_{k+1} \oplus z_{k+1} \tag{5.33}$$

In the last column we would obtain the coordinates $x,\ y,\ z$ that would be the absolute coordinates of that new landmark.

We would also have add information to the covariance matrix. We would have to add a new row and column to the covariance matrix.

$$\begin{bmatrix} P_{xx} & P_{xm1} & P_{xm2} & ... & P_{xx}\frac{\partial m_{new}}{\partial x}^T \\ P_{m1x} & P_{m1m1} & P_{m1m2} & ... & P_{m1x}\frac{\partial m_{new}}{\partial x}^T \\ ... & ... & ... & ... & ... \\ \frac{\partial m_{new}}{\partial x}P_{xx} & \frac{\partial m_{new}}{\partial x}P_{xm1} & \frac{\partial m_{new}}{\partial x}P_{mLm2} & ... & A \end{bmatrix} \tag{5.34}$$

With L representing the landmark we have just introduced to the map, and $\frac{\partial m_{new}}{\partial x}$ representing the Jacobian of the inverse observation function respect the localisation of the moving agent. A represents the following equation:

$$\frac{\partial m_{new}}{\partial x}P_{xx}\frac{\partial m_{new}}{\partial x}^T + \frac{\partial m_{new}}{\partial z_L}R\frac{\partial m_{new}}{\partial z_L}^T \tag{5.35}$$

With z_L being the observation of the landmark.

It is rarely found a full SLAM solution that keeps track of all previous time steps. Most solutions use **windows** which focus only on a finite and sometimes

small number of time steps that allows the algorithm to be functional, efficient and at the same time robust enough.

Chapter 6

Results

In our project we were not able to obtain experimental results from our implementation. Nonetheless we will show the the results obtained from Jose Luis Blanco in his implementation to demonstrate the functionality and advantages of this specific algorithm. In this implementation we have considered the observations sequentially, one scalar value at a time. The naive implementation would be to take the whole vector of observations at once and operating directly with it and the other complete matrices.

In his report, Blanco states that he has implemented the algorithm using the two ways and gives the results for both of them. For his test, he used 100 known landmarks that should have been mapped.At the end of his experiment 92 of this 100 landmarks were mapped.

Value	**Naive method**	**Improved method**
Overall time	293 sec	17.15 sec
Average execution rate	7.5 Hz	128.2 Hz
Average landmark error	0.157 m	0.138 m
Maximum landmark error	0.278 m	0.245 m
Minimum landmark error	0.037 m	0.064 m

Figure 6.1: Results obtained from "Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM" by Jose Luis Blanco

As we can see the improved method is a lot faster and the average landmark error is smaller.

Chapter 7

Budget

In this Chapter a budget for this project or for any that tries to study a similar solution will be given.

7.1 Working Hours

The budget for the working hour of this project is given in Table 7.1

Details	Hours	Cost/Hour	Cost
Engineer Hours	260	12 €	3120 €
Vicomtech PhD/Supervisor Hours	70	35 €	2450 €
TECNUN PhD/Supervisor Hours	30	35 €	1050 €

Table 7.1: Working hours budget

7.2 Hardware and Software

Details	Usage Hours	Cost/Hour	Cost
Desktop Computer	260	4,61 €	1200 €
Visual Studio C++	190	0 €	0 €
Matlab Student Suite	10	6,9 €	69 €
OpenCV Library	190	0 €	0 €

Table 7.2: Hardware and Software budget

Chapter 8

Conclusions

In the implementation , even though the code was compiled, logical experimental results could not be obtained.

However, we have given a theoretical solution to this slam problem in our system and created a code implementing many of the steps that could lead to a full slam solution.

This project has, first of all, reused the code from [1] to transform the odometry data into data that can serve as input for an EKF-SLAM algorithm. This project also has properly implemented all the mathematical operations between matrices needed to compute the EKF algorithm. Most of this operations have been done by taking advantage of the OpenCV library and the functions it offers for treating matrices, for example the rect() function, and as the results from Jose Luis Blanco say, this implementations is faster and more robust than a naive implementation. A solution to finding Jacobians was also found and implements using the Matlab symbolic toolbox. This is very helpful for future work in this subject.

Also data has been successfully associated, creating mask vectors that allow us to differentiate between re-observed landmarks and landmarks not observed yet.The framework for a future full slam solution that takes into account all the time steps 3D features has also been established for future work.

Finally all this was put together to create a compilable code that should implement a first approach solution to a SLAM problem. As it happens many times there are bugs and small errors that have not permitted coherent results to be taken from this project.

In the next Chapter we will try to give a couple of improvements that could make our project functional or more efficient.

Chapter 9

Future Work

Obvious work has to be done on correcting the probable errors mentioned before but in this Chapter the following steps that I think should be followed to make this solution a feasible one will be presented.

9.1 Improvements in algorithm

As we have seen we only take two consequent time steps for the operations. A full SLAM solution would need to at least take into account a small window to have a more robust map. Because of how we do the matching this really becomes a problem of code because the complication is the need to arrange, save and keep track every observed feature to realise if it has ever been seen before or not. Another option for this is, instead of keeping the masks, just keeping all the previous points and make a new matching algorithm. The question is if this would be computationally effective. The most probable errors can be found in the covariance matrix which we have defined with arbitrary values, used by some other implementations that have been posted to the public. As we said in the beginning, when we were just explaining the history of slam, the real breakthrough of slam came when researchers discovered that the key to the solution was in the correlation between the landmarks. This is a key value and obtaining it in a theoretical way is extremely complicated. The best way to obtain this is experimentally.

9.2 Improvements in code

This is probably where our project needs the most work. Not because the programmed code is bad, but because after the theoretical solution the implementation is very complicated. There are many possible sources of errors. The first source is all of the operations that are being done. We work with very big matrices and lots numbers. Sometimes this numbers can be very small and precision can be lost in these long operations involving big matrices.

Other possible source of error is when we obtain the yaw, pitch and roll from the rotation sub matrix. This process is very delicate and various solutions can be found.

Even though a full slam solution could not be implemented the theoretical work is correct according to the various reports and papers reference ins this project. The implementation has to be improved but is a good first approach to a functional code especially when implementing the operation in the filter to make the algorithm more efficient.

9.3 Obtaining Correlation between landmarks

Just like it was mentioned earlier in this project, the key to the solution lies on the correlation between the landmarks. This is actually easy to see, as the filter we are trying to implement focuses on a mean and variance of its elements and tries to eliminate the uncertainty by operating with these values.

A good project for a future student would be to experimentally try to obtain these correlation values. It could be done by providing a map of known landmarks and a trajectory of the moving agent known by the student and by implementing only a visual odometry algorithm like the one provided by [1], the student could try to find how the errors in the landmarks and vehicle positions relate to each other. It would mostly be an empirical project, as the student would have to do several experiments with videos but at the end theoretical knowledge would have to be applied to find a proper model to compute these correlations. If proper values are obtained for our system then the next step, in my opinion could create another interesting project.

9.4 Future Implementation

Once this correlation values have been found, an implementation should follow. But these implementation should not go directly to C++, but it should be done in a programming environment which facilitates treatment with math and matrices. For example, Matlab. These would help by reducing errors in the operations and by making it a lot easier to debug the code and the operations in case something is wrong. The real challenge here would be to pass the data form the visual odometry module in C++ to Matlab. It is not an impossible task but it could become a tedious one.

If the algorithm works well in Matlab and gives positive results, then a simple implementation in C++ should follow. If this implementation works then the problem would become a optimisation project, as it should be optimised as best as possible for the algorithm to be computationally effective. Then ways to make the algorithm more robust should start being thought about it. Maybe including more sensors or other type of navigation resource. After this the process to implementing it to hardware could begin. This will also surely be a complicated process, but a necessary one if someday a SLAM solution wants to

be placed on any type of moving agent.

With the theoretical work done in this project, and the implementation a framework has been established for future work that could hopefully end up one day being a full SLAM solution to autonomous navigating systems.

Bibliography

[1] L. de Maeztu, U. Elordi, M. Nieto, J. Barandiaran, and O. Otaegui, “A temporally consistent grid-based visual odometry framework for multi-core architectures”, Journal of Real Time Image Processing, 2014, (DOI: 10.1007/s11554-014-0425-y).

[2] Hugh Durrant-Whyte and Tim Bailey. ”Simultaneous Localization and Mapping: Part I” ,IEEE Robotics & Automation Magazine ,pages 99-108,June 2006.

[3] Søren Riisgaard and Morten Rufus Blas. ”SLAM for Dummies, Massachusetts Institute of Technology, $http : //ocw.mit.edu/courses/aeronautics - and - astronautics/16 - 412j - cognitive - robotics - spring - 2005/projects/1aslam_blas_repo.pdf$

[4] J.J. Leonard and H.F. Durrant-Whyte, “Simultaneous map building and localisation for an autonomous mobile robot,” in Proc. IEEE Int. Work- shop Intell. Robots Syst. (IROS), Osaka, Japan, 1991, pp. 1442–1447.

[5] R. Smith, M. Self, and P. Cheeseman, “Estimating uncertain spatial relationships in robotics,” in Autonomous Robot Vehicles, I.J. Cox and G.T. Wilfon, Eds. New York: Springer-Verlag, pp. 167–193, 1990.

[6] S. Thrun, D. Fox, and W. Burgard, “A probabilistic approach to con- current mapping and localization for mobile robots,” Mach. Learning, vol. 31, no. 1, pp. 29–53, 1998.

[7] S. Thrun, D. Fox, and W. Burgard, “Probabilistic Robotics”, Chapter 3, Section 3.3, pp. 48-49 .

[8] Bradley Hiebert-Treuer, “An Introduction to Robot SLAM (Simultaneous Localization And Mapping) ”, Middlebury College.

[9] Jose-Luis Blanco,“Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM”, 2008, University of Malaga.

[10] M. Dissanayake, P. Newman, S. Clark, HF Durrant-Whyte, and M. Csorba. “A solution to the simultaneous localization and map building (SLAM) problem.” IEEE Transactions on Robotics and Automation, 17(3):229–241, 2001.

[11] Nghia Ho,“Decomposing and composing a $3x3$ rotation matrix”, $http://nghiaho.com/?page_id = 846$.

List of Figures

1.1 Diagram of evolution of a SLAM system [1] 4

2.1 The coordinate system that will be used in this project for the position of the moving agent. 6
2.2 An example of a stereo camera system. 7

5.1 Only the green dot representing a feature will comply with the circular match [1]. 19
5.2 Cameras at different time steps observing different features . . . 26

6.1 Results obtained from "Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM" by Jose Luis Blanco 29

List of Tables

7.1 Working hours budget 30
7.2 Hardware and Software budget 30