POLITECNICO DI MILANO

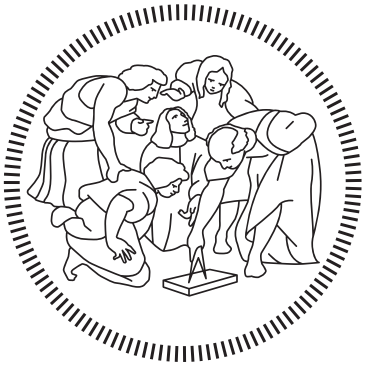
Facoltà di Ingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in

Computer Science and Engineering



A ROS implementation of a 6-DoF EKF for indoor drone Visual SLAM

Advisor: PROF. MATTEO MATTEUCCI

Master Graduation Thesis by:

DIEGO EMANUEL AVILA

Student Id n. 903988

Academic Year 2019-2020

A mis abuelos, a quienes siempre llevo en mi corazón.

ACKNOWLEDGMENTS

First of all, I want to thank Prof. Matteo Matteucci for trusting in me to work in this project and for his infinite patience to guide me. Without his guide, I would still be reading papers and trying to understand some processes and results. I want to thanks Simone Mentasti and Gabriele Roggi for their advises, I hope the next student find your advises and this work as useful as I did.

Second, I want to thank my girlfriend and future wife, Victoria, for convince me to start this journey, it wouldn't be the same without you and your support. Of course, I want to thank my family in Argentina, this wouldn't be possible without you either. Thank you for your support and encourage, it is a pity that because of this pandemic you cannot be here with me. I miss you, and I hope we can be together again.

Finally, I want to thank my friends, the ones I've made here in Italy and those in Argentina, particularly to Andrés. You've being a mentor and a guide, and without your advises and friendship I wouldn't be here.

To all of you, thank you.

Diego.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

EKF Extended Kalman Filter
ENU East-North-Up
IDL Interface Description Language
KF Kalman Filter
NED North-East-Down
NEES Normalized Estimation Error Squared
ROS Robot Operating System
RTAB-Map Real-Time Appearance-Based Mapping
RViz ROS Visualization
SLAM Simultaneous Localization And Mapping
UAV Unmaned Aerial Vehicle
UKF Unscented Kalman Filter
ViSP Visual Servoing Platform

ABSTRACT

In the context of the Leonardo Drone Contest, where specific constraints and characteristics apply, an implementation of a localization and mapping algorithm is proposed. The drone's constraints include the prohibition of usage of GNSS or laser devices, which enforces the need for the localization algorithm to be robust enough. Moreover, the characteristics of the environment make it possible to use a visual-based localization algorithm.

On the other hand, the contest requires to map specific landmarks in the environment in order to be able to follow those landmarks mapped in a sequence of take-off and landing in a specific order. In this sense, the algorithm is forced to map these landmarks so their coordinates can be used later on.

The current work proposes an implementation of an EKF-SLAM algorithm, and tries to shed some light on the importance of the usage of these landmarks, while trying to identify implementation details that increase the performance and robustness of the proposed algorithm.

SOMMARIO

Nell'ambito del Leonardo Drone Contest, dove si applicano vincoli e caratteristiche specifiche, viene proposta l'implementazione di un algoritmo di localizzazione e mappatura. I vincoli del drone includono il divieto di utilizzo di dispositivi GNSS o laser, che impone che l'algoritmo di localizzazione sia abbastanza robusto. Inoltre, le caratteristiche dell'ambiente consentono di utilizzare un algoritmo di localizzazione visuale.

D'altra parte, il concorso richiede di mappare punti di riferimento specifici nell'ambiente in modo da poter seguire una sequenza di decollo e atterraggio, in un ordine particolare, per quei punti di riferimento mappati. In questo senso, il concorso obbliga l'algoritmo a mappare questi punti di riferimento in modo che le loro coordinate possano essere utilizzate in seguito.

Il lavoro attuale propone un'implementazione di un algoritmo EKF-SLAM e cerca di comprendere l'importanza dell'uso di questi punti di riferimento, mentre cerca di identificare i dettagli di implementazione che aumentano le prestazioni e la robustezza dell'algoritmo proposto.

1

INTRODUCTION

In the last ten years the robotics industry has increased year by year, and companies are seeing robotics as a decisive technology. According to ABI Research's [1] "State of the robotics market" report, during 2018 the total investment in robotics was about 4.000 million U$S. Among these investments mobile robotics is recognized as one of the key trends, where the most important one is automated guided vehicles that move around contained environments, such as warehouses.

Within this context, Unmaned Aerial Vehicle (UAV) systems are a growing industry, and although expectations were not fulfilled last years, they are considered as an emerging market. The cited report states that "the largest use case is undeniably for inspection and maintenance" [1]. Nevertheless, the current pandemic has open a door to innovation, and UAVs are part of it [2]. Leonardo, one of the biggest Italian companies in the market, has noticed that autonomous mobile systems will be important for its products in the near future, and for this reason, have launched an Open Innovation project aimed to UAVs [3].

The implementation presented in this work should be framed in the participation of the Politecnico di Milano Colibrì Team for the Leonardo Drone Contest. The team is a collaboration of the DEIB and the DAER departments, and had very positive results winning the first edition of the contest.

THE CONTEST

The competition is composed of two phases. During the first phase the map of the environment is developed, while throughout the second phase the drone should follow a path of landmarks previously established. The drone should not have a GNSS signal nor can it be guided, this way, the drone is guaranteed to be fully autonomous, and as a result should complete both phases without any intervention.

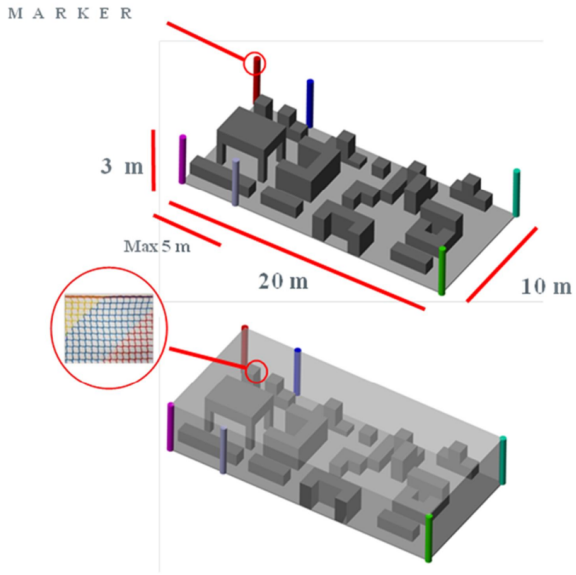


Figure 1.1: Example of proposed environment for the Leonardo Drone Contest.

The first phase the drone should map the environment, recognize the QR markers in the floor, avoid any obstacle and return to the initial position. During the second phase, and after the path of five landmarks is established, the drone should take off and land for at least five seconds in all the predefined markers in a specific order, and finally return to the initial position. The drone able to do the path in the right sequence is the winner of the competition.

The drone, as mentioned before, should not have any GNSS sensor nor LiDAR sensors, hence the localization and mapping algorithm should be based on visual information. The allowed sensors are inertial devices, like IMU or a magnetometer, range sensors, cameras and speed sensors, with the possibility of streaming capacity.

The indoor environment where the drone should move needs to have specific characteristics with obstacles and walls that delimit the space. Obstacles have at most three meters height with passages of at least one meter. An special QR marker of a size of one by one meter is disposed in the take-off area. The environment is 20 meters by 10 meters length, with the walls having a minimum of three meters height and made of net-like material. Furthermore, six poles are displaced in the following way: one per each corner and two in the middle axis, being each pole of different colors. The floor is a black, textured, pavement-like material. Obstacles are made of a high-visibility material, with a minimum volume of one cubic-meter.

This work aims to shed some light on the importance of the usage of different landmarks in the environment, and how the implementation reacts on these different landmarks. Furthermore, it has the purpose to understand the importance of different implementation decisions included in the algorithm and how these decisions impacted on the results.

ROAD MAP

The current work is divided in five chapters:

- **Chapter 2** introduces background information useful to completely understand the proposed implementation
- **Chapter 3** explains and comments the implementation in the context of the competition
- **Chapter 4** exposes the different experiments done in order to evaluate the implementation
- **Chapter 5** discusses over the results of the experiments and comments on possible future works and enhancements

2

BACKGROUND

2.1 ROBOT OPERATING SYSTEM

The Robot Operating System (ROS) [4] is an open source middleware that provides inter-process communication through a message passing mechanism. It is also a collection of libraries, tools and conventions with the aim of simplify the development of software for robots. In this sense, one of the main components of the framework are the nodes, which encapsulate processes and/or algorithms.

2.1.1 *Nodes*

Nodes are processes that perform a specific computation. In the system the nodes communicate between them using a publisher/subscriber infrastructure based on topics. A node subscribes to a particular topic, and it will receive messages from a publisher node. This way, a publisher node is hid to the subscriber, reducing the coupling between them. The advantage of this mechanism is that nodes can be developed separately once the message structure is defined, forcing developers to implement clear interfaces for communication by using a message Interface Description Language (IDL). Moreover, this enables a modular and distributed development of the robotic system, while providing some fault tolerance and reduced code complexity.

2.1.2 *Topics*

As mentioned before, communication between nodes is done via a publisher-subscriber architecture based on topics, which are named buses over which messages are exchanged. There can be several publishers (as well as subscribers) for a topic. A node that generates data publishes all its information in a specific topic bus, consequently this information is consumed by those nodes subscribed to that topic. The transport protocol used for exchanging messages is defined at runtime and can be of two types: TCPROS or UDPROS.

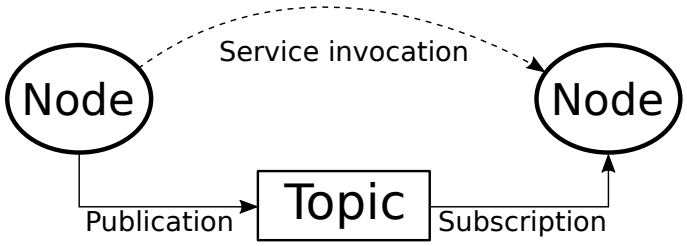


Figure 2.1: ROS basic concepts. Two nodes communicate between them using the messages published in a topic. One node is responsible of publishing the topic, while the other subscribes and receives the messages of that particular topic. Furthermore, the node in the left calls a service provided by the node in the right, so that node will perform some computation and will return the result. [4]

The description of these protocols is beyond the scope of this document and the reader is invited to read the ROS documentation for detailed information.

2.1.3 *Services*

As well as topics, services are a way to communicate nodes. The difference is that topics are an asynchronous way of communication, while services are synchronous. The key difference lies on the ability of nodes to decide when to trigger a service. Hence, services are used in between nodes to retrieve specific information or to request another node to carry out a computation beyond caller's node scope.

2.1.4 *Tools*

Regarding the tools provided by ROS, one that is crucial for debugging and/or experimentation is the bag recording and playback. Since the exchange of messages between nodes is anonymous (meaning that nodes communicate between each others without knowing which node sent or received a message) it is possible to record the messages during a period of time, without taking into consideration which node sent a message and which node received it. This recorder bag is useful for debugging since it can be played back, hence reproducing a previous experiment. Also, it can be used for the development of new nodes that depend on the messages contained in the bag.

Another useful tool provided is ROS Visualization (RViz), a 3D visualization tool. With this tool it is possible to see the robot, orientation, reference frames, covariance matrices, etc. In addition, it is possible to draw lines, arrows, text and others, onto the environment in order to see useful information that cannot be extracted from messages.

2.1.5 *MAVLink and MAVROS*

MAVLink is a binary telemetry protocol designed for resource-constrained systems and bandwidth-constraint links, more specifically for drones of all kinds. As with ROS, it adopts a publisher-subscriber architecture, where data streams are published as topics. Its key features, as published in their website are:

- Since its messages do not require any special framing, it is well suited for applications with very limited bandwidth.
- It provides methods for detecting package drops, corruption and for package authentication.
- Allows up to 255 concurrent systems on the network.
- Enables both offboard and onboard communications.

MAVROS is the extension of MAVLink in ROS, with the additive of being a proxy for Ground Control Station tool. Its main features, as published in its website are:

- Communication with autopilot via serial port, UDP or TCP .
- Internal proxy for Ground Control Station (serial, UDP, TCP).
- Plugin system for ROS-MAVLink translation.
- Parameter manipulation tool.
- Waypoint manipulation tool.
- PX4Flow support (by mavros_extras).
- OFFBOARD mode support.

One characteristic that worth mentioning, is that it translates the North-East-Down (NED) reference frames into East-North-Up (ENU) reference frames, and vice-versa, so as to be compliant with ROS standard reference frames.

2.1.6 *RTAB-Map*

Real-Time Appearance-Based Mapping (RTAB-Map) is a RGB-D, stereo and LiDAR graph-based Simultaneous Localization And Mapping (SLAM) approach based on an incremental appearance-based loop closure detector. It allows to build a 3 dimensional map of the environment using a stereo camera, a LiDAR sensor and/or the odometry estimation. The map is built using a 3D occupancy grid by means of Octomap library, which builds a

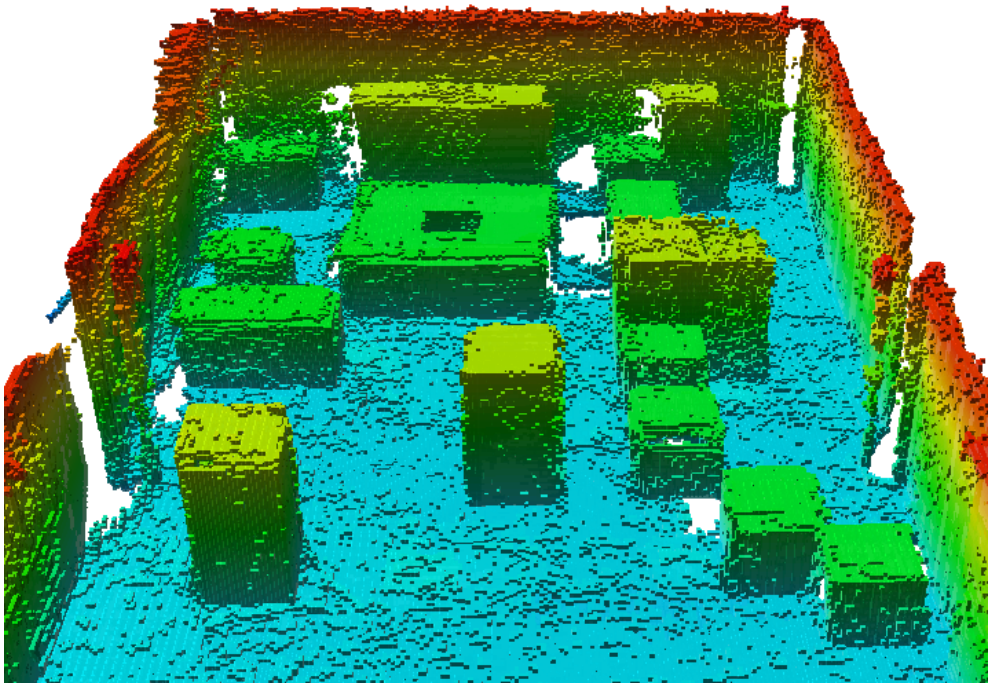


Figure 2.2: Octomap example. Octotree representation generated from data, showing occupied voxels only. [5]

tree-based representation of the mapped area. Octomap library "performs a probabilistic occupancy estimation to ensure updatability and to cope with sensor noise"[5].

The Octomap library generates an octotree, which is a hierarchical data structure used to generate spacial subdivision in 3 dimensions. Each node in an octotree represents a voxel, which is a cubic volume, which is at the same time, subdivided in eight smaller cubes until a given minimum voxel size called resolution. This way, the octotree data structure is used to build an occupancy grid of a volume. An example of a octotree can be seen in Figure 2.2, where the color of each occupied voxel represents its height: higher voxels are red-colored, while lower voxels are light-blue-colored.

RTAB-Map library, uses the Octomap library to store the 3D occupancy grid of the environment. The `rtabmap_ros` node is the extension of the RTAB-Map library for ROS. It uses the information of the stereo camera or a RGB-D camera and/or a LiDAR information to create a cloud point that is used to build the map. Furthermore, it uses the provided odometry to estimate and correct the robot's pose into the 3D map. An example of this procedure can be seen in Figure 2.3.

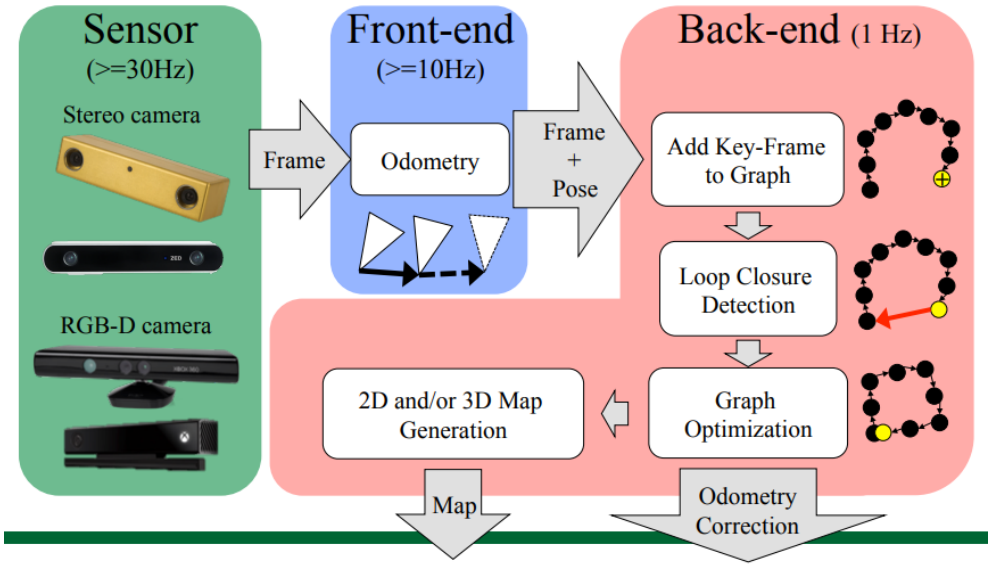


Figure 2.3: RTAB-Map example. An stereo camera or a RGB-D camera is used among the odometry estimation. Both are used to generate a 3D occupancy grid using Octomap library, and to correct the odometry estimation. [6]

2.2 TRANSFORMATIONS

A robot, in this case the drone, can be considered as a rigid body that moves and rotates around the environment. Hence, it is necessary to model the drone displacement in space, and this can be done by decomposing the movement as translations and rotations.

2.2.1 *Translation*

Given a vector $\boldsymbol{b} = \begin{bmatrix} b_x & b_y & b_z \end{bmatrix}^T$ that represents the center of mass of a rigid body, a translation that displaces $\boldsymbol{b}$ parallel to itself of a given vector $\boldsymbol{t}$ can be defined as:

$$\boldsymbol{b} + \boldsymbol{t} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = \begin{bmatrix} b_x + t_1 \\ b_y + t_2 \\ b_z + t_3 \end{bmatrix}$$

Given a point $\mathcal{P}$ represented in the reference frame $\mathcal{R}_m$ by $\boldsymbol{v}_P^m$ will be represented in the reference frame $\mathcal{R}_n$ by:

$$\boldsymbol{v}_P^n = \boldsymbol{v}_P^m + \boldsymbol{t}_m^k$$

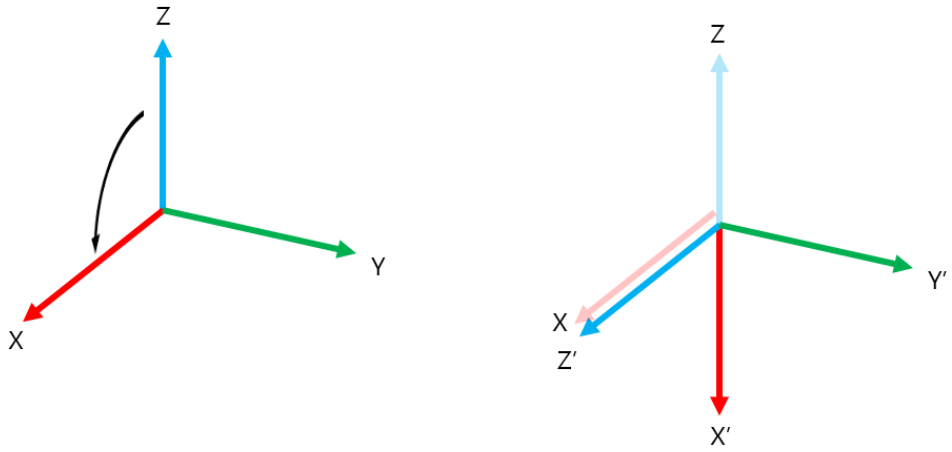


Figure 2.4: Rotation around origin. The reference frame at the left is rotated around Y axis and, as result, a new reference frame is obtained, both sharing the same origin.

2.2.2 *Rotation*

In three dimensional space, any displacement of a rigid body is equivalent to a single rotation of a given angle about some axis that contains the point. It is possible to express the previous statement as $\boldsymbol{v} = \boldsymbol{u} * \theta$, being $\boldsymbol{u}$ a unit vector representing the axis and θ the angle.

Unlike with translations, two or more rotations cannot be simply the sum of the related vectors. A rotation can be described assuming a rigid body in a reference frame with its origin fixed, while the unit vectors are changed under the rotation. This way, a rotation is characterized by the mathematical relation of these two reference frames. Hence, in order to represent a rotation, two reference frames with a common origin are needed, as shown in Figure 2.4. At the beginning the two reference frames coincide, then one of them is rotated around the origin by an arbitrary angle. After this procedure, the two reference frames are not coincident anymore, however, both share the same origin.

The rotation by any angle of any axis can be represented in a matrix form, $\boldsymbol{R}_n^m$, where each column of the matrix represents the unit vector of $\mathrm{R_n}$ in $\mathrm{R_m}$, so it represents the rotated frame $\mathrm{R_n}$ with respect the fixed frame $\mathrm{R_m}$ with common origin. Furthermore, it is possible to show that matrix $\boldsymbol{R}_n^m$ is *orthonormal*, which means that its inverse is equal to its transpose.

Given the matrix representation of a rotation, it is possible to provide a way to rotate a vector in the space. When the reference frames origins are the same,

the rotation of a vector is possible by multiplying it by the rotation matrix around an axis: $\boldsymbol{v}^m = \boldsymbol{R}_n^m \boldsymbol{v}^n$. This kind of rotations are called *elementary rotations*, and given that there are three axis in a 3D space, three different elementary rotations can be defined:

1. Rotation around axis x:

$$\boldsymbol{R}_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

2. Rotation around axis y:

$$\boldsymbol{R}_{y,\theta} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

3. Rotation around axis y:

$$\boldsymbol{R}_{z,\theta} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

As mentioned, unlike translations, rotations cannot be summed. Hence, if two consecutive rotations need to be performed around two axis, a multiplication is needed:

$$\boldsymbol{R}_{z,x} = \boldsymbol{R}_{x,\alpha} * \boldsymbol{R}_{z,\beta} \tag{2.1}$$

where $\boldsymbol{R}_{z,x}$ is the rotation around axis z by an angle β followed by a rotation around the new axis x by an angle α, as shown in Figure 2.5.

2.2.3 *Homogeneous Transform*

A rotation and a translation can be accomplished as follow: $\hat{\boldsymbol{v}} = \boldsymbol{R}\boldsymbol{v} + \boldsymbol{t}$. Nevertheless, this can be achieve by using *homogeneous coordinates*, where the point $\boldsymbol{v}$, expressed in its homogeneous representation, is $\boldsymbol{v} = \begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}^T$. This way, the rotation and translation of a point can be accomplished by multiplying the 3D point with the homogeneous transformation matrix:

$$\boldsymbol{T}_m^n = \begin{bmatrix} \boldsymbol{R}_m^n & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{2.2}$$

$$\hat{\boldsymbol{v}}^n = \boldsymbol{T}_m^n \boldsymbol{v}^m \tag{2.3}$$

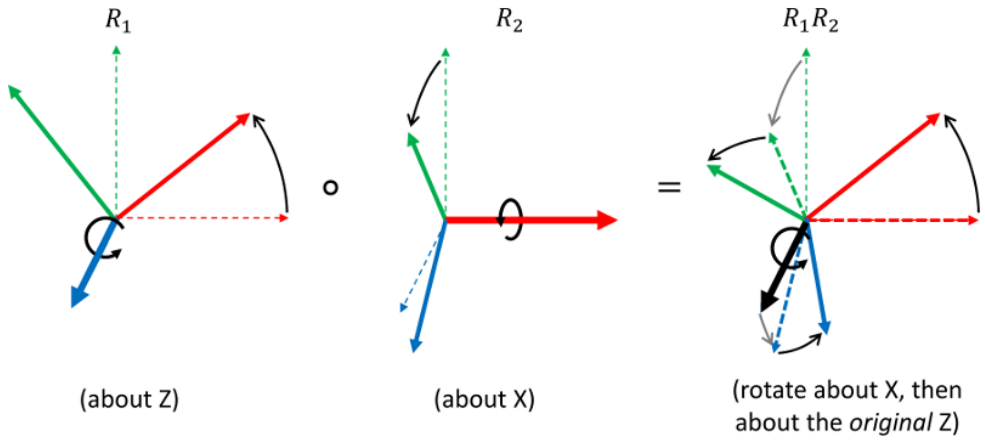


Figure 2.5: Rotation about axis z followed by rotation about axis x. The second rotation is performed about the original axis z. This final rotation can be achieve by multiplying R_1 and R_2.[7]

The rotation component of $\boldsymbol{T}_m^n$ needs to be defined. Several definitions are possible, and in this work the *RPY* order is used. RPY order of rotations establishes the order of the rotations applied to the three axis given three different angles. Therefore, $\boldsymbol{R}_m^n$ is a sequence of rotations around the X axis, Y axis and Z axis, given ϕ, θ and ψ angles, which is the same as $R_{z,\psi}R_{y,\theta}R_{x,\phi}$.

As can be noticed in equation (2.2), the homogeneous transformation is represented by a 4×4 matrix, and unlike the pure rotation matrix showed in Section 2.2.2, the inverse of the homogeneous transformation is not equal to its transpose. The inverse of the homogeneous transformation is equivalent to:

$$\boldsymbol{T}^{-1} = \begin{bmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T\boldsymbol{t} \\ 0 & 1 \end{bmatrix} \tag{2.4}$$

2.3 KALMAN FILTER

Introduced by [9], the Kalman Filter (KF) provides a recursive method for estimating the state of a dynamic system with presence of noise. It is a technique for filtering and prediction in *linear Gaussian systems* and applicable only to continuous states. One of its key features is that it simultaneously maintains estimates of the state vector and the estimate error covariance. It assures that the posteriors are Gaussian if the Markov assumption is hold, in addition to three conditions. The Markov assumption states that the past and future data are independent if one knows the current state; the other three conditions are the following:

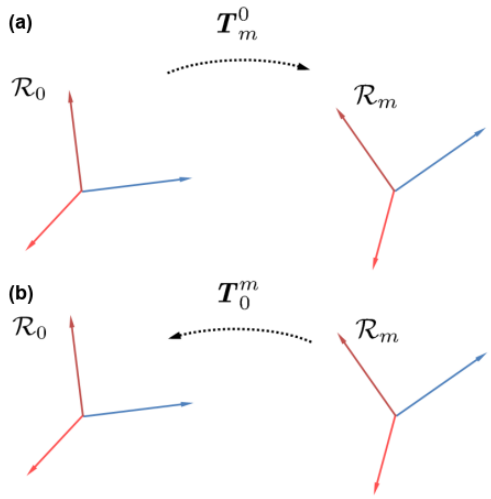


Figure 2.6: Example of a Homogeneous transform. **(a)** Transformation T_m^0 is applied to a point in reference frame $\mathcal{R}_0$ to express it in reference frame $\mathcal{R}_m$. **(b)** The inverse transformation is applied, so to express the point in reference frame $\mathcal{R}_m$ into reference frame $\mathcal{R}_0$. [8]

1. The state transition probability $p(x_t|u_t, x_{t-1})$ must be a linear function in its arguments. This is guaranteed by $x_t = A_t x_{t-1} + B_t u_t + \epsilon_t$. Here, x_t and x_{t-1} are the state vectors of size n, and u_t represents the control vector of size m at time t; A_t and B_t are matrices of size $n \times n$ and $n \times m$ respectively. In this way, the state transition function becomes linear in its arguments, hence KF assumes a linear system dynamics.

   ϵ_t represents the uncertainty introduced by the state transition, and it is a Gaussian random variable with zero mean and R_t covariance.

2. The measurement probability $p(z_t|x_t)$ must be linear in its arguments. This is guaranteed by $z_t = C_t x_t + \delta_t$, where z_t represents the measurement vector of size k, C_t is a matrix of size $k \times n$ and δ_t is a Gaussian noise with zero mean and Q_t covariance.

3. The initial belief should be normally distributed.

Given these three conditions, we are guaranteed that the posterior probability is Gaussian.

The KF can be seen in Algorithm (2.1) and its input is the belief at time $t-1$, represented by its mean (μ_{t-1}) and its covariance (Σ_{t-1}), in addition to the control vector (u_t) and the observations (z_t). As a result, it returns the current belief characterized by its mean μ_t and covariance Σ_t.

Algorithm 2.1: Kalman Filter algorithm

Input: μ_{t-1}, Σ_{t-1}, u_t, z_t

1 $\hat{\mu}_t = A_t \mu_{t-1} + B_t u_t$
2 $\hat{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
3 $K_t = \hat{\Sigma}_t C_t^T \left(C_t \hat{\Sigma}_t C_t^T + Q_t \right)^{-1}$
4 $\mu_t = \hat{\mu}_t + K_t \left(z_t - C_t \hat{\mu}_t \right)$
5 $\Sigma_t = (I - K_t C_t) \hat{\Sigma}_t$
6 **return** μ_t, Σ_t

The first step in the algorithm (lines 1 and 2), represent the prediction step. It calculates the current belief before incorporating the observations, but after adding the control vector. The estimated belief is characterized by its mean, $\hat{\mu}_t$, and its covariance $\hat{\Sigma}_t$.

The second step (from line 3 to 5) starts by calculating K_t, the Kalman gain, which specifies the degree in which the observation is incorporated to the new state estimate. K_t can be seen as the weighting factor that weights the relationship between the accuracy of the predicted state estimate and the observation noise. When K_t is large, the observations have more importance in the final estimate; while if K_t is small, the observations do not have much importance in the correction step. Following, at line 4, the new mean is estimated by means of the Kalman gain and the *innovation*, which is the difference between the observation z_t and the expected measurement $C_t \hat{\mu}_t$. Finally, the new covariance is calculated.

2.3.1 *Extended Kalman Filter*

The linearity conditions that make the KF to work are, in some cases, far from reality: state transition functions and measurements are rarely linear in practice. The Extended Kalman Filter (EKF) works through a process of linearization, where nonlinear state transition and observation functions are approximated by a Taylor series expansion.

The Figure 2.7a shows the linear transformation of a random Gaussian variable, whose density function is $\mathcal{N}\left(x; \mu, \sigma^2\right)$. Assuming that the random variable is transformed using a linear function $y = ax + b$, the resulting random variable will be Gaussian with mean $a\mu + b$ and variance $a^2\sigma^2$.

However, as shown in Figure 2.7b, this does not happen if the transformation is not linear. In this case, assuming the original random variable is

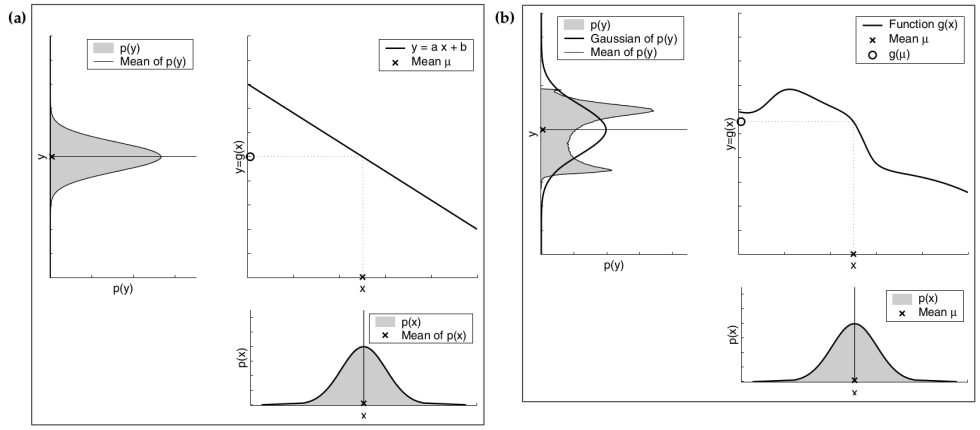


Figure 2.7: Linear **(a)** and nonlinear **(b)** transformation of a Gaussian random variable. The lower right plot shows the density function of the random variable. The upper right plot shows the transformation of the random variable. The upper left plot shows the resulting density function. [10]

transformed using a nonlinear function g, the density of the resulting random variable is not Gaussian anymore.

The state transition probability and observation probabilities are ruled by nonlinear functions g and h respectively. Matrices A and B are replaced by function $g(u_t, x_{t-1})$ and matrix H is replaced by function $h(x_t)$, making the belief not Gaussian. This is solved in EKF by approximating to the true belief, not the exact one as happens with linear KF. The approximation is done using a linearization method that approximates the nonlinear function by a linear function that is tangent to it, thereby maintaining the Gaussian properties of the posterior belief.

The used method is the first order Taylor expansion, which constructs a linear approximation of a function g from g's value and slope, which is given by

$$g'(u_t, x_{t-1}) = \frac{\partial g(u_t, x_{t-1})}{\partial x_{t-1}} \tag{2.5}$$

Since g depends on the control variable u and the state x, we need to define a value for x, and the logical choice is the mean of the posterior in the previous time step: μ_{t-1}. This way

$$g(u_t, x_{t-1}) = g(u_t, \mu_{t-1}) + g'(u_t, \mu_{t-1})(x_{t-1} - \mu_{t-1}) \tag{2.6}$$

where g' is the *Jacobian* of g, usually expressed as G_t, and it depends on u_t and μ_{t-1}, hence it changes through time.

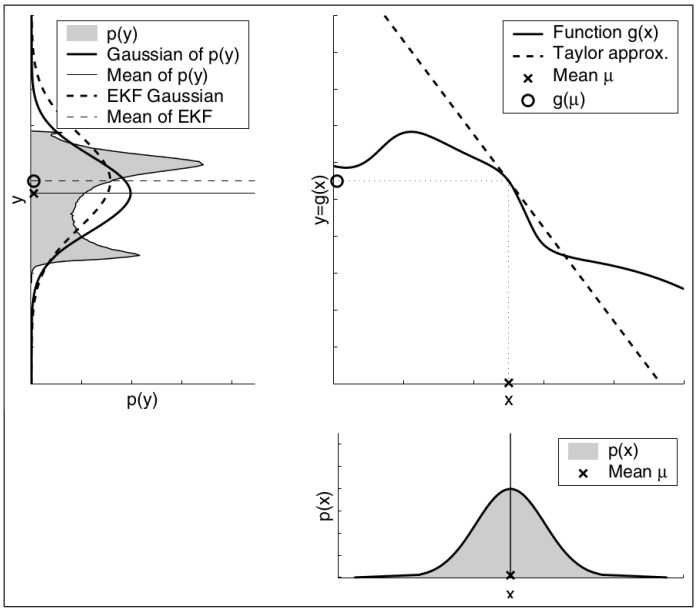


Figure 2.8: Linearization applied in EKF. In this case, the nonlinear function g is approximated using first order Taylor expansion, that is a linear function tangent to g at the mean of the original density function. The linearization is not perfect, so it adds an error, depicted in the upper left plot. This error is the difference between the dashed line and the solid line. [10]

The same linearization is applied to the observation function h:

$$h(x_t) = h(\hat{\mu}_t) + h'(\hat{\mu}_t)(x_t - \hat{\mu}_t) \tag{2.7}$$

$$h'(\hat{\mu}_t) = \frac{\partial h(x_t)}{\partial x_t} \tag{2.8}$$

where h' is the Jacobian of h, usually expressed as H_t. In this case, the linearization is done around $\hat{\mu}_t$, which is the state estimate just before computing h.

The Figure 2.8 depicts the approximation of g by a linear function that is tangent around its mean. The resulting density function is shown in the upper left plot with a dashed line, that is similar to the original density function.

The EKF algorithm can be seen in Algorithm (2.2), and it is similar to Algorithm (2.1). The difference lies in the usage of the nonlinear functions g and h and their Jacobians, G_t and H_t respectively.

Algorithm 2.2: Extended Kalman Filter algorithm

Input: μ_{t-1}, Σ_{t-1}, u_t, z_t

1 $\hat{\mu}_t = g(u_t, \mu_{t-1})$
2 $\hat{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
3 $K_t = \hat{\Sigma}_t H_t^T \left(H_t \hat{\Sigma}_t H_t^T + Q_t\right)^{-1}$
4 $\mu_t = \hat{\mu}_t + K_t (z_t - h(\hat{\mu}_t))$
5 $\Sigma_t = (I - K_t H_t)\hat{\Sigma}_t$
6 **return** μ_t, Σ_t

2.4 SIMULTANEOUS LOCALIZATION AND MAPPING

Among all the problems faced by autonomous mobile robots, two of them are relevant for this work: localization and mapping. The former one, is related to the problem of where the robot is, while the later is related to building a map of the environment. However, in order to accurately localize itself the robot needs a map of the environment in which it is immersed in, and, in order to build a map, it needs to know where it currently is, giving us a chicken-egg situation. Hence, the SLAM problem appears when the robot has no knowledge of its localization nor its environment map, while measurements and controls are given.

The problem of building a map can be summarized in the following steps:

1. The robot senses the environment using its sensors
2. It creates a representation of the acquired data
3. It integrates the processed sensor data with the previously learned map structure

While this process can be done by manually moving the robot around the environment, it is more challenging to build the map while the robot is moving autonomously.

On the other hand, assuming that the robot already knows the map, the localization problem could be trivial if no noise is present at all. The sensors, wheel encoders, different kinds of terrain, battery life, etc, all of these can make the robot to increase its uncertainty related to where it is. As robot moves around the environment it uses its sensors to estimate its position, increasing its uncertainty regarding its position relative to the map. At some point, the robot will "see" a known landmark or feature in the environment,

correcting its position while reducing the uncertainty.

The localization and mapping problems can be solved together by using a SLAM technique, with which the robot will build the map while localizing itself in it. An example of this problem can be seen in Figure 2.9, where a robot moves around the environment and sees some features or landmarks. The uncertainty regarding its position is low when it starts, and keeps growing while it moves around. At the end, it sees a known landmark (m_0) making the uncertainty to shrink. As can be seen in the figure, the robot adds new landmarks to the map (m_1 and m_2) with their corresponding uncertainty, and when the robot sees the first landmark, not only its own uncertainty decreases, but also the two new landmarks' uncertainty. In this way, the robot's position is correlated with the observations' position estimates.

The idea of the SLAM problem is to estimate a posterior belief that involves not only the robot pose, but also the map: $p(x_t, m|z_{1:t}, u_{1:t})$, where x_t is the robot's pose at time t, m is the map, $z_{1:t}$ are the measurements, and $u_{1:t}$ are the controls given to the robot.

2.4.1 *EKF-SLAM*

The SLAM problem can be addressed, between others, using an EKF approach. The algorithm proceeds in the same way as shown in Section 2.3.1, being μ a state vector containing the information for the robot pose (q_r) and the landmarks' pose (m_i):

$$\mu = \begin{bmatrix} q_r & m_0 & \dots & m_{n-1} \end{bmatrix}^T \tag{2.9}$$

One thing that is worth mentioning is that in EKF-SLAM maps are *feature based*. This means that the features or landmarks are assumed to be points in the space: if the robot sees, for example, a chair, it will store a point that represents the chair in space. Also, as explained before, it assumes Gaussian noise for the robot motion and observations.

The EKF-SLAM algorithm estimates the robot's pose in addition to all encountered landmarks' poses along its way. Thus, there is a correspondence between robot's pose and landmarks, and that is why it is necessary to include the landmarks information into the state vector. Hence, the algorithm estimates the posterior $p(\mu_t|z_t, u_t)$.

Assuming the robot's pose is composed by x, y, θ , the markers' pose is composed by x, y, and there are N markers, the state vector μ will have a length

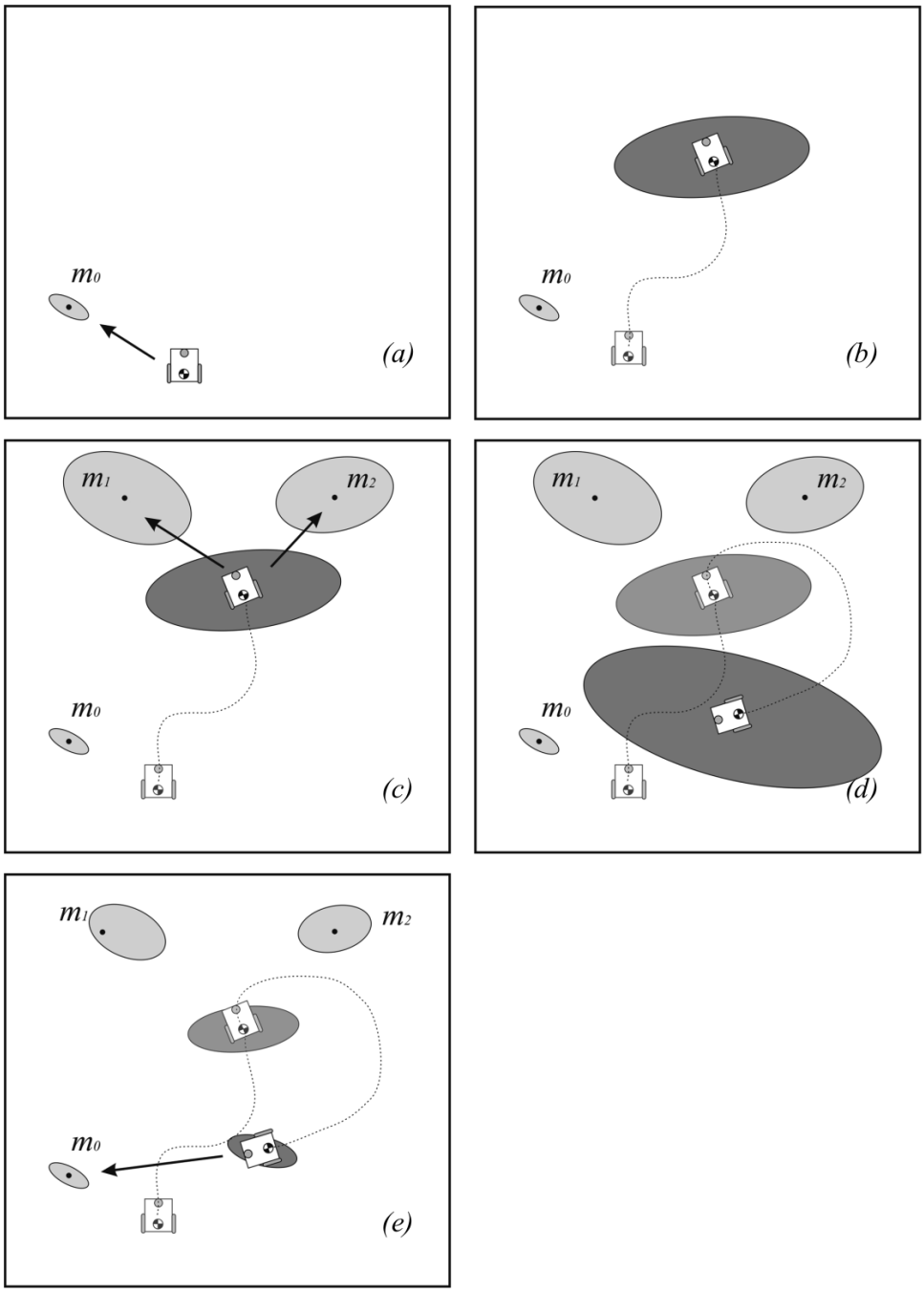


Figure 2.9: At the beginning **(a)** the robot has low uncertainty regarding its pose. As it moves around the environment its uncertainty, represented by the dark gray ellipsis, grows **(b), (c), (d)**, until it sees a known landmark **(e)**, making the position uncertainty to shrink. [11]

Algorithm 2.3: EKF-SLAM algorithm

Input: μ_{t-1}, Σ_{t-1}, u_t, z_t

1 $\hat{\mu}_t = g(\mu_{t-1}, u_t)$;
2 $G_t = \mathtt{computeJacobian}(g)$;
3 $\hat{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$;
4 **foreach** *landmark observation* z_t^i **do**
5 **if** *landmark* i *has not being seen before* **then**
6 $\mathtt{addLandmarkToStateVector}(z_t^i)$
7 $H_t^i = \mathtt{computeJacobian}(h^i)$;
8 $v^i = z_t^i - h^i(\hat{\mu}_t)$;
9 $S = H_t^i \hat{\Sigma}_t H_t^{iT} + Q_t$;
10 $K_t^i = \hat{\Sigma}_t H_t^{iT} S^{-1}$;
11 $\mu_t = \hat{\mu}_t + K_t^i (v^i)$;
12 $\Sigma_t = (I - K_t^i H_t^i)\hat{\Sigma}_t$;
13 **return** μ_t, Σ_t

of $3 \times 2N$, and the covariance matrix Σ will have a size of $(3 \times 2N) \times (3 \times 2N)$.

The algorithm can be seen in Algorithm (2.3). From lines 1 to 3, it computes the state vector and covariance matrix updates; the function $\mathtt{computeJacobian}$ computes, indeed, the Jacobian matrix for the motion model g, and the resulting matrix has the same size as the covariance matrix and has the following characteristic:

$$G_t = \begin{bmatrix} G_r & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{2.10}$$

$$G_r = \begin{bmatrix} \frac{\partial x'}{\partial \mu_{t-1,x}} & \frac{\partial x'}{\partial \mu_{t-1,y}} & \frac{\partial x'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial y'}{\partial \mu_{t-1,x}} & \frac{\partial y'}{\partial \mu_{t-1,y}} & \frac{\partial y'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial \theta'}{\partial \mu_{t-1,x}} & \frac{\partial \theta'}{\partial \mu_{t-1,y}} & \frac{\partial \theta'}{\partial \mu_{t-1,\theta}} \end{bmatrix} \tag{2.11}$$

where $\frac{\partial x'}{\partial \mu_{t-1,x}}$ is the derivative of g along x' dimension, taken with respect to x, y.. at μ_{t-1}.

At line 4, it iterates through every observation z_t. At each time step t, a sensor obtains a set of observations z^i of one of the N landmarks. Each observation is associated with a map feature, and this association is accomplished by a prediction of the measurement that each feature would generate and a measure of the difference between the prediction and the sensor measure-

ment. The prediction of the measurements can be obtained by the observation model h^i, which result is a vector of predicted features. If the observation z^i comes from a landmark i, the following relation is hold:

$$z^i = h^i(x_t) + w^i \tag{2.12}$$

where x_t is the true state and w^i is the observation noise with covariance Q_t, and assumed to be zero mean, Gaussian, additive and independent of the process noise. If the landmark is not already in the state vector, at line 6 it is added by projecting the observation and calculating the landmark's pose, adding two new elements to the state vector and two new more columns and rows to the covariance matrix. At line 7 the Jacobian of the observation model is computed.

At line 8, the innovation is calculated while its covariance is calculated at line 9. The innovation measures the discrepancy between the predicted observation and the actual sensor measurement. At line 10 the Kalman gain is computed, and at line 11 the state vector is updated. The gain propagates the information through all the state vector, updating not only the robot's pose, but also the landmarks' poses.

The fact that the Kalman gain is not sparse is important, because observing a landmark not only improves the estimate of that landmark's pose, but also all the others, along with the robot's pose. This effect can be seen in Figure 2.9, with an additional explanation: most of the uncertainty of the landmarks' poses is caused by the robot's own uncertainty, so the location of those previously seen landmarks are correlated. When the robot gains information about its own pose, this information is propagated to the landmarks, and as result it improves the localization of other landmarks in the map.

2.4.2 *Adding new landmarks*

So far, the existence and accuracy of a map was assumed. Nevertheless, this is not always true, and the construction of a map should somehow be done. Hopefully, EKF-SLAM algorithm considers the possibility of adding new landmarks to the map while doing the localization process with known landmarks. This process of adding new landmarks to the map is done via what is called inverse observation model, which handles the sensor measurements, identifies the new landmark, and adds it to the state vector.

The inverse observation model will produce the coordinates of the new landmark in the map, and this information will be added to the state vector, which will increase its size (in this case) by two new elements.

$$h^{-1}(\boldsymbol{x_r}, \boldsymbol{z}) = \begin{bmatrix} x_l \\ y_l \end{bmatrix} \tag{2.13}$$

$$\boldsymbol{x_t} = y(\boldsymbol{x}, \boldsymbol{z}) = \begin{bmatrix} \boldsymbol{x} \\ h^{-1}(\boldsymbol{x_r}, \boldsymbol{z}) \end{bmatrix} \tag{2.14}$$

In equation (2.13) $\boldsymbol{x_r}$ refers to the elements in the state vector that corresponds to the robot's pose, and $\boldsymbol{z}$ refers to the observation elements, which can be, for example, range and bearing data. Since the state vector has a variable length, extending the covariance matrix is also needed whenever the robot sees a landmark that has not being seen previously. The extension of the covariance matrix is achieved as shown in equation (2.15).

$$\hat{\boldsymbol{\Sigma}} = \boldsymbol{Y_x} \boldsymbol{\Sigma} \boldsymbol{Y_x}^T + \boldsymbol{Y_z} \boldsymbol{Q}_t \boldsymbol{Y_z}^T \tag{2.15}$$

$$\boldsymbol{Y_x} = \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial x}{\partial x} \\ \frac{\partial h^{-1}}{\partial x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{G_x} \end{bmatrix} \tag{2.16}$$

$$\boldsymbol{Y_z} = \frac{\partial y}{\partial z} = \begin{bmatrix} \frac{\partial x}{\partial z} \\ \frac{\partial h^{-1}}{\partial z} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{G_z} \end{bmatrix} \tag{2.17}$$

where matrices $\boldsymbol{Gx}$ and $\boldsymbol{Gz}$ are the Jacobian of function h^{-1} with respect to the state vector and the observation vector respectively. By substituting the Jacobians in equations (2.16) and (2.17) into equation (2.15), the following matrix is obtained:

$$\hat{\boldsymbol{\Sigma}} = \begin{bmatrix} \boldsymbol{\Sigma} & \boldsymbol{\Sigma} \boldsymbol{G_x}^T \\ \boldsymbol{G_x} \boldsymbol{\Sigma} & \boldsymbol{G_x} \boldsymbol{\Sigma} \boldsymbol{G_x}^T + \boldsymbol{G_z} \boldsymbol{Q}_t \boldsymbol{G_z}^T \end{bmatrix} \tag{2.18}$$

The linearized covariance update can be factored as shown in equation (2.19) by assuming that $A \equiv 1$, $B \equiv 0$, $C \equiv 0$ and $D \equiv \boldsymbol{G_z}$

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} P & 0 \\ 0 & Q \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix}^T = \begin{bmatrix} APA^T + BQB^T & APC^T + BQD^T \\ CPA^T + DQB^T & CPC^T + DQD^T \end{bmatrix} \tag{2.19}$$

$$\hat{\boldsymbol{\Sigma}} = \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} \\ \boldsymbol{G_x} & \boldsymbol{G_z} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Sigma} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_t \end{bmatrix} \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} \\ \boldsymbol{G_x} & \boldsymbol{G_z} \end{bmatrix}^T \tag{2.20}$$

$$\tag{2.21}$$

2.5 NORMALIZED ESTIMATION ERROR SQUARED

The Mahalanobis distance is a measure of the distance between a point and a distribution. It is a way to measure how many standard deviations is away the point from the mean of the distribution. The Mahalanobis distance of an observation $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \dots \end{bmatrix}$ with mean $\boldsymbol{\mu} = \begin{bmatrix} \mu_1 & \mu_2 & \dots \end{bmatrix}$, and covariance matrix Σ is defined as:

$$D = \sqrt{(\boldsymbol{x} - \boldsymbol{\mu})\, \Sigma^{-1}\, (\boldsymbol{x} - \boldsymbol{\mu})} \tag{2.22}$$

Every time the drone observes a pole or a marker, a node responsible of identifying the landmark processes the features that distinguish it with respect to other landmarks, and publishes these features as a message. In the case of poles range and bearing information is provided, while in the case of markers the position and orientation information with respect to the camera frame is provided. Given this, each observation z for a pole, is composed by three measurements, and each observation of a marker is composed by six measurements. After observing a landmark the Algorithm (2.3) calculates the discrepancy between the observation i and the predicted observation by the innovation (v^i), and its covariance (S), as

$$\begin{aligned} v^i &= z_t^i - h^i\,(\hat{\mu}_t) \\ S &= H_t^i \hat{\Sigma}_t H_t^{iT} + Q_t \end{aligned}$$

The square of the Mahalanobis distance can be used in order to establish a correspondence between the observed measurements with the landmark features if the following is hold:

$$D_i^2 = v^i S^{-1} v^i < \chi^2_{d,1-\alpha} \tag{2.23}$$

where d is the size of h^i and $1 - \alpha$ is the desired confidence level. This test is called individual compatibility, and when applied to the predicted state can be used to determine the subset of observation features that are compatible with the observation.

Furthermore, a state estimate is consistent if its state estimation error $\boldsymbol{x} - \hat{\boldsymbol{x}}$ satisfies

$$\begin{aligned} \mathbb{E}\left[\boldsymbol{x} - \hat{\boldsymbol{x}}\right] &= 0 \\ \mathbb{E}\left[(\boldsymbol{x} - \hat{\boldsymbol{x}})\,(\boldsymbol{x} - \hat{\boldsymbol{x}})^T\right] &\leqslant \Sigma \end{aligned}$$

When the ground truth $\boldsymbol{x}$ is available, a Normalized Estimation Error Squared (NEES) can be performed to check the consistency of the filter. NEES test can

be defined as the squared Mahalanobis distance for the difference between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$, and consistency can be checked with a χ^2 test

$$\text{NEES} = (\boldsymbol{x} - \hat{\boldsymbol{x}})\, \Sigma^{-1}\, (\boldsymbol{x} - \hat{\boldsymbol{x}}) \leqslant \chi^2_{d,1-\alpha}$$

However, the ground truth is only available in simulated environments thus, the consistency of the filter is maintained by using observations that satisfy the test in equation (2.23). This way, the filter will discard observations that do not satisfy the innovation test, maintaining a consistent estimation of the state and therefore, the map.

3

EKF-SLAM IMPLEMENTATION

In Chapter 2 the EKF-SLAM algorithm in the context of SLAM was explained. As mentioned in Section 2.3, the algorithm can be summarized in two steps: prediction and correction. The first stage involves the prediction of the next state of the system, meanwhile, during the correction stage this estimation is updated. However, differently from the Algorithm (2.3), the proposed implementation adds a NEES test with th objective of improving the consistency of the filter. The updated version of the algorithm can be seen in Algorithm (3.1).

Algorithm 3.1: EKF-SLAM algorithm with NEES test

Input: μ_{t-1}, Σ_{t-1}, u_t, z_t

1 $\hat{\mu}_t = g(\mu_{t-1}, u_t)$;
2 $G_t = \text{computeJacobian}(g)$;
3 $\hat{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$;
4 **foreach** *landmark observation* z_t^i **do**
5 **if** *landmark* i *has not being seen before* **then**
6 $\text{addLandmarkToStateVector}(z_t^i)$
7 $H_t^i = \text{computeJacobian}(h^i)$;
8 $\nu^i = z_t^i - h^i(\hat{\mu}_t)$;
9 $S = H_t^i \hat{\Sigma}_t H_t^{iT} + Q_t$;
10 $K_t^i = \hat{\Sigma}_t H_t^{iT} S^{-1}$;
11 $e = \nu^{iT} S^{-1} \nu^i$;
12 **if** $e < \chi^2_\alpha$ **then**
13 $\mu_t = \hat{\mu}_t + K_t^i(\nu^i)$;
14 $\Sigma_t = (I - K_t^i H_t^i)\hat{\Sigma}_t$;
15 **return** μ_t, Σ_t

Line 11, computes the NEES value, and line 12 compares it with the χ^2 value for the specific landmark. If the NEES value is lower than the χ^2 threshold,

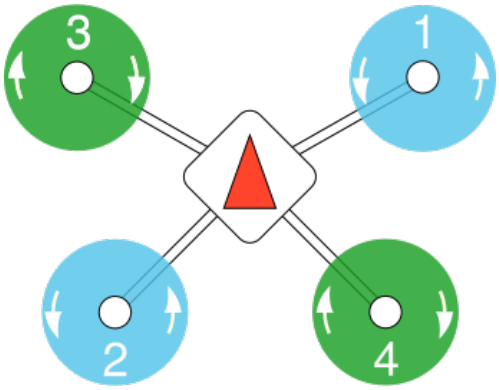


Figure 3.1: 3DR Iris Quadrotor frame [12].

the observation is accepted and the state vector and covariance matrix are updated.

In this chapter, an EKF-SLAM implementation is shown, starting from the used drone characteristics, going through the motion and observation models, visual process pipeline, and ending with the overall system's architecture.

3.1 THE DRONE

3.1.1 *Characteristics*

The drone considered for this analysis has a common characteristics, such as four rotors disposed as an X. The drone frame is called 3DR Iris Quadrotor, and can be seen in Figure 3.1.

3.1.1.1 *Flight Controller*

The flight controller used in this case is a PixHawk 4, and its characteristics are shown in Table 3.1. It is worth mentioning that it includes an integrated accelerometer, a gyroscope, a magnetometer and a barometer.

3.1.1.2 *Additional Sensors*

The drone was equipped with several sensors that are the input for localization, mapping and path planning algorithms, among others.

The localization and mapping algorithm makes use, in an indirect way, of monocular and stereo cameras, and range sensors. Four cameras were mounted in order to be able to have 360 range view, with cameras mounted every 90 degrees. Furthermore, two stereo cameras were mounted, one points

Item	Description
Main FMU processor	STM32F765 32 Bit Arm® Cortex®-M7, 216MHz, 2MB memory, 512KB RAM
IO Processor	STM32F100 32 Bit Arm® Cortex®-M3, 24MHz, 8KB SRAM
On-board sensors	Accel/Gyro: ICM-20689 Accel/Gyro: BMI055 Magnetometer: IST8310 Barometer: MS5611
GPS	ublox Neo-M8N GPS/GLONASS receiver; integrated magnetometer IST8310
Interfaces	8-16 PWM outputs (8 from IO, 8 from FMU) 3 dedicated PWM/Capture inputs on FMU Dedicated R/C input for CPPM Dedicated R/C input for Spektrum / DSM and S.Bus with analog / PWM RSSI input Dedicated S.Bus servo output 5 general purpose serial ports 3 I2C ports 4 SPI buses Up to 2 CANBuses for dual CAN with serial ESC Analog inputs for voltage / current of 2 batteries
Power System	Power module output: 4.9 5.5V USB Power Input: 4.75 5.25V Servo Rail Input: 0 36V
Weight and Dimensions	Weight: 15.8g Dimensions: 44x84x12mm
Operating temperature	-40 to 85°c

Table 3.1: Technical specification of the PixHawk 4 flight controller.

forward in order to update the Octomap and the other points downwards in order to see the markers; Also, both of them are used to build a 3-dimensional map, used for obstacle avoidance and with the height estimation algorithm. Furthermore, eight range sensors were mounted every 45 degrees, and one PIX4Flow optical flow camera points downwards in order to measure the distance to the ground.

3.1.2 *Reference Frames*

This system is composed by three reference frames linked with each other, as represented in Figure 3.2: *map*, *odom* and *base_link*. The *map* frame, also called *world* or *global* frame, is the static reference frame, where the global drone's position and global markers' position is set. The odom reference frame is similar to the map frame, but the difference is that this frame drifts with time, as happens with the pure odometry. Finally, the base_link frame, also referred as *body* or *local* frame, refers to the center of mass of the drone.

As mentioned before, all transformations within these reference frames are published by different nodes: the map to odom transformation is handled by the *rtabmap* node, the odom to base_link is handled by *mavros* node. There are many other transformations in the system, but they are mainly related to the base_link reference frame and the different cameras and sensors in the drone.

Finally, there is a transformation that worth mentioning: the NED to ENU. As mentioned in Section 2.1, ROS uses the ENU convention, while the convention adopted by the Aerospace community is the NED. Also, as mentioned in Section 2.1.5, MAVROS is in charge of doing and publish this transformation. In Figure 3.3 a simple scheme of the transformation can be seen. As explained before, this transform consists on rotating the X-axis and the Y-axis by 90 degrees, hence the homogeneous rotation matrix will have the following form:

$$R_{ENU}^{NED} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

3.2 MOTION MODEL

During the prediction step, the motion model and the covariance matrix update are computed. The motion model will update the state vector, which

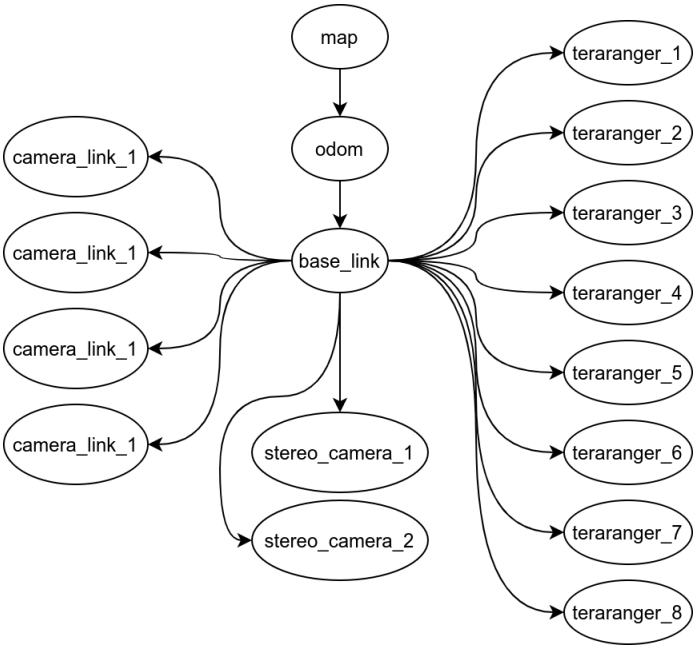


Figure 3.2: Main reference frames in the system.

will store the position X, Y and Z in the global reference frame, and the orientation in the Z-axis of the drone. Fortunately MAVROS provides a node that makes odometry estimation based on different sensors outputs. The messages published by the odometry node, of type `Odometry`, provide the linear velocity, the angular velocity and pose information. The velocity information is used to estimate the position in X, Y and Z, and the drone's orientation along the Z-axis. However, the velocity estimation provided by MAVROS is relative to the body reference frame, and this has to be transformed into the world reference frame, so before estimating the global position of the drone it is mandatory to do this transformation.

$$u = \begin{bmatrix} v_x^b & v_y^b & v_z^b & \omega_x^b & \omega_y^b & \omega_z^b & \phi^b & \theta^b & \psi^b \end{bmatrix}^T \tag{3.2}$$

$$v^w = \mathbf{T} * \begin{bmatrix} v_x^b \\ v_y^b \\ v_z^b \end{bmatrix} \tag{3.3}$$

where the control vector u is composed by the linear velocities in the body reference frame (v_x^b, v_y^b, v_z^b), the angular velocities in the body reference frame (ω_x^b, ω_y^b, ω_z^b) and the drone's orientation (ϕ^b, θ^b, ψ^b). Additionally, $\mathbf{T}$ is the homogeneous transform (see Section 2.2) using the current orientation of the drone: ϕ^b, θ^b and μ_ψ. Notice the third element of the orientation (μ_ψ), this is because the orientation over the Z-axis is estimated by the filter. As result, v^w

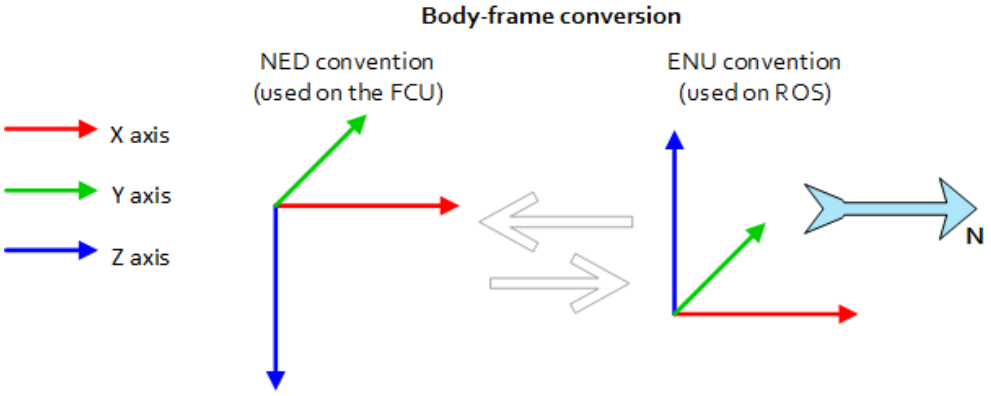


Figure 3.3: NED to ENU conversion scheme. [12]

will be a column vector with the linear velocities in the global reference frame.

Given this, the motion model update can be summarized in the following calculation:

$$\hat{\mu} = \begin{bmatrix} \mu_{t-1,x^w} \\ \mu_{t-1,y^w} \\ \mu_{t-1,z^w} \end{bmatrix} + \Delta t * v^w \tag{3.4}$$

$$\hat{\mu}_\psi = \mu_{t-1,\psi} + \Delta t * \omega_z^b \tag{3.5}$$

Then, $\mathbf{G_t}$, which is the Jacobian matrix of the motion model, should be computed. As mentioned in Section 2.4.1, it has the following characteristic:

$$G_t = \begin{bmatrix} G_r & \mathbf{o} \\ \mathbf{o} & \mathbf{I} \end{bmatrix} \tag{3.6}$$

$$G_r = \begin{bmatrix} 1 & 0 & 0 & G_{r,14} \\ 0 & 1 & 0 & G_{r,24} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

$$G_{r,14} = -\Delta t * (v_y^w * (c_\phi * c_\psi + s_\theta * s_\phi * s_\psi) - v_z^w * (c_\psi * s_\phi - c_\phi * s_\theta * s_\psi) + v_x^w * c_\theta * s_\psi)$$
$$G_{r,24} = -\Delta t * (v_z^w * (c_\psi * s_\phi + c_\phi * s_\theta * c_\psi) - v_y^w * (c_\phi * s_\psi - s_\theta * s_\phi * c_\psi) + v_x^w * c_\theta * s_\psi)$$

where

$$c_\phi = \cos(u_\phi), \quad c_\theta = \cos(u_\theta), \quad c_\psi = \cos(\mu_\psi)$$
$$s_\phi = \sin(u_\phi), \quad s_\theta = \sin(u_\theta), \quad s_\psi = \sin(\mu_\psi)$$

The function g that models the motion of the drone is assumed to be perfect and therefore noise free. So, before computing the covariance update, it is necessary to compute the process noise covariance matrix (R_t) that encodes the motion model noise which, in this case, is related to the underlying dynamics of the drone flight. The noise is assumed to be additive and Gaussian, and therefore, the motion model can be decomposed as:

$$x_t = g(u_t, x_{t-1}) + \mathcal{N}(0, R_t) \tag{3.8}$$

$$R_t = N * U * N^T \tag{3.9}$$

the noise part in equation (3.8) relates to the acceleration component that is, in this case, unknown. However, it is known from a theoretical perspective: the acceleration component in an accelerated movement is $\frac{1}{2}\Delta t^2 a$, where a is the body's acceleration. Given this, we can assume that the noise component in the motion model is:

$$\mathcal{N}(0, R_t) = \frac{1}{2}\Delta t^2 T_w^b \begin{bmatrix} a_x \\ a_y \\ a_z \\ a_\psi \end{bmatrix} \tag{3.10}$$

where T_w^b is the transformation matrix from body to world reference frame. The covariance of the process noise (R_t) can be decompose as shown in equation (3.9), where matrix N is the Jacobian of acceleration term with respect to the state vector, and matrix U is the estimated average acceleration.

$$N = \frac{\partial \frac{1}{2}\Delta t^2 T_w^b \mathbf{a}}{\partial \mu} \tag{3.11}$$

$$U = \mathbf{I} * \begin{bmatrix} a_{avg,x} \\ a_{avg,y} \\ a_{avg,z} \\ a_{avg,\psi} \end{bmatrix} \tag{3.12}$$

The multiplication in equation (3.9) provides an approximate mapping between the motion noise in control space and the motion noise in the state space.

Finally, the covariance update should be computed as follow

$$\hat{\Sigma} = G_t * \Sigma * G_t^T + R_t \tag{3.13}$$

3.3 OBSERVATION MODELS

While the drone is moving around the environment it senses different landmarks that may be included or not in the state vector. These observations will eventually improve the localization of the drone, and will improve the landmarks' poses if needed. The whole process involves the computation of the Jacobian of the observation model for the seen landmark, the computation of the Kalman gain, and the update of the state vector and covariance matrix.

To perform the correction step, EKF-SLAM needs a linearized observation model with additive Gaussian noise. In the case studied in this work, there are two kinds of landmarks and, therefore, two different observation models. The main difference between these two type of landmarks, is that the position of poles type is known, while it is not known in the case of markers. Hence, every time the robot "sees" a pole, the algorithm will update the drone's pose and the known markers' pose; while every time it "sees" a marker two course of action are possible:

a) if the marker is not known, it is added to the state vector, enlarging it along with the covariance matrix.

b) if the marker is known, its pose, the pose of all other markers, and the drone's pose are updated.

The observation model is, as with the motion model in equation (3.8), assumed to be perfect and with an additive Gaussian noise. The noise here is related to the observation process, and so, related to the used sensors.

$$z_i = h_i(x_t) + \mathcal{N}(0, Q_t) \tag{3.14}$$

Consequently, the noise covariance matrix of the observation model cannot be deducted as with the noise covariance matrix of the motion model. In this case, the matrix should be constructed empirically based on the sensors' characteristics, and it has the following characteristic:

$$Q_t = \begin{bmatrix} \sigma_1^2 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \sigma_n^2 \end{bmatrix} \tag{3.15}$$

where the diagonal elements $\sigma_{1..n}$ are the standard deviation of the sensor. Depending on the sensor used, the diagonal elements can be the standard deviation for the range and bearing components (distance, azimuth and elevation), or others.

As shown in Algorithm (2.3) and Algorithm (3.1) several steps are followed during the correction part of the algorithm. After computing the observation model and its Jacobian matrix, the Kalman gain and the innovation should be calculated, and finally, the state vector and covariance matrix updates should be done.

3.3.1 *Observation model for Poles*

In the case of Poles, a range and bearing method is used. In this case, since the poles have a known position, their information is not kept in the state vector and therefore, this information will be used for localization purposes.

A ROS node will publish the range and bearing information every time the drone sees a pole, and this information will be used to calculate the innovation based on the predicted range and bearing. Hence, the observation model used for poles is computed in the following way:

$$\begin{bmatrix} p^b_{i,x} \\ p^b_{i,y} \\ p^b_{i,z} \end{bmatrix} = \boldsymbol{T}_r^{-1} \begin{bmatrix} p^w_{i,x} \\ p^w_{i,y} \\ p^w_{i,z} \end{bmatrix} \tag{3.16}$$

$$h_i(\hat{\mu}_t) = \begin{bmatrix} p_{i,\rho} \\ p_{i,\alpha} \\ p_{i,\beta} \end{bmatrix} = \begin{bmatrix} \sqrt{p^2_{j,x^b} + p^2_{j,y^b}} \\ \operatorname{atan2}\left(p^b_{j,y}, p^b_{j,x}\right) \\ \operatorname{atan2}\left(p^b_{j,z}, p^b_{j,\rho}\right) \end{bmatrix} \tag{3.17}$$

In equation (3.16), $\boldsymbol{T}_r^{-1}$ corresponds to the inverse of the homogeneous transformation matrix with respect to the current drone's pose, and elements p^w_i are the x, y and z coordinates of the i pole's tip in the world reference frame. This way, the global position of the pole i is projected to the body reference frame. After that, the range (ρ), azimuth angle (α) and elevation angle (β) are calculated, as shown in equation (3.17). In Figure 3.4 an example of the range and bearing is shown.

Observing a pole affects only the drone's pose, and therefore, the Jacobian matrix of the observation model will have the following form:

$$H_i = \begin{bmatrix} \frac{\partial \rho'}{\partial \mu_x} & \frac{\partial \rho'}{\partial \mu_y} & \frac{\partial \rho'}{\partial \mu_z} & \frac{\partial \rho'}{\partial \mu_\psi} & \dots & 0 \\ \frac{\partial \alpha'}{\partial \mu_x} & \frac{\partial \alpha'}{\partial \mu_y} & \frac{\partial \alpha'}{\partial \mu_z} & \frac{\partial \alpha'}{\partial \mu_\psi} & \dots & 0 \\ \frac{\partial \beta'}{\partial \mu_x} & \frac{\partial \beta'}{\partial \mu_y} & \frac{\partial \beta'}{\partial \mu_z} & \frac{\partial \beta'}{\partial \mu_\psi} & \dots & 0 \end{bmatrix} \tag{3.18}$$

where ρ' corresponds to the distance part of the observation model, α' is the azimuth part, and β' the elevation part. The elements after the 4^{th} column are all 0, which means, as said before, that the observation of a pole will not affect the pose of the markers.

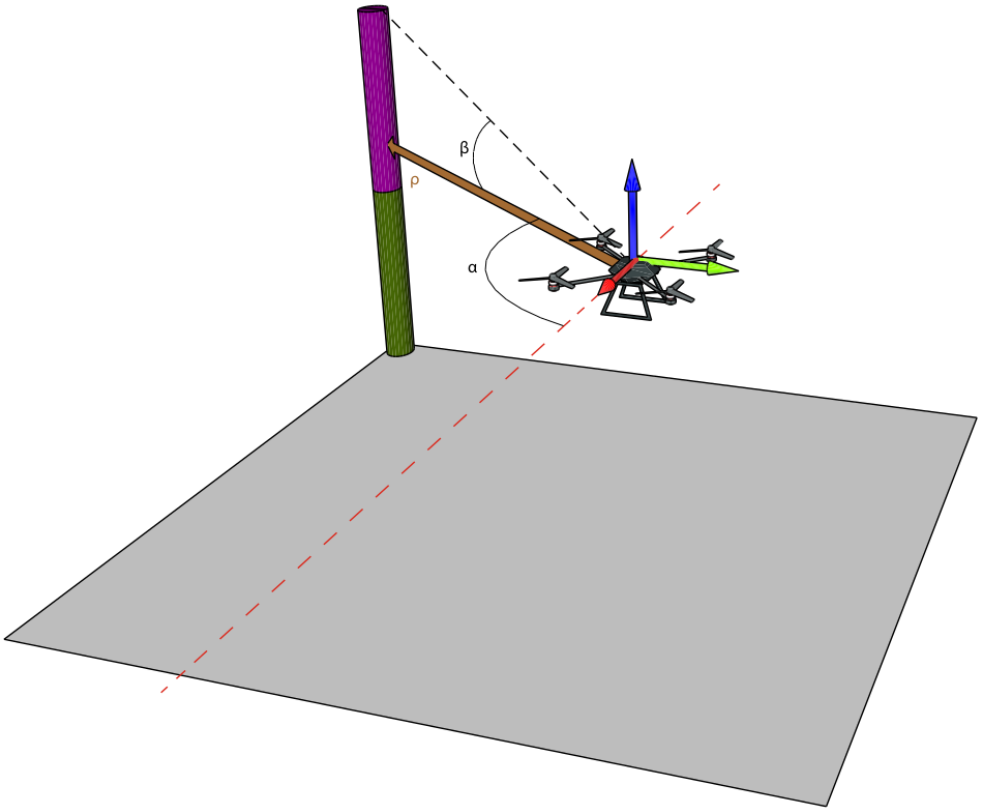


Figure 3.4: Range and bearing example. The drone observes a pole, process the data from the sensors and estimates the distance (ρ), the elevation angle (β), and the azimuth angle (α). The elevation angle is calculated based on the top extreme of the pole.

3.3.2 *Observation model for Markers*

The observation model for the markers is a bit different. In this case a ROS node is responsible of detecting, tracking and publishing the pose of the markers with respect to the camera that has seen it. The ROS package responsible of this process is called `visp_auto_tracker`, and publishes messages of type `PoseStamped`, which provides the position and orientation of the seen marker. An example of this situation can be seen in Figure 3.5.

Every time the camera that points down sees a known marker, the `visp_auto_tracker` node will publish the pose of that marker. This published pose is with respect to the camera which is not positioned at the drone's center of mass and so, a different transformation is needed. This process can be seen in equation (3.19),

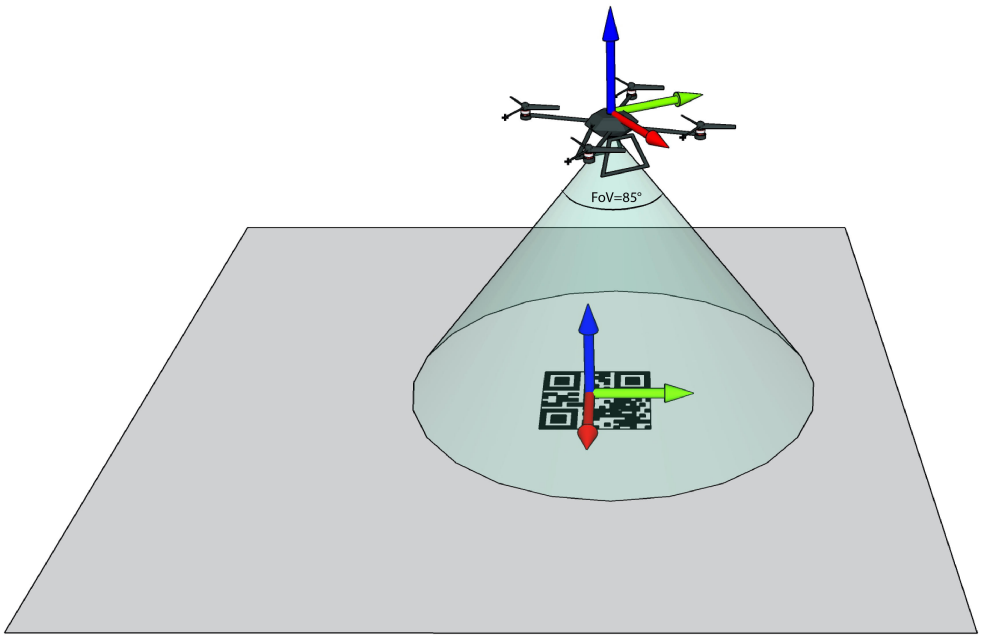


Figure 3.5: Example of the drone observing a marker. The camera visualize a marker and the node `visp_auto_tracker` estimates its pose with respect to the camera reference frame.

in which the transformation of the marker's pose in the world reference frame is transformed to the pose in the camera reference frame.

$$\begin{bmatrix} m^c_{i,x} \\ m^c_{i,y} \\ m^c_{i,z} \\ m^c_{i,\phi} \\ m^c_{i,\theta} \\ m^c_{i,\psi} \end{bmatrix} = (\boldsymbol{T_r} * \boldsymbol{T_c})^{-1} * \boldsymbol{T_m} \tag{3.19}$$

where, $\boldsymbol{T_r}$ is the homogeneous transformation matrix of the drone's pose, $\boldsymbol{T_c}$ is the homogeneous transformation matrix of the camera's pose, and $\boldsymbol{T_m}$ is the homogeneous transformation of the marker's pose. The result is a vector that contains the marker's pose in the camera reference frame.

As mentioned before, observing a marker will update the drone's pose and that marker's pose, and therefore the Jacobian matrix of the observation model will have a different aspect from the pole's case. Here, the Jacobian can be split in two parts as shown in equation (3.20), where the left part, as

with the poles, affects the drone's pose, while the right part of the matrix affects the seen marker's pose.

$$H_i = \begin{bmatrix} \frac{\partial h_i(\hat{\mu})}{\partial \mu_r} & \dots 0 \dots & \frac{\partial h_i(\hat{\mu})}{\partial \mu_{m_i}} & \dots 0 \dots \end{bmatrix} \tag{3.20}$$

The Jacobian matrices of the observation model with respect to the drone state and with respect the marker pose can be seen in [13]. It is worth to mention that the reason why this matrix only contains two sections different to zero is because of the relations between markers: observing a marker does not affect the pose of others, and therefore, the Jacobians of the observation model for marker i with respect to marker $i+1$ or any other marker is 0.

Furthermore, unlike the case of poles, the markers' pose is unknown the first time, and therefore the algorithm will introduce their pose into the state vector when the drone sees a previously unknown marker.

3.3.2.1 *Adding new Markers*

As mentioned in Section 2.4.2, to add new landmarks to the state vector an inverse observation model is needed. In the current case, the inverse observation model will project the observed marker's pose from the camera reference frame to the world reference frame. Equation 3.21 shows the inverse observation model for markers where, similarly to the observation model, a set of transformations are performed but, this time, in order to obtain the pose of the seen marker in the world reference frame:

$$\begin{bmatrix} m^w_{i,x} \\ m^w_{i,y} \\ m^w_{i,z} \\ m^w_{i,\phi} \\ m^w_{i,\theta} \\ m^w_{i,\psi} \end{bmatrix} = \boldsymbol{T_r} * \boldsymbol{T_c} * \boldsymbol{T_m} \tag{3.21}$$

As before, $\boldsymbol{T_r}$ is the homogeneous transformation matrix of the drone's pose, $\boldsymbol{T_c}$ is the homogeneous transformation matrix of the camera's pose, and $\boldsymbol{T_m}$ is the homogeneous transformation of the marker's pose. The result is a vector containing the marker's pose in the world reference frame, and with which will extend the state vector:

$$\mu_t = \left[\begin{array}{c|cccccc} \mu_t & m^w_{i,x} & m^w_{i,y} & m^w_{i,z} & m^w_{i,\phi} & m^w_{i,\theta} & m^w_{i,\psi} \end{array} \right]^T$$

Furthermore, the covariance matrix should be updated in order to contain the newly added marker. As shown in equation 2.15, the Jacobian of the

inverse observation model with respect to the drone's state and the Jacobian of the inverse observation model with respect to the marker's state are needed. Both Jacobian matrices can be seen in Appendix A. In this case, the matrix $\boldsymbol{Y_x}$ has a size of $(|\mu| + 6) \times (|\mu|)$ where $|\mu|$ is the size of the state vector before adding the projected pose of the seen marker. On the other hand, the matrix $\boldsymbol{Y_z}$ has a size of $(|\mu| + 6) \times 6$. Given these two matrices, the new covariance matrix Σ will be increased by 6 columns and 6 rows and will have the following form:

$$\Sigma = \begin{bmatrix} \Sigma_{r,r} & \Sigma_{r,m} \\ \Sigma_{r,m} & \Sigma_{m,m} \end{bmatrix}$$

where $\Sigma_{r,r}$ is the covariance matrix between the drone's state variables, the $\Sigma_{r,m}$ and $\Sigma_{m,r}$ is the covariance matrix between the drone and the markers' state variables, and between the markers and the drone's state variables, and $\Sigma_{m,m}$ is the covariance matrix between markers' state variables.

3.3.3 *Observation model for range sensor and height correction*

With respect to the height estimation, it is corrected by poles, markers and with a combination of PIX4Flow camera and Octomap. Both poles and markers update the altitude of the drone, but this estimation can be improved by using the optical flow camera and the Octomap generated by the stereo camera.

The Octomap is build with obstacle avoidance in mind, but it can be used jointly with the optical flow to correct the drone's Z position. To do so, the it can be assumed that the range sensor provided by the PIX4Flow camera, plus the Octomap voxel height below the camera should be equal to the drone's height. Hence, the range sensor model can be defined as:

$$h(\hat{\mu}_t) = \hat{\mu}_z + bias \tag{3.22}$$

and the observation is defined by:

$$\hat{z} = voxel_z + range_{distance} \tag{3.23}$$

where $voxel_z$ is the Z position of the voxel below the drone, and $range_{distance}$ is the sensed distance provided by the range sensor.

Hopefully, the Jacobian is a vector composed by zeroes except for the component related to the Z position of the drone:

$$H = \begin{bmatrix} 0 & 0 & 1 & \dots 0 \dots \end{bmatrix} \tag{3.24}$$

Since the observation model is composed by a single variable, the Jacobian is a row vector of the size of the current state vector.

3.4 VISUAL PROCESS PIPELINE

As mentioned in Section 3.1, the drone is composed by several cameras, specifically, two stereo cameras and four monocular cameras. These cameras are used to identify and avoid obstacles and for localization purposes. The localization process involves the identification of poles and markers, and therefore, the usage of the before mentioned cameras: the four monocular cameras are used to identify poles, while one stereo camera is used to identify the markers.

3.4.1 *Visual process for Poles*

Four ROS nodes are responsible of determining whether a pole is being seen or not. To do so, they subscribe to topics related to the camera information and the camera image, and every time a new image arrives, each node process it to identify a pole. Each pole is composed by two different colors, and each pole has a different combination of colors making possible to uniquely identify them. The overall process can be followed in Algorithm (3.2).

Algorithm 3.2: Poles localization algorithm

Input: `image_raw`, `camera_info`

```
1  Apply masks to image
2  Identify how many poles are in the image
3  if contours of poles where found then
4      foreach pole contour do
5          Find coordinates of contour
6          Identify pole
7          if the full width of the pole can be seen then
8              Compute Azimuth (α)
9              if the pole is not occluded then
10                 Compute Range (ρ)
11                 if the top of the pole can be seen then
12                     Compute Altitude (β)
13 return α, β, ρ
```

Lines 1 and 2, applies a mask in order to identify the amount of poles contours in the image. If there are pole contours in the image, at lines 5 and 6 the algorithm uniquely identifies the pole and its coordinates. If the width of the identified pole can be seen, at line 8 the algorithm computes the azimuth

angle (α) by using the drone's yaw and the center of the contour. At line 9, it estimates the probability of occlusion of the pole by comparing the contour shape with the form factor, computed as $\frac{y_{max}-y_{min}}{x_{max}-x_{min}}$ where y_{max} and y_{min} are the bottom and the top coordinates of the contour, and x_{max} and x_{min} are the right and left coordinates of the contour. If the probability is greater that 75%, the range is computed based on the diameter of the contour. Finally, if the top of the contour can be seen (line 11), at line 12 it computes the altitude.

3.4.2 *Visual process for Markers*

The markers detection pipeline is performed by the Visual Servoing Platform (ViSP) tracking package [14]. The package wraps an automated pattern-based tracker based on ViSP library, which allows to estimate an object pose with respect to the camera. In the case of this work a QR code is used as marker, which is identified by the tracker.

The `visp_auto_tracker` node subscribes to the camera image and information topics, and publishes the detected object's pose. The camera messages are provided by the stereo camera that points downwards.

3.5 OVERALL ARCHITECTURE

The system is composed by several ROS nodes that specialize and perform a wide range of operations. However, what is most interesting for this work is the node that is in charge of the EKF-SLAM process.

In Figure 3.6 the components that interact around the EKF Localization node can be seen. In the center of the figure the `EKFSLAM` component is placed, which is responsible of running the EKF-SLAM algorithm. Every time an `Odometry` message arrives, the prediction step takes place, and this happens every 30Hz. Furthermore, every time a `RangeAndBearingPoles` or a `QRCodeStamped` arrives the correction step takes place, and differently to the prediction step, this depends on whether the drone observes or not a pole or a marker. Additionally, the `EKFSLAM` component provides three services:

1. `get_drone_state`: takes no arguments, and returns the current localization of the drone `Pose` message.
2. `get_marker_state`: takes as argument the id of a marker, and returns the current estimated pose of that marker`Pose` message.

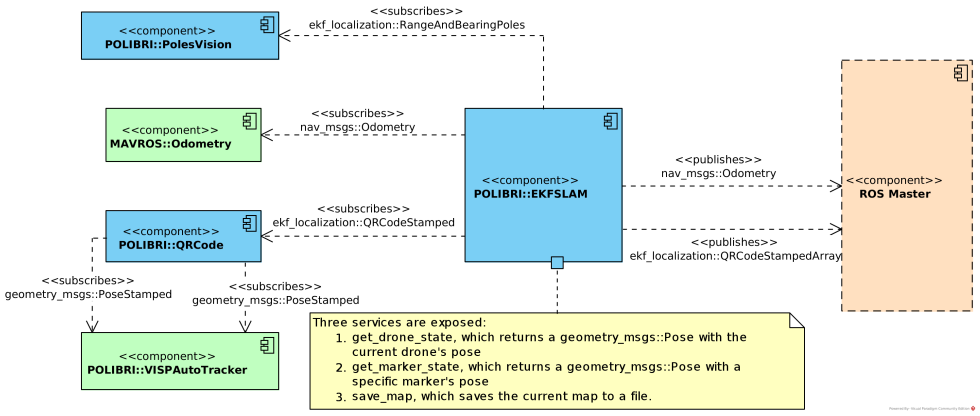


Figure 3.6: Components diagram of the EKF Localization node. The `EKFSLAM` class subscribes to different messages, between them the `Odometry` messages are used as control variables, while the `RangeAndBearingPole` and `QRCodeStamped` messages are used every time the drone observes a pole or a marker. Moreover, `EKFSLAM` class publishes two types of messages: `Odometry` which provides the filtered localization and `QRCodeStampedArray` which contains a list markers' poses.

3. `save_map`: takes no arguments, and it saves the current map in a YAML file. The map contains the pose of all the poles and all the markers seen so far.

It is worth to mention that the messages of type `QRCodeStamped` are provided by the `QRCode` node which is internal to the `ekf_localization` package. This node is needed because the `visp_auto_tracker`, responsible of identifying the markers and publish their poses, publish these data as separate messages: the marker's id as a `String` message and the pose as a `PoseStamped` message. Due to this situation, the `QRCode` node is responsible of match both data together and publish an `QRCodeStamped` message which contains the marker's identifier and its pose.

The main element of the EKF-SLAM node is the `EKFSLAM` class which is responsible of keeping track of the state vector and execute the EKF-SLAM algorithm. This class inherit from an abstract class called `EKF`, which is responsible of the implementation of the algorithm in a generic way, while the specifics for the current problem is contained in its child.

There are several components that interact within `EKFSLAM` class, but probably the most interesting one is the `MapManager` class. This class is responsible of maintaining an updated map of the environment with all the poles and markers, and to save and retrieve the map from a YAML file.

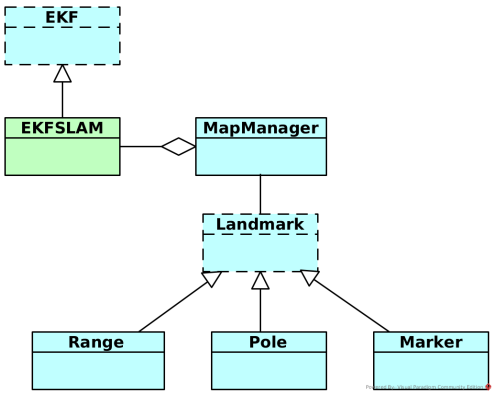


Figure 3.7: Class diagram of the EKFSLAM node. The `EKFSLAM` class is composed by a `MapManager` instance, which is composed by a list of `Landmarks`. The `Landmark` class is abstract and its concrete classes are `Range`, `Pole` and `Marker`.

The `MapManager` class contains a list of `Landmarks`, and every time the state vector is updated in `EKFSLAM` class, the `MapManager` class updates the information about each `Marker`. As mentioned before, the pose of each pole is known and needs no update. Furthermore, the `Landmark` class is specialized in a `Marker` class, a `Pole` class and a `Range` class, each of which is responsible of provide the observation model and the needed Jacobian matrices for the algorithm associated to a specific type of landmark or observation, hiding the implementation to the `EKFSLAM` class.

3.5.1 *ROS nodes*

Figure 3.8 shows the detail for the localization node, `/ekf_localization_node`, with all its subscriptions. It is worth to mention the `/qr_code_node`, which publishes the markers position with its identifier, subscribes to `/visp_auto_tracker /code_message` and `/visp_auto_tracker/object_position`, and publishes into `/visp_auto_tracker/stamped_object_position` topic.

Regarding the ROS nodes that comprise the system, in Figure 3.9 all the ones that interact with the localization node and all the others are shown. The EKF-SLAM node, and as mentioned before, interacts with `/mavros` node, all the poles identification nodes (`/poles_vision_#`), the `/qr_code_node`, and the `/rtabmap` node that provides the Octomap messages. The message passing is represented by arrows, where an incoming arrow means that a node subscribes to the message topic, while an outgoing arrow means that the node publishes that message topic. Arrows go from a node to a topic or from a topic to a node, but do not go from node to node. Its reason lies on the fact that nodes interact between each other using a message passing interface as mention in Section 2.1. Furthermore, it is worth to mention that

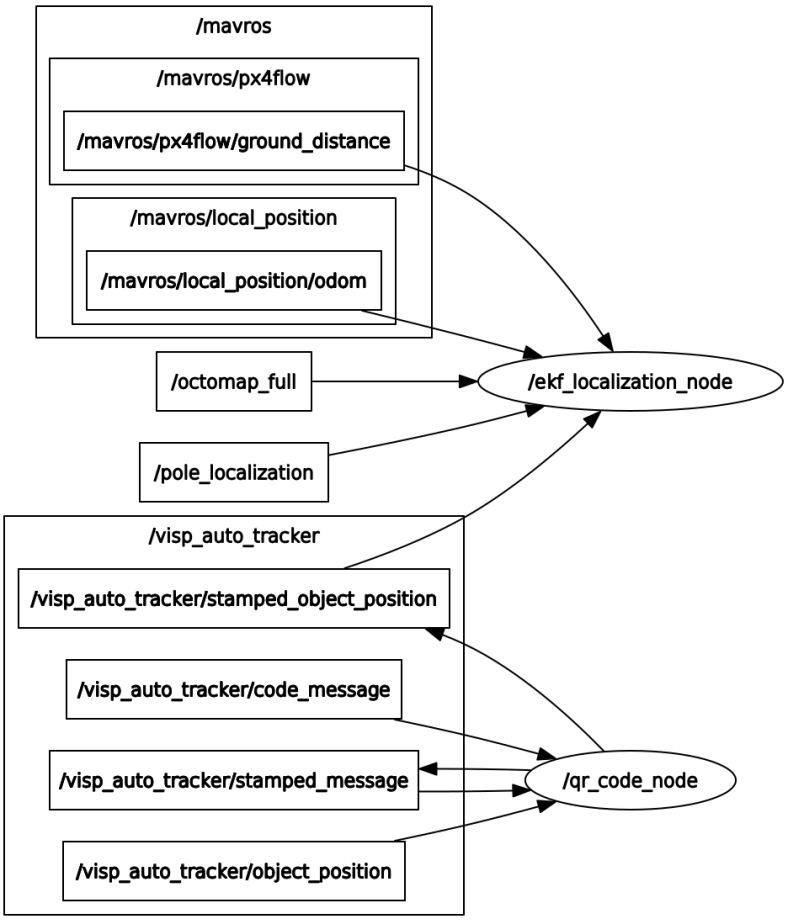


Figure 3.8: Detail of the EKF-SLAM node interactions. Beside the `/ekf_localization_node`, the `/qr_node_code` can be seen. This node, is responsible of subscribing the `/visp_auto_tracker` messages in order to, then, publish the identifier along with the pose of the marker that has been seen. On the other hand, the `/ekf_localization_node` subscribes to the `/mavros`, `/octomap`, `/pole_localization` and `/visp_auto_tracker` messages.

in Figure 3.9 a `/gazebo` node is present. This node appears only on simulation and represents the simulated environment, that is why it publishes all the cameras and stereo cameras information.

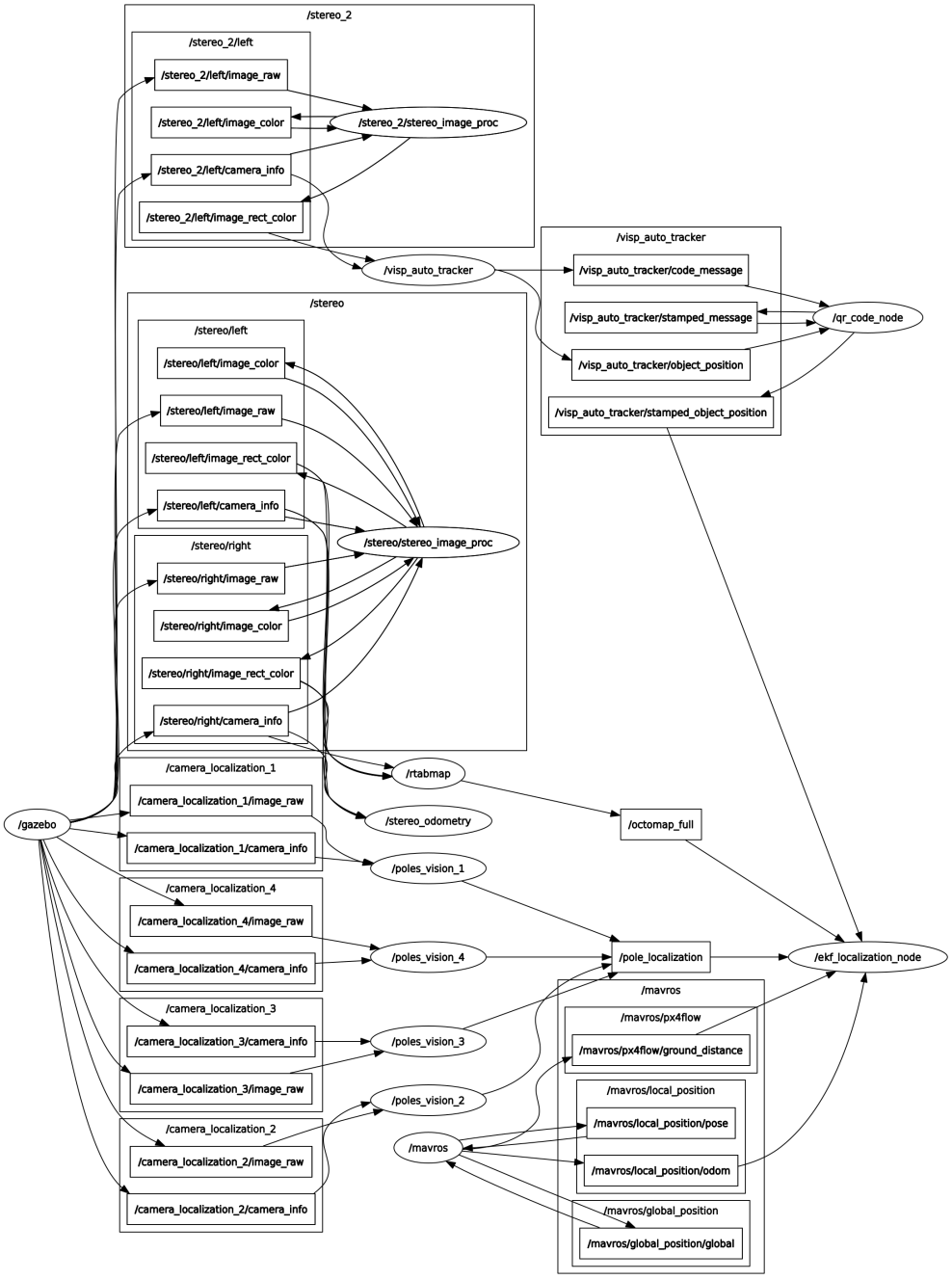


Figure 3.9: ROS graph of the system. Nodes are depicted as ellipses, while message topics are depicted as boxes. Each arrow means that a node publishes or subscribes to a specific message topic.

4

EXPERIMENTAL RESULTS

In order to evaluate the algorithm proposed in Chapter 3, in the context of the Leonardo Drone Contest, several experiments where performed. The aim of these experiments were to evaluate:

- the importance of known poles and known and unknown markers in the localization and mapping process
- the accuracy of markers' pose estimation
- the importance of the range and Octomap measurements in the height correction
- the importance of the NEES test to evaluate measurements and the filter's consistency

The whole process was tested using ROS bags in order to replicate an experiment using different configurations.

An important thing to mention is that the ground truth is provided by a Gazebo plug-in, only in a simulated environment. Odometry only, without correction of any kind, is provided as control process and used to compare the results.

Figure 4.1 shows the representation of the odometry used for control purposes, as well as the MAVROS estimate and the ground truth. The figure shows the drift in the MAVROS estimate and in the EKF-SLAM estimate. The yaw estimate follows the ground truth with almost no drift in both MAVROS and EKF-SLAM estimators, while the drone's height does not follow the ground truth at all in the EKF-SLAM process.

It is worth to mention that the negative sign in the Z position of the EKF-SLAM process is due to negative velocities in the control signal. This is not something to be concerned about and it is the expected behavior given the linear velocities provided by MAVROS. The positive velocity while the drone takes off is not hold during enough time to position the Z prediction near

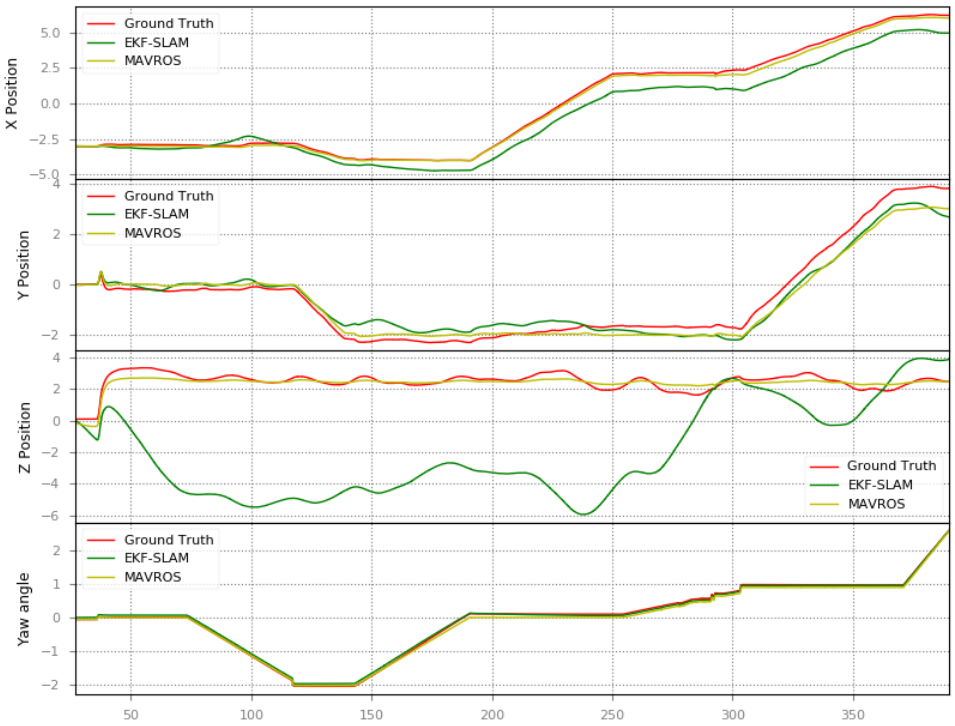


Figure 4.1: Odometry only plot. It can be seen three different lines: the red one is the ground truth, the yellow one is the MAVROS estimate and the green one is the EKF-SLAM motion model presented in this work.

the ground truth. Furthermore, the linear velocity control signal for the first 25 seconds is negative, which makes the drone "flight below ground". The take off starts at the second 25 and pushes the drone over the ground for some seconds, while the linear velocity published by MAVROS oscillates in between positive and negative values, being the last ones more often. This does not happen with the MAVROS estimator, because it has an own filter that produces the Z position estimation. Hence, the comparison in this case is worthless. The MAVROS EKF filter has nothing to do with the motion model in the EKF-SLAM algorithm which does not correct the prediction in any sense.

Less evident is the position estimate in X and Y, where the MAVROS estimation is near the ground truth despite it drifts over time. The EKF-SLAM motion model drifts in a more evident way, both in X and Y: it starts in the same position as the ground truth, and it constantly moves away the control signal. However, and unlike the Z position, both MAVROS and EKF-SLAM motion model follow the ground truth position. Again, comparing the MAVROS estimate with the EKF-SLAM prediction is worthless as one is filtered, while the other is composed by prediction-only.

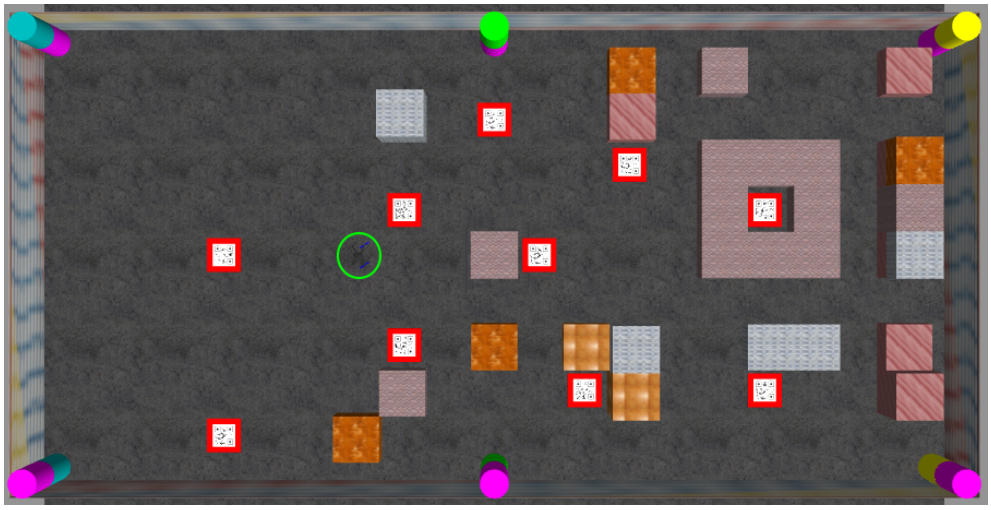


Figure 4.2: Gazebo simulated environment.

4.1 THE ENVIRONMENT

The simulated environment can be seen in Figure 4.2, where all the obstacles, poles and markers are disposed. Moreover, the walls define the limits of the environment, making it impossible to go off the limits. The markers were disposed arbitrarily around the world, while the poles are in the same place as they should be in the competition: one in each corner, and two along the middle axis of the space. The drone is placed at coordinates $(-3; 0)$, and can be depicted in the figure as the green circle in the middle left. As mentioned before, the poles have a different combination of colors, where two poles do not have the same combination. The markers, are QR codes framed in red, while obstacles are brown and gray boxes around the environment. The walls are made of a net-like material distinguishable from the obstacles, while the floor is a pavement-like material.

4.2 SIMULATED EXPERIMENTS

The following experiments were done using a simulated environment. The system was run and a ROS bag was recorded in order to analyze different aspects of the environment using the same simulation, and therefore, being able to compare between different configurations. Noise tuning was done empirically using the ROS bag in order to obtain the best possible results, while taking into consideration the fact that noise values have a physical meaning.

Four sets of experiments with different objectives were conducted:

a) **Experiments A**: These experiments aim to show the importance of poles in the localization process.

b) **Experiments B**: These experiments aim to show the importance of markers in the localization and mapping process.

c) **Experiments C**: These experiments aim to show the importance of the range sensor in the estimation of the drone's height.

d) **Experiments D**: These experiments aim to understand the importance of the NEES test in the acceptance of the observations.

All these experiments were carried out using the same bag, unless the contrary is mentioned.

4.2.1 *Experiments A: The importance of poles*

The experiments presented in this section were done with the objective of understanding the importance of poles in the localization of the drone in the environment. Two experiments were performed: the first experiment was carried out with known-noise observations of poles, while the second experiment was carried out using the real measurements taken from the ROS bag.

4.2.1.1 *Procedures*

As mentioned, the first experiment was performed with known-noise observations of poles, with the objective of showing the perfect localization compared with the MAVROS localization and the ground truth. On the other hand, the second experiment was performed with the real observations taken from the ROS bag, in order to understand if the proposed implementation works as well as with the perfect observations of the previous experiment.

In these experiments the same ROS bag is used, and the measurements are taken during approximately 360 seconds. During the path, the drone visits few markers and pass over some obstacles, both of which are not taken into account in the localization process.

4.2.1.2 *Results*

The results in these experiments are shown in Figure 4.3 for known-noise pole observations and in Figure 4.4 for real poles observations. With respect to the first experiment, it can be said that the correction is almost perfect as the path followed by the drone follows the ground truth all the time. However, and as it is expected, in the second experiment the EKF-SLAM estimation

follows the ground truth but with more noise than in the previous one. There are some moments in which the estimation is more noisy than usual: this has to do with occlusions and far-from-true measurements.

4.2.2 *Experiments B: The importance of markers*

Differently from poles, markers are sporadically seen, and therefore, it can be assumed that its contribution to the localization will be sparse. On the other hand, one can imagine that using poles and markers for localization-only purposes will improve the estimation seen in Figure 4.4.

A set of different experiments were carried out in this section, and all of them aim to understand the importance of markers during the localization. On the other hand, a subset of these experiments try to measure the mapping process and the mapping process jointly with the localization.

4.2.2.1 *Procedures*

The experiments presented in this section were carried out using the same ROS bag. The difference between them lies on the usage or not of real markers' observations, and lies on the usage or not of a previously built map of markers.

The experiments conducted are the followings:

1. The first experiment is similar to the ones presented in Section 4.2.1. The drone follows the path and estimates its pose based on markers only.
2. The second experiment is similar to the previous one, but it uses markers and poles to localize. However, the main difference with the previous experiments lies on the absence or not of a perfect map. This means that when the map is not available, the algorithm will create it, while if the map is available, the algorithm will use it to update the drone's pose.
3. Finally, the third experiment measures the distance between the true position of markers and the one estimated by the algorithm in a context of real observations, both for markers and poles.

4.2.2.2 *Results*

The first set of experiments related to the markers, aim to identify the marker's importance in the localization problem. Two plots are presented to this purpose, and are shown in Figure 4.5 and Figure 4.6. Figure 4.5 shows

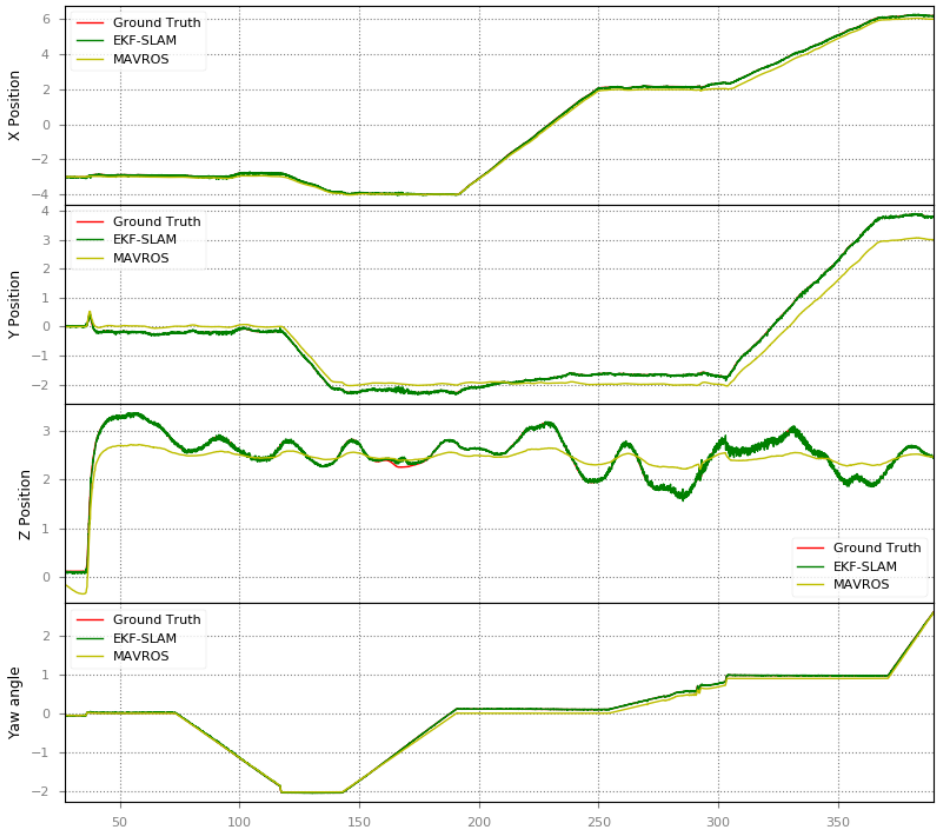


Figure 4.3: Localization with perfectly known noise of poles.

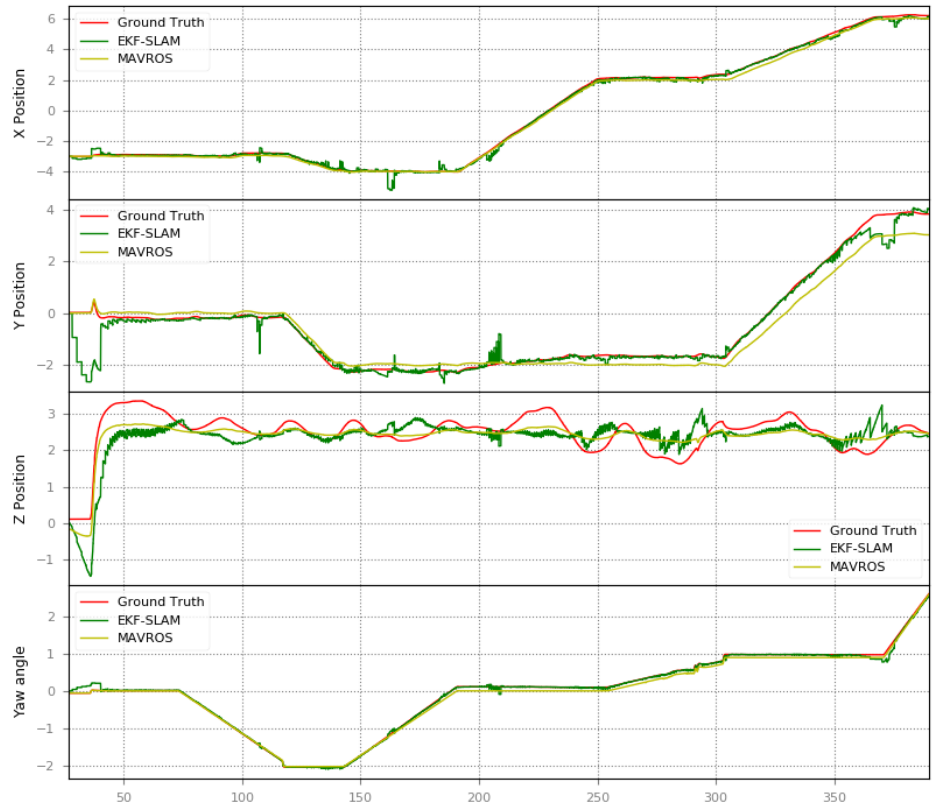


Figure 4.4: Localization using real poles' observations.

the path followed by the drone while using only the perfect observations of markers and the true map, while Figure 4.6 shows the path followed while using real observations of markers with the true map.

It can be seen in both figures that, even if the path taken by the drone does not follow perfectly the ground truth, it accommodates when it sees a marker. This behavior can be appreciated between seconds 180 and 220, and can be seen in Figure 4.7.

The second set of experiments was designed to understand the importance of markers and poles in the localization problem. In this case, the drone follows the path and corrects its position using known poles and unknown markers. As can be seen in Figure 4.8 and Figure 4.9, the drone follows the path almost perfectly, and gets lost in the periods where it does not see any marker. However, as the algorithm uses both poles and markers to update the drone's pose, it does not diverge when no marker is seen, making the estimation more robust in these situations. The difference between Figure 4.8 and Figure 4.9 is that, in the first one perfect markers are seen, while in the second one real markers are seen. This is particularly evident between seconds 130 and 150, where accepted observations that are far from the true marker position, make the filter to diverge from the ground truth. Insight into this problem will be discussed later on in this chapter.

Something worth understanding is how different is the localization process when the map is available, and when it is not. So far, the experiments shown were developed using a map with the true pose of markers. However, this is not always possible, and since SLAM is a two step problem (mapping and localization), a map needs to be built first. In the Figure 4.10 it can be seen the state estimation when no map is available, and real observations of both poles and markers are used. When comparing Figure 4.10 with Figure 4.9 no big differences can be appreciated.

Finally, the last set of experiments measure the distance between the true markers' pose and the one estimated by the algorithm. The Euclidean distance (in meters) is calculated for the position of the markers, and the orientation is shown as the relative rotation, needed to go from the estimated one to the true one, expressed as ϕ, θ and ψ (in radians). The Euclidean distance is defined as

$$D = \sqrt{(x - \hat{x})^2 + (y - \hat{y})^2 + (z - \hat{z})^2}$$

The results for those markers seen during the drone's path are presented in Table 4.1. The estimated pose and the ground truth can be seen in the Table 4.2. It is worth mentioning that all the markers' observations have an acceptance greater than 60%:

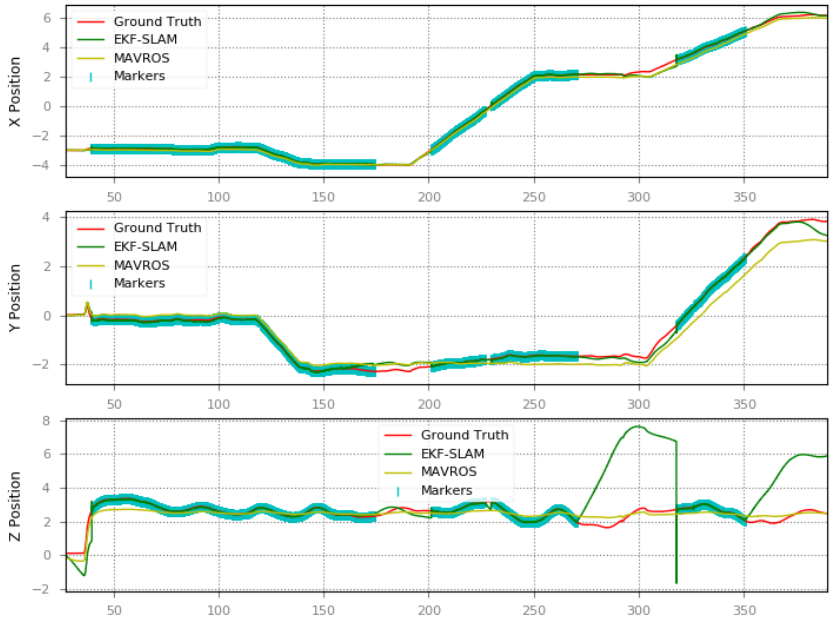


Figure 4.5: Localization with perfect marker observations

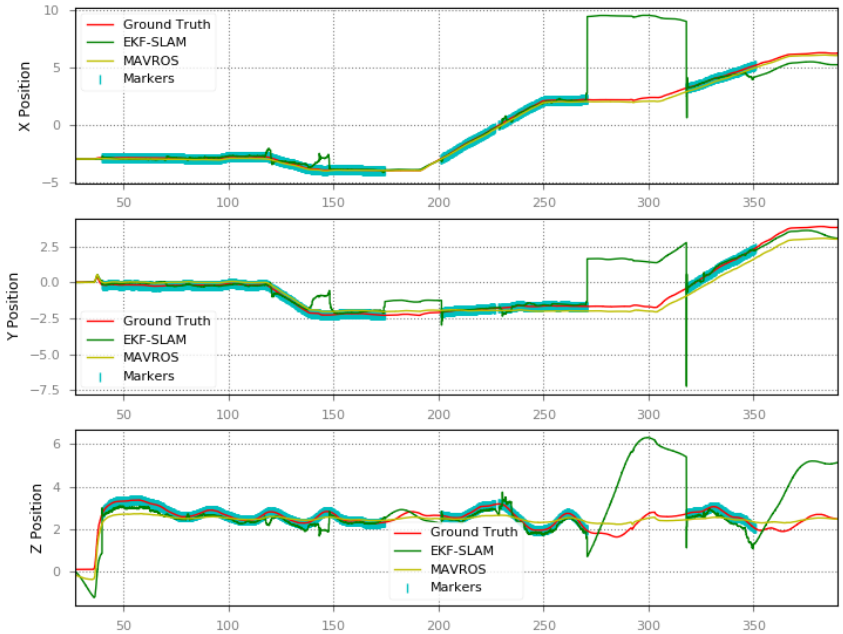


Figure 4.6: Localization with real marker observations

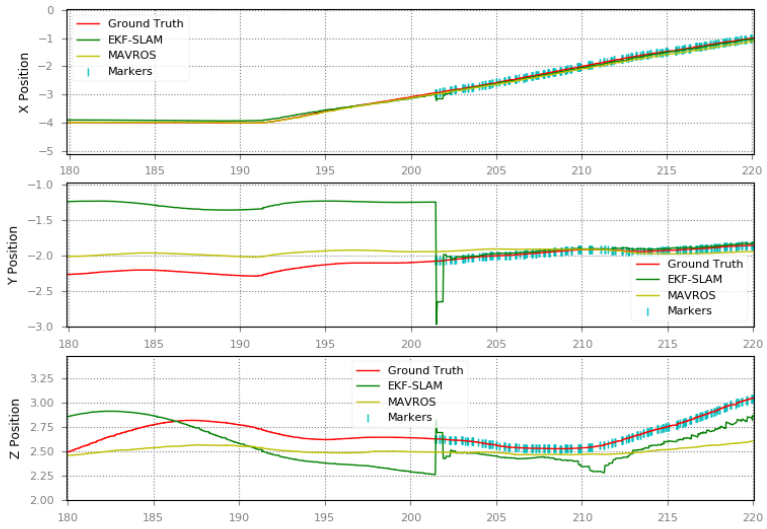


Figure 4.7: Detail of correction process in the case of markers-only localization. As soon as the drone visualizes a marker, it corrects its pose according to the observation.

- **Marker 0**: 69.36%
- **Marker 1**: 65.68%
- **Marker 4**: 70.21%
- **Marker 5**: 67.94%

Marker ID	Euclidean distance	ϕ	θ	ψ
0	0.076	0.089	0.142	0.034
1	0.186	0.407	0.145	0.052
4	0.119	0.153	0.132	0.055
5	0.155	0.024	0.057	0.013

Table 4.1: Distance to between true markers' pose and the pose estimated by the EKF-SLAM algorithm.

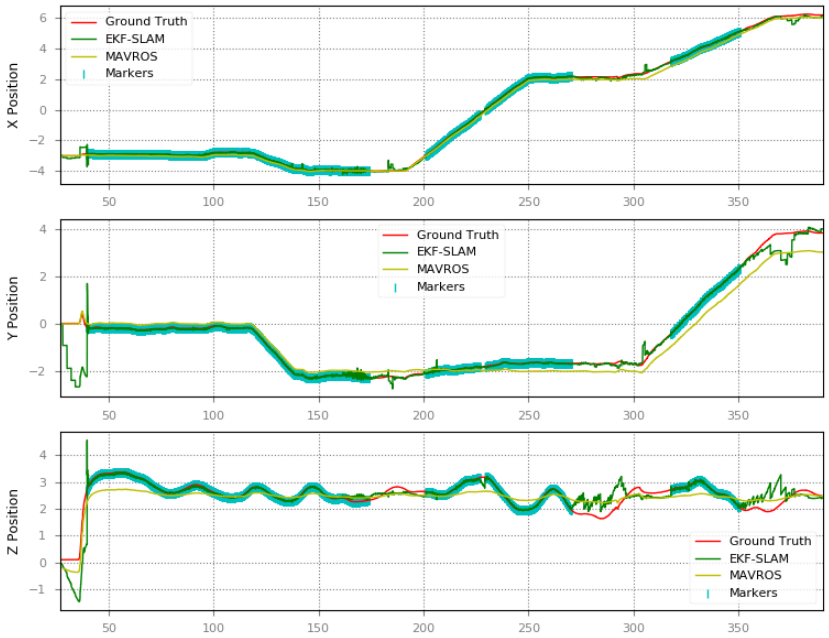


Figure 4.8: Localization using real pole observations and perfect marker observations.

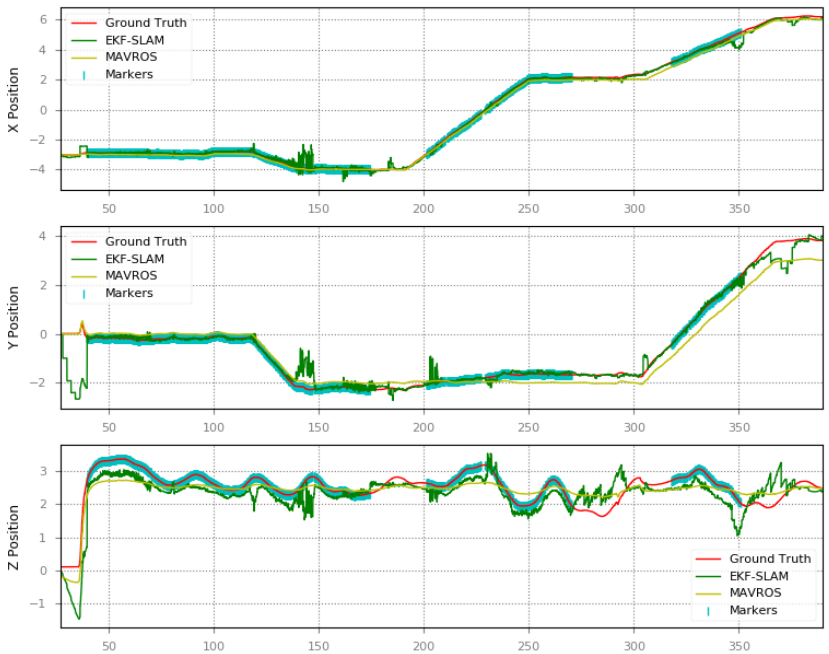


Figure 4.9: Localization using real pole and real marker observations.

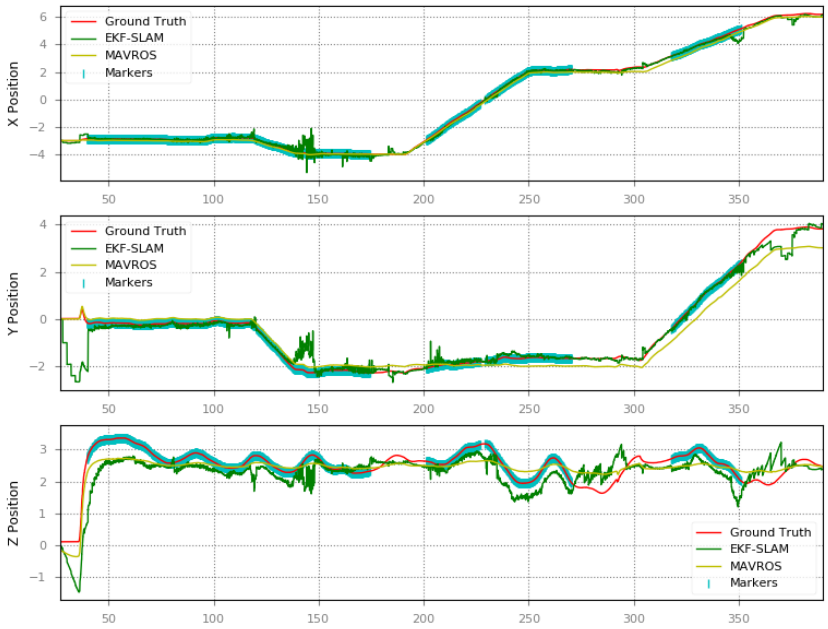


Figure 4.10: Localization and mapping using real pole and marker observations.

Marker ID	X	Y	Z	ϕ	θ	ψ
0	-2.0	1.0	0.1	0.0	0.0	-3.142
	-2.005	0.948	0.044	-0.088	-0.142	-3.108
1	3.0	2.0	0.1	0.0	0.0	3.142
	3.138	1.980	0.223	-0.407	0.145	3.089
4	-2.0	-2.0	0.1	0.0	0.0	-3.142
	-2.070	-1.912	0.062	0.153	0.132	-3.087
5	1.0	0.0	0.1	0.0	0.0	3.142
	1.040	-0.053	-0.040	-0.024	0.057	3.128

Table 4.2: Marker's estimated pose (blue) and marker's real pose (green) comparison.

4.2.3 *Experiments C: The height estimation*

The height estimation of the drone is composed by two types of observations: the Octomap generated based on the data provided by the front stereo camera, and the range sensor measurements provided by the Pix4Flow device. The proposed implementation takes care of this, with the solely condition that the Octomap is complete at the current coordinates of the drone, and in that case, the measurement is accepted and the correction step is triggered.

The experiment conducted in this section shows the importance of the height correction using both the Octomap and the range sensor measurements, but also, the importance of a well defined Octomap in order to avoid false positives.

4.2.3.1 *Procedures*

As with the previous experiments, this one was performed using a ROS bag with the range sensor and Octomap information. The drone follows the same path as before, but this time using both measurements to correct its height.

4.2.3.2 *Results*

The path followed by the robot can be seen in Figure 4.11. The plots are similar to those in Figure 4.9, however the height estimation is different from around second 250. After this second, the Octomap information is available and the correction step in the EKF-SLAM algorithm is triggered.

Figure 4.12 shows the same plot as before, with the aggregation of the range sensor information. It can be seen that the drone flew over few obstacles, and the range sensor outputs heights below the ground truth. Accepted and discarded measurements can be appreciated in detail in Figure 4.13, and some interesting conclusions can be deducted.

First of all, having an incomplete or noisy Octomap makes the altitude estimation to reach wrong values, as can be seen between seconds 315 to 320. During this 5 seconds, the Octomap returns values near 2.5 meters for the voxels below the drone while the range sensor returns similar values, this makes the algorithm to estimate the Z position to up to 4 meters.

On the other hand, when the Octomap returns values that are near the ground truth and the drone is over an object, the estimate approaches the ground truth. This can be seen between seconds 295 to 305, where the accepted measurements make the EKF-SLAM algorithm to follow the ground truth.

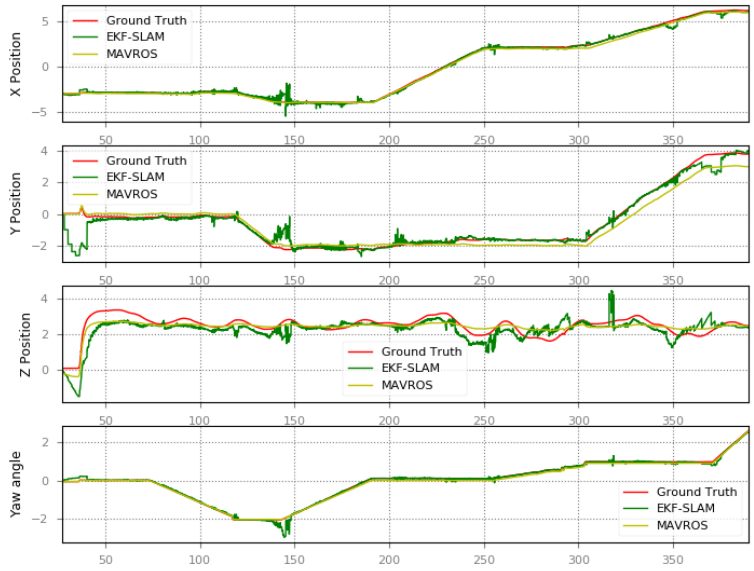


Figure 4.11: Path followed by the drone when correction of height is used.

4.2.4 *Experiments D: The importance of NEES test*

Explained in Section 2.5, the NEES test helps to the consistency of the filter. In the proposed implementation, the NEES test helps to discard measurements too far from the estimated ones, and to accept those that are close to the expected.

The experiments in this section are divided in two, and try to shed some light over the following:

1. The importance of using (or not) the NEES test
2. The importance of the χ^2 value

4.2.4.1 *Procedures*

Two experiments were conducted using the same ROS bag as before, using the real observations for poles and markers and using the measurements of the range sensor for the height estimation. Moreover, the path was followed using localization only with a perfect map. The difference between these two experiments lies on the usage or not of the NEES test to discard observations.

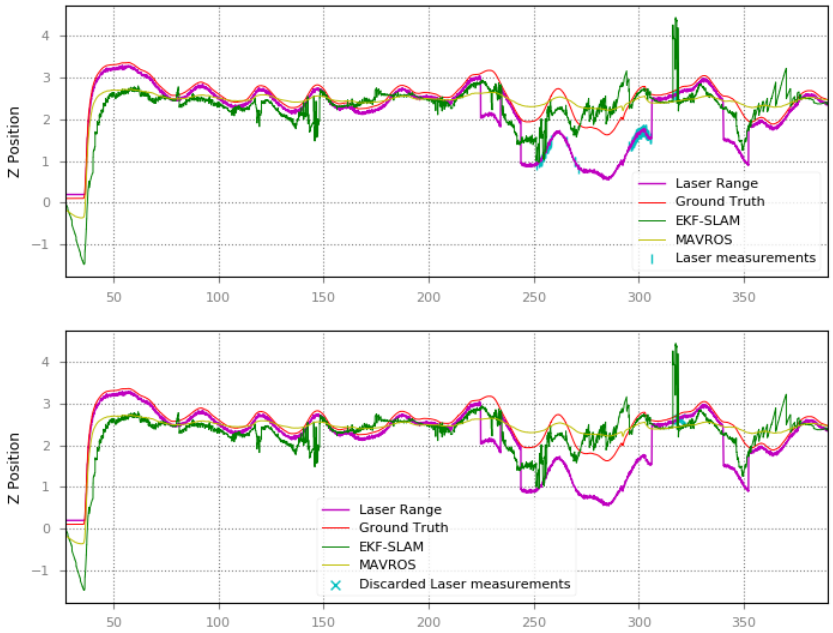


Figure 4.12: Height estimation aggregated with range sensor information. This plot shows the accepted and discarded observations in the height estimation.

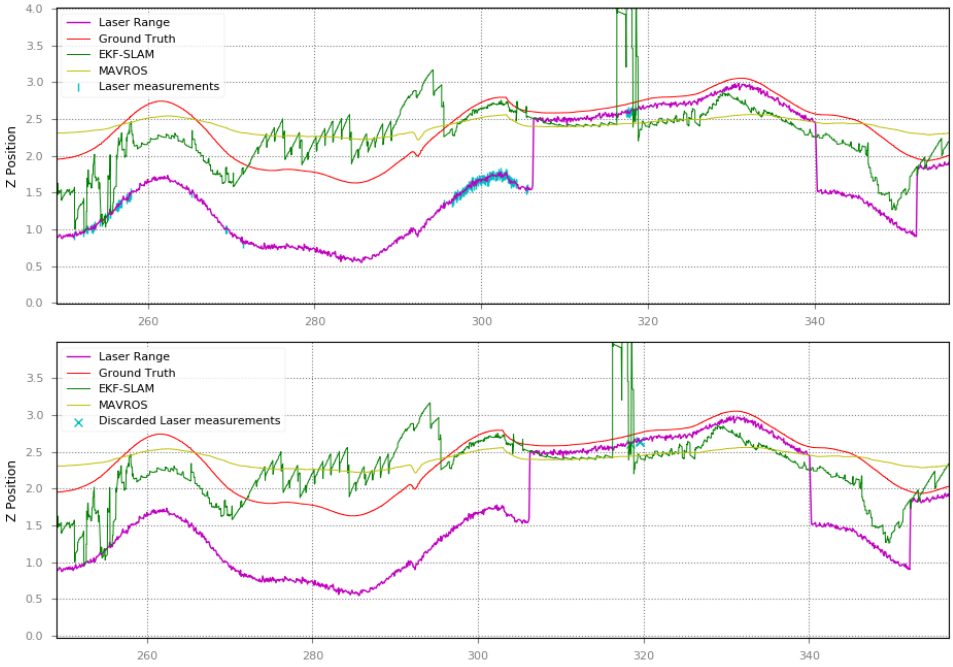


Figure 4.13: Detail of height estimation aggregated with range sensor information. This plot shows the detail of accepted and discarded observations when the Octomap is available and the Z position correction is triggered.

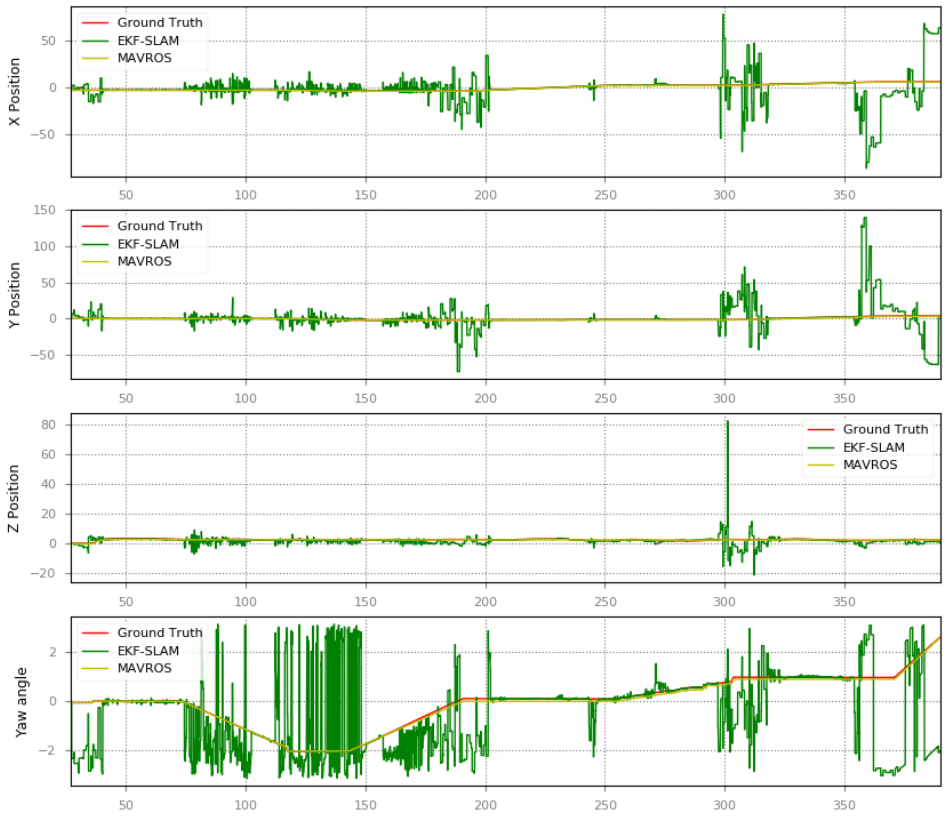


Figure 4.14: EKF-SLAM behavior when NEES test is not used.

4.2.4.2 *Results*

With respect to the importance of the usage of NEES test the Figure 4.14 shows the estimation of position in X, Y, Z and ψ when no NEES test is applied. This means that all the observations are used to update the state vector. It is evident from Figure 4.14 that without the NEES test to filter out the measurements far from the expected, makes the EKF-SLAM algorithm inconsistent.

On the other hand, the second experiment aims to find good values for the χ^2 threshold, and therefore, its importance. The results presented so far have used χ^2 values that corresponds to $\alpha = 0.05$, therefore a 95% of confidence that the measurements are valid. However, different results can be achieve if the threshold value changes, as shown in Figure 4.15 where a value of $\alpha = 0.9$, which corresponds of a confidence of 10% that the measurements are valid, is shown.

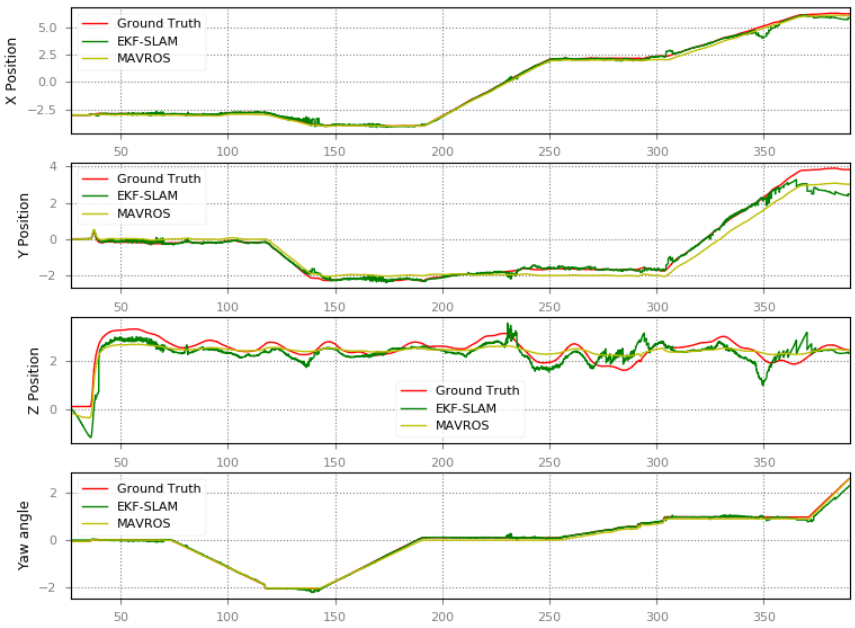


Figure 4.15: EKF-SLAM behavior when $\chi^2_{\alpha=0.9}$ corresponding of a 10% of valid observations.

This change produces an increment in the discarded observations, as it can be seen in Figure 4.16 and Figure 4.17 for markers. As shown in Figure 4.17 the issue commented in Section 4.2.2.2 around seconds 130 and 150 has vanished because those accepted measurements are discarded with the new configuration.

Similarly, the noisy Octomap measurements and the wrong corrections can be overcome thanks to the NEES test. This way, the χ^2 value for one degree of freedom can be reduced in order to accept measurements that are more correlated to those estimated by the filter, hence discarding those noisy Octomap measurements. This procedure can be seen in Figure 4.18, and the detail in Figure 4.19. In these plots an acceptance of 0.9 was used, and as shown the discarded measurements are more than in the ones shown in Figure 4.12.

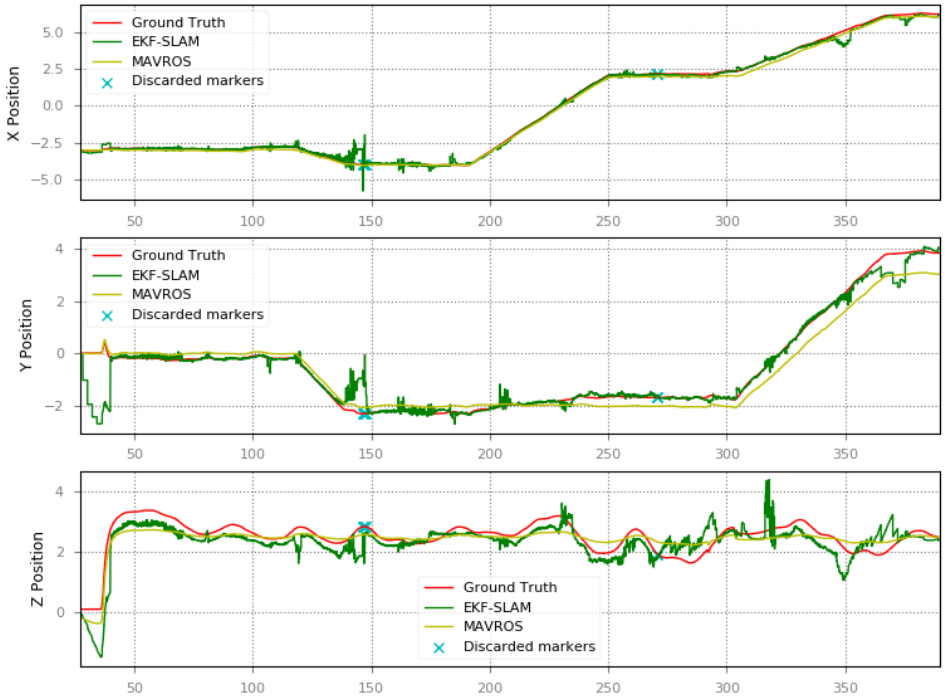


Figure 4.16: Discarded marker observations when $\chi^2_{\alpha=0.05}$ corresponding of a 95% of valid observations.

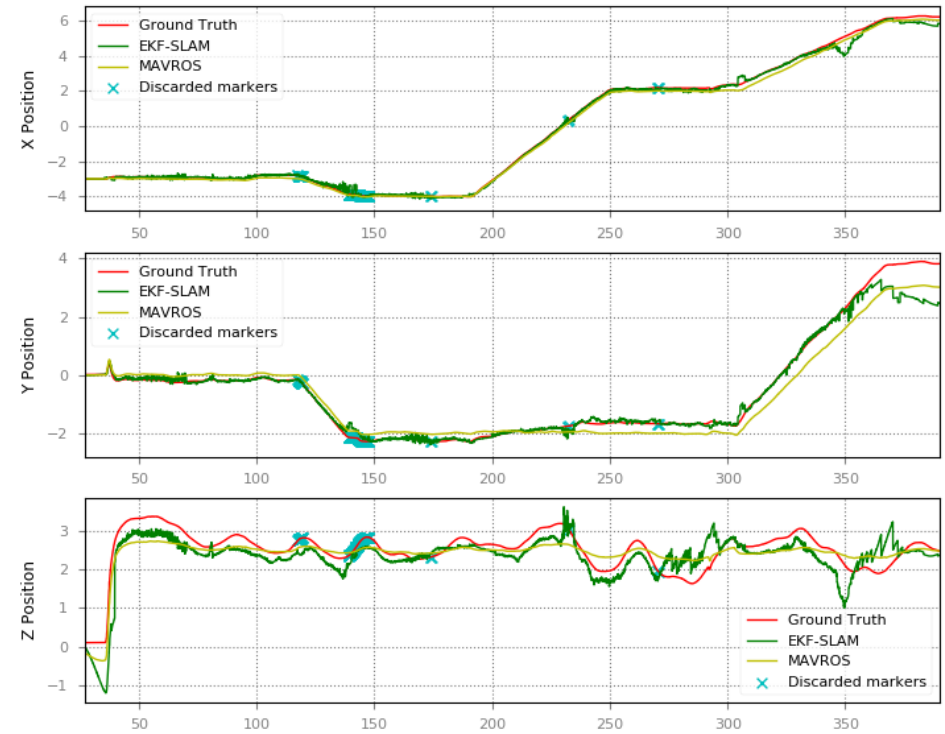


Figure 4.17: Discarded marker observations when $\chi^2_{\alpha=0.9}$ corresponding of a 10% of valid observations.

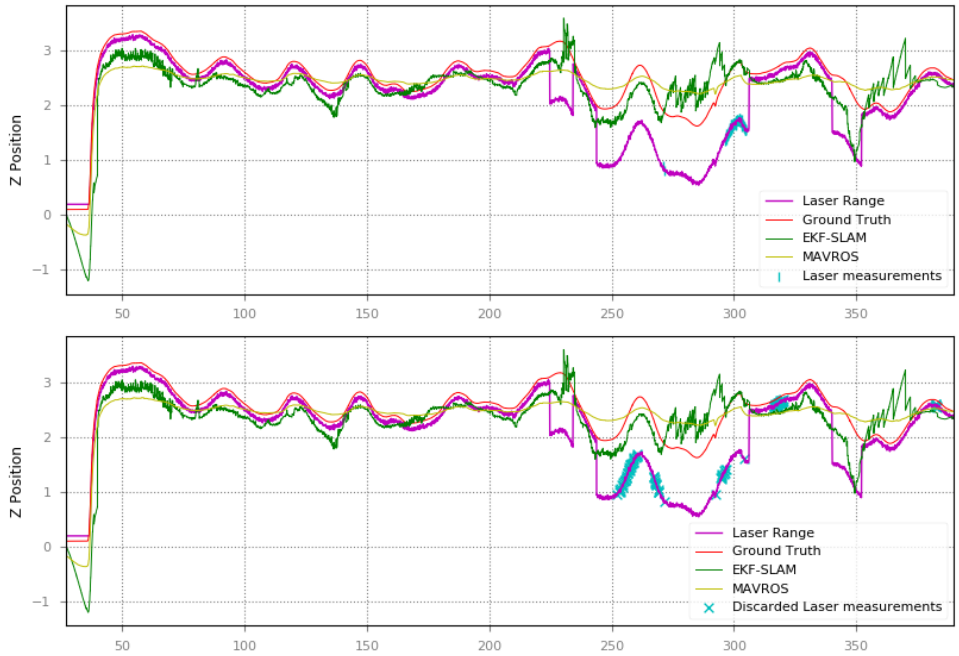


Figure 4.18: Accepted (top) and discarded (bottom) range sensor observations when $\chi^2_{\alpha=0.9}$ corresponding of a 10% of valid observations.

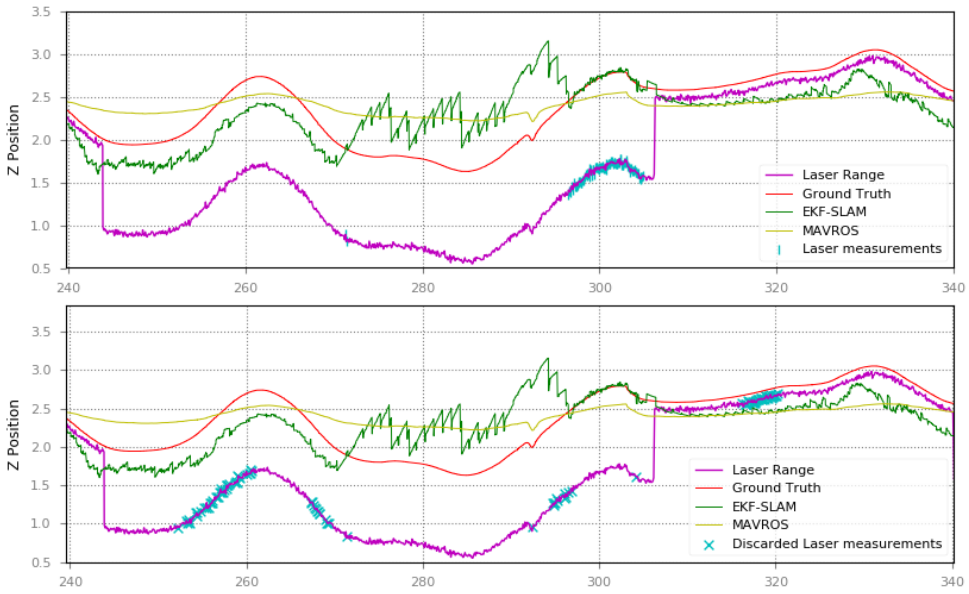


Figure 4.19: Detail of accepted and discarded range sensor observations when $\chi^2_{\alpha=0.9}$

5

CONCLUSIONS AND FUTURE WORK

In Chapter 3 the proposed implementation was presented and in Chapter 4, experiments and their results were showed. This Chapter presents the conclusions related to the performed experiments, and some thoughts about the proposed implementation.

Moreover, future lines of research are proposed for localization and mapping in the context of the Leonardo Drone Contest environment.

5.1 CONCLUSIONS

The EKF-SLAM algorithm is a proved and extensively used algorithm for localization and mapping problems in robotics. The KF introduced in 1960, later refined for non-linear systems and introduced in [15] for localization and mapping problems, is implemented by this work. The proposed implementation was developed in the context of the Leonardo Drone Contest, which has specific characteristics.

Several experiments were conducted with the aim of identifying specific characteristics and implementation details that are important for the objective of the algorithm. In this sense, and as explained in Chapter 4, four set of experiments were conducted, each of them with a specific objective. From these experiments, some conclusions can be extracted.

The first two sets of experiments shed some light about the importance of different types of landmarks, in this particular case, poles and markers. These experiments showed the importance of poles in the correction of X, Y and Z position and the orientation of the drone, and the importance of markers in the localization process. Moreover, the correction of drone's pose is important also for the prediction of the markers' poses as it can be seen in Table 4.1. The average Euclidean distance of the four markers is 0.134, which is good enough for the contest but also perfectible. The same can be observed in the

case of the orientation, which in the worst case is 3°.

The third experiment showed that the height estimation can be corrected using a combination of Octomap and range sensor. As mentioned in Section 4.2.3.2 a key component for the height estimation is the completeness and correctness of the Octomap. This experiment showed that when the Octomap and the sensor range produce good measurements, the height estimation follows the ground truth. However, the experiment could be defined as not conclusive since the amount of time with Octomap measurements is low: only 70 seconds out of 363.

Finally, the last set of experiments showed the importance of the NEES test for the consistency of the filter. As can be appreciated in Figure 4.14, when no NEES test is used, the pose correction is not good enough to be used in any environment. It can be concluded that NEES is needed to filter out invalid measurements. However, a value of χ^2 needs to be found in order to achieve better performance. As showed in Figure 4.17, lower confidence of valid measurements makes the filter discard truly bad measurements, but on the other hand it can discard measurements that can be useful to correct the drone's pose. The χ^2 value should be set for every type of landmark, thus it becomes a new parameter to be tuned in the filter.

It can be concluded that the current implementation works well in the current environment and in the context of the used ROS bags. However, a fine tuning is needed in order to improve its performance in simulation. The observation noise covariance matrices were barely tuned, and the χ^2 values used were the most common ones (confidence of 95%).

As mentioned before, EKF-SLAM is a proved algorithm and works with a decent performance in the current environment. The proposed implementation sets the bases for future developments and improvements, and can be used as baseline for comparison with more complex or novel algorithms. Moreover, the proposed implementation can be extended in order to be applied in other indoor or GNSS-denied environments with minimum modifications. Its architecture was thought to be slightly coupled and highly extensible in the sense that new landmarks and observations types can be added in an easy way.

5.2 FUTURE WORK

As stated in Chapter 1, the current implementation could not be deployed and tested in the real drone, hence it would be an important step to evaluate it. The real drone experimentation should be done considering both poles

and markers for the localization. Moreover, the performance while mapping the environment should be evaluated, as well as the height correction.

As already stated, the current implementation is perfectible and can be improved in different ways. Fine tuning of the algorithm's parameters to improve its performance is needed. Another improvement could be a camera self calibration procedure to improve the Z position estimation using poles and markers. Also, extensive experimentation with Octomap and range sensor for height update is needed.

Additionally, it could be possible to compare the current algorithm with other algorithms like Error-State EKF-SLAM, or Unscented Kalman Filter (UKF) SLAM, and evaluate their performance with the current implementation as baseline.

BIBLIOGRAPHY

[1] *State of the robotics market*. ABI Research, 2019. URL: https://go.abiresearch.com/lp-state-of-the-robotics-market (visited on 11/10/2020) (cit. on p. 1).

[2] *10 ways robots fight against the COVID-19 pandemic*. euRobotics, Apr. 2020. URL: https://www.eu-robotics.net/eurobotics/newsroom/press/robots-against-covid-19.html (visited on 11/10/2020) (cit. on p. 1).

[3] *Open Innovation - Leonardo Drone Contest*. Leonardo. URL: https://www.leonardocompany.com/en/innovation/open-innovation/drone-contest (visited on 04/10/2020) (cit. on p. 1).

[4] *Robot Operating System*. Oct. 6, 2020. URL: https://www.ros.org/ (visited on 06/10/2020) (cit. on pp. 5, 6).

[5] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees." In: *Autonomous Robots* (2013). Software available at http://octomap.github.com (cit. on p. 8).

[6] Mathieu Labbé. "Simultaneous Localization and Mapping (SLAM) with RTAB-Map." Québec, Nov. 2015. URL: https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/3/31/Labbe2015ULaval.pdf (cit. on p. 9).

[7] Kris Hauser. *Robotic Systems*. 2018. URL: http://motion.pratt.duke.edu/RoboticSystems (visited on 09/01/2020) (cit. on p. 12).

[8] Basilio Bona. *Dynamic Modelling of mechatronic systems*. CELID, 2018. ISBN: 8867890115 (cit. on p. 13).

[9] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems." In: *Journal of Basic Engineering* 82.1 (Mar. 1960), pp. 35–45. ISSN: 0021-9223 (cit. on p. 12).

[10] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. Intelligent robotics and autonomous agents. The MIT Press, 2006. ISBN: 978-0-262-20162-9 (cit. on pp. 15, 16).

[11] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. Ed. by Ronald C. Arkin. 2nd. Intelligent robotics and autonomous agents. The MIT Press, 2004. ISBN: 978-0-262-01535-6 (cit. on p. 19).

[12] *MAVROS*. Oct. 6, 2020. URL: https://github.com/mavlink/mavros (visited on 06/10/2020) (cit. on pp. 26, 30).

[13] Gabriele Rogi, Matteo Matteucci, and Diego Avila. *POLIBRI*. URL: https://github.com/AIRLab-POLIMI/POLIBRI (cit. on p. 36).

[14] E. Marchand, F. Spindler, and F. Chaumette. "ViSP for visual servoing: a generic software platform with a wide class of robot control skills." In: *IEEE Robotics and Automation Magazine* 12.4 (2005), pp. 40–52 (cit. on p. 39).

[15] Randall C. Smith and Peter Cheeseman. "On the Representation and Estimation of Spatial Uncertainty." In: *The International Journal of Robotics Research* 5.4 (1986), pp. 56–68 (cit. on p. 63).

[16] Howie Choset et al. *Principles of robot motion: theory, algorithms, and implementation*. Ed. by Ronald C. Arkin. Intelligent Robotics and Autonomous Agents. The MIT Press, 2005. ISBN: 9780262033275.

[17] José A. Castellanos, José Neira, and Juan D. Tardós. *Autonomous Mobile Robots: Sensing, Control, Decision Making and Applications*. Ed. by FRANK L. LEWIS. CRC Press, 2006. Chap. 9. ISBN: 978-0-8493-3748-2.

[18] Nikolaus Correl. *Introduction to Autonomous Robots*. Magellan Scientific, 2020. Chap. 11. ISBN: 978-0692700877.

[19] *MAVLink*. Oct. 6, 2020. URL: https://mavlink.io/en/ (visited on 06/10/2020).

[20] J.L. Blanco. *Derivation and implementation of a full 6D EKF-based solution to bearing-range SLAM*. Tech. rep. University of Malaga, 2008.

[21] Peter Corke. *Robotics, Vision and Control*. Ed. by Bruno Siciliano and Oussama Khatib. Vol. 118. Springer, 2017. ISBN: 978-3-319-54412-0.

[22] Mathieu Labbé and Francois Michaud. "RTAB-Map as an Open-Source Lidar and Visual SLAM Library for Large-Scale and Long-Term Online Operation." In: *Journal of Field Robotics* (2018), pp. 1–31.

[23] Gregory Plett. *Can we automatically detect bad measurements with a Kalman filter?* University of Colorado Boulder. URL: https://www.coursera.org/lecture/battery-state-of-charge/3-3-5-can-we-automatically-detect-bad-measurements-with-a-kalman-filter-tc7ce (visited on 07/20/2020).

A

DOCUMENTATION

An overall description of the proposed architecture can be seen in Section 3.5. The main classes that belong to the localization and mapping node are: `EKF`, `EKFSLAM`, `MapManager`, `Landmark`, `Pole`, `Marker` and `Range`.

If adding a new type of landmark is needed, the new class should extend the `Landmark` class and override and implement the pure virtual methods:

- `Eigen::VectorXd getObservationModel(const DroneState &drone_state) const`
- `Eigen::MatrixXd getJacobianWrtDroneState(const DroneState &drone_state, int drone_state_size) const`

Both methods are needed whatever the type of landmark is added. Moreover, if the new type of landmark is going to be added to the state vector, it is needed to override the following methods:

- `Eigen::MatrixXd getJacobianWrtLandmarkState(const DroneState &drone_state) const`
- `const Eigen::VectorXd & getInverseObservationModel(const DroneState &drone_state, const Eigen::VectorXd &observation)`
- `Eigen::MatrixXd getInverseJacobianWrtDroneState(const DroneState &drone_state, int drone_state_size) const`
- `Eigen::MatrixXd getInverseJacobianWrtLandmarkState(const DroneState &drone_state) const`

MAP

The map is a YAML file that can be found under `map` folder, with the name `landmarks.map.yaml`. It has the following structure:

```
- LandmarkType:
    - id: XXX
```

```
    x: XXX
    y: XXX
    z: XXX
    roll: XXX
    pitch: XXX
    yaw: XXX
    - id: YYY
    ....
- OtherLandmarkType:
    .....
```

PARAMETERS

A file used to configure the parameters of the node can be found under `config` folder. The file is called `params.yaml`:

- `ekf_localization_node`:
  - `enable_poles_subscriber` (bool): enable or disable the usage of poles in the localization process.
  - `enable_markers_subscriber` (bool): enable or disable the usage of markers in the localization and mapping process.
  - `enable_range_subscriber` (bool): enable or disable the usage of range information to correct the height estimation.
  - `drone_pose_topic` (string): defines the topic where the drone's updated odometry is published.
  - `markers_pose_topic` (string): defines the topic where the node will publish the markers' state.
  - `initial_position_x` (float): initial X position of the drone.
  - `initial_position_y` (float): initial Y position of the drone.
  - `initial_position_z` (float): initial Z position of the drone.
  - `initial_position_yaw` (float): initial YAW orientation of the drone.
  - `avg_linear_vel` (float): average drone's linear velocity. This is used in the estimation of the control noise covariance matrix
  - `avg_angular_vel` (float): average drone's angular velocity. This is used in the estimation fo the control noise covariance matrix.
  - `range_noise_covariance` (float): noise covariance value for range sensor.
  - `poles_noise_covariance` (list<float>): noise covariance matrix (3×3) values for the poles observations.

- `markers_noise_covariance` (list<float>): noise covariance matrix (6×6) values for the markers observations.
- `odometry_topic` (string): topic used as control signal. It is used in the prediction step and uses the velocities published in this topic.
- `pole_landmark_topic` (string): topic where the range and bearing information is published for poles observations.
- `marker_landmark_topic` (string): topic where the markers observations is published.
- `octomap_topic` (string): topic where the Octomap information is published.
- `range_sensor_topic` (stirng): topic where the range sensor information is published.

B

USER MANUAL

The implementation presented in this work can be downloaded from https://github.com/AIRLab-POLIMI/POLIBRI. As mentioned before, the ekf_localization package's objective is to estimate the localization of the drone using an EKF filter and map the markers in the environment. The control variables are the linear and angular velocities of the drone with respect to the map, and the state is composed by the drone's position with respect to the map, and the markers' position with respect to the map.

The transform tree can be seen in Figure 3.2. All the odometry messages use the ENU or right hand convention, and all the measurement units are in meters or meters/secs for velocities. Angle units are in radians.

There are currently two nodes: one, ekf_localization_node, is responsible of the state estimation; and the other, qr_code_node, will stamp the messages related to the QR markers. The qr_code_node node subscribes to the following messages:

- /visp_auto_tracker/code_message of type std_msgs/String
- /visp_auto_tracker/stamped_message of type ekf_localization/StringStamped
- /visp_auto_tracker/object_position of type geometry_msgs/PoseStamped

and publishes the following messages:

- /visp_auto_tracker/stamped_object_position of type ekf_localization/QRCodeStamped
- /visp_auto_tracker/stamped_message of type ekf_localization/StringStamped

The ekf_localization_node subscribes to the following messages (*all these messages can be configured in params.yaml*):

- /gazebo/ground_truth or /mavros/local_position/odom

- `/visp_auto_tracker/stamped_object_position`
- `/pole_localization`

and publishes

- `/drone/pose`
- `/markers/pose`

it also provides 3 services:

- `/get_drone_state` which provides the drone pose
- `/get_marker_state` which provides the estimated pose of a given marker
- `/save_map` which saves the landmarks map

SIMULATOR SETUP

To install and run the simulator refer to the `README.txt` in the Simulator folder of the repository.

BUILD AND RUN THE NODES

Dependencies

There are some dependencies needed to compile the code.

- **Eigen 3**: `http://eigen.tuxfamily.org/`
- **YAML-cpp**: `https://github.com/jbeder/yaml-cpp`
- **Boost**: `https://www.boost.org/`

Steps

Once you have set up the simulator, you can build the nodes.

1. Clone the repository on your `~/catkin_ws/src` folder
2. If you are using catkin tools, from `~/catkin_ws` run `catkin build ekf_localization` . If you are using plain catkin, run `catkin_make --only-pkg-with-deps ekf_localization`.
3. In different terminals run:
   a) `roscd px4` and then, `no_sim=1 make px4_sitl_default gazebo`

b) from `~/catkin_ws`, run `./launch_gazebo.sh`

c) `roslaunch rtabmap_ros my_stereo_mapping_2.launch` (if you want to run RViz, you can add at the end `rviz:=true`)

d) `roslaunch ekf_localization my_launch.launch`, this will run the node responsible of the state estimation and the one responsible of stamp the marker's information

e) If you want to run the planner scripts, you can run in different terminals:

   i. `python Simulator/offboard/start_offboard.py`

   ii. `python Simulator/offboard/path_generation.py`

Build and run the nodes using a rosbag

The nodes can be debugged by launching the [debug.launch](launch/debug.launch) file. To do so, you just need to run the following: `$ roslaunch ekf_localization debug.launch`.

Several arguments can be passed to the launch file, like the folder to find a rosbag, the name of the rosbag, record a new rosbag, specify topics to record, launch RViz, etc.