Marco Bergamin

Indoor localization using visual information and passive landmarks

Universidade de Aveiro Departamento de Engenharia Mecânica
2015

Marco Bergamin

Indoor localization using visual information and passive landmarks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em (designação do mestrado), realizada sob a orientação científica do Dr. Vitor Santos, Professor Associado do Departamento de Engenharia Mecânica da Universidade de Aveiro.

The jury

President

Prof. Dr. Miguel Armando Riem Oliveira

professor associado da Universidade de Aveiro

Prof. Dr. Vitor M. Ferreira dos Santos

professor associado da Universidade de Aveiro

Acknowledgements

Abstract   Nowadays the level of automation in industry is constantly increasing. We can easily find completely automated production lines, but for the moment the interactions between machines and human operators are limited to a small set of tasks. One possible way of increasing the efficiency of a given plant is to use intelligent robots instead of human resources for the transportation of objects across different places in the same industrial complex. Traditional AGVs (Automatically Guided Vehicles) are now commonly used for these tasks, but most of them follow strict paths marked by "wires" or special lines traced on the floor. There are also other solutions based on laser and special reflectors that allow triangulation of the robot inside the plant. Nonetheless, the "floor-based" solutions have properties that limit their usage, whereas laser/reflector solutions, besides being expensive, require a rather elaborate procedure to set up the layout changes. These restrictions open the way to explore and research new vision based solutions, especially if they can be made easier to configure and more cost-effective at the same time. The solution proposed aims to use simple markers, namely simple Data Matrix codes, to obtain a "raw" pose estimation through trilateration. Then the results are combined with heterogeneous data provided by odometry and (if present) from an inertial measurement unit using an Extended Kalman Filter. The advantages of this solution are that it is cheap, flexible and robust: the markers are common sheet of paper and they can therefore easily be printed and placed in the environment. Moreover the AGVs are not forced to follow a fixed path and this make it possible to use sophisticated path planning algorithms. The obtained results are promising, but the performance of this type of system depends on many factors: detection algorithm, localization method, quality of the odometry and efficiency of the sensor fusion algorithm. Despite these problems, the tests have shown that even with a non fully optimized algorithm, a precision of $0.2m$ can be reached, confirming the validity of this technology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Contex of the problem

Nowadays, the level of automation in industry is constantly on the rise. We can easily find completely automated production lines, but for the moment the interaction between machines and human operators is limited to a small set of tasks. One possible way of increasing the efficiency inside a given plant is to use smart robots to help human operators to transport objects and materials across different places in the same industrial complex. Traditional AGVs (Automatically Guided Vehicles) are now commonly used for these tasks, and most of them follows strict paths marked by "wires" or special lines traced on the floor (fig. 1.1a). There are also other solutions based on laser (fig. 1.1b) and special reflectors to allow triangulation of the robot.

Nonetheless, the "floor-based" solutions have specificities that limit their usage, while laser/reflector solutions are not only expensive, but also require a rather elaborate procedure in order to set up the layout changes. These restrictions open the way to research and develop vision based solutions, especially if they can be made easier to configure and simultaneously more cost-effective.

In real situations it is not always possible to predispose the environment to be fully robot-friendly (for example by designing dedicated paths inside the complex), but we can have some a priori information, such as the map of the environment. Magnetic stripes are cheap solution but, as some companies have already noticed, they aren't at all ideal in many environments, since other vehicles and transporters can damage them in time. For this reason as well as to keep productions costs low, using vision based technologies becomes appealing, despite the expected higher complexity of the needed algorithms.

(a) Magnetic guided AGV

(b) Laser guided AGV

Figure 1.1

1.2 State of the art

Nowadays, it is very common to find AGVs in many industry fields. It is interesting to notice that many types of technologies can be used to allow the self-localization of these specials robots, each one with its own advantages and disadvantages.

As it will be shown in subsection 1.2, using different approaches means using different kinds of sensors. The problem of how to efficiently combine all the information provided by a number of different (and in general heterogeneous) sensors is widely discussed in the literature of the last years and it is usually referred as the problem of *sensor fusion*. A common solution to this problem is using the *Kalman Filter*[1] or one of its declinations (for instance the *Extended Kalman Filter*[2] or the *Unscented Kalman Filter*[3]).

In the particular case of this thesis, the robot used to test the algorithms (AtlasMV [4]) is a car-like robot, and this makes the problem of the sensor fusion not much different from the problem of outdoor localization of a common car with a GPS (Global Positioning System)[5]; instead of the estimation of the position provided by a GPS, the estimation of the position provided by visual information can be used.

Also, the estimation of the position obtained through vision algorithms and special markers can be obtained using trilateration and/or triangulation algorithms. Many authors are working on this topic and the same problem can be solved using a variety of different approaches. [6] has presented a method based on an Extended Kalman Filter with a state-vector composed of the external angular measurements. [7] has presented a simple, fast and new three object triangulation algorithm based on the power center of three circles. [8] has presented an algorithm for automatic selection of optimal landmarks which can reduce the uncertainty by more than one order of magnitude.

This thesis must be considered as the continuation of a previous thesis work accomplished by Luís Carrão[9], where the library Libdmtx and the triangulation/trilateration algorithm have been tested. Also, an extended analysis of the localization accuracy has been accomplished.

Most common systems

This subsection gives a brief overview of which kind of systems are available on the market. One of the most innovative systems available on the industry is the *Kiva System*[10][11]: this system can coordinate hundreds of mobile robots in the same warehouse and the robot's navigation system involves a combination of dead reckoning and cameras that look for fiducial markers, that are placed on the floor during system installation.

The most commonly used systems are:

- magnetic stripe system;

- optical guided system;

- inertial navigation system;

- laser guide system;

- vision system.

Further information is provided in the following subsections.

Magnetic stripe system

The magnetic stripe system works thanks to electric current passing through a guide wire installed along the travel route on or in the floor; the AGV travels along the magnetic field produced by the current. This system is simple but any change on the route requires to remove the old magnetic stripe and to install a new one. Moreover, especially if the stripe is on the floor, constant maintenance is needed, since the stripe tends to deteriorate in time because of mechanical stress, and any breakage of the wires makes it impossible to detect the route.

Optical guided system

With the optical guided system, reflective tape made of aluminium or a similar material is laid along the travel route, and the AGV determines its route by optically detecting the tape. A common problem of this system is the difficulty to detect the tape when it is dirty or damaged.

Inertial navigation system

An inertial sensor (gyro and accelerometer) mounted on the AGV is used to measure the vehicle's attitude angle and travel distance. The current position is calculated using measurement data, and the AGV travels along the set route. Transponders embedded in the floor are used to verify that the vehicle is following the correct path. This system requires the installation of corrective markers along the paths because the error of inertial systems tend to accumulate as an integral term.

Laser guide system

A laser beam from the AGV is reflected by reflectors mounted on walls and the current position is determined by the angle (and sometimes the distance) of the reflected light, and the vehicle uses this data to travel along the set route. The collected information is compared to the map of the reflector layout stored in the AGV's memory and using a triangulation (or trilateration) algorithm the robot can calculate its position and follow its route.

Vision guide system

Vision-guided AGVs work using cameras to record features along the route. A robot that uses this system requires a map, in which the features have been previously recorded. It is possible to use different combination of cameras, for instance stereo and omnidirectional. The extraction of the features from an image requires a higher computational power in comparison to the one needed for the other systems, but this system has the advantage of not requiring any kind of landmarks, tape, wire or stripe.

1.3 Proposed solution

The proposed solution aims to use simple markers, namely stan-
dard data matrix codes (see the example in figure 1.2), to ob-
tain a "raw" pose estimation through trilateration and triangula-
tion. The idea is to encode in each data matrix the relative
pose with respect to a previously defined reference frame. Then,
knowing the length of the edge of each data matrix and the cal-
ibration parameters of the camera, it is possible to calculate the
relative pose of a given data matrix with respect to the camera
frame.
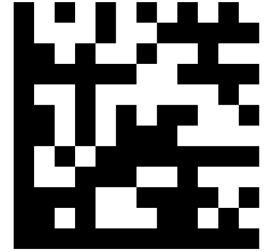
Figure 1.2: Example
of data matrix

The relative pose gives important information such as the distance between the camera and
the marker, which is necessary for the trilateraion, and the angle of arrival, necessary for the
estimation of the orientation. The "raw" pose is then fused with heterogeneous data provided by
odometry and (if present) by an IMU (Inertial Measurement Unit) using an Extended Kalman
Filter. This filter provides an estimation of the state of the AGV, which includes all information
required to perform the motion control: position, velocity and orientation.

The advantages of this solution are that it is cheap, flexible and robust: the markers are
printed on simple sheet of papers and they can therefor be easily placed inside the map with a
given acceptable margin of error. Note that this approach allows the possibility to use sophisti-
cated path planning algorithms, since the AGVs are not forced to follow a fixed path.

Chapter 2

Overview

2.1 Development platform

During this work many tools have been used. This section presents a brief introduction of the platform used, more details will be added in the following chapters.

2.1.1 ROS - Robot Operating System

ROS[12] is an open source framework used to build advanced robot applications. It was originally developed in 2007 by the SAIL (Sanford Artificial Intelligence Laboratory) and through the years it has become a de facto standard in the research field. Its appeal is growing even in the industry, thanks to the ROS-Industrial consortium. ROS is designed to be flexible, general-purpose and robust. It includes a constantly increasing number of tools, libraries and interfaces that can be reused and improved by anyone.

One of the key features of this framework is the possibility to use virtually any programming language. At this moment the main supported languages are C++, Java and Python. A ROS program is typically subdivided in two or more "nodes": which are is in fact a stand alone programs, able both to provide functionalities and to use functionalities provided by others nodes. The communication between nodes is made possible thanks to a standard common interface based on "messages". The interface is implemented over the TPC/IP protocol: this means that each message sent by a node is converted in a series of TCP/IP packet received by other nodes, even if they run in different machines connected to the same network.

2.1.2   Robotics System Toolbox

The new Robotics System Toolbox (RST)[13] has been introduced in MATLAB R2015a. This new toolbox successfully uses the Java implementation of ROS, presented by Google in 2011, in order to provide an interface between MATLAB-Simulink and ROS. This toolbox allows the rapid prototyping of algorithms and their integration directly in the ROS workspace, opening the possibility of using MATLAB and Simulink algorithms with real (or simulated) ROS-compliant robots with minimal code changes.

The following chapters will show how MATLAB and RST were widely used in order to perform simulations, to test algorithms before writing them in C++ and to analyse the collected data.

2.1.3   Libdmtx library

Libdmtx[14] is an open source software for encoding and decoding Data Matrix. This library is written in C, has a rather good performance level and a stability sufficient for the purpose of this thesis. This library is also used by many ROS packages, fox example cob-marker and visp-auto-tracker.

2.1.4   OpenCV library

OpenCV (Open Source Computer Vision)[15] is a cross-platform library mainly aimed at real-time computer vision. The library is written in C (version 1.x) and C++ (version 2.x and 3.x) and there are full interfaces in Python, Java, MATLAB and others languages. It is also the primary vision package in ROS. As explained in the next pages, in this thesis it has been used to handle the following operations:

- all geometrical transformations between different reference frames;

- the camera calibration;

- the calculation of the relative pose between data matrix and camera.

In addition to these features, a basic interface to usb cameras is provided.

2.1.5   Qt framework

Qt ("cute")[16] is a cross-platform application framework used mainly for developing application software with graphical user interfaces (GUI). It is perfectly integrated in ROS and it has been used to develop a utility which allows to:

- easily import maps (in a bitmap format);

- collocate and automatically generate data matrices containing their poses relative to a fixed reference frame;

- save and open projects;

- print or export each data matrix in .png or .pdf.

This application will be presented in the next chapter.

Figure 2.1: From left to right: OpenCV, ROS, Qt, Libdmtx, MATLAB logos.

2.2   Validation hardware platform

A real robot equipped with two cameras, one looking forward and one looking backward, has been used in order to validate the developed algorithms and to collect data. The collected information has been fundamental to understand which were the main sources of uncertainty and problems.

2.2.1   AtlasMV robot

The robot, named AtlasMV[4] (fig. 2.2a), has been developed by the University of Aveiro in 2008 to participate in robotics competitions. AtlasMV is a car-like robot with a fully functioning ROS interface. Through this interface, it is possible to control both speed and steering angle and to read their relative estimations at a frequency of 30Hz. Also, it is possible to get further information regarding the status of the robot.

2.2.2   Video capturing devices

Two Logitech® c310 cameras (fig. 2.2b) have been used for capturing visual information. These cameras are capable of capturing images with a frequency up to 30hz (depending on the exposure time) and at a resolution up to 1280x960 pixel. These cameras are not oriented to computer vision, with a good tuning it was nevertheless possible to obtain images with a sufficient quality even in non static situations.
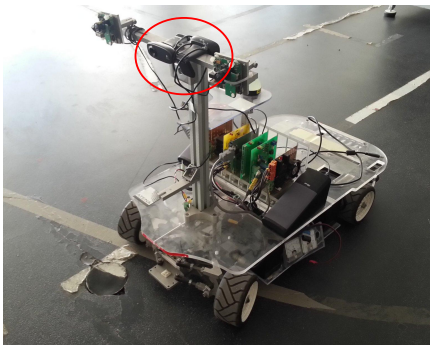
(a) AtlasMV equipped with two cameras (red cirle)

(b) Logitech® c310 usb camera

Figure 2.2

Chapter 3

Map characterization, information encoding and tools creation

3.1 Map characterization

The first problem that has been considered was how to properly prepare a given environment to allow the localization of a robot using visual information.

As a starting point it has been assumed that the map of the environment is known and available as a bitmap image. Then, the following step has been to take an arbitrary *frame* of the map as origin of the 2D Cartesian coordinate system, defined as *map frame* and indicated with $\mathbf{O}_{map}$. In analogy to the coordinate system often used in computer vision libraries (like OpenCV) the top-left corner of the image has been fixed as origin of the coordinate system, with X axis pointing right and the Y axis pointing up.

The figure 3.1 shows a partial map of LAR[1], where most of the tests with the AtlasMV have been executed. The fig.3.1 shows the *map frame* at the top-left corner of the image (the border of the image represented with a blue line).
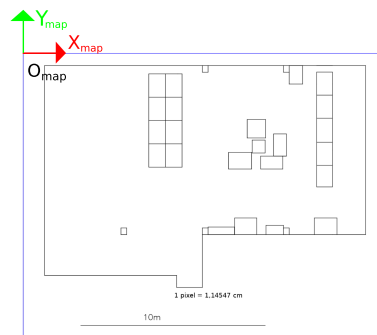
Figure 3.1: The map of LAR

3.2   Information encoding

The second problem that has been considered was that of finding the best trade-off between the amount of information encoded in a data matrix and the dimension of the symbol. This problem is related to the fact that the robot must be able to detect and decode a given data matrix from a distance of at least 4/5 meters, otherwise the localization system would be useless.

According to the last standard **ECC 200**[17][18], the symbol size can vary from 9x9 to 144x144; the table 3.1 resumes some interesting properties of the data matrices (not all the format numbers are reported for brevity).

It is reasonable to approximate the 2D coordinate system previously defined with a grid-based representation obtained by embedding the map into a discretized coordinate system with a step of **0.1m** (one order of magnitude smaller than the typical dimensions of an AGV).

As reported in the table 3.1, the minimum symbol size is 10x10 (*format number 0*) and in it 1 byte of information can be stored. A symbol with this size and printed on a A4 sheet of paper can be easily decoded[2], but 1 byte is not sufficient most in practical cases: if the byte is equally divided (4 bits for the X axis and 4 for the Y axis) then it is possible to represent only a map with a size of 1.6m x 1.6m.

The second option that was taken into consideration is the *format number 1*. With a symbol size of 12x12, the *format number 1* can store 3 bytes of data. Dividing the bytes equally (12 bits for the X axis and 12 for the Y axis) it is possible to represent a map with a size of 409.6m x 409.6m, that is more than enough for most applications.

In order to encode the orientation of a given data matrix as well as the format of the sheet of paper (for example A4 and A5, or A4 and A3), it has been chosen the following configuration:

- 10 bits for the X axis;

- 10 bits for the Y axis;

- 3 bits for the orientation (step of 45 degrees);

- 1 bit for the size (A4 and A5).

Using this method it is possible to represent a map with a size of 102.4m x 102.4m, the orientation (8 possible angles) and 1 bit for the dimension of the sheet of paper (figure 3.2). Saving 4 extra bits for the orientation and the size can be useful for future application. In this

---

[2]In practical cases, if the image is not blurred and if the light condition is good, the decoding process is possible even with a 640x480 resolution camera and from a distance up of 5 meters.
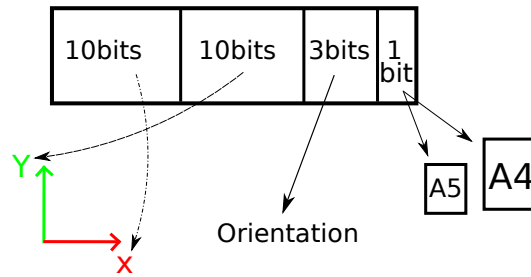
Figure 3.2: Data package encoded

thesis only the X and Y fields of the package have been used.

It is important to notice that with the chosen coordinate system, the X value is always positive, and the Y value is always negative. In order to save data, it has been chosen to encode only the absolute value of Y. The minus sign is later reintroduced during the decoding process.

Table 3.1: Data Matrix Formats

Format number	Size	Max binary capacity	% of codewords for error correction	correctable codewords
0	10x10	1	62.5	2-0
1	12x12	3	58.3	3-0
2	14x14	6	55.6	5-7
3	16x16	10	50	6-9
...	...	...	...	...
20	104x104	814	29.2	168-318
21	120x120	1048	28	204-390
22	132x132	1302	27.6	248-472
23	144x144	1556	28.5	310-590

Each codeword is represented in the data matrix by a square part of 8 modules, corresponding to 8 bits. Depending on the symbol size, there is a portion of codewords used to correct errors. The error-correction codes used are the Reed-Solomon codes[19]. For instance, the format number 2 has 58.3% of codewords dedicated to error correction and up 3 codewords can be corrected.

3.3 Application: "Datamatrix generator"

In order to provide a simple tool for generating special data matrices, it has been developed an application using the Qt framework. As anticipated in the sub section 2.1.5, this application covers all the processes of landmarks creation and is specifically dedicated to the generation of datamatrix for indoor labeling and localization.

The figure 3.3 shows how the application looks like on its first run. Six areas have been highlighted:

1. Open or save projects (a project contains the map, the scale and the list of data matrices);

2. zoom in and out the map view;

3. add a new marker with a given position and a given orientation;

4. load a map from an image (png, jpg and other formats are supported);

5. print an item (the marker is converted to a data matrix during the printing process);

6. display all the data matrices added to the map in a dedicated table.
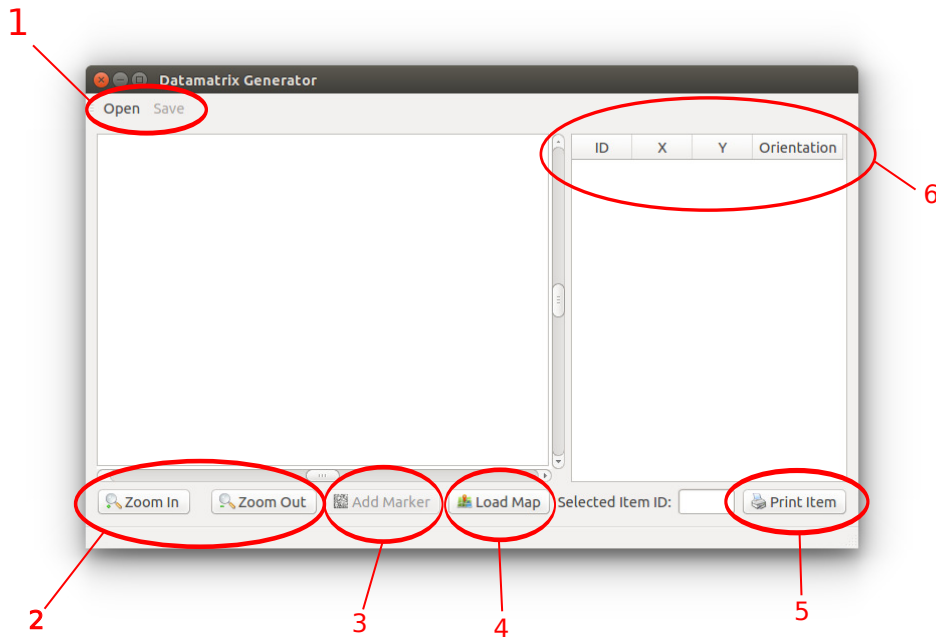
Figure 3.3: Data matrix application

By pressing the *Load Map* button, it is possible to select an image containing a map. The following step is to enter the correct scale factor (figure 3.4).
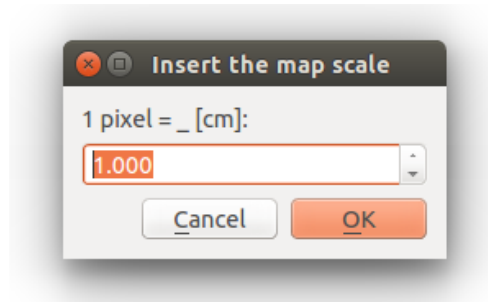
Figure 3.4: Set the scale factor.

The figure 3.5 shows how the creation and the positioning of a given data matrix works: each data matrix can be moved by using the drag and drop functionality or using the *Edit item* window, which opens with a double-click on the item. Using the *Edit item* window it is also possible to change the orientation or to delete a given item.
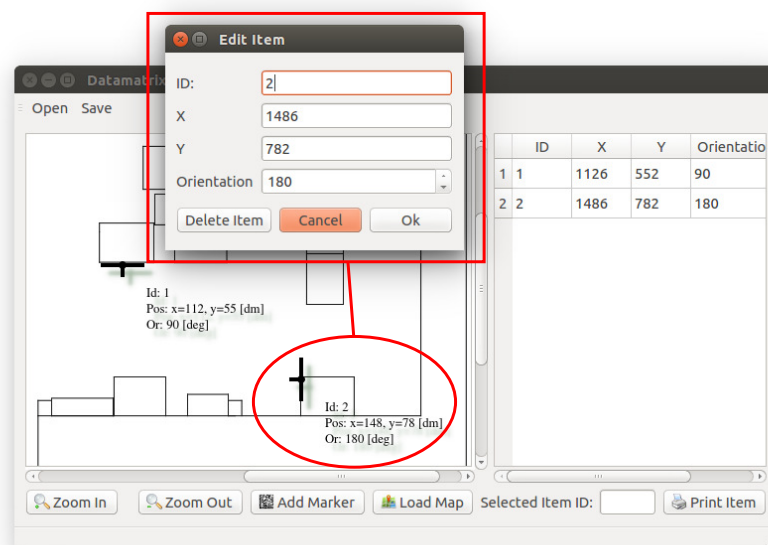
Figure 3.5: Add a new item.

As last step it is possible to print an item listed on the table by clicking the *Print* button.
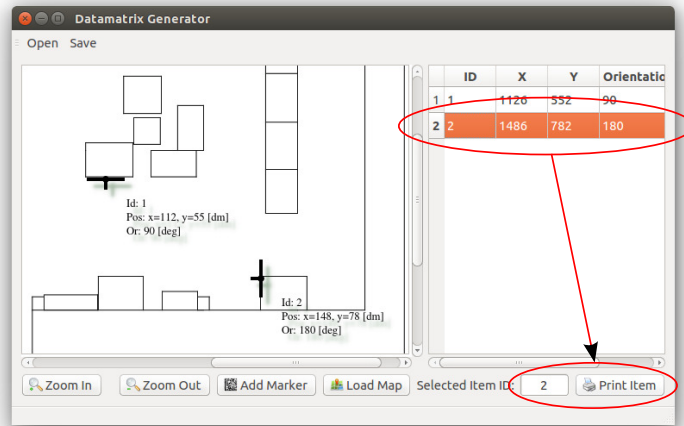
Figure 3.6: Select an item.

By clicking on the *Print* button, the data matrix will be automatically created and shown on a new window; it is then possible to *Export* or *Print* it using the relative buttons.
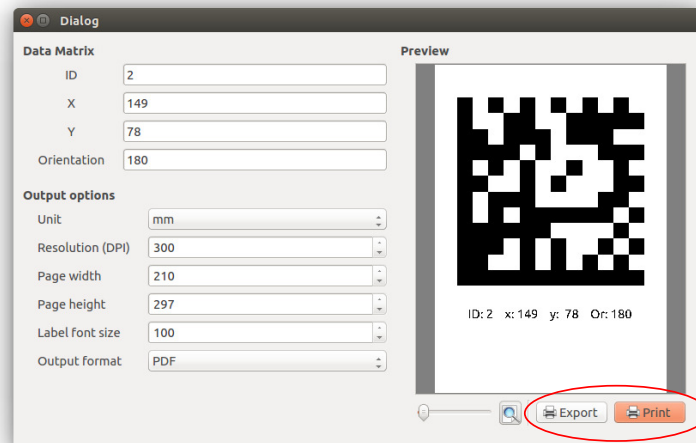
Figure 3.7: Print and export an item.

Note: the development of this software is still in progress. The GUI (*Graphical User Interface*) must be considered as a draft and some interesting features are still missing. For instance, it would be useful to have a support for CAD format files (such as dxf files).

Chapter 4

Perception

This chapter covers the topics relative to *perception*: from camera calibration to the calculation of the relative pose of a camera according to a data matrix.

4.1 Pinhole camera

The *pinhole camera model*[20] describes the mathematical relationship between the coordinates of a 3D point and its projection onto the *image plane* of an ideal *pinhole camera* (figure 4.1). This model is only a first order approximation of how a modern camera works and it does not take into account the effects introduced by the lens, the finite size apertures and other non-ideal parameters of a real camera. This model is widely used in computer vision applications because it is simple and because many of its limitations can be compensated by the software after a calibration process.

4.1.1 Pinhole camera model

The pinhole camera model (the geometry is reported in figure 4.2) is composed of a 3D orthogonal coordinate system, which is used to represent the coordinate of a point in the 3D space with respect to the camera frame, and a 2D orthogonal coordinate system, which represents the
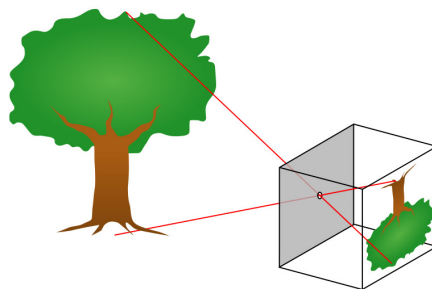
Figure 4.1: Pinhole camera diagram

projection of that point onto a plane called *image plane*.

The origin **O** of the 3D orthogonal coordinate system is located where the aperture is located (or in the geometric center of the aperture in case of a real camera with finite size aperture), the **X3** axis is pointing in the viewing direction of the camera and it refers to its *optical axis* (or *principal axis*). The **X1** and **X2** axes locate the *principal plane*.

The *image plane* is parallel to the axes **X1** and **X2** and it is located at a distance f from the origin **O** in the negative direction of the **X3** axis. The value f is referred to as the *focal length*. The point **R** is placed at the intersection of the optical axis and the image plane and it is called *principal point* (or *image center*).

A given point **P** at coordinate (x_1, x_2, x_3) relative to the axes **X1**, **X2**, **X3** projected onto the image plane is defined as **Q**. The coordinate of the point **Q** relative to the 2D coordinate system are expressed as (y_1, y_2).
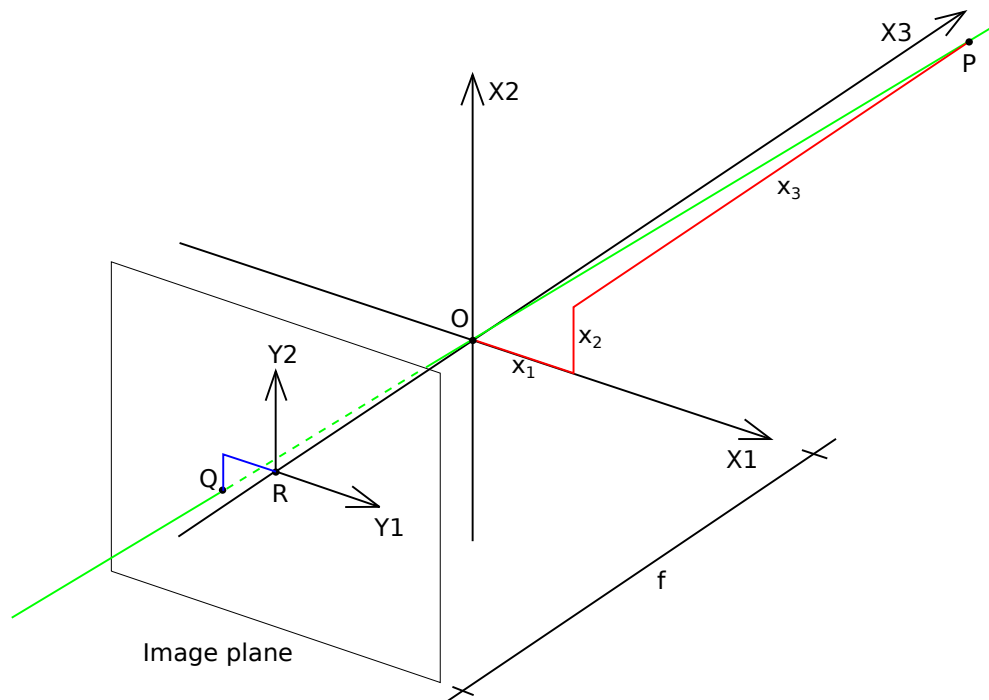
Figure 4.2: Pinhole model

The relation between the 3D coordinates (x_1, x_2, x_3) of point $\mathbf{P}$ and its image coordinates (y_1, y_2) given by point $\mathbf{Q}$ in the image plane is expressed by

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{f}{x_3} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}. \tag{4.1}$$

Note that the image in the *image plane* is rotate by π. In order to produce an unrotated image, it is useful to define a *virtual plane* so that it intersects the $\mathbf{X3}$ axis at f instead of $-f$. The resulting mapping from 3D to 2D coordinates is given by

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \frac{f}{x_3} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}. \tag{4.2}$$

The relation 4.2 can't provide the 3D coordinate of a point from its 2D projection but only the ratios $\frac{x_1}{x_3}$ and $\frac{x_2}{x_3}$ (fig.4.3): which means that with a single camera it is impossible to determine the 3D coordinate of a point.
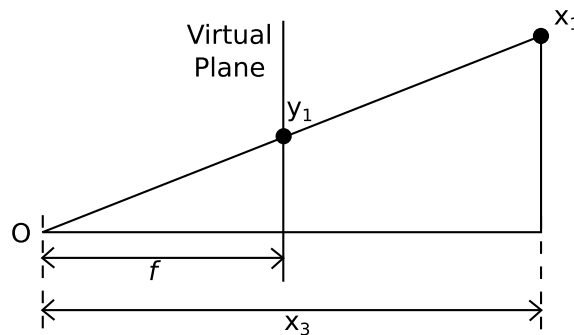
Figure 4.3: Relation between $\frac{y_1}{f}$ and $\frac{x_1}{x_3}$

In the general case, in order to determine the position of a given point it is necessary to use a *stereo vision system* and the *epipolar geometry*[21]. In the particular case that has been considered, it has not been necessary to use more than one camera: knowing the geometry of a data matrix, which is approximable to a square with a given edge length, it is possible to calculate its pose with only one camera if at least three correspondences between image points and objects points have been identified[22].

4.1.2 Camera calibration

The calibration process allows the identification of the *intrinsic parameters* and *distortion coefficients* of the camera. Without this step it is impossible to obtain a good accuracy, especially when the lens introduce a high distortion.

The intrinsic parameters are necessary for defining the *camera matrix*, that is

$$\mathbf{K} := \mathbf{K_s}\mathbf{K_f} = \begin{bmatrix} s_x & s_\theta & O_x \\ 0 & s_y & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} fs_x & fs_\theta & O_x \\ 0 & fs_y & O_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.3}$$

where

- s_x and s_y are scale factors;

- s_θ is the skew factor;

- O_x and O_y are the offsets of the central point;

- f is the focal length.

Using the *homogeneous coordinates*, the camera matrix defines the relation

$$\lambda \begin{bmatrix} y_1 \\ y_2 \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \lambda \neq 0. \tag{4.4}$$

The distortion coefficients are used to compensate the radial and tangential distortion introduced by lens. Figure 4.4 shows distorted chessboard (figure 4.4a) during the calibration procedure and the undistorted chessboard after the calibration (figure 4.4b). The software used for the calibration is the ROS package *camera-calibration*[23].
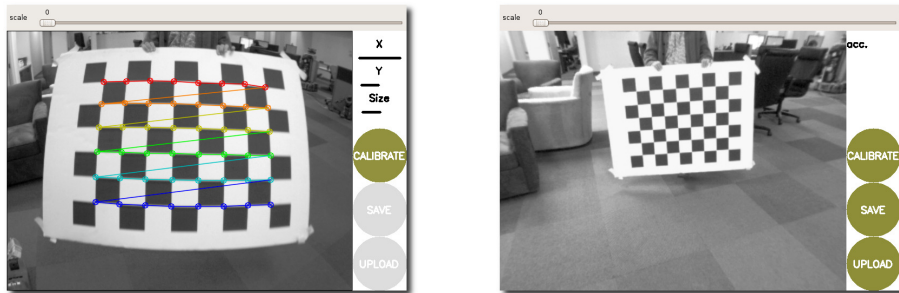
(a) A chessboard during the calibration   (b) A chessboard after the calibration

Figure 4.4: Camera calibration process

4.2 Data Matrix pose with respect to the camera frame

In the previous chapter the pinhole model and the concept of camera calibration were discussed. This section presents the approach used for obtaining the pose of a given Data Matrix with respect to the camera frame, defined as $\mathbf{O}_{cam}$.

The library Libdmtx allows to obtain extra information regarding each detected data matrix code, including the pixel coordinate of each corner. The figure 4.5a shows an example of detection: the green circles indicate the corners detected by the library (the red circle is simply the middle point between the top-left corner and the bottom-right).

Note that the data matrix does not have the top-right corner: Libdmtx, in fact, calculates the coordinates of the corners of a square that fits the detected data matrix.

Let $\mathbf{O}_{dm}$ be the reference frame attached to the data matrix (figure 4.5b) and let l be the length of the data matrix edges. Then, the coordinates of the corners detected by the Libdmtx library with respect to the reference frame $\mathbf{O}_{dm}$ are:

- top-left corner: $C_{tl}^{dm} := (-\frac{l}{2}, \frac{l}{2}, 0)$;

- top-right corner: $C_{tr}^{dm} := (\frac{l}{2}, \frac{l}{2}, 0)$;

- bottom-left corner : $C_{bl}^{dm} := (-\frac{l}{2}, -\frac{l}{2}, 0)$;

- bottom-right corner: $C_{br}^{dm} := (\frac{l}{2}, -\frac{l}{2}, 0)$.

Let the respective corners in the image coordinates be:

- top-left corner: C_{tl}^{img};

- top-right corner: C_{tr}^{img};

(a) Example of data matrix detection
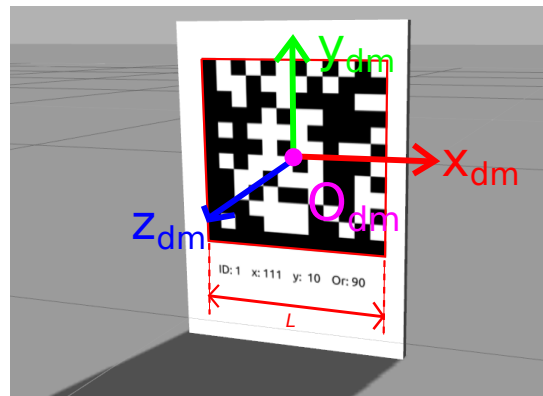
(b) Frame $\mathbf{O}_{dm}$ attached to data matrix

Figure 4.5: Data matrix detection and reference frame $\mathbf{O}_{dm}$

- bottom-left corner : C_{bl}^{img};

- bottom-right corner: C_{br}^{img}.

The OpenCV function *solvePnP*[24] was then used; this function allows to find the data matrix pose using the 3D-2D correspondences of the previously defined corners (figure 4.6). The function requires the following inputs:

- a Vector containing the 2D coordinates (image points): $\left[C_{tl}^{img}, C_{tr}^{img}, C_{bl}^{img}, C_{br}^{img} \right]$;

- a Vector containing the 3D coordinates (object points): $\left[C_{tl}^{dm}, C_{tr}^{dm}, C_{bl}^{dm}, C_{br}^{dm} \right]$;

- the calibration file of the camera containing the *camera matrix* and the *distortion coefficients*.

The outputs of the function are:

- the rotation matrix R_{dm}^{cam};

- the translation vector T_{dm}^{cam};

that are the pose of the data matrix with respect to the camera frame $\mathbf{T}_{dm}^{cam} = (R_{dm}^{cam}, T_{dm}^{cam})$.

The function *solvePnP* includes three different algorithms for computing the pose. The algorithm has been used is the iterative method based on Levenberg-Marquardt optimization [25][26].

The Levenberg-Marquardt optimization algorithm minimize the reprojection error, that is, the total sum of squared distances between the observed feature points *imagePoints* and the projected object points *objectPoints*.
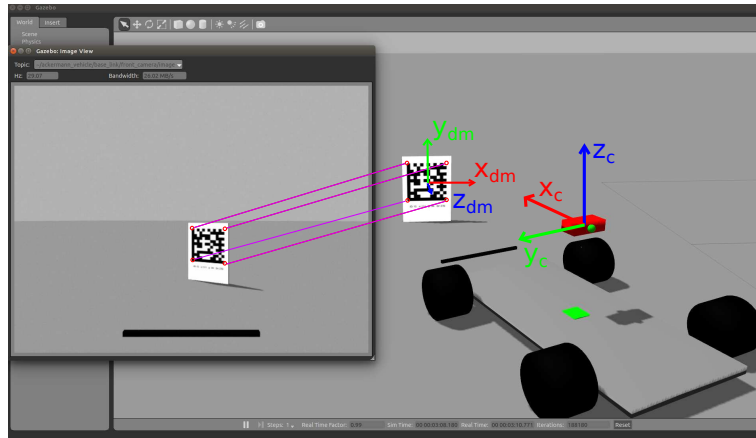
Figure 4.6: Correspondence between 2D corners e 3D corners

4.3   Data matrix pose with respect to the robot frame

One of the aims of the localization algorithm was to use two or more cameras to scan a wider area around the robot. In order to simplify the localization algorithm, it has been necessary to calculate the pose of each data matrix with respect to a given reference frame.

The adopted approach makes the whole algorithm *modular*: the part of the algorithm that estimates the position doesn't have to be aware of how many cameras the robot employs but only it needs to know the pose of each data matrix in relation to that given reference frame.

The natural choice has been to define the *robot frame* $\mathbf{O}_r$ attached at the geometric center of the robot (fig. 4.7a) and to consider all the poses referred to this frame.

Let be

- $\mathbf{T}^{c_i}_{dm_j} = \left(R^{c_i}_{dm_j}, T^{c_i}_{dm_j} \right)$ the pose of the data matrix j with respect to the *camera i frame*;

- $\mathbf{T}^{r}_{c_i} = \left(R^{r}_{c_i}, T^{r}_{c_i} \right)$ the pose of the camera i with respect to the *robot frame*;

- $\mathbf{T}^{r}_{dm_j} = \left(R^{r}_{dm_j}, T^{r}_{dm_j} \right)$ the pose of the data matrix j with respect to the *robot frame*.

The pose $\mathbf{T}^{r}_{dm_j}$ can be calculated using the following relations

$$R^{r}_{dm_j} = R^{r}_{c_i} \cdot R^{c_i}_{dm_j}, \qquad T^{r}_{dm_j} = R^{r}_{c_i} \cdot T^{c_i}_{dm_j} + T^{r}_{c_i}. \tag{4.5}$$

Note that the pose $\mathbf{T}^{r}_{c_i}$ must be calculated through a calibration process as detailed in the following paragraph.
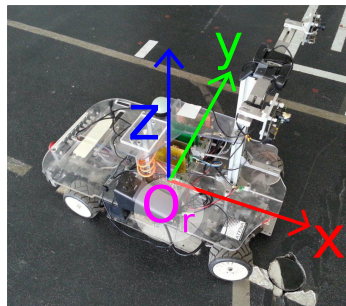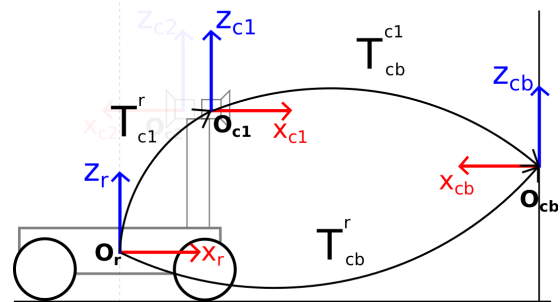
(a) Robot frame $\mathbf{O}_r$ (b) Relation between reference frames

Figure 4.7

Extrinsic camera parameters

The calculation of the pose $\mathbf{T}^r_{c_i}$ has required to develop a specific calibration process since only the pose of $\mathbf{T}^{c_i}_{dm_j}$ is computed and the direct calculation of the pose $\mathbf{T}^r_{c_i}$ (especially the orientation) is not easy in practice.

The calibration process uses a chessboard situated in a known position with respect to the robot frame $\mathbf{O}_r$. The pose of the chessboard with respect to the camera i frame $\mathbf{T}^{c_i}_{cb} := \left(R^{c_i}_{cb}, T^{c_i}_{cb} \right)$ has been calculated using the built-in OpenCV function *findChessboardCorners* (in order to get the corners coordinates) and then again the function *solvePnP*.

The relations that were used are the following:

$$R^r_{c_i} = R^r_{cb} \cdot \left(R^{c_i}_{cb} \right)^{-1} , \tag{4.6}$$

$$T^r_{c_i} = T^r_{cb} - R^r_{c_i} \cdot T^{c_i}_{cb} = T^r_{cb} - R^r_{cb} \cdot \left(R^{c_i}_{cb} \right)^{-1} \cdot T^{c_i}_{cb} . \tag{4.7}$$

Let $\mathbf{O}_{c_1}$ be the coordinate system of the camera that looks forward and $\mathbf{O}_{c_2}$ the coordinate system of the camera that looks backward. The figure 4.7b shows the relation between the reference frames $\mathbf{O}_r$, $\mathbf{O}_{c_1}$ and $\mathbf{O}_{cb}$.
In the example shown in figure 4.8, the chessboard has been positioned in a specific point such that the resulting pose $\mathbf{T}^r_{cb}$ (measured with a measuring tape) was

$$T^r_{cb} = \begin{bmatrix} 1.54 \\ 0 \\ 0.75 \end{bmatrix} [meters] , \quad R^r_{cb} = \begin{bmatrix} \cos \pi & -\sin \pi & 0 \\ \sin \pi & \cos \pi & 0 \\ 0 & 0 & 1 \end{bmatrix} . \tag{4.8}$$

The calibration has been executed using a C++ program developed for this task and the pose $\mathbf{T}^r_{c_i}$ has been calculated using the relations 4.6 and 4.7 and the OpenCV function *composeRT*[24].
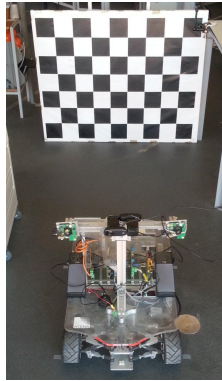
Figure 4.8: Extrinsic parameters calibration

4.4   Implementation of the algorithm

The algorithm described in this chapter has been implemented as a single ROS node called *datamatrix-pose-pub*. This node includes the following components:

- an interface to the cameras, using OpenCV's APIs;

- a data matrix detector, using Libdmtx;

- the computation of $\mathbf{T}^{c_i}_{dm_j}$, for each camera i and each data matrix j, using *solvePnP*;

- the computation of $\mathbf{T}^r_{dm_j}$, using *composeRT*.

The figure 4.9 is a schematic of the internal architecture of this node.
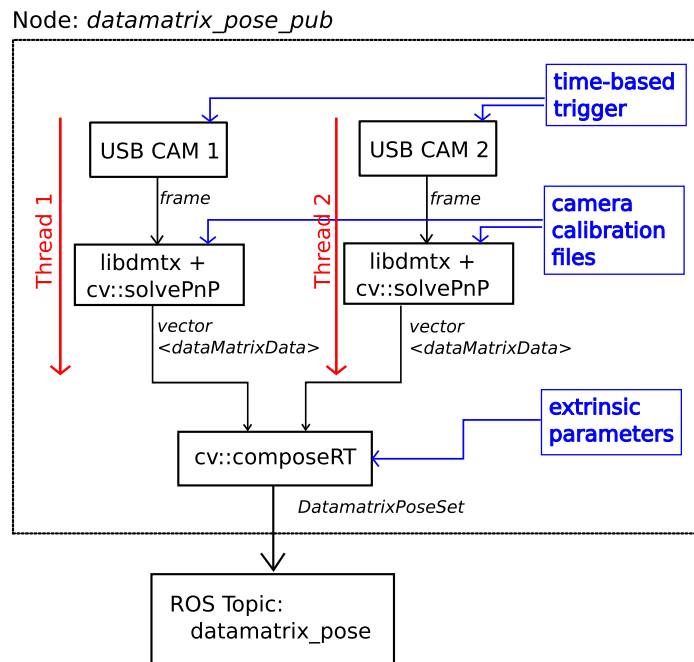
Figure 4.9: Structure of the node datamatrix-pose-pub

The frames are captured using the same target, timestamped for both cameras. This has been necessary for two reasons:

- to keep track of the acquisition timestamp (this information is important because the delay introduced by the datamatrix detection process must be taken into account);

- to avoid errors introduced when the localization algorithm attempts to estimate the position of the robot using a couple of data matrices detected in two different frames with different timestamps.

The frame is elaborated by the Libdmtx library and the pose of each detected data matrix is calculated using the proper calibration file and the OpenCV function *solvePnP*.

The output of this procedure is a vector of *dataMatrixData*, which is a special struct defined as

```
struct dataMatrixData{
 /* Encoded Information */
 unsigned char byte0, byte1, byte2;

 /* Dmtx region information */
 float topleftx;  float toplefty;
 float toprightx;  float toprighty;
 float bottomleftx;  float bottomlefty;
 float bottomrightx;  float bottomrighty;

 /* Datamatrix Encoded Information */
 int x, y, theta, size;

 /* ROI (rectangle) */
 float roiX;
 float roiY;
 float roiWidth;
 float roiHeight;

 /* Datamatrix Center */
 float centerX;
 float centerY;

 /* Datamatrix Pose */
 dataMatrixPose dmPose;
};
```

and the type *dataMatrixPose* is another struct defined as

```
struct dataMatrixPose{
float tx; float ty; float tz; // translation
float rx; float ry; float rz; // rotation
};
```

which contains the pose $\mathbf{T}^{c_i}_{dm_j}$.

The procedure can be distributed to be implemented on multiple cases: the cameras can grab and elaborate frames simultaneously on two different CPUs and the resulting elaboration process is twice as fast.

Reducing the elaboration time has been a critical step: if the elaboration takes too much time (for instance $250ms$) the poses of the data matrices detected will be, in fact, referred to a data matrix detected $250ms$ in the past. Considering a speed of $2m/s$ and an elaboration time of $250ms$, the resulting error is $0.5m$.

The vectors relative to each elaboration thread are finally processed sequentially using the function *composeRT* and the extrinsic parameters of each camera. The resulting output is a data structure called *DataMatrixPoseSet*, which is a custom ROS message with the following structure

```
Header header
float64 acq_timestamp
datamatrix_detection/DatamatrixPose[] dmpose
```

where *header* is a ROS data type used for managing the messages, *acq-timestamp* is the acquisition timestamp and *dmpose* is a vector of *DatamatrixPose*, which is a custom ROS message defined as

```
float64 dm_msg_x
float64 dm_msg_y
float64 dm_msg_or
float64 dm_msg_size
geometry_msgs/PoseWithCovariance pose
```

The message *DatamatrixPoseSet* is finally published on the ROS topic *datamatrix-pose* at a frequency between 10Hz and 15Hz. This frequency has been determined by the computational power of CPU used, an Intel® Core™ i3-2310m (dual-core processor with a frequency up to $2,1GHz$).

The messages published on this topic are read by the ROS node named *atlasmv-ekf*, which is presented in the chapter 5.

Chapter 5

Estimation of position and sensor fusion

This chapter covers the part of the thesis regarding position estimation and sensor fusion of the moving robot.

The covered topics are:

- localization using trilateration and triangulation techniques;

- AtlasMV modelling using a bicycle-like model;

- sensor fusion using an Extended Kalman Filter;

- model verification using Simulink.

5.1 Localization using trilateration and triangulation

In the context of this problem, the robot should be able to estimate its own position in the environment every time at least two data matrices are detected by its own camera system.

Let be:

- *raw-robot-pose*, the pose of the robot obtained using only the visual information at a given time t;

- *ekf-robot-pose*, the filtered pose of the robot obtained using the Extended Kalman Filter.

This section presents how *raw-robot-pose* is calculated.

As seen in the previous chapter, the *datamatrix-pose-pub* node provides a vector containing the poses of the $j - th$ data matrix with respect to the robot frame, $\mathbf{T}^r_{dm_j}$: the poses are referred to the 3D coordinate frame of the robot, $\mathbf{O}_r$. On the contrary, the estimated position is referred to the coordinate system of the map, $\mathbf{O}_{map}$.

Assuming that the robot is moving on a plane domain and hence the axes x_r and y_r of the robot frame $\mathbf{O}_r$ are parallel to the axes x_{map} and y_{map} of the map frame $\mathbf{O}_{map}$, it is possible to simply ignore the z_r axis and to keep on working on a 2D coordinate system. In this context, the robot pose with respect to the map frame $\mathbf{O}_{map}$ can be determined using the coordinates (x, y, θ)(fig. 5.1), where x and y are the coordinates with respect to the axes x_{map} and y_{map}, and θ is the orientation (rotation angle with respect to the z axis).

Let dm_j be the $j - th$ detected data matrix. The localization algorithm uses the following information extracted from the *dataMatrixData* structure:

- x_j^{enc} and y_j^{enc}, the encoded pose of the data matrix with respect to the map frame;

- $\mathbf{T}_{dm_j}^{r,2D} := (x_j, y_j)$, 2D translation vector extracted from $\mathbf{T}_{dm_j}^r = (x_j, y_j, z_j)$;

To keep the notation light, the vector $\mathbf{T}_{dm_j}^r$ will indicate both $\mathbf{T}_{dm_j}^r$ and $\mathbf{T}_{dm_j}^{r,2D}$.

If dm_1 and dm_2 are two detected data matrices, it is possible to define the distance and the angle with respect to the robot frame using the relations:

$$d_j := \sqrt{x_j^2 + y_j^2}, \quad \alpha_j = \arctan 3\left(\frac{y_j}{x_j}\right), \quad j = 1, 2 \tag{5.1}$$

where $\arctan 3\,(\cdot)$ is an $\arctan\,(\cdot)$ function defined in $[0, 2\pi]$.

Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be the circumferences with center in (x_1^{enc}, y_1^{enc}) and (x_2^{enc}, y_2^{enc}), and with radius d_1 and d_2, respectively. The robot position (x, y) is in one of the intersection points $\mathbf{P}_1$ and $\mathbf{P}_2$ (fig. 5.2) of the circumferences (the cases with one or zero intersections are simply ignored by the algorithm).
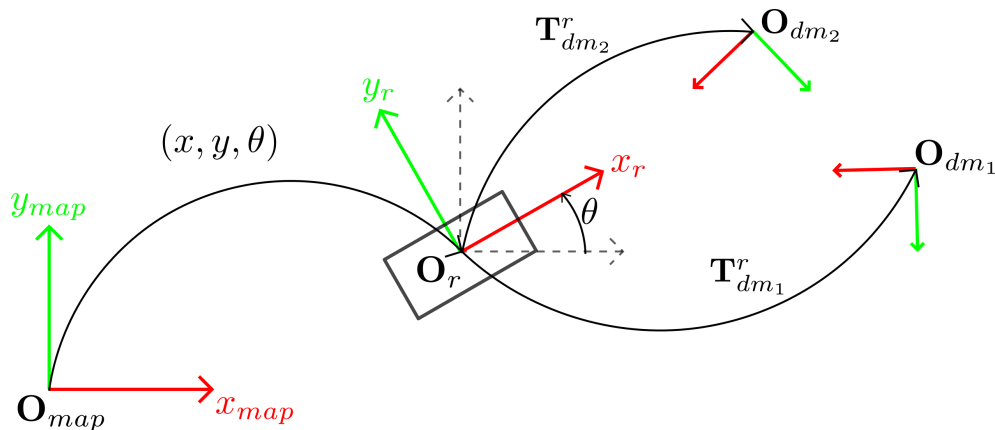
Figure 5.1: Robot pose (x, y, θ).
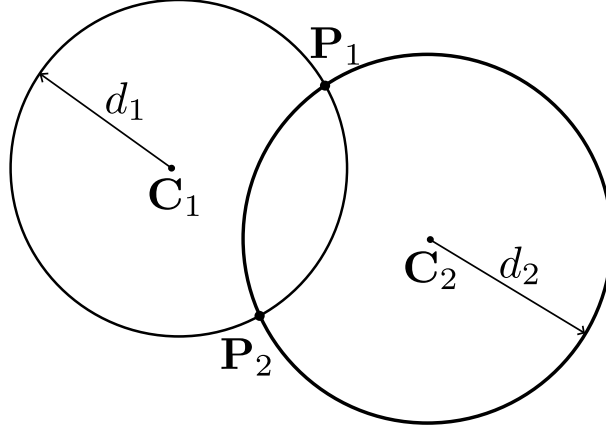
Figure 5.2: Intersection points $\mathbf{P}_1$ and $\mathbf{P}_2$ of the circumferences

It is easy to ascertain which position is the correct one, namely through a simple comparison between the measured angles α_j and the angles obtained using the coordinates points $\mathbf{P}_1$ and $\mathbf{P}_2$ and the coordinates x_j^{enc} and y_j^{enc}, with $j = 1,2$.

Finally, the orientation θ can be calculated using a single data matrix dm_j, the estimated position (x, y), α_j and $\mathbf{T}_{dm_j}^r$:

$$\theta_j = \arctan 3 \left(\frac{y_j - y}{x_j - x} \right) - \alpha_j \, . \tag{5.2}$$

If $n >= 2$ is the number of detected data matrices, the estimation of the orientation θ can be improved using the mean angle:

$$\theta = \arctan 3 \left(\frac{\sum_{k=1}^{n} \sin (\theta_j)}{\sum_{k=1}^{n} \cos (\theta_j)} \right) \, . \tag{5.3}$$

In order to use Extended Kalman Filter it is necessary to know the error associated to the pose estimation.

It was possible to identify three main sources of uncertainties:

- the discretization error u_j^m in the encoded coordinates $\left(x_j^{enc}, y_j^{enc} \right)$;

- a proportional error u_j^d in the distance d_j;

- an error u_j^α in the estimation of the angle α_j.

The uncertainties have been modelled with the Gaussian distributions:

- $u_j^m \sim \mathcal{N}(0, 0.1^2)$, standard deviation equal to the map resolution;

- $u_j^d \sim \mathcal{N}(0, (0.05 d_j)^2)$,   standard deviation equal to the distance d_j multiplied by the coefficient 0.05;

- $u_j^\alpha \sim \mathcal{N}(0, (\frac{5\pi}{180})^2)$,   standard deviation of 5 degrees.

It has been assumed by hypothesis that the uncertainties are uncorrelated.

Let

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = p \left(\begin{bmatrix} x_j^{enc} \\ y_j^{enc} \\ d_j \\ \alpha_j \end{bmatrix}, \quad j = 1, ..., n \right) \tag{5.4}$$

be the function that calculates the estimation of the pose.

The resulting error has been calculated numerically using the first order propagation of uncertainty formula:

$$u \left(\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \right) = \sum_{j=1}^{n} \sqrt{ \left(\frac{\partial p}{\partial x_j} \right)^2 (u_j^m)^2 + \left(\frac{\partial p}{\partial y_j} \right)^2 (u_j^m)^2 + \left(\frac{\partial p}{\partial d_j} \right)^2 (u_j^d)^2 + \left(\frac{\partial p}{\partial \alpha_j} \right)^2 (u_j^\alpha)^2 }$$

$$, \quad j = 1, ..., n. \tag{5.5}$$

Let

$$\mathbf{e} = \begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} := u \left(\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \right) \tag{5.6}$$

be the error vector associated to the estimation of the pose. The measurement obtained at the time t can be written as

$$\mathbf{y}_v(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}, \quad cov\left(\mathbf{y}_v(t) \cdot \mathbf{y}_v^T(t) \right) := \begin{bmatrix} e_x(t) & 0 & 0 \\ 0 & e_y(t) & 0 \\ 0 & 0 & e_\theta(t) \end{bmatrix}, \tag{5.7}$$

where the subscript v stands for *vision*.

5.2   AtlasMV modelling using a bicycle-like model

The robot has been modelled using a simple bicycle-like model. This model provides a basic kinematics, which is sufficient for filtering and estimation purposes.

According to chapter 3, the robot lies in the map with a Cartesian coordinate system and its pose is defined by (x, y, θ).

The discrete-time state-space description of its kinematics can be expressed using the following relation:

$$\begin{bmatrix} x(t_{k+1}) \\ y(t_{k+1}) \\ \theta(t_{k+1}) \end{bmatrix} = \begin{bmatrix} x(t_k) + \Delta t\, s(t_k) \cos(\theta(t_k)) \\ y(t_k) + \Delta t\, s(t_k) \sin(\theta(t_k)) \\ \theta(t_k) + \Delta t\, s(t_k) \frac{\tan(\psi(t_k))}{l} \end{bmatrix} \tag{5.8}$$

where:

- l is the wheelbase length;

- $\psi(t_k)$ is the steering angle;

- $s(t_k)$ is the speed;

- $\Delta t = t_{k+1} - t_k$ is the sample time;

- $t_k \in \mathbb{Z}$ is the time index.

This model is used as starting base to design the Extended Kalman Filter. Note that $s(t)$ and $\psi(t)$ are measurements provided by the ROS node *atlasmv*, which will be described in section 6.
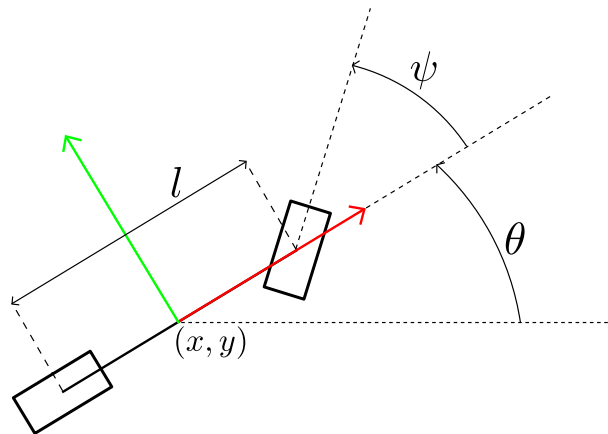
Figure 5.3: Bicycle model

5.3   Sensor fusion using an Extended Kalman Filter

This section introduces the problem of how to obtain sensor fusion using an Extended Kalman Filter. The goal is to obtain a filtered pose, namely *ekf-robot-pose*, which can be used by a client node for controlling and path planning purposes.
The topics of this section are:

- introduction to the Extended Kalman Filter framework;

- sensor fusion using visual and odometry information;

- sensor fusion using visual, odometry and inertial information;

5.3.1   Extended Kalman Filter framework

At first it is necessary to recall the equations of the Extended Kalman Filter framework. The notation used in sections 5.3.2 and 5.3.3 follows the notation introduced in this section.
Let $t \in \mathbb{R}$ be the t continuous-time variable. The reference model is

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t)) + \boldsymbol{\xi}(t) \tag{5.9}$$

$$\mathbf{y}(t_k) = h(\mathbf{x}(t_k)) + \mathbf{w}(t_k) \tag{5.10}$$

where:

- $\mathbf{x}(t)$ is the n dimensional state;

- $f(\cdot)$ is the process function;

- $h(\cdot)$ is the observing function;

- $\mathbf{x}(t_0) = \mathbf{x}_0$ is the initial state;

- $\{\boldsymbol{\xi}(t)\}$ is the continuous-time n-dimensional process noise, with zero mean and covariance matrix $Q = Q^T \geq 0$;

- $\{\mathbf{w}(t_k); k = 0, 1, ...\}$ is the continuous-time and m-dimensional observation noise, with zero mean and variance $R > 0$;

- $\{\boldsymbol{\xi}(t)\}$, $\{\boldsymbol{w}(t_k)\}$ and $\mathbf{x}(t_0)$ are uncorrelated;

- f, h, Q, R are, in general, time-varying;

- $t_k, k = 0,1,...$ is the sample time, in general aperiodic.

Let be $\bar{\mathbf{x}}(t)$ the reference state trajectory, the Jacobian matrices are:

$$F(\bar{\mathbf{x}}(t)) := \frac{\partial f}{\partial x}\bigg|_{\mathbf{x}=\bar{\mathbf{x}}(t)}, \tag{5.11}$$

$$H(\bar{\mathbf{x}}(t)) := \frac{\partial h}{\partial x}\bigg|_{\mathbf{x}=\bar{\mathbf{x}}(t)}. \tag{5.12}$$

The algorithm works in two steps: *prediction* and *update.*

Prediction step

Assuming that the dynamic of the robot is slow with respect to the sampling time, it is possible to calculate the a priori state using the Euler discretization:

$$\hat{\mathbf{x}}(k+1|k) := \hat{f}_k(\hat{\mathbf{x}}(k|k)), \tag{5.13}$$

in the interval $[t_k, t_{k+1}]$, where:

$$\hat{f}_k(\hat{\mathbf{x}}(k|k)) = \hat{\mathbf{x}}(k|k) + (t_{k+1} - t_k)f(\hat{\mathbf{x}}(k|k)). \tag{5.14}$$

The a priori variance is:

$$P(k+1|k) = \hat{\Phi}(k|k)P(k|k)\hat{\Phi}^T + Q(k), \tag{5.15}$$

where:

$$\hat{\Phi}(k|k) = \frac{\partial \hat{f}_k}{\partial x}\bigg|_{\mathbf{x}=\hat{\mathbf{x}}(k|k)}. \tag{5.16}$$

Update step

The a posteriori state is:

$$\hat{\mathbf{x}}(k+1|k+1) = \hat{\mathbf{x}}(k+1|k) + L(k+1)\left[\mathbf{y}(k+1) - h(\hat{\mathbf{x}}(k+1|k))\right], \tag{5.17}$$

where the gain $L(k+1)$ is calculated using:

$$\Lambda(k+1) = \hat{H}(k+1|k)P(k+1|k)\hat{H}(k+1|k)^T + R(k), \tag{5.18}$$

$$L(k+1) = P(k+1|k)\hat{H}(k+1|k)^T\Lambda(k+1)^{-1}, \tag{5.19}$$

where

$$\hat{H}(k+1|k) := H(\hat{\mathbf{x}}(t_{k+1}|t_k)). \tag{5.20}$$

The a posteriori variance is:

$$P(k+1|k+1) = [I - L(k+1)\hat{H}(k+1|k)]P(k+1|k)[I - L(k+1)\hat{H}(k+1|k)]^T$$
$$+ L(k+1)RL(k+1)^T . \quad (5.21)$$

Initial values

The initial values of the filter are:

$$\hat{\mathbf{x}}(0|-1) = E[\mathbf{x}(t_0)] \quad P(0|-1) = Var(\mathbf{x}(t_0)) . \quad (5.22)$$

Innovation for angular quantities

The third state variable that will be defined in the following section is the orientation $\theta \in [0,2\pi]$, which is an angular quantity. The equation 5.17 needs to be modified in order to avoid unexpected behaviours.
Let

$$I(k+1) := \mathbf{y}(k+1) - h(\hat{\mathbf{x}}(k+1|k)) \quad (5.23)$$

be the innovation vector and let $i_3(k+1)$ be the third component of this vector.
Let $y_3(k+1)$ and $h_3(\hat{\mathbf{x}}(k+1|k))$ be the third component of $\mathbf{y}(k+1)$ and $h(\hat{\mathbf{x}}(k+1|k)$, respectively. For instance, if $y_3(k+1) = 0.1$ and $h_3(\hat{\mathbf{x}}(k+1|k)) = 6.2$, the respective innovation is $i_3(k+1) = -6.1$: this makes the filter unstable.

The innovation $i_3(k+1)$ must be calculated using

$$i_3(k+1) = atan2 \left(\frac{sin(y_3(k+1) - h_3(\hat{\mathbf{x}}(k+1|k)))}{cos(y_3(k+1) - h_3(\hat{\mathbf{x}}(k+1|k)))} \right) , \quad (5.24)$$

where $atan2(\cdot)$ is the four-quadrant inverse tangent defined in $[-\pi, \pi]$.
After this correction the third state variable must be remapped into the interval $[0, 2\pi]$ using the function $atan3(\cdot)$.

5.3.2   Sensor fusion using visual and odometry information

This section introduces a model for the sensor fusion using the pose calculated by the node *datamatrix-pose-pub*, as discussed in the chapter 4 and using the the speed $s(\cdot)$ and the steering angle $\psi(\cdot)$ published by the *atlasmv-base* node on the topic *atlasmv-base/AtlasmvStatus*[1].

Model 1: equations

Let

$$\mathbf{x}(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \\ s(t) \\ \psi(t) \end{bmatrix} \tag{5.25}$$

be the state vector.

The $f(\cdot)$ function is defined as:

$$f(\mathbf{x}(t)) = \begin{bmatrix} s(t)\cos\theta(t) \\ s(t)\sin\theta(t) \\ s(t)\tan(\psi(t))\frac{1}{l} \\ 0 \\ 0 \end{bmatrix}, \tag{5.26}$$

where the components 1, 2, 3 are the continuous equivalent of the 5.8 and the components 4 and 5 are set to zero because the acceleration $\dot{s}(\cdot)$ and the steering angular velocity $\dot{\psi}(\cdot)$ are not measurable without an IMU. The dynamic of the components 4 and 5 is determined by the process noise.

Since we have two different sources of information, there are also two observation functions:

$$h_v(\mathbf{x}(t)) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}, \quad h_a(\mathbf{x}(t)) = \begin{bmatrix} s(t) \\ \psi(t) \end{bmatrix}, \tag{5.27}$$

where subscript v is for vision and a is for AtlasMV.

The Jacobian matrices are defined as:

$$H_v(\mathbf{x}(t_k)) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad H_a(\mathbf{x}(t_k)) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{5.28}$$

---

[1]More details about this node are discussed in chapter 6.

and the sampled state equation is defined as:

$$\hat{f}_{t_k}(\hat{\mathbf{x}}(t_k|t_k)) = \begin{bmatrix} x(t_k) + s(t_k)\Delta t_k \, \cos(\theta(t_k)) \\ y(t_k) + s(t_k)\Delta t_k \, \sin(\theta(t_k)) \\ \theta(t_k) + s(t_k)\Delta t_k \tan(\psi(t_k))\frac{1}{l} \\ 0 \\ 0 \end{bmatrix}, \tag{5.29}$$

where $\Delta t_k := t_{k+1} - t_k$. The remaining matrices are:

$$\hat{\Phi}(t_k|t_k) = \begin{bmatrix} 1 & 0 & -s(t_k)\Delta t \, \sin(\theta(t_k)) & T\cos(\theta(t_k)) & 0 \\ 0 & 1 & s(t_k)\Delta t \, \cos(\theta(t_k)) & T\sin(\theta(t_k)) & 0 \\ 0 & 0 & 1 & \frac{\Delta t}{l}\tan(\psi(t_k)) & \frac{\Delta t}{l}s(t_k)\sec(\psi(t_k))^2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{5.30}$$

$$Q(t_k) = (k_i \Delta t)^2 I_5 +$$

$$\begin{bmatrix} (s(t_k)\Delta t \, k_s)^2 & 0 & 0 & 0 & 0 \\ 0 & (s(t_k)\Delta t \, k_s)^2 & 0 & 0 & 0 \\ 0 & 0 & (s(t_k)\Delta t \, k_s)^2 & 0 & 0 \\ 0 & 0 & 0 & (\Delta t \, a_{max})^2 & 0 \\ 0 & 0 & 0 & 0 & (\Delta t \, \dot{\psi}_{max})^2 \end{bmatrix} \tag{5.31}$$

where

$$\Delta t = t_{measure} - t_{ekfState} > 0$$

is the difference between the measurement timestamp and the EKF state timestamp.
Finally, the covariance matrices associated to the observation functions are:

$$R_v(t_k) = \begin{bmatrix} e_x(t_k) & 0 & 0 \\ 0 & e_y(t_k) & 0 \\ 0 & 0 & e_\theta(t_k) \end{bmatrix}, \quad R_a(t_k) = \begin{bmatrix} e_s & 0 \\ 0 & e_{psi} \end{bmatrix}, \tag{5.32}$$

where $e_x(t_k)$, $e_y(t_k)$ and $e_\theta(t_k)$ are the error variance relative to the measure $y_v(t_k)$, e_s is the error variance relative to the measure of speed and e_ψ is the error variance relative to the measure of steering angle.

Note that $Q(t_k)$ uses the maximum acceleration and the maximum steering angle speed to regulate the covariance matrix. The matrix $(k_i \Delta t)^2 I_5$ is used in order to keep the matrix positive even when the speed is zero (I_5 is the identity matrix of order 5).

The numerical values of the parameters are reported in the table 5.2. The coefficients k_s, k_i, e_s,

Table 5.1: Extended Kalman Filter 1 parameters

Parameter	Value
l	$0.497\,[m]$
k_s	$0.2\,[\,]$
k_i	$0.5\,[\,]$
a_{max}	$4\,[\frac{m}{s^2}]$
$\dot{\psi}_{max}$	$1.5\,[\frac{rad}{s}]$
e_s	$(0.2)^2\,[(m/s)^2]$
e_ψ	$(5\pi/180)^2\,[rad^2]$

e_ψ have been found experimentally using the real robot.

This model has been implemented in MATLAB and used with the robot (both with real one and with a robot that was simulated using Gazebo). Its implementation is thoroughly described in the chapter 6.

5.3.3   Sensor fusion using visual, odometry and inertial information

This section introduces a model for the sensor fusion in which an IMU (**i**nertial **m**easurements **u**nit) is also used. An additional sensor can be very useful both to increase the redundancy of the system and to increase the accuracy of the pose estimation.
Moreover, the redundancy of information opens to the possibility of identifying some parameters of the model. For instance, the model proposed in this section can identify and correct a proportional error in the speed measurement.

Since the IMU wasn't available, this model hasn't been used with the real robot. Its validity was nonetheless studied and clearly proven through simulations.

Model 2: equations

In order to simplify the analysis, the discrete-time model have reported assumes that the variables are sampled at a fixed sample time T.

Let

$$\mathbf{x}(k) = \begin{bmatrix} x(k) \\ y(k) \\ \theta(k) \\ s(k) \\ \psi(k) \\ a(k) \\ \omega(k) \\ s_g(k) \end{bmatrix} \tag{5.33}$$

be the state vector, where:

- $a(k)$ is the acceleration along x_r axis;

- $\omega(k)$ is the angular velocity around z_r axis;

- $s_g(k)$ is the time-varying coefficient relative to a multiplicative error in speed measurement.

The state equation is defined as:

$$\hat{f}_k(\mathbf{x}(k)) = \begin{bmatrix} x(k) + \left(s(t)T + a(k)\frac{T^2}{2} \right) \cos\theta(k) \\ y(k) + \left(s(t)T + a(k)\frac{T^2}{2} \right) \sin\theta(k) \\ \theta(k) + T\left(s(k)\tan\psi(t) \right)\frac{1}{l}\alpha + T\omega(k)(1-\alpha) \\ s(k) + a(k)T \\ \psi(k) \\ a(k) \\ \omega(k) \\ s_g(k) \end{bmatrix}, \tag{5.34}$$

where $\alpha \in (0,1)$ is used to calculate a weighted mean between the orientation obtained using the steering angle and the speed:

$$\theta(k+1) = \theta(k) + Ts(k)\tan\psi(k)\frac{1}{l},$$

and the orientation calculated using angular velocity provided by the IMU:

$$\theta(k+1) = \theta(k) + T\omega(k).$$

The observation function is:

$$h(\mathbf{x}(k)) = \begin{bmatrix} x(k) & y(k) & \theta(k) & s(k)s_g(k) & \psi(k) & a(k) & \omega(k) \end{bmatrix}^T, \tag{5.35}$$

where the $s(k)s_g(k)$ is used to model the fact that the measured speed is the real speed $s(k)$ multiplied by a gain $s_g(k)$.

Assuming that the speed is estimated using an encoder attached to a wheel, this model can explain and correct those errors which are due to a wrong estimation of the wheel diameter, which can be caused, for example, by the variable tire pressure.

The Jacobian matrices is defined as:

$$H(\mathbf{x}(t_k)) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & s_g(k) & 0 & 0 & 0 & s(k) \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{5.36}$$

The remaining matrices are:

$$Q(k) =$$

$$\begin{bmatrix} (0.1\Delta t)^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & (0.1\Delta t)^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & (0.1\Delta t)^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & (20a_{max}\Delta t)^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & (10\dot{\psi}_{max}\Delta t)^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & (10\Delta t)^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & (20\Delta t)^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{5.37}$$

$$\hat{\Phi}(k|k) = \begin{bmatrix} 1 & 0 & -(s(k)\Delta t + a(k)\Delta t^2)\sin\theta(k) & \Delta t\cos\theta(k) & 0 & \Delta t^2\cos\theta(k) & 0 & 0 \\ 0 & 1 & (s(k)\Delta t + a(k)\Delta t^2)\cos\theta(k) & \Delta t\sin\theta(k) & 0 & \Delta t^2\sin\theta(k) & 0 & 0 \\ 0 & 0 & 1 & \frac{\Delta t}{l}\tan\psi(k)\alpha & \frac{\Delta t}{l}\alpha s(k)(\sec\psi(k))^2 & 0 & \Delta t(1-\alpha) & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{5.38}$$

$$R(k) = diag\left\{e_x, e_y, e_\theta, e_s, e_\psi, e_a, e_\omega\right\}, \quad \Delta t := t_{k+1} - t_k. \tag{5.39}$$

Note 1 $Q(k)$ has a less complex structure in comparison to the previous case, in order to underline the impact of the IMU.

Note 2 The variance of the error associated to the process $s_g(\cdot)$ is zero because it has been assumed that it is a constant, though not exactly known value, and that is ideally equal to 1. Otherwise, if $s_g(\cdot)$ is not constant but slowly variable, the associated variance can be a small but non-zero value (for example $s_g(\cdot) = 10^{-6}$). In order to initialize correctly the filter, the state variable $s_g(0)$ must be set to 1 with an associated variance grater then zero.

The numerical values of the parameters are reported in the table 5.2.

Table 5.2: Extended Kalman Filter 2 parameters

Parameter	Value
l	$0.497\,[m]$
a_{max}	$4\,[\frac{m}{s^2}]$
$\dot{\psi}_{max}$	$1.5\,[\frac{rad}{s}]$
α	$0.1\,[]$
e_s	$(0.2)^2\,[(m/s)^2]$
e_ψ	$(5\pi/180)^2\,[rad^2]$
e_x	$(0.2)^2\,[m^2]$
e_y	$(0.4)^2\,[m^2]$
e_θ	$(15\pi/180)^2\,[rad^2]$
e_a	$0.001\,[(m/s^2)^2]$
e_ω	$0.001\,[(rad/s)^2]$

5.3.4   Model verification using Simulink

This subsection presents a simulation used to prove the effectiveness of the model presented in the previous section.

The simulator is based on a realistic car-like model with equation:

$$\frac{d}{dt}\begin{bmatrix} x(t) \\ y(t) \\ s(t) \\ \theta(t) \end{bmatrix} = \begin{bmatrix} s(t)\cos\theta(t) \\ s(t)\sin\theta(t) \\ a(t) \\ s(t)\tan(u_\psi(t))\frac{1}{l} \end{bmatrix}. \tag{5.40}$$

The acceleration has equation:

$$a(t) = \left(P\frac{u_T(t)}{s(t)} - AC_d s(t)^2 \right)\frac{1}{m}, \tag{5.41}$$

where

- $A = 0.5m$ is the frontal area of the car;

- $C_d = 0.3$ is the drag coefficient;

- $m = 100kg$ is the mass;

- $l = 1m$ is the wheelbase length;

- $u_T(t) \in [-1,1]$ is the throttle position;

- $u_\psi(t) \in [-0.3,0.3]$ is the steering angle;

- $P = 150W$ is the engine power.

The Simulink model is contained in the following block: The throttle position and the steering
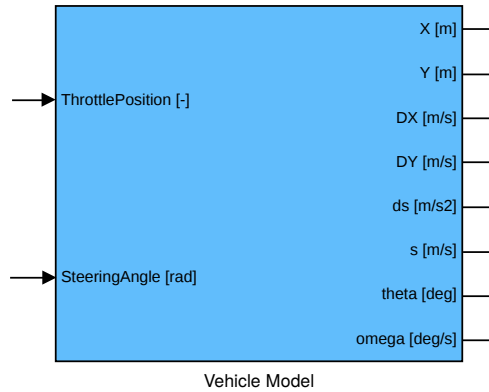

Figure 5.4: Vehicle Simulink block

angle are driven by two random signals in order to obtain a random path.
The inputs and outputs are used to simulate the sensors:

- $x(\cdot)$, $y(\cdot)$ and $\theta(\cdot)$ are used to simulate the pose calculated using visual information;

- $s(\cdot)$ and the steering angle are used to simulate the information provided by AtlasMV;

- ds $(\dot{s}(\cdot))$ and $\omega(\cdot)$ are used to simulate the IMU.

A Gaussian additive noise has been added to each signal, and all the signals have been sampled at the same frequency $f_s = 30Hz$.
The table 5.3 resumes the noise characteristics.

Table 5.3: Gaussian noise parameters

signal	mean	variance
$x(\cdot)$	0	0.4^2
$y(\cdot)$	0	0.4^2
$\theta(\cdot)$	0	$(15\pi/180)^2$
$s(\cdot)$	0	0.02^2
$\dot{s}(\cdot)$	0	0.001
$\psi(\cdot)$	0	$(5\pi/180)^2$
$\omega(\cdot)$	0	0.001

Simulations

The simulation parameters are as follows:

- the initial condition of the simulated robot have been set to zero ($x(0) = y(0) = \theta(0) = s(0) = \psi(0) = 0$) and the coefficient s_g (constant) is equal to 1.2;

- the initial values of the EKF have been set to zero, each variable with a variance 0.1, except for s_g, which is initialized to 1 with a variance of 0.15^2;

- the simulation time is $15s$;

- the model with IMU corresponds to the model introduced in section 5.3.3 and the model without IMU corresponds to the model introduced in section 5.3.2;

- Where applicable, the model without IMU uses the same parameters as the model with IMU, and it reads the real speed (not the real speed multiplied by 1.2).

The figure 5.5 shows the random path followed by the robot. The blue line represents the real trajectory, and the green and black line the estimated trajectory without and with IMU, respectively.

Note that, as expected, the trajectory estimated using the IMU is less noisy.
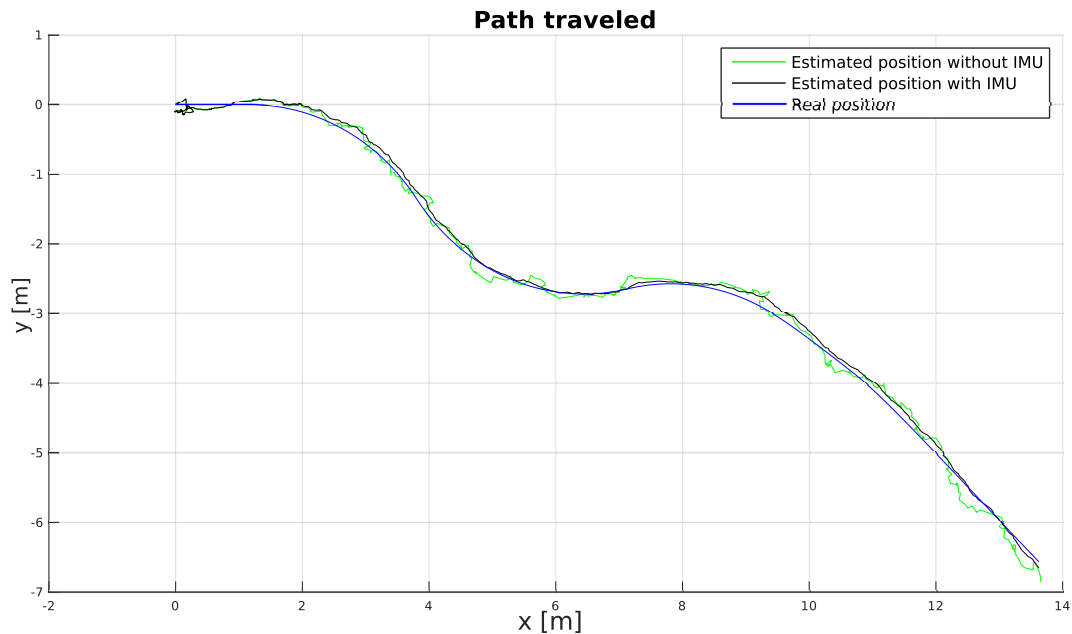

Figure 5.5: Path traveled

The figures 5.6 and 5.7 show the errors relative to the pose estimation (orientation and position).
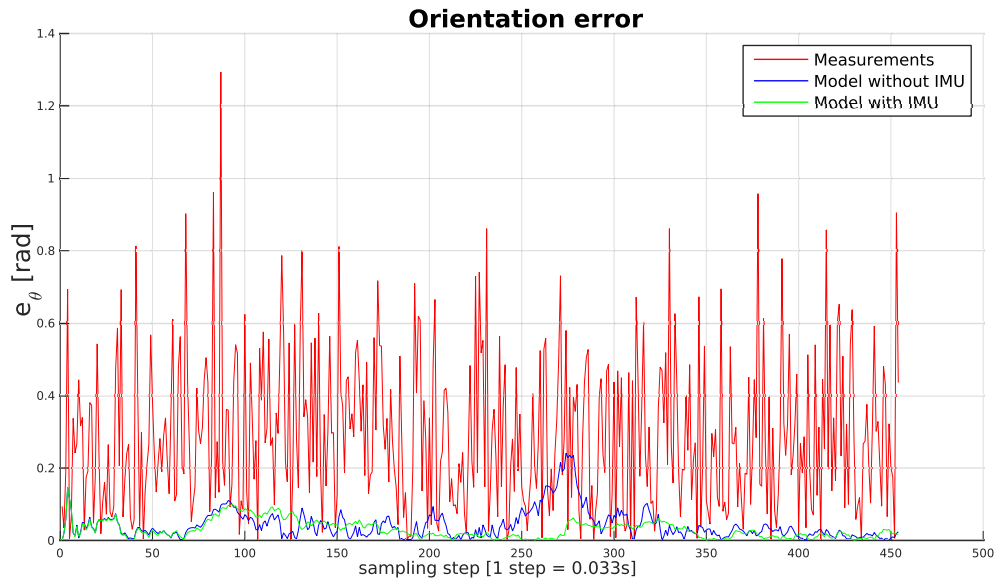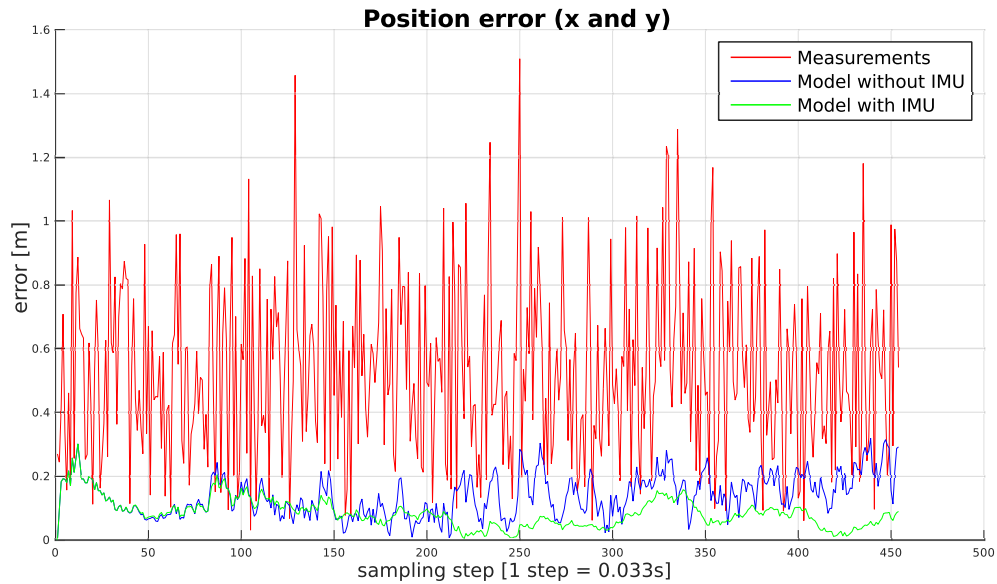

Figure 5.6: Orientation error

Figure 5.7: Position error

The figures 5.8 and 5.9 show how the model with IMU can properly identify the parameter s_g and can also correct the speed estimation in less than 200 steps (about $6.6s$).
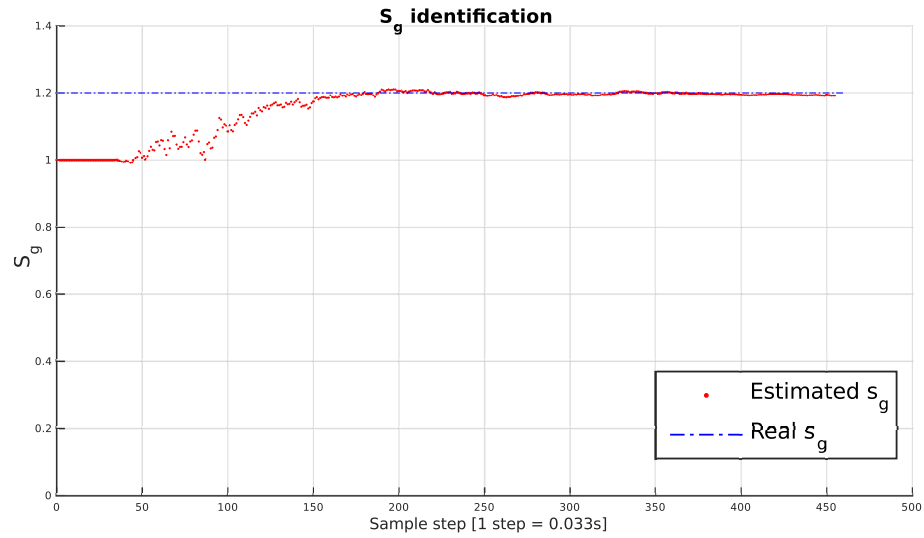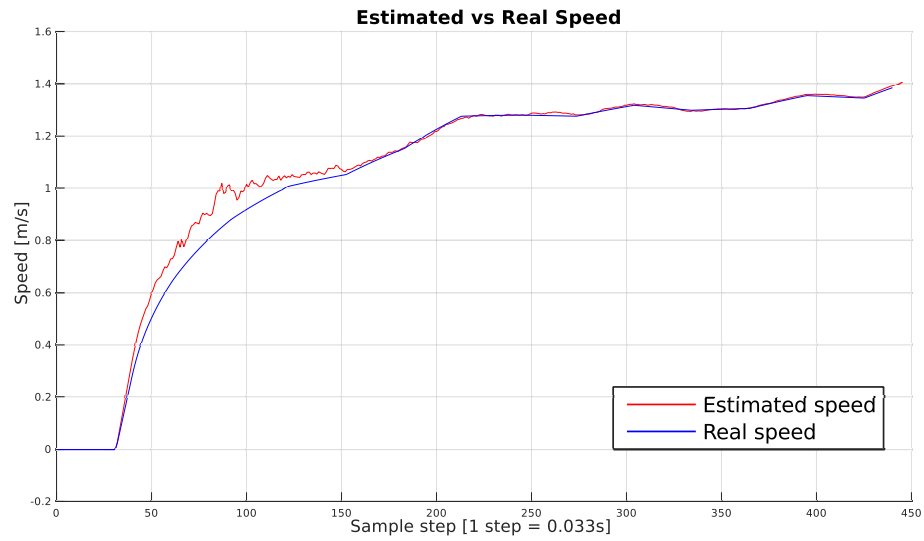

Figure 5.8: S_g identification

Figure 5.9: Real vs estimated speed $s(\cdot)$

Table 5.4 shows a quantitative comparison between the EKF with and without IMU. The EKF works fine in both the cases, but an additional improvement can be obtained using an IMU.

Table 5.4: Average mean error

-	Measurements	EKF w-o IMU	EKF with IMU	Gain
position [m]	0.531	0.138	0.084	+63.2%
orientation [rad]	0.288	0.043	0.030	+39.7%

Final consideration about the IMU

Adding an IMU is probably the best method to increase the efficiency of the EKF. The reason is that the EKF works with the hypothesis of Gaussian error with zero mean, but in fact odometry information and expecially vision information are affected by systematics errors with a completely different statistical distribution, which compromises the performance of the filter. The proposal would be to add an IMU and then to overstimate the errors associated to the remaining sources of infomation (vision and odometry). This problem could and should be studied in a future, as a separated work.

Chapter 6

Implementation in MATLAB

During the creation of an Extended Kalman Filter for ROS, it has been necessary to deal with a problem connected to ROS and its implementation: ROS is not a real-time system and the messages are exchanged between nodes using the TCP/IP protocol. For this reason, the non deterministic behaviour of ROS causes some uncertainty. Furthermore, the elaboration of visual information introduces a non-negligible delay. These problems have been solved using a time-varying Extended Kalman Filter.

The EKF has been implemented in MATLAB and connected to ROS using the Robotics System Toolbox^{TM} (RST) introduced in MATLAB r2015a.

During the practical implementation of the filter two main problems have been highlighted:

1. the measurements from the nodes *atlasmv* and *datamatrix-pose-pub* are aperiodic and can be received Out-of-Order;

2. MATLAB doesn't provide any native mechanism for regulating the access to the variables, like mutex (**mut**ual **ex**clusion)[27] does in C++. The multi-thread programming is also limited.

These problems have been partially solved using an algorithm specifically designed for MATLAB.

6.1 First proposed algorithm

The first proposed algorithm is the algorithm represented in fig. 6.1. This algorithm is subdivided in three different execution threads:

1. a first thread for the buffer management, for it to receive new measurements and store them ordered by timestamp;

2. a second thread for updating the state $\hat{\mathbf{x}}(t_k|t_k)$ when the contents of the buffer changes;
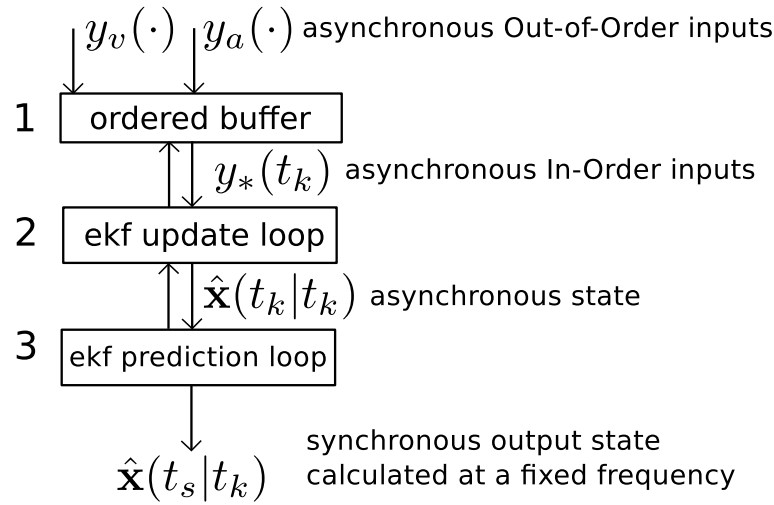
Figure 6.1: EKF - Structure of the algorithm

3. a third thread that *publishes* the predicted pose of the robot *ekf-robot-pose* on a specific ROS topic at a fixed publishing frequency f_p (for example $f_p = 30Hz$) and using the last estimated state, as calculated by the second thread.

The timestamp t_s in fig. 6.1 must satisfy the following conditions:

$$t_s \geq t_k \,, \quad t_{s+1} = t_s + \frac{1}{f_p} \,,$$

where t_k is the timestamp of the last received measurement.
Note that the use of two mutex (one to exchange data between thread 1 and 2, and the other one exchange data between thread 2 and 3) is required to preserve the consistency of the information.

This algorithm provides the prediction of the robot pose at a given fixed frequency (useful for controlling purposes) and at the same time it can solve the problems related to the long elaboration time of the visual information and, in general, to the non deterministic behaviour of ROS. It hasn't been possible to develop this algorithm using MATLAB because of its lack of native multithread programming.
Nonetheless, the possibility of developing the algorithm in C++ using the library *ecl-linear-algebra*[28] officially included in ROS has been studied, but this option has been abandoned for time reason and because it is difficult to monitor the behaviour of the algorithm without the tools included in MATLAB.

6.2   Second proposed algorithm

The second algorithm has been redesigned taking into account the limitations of MATLAB. The buffer has been removed, and the second and third threads have been merged into a single thread.

The resulting pseudo code is:

```
1   initializeROS()
2   initializeEKF()
3   while 1 {
4      if newDatamatrixVectorAvailable() == TRUE {
5          msg = getNewDatamatrixVector()
6          rawRobotPose = poseMsg2rawPoseEst(msg)
7          rawRobotPose = forwardEulerCorrection(rawRobotPose)
8          publishRaw(rawRobotPose)
9          ekfState = ekfUpdate(rawRobotPose)
10     }
11     if newAtlasMsgAvailable() == TRUE {
12         pose = getNewAtlasMsg()
13         ekfState = ekfUpdate(pose)
14     }
15     currentTime = getCurrentTime()
16     prediction = ekfPrediction(ekfState,currentTime)
17     publishOnRos(prediction)
18     sleep(20ms)
19 }
```

This code runs on MATLAB and it can be seen as a single execution thread [1].

---

[1] In fact, the RST hides a low-level layer based on the Java implementation of ROS, which uses multiple threads to manage the connections between ROS nodes.

Explanation of the code

ROS initialization

The first line of code

```
1  initializeROS()
```

creates the node *atlasmv-ekf* and initializes the connections. The correspondent MATLAB code is

```
...
% node creation
roscoreIp = '127.0.0.1';
nodeName = 'atlasmv_ekf';
rosinit(roscoreIp,'NodeName',nodeName);
...
% subscribers
dmPoseTopicName = '/datamatrix_pose_pub/datamatrix_pose';
dmPoseSet_sub = rossubscriber(dmPoseTopicName,'BufferSize', 1);
...
atlasmvStatusTopicName = '/atlasmv/base/status';
atlasStatus_sub = rossubscriber(atlasmvStatusTopicName, ...
                        'atlasmv_base/AtlasmvStatus','BufferSize', 1);
...
% publishers
poseEkf_pub = rospublisher(strcat(nodeName,'/ekf_robot_pose'), ...
                'geometry_msgs/PoseWithCovarianceStamped' );

poseRaw_pub = rospublisher(strcat(nodeName,'/raw_robot_pose'), ...
                'geometry_msgs/PoseWithCovarianceStamped' );
```

This section of code is divided in three parts:

1. node creation and its connection to the *roscore* node;

2. connection to the subscribed topics *datamatrix-pose-pub/datamatrix-pose* and *atlasmv-base/AtlasmvStatus*;

3. initialization of the topics *atlasmv-ekf/ekf-robot-pose* and *atlasmv-ekf/raw-robot-pose*.

Note 1 In the case here examined, the *roscore* node has address 127.0.0.1 because it has been executed in the same machine. This node provides basics ROS functionalities and it must always be the first node to be launched.

Note 2 The parameter BufferSize is setted to 1 because only the last message published on the relative topic has to be processed by the algorithm.

EKF initalization

The variable *ekfState* is a structure with the following fields:

1. wheelbase length l $[m]$;

2. maximum velocity v_{max} $[m/s]$;

3. maximum acceleration a_{max} $[m/s^2]$;

4. maximum steering angle ψ_{max} $[rad]$;

5. maximum steering angle speed $\dot{\psi}_{max}$ $[rad/s]$;

6. state vector *stateVec* $[[m][m][rad][m/s][rad]]^T$;

7. covariance matrix *covMatrix* [unit of measurement derivable from stateVec];

8. timestamp of associated to the data structure *timeStamp* $[s]$;

where parameters 1 to 5 are characteristics of the robot. Parameters 6 and 7 represent the state of the filter.

When the initialization step is finished, the function

```
2   initializeEKF()
```

waits for a message from *datamatrix-pose-pub/datamatrix-pose* and tries to calculate the robot pose (x, y, θ). The filter is initialized with the first valid pose (x, y, θ) that was calculated. This approach requires that at least two data matrices are visible during the node initialization, otherwise the filter can't be initialized and the main loop doesn't starts.

EKF loop

At line 3, the EKF starts to work.
The code

```
4       if newDatamatrixVectorAvailable() == TRUE {
5           msg = getNewDatamatrixVector()
6           rawRobotPose = poseMsg2rawPoseEst(msg)
7           rawRobotPose = forwardEulerCorrection(rawRobotPose)
8           publishRaw(rawRobotPose)
9           ekfState = ekfUpdate(rawRobotPose)
10      }
```

checks if a new vector of data matrices has been published on the topic *datamatrix-pose-pub/datamatrix-pose*. If it has been received, the message is processed by the function:

```
6        rawRobotPose = poseMsg2rawPoseEst(msg)
```

which implements the algorithm presented in section 5.1. If three or more data matrices have been detected, the function selects only two of them at random. It does so in order to make the error associated to the measure as less systematic and more random as possible.
The input message is the vector of *datamatrixData*, which is the structure defined in section 4.4. The output is a structure containing the measurement, the relative covariance matrix defined in (5.7) and the acquisition timestamp t_{acq}.

The function

```
7        rawRobotPose = forwardEulerCorrection(rawRobotPose)
```

applies the forward Euler method in order to reduce the effect of the systematic error due to the high elaboration time of visual information. The "raw" robot pose is published on the ROS topic *atlasmv-ekf/raw-robot-pose*.
Let $t_{ekfState}$ be the timestamp of the EKF state and $t_{current}$ the current time. Considering the (5.7), the correction applied to $\mathbf{y}_v(t_{acq})$ is

$$\begin{bmatrix} x(t_{current}) \\ y(t_{current}) \\ \theta(t_{current}) \end{bmatrix} = \begin{bmatrix} x(t_{acq}) \\ y(t_{acq}) \\ \theta(t_{acq}) \end{bmatrix} + \begin{bmatrix} (t_{current} - t_{acq})s(t_{ekfState})\cos\theta(t_{ekfState}) \\ (t_{current} - t_{acq})s(t_{ekfState})\sin\theta(t_{ekfState}) \\ (t_{current} - t_{acq})s(t_{ekfState})\tan\psi(t_{ekfState})\frac{1}{l} \end{bmatrix} . \qquad (6.1)$$

To keep in account the fact that this correction increases the uncertainty associated with the measure, the covariance matrix has been multiplied by a scale factor proportional to $(t_{current} - t_{acq})$.
The resulting covariance matrix is

$$cov\left(\mathbf{y}_v(t_{current}) \cdot \mathbf{y}_v^T(t_{current})\right) = (1 + t_{current} - t_{acq})cov\left(\mathbf{y}_v(t_{acq}) \cdot \mathbf{y}_v^T(t_{acq})\right) . \qquad (6.2)$$

Finally, the timestamp associated to the measure changes from t_{acq} to $t_{current}$. This approach has revealed itself to be good in practical cases, because it actually reduces systematics errors.

The line of code

```
8        ekfState = ekfUpdate(rawRobotPose)
```

calls the function *ekfUpdate*, which updates the status using the received measurement.

The following instructions

```
11    if newAtlasMsgAvailable() == TRUE {
12        pose = getNewAtlasMsg()
13        ekfState = ekfUpdate(pose)
14    }
```

check if a new status message has been published on the *atlasmv/base/status* topic. If a new message has been received, the speed and the steering angle are used to update the EKF status.

In computer science, this particular activity is called *polling*.
In this case there are two buffers:

- one containing the messages published on the *datamatrix-pose-pub/datamatrix-pose* topic;

- one containing the messages published on the *atlasmv/base/status* topic.

Polling is the process which involves checking of one or more "client programs". In this case the RST can be considered the client program, which is repeatedly called in order to check if the buffers have received new messages. This method is not efficient, but it represents the only choice in all those cases where the multithread programming isn't possible.

The last code section

```
15    currentTime = getCurrentTime()
16    prediction = ekfPrediction(ekfState,currentTime)
17    publishOnRos(prediction)
18    sleep(20ms)
```

is dedicated to the prediction of the robot status. The prediction is based on the actual time of the system (*currentTime*) and on the last calculated state (*ekfState*). Finally, the EKF prediction is published on the ROS topic *atlasmv-ekf/ekf-robot-pose*.

6.3   Connection between nodes

This sections presents a final overview of how the ROS nodes are connected and which messages they exchange.
The figure 6.2 shows four different nodes:

- *datamatrix-pose-pub* - used to detect the data matrices and to calculate the poses $\mathbf{T}^r_{dm_j}$;

- *atlasmv* - used to control the robot and to read odometry information;

- *atlasmv-ekf* - used to estimated the pose using to output of the node *datamatrix-pose-pub*;
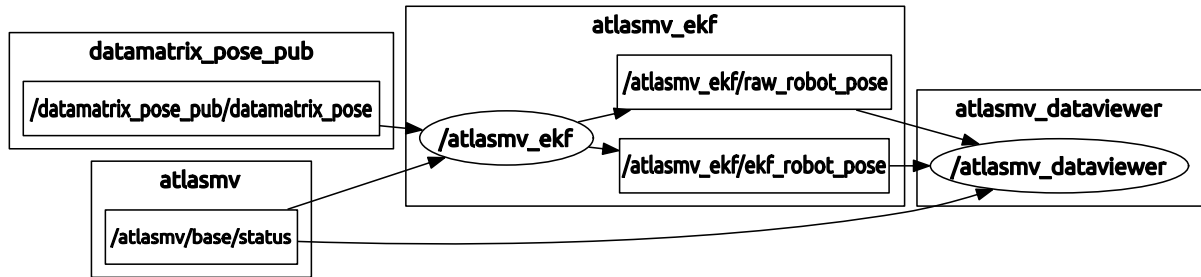
Figure 6.2: Node graph

- *atlasmv-dataviewer* - used to collect and visualize data.

The nodes *datamatrix-pose-pub* and *atlasmv-ekf* have already been discussed respectively in chapter 4 and in chapter 6.

The node *atlasmv* has been developed by the Laboratory for Automation and Robotics of the University of Aveiro. This node provides a ROS interface for the robot.
A message containing the status of the robot is published on a topic called *atlasmv-base-status*. This particular message is called *AtlasmvStatus* and it has the following structure:

```
Header header
float64 brake
float64 dir
float64 speed
float64 x
float64 y
float64 orientation
float64 distance_traveled
int32 turnright
int32 turnleft
int32 headlights
int32 taillights
int32 reverselights
int32 cross_sensor
int32 vert_sign
int32 errors
```

When the node *atlasmv-ekf* receives this type of message, it extracts the timestamp (from the *Header*), the *speed* $s(\cdot)$ and the steering angle *dir* $\psi(\cdot)$.
Header is a standard ROS message with the following structure:

```
uint32 seq
```

```
time stamp
string frame_id
```

where *seq* is a sequential number, *stamp* is the timestamp and *frame-id* is an optional string.

Finally, the node *atlasmv-dataviewer* is created using MATLAB and it is used only to collect and analyse data.

Chapter 7

Experimental results

This chapter presents the experimental results obtained using AtlasMV and its simulated version in Gazebo. Using the real robot, only qualitative considerations could be expressed, since it wasn't possible to measure the real position of the robot.

On the contrary, through Gazebo it was possible to perform a quantitative analysis: the robot pose can be obtained reading the messages published by Gazebo on the topic */ackermann-vehicle/gazebo/model-states*, which contains a vector of poses, one pose for each object included in the simulated world.

7.1 Test environment

Figure 7.1 shows 21 data matrices generated and positioned inside the LAR. In order to recreate a similar scenario, the same data matrices have been included in the simulator.
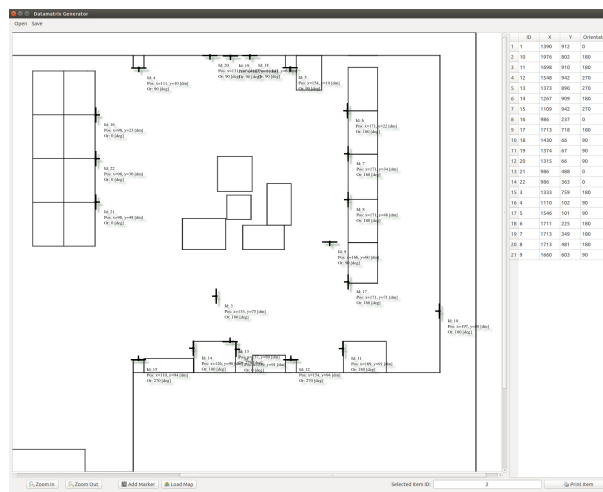
Figure 7.1: Data matrices positioned inside the LAR.

The figure 7.2 shows the simulated robot, the data matrices and the simulated cameras outputs (front and rear cameras).

For hardware limitations[1], cameras output have been limited to a VGA resolution (640x480 pixels) at 5 frames per second. A Gaussian noise with standard deviation 0.01 has been added to the generated frames.
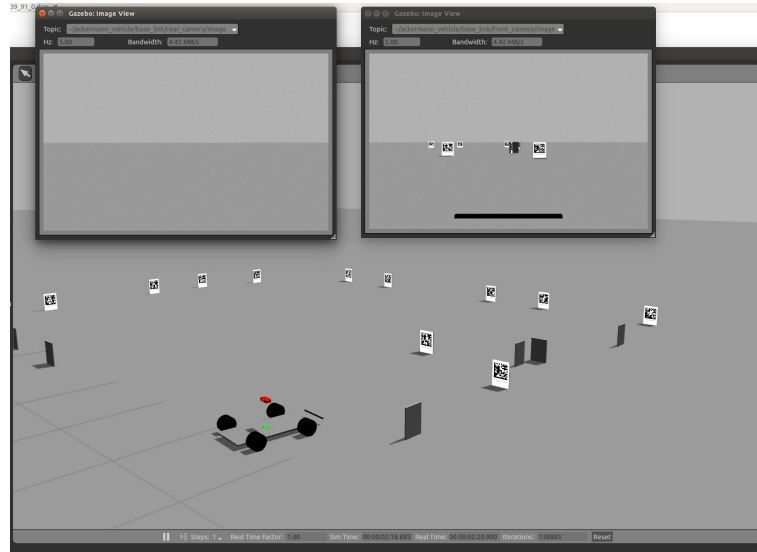

Figure 7.2: Gazebo simulator

The tests with the real robot have been executed at a resolution of 1280x960 pixels and at 10 frames per second. The exposure time has been manually set to $8-11ms$ (depending on light conditions). The detection algorithm doesn't work if the image is blurred, because it is based on a corner detector (which is a gradient-based function). The low exposure time produces images with a low contrast but also with a low blur effect: many tests have been executed in order to find the best *Libdmtx* parameters.
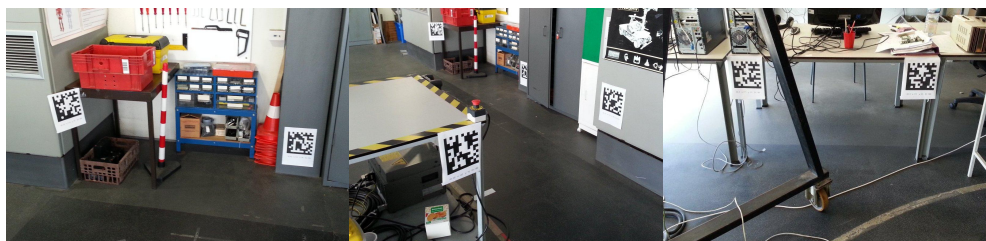

Figure 7.3: Data matrices inside the LAR

---

[1]The processor Intel® Core™ i3-2310m wasn't able to run the nodes and the simulator using two virtual cameras at resolution of 1280x960 pixels.

7.2   Simulations using Gazebo

7.2.1   First simulation: "S" trajectory

In the first simulation, the robot was driven manually using an Xbox[TM] 360 controller. The figure 7.4 shows the path travelled by the robot. There is a small offset between real position (blu dots) and estimated position (black dots) (the measurements are indicated with the red crosses). Considering that this simulation was running in real time, this problem can be caused by the low MATLAB performance, which introduces a considerable delay on data processing.
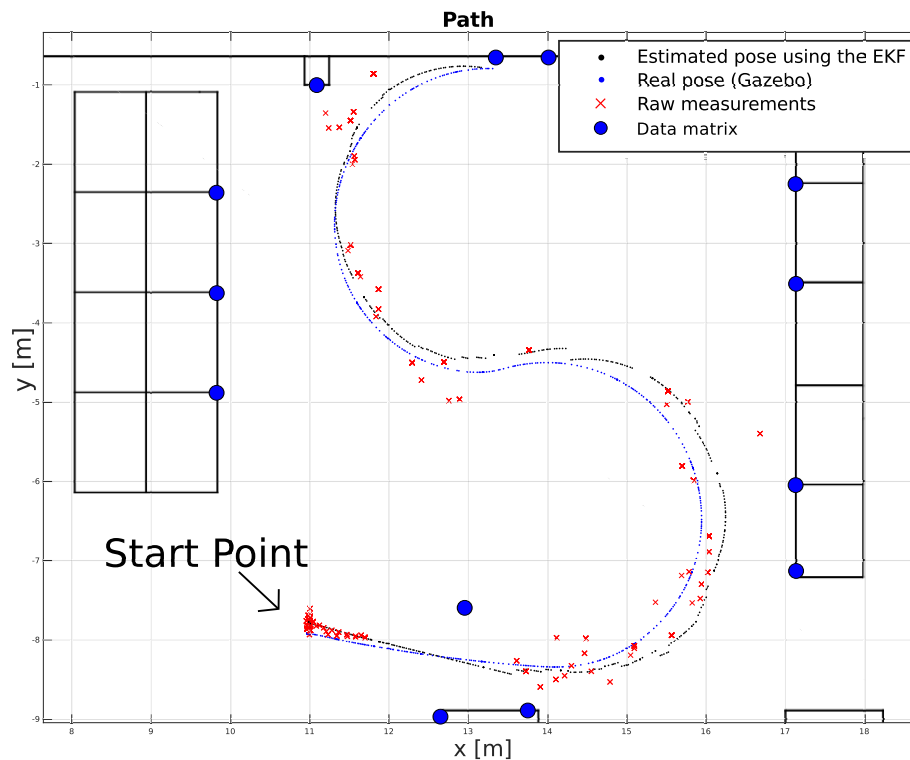
Figure 7.4: Gazebo simulation 1: trajectory

The figures 7.5 and 7.6 show a comparison between real and estimated position and between real and estimated orientation. It has been used the EKF without IMU introduced in chapter 6.

It is very interesting to note that the mean errors associated to this simulation are very similar to the errors (table 5.4, reported below) obtained using the Simulink model presented in section 5.3.4.

The table 7.1 shows a comparison between the mean error obtained with Simulink and the one obtained with Gazebo.
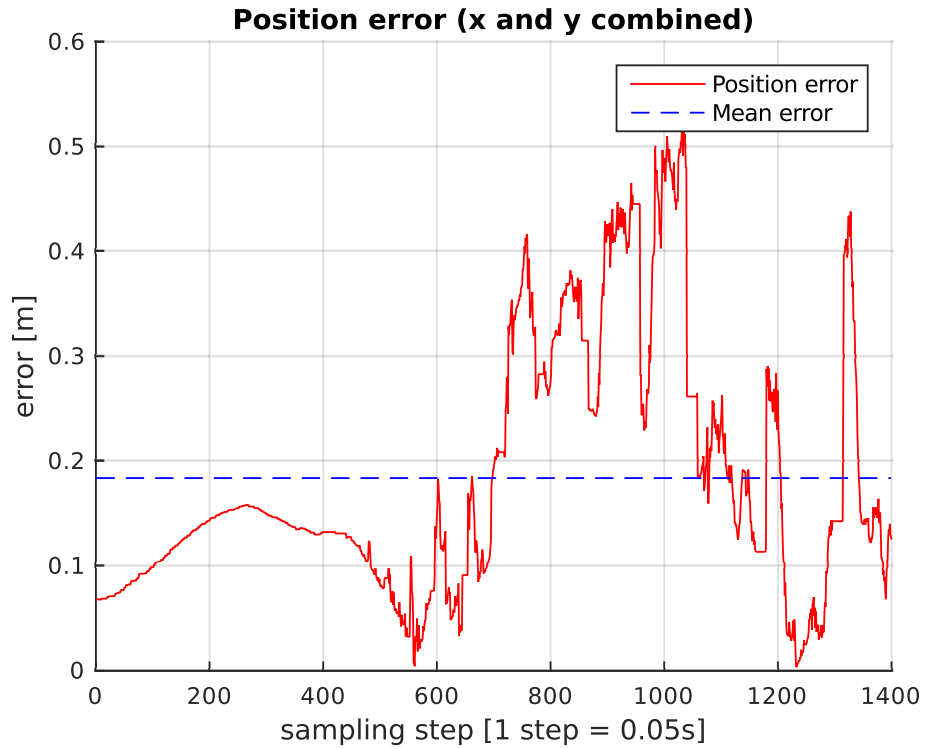
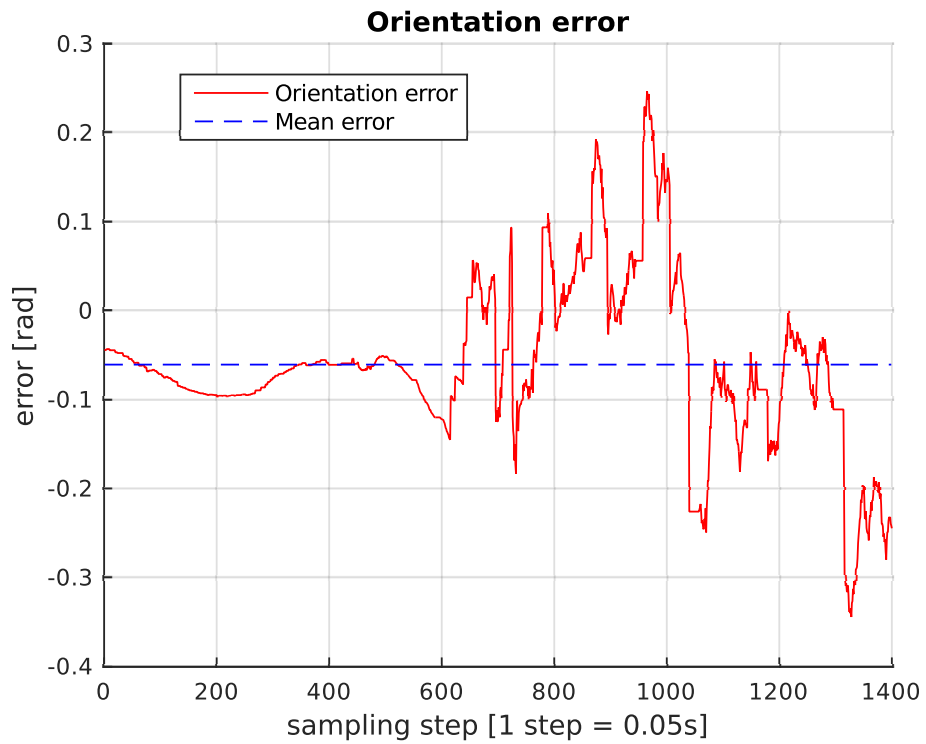Figure 7.5: Gazebo simulation 1: position error

Figure 7.6: Gazebo simulation 1: orientation error

Table 7.1: Average mean error (Simulink-Gazebo) and standard error (Gazebo)

-	Simulink Mean error	Gazebo mean error	Gazebo Str.Dev
position [m]	0.138	0.183	0.121
orientation [rad]	0.043	-0.061	0.091

In general, this level of precision can be considered acceptable for many real life applications (the standard error is smaller than the robot dimension). A better performance could probably be obtained using a high optimized EKF, such as the one proposed in section 6.1.

Note that also in this case the relative poses $\mathbf{T}_{c1}^{r}$ and $\mathbf{T}_{c2}^{r}$ have been calculated using the same procedure used with the real robot. This choice has been made in order to keep the results as more realistic as possible.

7.2.2   Second simulation: "round trip"

This simulation is similar to the first, except that the robot comes back to the starting point. Figure 7.7 shows the path followed by the robot. Note that, often, the estimated position and the real position are almost the same.
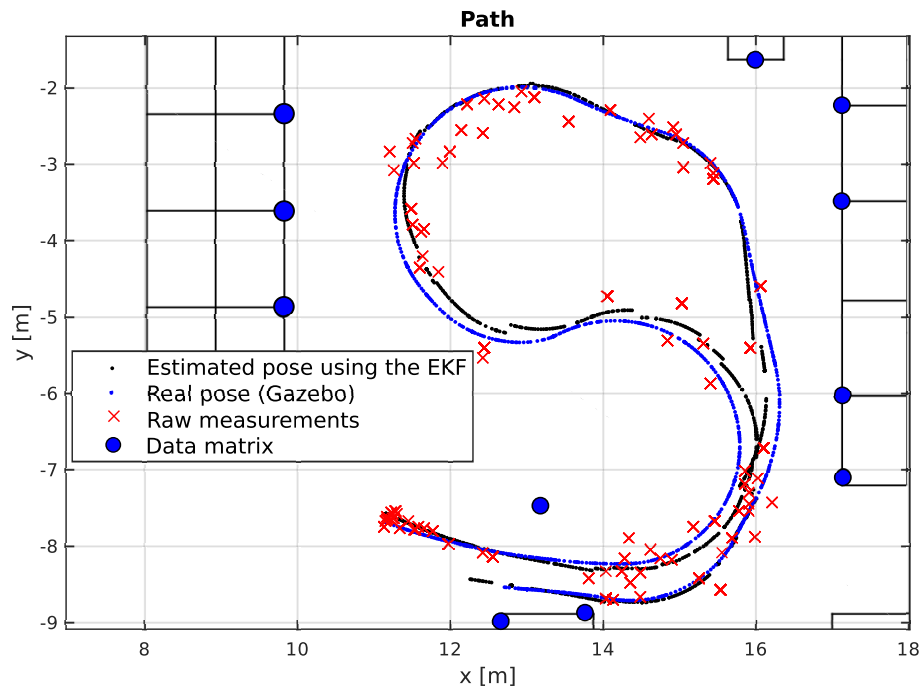

Figure 7.7: Gazebo simulation 2: trajectory

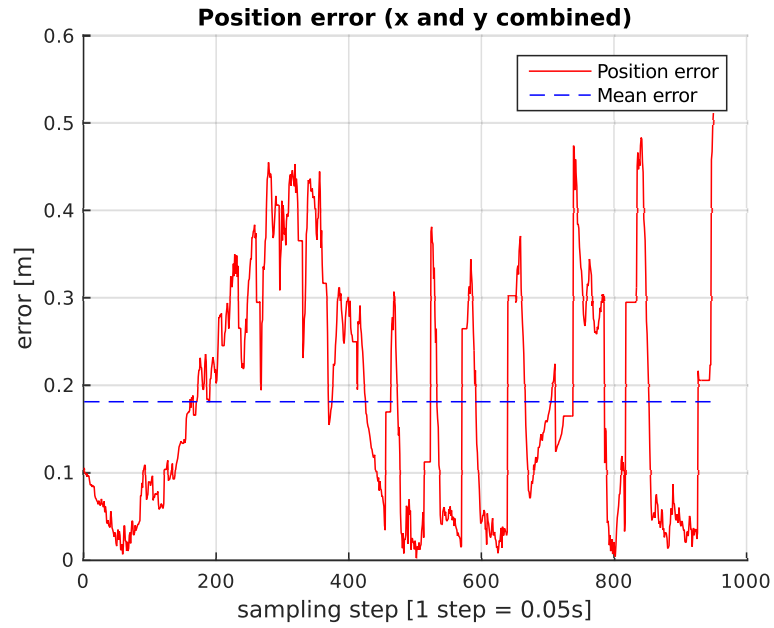Figure 7.8 and 7.9 show the respective errors.

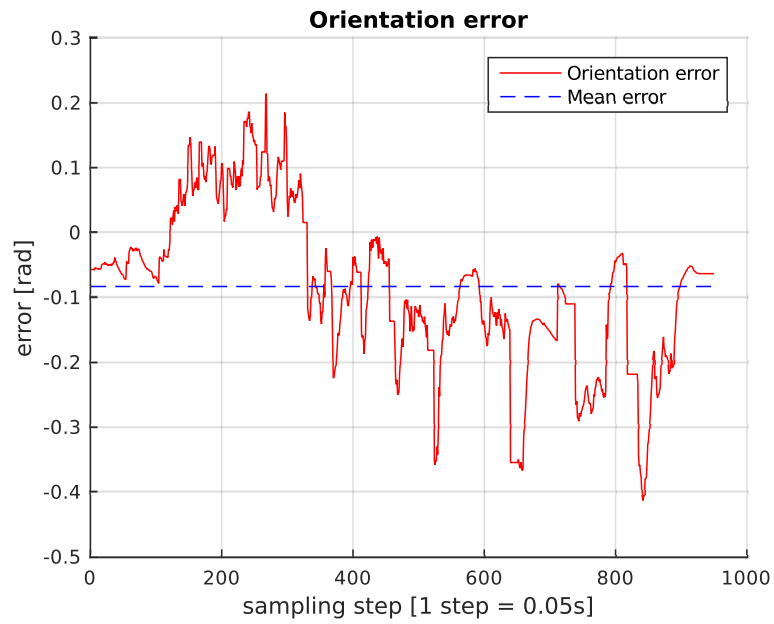Figure 7.8: Gazebo simulation 2: position error

Figure 7.9: Gazebo simulation 2: orientation error

Finally, table 7.2 confirms the results reported in table 7.1.

Table 7.2: Mean error and standard deviation

-	Standard deviation	Mean error
position [m]	0.130	0.177
orientation [rad]	0.120	-0.081

7.2.3   Third simulation: straight trajectory

The third simulation shows the robot following a straight trajectory. Four data matrices have been positioned, as shown in figure 7.10.
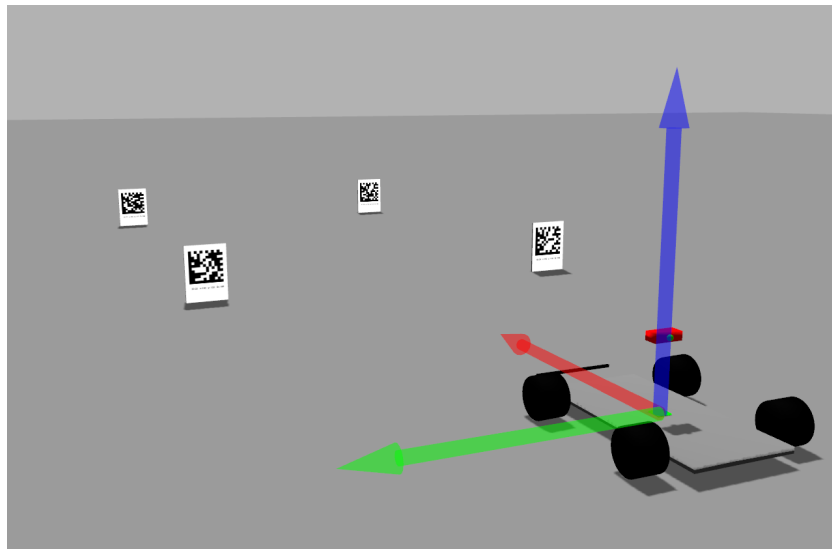

Figure 7.10: Gazebo simulation 3: straight trajectory - screenshot

Figure 7.11 shows the real position (blue dots) and the estimated one (black dots). As reported in table 7.3, the offset (mean error) is about $0.11m$ and its standard deviation is about $0.06m$: this means that there is a small systematic error. This systematic error is probably related to the vision algorithm; it should therefore be furtherly investigated. Nonetheless, this level of accuracy seems acceptable. Figures 7.12 and 7.13 show respectively the position and the orientation errors.

Table 7.3: Mean error and standard deviation

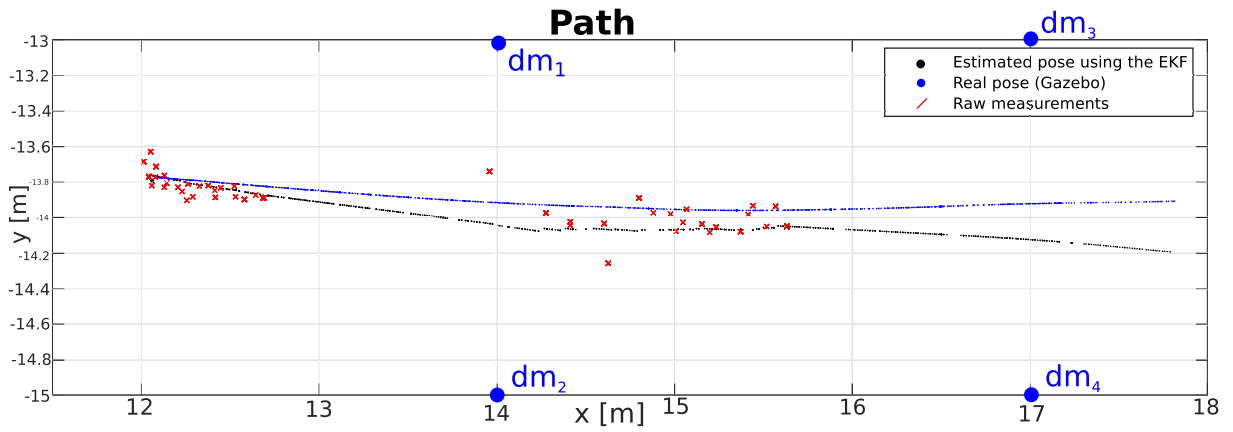| -	Standard deviation	Mean error
position [m]	0.061	0.111
orientation [rad]	0.019	-0.063

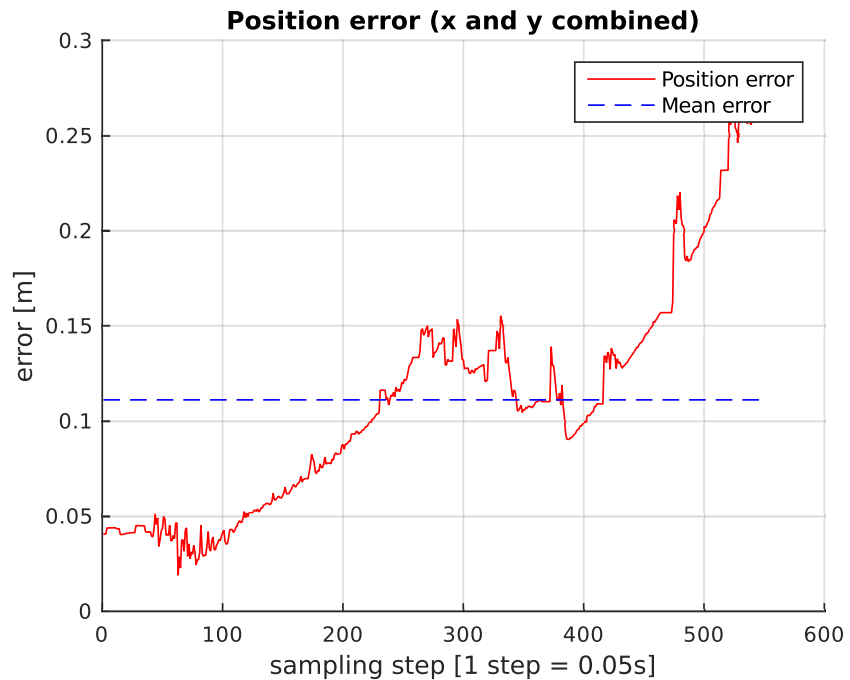Figure 7.11: Gazebo simulation 3: straight trajectory

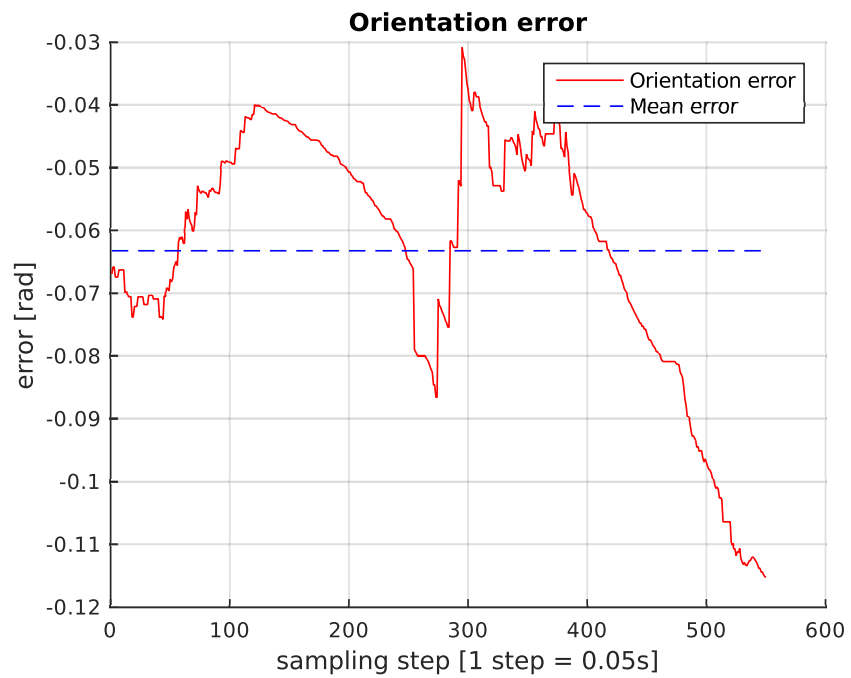Figure 7.12: Gazebo simulation 3: straight trajectory - position error

Figure 7.13: Gazebo simulation 3: straight trajectory - orientation error

7.3   Test using the real robot

The following tests show the behaviour of the localization system using the AtlasMV robot. Unfortunately it has not been possible to compare the estimated pose with the real pose for the lack of ground truth, but this tests remain important because they show a behaviour very similar to the one obtained in the simulator.

7.3.1   First test: "S" trajectory

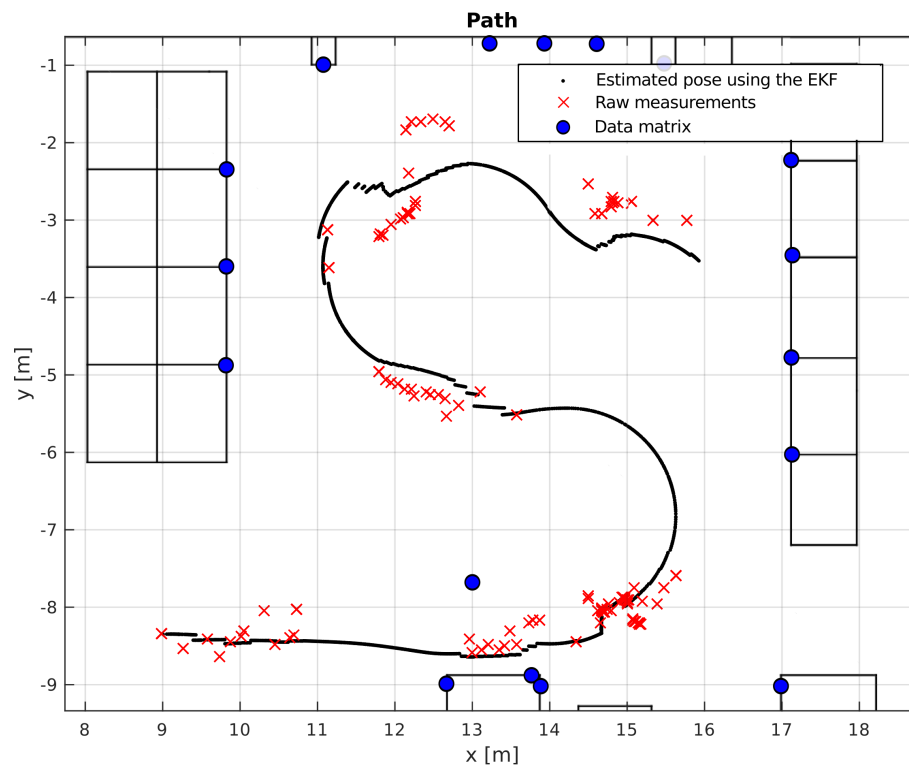This simulation shows a "S" trajectory similar to the one obtained using Gazebo.

Figure 7.14: AtlasMV: "S" trajectory

The figure 7.14 shows the trajectory obtained using the AtlasMV robot with two enabled cameras. The trajectory results less smooth if compared to the one obtained with Gazebo, and this for two reasons:

- difficulties in positioning the data matrices correctly;

- the speed estimation provided by AtlasMV is not accurate since the encoder is attached to the engine rotor and not to the wheel(s). Also, the differential introduces an additional error proportional to the steering angle.

Despite these problems, the result seems promising.

7.3.2   Second test: "S" trajectory - one camera

This second test is similar to the first one: the only difference is that the rear camera was disabled.
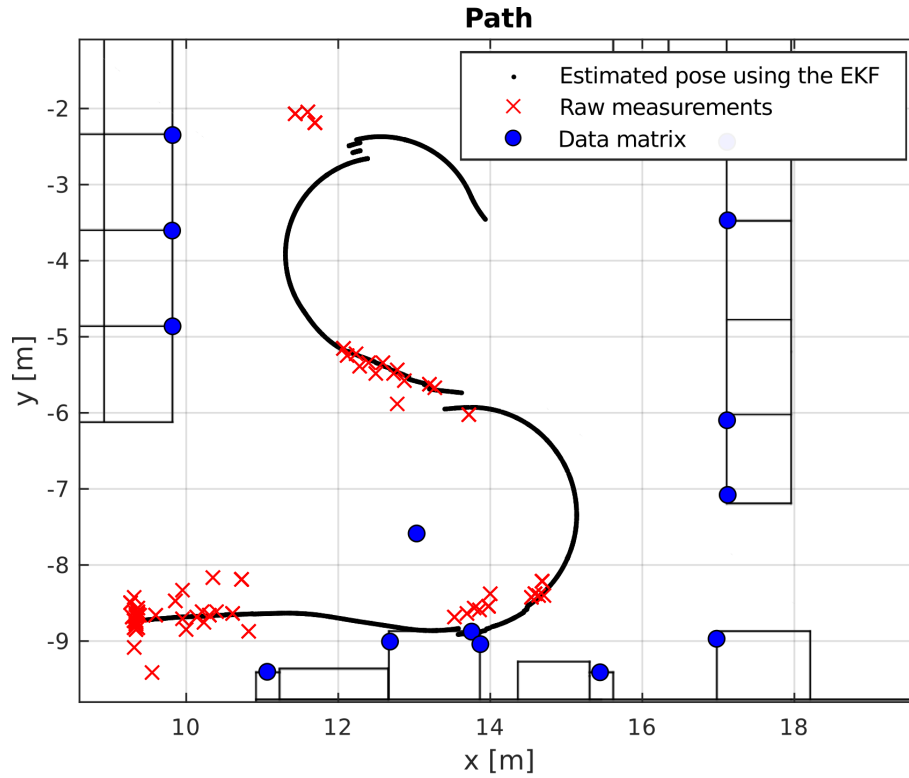

Figure 7.15: AtlasMV: "S" trajectory - only frontal camera

The trajectory presents discontinuities, probably caused by the lower number of data matrices detected as well as the poor odometry, which introduces and error that grows over time.

7.3.3   Third test: straight trajectory

In analogy to the third simulation, this third test shows the AtlasMV following a straight trajectory. The figure 7.16 shows the corridor with four data matrices.

Figure 7.16: LAR's corridor

The figure 7.17 shows the estimated position. This test shows results very similar to those obtained using Gazebo; probably thanks to the fact that only four data matrices (although positioned accurately) have been used.
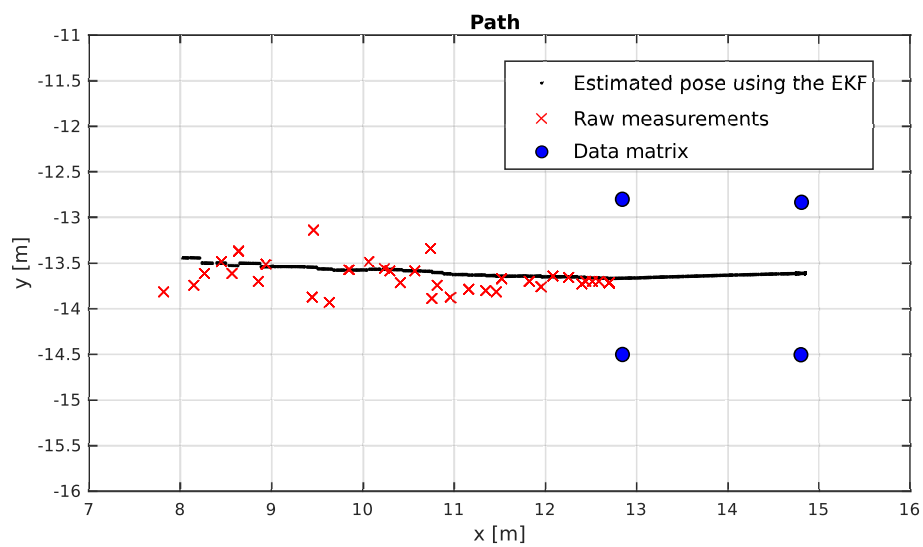

Figure 7.17: AtlasMV: straight trajectory

The tests with the real robot have confirmed that the localization system works as expected.

The results does not seems as good as the results obtained with Gazebo, but this can be easily explained keeping in consideration the following facts:

- every data matrix has been manually positioned with a given error;

- the map used was only an approximation of the real map;

- the odometry system was poor (encoders attached to the motors).

In this particular case, the main source of errors was determined by the measure of speed provided by the robot (as already mentioned, the differential causes an error proportional to the steering angle of the robot). Similar results to the ones obtained using Gazebo can be obtained using a robot with a more accurate odometry system.

Chapter 8

Conclusions

The aim of this thesis was to implement an indoor localization system using visual information and passive markers. The obtained results are promising, but the performance of this type of system depends on many factors: detection algorithm, localization method, quality of the odometry and efficiency of the sensor fusion algorithm.

In chapter 3 the problem was discussed of how to encode efficiently the required information. Moreover, a graphic tool has been developed in order the create the required markers. The encoding system was well dimensioned for this application: the resolution of $0.1m$ was resulted sufficient (in practice it is difficult to position a data matrix with an error smaller than $0.1m$).

Chapter 4 has described the perception algorithm based on the open source library *Libdmtx*, which was in general slow and not always stable. Despite these limitations, it was accurate enough for this kind of application and it didn't represent a bottleneck. The most delicate part of the operation has revealed itself to be the calibration: a bad calibration adds a systematic error to every single measurement.

Chapter 5 deals with the EKF as well as with the problem of sensor fusion. In order to reduce the errors and also to eventually identify and correct systematic errors, the use of an inertial sensor was suggested. Since the systematic errors are the main problem of this technology, it would be advisable to further study this solution.

Chapter 6 deals with the implementation of the EKF and the problems connected to its MATLAB implementation and its integration in ROS. The EKF has a big impact on the final accuracy of the localization system and its further optimization should be considered in future works.

In Chapter 7 have been introduced the experimental results obtained using the simulator

Gazebo and the robot AtlasMV. These tests show that the localization algorithm can reach a precision smaller than $0.2m$, which is acceptable for indoor autonomous navigation. On the other hand, the tests with the real robot show that the performance can degrade quickly if the system is not well calibrated, if the data matrices are not positioned exactly in the right position and if the robot odometry is not accurate.

An interesting continuation of this thesis would be to optimize the EKF, as suggested in section 6.1. Also, it is crucial to tackle the problems caused by systematic errors. The best way to deal with this problem could be to add an inertial unit.

Bibliography

[1] R. E. Kalman, "A new approach to linear filtering and prediction problems," *ASME Journal of Basic Engineering*, 1960.

[2] C. Suliman, C. Cruceru, and F. Moldoveanu, "Mobile robot position estimation using the kalman filter," *Scientific Bulletin of the "Petru Maior" University of Tîrgu-Mureş*, vol. 6, 2009.

[3] S. J. Julier and J. K. Uhlmann, "A new extension of the kalman filter to nonlinear systems," 1997, pp. 182–193.

[4] LAR - University of Aveiro. (2015) Atlas project. [Online]. Available: http://atlas.web.ua.pt/

[5] F. Caron, E. Duflos, D. Pomorski, and P. Vanheeghe, "Gps/imu data fusion using multisensor kalman filtering: Introduction of contextual aspects," *Inf. Fusion*, vol. 7, no. 2, pp. 221–230, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.inffus.2004.07.002

[6] J. M. Font-Llagunes and J. A. Batlle, "Consistent triangulation for mobile robot localization using discontinuous angular measurements," *Robot. Auton. Syst.*, vol. 57, no. 9, pp. 931–942, Sep. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2009.06.001

[7] V. Pierlot and M. Van Droogenbroeck, "A new three object triangulation algorithm for mobile robot positioning," *Trans. Rob.*, vol. 30, no. 3, pp. 566–577, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1109/TRO.2013.2294061

[8] C. B. Madsen and C. S. Andersen, "Optimal landmark selection for triangulation of robot position," *Journal of Robotics and Autonomous Systems*, vol. 13, pp. 277–292, 1998.

[9] L. Carrão, "Sistema de visão pare guiamento de agv em ambiente industrial," Master's thesis, University of Aveiro, 2014.

[10] R. D'Andrea, "Guest editorial: A revolution in the warehouse: a retrospective on kiva systems and the grand challenges ahead." *IEEE T. Automation Science and Engineering*, vol. 9, no. 4, pp. 638–639, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/tase/tase9.html#DAndrea12

[11] J. J. Enright, P. R. Wurman, and N. R. Ma, "Optimization and coordinated autonomy in mobile fulfillment systems."

[12] Willow Garage, University of Stanford. (2015) Ros: Robot operating system. [Online]. Available: http://www.ros.org/

[13] MATLAB. (2015) Robotics system toolbox. [Online]. Available: http://it.mathworks.com/products/robotics/

[14] M. Laughton. (2015) Libdmtx website. [Online]. Available: http://libdmtx.sourceforge.net/

[15] G. Bradski, "Opencv library," *Dr. Dobb's Journal of Software Tools*, 2000.

[16] MATLAB. (2015) Qt framework. [Online]. Available: http://www.qt.io/

[17] ISO, "Data matrix," International Organization for Standardization, Geneva, Switzerland, ISO 16022:2000, 2000.

[18] ——, "Data matrix bar code symbology specification," International Organization for Standardization, Geneva, Switzerland, ISO 16022:2006, 2006.

[19] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [Online]. Available: http://dx.doi.org/10.1137/0108018

[20] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach.* Prentice Hall Professional Technical Reference, 2002.

[21] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, ISBN: 0521540518, 2004.

[22] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, "Complete solution classification for the perspective-three-point problem," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 8, pp. 930–943, Aug. 2003. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2003.1217599

[23] G. Bowman and P. Mihelich. (2015) Camera calibration - ros package. [Online]. Available: http://wiki.ros.org/camera_calibration/

[24] A. Bradski, *Learning OpenCV, [Computer Vision with OpenCV Library ; software that sees]*, 1st ed. O'Reilly Media, 2008.

[25] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *The Quarterly of Applied Mathematics*, vol. 2, pp. 164–168, 1944.

[26] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM Journal on Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963. [Online]. Available: http://dx.doi.org/10.1137/0111030

[27] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[28] D. Stonier. (2015) Ecl linear algebra - ros package. [Online]. Available: http://wiki.ros.org/ecl_linear_algebra