POLITECNICO DI MILANO

AIR Lab - Artificial Intelligence and Robotics Lab
Dipartimento di Elettronica, Informazione e Bioingegneria

MSc. in Computer Science and Engineering

Implementation of a 6 DoF EKF-SLAM for Leonardo Drone Contest

Author:
Diego Emanuel Avila
Matricola: 903988

Supervisor:
Prof. Matteo Matteucci

December 2020

Preface

Some preface

Diego Emanuel Avila
Milano
October 2020

Abstract

Abstract

Contents

List of Figures vi

List of Tables viii

List of Acronyms xi

1 Introduction 1

2 Theoretical Background 3
- 2.1 Robot Operating System . . . 3
  - 2.1.1 Nodes . . . 3
  - 2.1.2 Topics . . . 3
  - 2.1.3 Services . . . 3
  - 2.1.4 Tools . . . 3
  - 2.1.5 MAVLink and MAVROS . . . 4
  - 2.1.6 Octomap . . . 4
- 2.2 Homogeneous Transformation . . . 4
- 2.3 Kalman Filter . . . 5
  - 2.3.1 Extended Kalman Filter . . . 6
- 2.4 Simultaneous Localization and Mapping . . . 7
  - 2.4.1 EKF-SLAM . . . 9
  - 2.4.2 Adding new landmarks . . . 12

3 EKF-SLAM Implementation 13
- 3.1 The Drone . . . 13
  - 3.1.1 Characteristics . . . 13
  - 3.1.2 Reference Frames . . . 15
- 3.2 Prediction . . . 15
- 3.3 Correction . . . 17
  - 3.3.1 Observation model for Poles . . . 18
  - 3.3.2 Observation model for Markers . . . 20
- 3.4 Normalized Estimation Error Squared . . . 22
- 3.5 Overall Architecture . . . 23
  - 3.5.1 ROS nodes . . . 24

4 Experimental Results 27
- 4.1 Simulation Results . . . 27
- 4.2 Empirical Results . . . 27

5 Conclusions and Future Work 29
- 5.1 Conclusion . . . 29
- 5.2 Future Work . . . 29

A Source Code **31**

B Matlab Source Code **33**

Bibliography **35**

List of Figures

2.1 Linear and nonlinear transformation of a Gaussian random variable 6
2.2 Linearization applied in EKF 8
2.3 Example of SLAM problem 10

3.1 3DR Iris Quadrotor frame. 13
3.2 Main reference frames in the system. 15
3.3 NED to ENU conversion scheme. mav (b) 16
3.4 Range and Bearing example 19
3.5 Example of the drone observing a marker 20
3.6 Components diagram of the EKF Localization node 23
3.7 Class diagram of the EKFSLAM node 24
3.8 ROS graph of the system 25
3.9 Detail of the EKF-SLAM node interactions 26

List of Tables

3.1 PixHawk 4 Specification 14

List of Acronyms

IDL Interface Description Language . . . 3

ROS Robot Operating System . . . 3

RViz ROS Visualization . . . 4

SLAM Simultaneous Localization And Mapping . . . 8

EKF Extended Kalman Filter . . . 6

KF Kalman Filter . . . 5

NEES Normalized Estimation Error Squared . . . 22

ENU East-North-Up . . . 4

NED North-East-Down . . . 4

Introduction 1

Theoretical Background 2

2.1 Robot Operating System

The *Robot Operating System* (ROS) 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. 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 Octomap

something about octomap

2.2 Homogeneous Transformation

something about matrices

2.3 Kalman Filter

Introduced by Kalman (1960), 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:

1. The state transition probability $p\left(x_t|u_t, x_{t-1}\right)$ 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 $p\left(z_t|x_t\right)$ 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.

Algorithm 2.1: Kalman Filter algorithm

```
  Input: $\mu_{t-1}$, $\Sigma_{t-1}$, $u_t$, $z_t$
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 $\mu_t$, $\Sigma_t$
```

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.

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.

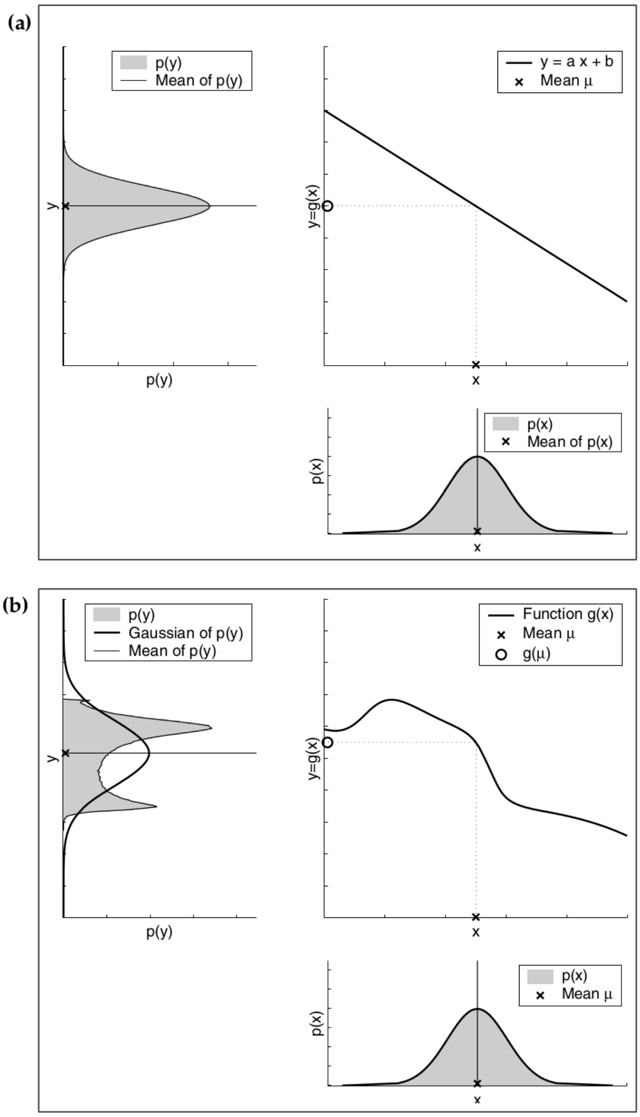


FIGURE 2.1: Linear **(a)** and nonlinear **(b)** transformation of a Gaussian random variable. The lower right plot shows the density function of the random variable. The upper right plot shows the transformation of the random variable. The upper left plot shows the resulting density function. Thrun et al. (2006)

The Figure 2.1a 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 2.1b, this does not happen if the transformation is not linear. In this case, assuming the original random variable is 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 $g\left(u_t, x_{t-1}\right)$ and matrix H is replaced by function $h\left(x_t\right)$, 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

$$g'\left(u_t, x_{t-1}\right) = \frac{\partial g\left(u_t, x_{t-1}\right)}{\partial x_{t-1}} \tag{2.1}$$

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

$$g\left(u_t, x_{t-1}\right) = g\left(u_t, \mu_{t-1}\right) + g'\left(u_t, \mu_{t-1}\right)\left(x_{t-1} - \mu_{t-1}\right) \tag{2.2}$$

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.

The same linearization is applied to the observation function h:

$$h\left(x_t\right) = h\left(\hat{\mu}_t\right) + h'\left(\hat{\mu}_t\right)\left(x_t - \hat{\mu}_t\right) \tag{2.3}$$

$$h'\left(\hat{\mu}_t\right) = \frac{\partial h\left(x_t\right)}{\partial x_t} \tag{2.4}$$

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 2.2 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.

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

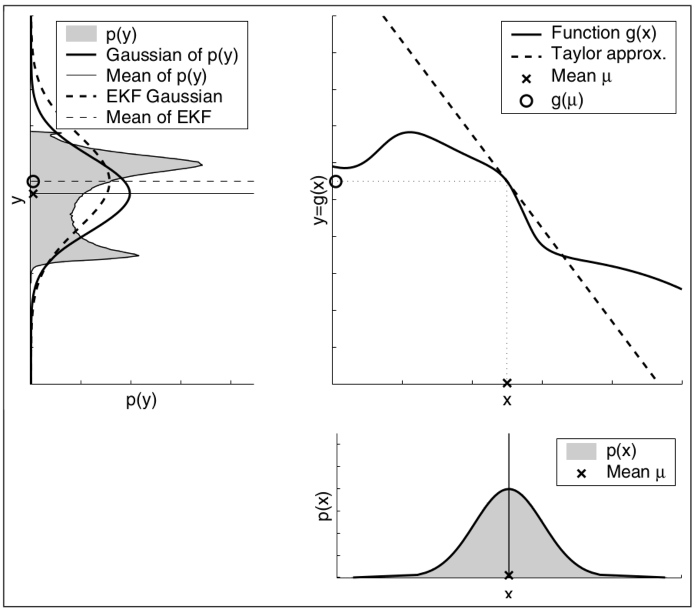


Figure 2.2: Linearization applied in EKF. In this case, the nonlinear function g is approximated using first order Taylor expansion, that is a linear function tangent to g at the mean of the original density function. The linearization is not perfect, so it adds an error, depicted in the upper left plot. This error is the difference between the dashed line and the solid line. Thrun et al. (2006)

Algorithm 2.2: Extended Kalman Filter algorithm

```
Input: $\mu_{t-1}$, $\Sigma_{t-1}$, $u_t$, $z_t$
$\hat{\mu}_t = g\left(u_t, \mu_{t-1}\right)$
$\hat{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
$K_t = \hat{\Sigma}_t H_t^T \left(H_t \hat{\Sigma}_t H_t^T + Q_t\right)^{-1}$
$\mu_t = \hat{\mu}_t + K_t\left(z_t - h\left(\hat{\mu}_t\right)\right)$
$\Sigma_t = (I - K_t H_t)\hat{\Sigma}_t$
return $\mu_t$, $\Sigma_t$
```

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 *Simultaneous Localization And Mapping* (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 2.3, 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: $p\left(x_t, m | z_{1:t}, u_{1:t}\right)$, 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):

$$\mu = \begin{bmatrix} q_r & m_0 & \dots & m_{n-1} \end{bmatrix}^T \tag{2.5}$$

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 $p\left(\mu_t | z_t, u_t\right)$.

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 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 *computeJacobian* computes, indeed, the Jacobian matrix for the motion model g, and the resulting matrix has the same size as the

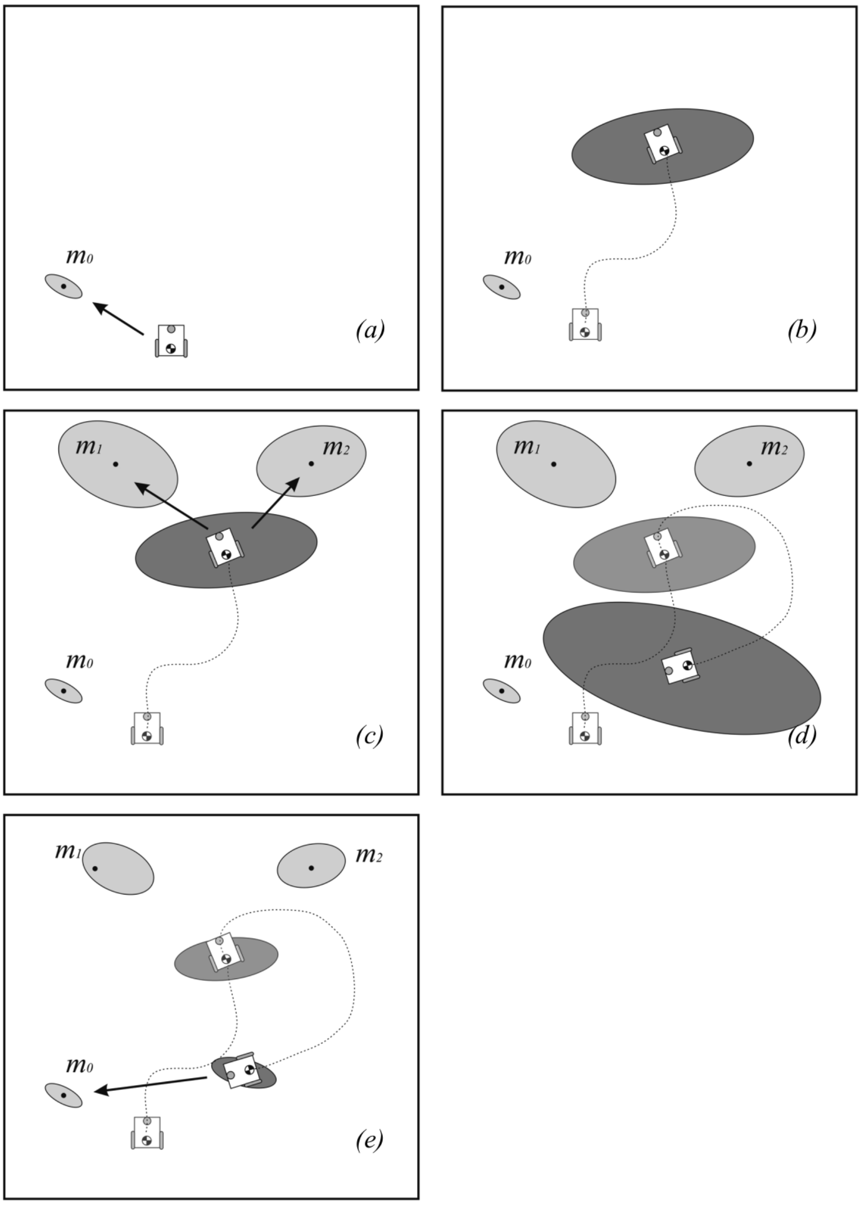


Figure 2.3: At the beginning **(a)** the robot has low uncertainty regarding its pose. As it moves around the environment its uncertainty, represented by the dark gray ellipsis, grows **(b), (c), (d)**, until it sees a known landmark **(e)**, making the position uncertainty to shrink. Siegwart et al. (2004)

covariance matrix and has the following characteristic:

$$G_t = \begin{bmatrix} G_r & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{2.6}$$

$$G_r = \begin{bmatrix} \frac{\partial x'}{\partial \mu_{t-1,x}} & \frac{\partial x'}{\partial \mu_{t-1,y}} & \frac{\partial x'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial y'}{\partial \mu_{t-1,x}} & \frac{\partial y'}{\partial \mu_{t-1,y}} & \frac{\partial y'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial \theta'}{\partial \mu_{t-1,x}} & \frac{\partial \theta'}{\partial \mu_{t-1,y}} & \frac{\partial \theta'}{\partial \mu_{t-1,\theta}} \end{bmatrix} \tag{2.7}$$

Algorithm 2.3: EKF-SLAM algorithm

```
   Input: $\mu_{t-1}$, $\Sigma_{t-1}$, $u_t$, $z_t$
1  $\hat{\mu}_t = g\left(\mu_{t-1}, u_t\right)$;
2  $G_t = computeJacobian\left(g\right)$;
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          $addLandmarkToStateVector\left(z_t^i\right)$
7      $H_t^i = computeJacobian\left(h^i\right)$;
8      $v^i = z_t^i - h^i\left(\hat{\mu}_t\right)$;
9      $S = H_t^i\hat{\Sigma}_tH_t^{iT} + Q_t$;
10     $K_t^i = \hat{\Sigma}_tH_t^{iT}S^{-1}$ ;
11     $\mu_t = \hat{\mu}_t + K_t^i\left(v^i\right)$ ;
12     $\Sigma_t = (I - K_t^iH_t^i)\hat{\Sigma}_t$ ;
13 return $\mu_t$, $\Sigma_t$
```

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 measurement. 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:

$$z^i = h^i\left(x_t\right) + w^i \tag{2.8}$$

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 2.3, 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.

$$h^{-1}\left(\mathbf{x_r}, \mathbf{z}\right) = \begin{bmatrix} x_l \\ y_l \end{bmatrix} \tag{2.9}$$

$$\mathbf{x_t} = y\left(\mathbf{x}, \mathbf{z}\right) = \begin{bmatrix} \mathbf{x} \\ h^{-1}\left(\mathbf{x_r}, \mathbf{z}\right) \end{bmatrix} \tag{2.10}$$

In equation (2.9) $\mathbf{x_r}$ refers to the elements in the state vector that corresponds to the robot's pose, and $\mathbf{z}$ 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 (2.11).

$$\hat{\mathbf{\Sigma}} = \mathbf{Y_x}\mathbf{\Sigma}\mathbf{Y_x}^T + \mathbf{Y_z}\mathbf{Q}_t\mathbf{Y_z}^T \tag{2.11}$$

$$\mathbf{Y_x} = \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial x}{\partial x} \\ \frac{\partial h^{-1}}{\partial x} \end{bmatrix} = \begin{bmatrix} \mathbf{I} \\ \mathbf{G_x} \end{bmatrix} \tag{2.12}$$

$$\mathbf{Y_z} = \frac{\partial y}{\partial z} = \begin{bmatrix} \frac{\partial x}{\partial z} \\ \frac{\partial h^{-1}}{\partial z} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{G_z} \end{bmatrix} \tag{2.13}$$

where matrices $\mathbf{Gx}$ and $\mathbf{Gz}$ are the Jacobian of function h^{-1} with respect to the state vector and the observation vector respectively. By substituting the Jacobians in equations (2.12) and (2.13) into equation (2.11), the following matrix is obtained:

$$\hat{\mathbf{\Sigma}} = \begin{bmatrix} \mathbf{\Sigma} & \mathbf{\Sigma}\mathbf{G_x}^T \\ \mathbf{G_x}\mathbf{\Sigma} & \mathbf{G_x}\mathbf{\Sigma}\mathbf{G_x}^T + \mathbf{G_z}\mathbf{Q}_t\mathbf{G_z}^T \end{bmatrix} \tag{2.14}$$

The linearized covariance update can be factored as shown in equation (2.15) by assuming that $A \equiv 1$, $B \equiv 0$, $C \equiv 0$ and $D \equiv \mathbf{G_z}$

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} P & 0 \\ 0 & Q \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix}^T = \begin{bmatrix} APA^T + BQB^T & APC^T + BQD^T \\ CPA^T + DQB^T & CPC^T + DQD^T \end{bmatrix} \tag{2.15}$$

$$\hat{\mathbf{\Sigma}} = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{G_x} & \mathbf{G_z} \end{bmatrix} \begin{bmatrix} \mathbf{\Sigma} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_t \end{bmatrix} \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{G_x} & \mathbf{G_z} \end{bmatrix}^T \tag{2.16}$$

$$\tag{2.17}$$

EKF-SLAM Implementation 3

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. In the first stage involves the prediction of the next state of the system, meanwhile, during the correction stage, this estimation is updated.

In this chapter, an EKF-SLAM implementation is shown, starting from the used drone characteristics, going through the prediction and correction steps, 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. Its total weight is XX kg., and it diameter is XX cm.

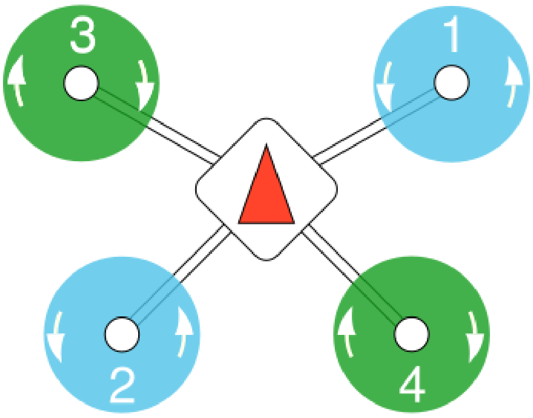


FIGURE 3.1: 3DR Iris Quadrotor frame.

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.

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.

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 laser 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 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 laser range sensors were mounted every 45 degrees, and one laser range sensor was mounted pointing downwards in order to estimate the current drone height.

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.

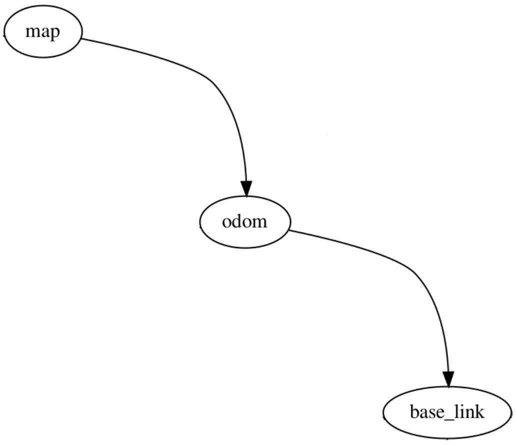


FIGURE 3.2: Main reference frames in the system.

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 Prediction

During the prediction step, the motion model and the covariance matrix update are computed. The motion model will update the state vector, which 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 `nav_msgs::Odometry`, provide the linear velocity, the angular velocity and pose information. The velocity information is used to

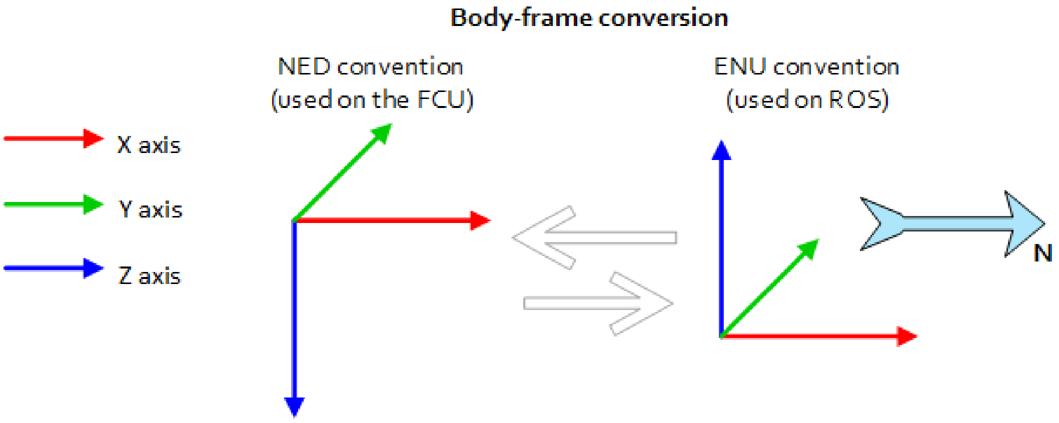


Figure 3.3: NED to ENU conversion scheme. mav (b)

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 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:

$$G_t = \begin{bmatrix} G_r & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{3.6}$$

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

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

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 (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 Correction

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.

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

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 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:

$$\begin{bmatrix} p_{i,x}^b \\ p_{i,y}^b \\ p_{i,z}^b \end{bmatrix} = \boldsymbol{T_r}^{-1} \begin{bmatrix} p_{i,x}^w \\ p_{i,y}^w \\ p_{i,z}^w \end{bmatrix} \tag{3.16}$$

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

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_i^w 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.

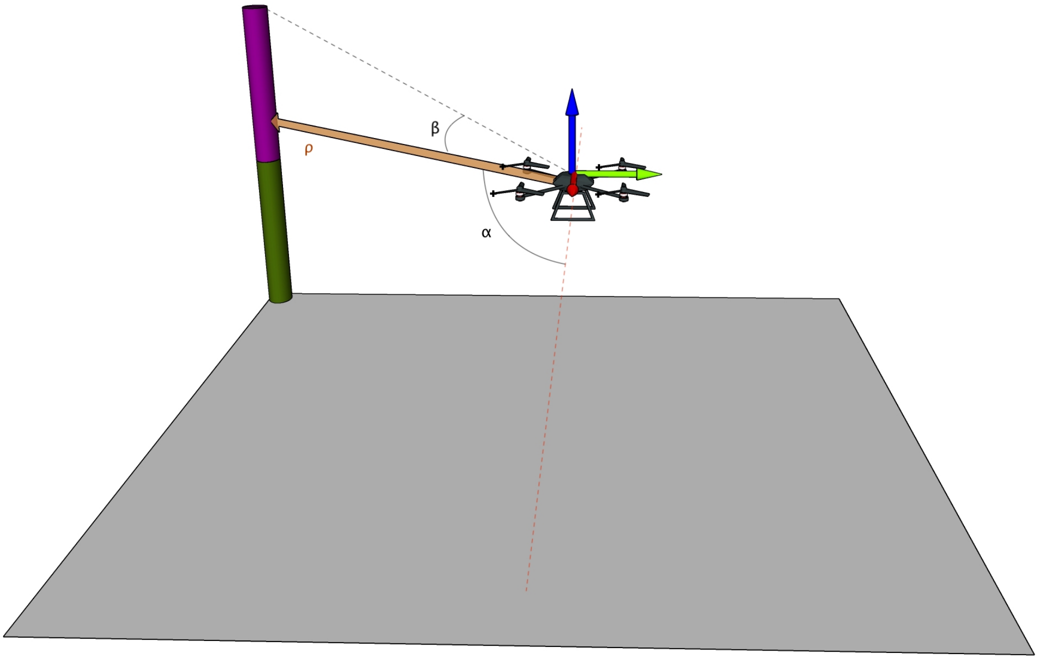


FIGURE 3.4: Range and bearing example. The drone observes a pole, process the data from the sensors and estimates the distance (ρ), the elevation angle (β), and the azimuth angle (α). The elevation angle is calculated based on the top extreme of the pole.

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 4th column are all 0, which means, as said before, that the observation of a pole will not affect the pose of the markers.

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 `geometry_msgs::PoseStamped`, which provides the position and orientation of the seen marker. An example of this situation can be seen in Figure 3.5.

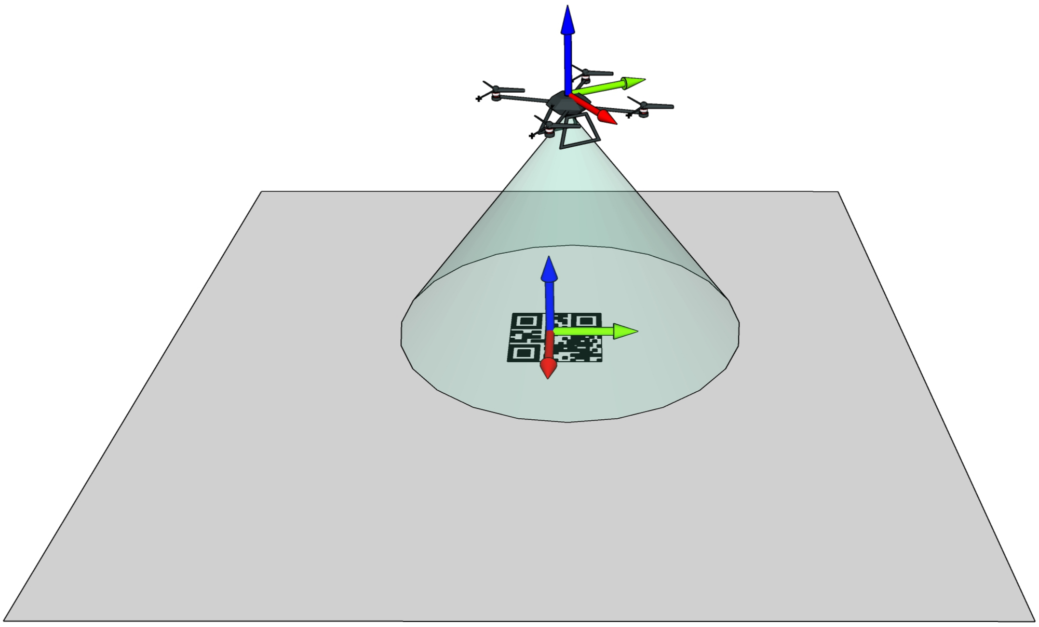


FIGURE 3.5: Example of the drone observing a marker. The camera visualize a marker and the node `visp_auto_tracker` estimates its pose with respect to the camera reference frame.

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), 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 Appendix A. 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 2.11, 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.

3.4 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:

$$D = \sqrt{(\boldsymbol{x} - \boldsymbol{\mu})\,\Sigma^{-1}\,(\boldsymbol{x} - \boldsymbol{\mu})} \tag{3.22}$$

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, the message type is `ekf_localization ::RangeAndBearingPole`, which contains the range and bearing information, and in the case of markers the message type is `geometry_msgs::Pose`, which contains the position and orientation information of the observed marker. 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

$$v^i = z_t^i - h^i\left(\hat{\mu}_t\right)$$
$$S = H_t^i \hat{\Sigma}_t H_t^{iT} + Q_t$$

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:

$$D_i^2 = v^i S^{-1} v^i < \chi^2_{d,1-\alpha} \tag{3.23}$$

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

$$\mathbb{E}\left[\boldsymbol{x} - \hat{\boldsymbol{x}}\right] = 0$$
$$\mathbb{E}\left[(\boldsymbol{x} - \hat{\boldsymbol{x}})\,(\boldsymbol{x} - \hat{\boldsymbol{x}})^T\right] \leq \Sigma$$

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

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

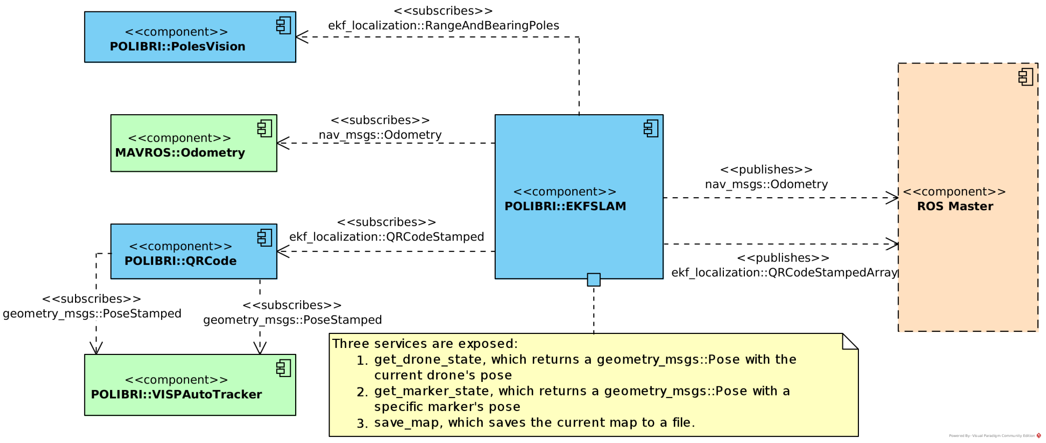
However, the ground truth is only available in simulated environments thus, the consistency of the filter is maintained by using observations that satisfy the test in equation (3.23). This way, the filter will discard observations that do not satisfy the innovation test, maintaining a consistent estimation of the state and therefore, the map.

3.5 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 `geometry_msgs::Pose` message.
2. `get_marker_state`: takes as argument the id of a marker, and returns the current estimated pose of that marker`geometry_msgs::Pose` message.
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.


<<subscribes>>
ekf_localization::RangeAndBearingPoles
<<component>>
POLIBRI::PolesVision
<<subscribes>>
nav_msgs::Odometry
<<component>>
MAVROS::Odometry
<<component>>
POLIBRI::EKFSLAM
<<publishes>>
nav_msgs::Odometry
<<component>>
ROS Master
<<subscribes>>
ekf_localization::QRCodeStamped
<<component>>
POLIBRI::QRCode
<<publishes>>
ekf_localization::QRCodeStampedArray
<<subscribes>>
geometry_msgs::PoseStamped
<<subscribes>>
geometry_msgs::PoseStamped
<<component>>
POLIBRI::VISPAutoTracker
Powered By: Visual Paradigm Community Edition

Three services are exposed:
1. get_drone_state, which returns a geometry_msgs::Pose with the current drone's pose
2. get_marker_state, which returns a geometry_msgs::Pose with a specific marker's pose
3. save_map, which saves the current map to a file.

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

It is worth to mention that the messages of type `ekf_localization::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 `std_msgs::String` message and the pose as a `geometry_msgs::PoseStamped` message. Due to this situation, the `QRCode` node is responsible of match both data together and publish an `ekf_localization::QRCodeStamped` message which contains the marker's identifier and its pose.

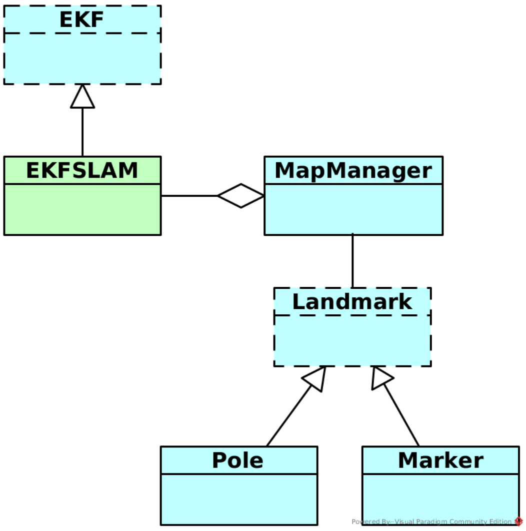


FIGURE 3.7: Class diagram of the EKFSLAM node. The `EKFSLAM` class is composed by a `MapManager` instance, which is composed by a list of `Landmarks`. The `Landmark` class is abstract and its concrete classes are `Pole` and `Marker`.

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.

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 and a `Pole` class, each of which is responsible of provide the observation model associated to it and the needed Jacobian matrices for the algorithm, hiding the implementation to the `EKFSLAM` class.

3.5.1 ROS nodes

Regarding the ROS node that comprise the system, in Figure 3.8 all the ones that interact with the localization node are shown. The EKF-SLAM node is called `/ekf_localization_node`, and as mentioned before, it interacts with `/mavros` node, all the poles identification nodes (`/poles_vision_#`), the `/qr_code_node` which publishes the markers position with its identifier, and the `/rtabmap` node that provides the Octomap messages.

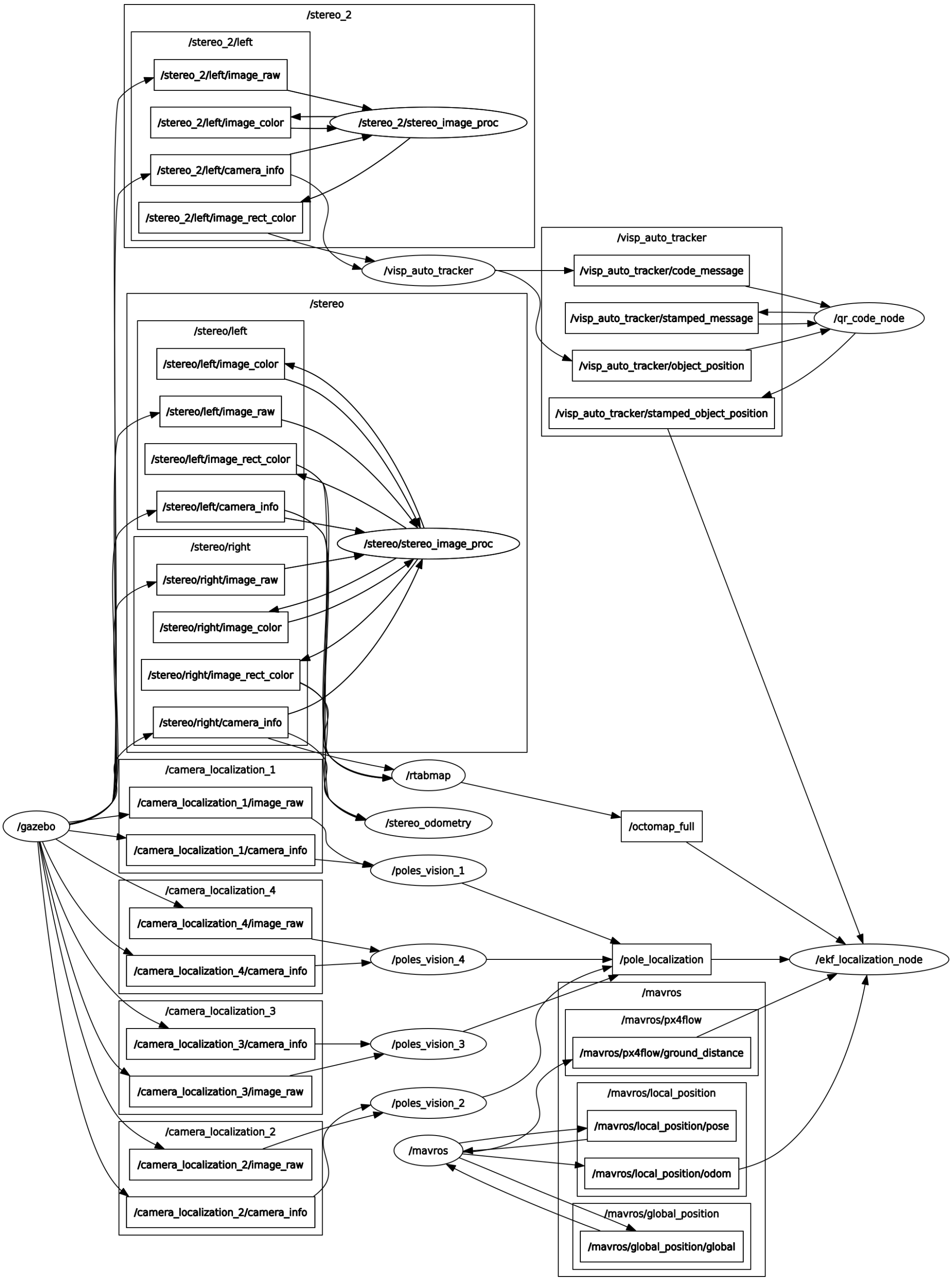

/stereo_2
/stereo_2/left
/stereo_2/left/image_raw
/stereo_2/left/image_color
/stereo_2/stereo_image_proc
/stereo_2/left/camera_info
/stereo_2/left/image_rect_color
/visp_auto_tracker
/visp_auto_tracker
/visp_auto_tracker/code_message
/visp_auto_tracker/stamped_message
/qr_code_node
/visp_auto_tracker/object_position
/visp_auto_tracker/stamped_object_position
/stereo
/stereo/left
/stereo/left/image_color
/stereo/left/image_raw
/stereo/left/image_rect_color
/stereo/left/camera_info
/stereo/stereo_image_proc
/stereo/right
/stereo/right/image_raw
/stereo/right/image_color
/stereo/right/image_rect_color
/stereo/right/camera_info
/camera_localization_1
/rtabmap
/camera_localization_1/image_raw
/gazebo
/stereo_odometry
/octomap_full
/camera_localization_1/camera_info
/poles_vision_1
/camera_localization_4
/camera_localization_4/image_raw
/poles_vision_4
/pole_localization
/ekf_localization_node
/camera_localization_4/camera_info
/mavros
/camera_localization_3
/mavros/px4flow
/camera_localization_3/camera_info
/poles_vision_3
/mavros/px4flow/ground_distance
/camera_localization_3/image_raw
/mavros/local_position
/poles_vision_2
/camera_localization_2
/mavros/local_position/pose
/mavros
/camera_localization_2/image_raw
/mavros/local_position/odom
/mavros/global_position
/camera_localization_2/camera_info
/mavros/global_position/global

Figure 3.8: 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.

As mentioned, the Figure 3.8 shows all the nodes (depicted as ellipses) with its interactions: a node publishes a message topic, and the topic is consumed by a node. This interaction is represented by the 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 in Figure 3.8 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.

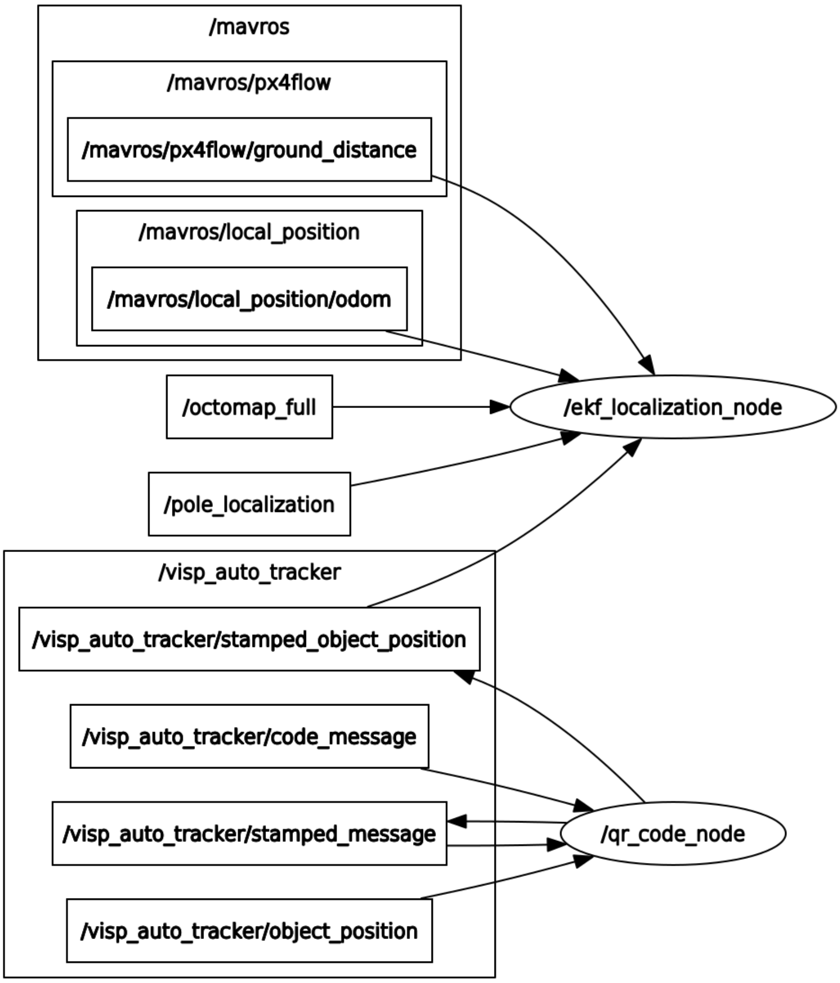

/mavros
/mavros/px4flow
/mavros/px4flow/ground_distance
/mavros/local_position
/mavros/local_position/odom
/octomap_full
/ekf_localization_node
/pole_localization
/visp_auto_tracker
/visp_auto_tracker/stamped_object_position
/visp_auto_tracker/code_message
/visp_auto_tracker/stamped_message
/qr_code_node
/visp_auto_tracker/object_position

Figure 3.9: Detail of the EKF-SLAM node interactions

Figure 3.9 shows the detail for the `/ekf_localization_node` with all its subscriptions. It is worth to mention the `/qr_code_node`, which subscribes to `/visp_auto_tracker/code_message` and `/visp_auto_tracker/object_position`, and publishes a `/visp_auto_tracker/stamped_object_position`.

Experimental Results 4

4.1 Simulation Results

4.2 Empirical Results

Conclusions and Future Work 5

5.1 Conclusion

5.2 Future Work

Appendix A

Source Code

Appendix B

Matlab Source Code

Bibliography

Mavlink, a. URL `https://mavlink.io/en/`.

Mavros, b. URL `https://github.com/mavlink/mavros`.

Robot operating system. URL `https://www.ros.org/`.

José A. Castellanos, José Neira, and Juan D. Tardós. *Autonomous Mobile Robots: Sensing, Control, Decision Making and Applications*. CRC Press, 2006. ISBN 978-0-8493-3748-2.

Howie Choset et al. *Principles of robot motion: theory, algorithms, and implementation*. Intelligent Robotics and Autonomous Agents. The MIT Press, 2005. ISBN 9780262033275.

R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 03 1960. ISSN 0021-9223. doi: 10.1115/1.3662552. URL `https://doi.org/10.1115/1.3662552`.

Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. Intelligent robotics and autonomous agents. The MIT Press, 2nd edition, 2004. ISBN 978-0-262-01535-6.

Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. Intelligent robotics and autonomous agents. The MIT Press, 2006. ISBN 978-0-262-20162-9.