POLITECNICO DI MILANO
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

A Library for Simultaneous Localization and Mapping using Data Matrix Visual Markers

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Relatore: Ing. Matteo Matteucci
Correlatore: Ing. Simone Ceriani

Tesi di Laurea di:
Andrea Premarini, matricola 750863
Andrea Scalise, matricola 749935

Anno Accademico 2010-2011

To Krzysztof Zaremba, dziękuję "dla" wszystko

Abstract

In the last twenty years, Mobile Robotics has been trying to study and build autonomous agents, able to operate within an environment without any external human help. One of the keystones to make robots really autonomous is the ability of a robot moving and localizing itself in unknown environments, in real time, using exclusively information gathered by the on board sensors. The techniques which allow such ability are commonly called SLAM (Simultaneous Localization and Mapping).

This thesis purpose is to design and build a library within the MoonSLAM framework, allowing to implement software for SLAM based on the use of visual markers as artificial landmarks, as opposite to environment natural landmarks.

Experimental tests were conducted in order to verify markers image detection through computer vision algorithms, after which it was possible to integrate such markers as features in an extended Kalman filter for SLAM algorithms execution.

Sommario

Il presente lavoro di tesi si colloca nell'ambito della robotica mobile, disciplina che, negli ultimi venti anni, si occupa di studiare e progettare agenti fisici realmente autonomi, cioè dotati di funzionalità intelligenti che li rendono in grado di agire senza bisogno dell'intervento esterno da parte dell'uomo. In questa tesi ci siamo focalizzati sulla capacità di inferire, tramite una telecamera, informazioni metriche dall'ambiente in cui si opera, di localizzarsi al suo interno senza alcuna conoscenza a priori e di costruire allo stesso tempo in modo incrementale una mappa delle regioni esplorate. L'insieme delle tecniche che permettono di implementare tali abilità viene comunemente denominato SLAM (Simultaneous Localization and Mapping).

L'obiettivo del lavoro di tesi è la progettazione e la realizzazione di una libreria per il framework MoonSLAM che consenta lo sviluppo di software per lo SLAM basato su marker visuali utilizzati come landmark artificiali, in aggiunta ai classici landmark naturali dell'ambiente.

Sono stati effettuati esperimenti per verificare il riconoscimento dei marker tramite algoritmi di computer vision, dopodichè è stata resa possibile l'integrazione di tali marker come feature di un filtro di Kalman esteso per l'esecuzione di algoritmi di SLAM.

Acknowledgements

First of all, we would like to thank professor Matteo Matteucci, who gave us the possibility to work on this project and write our Master's thesis, following us and giving us precious hints.

A special thank goes to Simone Ceriani, our co-advisor, for his infinite patience and for helping us score of times during the difficult moments.

We thank Davide Rizzi, always present in AIRLab and available to answer every our question, all the other students who spent their time in the laboratory with us during the past months, and the pizza maker from La Corte, who fed us during long work days.

Finally, we thank all our university friends, Matteo, Giuliano, Niccoló, Tommaso, Dario and Emanuele, for sharing all these university years together.

I, Andrea Scalise, would like to thank my parents, Antonio and Donatella, who sustained me during all my university years and gave me the possibility to graduate.

A particular thank is for Ania, for standing me all the time in a very difficult period and for believing in me and in my possibilities, and for her family, who always pushes me to do my best.

Thank also to all the persons who supported and encouraged me to continue along my way.

I, Andrea Premarini, am grateful to my parents, who sustained me during these years of university and gave me the possibility to finish my studies.

I would like also to thank my brothers, Pietro and Sofia, who supported me everyday with their love and their smiles.

Contents

Abstract **I**

Sommario **III**

Acknowledgements **V**

1 Introduction **1**

2 Monocular EKF-SLAM with visual markers **5**

2.1 Simultaneous Localization And Mapping 5

2.2 Kalman Filter Algorithm 8

2.2.1 State vector 8

2.2.2 Initialization 9

2.2.3 Prediction step 9

2.2.4 Feature addition 11

2.2.5 Measurement 11

2.2.6 Update step 12

2.3 Monocular SLAM with fiducial marker 13

2.3.1 Localization with Alphabet fiduciary marker 13

2.3.2 SLAM with ARToolKit Plus 14

2.3.3 SLAM with ARTag 15

3 Visual Markers **17**

3.1 Two-dimensional barcodes 17

3.1.1 Data Matrix 20

3.1.2 QR Code 34

3.1.3 Other barcodes 35

3.2 Fiducial markers 36

3.2.1 ARToolKit 37

3.2.2 ARTag 38

3.2.3 ARToolKitPlus 39

4 Data Matrix visual detection and pose estimation 41
4.1 Camera calibration 41
4.1.1 Pinhole camera model 41
4.1.2 Calibration experiments 46
4.2 Data Matrix encoding 48
4.3 Perspective N-Point 50
4.4 Data Matrix detection 53
4.4.1 *libdmtx* library 54
4.4.2 GFTT - Good Features To Track algorithm 59
4.4.3 Canny edge detection and Hough algorithms 65
4.4.4 FAST algorithm 69
4.5 Data Matrix pose quantitative analysis 71
4.5.1 Data Matrix detection - static error 72
4.5.2 Data Matrix detection - dynamic error 95

5 AIRVisualMarker library for MoonSLAM 103
5.1 MoonSLAM 103
5.1.1 Main features 103
5.1.2 External libraries 105
5.1.3 Framework architecture 106
5.1.4 EKF-SLAM in MoonSLAM 110
5.2 AIRVisualMarker library 112
5.2.1 Library structure 112
5.2.2 Marker detection 114
5.2.3 Adding markers to the filter 115
5.2.4 Marker image points measurement 116
5.2.5 Deleting a marker from the container 118
5.3 Experiments and results 119
5.3.1 Motion model 119
5.3.2 Management policies 121
5.3.3 EKF-SLAM with visual markers 124
5.3.4 Results - trajectory scaling 126
5.3.5 Results - landmarks scaling 128
5.3.6 Results - map building 132

6 Conclusions and future works 137

Bibliography 141

A Notation: 3D Rototraslations 145
A.1 Rotation 145
A.2 Translation 146
A.3 Rototranslation 147
A.4 Homogeneous coordinates 147

List of Figures

2.1 Fiduciary Marker with alphabet ‘K’and detected corners. . . . 14
2.2 ARToolKitPlus 15
2.3 Example of ARTag marker 16
2.4 Experimental Environment 16

3.1 Different kinds of stacked codes 19
3.2 Different kinds of matrix codes 20
3.3 ECC 200 Data Matrix codes 22
3.4 ECC 200 Data Matrix finder pattern 23
3.5 ECC 200 Data Matrix encoded data 24
3.6 ECC 200 Data Matrix quiet zone 24
3.7 ECC 200 Data Matrix polarities 25
3.8 ECC 100 Data Matrix 26
3.9 Codeword graphical representation 32
3.10 Data Matrix nominal layout and codeword shifts 33
3.11 Layout of a Data Matrix encoding the string “wikipedia” . . . 33
3.12 QR Code structure example 35
3.13 Some ARToolKit visual markers 37
3.14 Some ARTag visual markers 38
3.15 Some ARToolKitPlus with simple ID (upper) and BCH ID (lower) visual markers 40

4.1 Pinhole camera model 43
4.2 Camera extrinsic parameters 45
4.3 Chessboard images for camera calibration 46
4.4 MATLAB representation of camera extrinsic parameters (Camera frame) 47
4.5 MATLAB representation of camera extrinsic parameters . . . 48
4.6 Marker coordinates system 51
4.7 Marker pose estimation 54
4.8 Data Matrix parameters returned by *libdmtx* 57

4.9 Data Matrix missing corners 58
4.10 Data Matrix detection with *libdmtx* corners 59
4.11 Region-Of-Interest inside which compute GFTT algorithm . . 60
4.12 GFTT mask for bottom-left point detection 61
4.13 Linear masks for top and right edges in the first sweep 62
4.14 GFTT points detection and deletion 64
4.15 Data Matrix final contour with GFTT 64
4.16 Data Matrix final contour with GFTT - far perspective 65
4.17 ROIs around the marker borders for Canny application 67
4.18 Data Matrix edges after Canny detection 67
4.19 Hough algorithm lines selection 68
4.20 Data Matrix final contour with Canny 69
4.21 Data Matrix final contour with Canny - far perspective 70
4.22 FAST points detection 71
4.23 Data Matrix final contour with FAST algorithm 71
4.24 Marker on chessboard 73
4.25 Marker-camera configurations for static error analysis 74
4.26 Errors between chessboard and marker poses with mean (green) and median (red) values using GFTT method in lateral close configuration 76
4.27 Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in lateral close configuration 77
4.28 Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in lateral close configuration 78
4.29 Errors between chessboard and marker poses with mean (green) and median (red) values usin GFTT method in frontal close configuration 81
4.30 Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in frontal close configuration 82
4.31 Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in frontal close configuration 83
4.32 Errors between chessboard and marker poses with mean (green) and median (red) values using GFTT method in lateral far configuration 86

4.33 Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in lateral far configuration . . . 87
4.34 Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in lateral far configuration . . . 88
4.35 Errors between chessboard and marker poses with mean (green) and median (red) values using GFTT method in frontal far configuration . . . 91
4.36 Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in frontal far configuration . . . 92
4.37 Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in frontal far configuration . . . 93
4.38 Trajectory followed in the dynamic scene . . . 96
4.39 Errors between chessboard and marker poses using GFTT method in dynamic scene . . . 98
4.40 Errors between chessboard and marker poses using FAST method in dynamic scene . . . 99
4.41 Errors between chessboard and marker poses using Canny method in dynamic scene . . . 100
4.42 Chessboard (blue) and marker (red) trajectories respectively using GFTT, FAST and Canny methods . . . 102

5.1 MoonSLAM main components and relations with third-party libraries . . . 106
5.2 MoonSLAM main modules graph and dependencies between them . . . 107
5.3 AIREkfSlam library main components and variables . . . 108
5.4 Objects creations from external configuration file . . . 110
5.5 Main steps of an EKF-SLAM algorithm in the MoonSLAM framework . . . 111
5.6 Dependencies between classes within AIRVisualMarker library 113
5.7 Visual odometry example . . . 120
5.8 Inverse depth initialization . . . 124
5.9 Main steps of an EKF-SLAM algorithm in the MoonSLAM framework using visual markers . . . 125

5.10 Camera trajectory using only visual odometry (in red) and with markers information integration (in blue) - holding camera in hand (first scene) 129
5.11 Camera trajectory using only visual odometry (in red) and with markers information integration (in blue) - camera far from the scene (second scene) 130
5.12 Camera trajectory using only visual odometry (in red) and with markers information integration (in blue) - camera close to the scene (third scene) 131
5.13 Visual landmarks scaling - holding camera in hand (first scene) 133
5.14 Visual landmarks scaling - camera close to the scene (third scene) 134
5.15 Map building - camera in hand (first scene) 135
5.16 Map building - camera close to the scene (third scene) 136

List of Tables

3.1 Data Matrix high level encoding modes 27
3.2 Codewords meaning in ASCII mode 28
3.3 EDIFACT characters 29
3.4 Table of Data Matrix ECC 200 Symbol Attributes (Square form) 31
3.5 Table of Data Matrix ECC 200 Symbol Attributes (Rectangular form) 32

4.1 Percentage of detected Data Matrix markers and average time search over 1000 static frames 56
4.2 Lateral close marker-camera configuration - summary 79
4.3 Frontal close marker-camera configuration - summary 84
4.4 Lateral far marker-camera configuration - summary 89
4.5 Frontal far marker-camera configuration - summary 94

Chapter 1

Introduction

> *"Theory is when we know everything but nothing works. Praxis is when everything works but we do not know why. We always end up by combining theory with praxis: nothing works and we do not know why"*
>
> Albert Einstein

One of the keystones to make robots really autonomous is the ability of moving in unknown environments in real time using exclusively information gathered by the on board sensors. The techniques which allow such ability are commonly called SLAM (Simultaneous Localization and Mapping).

This thesis presents how SLAM is used employing a single pinhole camera and visual markers in indoor environments. SLAM is a technique to build and update a map with relative measurements and control inputs while the robot navigates an unknown environment. The solution to the SLAM is considered a key prerequisite for making a robot fully autonomous, being able to navigate in a general environment without any help from humans.

In the past years, many different algorithms have been developed to solve the SLAM problem: one that has been proved to give excellent results is based on the extended Kalman filter (EKF) [20, 42], a technique for filtering and prediction in linear Gaussian systems, extended to non linear system using Taylor linearisation or unscented transform and adapted for autonomous robots. The EKF-SLAM [10, 11] is considered by now a solved problem, and it is the method employed in this thesis within the existing framework MoonSLAM, developed at the Artificial Intelligence & Robotics laboratory of Politecnico di Milano.

The recent trends in visual SLAM has been towards the use of natural landmarks, that is intrinsic relevant features of the environment. However, one of the most critical limitation to the SLAM based on natural landmarks

is the data association problem. Several techniques have been applied to this problem, but the natural-landmark-based SLAM methods are vulnerable to dynamic and moving objects located in a captured scene, because such objects may cause incorrect landmark association. This crucial ability has been for long an obstacle for natural features-based SLAM. For this reason, researchers thought to employ a new type of landmarks.

Artificial landmarks can guarantee better localization and map building than natural landmarks, thanks to the fact that they are easily distinguishable. Lot of work has been already done in pure robot localization field using fiducial markers as artificial landmarks [28, 31], where the world map is known "a priori". The general idea behind these methods is that modifying the environment at a minor cost (visual markers can just be printed on sheets of paper), in order to give to the robot important information about the world, is an acceptable trade-off.

In recent years, the use of visual markers has been extended to SLAM applications as well [24, 43]. This expedient can give the robot useful information about its pose in the world and the environment map, because they facilitate data association. Markers generally used as landmarks are ARToolKitPlus [21] or new designed markers [39], because they are easily detectable.

Other visual markers are available, but they have been employed for other applications [34]. Between them we find Data Matrix barcodes [14]. These markers have been widely used in the past decade mostly for information purposes, e.g., identifying raw products inside industries or labelling mail parcels.

In this thesis we present a library for visual markers integration in a single camera EKF-SLAM using Data Matrix barcodes. Although Data Matrix markers are mainly used for other applications, because of their capacity to encode a high quantity of information within a very small area, we thought to extend their purposes also to robotics applications as SLAM, taking advantage of their encoding power. With the integration of visual markers in SLAM, we aimed at improving the robot localization and the map building, giving to the system information about where a marker is situated with respect to the camera.

The major advantages of this work can be summarized as follows: first, we use a cost-effective SLAM method since the system requires a single camera and visual markers easily generated using a desktop printer. Second, landmarks can be installed flexibly and randomly in the environment without any a priori knowledge. This means that it is not necessary to know the global landmark poses in world coordinates, because EKF-SLAM itself

provides a way to compute those. Third, the proposed method can help data association, since visual markers are more easily identified than natural landmarks; moreover it facilitates robot localization, being these markers a source of information regarding their relative pose in camera reference frame.

The thesis is structured as follow.

- In Chapter 2 we describe Kalman filter and SLAM in details and how they work. Then we show the background about SLAM techniques, their applications and their results.
- In Chapter 3 we talk about visual markers, describing their main applications. We analyse in particular Data Matrix barcodes, the ones we use in this thesis, explaining the current standard, their encoding power and their structure.
- In Chapter 4 we talk about Data Matrix recognition and different computer vision techniques used to improve the marker detection within camera images. We then describe how to estimate the marker position and orientation in camera reference frame and we inspect how the chosen vision techniques can affect this phase.
- In Chapter 5 we talk about our library for marker recognition and integration in SLAM application, describing how we included it in MoonSLAM framework. We show some experiments we led for camera localization and mapping and we present the correspondent results.
- In Chapter 6 we summarize the conclusions about our thesis and we discuss possible future works in the presented field.
- In Appendix A we give an overview of the mathematical notation we used in the thesis about system rototranslations.

Chapter 2

Monocular EKF-SLAM with visual markers

> *"Quando un uomo viene scaricato diventa più forte, se non hai abbastanza esperienza da poterci ridere su o da poterne trarre ispirazione... beh allora hai fallito come uomo!"*
>
> Jiraya

The issue of Simultaneous Localization and Mapping (SLAM) has received quite a lot of attention in the last decades because of its relevance for many mobility-related applications such as service robotics and intelligent transportation. In this chapter we describe first, SLAM with EKF and briefly how it works. We focus at first on SLAM with natural landmarks and then with different visual markers.

2.1 Simultaneous Localization And Mapping

The SLAM problem has been well defined by Durrant-Whyte, see e.g., [10].

> *The simultaneous localization and mapping (SLAM) asks if it is possible for a robot to be placed at an unknown location in an unknown environment and for the robot to incrementally build a consistent map of this environment while simultaneously determining its location within this map.*

It is therefore required, in SLAM, that both the trajectory of the mobile robot and the position of all scene features, i.e., the map, to be estimated online during the exploration. Pure localization can be seen as the case

where the map is known, while pure mapping as the case where the observer pose is known at all time instants. Consider a mobile robot moving in a scene and activating its sensing apparatus a discrete number of times. Each sensor activation allows to measure some of the scene features, in a reference frame relative to the robot. At each time t, we have the following relevant quantities:

- $\mathbf{\Gamma}_t$: the pose of the robot, which changes with time;
- $\mathbf{Y}$: the location of the y_i scene features, i.e., the map, which are considered static;
- $\mathbf{u}_t$: the control applied, at previous time, to move the robot to the current pose;
- $\mathbf{z}_t$: the observations of the scene features, at the current time t.

The SLAM problem can be recursively expressed in terms of the joint probability distribution of state $\mathbf{X}_t = [\mathbf{\Gamma}_t,\ \mathbf{Y}]$, as a function of the control inputs and the data collected by the observer sensorial apparatus. Such probability $P(\mathbf{\Gamma}_t,\ \mathbf{Y}|\mathbf{z}_{1:t},\ \mathbf{u}_{1:t},\ \mathbf{\Gamma}_0)$ can be determined in a recursive form and assuming a Markov property in the motion. The joint posterior, after the application of the control $\mathbf{u}_t$, and the integration of the measure $\mathbf{z}_t$, can be determined exploiting the Bayes rule as:

$$P(\mathbf{\Gamma}_t, \mathbf{Y}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) \propto P(\mathbf{z}_t|\mathbf{\Gamma}_t, \mathbf{Y}) \int P(\mathbf{\Gamma}_t|\mathbf{\Gamma}_{t\text{-}1}, u_t) P(\mathbf{\Gamma}_{t-1}, \mathbf{Y}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1}) d\mathbf{\Gamma}_{t-1} \quad (2.1)$$

This recursive formulation requires a state transition equation, and a measurement equation. The state transition equation, i.e., $P(\mathbf{\Gamma}_t|\mathbf{\Gamma}_{t-1}, \mathbf{u}_t)$, is also called *motion model* because it describes how the belief about the robot pose changes after the application of the control. The measurement equation, i.e., $P(\mathbf{z}_t|\mathbf{\Gamma}_t, \mathbf{Y})$, is also called *sensor model*, as it describes the probability of obtaining a measure $\mathbf{z}_t$ given the robot pose.

The most frequent representation of this recursive formulation is in the form of a state-space model, with Gaussian noise. This representation allows the usage of the Kalman filtering, in the so-called *extended* variant (EKF), to solve the SLAM problem. The need for the extended version of the Kalman filter is due to the non-linear nature of the dynamic part of the state transition relationship, with respect to the state variables, and also to the same non linearity in the measurement equation.

In order to understand the particularities of the SLAM problem, it is important to notice that the uncertainty in the estimates of any two scene features, observed from the same pose, is not uncorrelated, as one might

expect, because the pose where the observations have been gathered is not known. After the robot moves to a new pose and it again observes the scene, it might collect just a subset of the features previously observed. The latter observed scene features, and also the robot pose, can be updated by integrating the new measures; this in turn updates also the scene features that were seen previously, but not in the last sensor activation. Moreover, the correlations between the estimates of the features grow with subsequent observations. After some time the features will be highly correlated each other and with the robot pose, and the correlation increases with the number of observations performed [10].

The first successes with the SLAM problem were initially obtained by systems exploiting mainly laser scanners as sensors [25], but some negative aspects of such devices (cost, weight, power consumption) pushed research towards less expensive, less power-hungry, less bulky sensors. The main sensor satisfying these criteria (low cost, power, and weight) are cameras. Although appreciable under these points of view, the cameras, i.e., the data from cameras, suffer for a specific issue that made vision-based SLAM a research area widely studied in recent years: the inherent impossibility to recover both the range and the bearing of a scene feature, when using a single image.

The case of using one single camera is known as the Monocular SLAM problem. The case of using more than one camera, i.e., gathering measures on the same scene feature from more than one point of view, at the same time, is known as Stereo SLAM problem. A single camera actually allows quite accurate measurement of bearing, but it only allows an uniform uncertainty in the depth of the feature, being its position equally likely along the entire viewing ray. For this reason, in Monocular SLAM, the observed needs to gather data from different points of view, in order to develop a more peaked belief about the scene features. After the seminal work by Davison [8], we assisted to the diffusion of approaches trying to solve Monocular SLAM, subsequent works showed quite impressive results also in fields such as augmented reality [23] or 3D dense reconstruction [29].

In the realm of Bayesian filtering, Davison [8] used an Extended Kalman Filter in combination with a delayed initialization of 3D features, representing 3D points with their Euclidean coordinates. Unfortunately, delayed initialization makes difficult the data association at the beginning, i.e., when the feature has been observed very few times, which is when it is more important to get it right. To overcome the shortcomings of delayed initialization, Solà [36] proposed a non parametric approach, based on a Mixture of Gaussians. However, this solution requires the presence of multiple depth

hypotheses in the EKF filter state, and this affects the performance of the EKF. Differently from previous works, Montiel et al. [6] proposed the Unified Inverse Depth (UID) parametrization, the first instance of *inverse* parameterization of features; they demonstrated that it is possible to represent in a proper way the feature uncertainty, in an un-delayed fashion, using a single Gaussian, by changing the representation of the scene feature.

2.2 Kalman Filter Algorithm

In the previous section we have shown how it is possible to cast the SLAM problem in the realm of recursive Bayesian filtering. Here, in particular, we focus on the use of the EKF to solve the Visual SLAM problem: i.e., a specific application of the EKF to recursively estimate, from visual measurements, the joint distribution of actual camera pose and 3D points in the environment as a single multidimensional Gaussian distribution.

Next, we will briefly explain the base techniques that allow to treat the EKF-SLAM problem in real time, taking advantage from the particular structure of the problem; we will define the state vector, the prediction step equations and the measurement equations in a general form. Here we only remark two relevant concepts. Firstly, the EKF-SLAM was demonstrated able to treat the *Monocular* SLAM problem, i.e., to solve the SLAM using a single camera as source of information up to a non observable scale factor. This is done thanks to the usage of specific *parameterizations* for scene features, which allow to represent, in a proper way, the unknown depth of a point perceived from a single image. Secondly, it was shown in [38] that the usage of these parameterizations simplify the treating of multi-camera systems (e.g., a stereo or trinocular camera rig) and it allows to build a modular architecture for n-camera-SLAM systems.

2.2.1 State vector

In Monocular EKF-SLAM, the state of the filter $\mathbf{X}_t$, at time t, is composed by the actual camera pose $\mathbf{\Gamma}_t$ in a world reference frame and the map $\mathbf{Y}_t$ represented by a set of features $\mathbf{Y}_t = \mathbf{y}_i$. The camera pose $\mathbf{\Gamma}_t = [\mathbf{t}_t \mathbf{q}_t]^T$ includes a translation term $\mathbf{t}_t = [\mathbf{t}_{xt}\ \mathbf{t}_{y_t}\ \mathbf{t}_{zt}]^T$ and an orientation term $\mathbf{q}_t = [\mathbf{q}_{wt}\ \mathbf{q}_{xt}\ \mathbf{q}_{y_t}\ \mathbf{q}_{zt}]^T$, the latter being usually coded with quaternions. Other parameters $\mathbf{\Lambda}_t$, describing camera motion (e.g., tangential and rotational speed or accelerations), could be present in the EKF state as well. They could be represented with respect to the camera reference frame or the world reference frame, in accordance with the *motion model* of the system. The

complete state of the filter is thus represented as:

$$\mathbf{X} = [\ \mathbf{\Gamma}_t^T\ \mathbf{\Lambda}_t^T\ \mathbf{y}_{1t}^T\ \mathbf{y}_{2t}^T\ \dots\ \mathbf{y}_{nt}^T\ ]^T, \tag{2.2}$$

and the EKF-SLAM algorithm follows the steps described in Algorithm 2.1 to update recursively the state estimate, together with its covariance matrix $\mathbf{\Sigma}_t$. In the following, this algorithm will be explained in detail.

2.2.2 Initialization

Unless we have some prior information, at $t = 0$ we start with an empty map and the camera pose is usually considered the origin of the world ($\mathbf{\Gamma}_0 = [0, 0, 0, 1, 0, 0, 0]^T$) with no uncertainty. Indeed, if we are considering some parameters to describe the camera motion, we can initialize them at zero ($\mathbf{\Lambda}_0$), but with a non zero initial covariance, to allow the estimation of the motion of the camera in the first steps.

$$\boldsymbol{X}_0 = [\mathbf{\Gamma}_0^T, \quad \mathbf{\Lambda}_0^T], \quad \mathbf{\Sigma}_0 = \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{\Sigma}_{\mathbf{\Lambda}0} \end{bmatrix} \tag{2.3}$$

2.2.3 Prediction step

At each iteration, we perform first the prediction step of the Kalman filter; this step aims at the prediction of the next system state, using prior state values and an optional control input $\mathbf{u}$; we assume a non additive Gaussian noise $\eta \sim N(0, \mathbf{\Sigma}_\eta)$ perturbates the motion. To update the state covariance matrix we use the Jacobian matrices $\boldsymbol{F}_{\boldsymbol{X}} = \partial f(\boldsymbol{X}, \boldsymbol{u}, \eta)/\partial \boldsymbol{X}$ and $\mathbf{F}_\eta = \partial f(\boldsymbol{X}, \boldsymbol{u}, \eta)/\partial \eta$ evaluated at $\hat{\mathbf{X}}_t$, $\boldsymbol{u}_t$, and $\eta_t = 0$. Odometric information are usually used as control input $\boldsymbol{u}_t$, if available, otherwise we have a pure Visual EKF-SLAM which uses the velocity motion model represented by $\mathbf{\Lambda}_t$:

$$\hat{\boldsymbol{X}}_t = f(\boldsymbol{X}_{t-1}, \boldsymbol{u}_t, \eta_t = 0), \tag{2.4}$$

$$\hat{\mathbf{\Sigma}}_t = \boldsymbol{F}_{\boldsymbol{X}} \mathbf{\Sigma}_{t-1} \boldsymbol{F}_{\boldsymbol{X}}^T + \boldsymbol{F}_\eta \mathbf{\Sigma}_\eta \boldsymbol{F}_\eta^T \tag{2.5}$$

Since features are fixed elements in the environment the state prediction equation is simplified:

$$\begin{aligned} [\hat{\mathbf{\Gamma}}_t^T, \hat{\mathbf{\Lambda}}_t^T] &= f(\hat{\mathbf{\Gamma}}_{t-1}, \mathbf{\Lambda}_{t-1}^T, \boldsymbol{u}_t, \eta_t = 0), \\ \hat{\boldsymbol{Y}}_t &= \hat{\boldsymbol{Y}}_{t-1}, \end{aligned} \tag{2.6}$$

and the uncertainty propagation can exploit sparse Jacobians:

$$\boldsymbol{F}_{\boldsymbol{X}} = \begin{bmatrix} \frac{\partial f(\dots)}{\partial(\mathbf{\Gamma}, \mathbf{\Lambda})} & 0 \\ 0 & 0 \end{bmatrix}, \quad \boldsymbol{F}_\eta = \begin{bmatrix} \frac{\partial f(\dots)}{\partial \eta} \\ 0 \end{bmatrix}. \tag{2.7}$$

Algorithm 2.1 EKF-SLAM

1. {Initialization}
2. $t \leftarrow 0, \mathbf{X}_t \leftarrow [\mathbf{\Gamma}_0^T \ \mathbf{\Lambda}_0^T]^T, \mathbf{\Sigma}_t \leftarrow diag(\mathbf{\Sigma}_{\mathbf{\Gamma}_0}, \mathbf{\Sigma}_{\mathbf{\Lambda}_0})$
3. **loop**
4. {Prediction step}
5. $t \leftarrow t+1$
6. $\hat{\boldsymbol{X}}_t \leftarrow f(\mathbf{X}_{t-1}, \mathbf{u}_t, \eta = 0)$
7. $\hat{\mathbf{\Sigma}}_t \leftarrow \mathbf{F}_{\mathbf{X}} \mathbf{\Sigma}_{t-1} \mathbf{F}_{\mathbf{X}}^T + \mathbf{F}_\eta \mathbf{\Sigma}_\eta \mathbf{F}_\eta^T$
8. {Measurement step}
9. $\boldsymbol{e} \leftarrow \varnothing, \boldsymbol{H} \leftarrow \varnothing, \boldsymbol{R} \leftarrow \varnothing$
10. **for all** $\boldsymbol{y}_{it}$**do**
11. **if** $\exists \mathbf{z}_i$ compatible with $h_i(\hat{\mathbf{X}}_t)$ then
12. $\boldsymbol{e} \leftarrow [\boldsymbol{e}; \boldsymbol{z}_i - h_i(\hat{\mathbf{X}}_t)]$
13. $\boldsymbol{H} \leftarrow [\boldsymbol{H}; \boldsymbol{H}_{i\boldsymbol{X}}]$;
14. $\boldsymbol{R} \leftarrow diag(\boldsymbol{R}, \boldsymbol{H}_{i\nu} \ \mathbf{\Sigma}_\nu \ \boldsymbol{H}_{i\nu})$
15. **end if**
16. **end for**
17. {Update step}
18. **if** $e \neq \varnothing$ **then**
19. $\mathbf{S}_t \leftarrow \boldsymbol{H} \hat{\mathbf{\Sigma}}_t \boldsymbol{H}^T + \boldsymbol{R}$
20. $\mathbf{K}_t \leftarrow \hat{\mathbf{\Sigma}}_t \boldsymbol{H}^T \boldsymbol{S}_t^{-1}$
21. $\mathbf{X}_t \leftarrow \hat{\boldsymbol{X}}_t + \boldsymbol{K}_t \boldsymbol{e}_t$
22. $\mathbf{\Sigma}_t \leftarrow \hat{\mathbf{\Sigma}} - \boldsymbol{K}_t \boldsymbol{S}_t \boldsymbol{K}_t^T$
23. **else**
24. $\boldsymbol{X}_t \leftarrow \hat{\boldsymbol{X}}_t, \mathbf{\Sigma}_t \leftarrow \hat{\mathbf{\Sigma}}_t$
25. **endif**
26. {New feature addition}
27. **for all** *new_measurement* **s do**
28. $\boldsymbol{y}_{new} \leftarrow g(\boldsymbol{X}_t, \boldsymbol{s}, \boldsymbol{\xi})$
29. $\boldsymbol{X}_t \leftarrow [\boldsymbol{X}_t^T \ \boldsymbol{y}_{new}]^T$
30. $\mathbf{\Sigma}_t \leftarrow \begin{bmatrix} I & 0 \\ \boldsymbol{G}_{\boldsymbol{X}} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{\Sigma}_t & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} I & 0 \\ \boldsymbol{G}_{\boldsymbol{X}} & 0 \end{bmatrix}^T + \begin{bmatrix} 0 & 0 \\ 0 & \boldsymbol{G}_{\boldsymbol{\xi}} \mathbf{\Sigma}_{\boldsymbol{\xi}} \boldsymbol{G}_{\boldsymbol{\xi}}^T \end{bmatrix}^T$
31. **end for**
32. **end loop**

Taking in account this special structure allows to reduce the computational complexity of the prediction step from $O(n^{2.807})$ to $O(n)$, being n the number of features, also when the map, and thus the state vector, grows.

2.2.4 Feature addition

In SLAM we aim at the online construction of the environment map, thus we need to enlarge the filter state each time a new landmark is perceived (see steps 27-31 of Algorithm 2.1). When a new feature y_{new} is created, its initialization is computed as a function of the actual state vector and some information $\mathbf{s}$ that represents the first measure of the new landmark perturbated by Gaussian noise $\xi \sim N(0, \mathbf{\Sigma}_\xi)$, in general non additive:

$$y_{new} = g(\boldsymbol{X}_t,\ \boldsymbol{s}, \xi = 0). \tag{2.8}$$

This new feature is added to the filter

$$\boldsymbol{X}_t = [\boldsymbol{X}_t^T \boldsymbol{y}_{new}]^T, \tag{2.9}$$

and its covariance updated through the use of its Jacobians evaluated at X_t and $\xi = 0$

$$\boldsymbol{G}_{\boldsymbol{X}} = \frac{\partial g(\dots)}{\partial \boldsymbol{X}}, \quad \boldsymbol{G}_\xi = \frac{\partial g(\dots)}{\partial \xi} \tag{2.10}$$

$$\mathbf{\Sigma}_t = \begin{bmatrix} \mathbf{I} & 0 \\ \boldsymbol{G}_{\boldsymbol{X}} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{\Sigma}_t & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{I} & 0 \\ \boldsymbol{G}_{\boldsymbol{X}} & 0 \end{bmatrix}^T + \begin{bmatrix} 0 & 0 \\ 0 & \boldsymbol{G}_\xi \mathbf{\Sigma}_\xi \boldsymbol{G}_\xi^T \end{bmatrix}^T \tag{2.11}$$

2.2.5 Measurement

The standard equations of the EKF update step can be revisited in two different steps in the realm of EKF-SLAM: the *measurement step* and the *update step*. The measurement step (see steps 9-16 of Algorithm 2.1) allows, for each feature y_i, to predict the expected measure according to its measurement model $h_i(\mathbf{X}, \nu)$, being $\nu \sim N(0, \mathbf{\Sigma}_\nu)$ the associated noise model.

The total innovation $\mathbf{e}$ is computed as the difference between the expected measures and the real ones. It is composed by iteratively appending the difference between one expected measure $h_i(\hat{\boldsymbol{X}}_t)$ and its *individually compatible real measure* $\mathbf{z}_i$ whenever it exists:

$$\boldsymbol{e} \leftarrow [\boldsymbol{e}; \boldsymbol{z}_i - h_i(\hat{\boldsymbol{X}}_t)]. \tag{2.12}$$

The individual compatibility of measurements estimation is usually performed by a Mahalanobis distance test on:

$$d = (\boldsymbol{z}_i - h_i(\hat{\boldsymbol{X}}))^T \boldsymbol{S}_i^{-1} (\boldsymbol{z}_i - h_i(\hat{\boldsymbol{X}}_t)), \tag{2.13}$$

with

$$\boldsymbol{S}_i = \boldsymbol{H}_{i\boldsymbol{X}} \hat{\boldsymbol{\Sigma}}_t \boldsymbol{H}_{i\boldsymbol{X}}^T + \boldsymbol{H}_{i\nu} \hat{\boldsymbol{\Sigma}}_\nu \boldsymbol{H}_{i\boldsymbol{\nu}}^T, \tag{2.14}$$

and

$$\boldsymbol{H}_{i\boldsymbol{X}} = \frac{\partial h_i(\dots)}{\partial \boldsymbol{X}}, \quad \boldsymbol{H}_{i\nu} = \frac{\partial h_i(\dots)}{\partial \xi}, \tag{2.15}$$

are the Jacobian matrices of $h_i(\cdot)$ with respect to the state and noise. Alike the innovation vector, Jacobian matrices are properly stacked to $\mathbf{H}$ and to the block diagonal matrix $\mathbf{R}$:

$$\boldsymbol{H} = [\boldsymbol{H}; \boldsymbol{H}_{i\boldsymbol{X}}], \tag{2.16}$$

$$\boldsymbol{R} = diag(\boldsymbol{R}, \boldsymbol{H}_{i\nu} \boldsymbol{H}_{i\nu} \boldsymbol{H}_{i\nu}^T). \tag{2.17}$$

2.2.6 Update step

Thanks to the measurement step, a set of observation was created and the resuming innovation is stored in the $\mathbf{e}$ vector with the correspondent matrixes $\mathbf{H}$ and $\mathbf{R}$. The Kalman gain is calculates as

$$\boldsymbol{S}_t = \boldsymbol{H}_{\hat{\boldsymbol{X}}_t} \hat{\boldsymbol{\Sigma}}_t \boldsymbol{H}_{\hat{\boldsymbol{X}}_t}^T + \boldsymbol{R}, \tag{2.18}$$

$$\boldsymbol{K}_t = \hat{\boldsymbol{\Sigma}}_t \boldsymbol{H}_{\hat{\boldsymbol{X}}_t}^T \boldsymbol{S}_t^{-1}, \tag{2.19}$$

and the filter state is updated accordingly to

$$\boldsymbol{X}_t = \hat{\boldsymbol{X}}_t + \boldsymbol{K}_t \boldsymbol{z}_t, \tag{2.20}$$

$$\boldsymbol{\Sigma}_t = \hat{\boldsymbol{\Sigma}}_t - \boldsymbol{K}_t \boldsymbol{S}_t \boldsymbol{K}_t^T \tag{2.21}$$

Notice that, in case there are no measurements available (i.e., the innovation vector is empty), the update step is skipped and the prediction state is assumed as the new state. There might be alternatives way to perform EKF update. For instance, it is possible to iterate the update step in the following way: (1) sort the $\mathbf{e}$ vector in decreasing order of innovation (evaluated with 2.12); (2) integrate one measure at time recomputing the Jacobians at each step. This procedure increases the complexity of the update step, but provides a better linearization of Jacobians. This approach is highly sensitive to outliers, e.g., wrong data associations, so it is not practical on real data. Indeed, in real cases the data association plays a key role to avoid wrong results. Different techniques were developed to overcome this situation, in particular one of the simplest and effective is the 1-Point RANSAC technique presented in [7] that executes two update steps on two disjoint set of measurement characterized by low and high innovation after a selection of outliers.

2.3 Monocular SLAM with fiducial marker

Monocular SLAM is a method that uses as acquisition sensor only a single camera, being camera cheap, light-weight, passive, low power requirements and provide high-resolution images. The first SLAM systems were based on the detection of natural landmarks, that is intrinsic relevant features of the environment. One of the most critical limitation of the SLAM based on natural landmarks is the data association problem. The data association is the issue of matching current observations with previously obtained observations from a captured scene. Natural landmarks based SLAM methods are vulnerable to moving objects, because the moving objects cause miss data association.

An other issue with natural landmarks, is that in monocular SLAM is not possible to evaluate correctly the depth of the objects in the image, this can cause errors in the evaluation of the route of the robot and in the mapping of the environment. These crucial problems have been an obstacle of industrialization of SLAM into robots. To solve issues the natural landmarks, the researchers started to solve SLAM problem with the use of fiducial markers.

Fiducial markers are objects placed within a scene to provide visual cues, or reference points, such that a computer vision system can easily isolate the object's location. They are used to supply reliable position estimates for robotic and image registration applications. Fiducial markers can contain several informations that can be used from SLAM system. We report in the following a set of example of monocular SLAM with fiducial markers.

2.3.1 Localization with Alphabet fiduciary marker

In project [22] the fiduciary marker consists of a black quadrilateral border used as a marker recognition beacon, and an alphabet letter as shown in Figure 2.1. There are 8 feature points, highlighted by the orange circles in Figure 2.1, which are the corners of the two rectangles those points are used for calculation of 6-DOF camera pose. The marker detection and letter recognition process works as follows. First, an image is captured by the vision sensor and the captured image is binarized with adaptive thresholding method. Then, the binarized image is color-inverted and if a white blob is bigger than certain pixels and has a proper ratio $r(height/width)$, then the area is labeled. As it is shown in Figure 2.1, since the markers are composed of three blobs, the outer black border, the inner white rectangle, and the alphabet letter, by doing blob labeling three times hierarchically, the marker could be reliably detected even in a messy indoor environment.

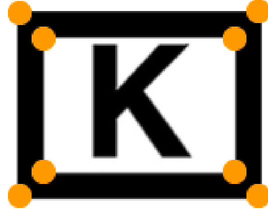


Figure 2.1: Fiduciary Marker with alphabet 'K'and detected corners.

If the marker is detected, the 8 corners of the marker are found. By using the planar homography transform technique with the 8 correspondences, the rotation vector and the translation vector are calculated and the 6-DOF pose of camera with respect to the marker coordinate is obtained. If the world coordinate of each marker has been known, the world coordinate of the camera can be deduced. The letter inside the black contour is then recognized by an Artificial Neural Networks, that in this project could identify the correct letter with percentage higher than 10%.

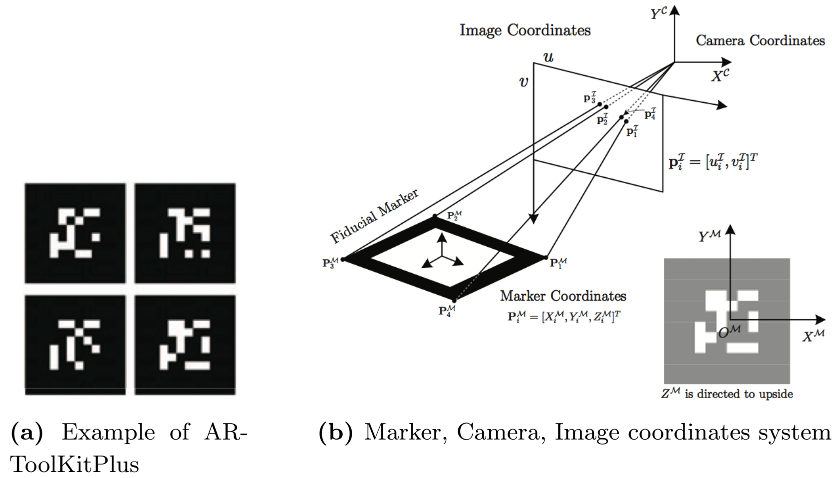
2.3.2 SLAM with ARToolKit Plus

Project [24] considers basic fiducial markers shown in Figure 2.2a. Fiducial markers consists of four vertices and an identification code at the center of a marker. The number of markers are bounded by the 10 bits of identification code, so markers contain 10-bit binary number. The colours of fiducial markers are black and white, in such a way that it is easy to distinguish the binary code and the vertices can be easily extracted from line crossing thanks to its square shape. The fiducial marker coordinates are shown in Figure 2.2b, the position of a marker from camera center is computed from transformation of coordinates between marker and camera.

$\mathbf{P_i} = [X, Y, Z]^T$ is the variable that denotes vertices on the marker in the global coordinates. If there are four points on a plane, it is assumed that the 3D coordinates of known four points are on the Z=0 plane without lack of generality. When a fiducial marker is captured from a camera, the four vertices of a marker are projected to the 2D image. Without entering in details, the steps to extract marker vertices and corners are the followings:

1. Thresholding of the image with threshold.
2. Get connected components from the thresholded image.
3. Get contours from the connected components image.
4. Extract four vertices from the contour image.

The coordinates that are extracted are used to add a new landmark into the



(a) Example of ARToolKitPlus

(b) Marker, Camera, Image coordinates system

Figure 2.2: ARToolKitPlus

SLAM EKF filter or just to update position about existing landmarks and camera. For the experiments several markers have been positioned on a wall of a corridor, the robot was able to recognize all the marker and to travel in the corridor.

2.3.3 SLAM with ARTag

The goal of project [43] is to construct a 3D map of the landmarks in the environment and estimate the path taken by an indoor blimp. ARTag, Figure 2.3, are used as visual markers. ARTag system can read 3D position and attitude of a marker. When a camera detects ARTag markers, pose and attitude between camera and marker is obtained.

The steps that are followed are similar to Algorithm 2.1. The environment (Figure 2.4b) is composed by 15 landmarks on the floor, five of whom have known coordinates and IDs. A blimp, Figure 2.4a, flies over the floor in order to see all the landmarks and to draw a map of the environment. The experiment has verified that is possible to construct a 3D map of the landmarks and the path taken by the indoor blimp by assuming that the blimp at time t would move in the same way as time $t-1$.

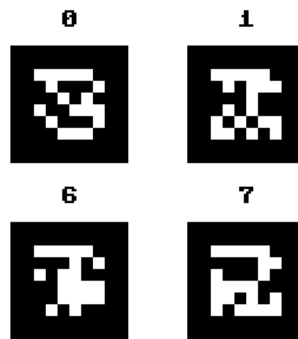


Figure 2.3: Example of ARTag marker

(a) Indoor Blimp

(b) ARTag on the floor

Figure 2.4: Experimental Environment

Chapter 3

Visual Markers

> *"Il fatto che a me o a tutti sembra che le cose stiano così non significa necessariamente che stiano così; ma il vero interrogativo è se abbia senso dubitarne"*
>
> Il teorema del Pappagallo - Denis Guedj

In this chapter we analyse the different types of visual markers and their most typical applications. First, we introduce the two-dimensional barcodes' technologies, which are the most common codes, what they are useful for and how they encode information, in particular we talk about Data Matrix barcodes, which we used while building the library for robot localization and mapping described in further chapters. Secondly we talk about the classical fiducial markers used more often in robotics for indoor and outdoor localization and augmented reality applications. Each of the markers described in this chapter could potentially be used for robot SLAM in the library we implemented, just by adding a function able to decode the chosen marker.

3.1 Two-dimensional barcodes

The first barcodes invented were represented in just one dimension: they were composed of lots of vertical lines, more or less close to each other. This means that an ordinary barcode is *vertically redundant*, meaning that the same information is repeated vertically (since it is a one-dimensional code, the information lays on the horizontal dimension). The heights of the bars can be truncated without any lose of information. However, the vertical redundancy allows a symbol with printing defects, such as spots or voids, to still be read. The higher the bar heights, the more probability that at least one path along the bar code will be readable.

A two-dimensional code stores information along the height as well as the length of the symbol, making possible to encode more data than the correspondent one-dimensional codes. For example, all human alphabets are considered two-dimensional codes. Since both dimensions contain information, at least some of the vertical redundancy is gone, but other techniques must be used to prevent misreads. However, misread prevention is relatively easy: most two-dimensional codes use check words to insure accurate reading.

Initially, two-dimensional symbologies were developed for applications where only a small amount of space was available for an automatic ID symbol. The first application for such symbols was unit-dose packages in the healthcare industry. These packages were small and had little room to place a barcode. The electronics industry also showed an early interest in very high density barcodes and two-dimensional symbologies, since free space on electronics assemblies was limited.

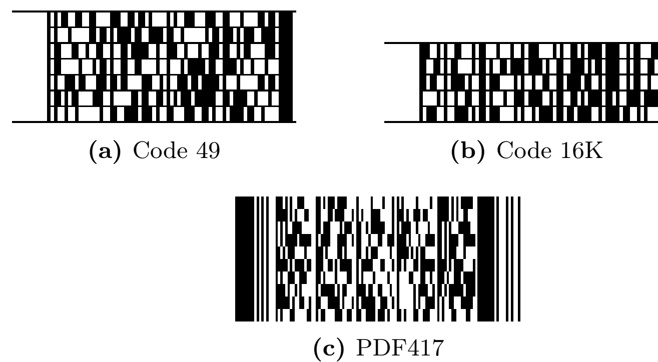
More recently, they can also be used to keep track of objects and people: they are used to keep track of rental cars, airline luggage, nuclear waste, registered mail, express mail and parcels. Barcoded tickets allow the holder to enter sports arenas, cinemas, theatres and transportation (i.e. trains), and are used to record the arrival and departure of vehicles from rental facilities etc. This can allow proprietors to identify duplicate or fraudulent tickets more easily. Barcodes are widely used in shop floor control applications software where employees can scan work orders and track the time spent on a job.

Nowadays, thanks to the smartphones technology, people is able to read a wide variety of these codes directly from a cell phone: so these 2D barcodes can contain advertisements, societies contact information, website addresses, and so on, which can directly embed a hyperlink to the very internet page, allowing the user to easily reach the information from the phone terminal.

Two Dimensional bar codes have been developed, placed in the public domain, and published as AIM standards since 1987. These codes can be classified into two groups:

- *stacked codes*, sometimes referred to as stacked linear, shown in Figure 3.1, which refer to those symbologies made up of a series of one-dimensional bar codes;
- *matrix codes*, shown in Figure 3.2 , which code the data based on the position of black spots within a matrix.

Briefly talking about the most popular stacked codes, Code 49, shown in Figure 3.1a, was the first 2D stacked bar code invented by David Allais in



(a) Code 49 **(b)** Code 16K

(c) PDF417

Figure 3.1: Different kinds of stacked codes, encoding the string *"Politecnico di Milano"*

1987 at the Intermec Corporation to fill a need to pack a lot of information into a very small symbol, followed by Code 16K, in Figure 3.1b, developed by Ted Williams in 1989 at Laserlight Systems. PDF417, in Figure 3.1c, developed by Ynjiun Wang in 1991 at Symbol Technologies, now owned by Motorola, is also a stacked code, but it can accommodate a much higher data capacity (over 1800 characters) than Code 49 or Code 16K, which can encode around 50 characters given reasonable symbol density.

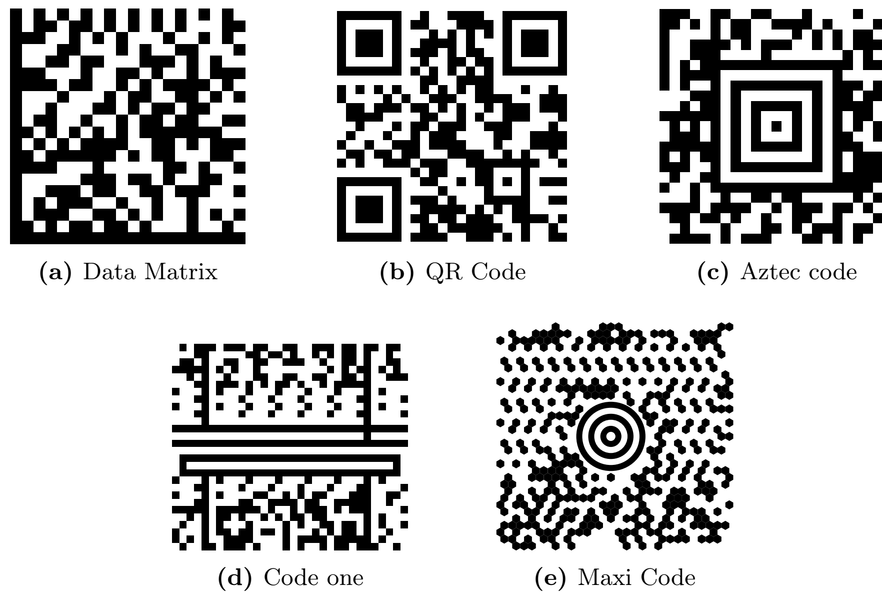
Matrix codes, on the other hand, also can hold large amounts of data, but within a much smaller size. These codes require reading by a 2D imaging device, like Charge Coupled Device(CCD) scanners and cameras.

Between the most popular code of this type, we can find Data Matrix (Figure 3.2a), QR Code (Figure 3.2b), MaxiCode (Figure 3.2e), Aztec Code (Figure 3.2c), and Code One (Figure 3.2d).

Reduced Quiet Zones, i.e. blank areas around markers, for matrix codes (except for Aztec Code and Code One in which none is required) allow symbol location in much closer proximity to surrounding text and/or graphics.

Each 2D matrix code has a unique *finder pattern* or recognition pattern that references the symbol type for the scanner. MaxiCode has three concentric circles or a "bulls-eye" pattern; Data Matrix uses an "L" pattern bordering the left and bottom edge; Aztec Code uses a square bulls-eye; QR Code uses three square patterns on the corners of the code; and Code One uses a series of horizontal bars (from three to six depending on the version) and from two to fourteen vertical recognition patterns.

We introduced only some of the matrix codes, since they can be used more easily and efficiently for the robotics purposes discussed further in this thesis, and in particular we described in detail the Data Matrix markers, because



(a) Data Matrix (b) QR Code (c) Aztec code (d) Code one (e) Maxi Code

Figure 3.2: Different kinds of matrix codes, encoding the string *"Politecnico di Milano"*

we used them to develop our robot localization-and-mapping application.

3.1.1 Data Matrix

In its latest version, Data Matrix is a matrix (2D or two-dimensional) barcode which may be printed as a square or rectangular symbol made up of a series of equally spaced square dots. This representation is an ordered grid of dark and light dots bordered by a finder pattern. The finder pattern is partly used to specify the orientation and structure of the symbol. The data is encoded using a series of dark or light dots based upon a pre-determined size. The minimum size of these dots is known as the X-dimension.

Since the information is encoded by absolute dot position rather relative dot position, this type of code is not as susceptible to printing defects as is traditional bar code. The coding scheme has a high level of redundancy with the data scattered throughout the symbol. According to the company, this allows the symbol to be read correctly even if part of it is missing.

In general, a Data Matrix square symbol can store up to 3116 characters from the entire ASCII character set (with extensions), up to 2335 alphanumeric characters and up to 1556 bytes. As more data is encoded in the symbol, the number of cells (rows and columns) increases. Symbol sizes vary from 10×10 to 144×144. Instead, a rectangular symbol can store up

to 98 characters from the entire ASCII character set (with extensions),and up to 72 alphanumeric character, with symbol sizes varying from 8×18 to 16×48.

Applications

The most popular application for Data Matrix is marking small items, due to the code's ability to encode fifty characters in a symbol that is readable at 2 or 3 mm^2. The Data Matrix is scalable, with commercial applications as small as 300 micrometres and as large as a 1 meter square.

Data Matrix codes are becoming common on printed media such as labels and letters. The code can be read quickly by a barcode reader which allows the media to be tracked, for example when a parcel has been dispatched to the recipient.

For industrial engineering purposes, Data Matrix codes can be marked directly onto components, ensuring that only the intended component is identified with the Data Matrix encoded data. The codes can be marked onto components with various methods, but within the aerospace industry these are commonly industrial ink-jet, dot-peen marking, laser marking, and electrolytic chemical etching (ECE). These methods give a permanent mark which should last the lifetime of the component.

Data Matrix codes, along with other Open Source codes such as 1D Barcodes can also now be read with mobile phones, simply by downloading the application to compatible mobile phones. Although the majority of these mobile readers are capable of reading Data Matrix, only a few can extend the decoding to enable mobile access and interaction, whereupon the codes can be used securely and across media; for example, in track and trace, anti-counterfeit and banking solutions.

ISO standards

Data Matrix was invented in 1989 by International Data Matrix, Inc. (ID Matrix) which was merged into RVSI/Acuity CiMatrix, who were acquired by Siemens AG in October, 2005 and Microscan Systems in September 2008. Data Matrix is covered today by several ISO/IEC standards and is in the public domain for many applications, which means it can be used free of any licensing or royalties:

- ISO/IEC 16022:2006 - Data Matrix bar code symbology specification. This is the original standard, and it is available to the public under payment;

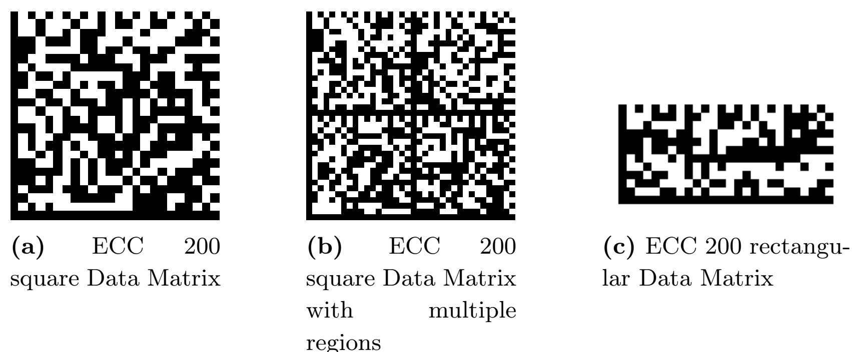
- ISO/IEC 15415 - 2D Print Quality Standard:
- ISO/IEC 15418:2009 - Symbol Data Format Semantics (GS1 Application Identifiers and ASC MH10 Data Identifiers and maintenance) [14];
- ISO/IEC 15424:2008 - Data Carrier Identifiers (including Symbology Identifiers) [IDs for distinguishing different bar code types];
- ISO/IEC 15434:2009 - Syntax for high-capacity ADC media (format of data transferred from scanner to software, etc.);
- ISO/IEC 15459 - Unique Identifiers.

However, it must be said that although this is a free standard, there are no free documents that explain the encoding process. All the technical specifications about Data Matrix can be purchased at ISO/IEC 16022:2006's website[1].

ECC 200 version

ECC (Error Checking and Correction) 200 is the newest version of Data Matrix and supports advanced encoding error checking and correction algorithms (such as Reed-Solomon). ECC 200 allows the routine reconstruction of the entire encoded data string when the symbol has sustained 30 percent damage, assuming the matrix can still be accurately located. Data Matrix has an error rate of less than 1 in 10 million characters scanned.

In Figure 3.3 you can see different types of ECC 200 version Data Matrix (the classical one in Figure 3.3a).



(a) ECC 200 square Data Matrix

(b) ECC 200 square Data Matrix with multiple regions

(c) ECC 200 rectangular Data Matrix

Figure 3.3: ECC 200 Data Matrix codes

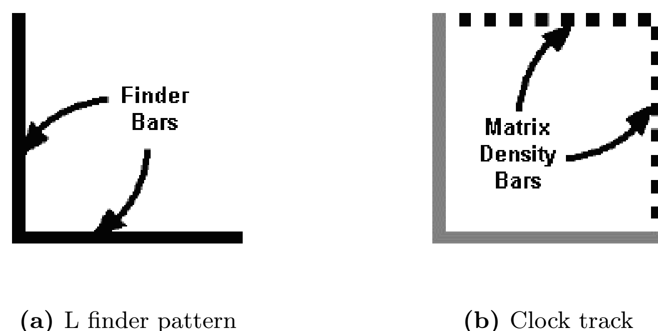
[1]http://www.iso.org/iso/catalogue_detail.htm?csnumber=44230

Data Matrix ECC 200 is composed of two separate parts: the finder pattern, which is used by the scanner to locate the symbol, and the encoded data itself. The finder pattern defines the shape (square or rectangle), the size, X-dimension and the number of rows and columns in the symbol and allows the scanner to identify the symbol as a Data Matrix.

- The solid dark is called the *L finder pattern* (Figure 3.4a). It is primarily used to determine the size, orientation and distortion of the symbol.
- The other two sides of the finder pattern are alternating light and dark elements, known as the *Clock Track* (Figure 3.4b). This defines the basic structure of the symbol and can also help determine its size and distortion.

The data is then encoded in a matrix within the Finder pattern, as you can see in Figure 3.5. This is a translation into the binary Data Matrix symbology characters (numeric or alphanumeric). As explained further in this chapter, the encoding is done in two stages: first the data are converted to 8 bits *codeword* (High level encoding), then those are converted to small black and white squares (Low level encoding). Moreover an error correction system is included, it allows to reconstitute badly printed, erased, fuzzy or torn off data.

Just like linear (1D) bar codes Data Matrix has a mandatory Quiet Zone (see Figure 3.6). This is a light area around the symbol which must not contain any graphic element which may disrupt reading the bar code. It has



(a) L finder pattern **(b)** Clock track

Figure 3.4: ECC 200 Data Matrix finder pattern

Figure 3.5: ECC 200 Data Matrix encoded data

a constant width equal to the X-dimension of the symbol on each of the 4 sides. Each Data Matrix symbol is made up of number of rows and columns. In version ECC 200, the number of rows and columns is always an even number. Most of the symbols are square with sizes from 10×10 to 144×144. Some symbols however are rectangular with sizes from 8×18 to 16×48 (even values only, see Figure 3.3c). Larger square ECC 200 symbols, those requiring at least 32 rows and 32 columns, will include alignment patterns to separate the data regions, as it is visible in Figure 3.3b. Symbols with more than 104 rows and columns have 36 data regions. All symbols utilizing the ECC 200 error correction can be recognized by the upper right corner module being the same as the background color (binary 0). Therefore ECC 200 always has a light "square" in the upper right hand right corner (circled in the figure above). Obviously, this corner will be dark if the Data Matrix symbol is printed in negative (complementary colors): in fact, Data Matrix codes can be represented with black square dots and black finder pattern on a white background (black-on-white polarity), or the other way round (white-on-black polarity), like it is shown in Figure 3.7.

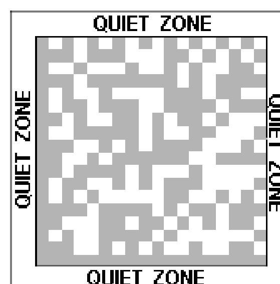
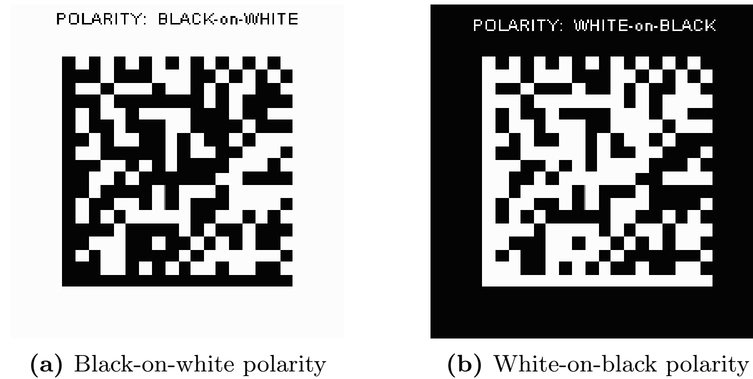


Figure 3.6: ECC 200 Data Matrix quiet zone



(a) Black-on-white polarity **(b)** White-on-black polarity

Figure 3.7: ECC 200 Data Matrix polarities

ECC 000 - 140 version

Older versions of Data Matrix include ECC 000, ECC 050, ECC 080, ECC 100, ECC 140. Each of these varies in the amount of error correction they offer, with ECC 000 offering none, and ECC 140 offering the greatest. Some of the properties cited for the ECC 200 version still hold for these old ones, like the presence of the finder pattern, of the quiet zone and the possibility of colors swap, but there are some major differences: these older versions always have an odd number of modules, and can be made in sizes ranging from 9×9 to 49×49, which means that no rectangular shape is allowed. Moreover, all symbols utilizing the ECC 000 through 140 error correction can be recognized by the upper right corner module being the inverse of the background color (binary 1), and do not use Reed-Solomon, but an algorithm called Convolutional Coding for error correction.

According to ISO/IEC 16022, "ECC 000 - 140 should only be used in closed applications where a single party controls both the production and reading of the symbols and is responsible for overall system performance". In Figure 3.8 is shown an example of an old version of Data Matrix (ECC 100).

In this thesis we did not consider such older versions, but we described in detail only the ECC 200 version, which is also the one we used in our application for the SLAM algorithm cited further in this thesis.

High level encoding

The encoding is done in two stages: first the data are converted to 8 bits *codeword* (High level encoding), then those are converted to small black and white squares (Low level encoding). Moreover an error correction system is

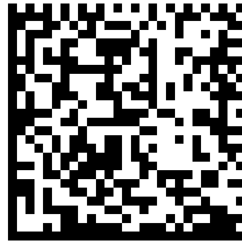


Figure 3.8: ECC 100 Data Matrix

included, it allows to reconstitute badly printed, erased, fuzzy or torn off data.

The general version Data Matrix ECC 200 supports various encoding structures, or *modes*, which can be used in the same symbol simultaneously. Examples include: ASCII (divided in three sub-modes), C40, Text, X12, EDIFACT and Base 256. These structures provide an opportunity to maximize efficiency of encoding the required data in a Data Matrix symbol.

Briefly, ASCII is used to encode data that mainly contains ASCII characters in the base version (0-127) or in the extended version (128-255), as well as ASCII digits. It encodes approximately one alphanumeric or two numeric characters per byte. As a general rule, ASCII mode is used to encode text that includes upper-case and lower-case letters with or without numbers and punctuation.

C40 is used to encode data that contains only numeric and upper-case characters. C40 encodes approximately three alphanumeric data characters into two bytes.

TEXT is used to encode data that mainly contains numeric and lower-case characters. TEXT encodes approximately three alphanumeric data characters into two bytes.

X12 is used to encode ANSI X12 characters. X12 encodes approximately three alphanumeric data characters into two bytes.

EDIFACT mode uses six bits per character, with four characters packed into three bytes. It can store digits, upper-case letters, and many punctuation marks, but has no support for lower-case letters. EDIFACT encodes approximately four data characters into three bytes.

BASE 256 is used to encode images, double-byte characters, binary data and 8 bit values.

The default character encoding method is ASCII, but some special codewords allow to switch between the other encoding methods. In Table 3.1 you can see a summary of the cited encoding modes.

Encoding mode	Data to encode	Encoding rate
ASCII	ASCII character 0 to 127	1 character per codeword
ASCII extended	ASCII character 128 to 255	0.5 character per codeword
ASCII numeric	ASCII digits 00 to 99	2 character per codeword
C40	Upper-case alphanumeric	1.5 character per codeword
Text	Lower-case alphanumeric	1.5 character per codeword
X12	ANSI X12	1.5 character per codeword
EDIFACT	ASCII character 32 to 94	1.33 character per codeword
Base 256	ASCII character 0 to 255	1 character per codeword

Table 3.1: Data Matrix high level encoding modes

The result of such operations is to get an array of 8 bits codewords directly from the data. So every codeword represents a non-negative integer number, in the range [0, 255] (note that $2^8 = 256$ in binary language), that has some particular meaning, according to which of the encoding modes is used.

ASCII mode In Table 3.2 is reported how to interpret the bytes in the range [0, 255], i.e. the codewords, for the default ASCII mode, which is divided into three sub-modes.

So it is straightforward to compute a codeword(CW) given a particular data:

- ASCII character in the range 0 to 127

$$CW = \text{“ASCII value”}{+}1$$

- Extended ASCII character in the range 128 to 255
  These characters are represented with two codewords:

$$CW1 = 235$$
$$CW2 = \text{“ASCII value”}{-}127$$

- Pair of digits in the range 00 to 99

$$CW = \text{“Pair of digits numerical value”}{+}130$$

Codeword	Data or function encoded
0	Not used
1 to 128	ASCII data
129	Padding - end of message
130 to 229	Pair of digits: 00 to 99
230	Switch to C40 encoding method
231	Switch to Base 256 encoding method
232	FNC1 character
233	Allows a message to be split across multiple symbols
234	Reader programming
235	Shift to extended ASCII encoding method for one character
236	05 Macro
237	06 Macro
238	Switch to ANSI X12 encoding method
239	Switch to TEXT encoding method
240	Switch to EDIFACT encoding method
241	Extended Channel Interpretation code
242 to 253	Not used
254	If ASCII method is in force: End of data, next CWs are padding; If other method is in force: Switch back to ASCII method or indicate end of data
255	Not used

Table 3.2: Codewords meaning in ASCII mode

Text modes: C40, TEXT, X12 The C40, Text and X12 modes are potentially more compact for storing text messages. They use character codes in the range [0, 39], so totally just 40 character. C40 and TEXT modes are similar: only upper-case and lower-case characters are inverted. In these modes 3 data characters are compacted in two codewords. In C40 and TEXT modes 3 shift characters allow to indicate an other character set for the next character.

The 16 bits value of a codeword pair is computed as following:

$$Value = C1 * 40^2 + C2 * 40^1 + C3 * 40^0$$

$$CW1 = \left\lfloor \frac{Value}{256} \right\rfloor$$

$$CW2 = Value \mod 256$$

EDIFACT value	ASCII value	Comment
0 to 30	64 to 94	EDIFACT value = ASCII value - 64
31		End of data, return to ASCII mode
32 to 63	32 to 63	EDIFACT value = ASCII value

Table 3.3: EDIFACT characters

where C1, C2 and C3 are the 3 character values to compact.

The resulting value of CW2 is in the range [0, 249]. The special value 254 is used to return to ASCII encoding mode, except if this mode allows to fill completely the symbol.

EDIFACT mode In this mode four data characters are compacted into three codewords. Each EDIFACT character is coded with 6 bits which are the 6 last bits of the ASCII value. It can store digits, upper-case letters, and many punctuation marks, but has no support for lower-case letters. Table 3.3 shows the EDIFACT characters relation with the ASCII values.

The 24 bits value of the three codewords is computed as following:

$$Value = C1 * 64^3 + C2 * 64^2 + C3 * 64^1 + C4 * 64^0$$

$$CW1 = \left\lfloor \frac{Value}{65536} \right\rfloor$$

$$CW2 = \left\lfloor \frac{Value \mod 65536}{256} \right\rfloor$$

$$CW3 = Value \mod 256$$

where C1, C2, C3 and C4 are the 4 character values to compact.

BASE 256 mode This mode can encode any byte. After the 231 codeword which switches to Base 256 mode, there is a length field, build with 1 or 2 bytes, followed by data bytes. Let N the number of data to encode:

- if $N < 250$, a single byte is used, its value is N (from 0 to 249);
- if $N >= 250$, two bytes are used:

$$L1 = \left\lfloor \frac{N}{250} \right\rfloor + 249$$

$$L2 = N \mod 250$$

It is desirable to avoid long strings of zeros in the coded message, because they become large blank areas in the Data Matrix symbol, which may cause a scanner to lose synchronization. (The default ASCII encoding does not use zero for this reason.) In order to make that less likely, the length and data bytes are obscured by adding a pseudo-random value R(n), where n is the position in the byte stream, so that each bit b_n is transformed in another bit v_n.

$$R(n) = (149 * n) \mod 255 + 1$$
$$v_n = (b_n - R(n)) \mod 256$$

Low level encoding

Once data to encode have been turned into an array of bytes, the low level encoding part allows to put these bytes in a graphical form, which will represent the final Data Matrix. This procedure is non-trivial and depends from the code symbol size.

Before explaining how this procedure is carried on, we show the detailed graphical properties of the Data Matrix codes in Table 3.4 and Table 3.5, respectively for square and rectangular shapes.

In general, symbol size (including the finder pattern, but without considering the needed quiet zone around the code) can vary between 10×10 and 144×144 for the square form, and between 8×18 to 16×48 for the rectangular form.

A symbol consists of one or several data regions, within the finder pattern, depending on how many data are needed to encode. Each region has a one module wide perimeter.

Independently of the number of regions, there is one, and only one, mapping matrix. The size of such matrix is:

"region size" per "number of regions"

Examples for 36×36 and 16×48 symbols:

"16×16" multiplied by "2×2" regions: matrix size is 32×32
"14×22" multiplied by "1×2" regions: matrix size is 14×44

The other fields of the tables are related to the error correction method used (Reed-Solomon algorithm in ECC 200 version, using the Galois field $\mathbb{F}_{256}$), indicating how many codewords are present in the matrix, how they are split in "data" and "error" bytes, and which percentage of codewords is used the error correction.

As we said, a codeword is a 8 bits word, which encode a part of the original data, like a digit or an ASCII character. Inside a Data Matrix a

Symbol Size	Data Region		Mapping Matrix Size	Total Codewords		Max. Data Capacity		% of Codewords used for Error Correction
	Size	Num.		Data	Error	Num.	Alphanum	
10×10	8×8	1	8×8	3	5	6	3	62.5
12×12	10×10	1	10×10	5	7	10	6	58.3
14×14	12×12	1	12×12	8	10	16	10	55.6
16×16	14×14	1	14×14	12	12	24	16	50
18×18	16×16	1	16×16	18	14	36	25	43.8
20×20	18×18	1	18×18	22	18	44	31	45
22×22	20×20	1	20×20	30	20	60	43	40
24×24	22×22	1	22×22	36	24	72	52	40
26×26	24×24	1	24×24	44	28	88	64	38.9
32×32	14×14	4	28×28	62	36	124	91	36.7
36×36	16×16	4	32×32	86	42	172	127	32.8
40×40	18×18	4	36×36	114	48	228	169	29.6
44×44	20×20	4	40×40	144	56	288	214	28
48×48	22×22	4	44×44	174	68	348	259	28.1
52×52	24×24	4	48×48	204	84	408	304	29.2
64×64	14×14	16	56×56	280	112	560	418	28.6
72×72	16×16	16	64×64	368	144	736	550	28.1
80×80	18×18	16	72×72	456	192	912	682	29.6
88×88	20×20	16	80×80	576	224	1152	862	28
96×96	22×22	16	88×88	696	272	1392	1042	28.1
104×104	24×24	16	96×96	816	336	1632	1222	29.2
120×120	18×18	36	108×108	1050	408	2100	1573	28
132×132	20×20	36	120×120	1304	496	2608	1954	27.6
144×144	22×22	36	132×132	1558	620	3116	2335	28.5

Table 3.4: Table of Data Matrix ECC 200 Symbol Attributes (Square form)

codeword is placed in a 3×3 square with a corner taken out. Each bit is referenced with a pixel (placed at the top-left corner). These pixels are ordered right to left, bottom to top, and a colored pixel means that the correspondent codeword bit is set to 1, otherwise is set to 0. The origin pixel of a byte is at its lower right, corresponding to bit 0. In Figure 3.9 there is an example of what we just explained: the codeword encoded is the string 10001101, which in decimal base is $128 + 8 + 4 + 1 = 141$.

With the graphical representation described, each codeword is then placed in the matrix on 45 degree parallel diagonal lines, starting at the upper left of the matrix and then the rest spread out from there on the lattice spanned by vectors [2, -2] and [3, 1]. The nominal layout resulting is shown in Fig-

Symbol Size	Data Region		Mapping Matrix Size	Total Codewords		Max. Data Capacity		% of Code-words used for Error Correction
	Size	Num.		Data	Error	Num.	Alphanum	
8×18	6×16	1	6×16	5	7	10	6	58.3
8×32	6×14	2	6×28	10	11	20	13	52.4
12×26	10×24	1	10×24	16	14	32	22	46.7
12×36	10×16	2	10×32	12	18	44	31	45.0
16×36	14×16	2	14×32	32	24	64	46	42.9
16×48	14×22	2	14×44	49	28	98	72	36.4

Table 3.5: Table of Data Matrix ECC 200 Symbol Attributes (Rectangular form)

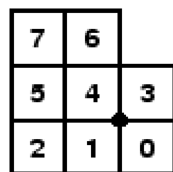
ure 3.10a.

Bytes that do not fit inside the symbol wrap around on the other side, with a shift as indicated by arrows in Figure 3.10b (other shifts are possible).

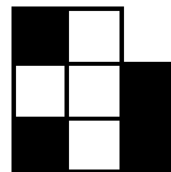
It must be noted that the total number of codewords per symbol is equal to the number of cells in the matrix divided by 8 (without decimal part): this means that if the number of cells is not a multiple of 8, i.e., the number of rows multiplied by the number of columns in the data region is not equal to $0 \mod 8$, in the final graphical representation there will be some corners not used during the encoding.

Moreover, referring again to Table 3.4, you should notice that also correction codewords are encoded in the symbol: for square codes of 48×48 and smaller, Reed-Solomon codewords are appended after the data; for other symbols they are interleaved and data are divided in blocks.

If then, during the high level encoding, data do not cover all the region

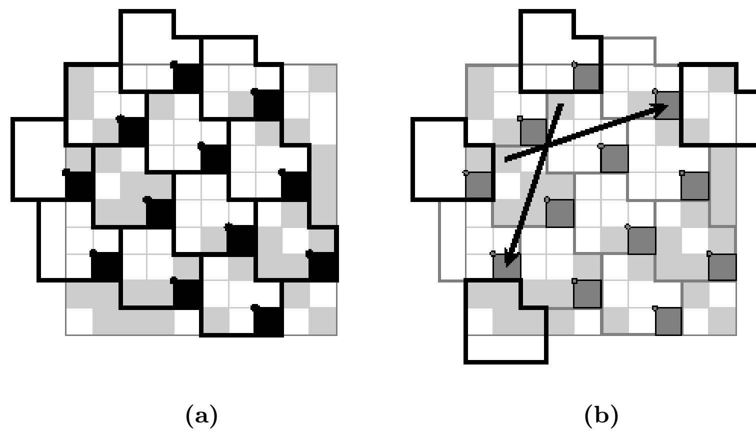


(a) Byte ordering



(b) Byte graphical representation

Figure 3.9: Codeword graphical representation



(a) (b)

Figure 3.10: Data Matrix nominal layout and codeword shifts

size, some padding codewords are added. You can see a clear example of what explained in the Data Matrix reported in Figure 3.11, which encodes the string "wikipedia" according to the layout we described: the string is visible in the top-left part of the symbol, and after its end, you can see padding symbols (P), error correction symbols (E) and, in this case, some unused bytes (X). Notice also that some letters are split and wrapped around in the Data Matrix.

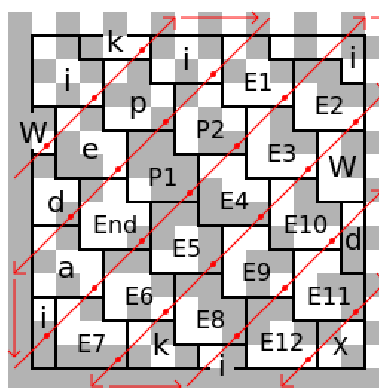


Figure 3.11: Layout of a Data Matrix encoding the string "wikipedia"

3.1.2 QR Code

QR Code (Quick Response Code), shown in Figure 3.2b, is a matrix code developed by Nippondenso ID Systems in 1994 and is in the public domain. Although initially used to track parts in vehicle manufacturing, QR codes are now (as of 2011) used over a much wider range of applications, including commercial tracking, entertainment and transport ticketing, product marketing and in-store product labeling. Many of these applications target mobile-phone users (via mobile tagging). Users may receive text, add a vCard contact to their device, open a Uniform Resource Identifier (URI), or compose an e-mail or text message after scanning the codes. QR codes storing addresses and Uniform Resource Locators (URLs) may appear in magazines, on signs, on buses, on business cards, or on almost any object about which users might need information. Users with a camera phone equipped with the correct reader application can scan the image of the QR code to display text, contact information, connect to a wireless network, or open a web page in the telephone's browser.

QR Code symbols are square in shape and can be identified by their finder pattern of nested alternating dark and light squares at three corners of the symbol. A quiet zone is required around the symbol. Maximum symbol size is 177 modules square, capable of encoding 7366 numeric characters, 4464 alpha numeric characters,2,953 bytes or 1817 Kanji and Kana characters (i.e. the peculiar Japanese symbols). QR Code is designed for rapid reading using CCD array cameras and image processing technology because of the layout of the finder pattern.

Like with Data Matrix codes, codewords are 8 bits long and use the Reed-Solomon error correction algorithm with four error correction levels. The higher the error correction level, the less storage capacity. The following table lists the approximate error correction capability at each of the four levels:

- Level L 7% of codewords can be restored.
- Level M 15% of codewords can be restored.
- Level Q 25% of codewords can be restored.
- Level H 30% of codewords can be restored.

Due to the design of Reed-Solomon codes and the use of 8-bit codewords, an individual code block cannot be more than 255 codewords in length. Since

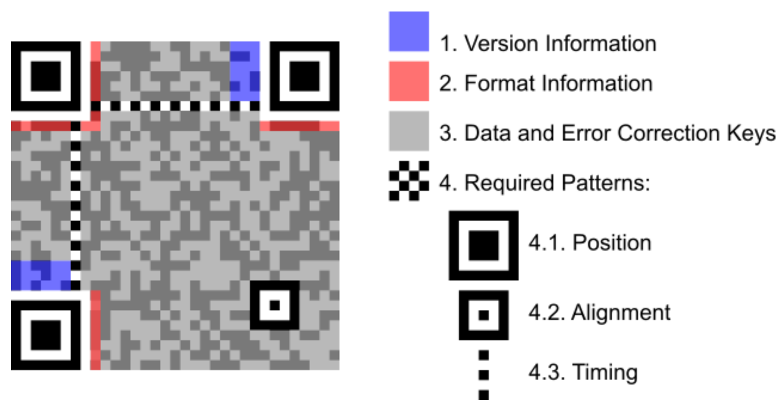


Figure 3.12: QR Code structure example

the larger QR symbols contain much more data than that, it is necessary to break the message up into multiple blocks. The largest possible block size is never used, though. The QR specification defines the block sizes so that no more than 15 errors can be corrected within each block. This limits the complexity of certain steps in the decoding algorithm. The code blocks are then interleaved together, making it less likely that localized damage to a QR symbol will overwhelm the capacity of any single block. We report in Figure 3.12 an example of a QR Code structure.

We do not go further in detail for this kind of barcodes, because in our application we used only Data Matrix codes, even if future developments could include the implementation of the support for other 2D codes.

3.1.3 Other barcodes

Briefly, we just cite the other most important two-dimensional barcodes, without going to much in detail

Aztec Code

Aztec Code (Figure 3.2c), invented and announced in 1995 by Welch Allyn, has been characterized as a "second generation" 2D matrix code. Aztec Code symbols are nominally square, with a square central bulls eye finder. They are scalable, and can encode up to 3067 text characters, 3832 digits, or 1914 bytes. Symbols range in size from 15 x 15 modules up to 151×151 modules. Error Correction is user selectable between amounts of Reed-Solomon error encoding from 5% to 95% of data region. The symbology is made of square modules with a square bulls-eye finder pattern in the center. No quiet zone around the symbol is required.

Code One

Code 1 (Figure 3.2d) was invented by Ted Williams in 1992 and is the earliest public domain matrix symbology. It uses a finder pattern of horizontal and vertical bars crossing the middle of the symbol. The symbol can encode ASCII data, error correction data, function characters, and binary encoded data. There are 8 sizes ranging from code 1A to code 1H. Code 1A can hold 13 alphanumeric characters or 22 digits while code 1H can hold 2218 alphanumeric characters or 3550 digits. The largest symbol version measures 134×148. The code itself can be made into many shapes such as an L, U, or T form.

Code 1 does not require a quiet zone, which saves space around the symbol for labelling and direct marking applications.

Code 1 is currently used in the health care industry for medicine labels and the recycling industry to encode container content for sorting. There does not appear to be any US Patent.

MaxiCode

Maxicode (Figure 3.2e) is a matrix code developed by United Parcel Service in 1992. However, rather than being made up of a series of square dots, MaxiCode is made up of an a 28.14mm wide by 26.91mm high array of 866 interlocking hexagons arranged in 33 rows around a central finder pattern. This allows the code to be at least 15 percent denser than a square dot code, but requires higher resolution printers like thermal transfer or laser to print the symbol.

A quiet zone of at least one cell borders the symbol. The symbol has a maximum data capacity of 93 alphanumeric characters and 138 numeric characters depending on the level of error correction. The symbol can still be read even when up to 25 percent of the symbol has been destroyed and can be read by CCD camera or scanner.

3.2 Fiducial markers

Beyond barcodes, there are other types of codes which are used for different and various applications: we refer to these barcodes as fiducial markers. There is plenty of applications which use these codes: one of the most common concerns augmented reality [1] [15], which consist in overlaying virtual imagery on the real world objects. Markers accomplishing this result are for example ARToolKit markers, where ARToolKit is a special computer vision library for augmented reality.

Also in robotics field these markers are widely used, for example for robot navigation and localization: in these specific cases, the markers are particular squared codes which contain some information that a robot can use to help himself to navigate inside an environment and to localize his position within such environment, in order to make the robot really autonomous and not depending from humans, i.e. what is called an intelligent robot. ARToolKitPlus and ARTag marker are mostly used in this task. In [13] these two last marker detection systems are qualitatively compared. Their advantages respect to 2D barcodes in these kind of applications is that these markers can be detected also within a large field of view in the image, so even if they are very distorted.

3.2.1 ARToolKit

ARToolKit [21] is a popular planar marker system for Augmented Reality and Human Computer Interaction (HCI) systems due to its available source code. The bi-tonal markers consist of a square black border and a pattern in the interior. Some examples of ARToolKit markers are shown in Figure 3.13.

The first stage of the recognition process is finding the markers' black borders, that is finding connected groups of pixels below a certain gray value threshold. Then the contour of each group is extracted, and finally those groups surrounded by four straight lines are marked as potential markers. The four corners of every potential marker are used to calculate a homography in order to remove the perspective distortion. Once the internal pattern of a marker is brought to a canonical front view one can sample a grid of $N \times N$ (usually 16×16 or 32×32) gray values inside. These gray values form a feature vector that is compared to a library of feature vectors of known markers by correlation. The output of this template matching is a confidence factor. If this confidence factor is greater than a threshold, a marker

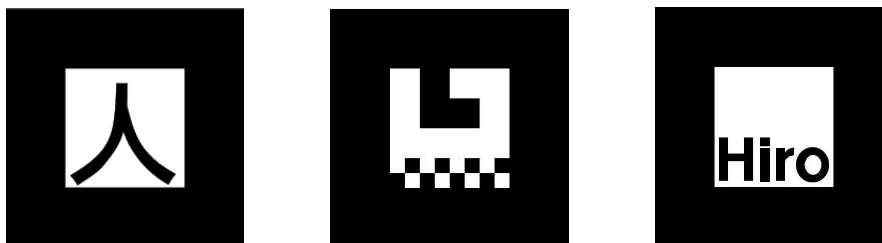


Figure 3.13: Some ARToolKit visual markers

has been found.

Although ARToolKit is useful for many applications, there are some drawbacks. First of all the detection process is threshold based. A single threshold can easily fail to detect markers under different illumination conditions, even within the same image. For example, it can happen that the white level at one marker edge is darker than the black level at the opposite edge. Furthermore the marker verification and identification mechanism using correlation causes high false positive and inter-marker confusion rates. With increasing library size the marker uniqueness is reduced, which again increases the inter-marker confusion rate. The processing time also depends on the library size, since every feature vector must be correlated with every prototype vector in the library. And for each marker there exist several prototype vectors to cover the four possible rotations as well as different lightning and distance conditions.

3.2.2 ARTag

ARTag [12] is another planar marker system for Augmented Reality and Human Computer Interaction systems. ARTag also uses markers with a square border (black or white), that you can see in Figure 3.14.

In contrast to ARToolKit, ARTag finds markers with an edge based approach, so one need not to deal with thresholds under different illumination conditions. Edge pixels found by an edge detector serve as basis for the marker detection process. They are linked into segments, which in turn are grouped into quadrangles. As with ARToolKit the corners of a quadrangle are used to calculate a homography so that the marker's interior can be sampled. In contrast to the patterns used in ARToolKit, the interior region of an ARTag marker is filled with a 6x6 grid of black or white cells, representing 36 binary '0' or '1' symbols. This 36-bit word is then processed in the digital domain. For each of the four possible marker orientations one 36-bit

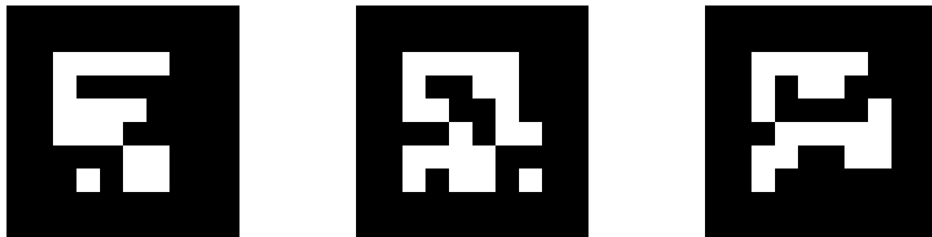


Figure 3.14: Some ARTag visual markers

sequence is obtained from the 36-bit word, with only one sequence ending up being used in the decoding process. Every 36-bit sequence encodes a 10-bit marker id, leaving 26 redundant bits for error detection, correction and uniqueness over the four possible rotations of a marker.

The edge based approach of ARTag makes the system more robust to changing illumination conditions than ARToolKit. ARTag can even cope with occlusions, broken sides and missing corners to a certain extent. This is possible because of heuristics of line segments that almost meet, so that missing segments of a marker can be estimated. Furthermore ARTag's id based markers do not require image matching with a library and therefore allow for a much faster identification than the template based markers used in ARToolKit.

3.2.3 ARToolKitPlus

ARToolKitPlus is an evolution of ARToolKit: it uses the same recognition algorithm to detect a marker borders, but it introduces a threshold auto-calibration system that makes it more flexible to the light condition variations. It is possible to use both markers with generic information content (i.e., an image to compare, like in the ARToolKit case) and with a digital ID (i.e., a 2D barcode, like in the ARTag case).

The default markers with digital content are divided in two categories: *simple ID*, with 512 available markers, and *BCH ID*, with 4096 available markers. Examples of both categories are shown in Figure 3.15.

The main improvements introduced by ARToolKitPlus respect to ARToolKit are:

- An automatic threshold mechanism for the creation of the black-and-white image to analyse.

- Compensation of the vignetting phenomenon (dark zones in the image corners), which improves the marker recognition chances, even if they are close to the image borders.

- Simplification of the camera calibration process, which can be done with any standard system (e.g., Camera Calibration Toolbox for Matlab).

- Employment of an algorithm for the estimate stabilization of the roto-translation matrix (*Robust Planar Pose Tracking, RPP*, which brings from the camera coordinates system to the marker coordinate system.

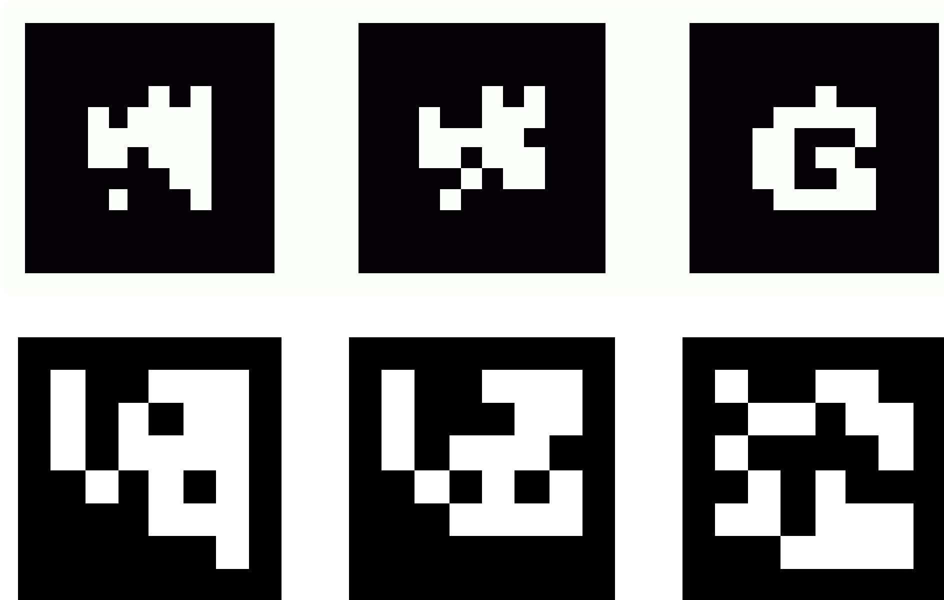


Figure 3.15: Some ARToolKitPlus with simple ID (upper) and BCH ID (lower) visual markers

ARToolKitPlus is also portable on mobile devices, thanks to the availability of fixed point operations, even if RPP algorithm cannot be implemented on such devices, since it requires a deeper calculus precision.

Chapter 4

Data Matrix visual detection and pose estimation

> *"Do you know what first interested me about Pythagoras? He invented the word 'friendship'. When someone asked him what a friend was, he replied 'Someone who is another me, like the numbers 220 and 284.' Two numbers are 'friends' if each is the sum of everything that measures the other. The most famous of these so-called 'amicable numbers' are 220 and 284. They're a fine couple. You should try checking them sometimes."*
>
> Parrot's theorem

In this chapter we describe the process of computing the marker pose with respect to the camera coordinates system, i.e., we want to know how a marker is orientated and translated in reference to the camera observing it. Then we talk about the vision methods we used to detect a visual marker from an image captured by the camera and we show how the these methods affect the pose estimation process by comparing them numerically and analysing their pros and cons.

4.1 Camera calibration

Camera calibration [44] is a necessary step in 3D computer vision in order to extract intrinsic, extrinsic and distortion parameters from 2D image observations.

4.1.1 Pinhole camera model

The pinhole camera model describes the mathematical relationship between the coordinates of a 3D point and its projection onto the image plane of an

ideal pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light. This model represents a simplification of the real model, since no radial lens distortion is modelled and this relationship is called *perspective projection*.

We denote the center of the perspective projection (the point in which all the rays intersect) as the optical center or camera center and the line perpendicular to the image plane passing through the optical center as the optical axis (see Figure 4.1). Additionally, the intersection point of the image plane with the optical axis is called the principal point.

The projection of a 3D world point $\mathbf{M} = [X, Y, Z]^T$ onto the image plane at pixel position $\mathbf{m} = [u, v]^T$ can be written as:

$$u = -f\frac{X}{Z} \tag{4.1}$$

$$v = -f\frac{Y}{Z} \tag{4.2}$$

where f is the focal length of the camera.

The previous relation can be reformulated in matrix notation, using homogeneous coordinates ($\tilde{\mathbf{M}} = [X, Y, Z, 1]^T$ and $\tilde{\mathbf{m}} = [u, v, 1]^T$) and a scaling factor λ, as follows:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4.3}$$

Camera intrinsic parameters

Unfortunately, the pinhole camera model is ideal, because it assumes perfect lenses and the principal point at the center of the image. In physical cameras instead, the imaging projection are usually distorted, there is finite resolution defined by the sensing device of a digital camera and there is an offset between the image center and the optical center.

In order to model these characteristics, new parameters are introduced, called *camera intrinsic parameters*. They characterize internal properties of the camera, like optical, geometric, and digital characteristics:

- the focal length f
- the transformation between image plane coordinates and pixel coordinates

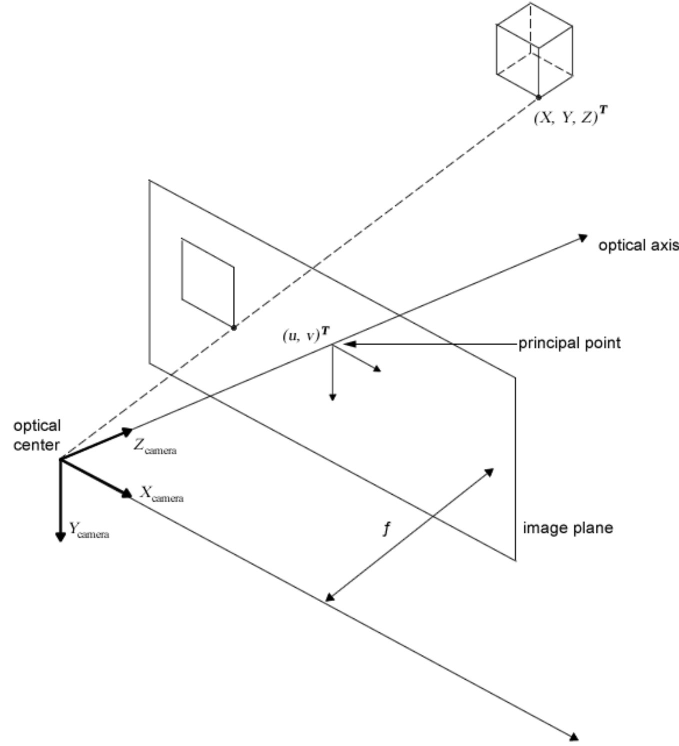


Figure 4.1: Pinhole camera model: perspective projection between a 3D point $[X, Y, Z]^T$ and its corresponding 2D projection $[u, v]^T$

- the geometric distortion introduced by the optics

First of all, the image coordinates are transformed into pixel coordinates:

$$\begin{bmatrix} u_p \\ v_p \end{bmatrix} = \begin{bmatrix} p_x & 0 \\ 0 & p_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \tag{4.4}$$

where p_x and p_y are the scale factors relating pixels to distance. Then the image origin is translated, since most of the current imaging systems define the origin of the pixel coordinate system at the top-left pixel of the image, and not at its center, corresponding to the principal point:

$$u^{'} = u_p + c_x \tag{4.5}$$

$$v^{'} = v_p + c_y \tag{4.6}$$

So in homogeneous coordinates we obtain:

$$\tilde{\mathbf{m}}^{'} = \begin{bmatrix} u^{'} \\ v^{'} \\ 1 \end{bmatrix} = \begin{bmatrix} p_x & 0 & c_y \\ 0 & p_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{4.7}$$

which leads to the complete transformation from 3D to 2D pixel coordinates

$$\lambda \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} p_x & 0 & c_y \\ 0 & p_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \\ = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4.8}$$

where $f_x = f \cdot p_x$ and $f_y = f \cdot p_v$ represent focal length in terms of pixels.

The matrix

$$\mathbf{A} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.9}$$

is called matrix of intrinsic parameters, and γ represents the skew coefficient between the x and the y axis, and is often 0.

The matrix

$$\mathbf{\Pi_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.10}$$

is called initial projection matrix, when the camera and the world coordinate system correspond.

Camera extrinsic parameters

In general, the camera coordinate system and the world coordinate system are different, since the camera might move inside the environment. The camera extrinsic parameters denote the coordinate system transformations from 3D world coordinates to 3D camera coordinates.

The previously defined projection matrix is thus composed by two elements (Figure 4.2), which relates the world coordinate system to the camera coordinate system:

- **R**, a 3×3 rotation matrix that brings the corresponding axes of the two frames into alignment (i.e., onto each other)
- **t**, a 3×1 translation vector between the relative positions of the origins of the two reference frames

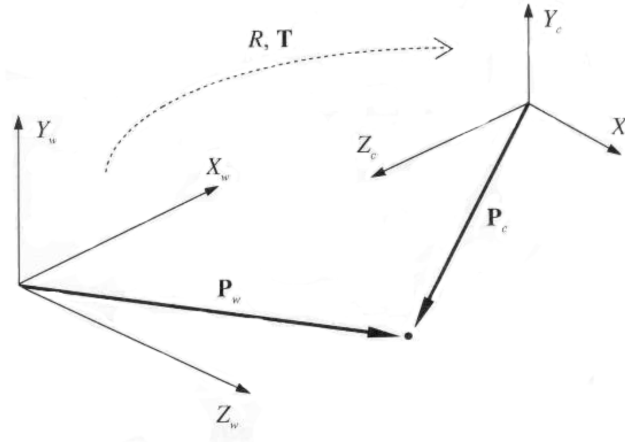


Figure 4.2: Camera extrinsic parameters

so that we can define the projection matrix

$$\mathbf{\Pi} = \begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix} \tag{4.11}$$

In this way, we obtained the complete pinhole camera model, defined by the following relation:

$$\lambda \tilde{\mathbf{m}}^{'} = \mathbf{A}\mathbf{\Pi}\mathbf{M} \tag{4.12}$$

where $\tilde{\mathbf{M}} = [X, Y, Z, 1]^T$ are the coordinates of a 3D object in the world frame.

Distortion model

Real camera lenses typically suffer from non-linear lens distortion, typically radial. In practice, radial lens distortion causes straight lines to be mapped as curved lines. To model this behaviour, we re-write the Equation 4.12 in an equivalent way (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{R} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \mathbf{t} \tag{4.13}$$

Then we set

$$x^{'} = \frac{x}{z}$$

$$y^{'} = \frac{y}{z}$$

$$x^{''} = x^{'} \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2 p_1 x^{'} y^{'} + p_2 (r^2 + 2 x^{'2})$$

$$y^{''} = y^{'} \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2 p_2 x^{'} y^{'} + p_1 (r^2 + 2 y^{'2})$$

$$u = f_x \cdot x^{''} + c_x$$

$$v = f_y \cdot y^{''} + c_y$$

where

$$r^2 = x^{'2} + y^{'2}$$

$k_1, k_2, k_3, k_4, k_5, k_6$ are radial distortion coefficients, and p_1, p_2 are tangential distortion coefficients.

4.1.2 Calibration experiments

The pinhole camera employed for our experiments is a GigE Prosilica, model GC750C, with 4 mm focal length and a resolution of 752 × 480 [18]. We decided to calibrate our camera using the MATLAB toolbox[1].

We took 15 images of a chessboard in different poses to calibrate the camera (see Figure 4.3). The toolbox allows to manually select the chessboard grid corners for every image and then automatically computes camera calibration.

The result of the calibration gave us the camera intrinsic parameters, ready to use in our vision application. They are reported below.

Figure 4.3: Chessboard images for camera calibration

[1] http://www.vision.caltech.edu/bouguetj/calib_doc/index.html

Calibration results after optimization (with uncertainties):
Focal Length:
fc = [687.18874 684.95763] ± [1.50973 1.64780]

Principal point:
cc = [413.62924 221.45226] ± [3.19102 2.69161]

Skew:
alpha_c = [0.00000] ± [0.00000] ⇒ angle of pixel axes = 90.00000 ± 0.00000 degrees

Distortion:
kc = [-0.03600 0.05991 -0.00149 0.00321 0.00000] ± [0.00841 0.04336 0.00126 0.00154 0.00000]

Pixel error:
err = [0.08825 0.10541]

Note: The numerical errors are approximately three times the standard deviations (for reference).

The MATLAB toolbox also computes the camera extrinsic parameters for each sample image, showing the relative pose of the camera in world coordinates system, or vice-versa. In Figure 4.4 and Figure 4.5 you can

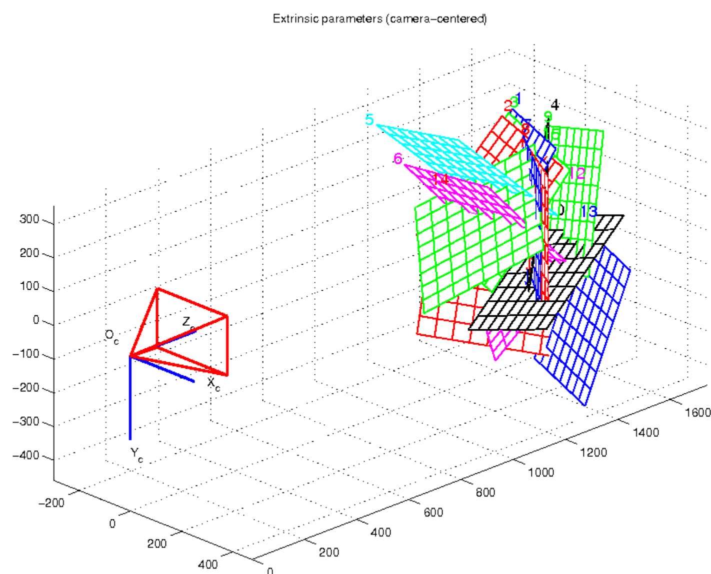


Figure 4.4: MATLAB representation of camera extrinsic parameters (Camera frame)

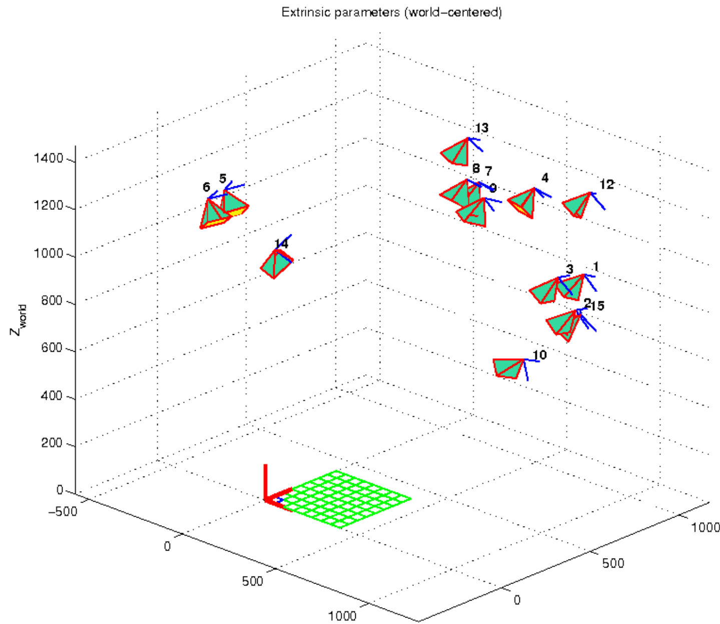


Figure 4.5: MATLAB representation of camera extrinsic parameters

respectively see position and orientation of each chessboard in the camera frame, and position and orientation of the camera in the world frame, which is aligned with the chessboard.

4.2 Data Matrix encoding

We decided to use Data Matrix barcodes as visual markers instead of AR-ToolkitPlus, usually used for localization problems, due to the availability of a C/C++ open source library for marker detection, called libdmtx [19], and to the encoding power that characterizes these barcodes. Moreover, there is already a ROS package which implements Data Matrix recognition and camera pose computation [17], so it can be useful for future development works, as porting MoonSLAM to ROS framework.

Before describing the actual pose estimation process, we need to make some preliminary remarks about the information we decided to encode in each marker. As from Chapter 3, Data Matrix barcodes are able to store alphanumeric symbols, giving the possibility to save inside them a high number of information: this was one of the main reasons we selected them as visual markers for localization and mapping purposes.

We chose a unique pattern for information to be encoded, so to make

all the markers standardized for our application. Our Data Matrix can only encode a 5-symbols string, where each symbol is represented by an alphanumeric character: numbers (from 0 to 9), lower case letters (from “a” to “z”) and upper case letters (from “A” to “Z”), for a total of 62 symbols. We decided to use the first 3 values to be part of the marker ID, allowing us to distinguish up to $62^3 = 238328$ different Data Matrices: enough to cover every kind of indoor environment. Surely only 2 ID symbols ($62^2 = 3844$ total markers) would also be enough, but the barcode internal density with 4 encoded symbols does not change, so even using only 2 characters for the ID would not reduce the graphical complexity, without consequently helping GFTT algorithm to find relevant features more accurately.

Regarding the remaining 2 characters, they represent the marker side dimensions in millimetres (we always suppose to handle with squared Data Matrix): we designed a simple function to convert a length size to the 2 alphanumeric symbols. First of all, since we can use 62 characters per symbol, we can encode up to $62^2 = 3844$ different values, that is from 0 to 3844 mm (0 to 3.84 m). A marker cannot have a size of 0 m, usually the minimal dimension is 10×10 cm, and also Data matrices more than 1 m large are quite uncommon.

Given the dimension in millimetres, we computed the 2-symbols string in the following way:

$$c1 = \left\lfloor \frac{dimension}{62} \right\rfloor$$

$$c2 = dimension \mod 62$$

and then we mapped the resulting integer values ($value_1$ and $value_2$, in the range [0,61]) to the 62 alphanumeric symbols:

- numbers from 0 to 9 are mapped to themselves
- numbers from 10 to 35 are mapped to lower case letters “a-z”
- numbers from 36 to 61 are mapped to upper case letters “A-Z”

We report a simple example of information encoding: if we want to print a Data Matrix with ID *9wJ* and a dimension of 183 mm (18.3 cm), the two remaining symbols are

$$c1 = \left\lfloor \frac{183}{62} \right\rfloor = 2 \rightarrow value_1 = 2$$

$$c2 = 183 \mod 62 = 59 \rightarrow value_2 = X$$

giving the final encoded string equal to *9wJ2X*. The inverse function is also straightforward: taken the two encoded symbols, first they are mapped following the previous schema in the inverse direction, obtaining the values $c1$ and $c2$, then the marker size in millimetres is computed as follows:

$$dimension = c1 \times 62 + c2$$

4.3 Perspective N-Point

The Perspective N-Point problem (PnP) [32] is a notable problem in computer vision. Given a set of N correspondences between 3D reference points and their projections onto an image, it consists in estimating the position and orientation of the calibrated camera with respect to the known reference points: in other words it solves the problem of finding the camera extrinsic parameters, returning the rotation matrix and the translation vector of the object with respect to the camera.

To obtain the marker pose in the camera frame, i.e. in the camera coordinates system, we needed the following data:

- a camera image containing a Data Matrix, of course;
- camera intrinsic parameters;
- camera distortion coefficients;
- at least 4 two-dimensional projections of points belonging to the marker on the image plane;
- at least 4 (and in any case the same number of the image points) three-dimensional points belonging to the 3D object, each of them corresponding to a 2D image point.

The camera intrinsic parameters were already obtained as explained in Section 4.1, as well as the distortion coefficients.

As far as it concerns the choice of the plane points, we decided to use the minimum needed number to compute the PnP problem. A natural choice was obviously to pick the four Data Matrix corners. Since the precision of the PnP solution depends on the 2D points, we show in the following sections some vision methods to try to get the most accurate results in finding the marker corners.

Regarding the object points in the 3D space, we chose the Data Matrix corners, but considering them in 3D. We used an intuitive representation for the three-dimensional coordinates systems: a Data Matrix, being a planar

object, lays on a XY plane, with the X-axis corresponding to the bottom black edge of the marker finder pattern, the Y-axis corresponds to the left black edge of the finder pattern and the system origin coincides with the bottom left corner; Z-axis is consequently directed toward up, starting from the origin point. In Figure 4.6 a graphical explanation of what we just described is given.

The four Data Matrix corners can be expressed in 3D coordinates $[X, Y, Z]^T$ as it follows:

$$\begin{aligned}
\mathbf{p}_0 &= [0, 0, 0]^T \\
\mathbf{p}_1 &= [DIM, 0, 0]^T \\
\mathbf{p}_2 &= [DIM, DIM, 0]^T \\
\mathbf{p}_3 &= [0, DIM, 0]^T
\end{aligned} \tag{4.14}$$

where DIM is the actual marker edge length (usually expressed in metres).

Since we were able to encode the marker dimension inside each Data Matrix, as explained in the previous section, our library does not need to know in advance a fixed size for each 3D object: in fact, every time a Data Matrix is detected by the camera, the information about the edge length encoded inside the marker itself can be easily retrieved and used to set the 3D object

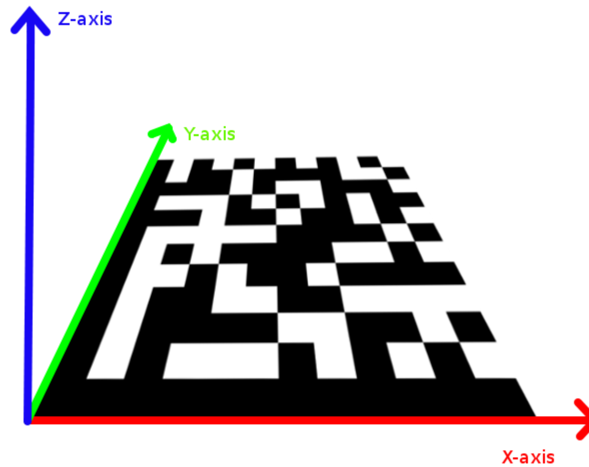


Figure 4.6: Marker coordinates system

points coordinates within the marker frame. This expedient is very useful for two main reasons: first, the 3D model points are set in automatic way by the internal procedures, without the need for manual settings, and secondly, it allows to use Data Matrices printed with different sizes, making them more adaptable to every type of environment.

OpenCV provides an efficient function to compute the pose estimation given the listed variables, called *solvePnP*. Of course it is necessary to specify the image points and the object points in the same order, so that each 3D point is mapped to its correspondent one on the image plane. Given the elements listed above, this function returns two variables:

$$\mathbf{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{4.15}$$

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{4.16}$$

The variable t is called translation vector and it specifies how the coordinates system of the object represented in the image plane (in our case a Data Matrix) is shifted with respect to the camera coordinates system along x, y and z-axis.

The variable v is a vector of a rotation axis represented as Rodrigues angles and it specifies how the coordinates system of the object is rotated by an angle $\theta = \sqrt{{v_x}^2 + {v_y}^2 + {v_z}^2}$ about that axis with respect to the camera frame. The rotation matrix $\in R^3$ corresponding to Equation 4.16 can be computed through the Rodrigues formula [2], obtaining:

$$\mathbf{R}_C^M = \mathbf{I} + sin(\theta)\mathbf{r} + (1 - cos(\theta))\mathbf{r}\mathbf{r}^T \tag{4.17}$$

where $\mathbf{I}$ is a 3×3 identity matrix and $\mathbf{r} = \begin{bmatrix} 0 & -v'_z & v'_y \\ v'_z & 0 & -v'_x \\ -v'_y & v'_x & 0 \end{bmatrix}$, where $v'_x = \frac{v_x}{\theta}$, $v'_y = \frac{v_y}{\theta}$ and $v'_z = \frac{v_z}{\theta}$.

A rotation matrix is not a minimal representation of a generic rotation, because it uses 9 variables. For this reason we chose to use another way to specify the marker attitude in the camera frame [9]: for graphical operations, such as graphics MATLAB plots, some of which also reported later in this thesis, we adopted the Euler angles, because they are simple and intuitive to understand even at a first glance. They consist in 3 angles called *roll, pitch*

and *yaw* (ϕ, θ, ψ) which correspond respectively to a rotation about the x, y or z-axis by the given angle value. Instead, in order to calculate the marker pose in the world coordinates system and to compute the image points measurements in the Kalman filter during the SLAM algorithm (described in the next chapter), we used a non-minimal representation given by quaternions: a quaternion, $\mathbf{q} \in H$, may be represented as a vector $\mathbf{q} = [a, b, c, d]^T$ and if it is a unit quaternion (i.e., $\|\mathbf{q}\| = 1$) it can represent the attitude of a rigid body.

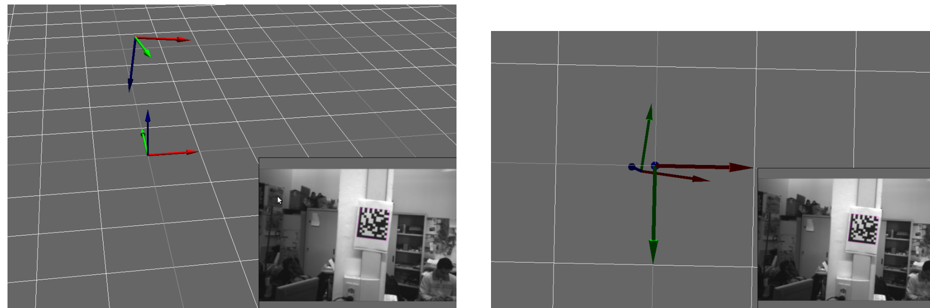
In Figure 4.7a, we show what are the results of solving the PnP problem. On the right you can see a camera image where a Data Matrix is detected, while on the left the camera and the marker coordinates systems are shown. Since the function computes the marker position and attitude with reference to the camera, the latter is considered as the world center and it is represented by the three axis on the bottom: for a generic camera they are usually arranged with the z-axis pointing out of the camera, the x-axis pointing to the right of the camera and the y-axis pointing downward. Marker axis, instead, are depicted as explained earlier, with x-axis coloured in red, y-axis in green and z-axis in blue.

In Figure 4.7b the same scene is reported, but from the camera point of view: the marker is exactly rotated like in the reality, as you can see in the near image.

In this example we got that the Data Matrix was translated of -0.0986 m along the camera x-axis, 0.0407 m along the y-axis and 1.1664 m along the z-axis (which is the distance from the camera), while it was rotated of 176.289 degrees about the x-axis (roll angle), 8.512 degrees about y-axis (pitch angle) and 6.897 degrees about z-axis (yaw angle). As you can see, these values are quite intuitive and can be easily used to analyse the results obtained during the pose estimation process.

4.4 Data Matrix detection

After the necessary camera calibration step, we needed to introduce a fast and precise way to detect Data Matrix markers from raw images captured from the camera: this was needed in order to successively compute the marker pose in the camera coordinate frame and feed such information to an Extended Kalman Filter, needed to the SLAM algorithm to compute and update the camera pose (and consequently the robot pose, since the camera has hypothetically a rigid pose with reference to the robot) in the yet unknown world frame.



(a) Marker pose estimation in camera coordinates system

(b) Marker pose from camera point of view

Figure 4.7: Marker pose estimation

This phase is very delicate, because from the precision and speed in marker recognition and in camera extrinsic parameters estimation partially depends the accuracy with which the EKF-SLAM algorithm can operate to build the environment map and localize the camera. For this reason, we dedicated lot of effort on this part, trying to get the best results in marker recognition using different computer vision techniques[41] and performing various recognition simulations, analysing and trying to minimize the errors and the noise during the detection.

To work with camera images we made use of OpenCV[2], a complete C/C++ library [3] for Linux, Windows and Mac OS machines, which offers an extensive set of computer vision functionalities, particularly indicated for real-time and computationally efficient purposes.

Some of the most useful functions available with OpenCV are:

- geometric image transformations (rotations, resizing);
- feature detection (Canny, Harris, Hough, GFTT, FAST);
- camera calibration and 3D reconstruction;
- machine learning (k-nearest neighbours, support vector machines, linear regression).

4.4.1 *libdmtx* library

The first step in marker recognition is to detect in a fast and efficient way whether or not in a camera image is present a Data Matrix, and in that case what is encoded inside it. *libdmtx* library was created for this specific

[2] http://opencv.willowgarage.com/wiki/

purpose: it allows to find and decode any number of Data Matrix barcodes inside an image in a relatively short time, depending on the size of the camera image, on the number of barcodes present and on the Data Matrix dimension relative to the image size (normal time can go from 1 ms to 400-500 ms, but, in general, when the marker is clearly visible within a camera frame and not too far from the camera, the average time can be 50-60 ms at most) it is possible to specify a maximum searching time-out or a maximum number of barcodes to detect. The library also allows to encode information and to create a graphical marker, in the limits imposed by the Data Matrix standard (see Chapter 3 for details).

It must be noticed that ambient light intensity is a fundamental factor which affects the library performance: in fact, in order to capture an image with good contrast value, the light intensity should be above certain level. Absence of light could decrease the probability of marker detection. Also sudden changes in light intensity could lower this probability. We report in Table 4.1 the percentage of detected markers at different distances from the camera and imposing different timeouts over a sample of 1000 static frames. As you see, the library needs a minimal time search (as the average times show), so setting a timeout higher than that value results in high detection rates. However, detection rate depends on where the marker lays in the image: markers very close to image borders can take more time than shown to be recognized. Also Data Matrix very slanted need more time to be detected. Moreover, camera motion can introduce image blurring, potentially lowering shown detection percentages.

As we said in the previous chapter, Data Matrix barcodes were created mostly to carry information about products during industrial processes or to identify postal parcels: *libdmtx* library was developed considering this main function for Data Matrix codes, so its accuracy is more oriented to the correct information decoding, without optimizing the marker localization within the image (in terms of pixels).

However, whenever the library finds a marker within an image, in addition to decoding the barcode content, it also stores some basic information regarding Data Matrix pixel position in the image, which can be considered to roughly localize the marker (i.e., to find the four barcode corners in pixels) and for further vision optimization. Between these data, the ones most useful to initially calculate the four pixel corner points of the Data Matrix are:

- the top-left corner on the L finder pattern;
- the bottom-right corner on the L finder pattern;

		Timeout			
		15 ms	**30 ms**	**60 ms**	**120 ms**
Approximate distance from camera	**4.5 m**	0 % NA	68.9 % 29.41 ms	99.9 % 28.83 ms	99.9 % 27.92 ms
	2.3 m	19.8 % 19.12 ms	99.5 % 22.83 ms	99.9 % 23.44 ms	100 % 22.61 ms
	1 m	85.8 % 16.07 ms	99.8 % 16.08 ms	99.7 % 16.76 ms	100 % 16.42 ms

Table 4.1: Percentage of detected Data Matrix markers and average time search over 1000 static frames

- the angle slope of every side of the marker, measured with respect to the horizontal axis (in degrees, from 0 to 180): we referred to them as bottom, right, top and left angle.

You can see the listed quantities in Figure 4.8.

As explained few lines before, *libdmtx* library is mostly used for information decoding, minimizing the misclassification error during this process, and not for computer vision refinements of the detected marker, so the information data cited before are quite rough, and indeed their values are treated with integer data structures at programming level, obviously loosing the fundamental accuracy needed for a good marker pose estimation in camera coordinates (i.e. computation of the camera extrinsic parameters), described later in this chapter. What we really need, to effectively use Data Matrix markers (or other visual markers) in a EKF-SLAM algorithm, are at least 4 points from the 2D camera image. The simplest choice of these points, for a Data Matrix, is to consider the 4 marker corner points (of course the top-right corner is a blank point in ECC 200 Data Matrix standard, so it is just a geometric point not visible in the image). Since we do not directly have the bottom-left and the top-right corners from the library decoding information, we have to calculate them using the available information.

Here is a brief summary to obtain the remaining corners (for example the bottom-left one):

- Starting from the bottom-right corner, lengthen the line passing through this point with a slope given by the bottom angle.

- Find the point on the image border that belongs to such line (P2), with some simple trigonometry operations.

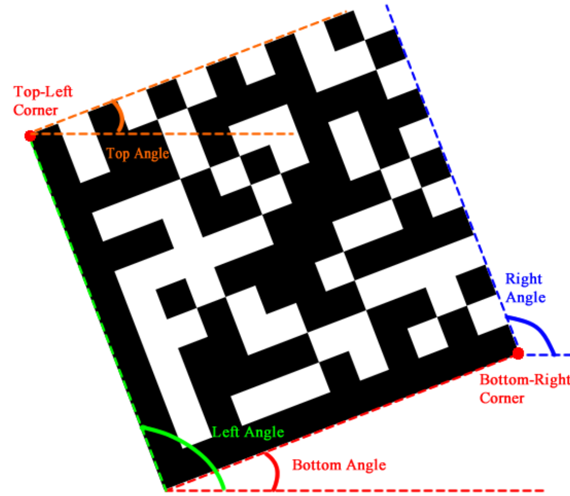


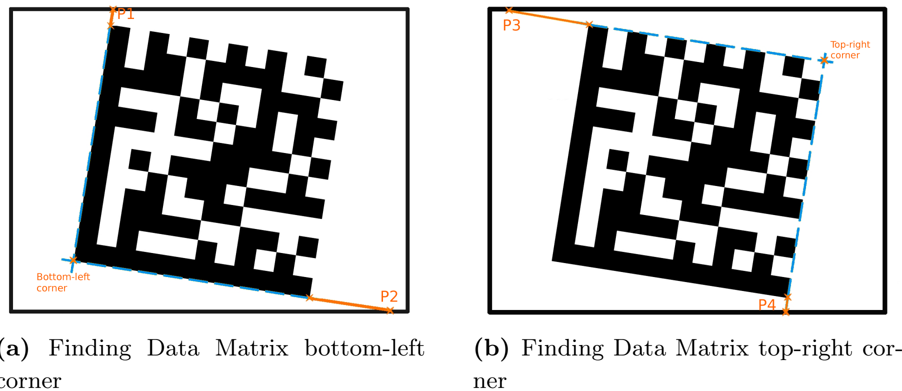
Figure 4.8: Data Matrix parameters returned by *libdmtx*

- Similarly, starting from the top-left corner, lengthen the line passing through this point with a slope given by the left angle and find the point on the image border that belongs to such line (P1).
- Given the four points, compute intersection between the lines passing through them (by twos): the intersection point is the bottom-left corner of the Data matrix.

In the same way it is possible to find the top-right corner, considering the right and top slope angles. In Figure 4.9a and Figure 4.9b you can see the graphical explanation of this process.

As we said, unfortunately the values of the corners and of the angle slopes are reported as integers, so it is enough a simple movement, a light variation or some other noises to change with discrete values these variables (e.g., the library could skip from 68 to 70 degrees of marker slope, because it does not consider decimal values, widening also the error in finding the two corners not given by the library): this fact has a serious drawback in the camera extrinsic parameters computation phase, since it could heavily affect the correct marker pose estimation in the camera frame (and consequently in the world frame), making less accurate the localization of the camera in the world, after the EKF-SLAM phase.

Detecting a Data Matrix with *libdmtx* library, drawing its contour using the four corners given by the library itself (two available and two obtained



(a) Finding Data Matrix bottom-left corner

(b) Finding Data Matrix top-right corner

Figure 4.9: Data Matrix missing corners

with the procedure just explained), can lead to a very inaccurate estimate of the marker position within the camera image, as it is shown in Figure 4.10. It is clearly visible that the four corners are way too inaccurate than they should to be, even if the marker is correctly and quickly decoded. For this reason, we did not considered a good solution to use data which may be consistently wrong (in the average case, because not always the results are as bad as in the figure, but they can be sometimes even worse) for our further works within this thesis.

For this reason we decided to improve the corner detection process, making use of some computer vision techniques: we took into consideration three methods, the Good Features To Track (GFTT) and the FAST algorithms, to find the best visible features inside a Data Matrix, and the Canny algorithm together with Hough algorithm, to detect possible straight lines as marker edges. However, in all the cases, we used the marker corner points returned by *libdmtx* as starting points, in order to make then some further refinements. Notice that from now on, when we talk about points "returned by *libdmtx*" we refer to all the four corners, both the ones directly available and the ones retrieved with the operations reported before.

In the next sub-sections we describe the methods cited, comparing them graphically and selecting the one which gave us better results. We used the same camera frame (already shown in the previous figure) for every example, in order to better analyse differences between the different techniques.

Figure 4.10: Data Matrix detection with *libdmtx* corners

4.4.2 GFTT - Good Features To Track algorithm

A Data Matrix, due to its design, has a dense zone of black and white squares; we can use them to find a more accurate estimate of the marker corners, especially the ones that the library does not provide, i.e., the bottom-left and the top-right, which are the ones more subject to noises.

Given a camera image, the Good Features To Track (GFTT) algorithm [35] finds a set of relevant points in the frame, i.e., pixels with a high contrast with respect to other surrounding pixels. The OpenCV function which computes GFTT takes into consideration several parameters, like the number of salient points to detect, the minimum possible Euclidean distance between the returned corners, a parameter characterizing the minimal accepted quality of image corners and a mask which defines a subset of the camera image in which look for relevant points. The general idea we had was to detect all the possible corners on the external top and right borders, more specifically the ones laying on the *clock track* of the marker finder pattern, and then, with these points, through linear regression, fit two lines which should represent exactly the top and the right edges of the Data Matrix. The intersection of these two lines would be the correct top-right marker corner. We used GFTT algorithm also to detect correctly the bottom-left Data Matrix corner, since it stands out pretty clearly among the surrounding lighter pixels.

To gain even more accuracy during the detection of the top-right corner, we decided to process these operations (explained further in details) in two sweeps, the first one useful to get a more accurate estimate of the marker top-right and bottom-left corners than the ones obtained through the *libdmtx* library, the second one useful to further refine these estimates, in case of poor

results after the first sweep, due to very wrong slope values calculated by the library.

Unfortunately, there is a major drawback in the mentioned process: GFTT is very inefficient in terms of time proportionally to the image size, because it analyses the whole image to find the salient points, even if we use a mask around the marker, since the algorithm searches anyway inside the whole frame and then shows only the corners within the specified mask. The average time of a GFTT function call is around 18-23 milliseconds, and considering that we use it five times (three in the first sweep to find the bottom-left corner and the two fragmented edges, and two in the second sweep to refine the edges), the total time costs for corners refinement can be excessive, in particular if the application should run in real time.

For this reason, we process with GFTT only a region of interest (ROI) of the camera image, containing only the zone around the actual Data Matrix: we established a fixed offset from the marker corners in order to "cut" the frame and analyse only the sub-image, just like if it is a different new image. In Figure 4.11 you can see an image with a marker and the ROI inside which the GFTT algorithm operates. This expedient lowers the execution time to about 5-8 milliseconds, depending on how big is the ROI with respect to the actual frame size: the bigger it is, the longer it takes to GFTT to execute, with the limit case in which the ROI is as big as the whole image. After having specified a ROI, we can use the GFTT algorithm to find more precisely the bottom-left and top-right marker corners.

Figure 4.11: Region-Of-Interest inside which compute GFTT algorithm

Figure 4.12: GFTT mask for bottom-left point detection

Bottom-left point

Regarding the bottom-left point, we decided to keep its estimate after just the first sweep, because in almost all the situations a second sweep would not add any better refinement than it already got after the first sweep.

In Figure 4.12 it is graphically explained how GFTT finds the bottom-left corner: first of all, we set a circular mask with center in the unstable bottom-left point returned by the library and diameter proportional to the Data Matrix length in the image (the smaller the marker appears inside the frame, the smaller the circle is).

Secondly, we used GFTT to find the corner (coloured in red in the figure), imposing to return only one point, that in most cases is the corner we were looking for, since it is the only one present, even if it may rarely happen that distortion or light variations affect the results and get GFTT to find a close, but wrong, point. After the GFTT operation we computed a corner sub-pixel refinement [40], to minimize possible errors and get a corner with better accuracy.

In order to find the top-right corner we had to use alternative techniques, because such corner in Data Matrix is actually a blank point, so it is not possible to directly use GFTT to find it, as in the bottom-left point case. We have to find the two lines corresponding to the top and right marker edges, starting from the points present on such dashed borders, and then we have to compute the intersection between these two lines. GFTT could help to select the points we are talking about.

Masks selection

Still considering only the ROI within the image, in order to find only the relevant points laying on the Data Matrix top and right edges we use two masks that extend along the mentioned marker borders. From the four raw points obtained from *library*, we created two linear masks, for the top edge (starting from the top-left to the top-right marker corner) and for the right edge (starting from the bottom-right to the top-right corner), as shown in Figure 4.13.

To handle cases in which the camera is very slanted with respect to the visual marker, we set the top mask thickness to be proportional to the left edge length, and the right mask thickness to be proportional to the bottom edge length. For instance, if the Data Matrix is very stretched along its height (so that it appears as a rectangle), the top mask would be shorter but also thicker, to detect the stretched squares.

GFTT with white points deletion

Once we obtained our masks, we need to execute the GFTT algorithm. For the first sweep, we set a very low level of quality for the features to be detected and a relatively high number of maximum corners to detect (about 30), so to be sure to find at least all the points laying on the external borders. Since there is the possibility that GFTT finds also a large number of useless dots, even laying on white zones, we processed the set of detected points to a further refinement: we deleted from the set all the points above a

Figure 4.13: Linear masks for top and right edges in the first sweep

specified color threshold (in grayscale terms), i.e., the points with a non-dark colouring, to be sure to get rid of corners on white areas. The results before and after this operation are shown in Figure 4.14.

Fit lines

Now that we obtained two sets of corners which represent in a quite accurate way all the corners of the squares composing the top and right edges, we used these points to fit two lines passing through them, using the OpenCV function *fitLine*.

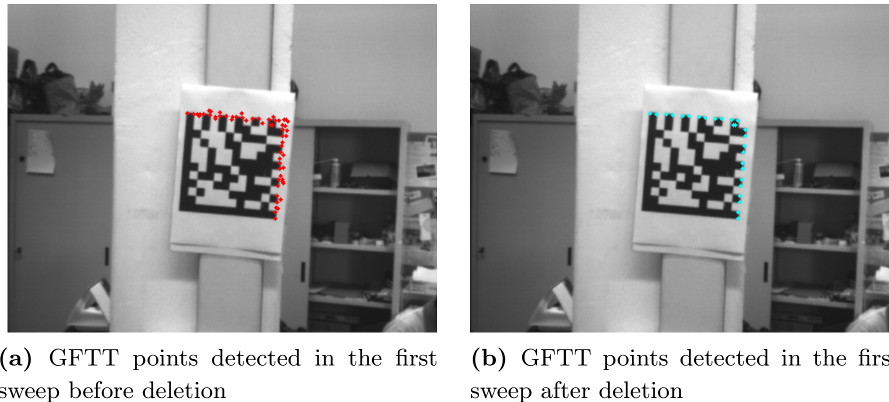
Computing the intersection bewteen these two lines, we found a new top-right corner, more precise than the one directly calculated with *libdmtx*.

There is a major limit in this operation: in order to compute a linear regression on a set of points with the OpenCV function, there must be at least 4 such points, otherwise the algorithm fails to execute. For this reason, if GFTT finds only 3 features or less, or after the points deletion we do not have at least 4 points, the whole procedure stops and the top-right corner of the barcode is not refined.

After the first phase, we executed again the same steps just described (with the exception of the bottom-left point, which we explained was already accurate after just one GFTT execution), considering the new top-right point found for all the operations. Even if in most cases the first sweep would have been sufficient, we decided to execute twice the operations previously described to gain more accuracy in marker detection within the camera image, in order to get a better barcode pose estimation in camera coordinates system.

The operations executed are further refined: for example, the masks along the dotted edges are built to be thinner than previously and we force GFTT to search a smaller number of points (about 15), with a high quality value, so that only salient corners could be recognized. In the end, we executed again two linear regression operations to fit the straight lines passing through the detected points, which in the optimal case are exactly the top and the right borders of the Data Matrix: the intersection between them gave us the correct top-right marker point, that could now replace the original one provided by *libdmtx*.

After the two-sweep operations, we got as result an improved estimate of two Data Matrix points: the top-right and the bottom-left ones. We are now able to draw a precise contour around the marker edges, using the two new corners and the ones initially provided by the library, i.e., the bottom-right and the top-left points. These latter ones are in general precise, because



(a) GFTT points detected in the first sweep before deletion

(b) GFTT points detected in the first sweep after deletion

Figure 4.14: GFTT points detection and deletion

they are directly returned by *libdmtx*, without any supplementary operation, but for this reason they still suffer of a slight inaccuracy: they are retrieved with integer values. A further application of a sub-pixel corner refinement generally helps to sharpen their precision in pixel terms. Figure 4.15 shows the sought marker contour. It is clearly visible that it fits almost perfectly with the effective marker perimeter, improving the pixel estimates with respect to the case in which the original four corners without refinements were used. Of course results are not always so perfect, because there can be still some imprecisions due to image distortion and blurring or to changes in light intensity, but in the general case the improvements using GFTT are

Figure 4.15: Data Matrix final contour with GFTT

Figure 4.16: Data Matrix final contour with GFTT - far perspective

remarkable.

Using GFTT methods for marker corners polishing also gives good results when the marker is positioned far from the camera, like 5-6 m (see Figure 4.16): of course we cannot expect a precision comparable to the case in which the Data Matrix is very close, but on average we found it acceptable (numerical results are proposed in the next chapter) and better than without using GFTT, anyway.

It is important to point out that, as we said, this set of operations does not always terminate correctly, because GFTT should find at least 4 features on the marker edges in order to let the fitLine OpenCV function to fit the two lines we were talking about. This can sometimes be an important limitation, because it lowers the marker recognition rate: even if *libdmtx* correctly detects a Data Matrix, whenever GFTT does not manage to find a sufficient quantity of features along the barcode edges the marker it is considered lost, like if there is none in the image, leading to a heavy information loss for the EKF-SLAM algorithm. For this reason we decided to handle with the original Data Matrix contour (with the points returned by *libdmtx*) in case of GFTT failure, in order to have good corner estimates in most situations, but without loosing too many detections at the same time.

4.4.3 Canny edge detection and Hough algorithms

Beyond GFTT applications, we tried to use and analyse other computer vision methods to see if we could get more accuracy in marker graphical

detection. Since a Data Matrix has two clear black borders, we considered to use an edge detector to find straight lines patterns within the image and hopefully to precisely locate the marker on the two-dimensional plane.

A consolidate algorithm widely used for edge detection is the one created by Canny [4], which, given a camera image, returns a black image with a set of white curved and straight lines corresponding to the most outstanding edges: this image is computed by finding all the luminosity gradients in the original frame, where bigger gradients corresponds with higher probability to edges. Given the output image with highlighted edges, a way to pick out straight lines is to employ Hough algorithm [16], which, with the use of an accumulator, manages to return a set of straight infinite lines equations that characterize exactly all the edges available in the edge image (i.e., the one resulting after Canny application).

There are some parameters to set while using these two algorithms, like intensity threshold, maximum number of lines, accumulator threshold, that may help to better spot the correct lines we are looking for, i.e., the four Data Matrix borders.

Computing the algorithms just described over the whole camera frame would be a nonsense, because in a full image the number of relevant edges is huge. This fact makes highly improbable to locate correctly a small Data Matrix within the image: even if setting the thresholds to be very restrictive can reduce the number of edges, we could actually loose edges belonging to the marker as well.

Like in the case of GFTT application, we decided to execute the two algorithms on sub-areas of the whole image, trying to catch only the lines on the marker borders, in order to operate only on them and not on a vast number of lines. Starting from the classic points returned by *libdmtx* we are able to build four little rectangular masks around each Data Matrix border, in the way shown in Figure 4.17.

With this trick we were able to run the Canny edge detector only on the specified ROIs, in order to find more accurately only the marker borders. The result after Canny algorithm execution is shown in Figure 4.18, where the Data Matrix borders are easily distinguishable in the four masks.

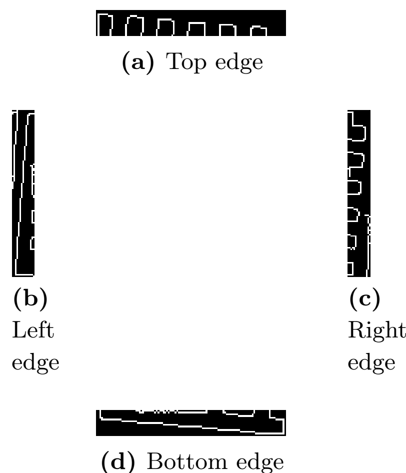
Now we can use each small image with relevant edges to execute Hough algorithm, find linear patterns and pick out a set of candidate lines: setting a proper value for the accumulator threshold, to avoid to detect too many or too few lines, gave us a good result, like the one shown in Figure 4.19a.

Between all the retrieved lines, we had to find the ones corresponding exactly to the marker edges. To do that we could only use the approximative information we had from the Data Matrix library: the edge slope (in integer

Figure 4.17: ROIs around the marker borders for Canny application

degrees) and the edge endpoints (i.e., the marker corners, taken two by two). For each set of lines of a marker edge, we did the following: first of all, we picked out the ones having a slope angle value the closest possible to the one suggested by *libdmtx* (in case of multiple parallel candidates with such slope, we chose all of them); then, among the remaining lines, we selected the only one for which is minimal the distance from the two library edge corners. We obtained with this method the four definitive candidate lines for the marker borders, as shown in Figure 4.19b.

At this point, having four line equations, it was pretty straightforward



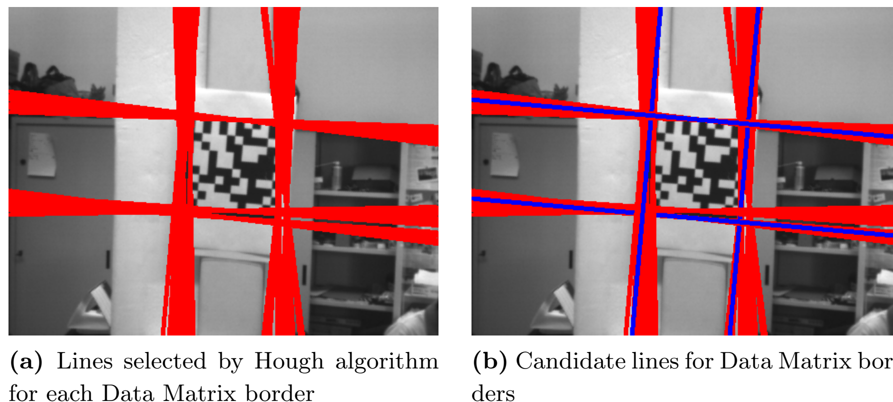
(a) Top edge

(b) Left edge

(c) Right edge

(d) Bottom edge

Figure 4.18: Data Matrix edges after Canny detection



(a) Lines selected by Hough algorithm for each Data Matrix border

(b) Candidate lines for Data Matrix borders

Figure 4.19: Hough algorithm lines selection

to calculate the four intersections, obtaining the final four Data Matrix corners, and consequently the whole marker contour. Also in this case, if the algorithm does not return at least one line per ROI, we use the original Data Matrix points for pose estimation.

In Figure 4.20 we reported the final result for the Data Matrix already considered in the previous figures: since the marker is pretty close to the camera (about 1 meter), Canny and Hough algorithm managed to perfectly detect the barcode edges and locate it within the image, exactly like the case in which we used GFTT.

This method has a huge drawback, though: the candidate lines selection after executing Canny and Hough algorithms is based on information given from *libdmtx*, which are very inaccurate, because they are handled with integer values, without using a better precision.

A slight external noise is enough to make the library think that the Data Matrix is, for example, rotated of a wrong angle respect to its real inclination or that the corner points are shifted of some pixels respect their real location, especially when the marker is at a substantial distance from the camera, in which case the library looses precision.

This can affect the lines selection in the described method in two ways:

- a line with a wrong slope is picked, so that it may not represent the correct marker edge;
- a line parallel to the real edge is picked, because even if the slope is the right one, the wrong estimates of the corner points may lead to choose a line further than the correct border.

Figure 4.20: Data Matrix final contour with Canny

Figure 4.21 is an example of the latter case, considering a Data Matrix far from the camera: it is visible that the left edge is misclassified because a near parallel line was chosen as edge candidate.

4.4.4 FAST algorithm

FAST (Features from Accelerated Segment Test) algorithm [33] is another efficient corner detector. OpenCV provides a function to compute it over an image with return as set of points corresponding to salient features. Unfortunately, the function does not provide the possibility to specify masks where compute the search on. For this reason, and also to make the scan much faster, we decided to run the algorithm over the four ROIs already described talking about Canny algorithm (refer again to Figure 4.17). The average time employed to detect features over a small ROI is about 1-3 ms, so the algorithm time complexity can be ignored.

The only problem using this method is that OpenCV ROIs can have only rectangular shape. If the Data Matrix is slanted (the worst case when it has a 45 degrees inclination), the regions of interest would cover also internal parts of the marker, detecting points not laying on the marker edges, therefore useless. In order to pass over this problem, we apply two corrections:

- like in the case of GFTT refinement, we delete all the points that have a white colouration over a certain threshold: in this way we do not risk to consider points falling out of the Data Matrix black corners;

Figure 4.21: Data Matrix final contour with Canny - far perspective

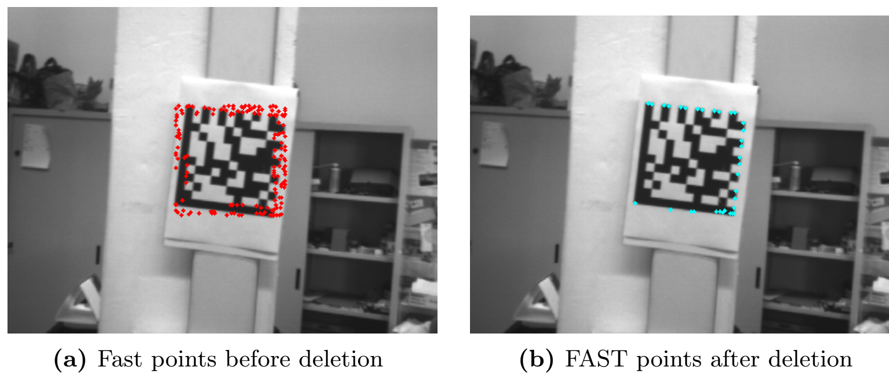
- given the *libdmtx* corner points, we select the features whose orthogonal distance from the segments joining the marker corners (considered by twos) is under a certain threshold (i.e., 2 pixels).

In this way we aim to select only features laying on the marker edges and then fit the lines passing through them, finding the four intersections which hopefully correspond to the Data Matrix corners.

Figure 4.22 shows the results of the described corrections: as you can see, the algorithm finds a very high number of salient points, even off Data Matrix area, but after deleting the points according to what just explained, we get a nice result.

Of course, it is more difficult for FAST algorithm to find features along the marker straight edges, because there are no corners along them. In this cases, we do not fit the corresponding line and we do not refine the correspondent corners, keeping the ones returned by *libdmtx*. A generic result of corners refinement using FAST algorithm is shown in Figure 4.23.

We briefly mention that the method just described for FAST algorithm can be applied in the same way (which means executing exactly the same operations) using GFTT algorithm for features detections. The results are very similar to the ones here reported, but in this specific case FAST algorithm is more desirable, because it is faster than GFTT. On the other hand, we already specified the pros of GFTT, having the possibility to select precise masks where to execute the search.



(a) Fast points before deletion (b) FAST points after deletion

Figure 4.22: FAST points detection

4.5 Data Matrix pose quantitative analysis

In this section, we report the numerical analysis about marker pose estimation with the vision techniques described in the previous section. We compare the the marker pose values (translation and Euler angles) obtained after pose estimation with marker detection using GFTT, FAST and Canny together with Hough algorithms.

We did several trials, first to analyse the marker detection static error on the six pose variables over a static scene, comparing their values with a classic chessboard pose values (which can be calculated with high accuracy), then we computed the marker pose error over a dynamic scene (which means

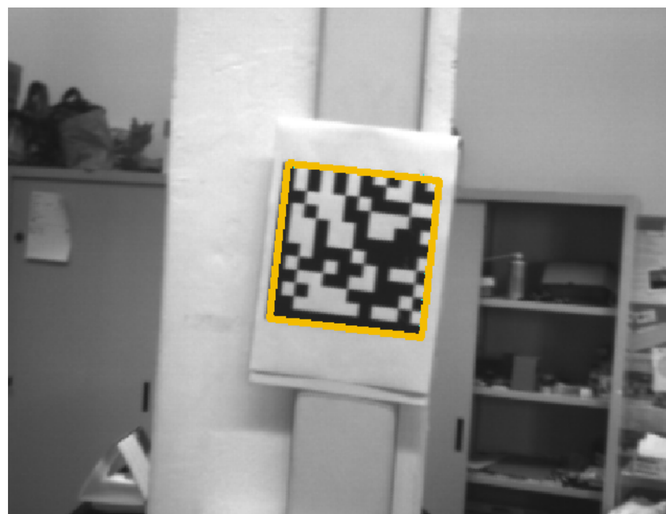


Figure 4.23: Data Matrix final contour with FAST algorithm

with a moving camera), always analysing the difference between the marker and the chessboard pose values.

4.5.1 Data Matrix detection - static error

In order to choose the best computer vision algorithm to refine the marker image points, we run the detection process for marker recognition with a specific refinement algorithm for a certain number of cycles under different conditions, on a static scene; then we solved the perspective N-points problem obtaining how the marker is translated and rotated with respect to the camera, i.e. translation coordinates and Euler angles (a total of 6 values, for 6 degrees of freedom) of the marker system. Similarly, we retrieved the same pose values in camera reference frame for a 9×9 chessboard. Finally, for each of the 6 variables, we compared the obtained results for the marker and the chessboard: in fact, since the chessboard has a large number of internal points (exactly 81 internal intersections) easily detectable with the help of OpenCV because they follow a repetitive pattern, it is possible to calculate almost perfectly its pose in camera coordinates, with a very low error rate. The comparison gave us an indication of how good was the chosen corners refinement algorithm.

We built a rigid composition of a marker with a chessboard, as depicted in Figure 4.24: chessboard system origin is on the top-left corner (x-axis points toward down and y-axis toward right), without considering the most external squares, while Data Matrix origin is as usual its bottom-left corner but it is rotated in this case to make the two coordinates systems equivalent, except of a simple translation along x-axis. Notice that in both systems the objects lay only on XY plane, with null values along z-axis.

With this configuration, it is possible to detect a Data Matrix and a chessboard in the same frame, compute their pose with respect to the camera system and compare the resulting poses. Of course, first it is necessary to overlap the two coordinates systems, in order to have comparable pose values. If we call $\mathbf{RT}_{Ch'}^{C}$ the new chessboard pose in camera system (i.e., the chessboard system shifted), we get

$$\begin{aligned}\mathbf{RT}_{Ch'}^{C} &= \begin{bmatrix}\mathbf{R}_{Ch}^{C} & \mathbf{T}_{Ch}^{C} \\ 0 & 1\end{bmatrix}\begin{bmatrix}\mathbf{R}_{Ch'}^{Ch} & \mathbf{T}_{Ch'}^{Ch} \\ 0 & 1\end{bmatrix} = \\ &= \begin{bmatrix}\mathbf{R}_{Ch}^{C}\mathbf{R}_{Ch'}^{Ch} & \mathbf{R}_{Ch}^{C}\mathbf{T}_{Ch'}^{Ch} + \mathbf{T}_{Ch}^{C} \\ 0 & 1\end{bmatrix} = \begin{bmatrix}\mathbf{R}_{Ch}^{C} & \mathbf{R}_{Ch}^{C}\mathbf{T}_{Ch'}^{Ch} + \mathbf{T}_{Ch}^{C} \\ 0 & 1\end{bmatrix}\end{aligned} \tag{4.18}$$

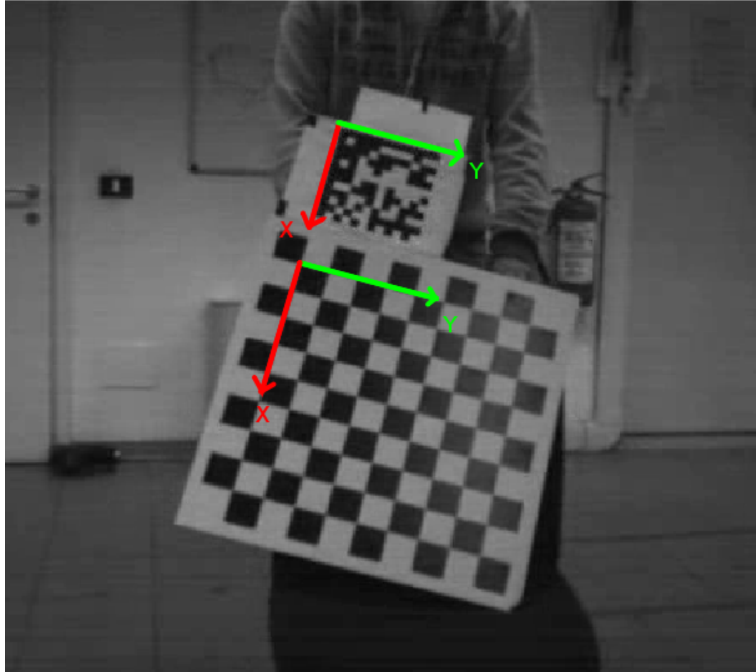
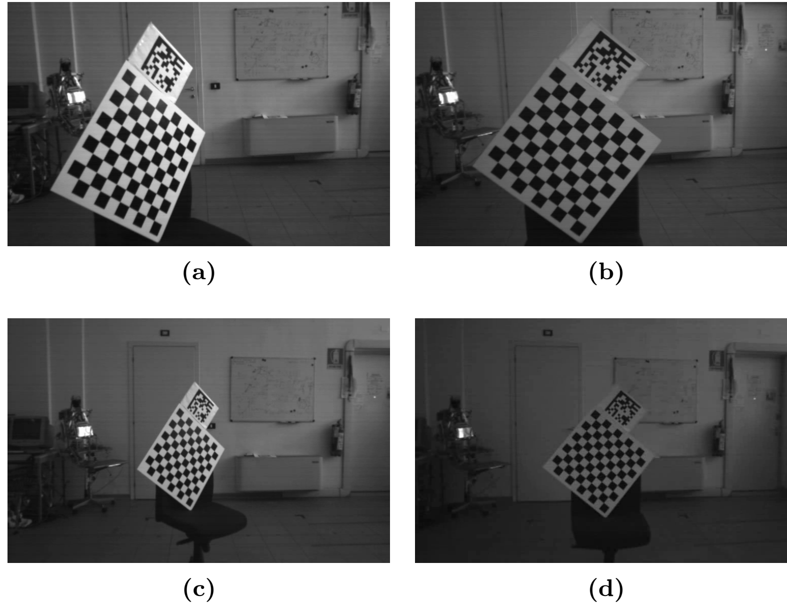


Figure 4.24: Marker on chessboard

where $\mathbf{R}^{C}_{Ch}$ and $\mathbf{T}^{C}_{Ch}$ represent the original marker pose in camera frame, $\mathbf{R}^{Ch}_{Ch'}$ is an identity matrix (there is no rotation translating the chessboard coordinates system on the marker) and $\mathbf{T}^{Ch}_{Ch'} = [d, 0, 0]^T$ is the chessboard translation to overlap the marker coordinates (note that it is just a shift along x-axis of a certain value d).

In order to analyse the static error, we processed the Data Matrix detection with all the three described refinement algorithms for some known marker-camera (and consequently chessboard-camera) configurations. In particular:

- marker positioned close to the camera (about 1.5 m) and with a consistent inclination (Figure 4.25a)
- marker positioned close (about 1.4 m) and frontal to the camera (Figure 4.25b)
- marker positioned far from the camera (about 2.55 m) and with a consistent inclination (Figure 4.25c)
- marker positioned far (about 2.45 m) and frontal to the camera (Figure 4.25d)



(a) (b) (c) (d)

Figure 4.25: Marker-camera configurations for static error analysis

For each one of these static configurations we run the recognition process over about 400 image frames and then we show the results obtained comparing the estimated chessboard pose with the estimated marker pose, which means calculating their errors as the difference between their position values (X, Y and Z translations - in meters) and attitude values (roll (ϕ), pitch (θ) and yaw (ψ) Euler angles - in degrees), considering all the vision algorithm for corner refinement previously described (GFTT, FAST and Canny).

In the figures shown below we report the cited errors, with their mean and median values. We expected to see errors values with zero mean, because we assumed that the rototranslation between the chessboard and the marker systems corresponded only to a simple translation along X-axis, as explained before. This is an optimistic assumption, because minor rotations or translations may have occurred from human errors, which can cause a slightly wrong rototranslation than the real one. For this reason, we obtained errors with mean slightly different from zero. We report also the median values, because it happens (especially when the marker is far from the camera) that PnP problem is not solved correctly, so there may be outliers in error pose values, and the median is more robust than the mean in these situations.

We need to say that the marker-chessboard composition we built can influence the marker pose estimate, because when the marker is turned upside down, *libdmtx* returns corners less precise than usual, and this can affect the

pose estimate, in particular when we use GFTT algorithm, in which we refine only two of the four marker corners. However, we present this composition to show the possible errors in the worst case.

Notice that in the next set of figures all the graphics are reported with the same axis range for each method in the same configuration, but the range changes in every configuration, so refer to the values next to the chart axes.

Marker transversal and close to camera objective

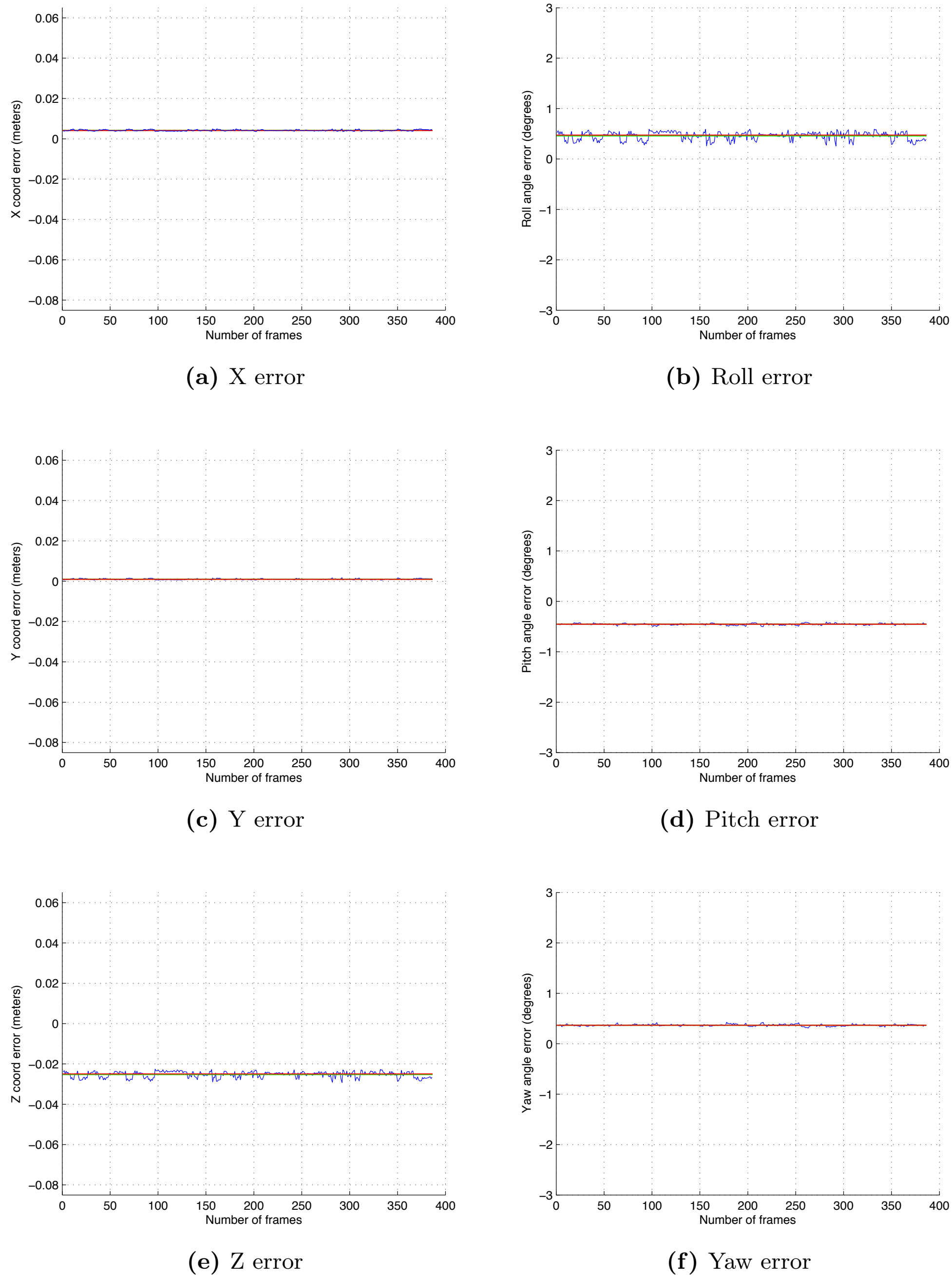
The pose errors for this configuration using GFTT, FAST and Canny algorithms are respectively reported in Figure 4.26, 4.27 and 4.28, and summarized in Table 4.2.

Considering the errors on position values, we notice that for all the three vision algorithms the mean and median values are very close to zero (for the considerations previously explained), moved away of few millimetres for X and Y coordinates and and few centimetres for Z coordinate. As you can see in the mentioned figures, also the relative maximum error gap is very low, with GFTT that has the lowest gaps (millimeters order) than the other two algorithms (centimeters order).

As far as it concerns the error on attitudes, the same considerations about means and medians hold. We notice that the errors are very contained around the mean value for GFTT algorithm, while using FAST there is a wider error gap on the angles (i.e., about 3.5 degrees on roll, comparing to 0.33 degrees in GFTT).

Canny method provides us a particular case: since the detected lines are always the same around a Data Matrix (i.e., the algorithm chooses often the right line or a very close parallel one), the error graphics mostly report discrete values, so we see the errors to "jump" from two values (i.e., considering roll angle, from -2 to 2 degrees). In this cases, we notice the difference between means and medians: for example, considering the error on pitch angle, we see that the median value is -0.0245 degrees, but because of a consistent number of outliers, the mean value is brought up to about 1.1 degrees. Unfortunately, this pattern is quite common using Canny algorithm, and as we will show, it causes bad results on far distances.

To summarize, in this configuration we get the lowest errors using GFTT method, because it is stable and keeps the error gaps very low, without any visible outlier. FAST and Canny algorithms, instead, show wider error ranges over all the pose values, with clearly visible wrong values which highly increase the pose errors, especially using Canny, even though they are still low, due to the small distance from marker to camera.



(a) X error

(b) Roll error

(c) Y error

(d) Pitch error

(e) Z error

(f) Yaw error

Figure 4.26: Errors between chessboard and marker poses with mean (green) and median (red) values using GFTT method in lateral close configuration

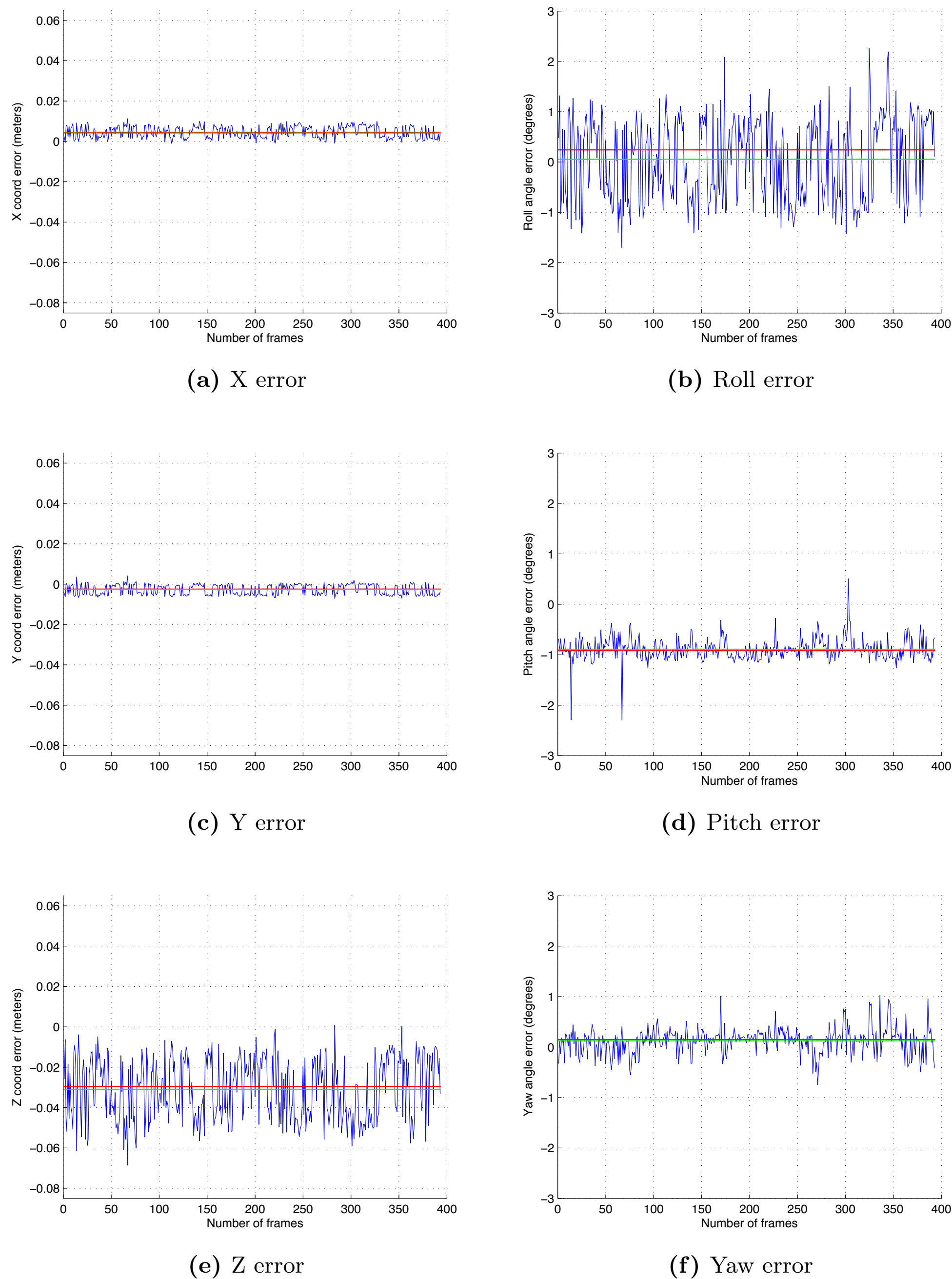
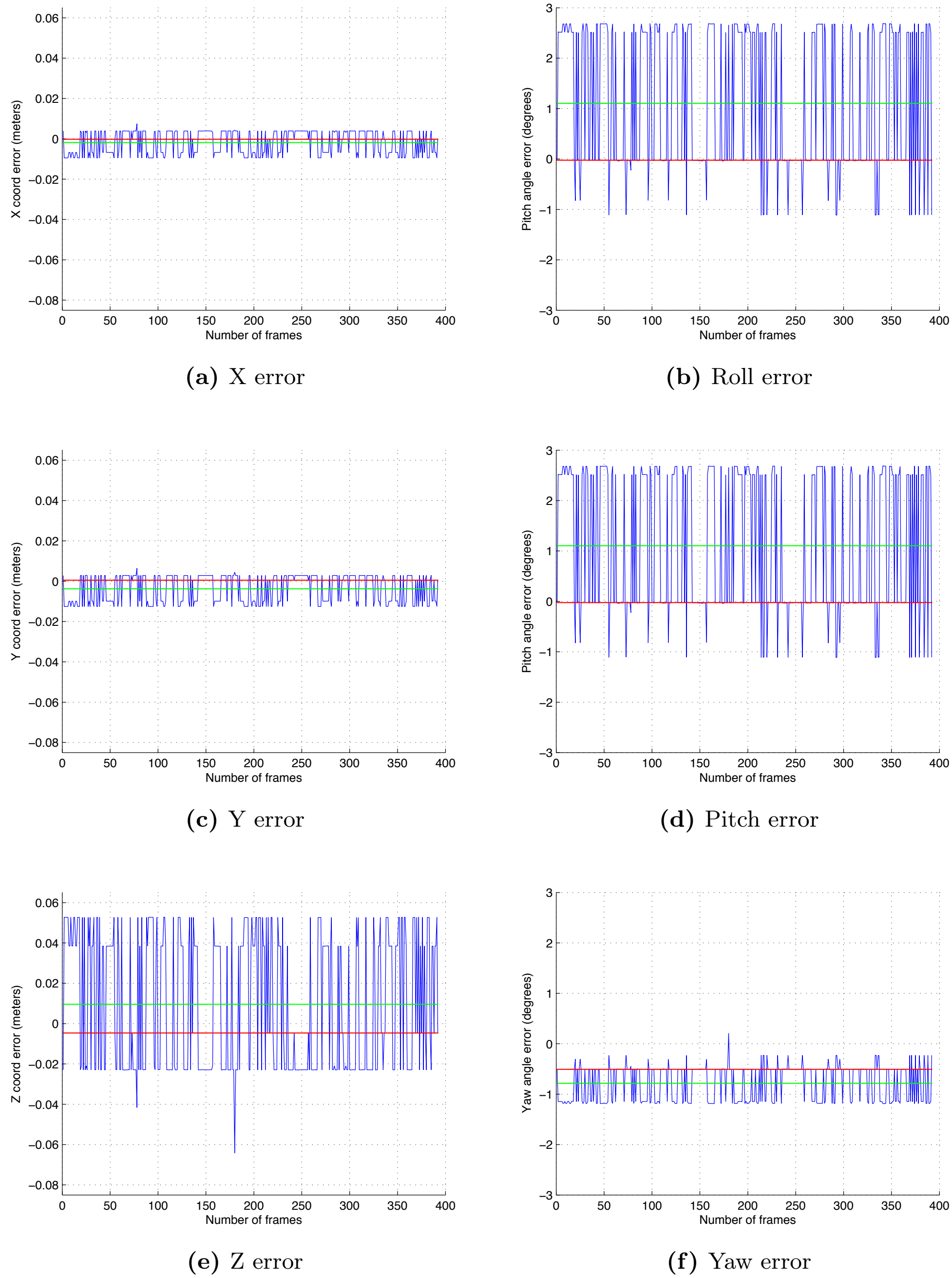


Figure 4.27: Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in lateral close configuration



(a) X error

(b) Roll error

(c) Y error

(d) Pitch error

(e) Z error

(f) Yaw error

Figure 4.28: Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in lateral close configuration

		Vision algorithm		
		GFTT	**FAST**	**Canny**
Mean value	ϕ (degrees)	0.4567	0.1065	-0.1598
	θ (degrees)	-0.4539	-0.9747	1.1043
	ψ (degrees)	0.3684	0.2336	-0.7857
	X (m)	0.0042	0.0045	-0.002
	Y (m)	0.001	-0.003	-0.0037
	Z (m)	-0.0253	-0.0289	0.0096
Median value	ϕ (degrees)	0.4735	0.314	-0.19
	θ (degrees)	-0.4537	-0.9659	-0.0245
	ψ (degrees)	0.368	0.219	-0.507
	X (m)	0.0041	0.0055	-0.0002
	Y (m)	0.0009	-0.0015	0.0005
	Z (m)	-0.025	-0.0331	-0.0047
Max error gap	ϕ (degrees)	0.336	3.547	5.894
	θ (degrees)	0.0963	2.131	3.805
	ψ (degrees)	0.116	1.321	1.393
	X (m)	0.0013	0.0115	0.017
	Y (m)	0.0012	0.0103	0.019
	Z (m)	0.0066	0.062	0.117

Table 4.2: Lateral close marker-camera configuration - summary

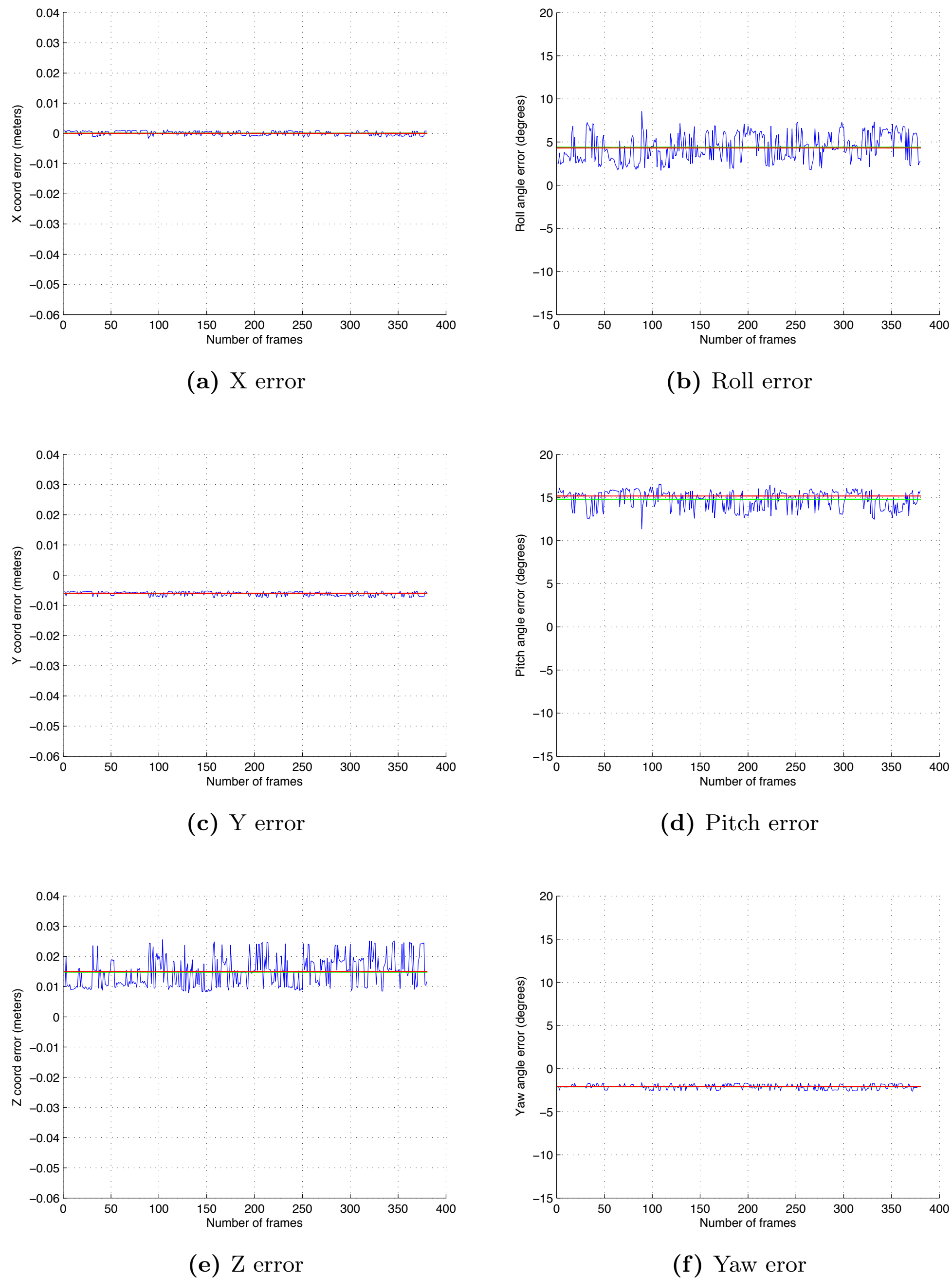
Marker frontal and close to camera objective

When a marker is frontal to the camera, the PnP problem is critic, using a low number of image points. In fact, considering the plane where the marker lays (XY plane in our convention), a change in its inclination of few degrees around X or Y axis does not affect in a noticeable way the position of the pixels in the image, from the point of view of the camera. For this reason, a slight variation in pixel positions due to external noises may result in consistent mistakes in marker orientation (and, in minor way, position) estimates. This situation can happen even if the marker is not frontal, because the noise can always shift pixel positions, but of course it happens much less.

The pose errors for the considered configuration using GFTT, FAST and Canny algorithms are respectively reported in Figure 4.29, 4.30 and 4.31, and summarized in Table 4.3.

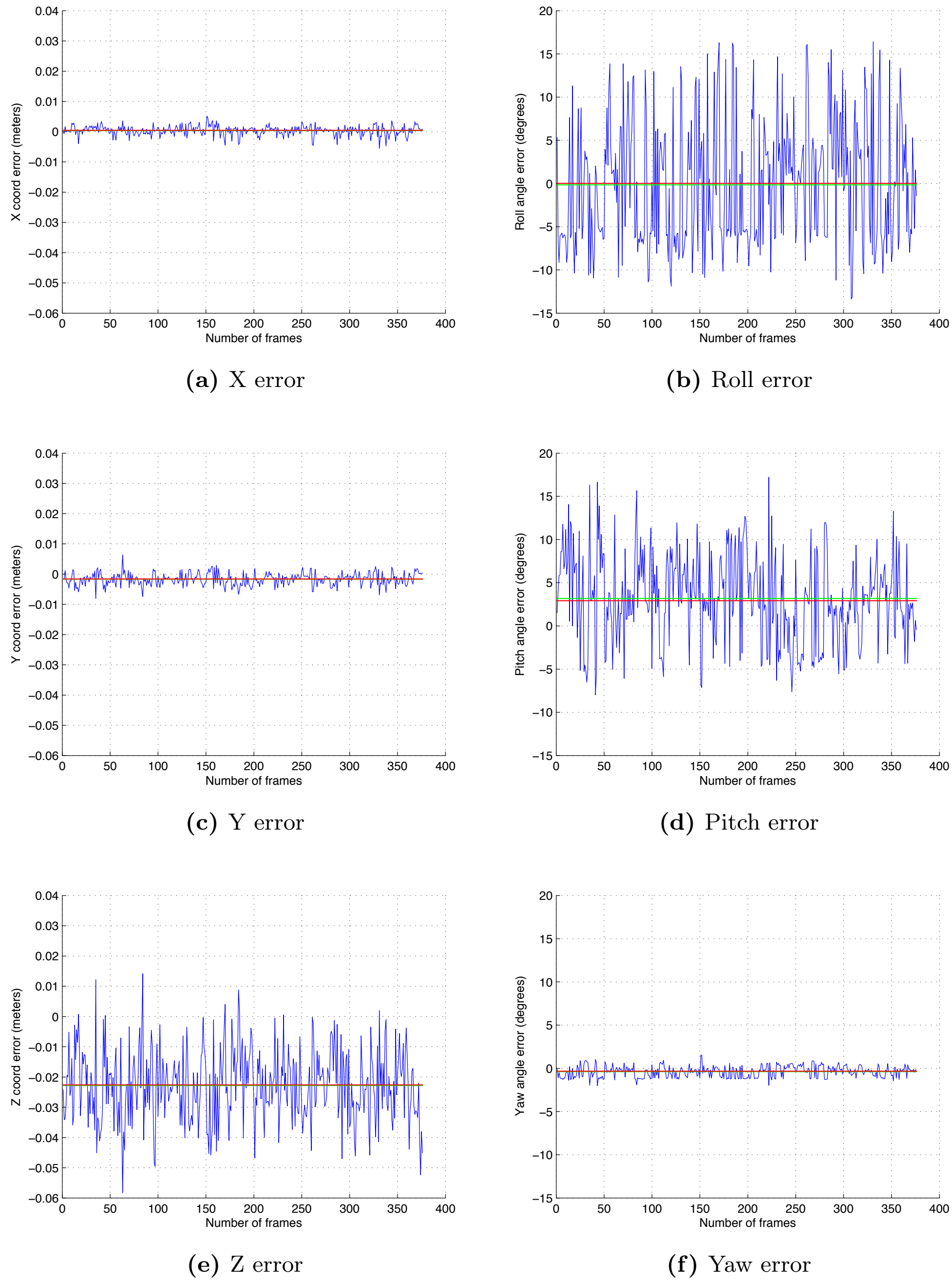
Also in this configuration, we notice that for the position values, they same considerations mentioned before hold. That is, we have mean and median values very close to zero for each refinement algorithm, and in general very low error gaps (in the order of centimeters using FAST and Canny, and millimeters usign GFTT), even if we notice some unbalanced peaks, especially for Canny algorithm. Such peaks are due to the fact just explained, but on position errors are still limited, while if we look at orientation errors we see that the peaks heavily affect the error gaps, because they show that the marker attitude was wrongly estimated (and this is clearly visible looking at Canny, where the go out from the reported graphics, and, less clearly, looking at FAST algorithms), even if it is not a frequent case.

GFTT seems to work in good way, because errors are contained around mean values. However, looking carefully at mean and median values, we can notice that the average error on pitch angle using GFTT is very far from zero, in fact it is about 14-15 degrees. Since only GFTT presents this oddity, we understand that it represents an exception. Even though the algorithm itself returns stable results (as we said, low error gaps), a high mean value like the one reported means that the marker is detected with a wrong orientation most of the times. This is a huge flaw for detection using GFTT, as we will see again later talking about dynamic errors. We have to say that this fact is not a common pattern, because it depends much on how the Data Matrix is rotated: when its orientation is straight, this problem does not occur. We reported anyway the worst case, to show every possible flaw.



(a) X error
(b) Roll error
(c) Y error
(d) Pitch error
(e) Z error
(f) Yaw eror

Figure 4.29: Errors between chessboard and marker poses with mean (green) and median (red) values usin GFTT method in frontal close configuration



(a) X error

(b) Roll error

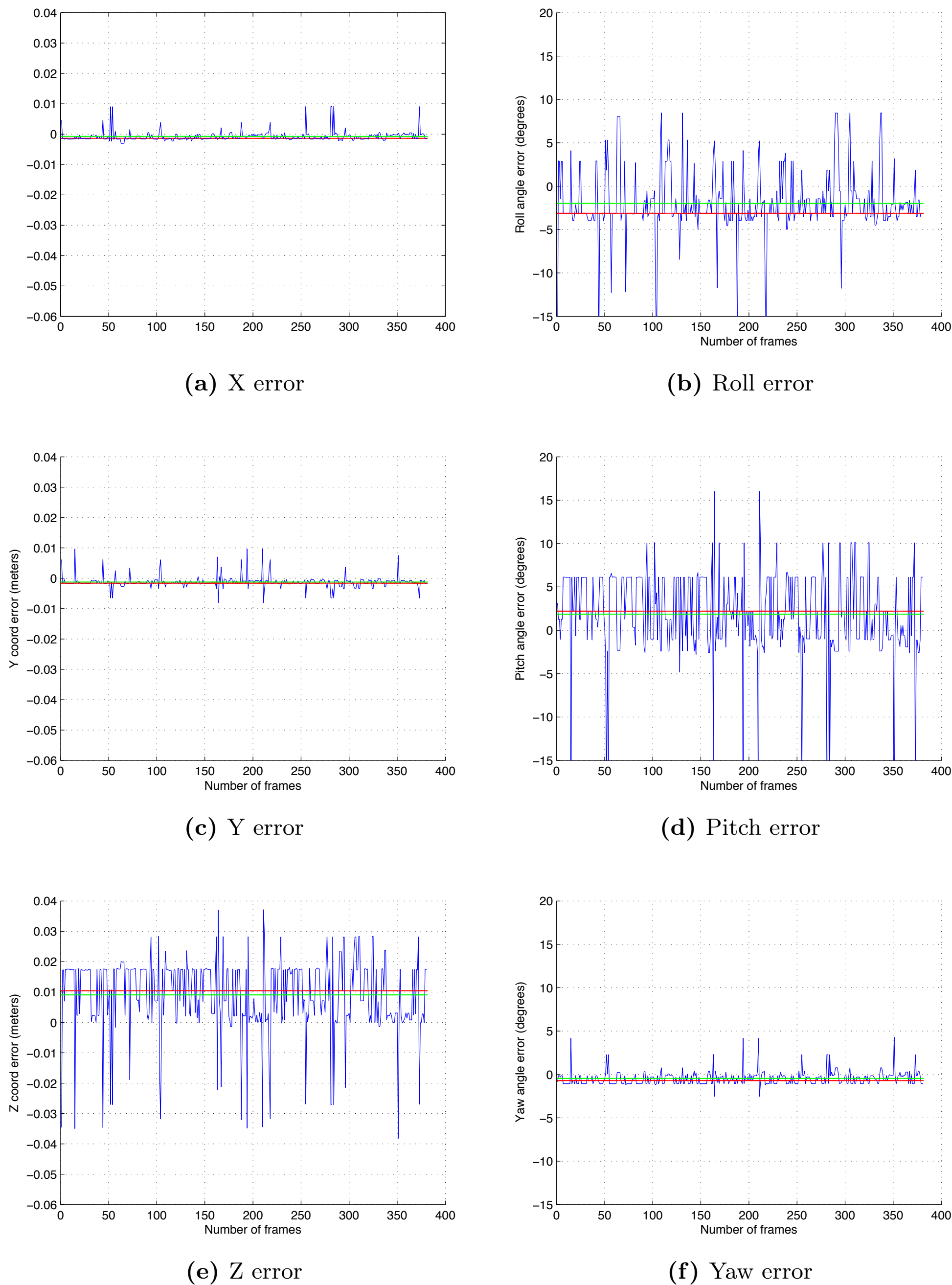
(c) Y error

(d) Pitch error

(e) Z error

(f) Yaw error

Figure 4.30: Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in frontal close configuration



(a) X error

(b) Roll error

(c) Y error

(d) Pitch error

(e) Z error

(f) Yaw error

Figure 4.31: Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in frontal close configuration

		Vision algorithm		
		GFTT	**FAST**	**Canny**
Mean value	ϕ (degrees)	4.3962	-0.57	-1.98
	θ (degrees)	14.7852	3.1695	1.859
	ψ (degrees)	-2.0962	-0.4181	-0.4529
	X (m)	0.0001	0.0003	-0.0008
	Y (m)	-0.0061	-0.0015	-0.0012
	Z (m)	0.0148	-0.0234	0.0091
Median value	ϕ (degrees)	4.3165	0.267	-3.128
	θ (degrees)	15.1734	3.996	2.2097
	ψ (degrees)	-2.078	-0.473	-0.707
	X (m)	0.0000	0.0004	-0.0014
	Y (m)	-0.006	-0.0014	-0.0016
	Z (m)	0.015	-0.0248	0.0104
Max error gap	ϕ (degrees)	6.837	27.849	27.344
	θ (degrees)	5.183	20.7841	44.2126
	ψ (degrees)	0.99	2.312	6.838
	X (m)	0.003	0.0083	0.0122
	Y (m)	0.0024	0.011	0.0177
	Z (m)	0.0176	0.0568	0.0753

Table 4.3: Frontal close marker-camera configuration - summary

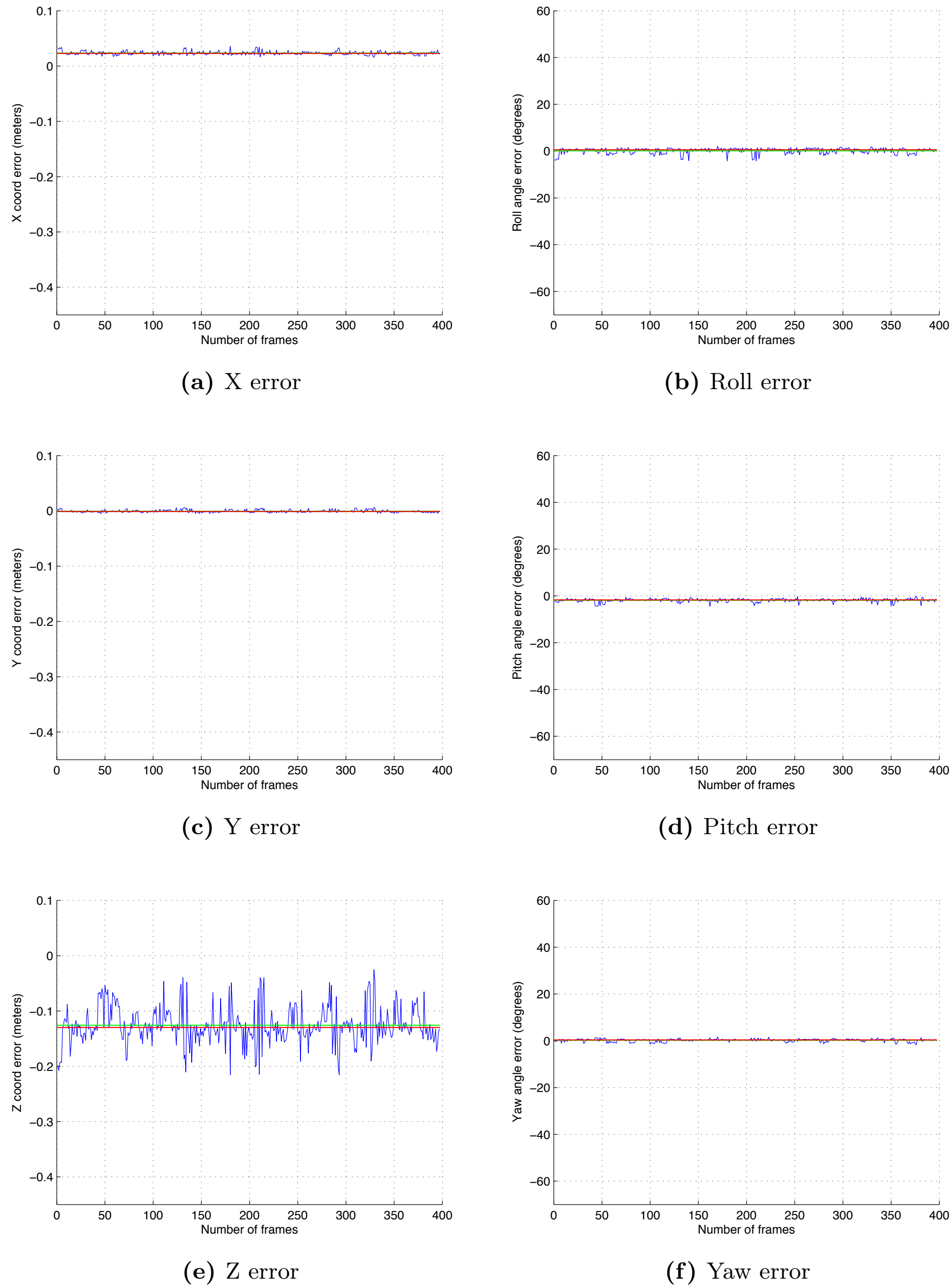
Marker transversal and far from camera objective

When the Data Matrix is positioned relatively far from the camera (over 2 metres), more uncertainty is introduced in the marker pose estimation process, due to the fact that the marker has a very small size compared to the whole image size. Therefore, we expect to see higher error values on every pose value, but keeping the marker transversal to the camera, we hope to see stable patterns respect to the case of frontal marker.

The pose errors for the considered configuration using GFTT, FAST and Canny algorithms are respectively reported in Figure 4.32, 4.33 and 4.34, and summarized in Table 4.4. We still notice precise values for X and Y coordinates errors (few centimetres), with almost zero-mean, for all the algorithms (Canny performs a little bit worse). Regarding Z coordinate, we see that when the marker is far (about 2.5 m), the error is quite high for GFTT and FAST, with an error gap of about 20 cm, while using Canny the error gap is even 40 cm, with some outliers that increase the total gap (they go out of the reported graphics). Also median values on Z show that the marker distance is always slightly overestimated than the real one.

Talking about orientation values, we observe that GFTT still is quite stable, since error gaps are relatively reduced and mean values close to zero. Some outliers are present, but their value is restrained, anyway. We find a different situation for the remaining algorithms. FAST shows some outliers (especially visible in roll and pitch angle errors), due to the wrongly estimated pose, which highly increase the error gap, but they are very few respect to the whole dataset. In fact, in the figure you can see that the median value is drawn along the correct error distribution, because it is less sensible to said outliers. Canny algorithm instead, performs quite poorly because there is a larger number of outliers, and in general we do not have zero-median errors, because it is likely that the marker pose was estimated wrongly. This was due to the fact that Canny algorithm could be sensibly wrong in detecting marker edges from large distances, because of the parallel lines near the Data Matrix borders and the low accuracy granted by *libdmtx* (refer again to Section 4.4).

As general results, we still confirm the stability given by GFTT algorithm, but also FAST errors follow a quite linear pattern, even if the detection process seldom suffers of wrong pose estimation. Canny algorithm instead is much more sensible to noise and this results in high pose errors.

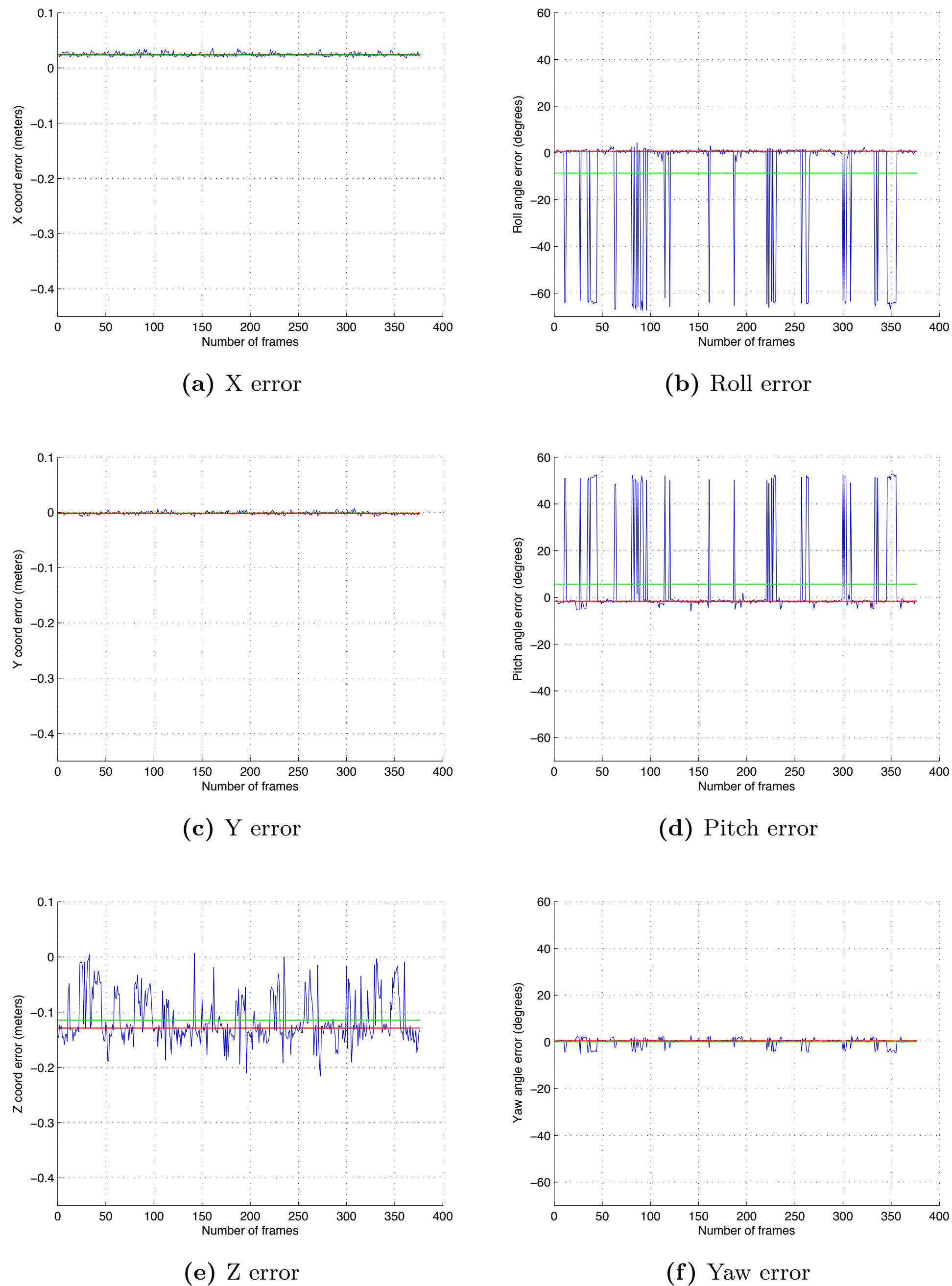


(a) X error **(b)** Roll error

(c) Y error **(d)** Pitch error

(e) Z error **(f)** Yaw error

Figure 4.32: Errors between chessboard and marker poses with mean (green) and median (red) values using GFTT method in lateral far configuration

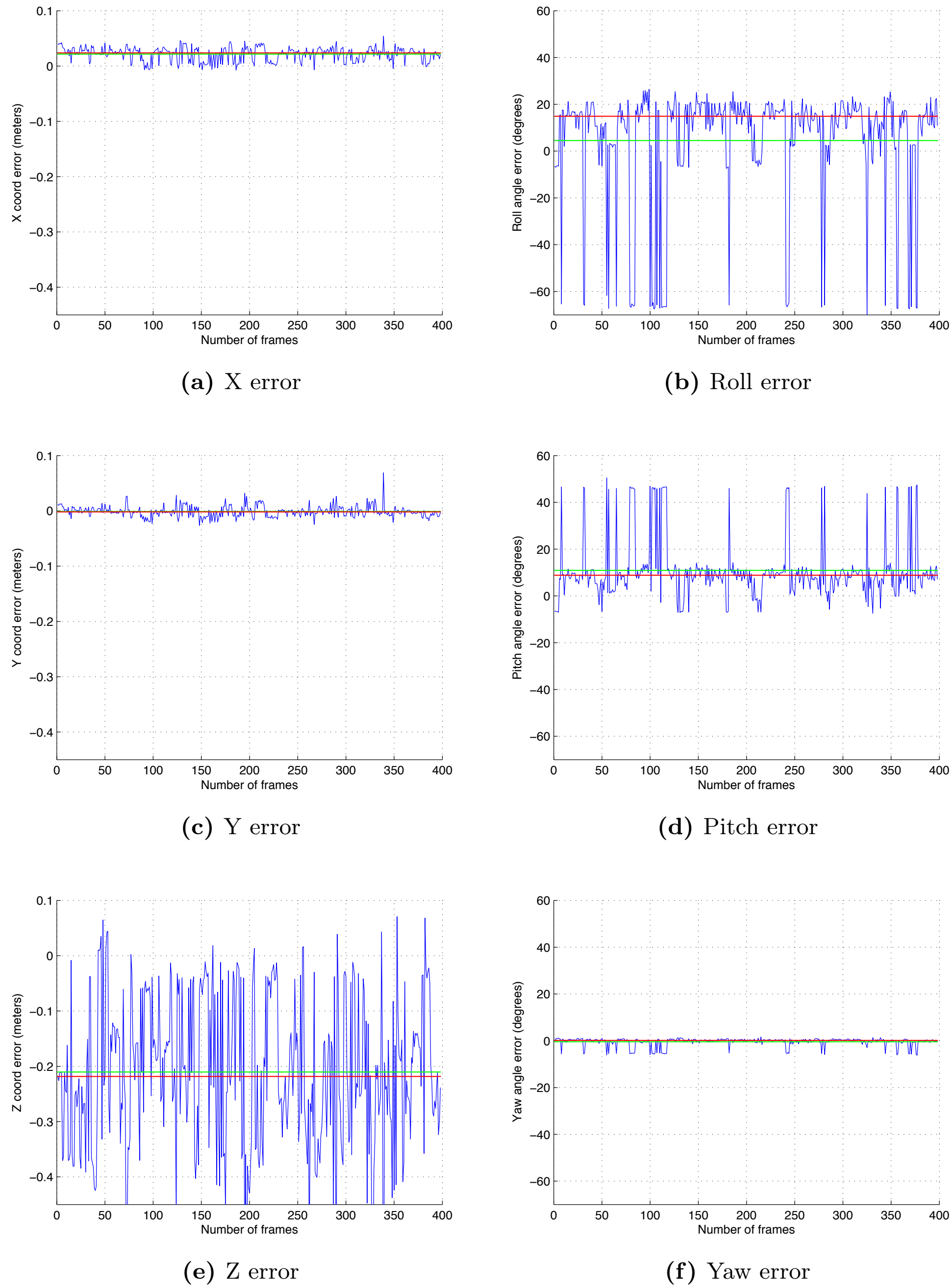


(a) X error **(b)** Roll error

(c) Y error **(d)** Pitch error

(e) Z error **(f)** Yaw error

Figure 4.33: Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in lateral far configuration



(a) X error

(b) Roll error

(c) Y error

(d) Pitch error

(e) Z error

(f) Yaw error

Figure 4.34: Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in lateral far configuration

		Vision algorithm		
		GFTT	**FAST**	**Canny**
Mean value	$\boldsymbol{\phi}$ (degrees)	0.157	-7.4024	4.5229
	$\boldsymbol{\theta}$ (degrees)	-1.851	4.4572	10.9072
	$\boldsymbol{\psi}$ (degrees)	0.2435	0.0232	-0.4232
	X (m)	0.0234	0.0244	0.0216
	Y (m)	-0.0007	-0.001	-0.0012
	Z (m)	-0.1258	-0.1218	-0.2103
Median value	$\boldsymbol{\phi}$ (degrees)	0.591	0.653	14.905
	$\boldsymbol{\theta}$ (degrees)	-1.7326	-1.6653	8.8438
	$\boldsymbol{\psi}$ (degrees)	0.384	0.442	0.1595
	X (m)	0.0228	0.0238	0.0238
	Y (m)	-0.0011	-0.001	-0.0018
	Z (m)	-0.1298	-0.13	-0.2184
Max error gap	$\boldsymbol{\phi}$ (degrees)	6.321	71.212	97.091
	$\boldsymbol{\theta}$ (degrees)	4.4145	58.1294	57.9923
	$\boldsymbol{\psi}$ (degrees)	3.388	8.153	7.871
	X (m)	0.0207	0.016	0.0616
	Y (m)	0.0109	0.0191	0.0954
	Z (m)	0.1906	0.2126	0.9571

Table 4.4: Lateral far marker-camera configuration - summary

Marker frontal and far from camera objective

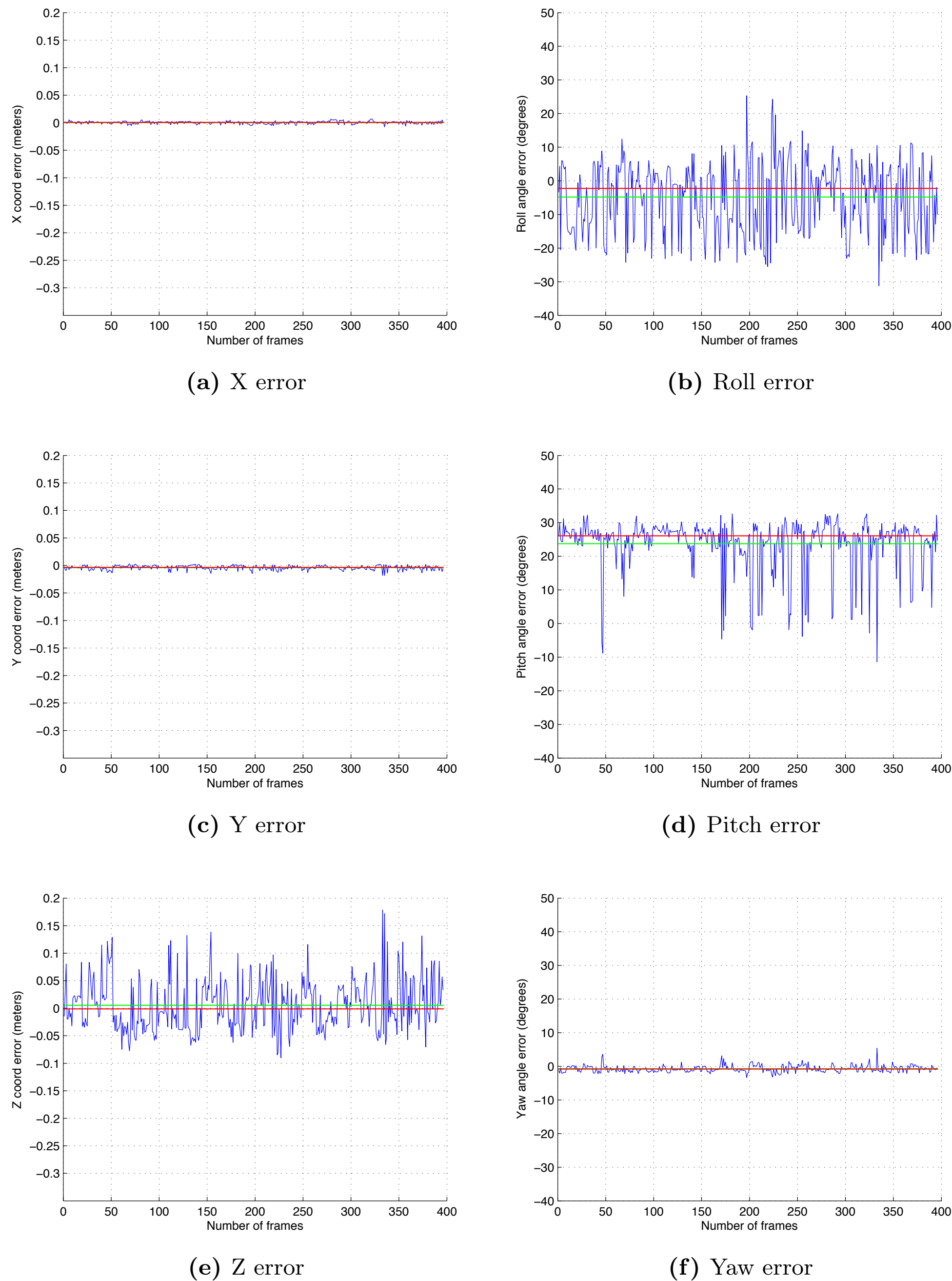
This last situation represents the worst case. It introduces errors due to the high distances and to the fact that PnP has a high error rate in pose estimation when the marker is frontal, because, as said, a little change in pixel location due to noises can cause a wrong pose estimate.

The pose errors for the considered configuration using GFTT, FAST and Canny algorithms are respectively reported in Figure 4.35, 4.36 and 4.37, and summarized in Table 4.5.

As the other configurations, errors on X and Y coordinates are very low, with average close to zero, for all the three algorithms. Also error values on Z coordinate follow the same trend as in the other configuration with the marker far from the camera. GFTT has an error with almost zero mean ranging in 20 cm, FAST slightly overestimates the marker distance of few centimeters but has an error range of about 15 cm, with some isolated higher peaks, and finally Canny highly overestimates the marker distance and presents an average error gap of about 30 cm, with some really high peaks which increase the maximum error range (these peaks go out of the reported graphics).

Unfortunately, all the algorithms do not perform very well over attitude errors. GFTT suffers of the same drawback described earlier, when the marker is frontal to the camera. Consider, for example, the pitch angle error: along the median, the error is quite stable, but the high median means that the pose was badly estimated. The high peaks are isolated, but they represent the real angle value, because they are close to zero. The same thing happens also using FAST algorithm, even though there are much more peaks, which means that the pose estimation process gets the right results more often. In general, anyway, errors on orientation angles are quite high, especially using Canny algorithm, where we can notice very big gaps.

After analysing the marker pose estimation using the three vision algorithms in all the described marker-camera configurations, we can draw some conclusions. Canny algorithm does not fit as refinement algorithm for marker corners, because it can cause very wrong pose estimates. As explained talking about this algorithm, this is due from the fact that often Canny chooses the wrong marker edges (i.e., a parallel one very close to the real edge), like the ones internal, making the marker appear smaller than it really is. The static error using this algorithm is too high and as we will see, in dynamic scenes it get worse. For this reason, till this point we have information to discard Canny algorithm from the candidate refinement processes.

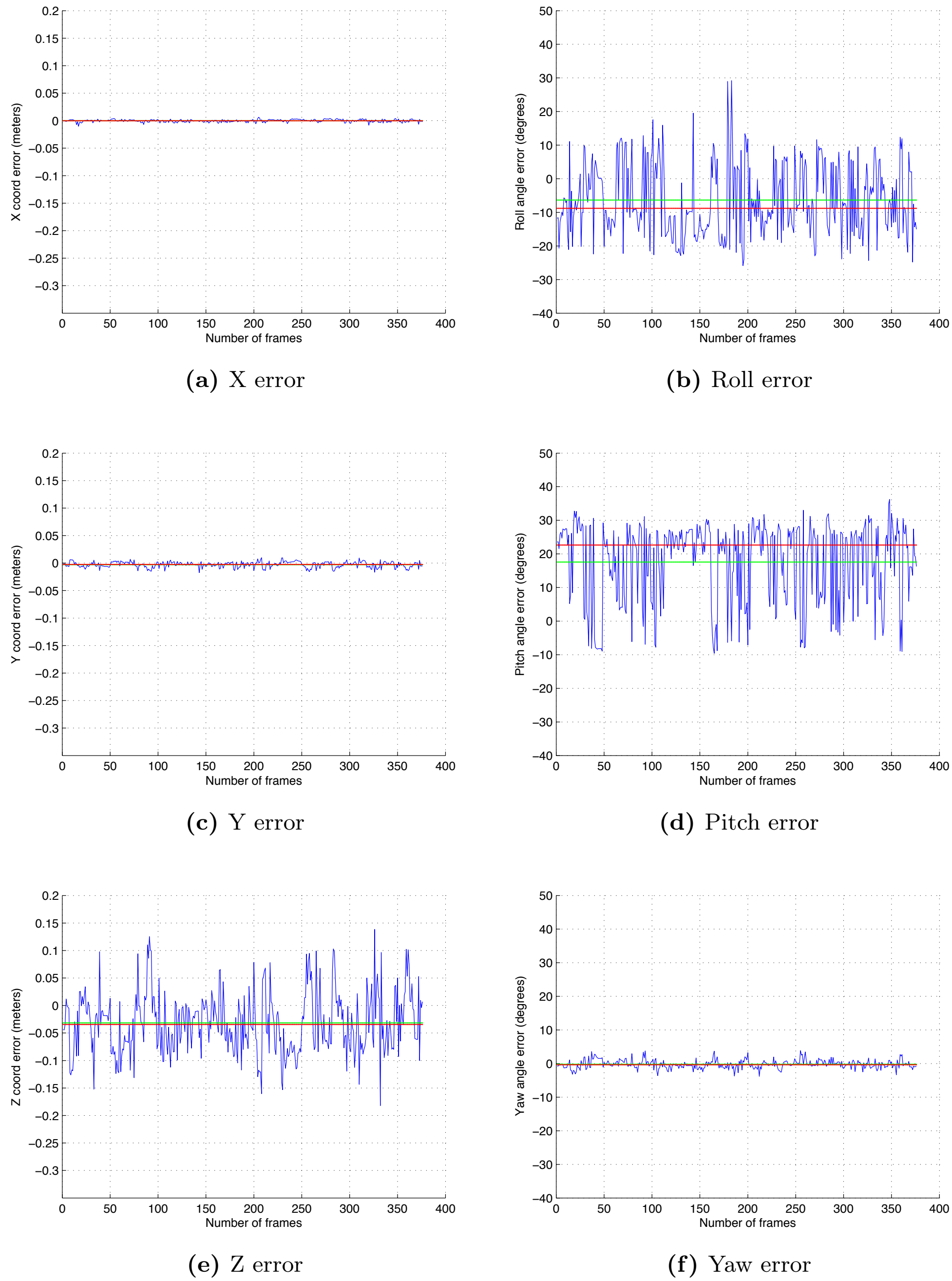


(a) X error **(b)** Roll error

(c) Y error **(d)** Pitch error

(e) Z error **(f)** Yaw error

Figure 4.35: Errors between chessboard and marker poses with mean (green) and median (red) values using GFTT method in frontal far configuration



(a) X error

(b) Roll error

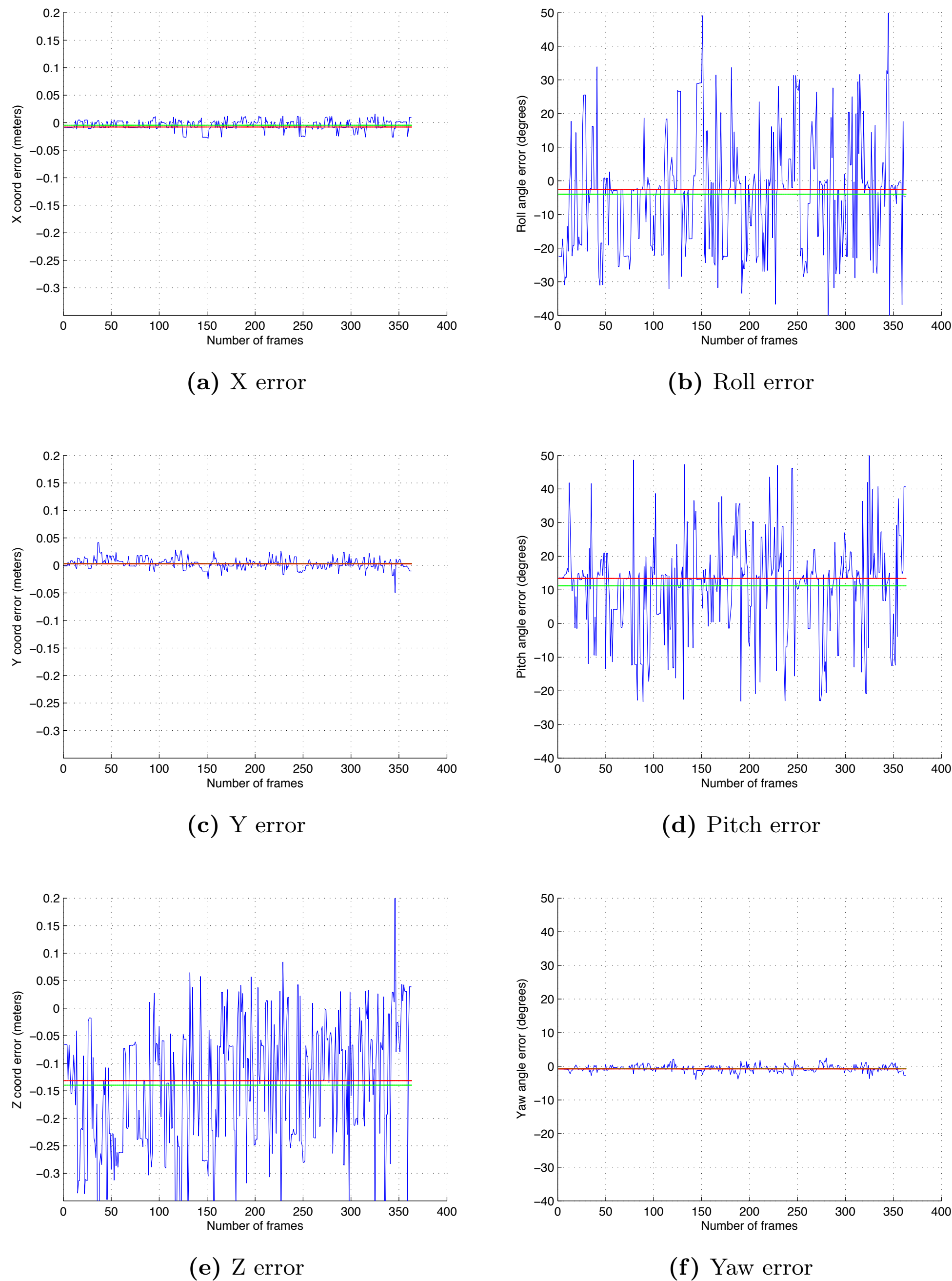
(c) Y error

(d) Pitch error

(e) Z error

(f) Yaw error

Figure 4.36: Errors between chessboard and marker poses with mean (green) and median (red) values using FAST method in frontal far configuration



(a) X error

(b) Roll error

(c) Y error

(d) Pitch error

(e) Z error

(f) Yaw error

Figure 4.37: Errors between chessboard and marker poses with mean (green) and median (red) values using Canny method in frontal far configuration

		Vision algorithm		
		GFTT	**FAST**	**Canny**
Mean value	ϕ (degrees)	-4.8348	-6.5459	-3.9904
	θ (degrees)	23.7559	16.7176	11.2258
	ψ (degrees)	-0.7239	-0.4419	-0.6449
	X (m)	0.0003	-0.0005	-0.0045
	Y (m)	-0.004	-0.0024	0.0036
	Z (m)	0.0054	-0.0468	-0.1397
Median value	ϕ (degrees)	-2.2485	-8.226	-2.563
	θ (degrees)	26.0806	20.1131	13.4123
	ψ (degrees)	-0.8245	-0.587	-0.793
	X (m)	0.0007	-0.0005	-0.0077
	Y (m)	-0.0034	-0.0022	0.0029
	Z (m)	-0.0013	-0.0545	-0.1316
Max error gap	ϕ (degrees)	56.56	52.801	99.601
	θ (degrees)	44.0586	42.6147	89.3362
	ψ (degrees)	8.734	7.379	6.372
	X (m)	0.0144	0.014	0.0445
	Y (m)	0.0212	0.0266	0.0908
	Z (m)	0.2684	0.2805	0.9062

Table 4.5: Frontal far marker-camera configuration - summary

The analysis between the other two algorithms is more complicated. We saw that they behave similarly in almost all the presented cases, with GFTT having on average lower error gaps, which confirms a better algorithm stability than FAST. This is due to the fact that using GFTT, we refine the bottom-left and top-right Data Matrix corners in accurate way, defining selected and thorough masks which help improving the overall stability. Unfortunately, the remaining marker corners are the original returned by *libdmtx*, with all the problems already widely described. This can cause wrong pose estimates in certain marker configurations (like the one presented, where the marker is turned upside down), because the points not refined are detected in a slightly wrong location, but enough to misclassify the marker pose, and since the algorithm itself is stable, the pose is continuously badly estimated. As we showed, this fact was clearly visible in the configuration with the marker close and frontal to the camera, looking at pitch rotation. Of course this cannot be considered as a general case, because markers are more often positioned in straight way within the environment, but showing a particular case can dig out the possible flaws as the one just cited. FAST algorithm instead presents higher errors over pose values, but since all the four marker corners are refined, completely wrong estimates are less common than using GFTT.

From these results, we can conclude that in a general situation in which we do not know how markers can be positioned in the world, using FAST algorithm would be more appropriate, even if less stable, because the Kalman filter manages to handle the noises, but if the pose is always estimated in the wrong way (like using GFTT), the filter does not handle it, thinking it is the correct estimate.

4.5.2 Data Matrix detection - dynamic error

In the previous section, we described and analysed the errors between marker and chessboard pose values in camera reference frame, on a static scene in different marker-camera configurations. We assumed that chessboard pose was very accurate to be chosen as the real pose in camera coordinates, having in this way a comparison method with the marker pose.

We repeated the same processes described talking about static errors also on a dynamic scene. We used the same marker-chessboard composition presented before and we moved it following a certain trajectory. In Figure 4.38 we report the trajectory we followed. While moving the marker, we took care to vary its orientation with respect to the camera, to cover all the configurations mentioned in the previous section. So we rotated marker and

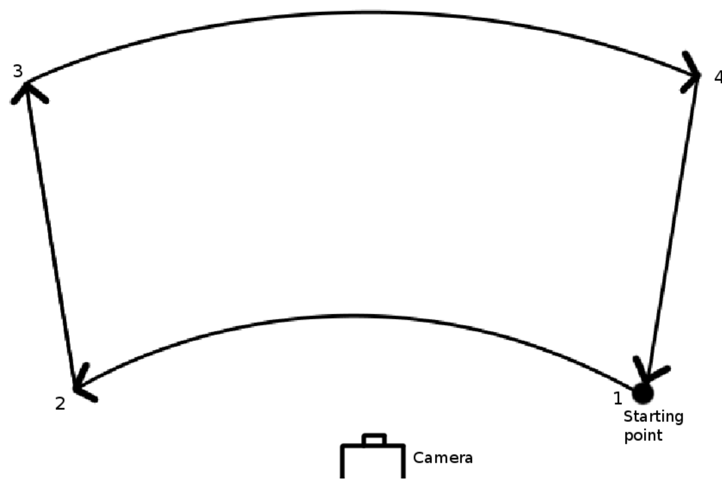


Figure 4.38: Trajectory followed in the dynamic scene

chessboard along the tangent to the shown trajectory. This means that when the marker is in the middle of the two bends, it is frontal to the camera, while when it is at the trajectory sides, it is transversal respect to the camera.

The pose errors between chessboard and marker for the considered motion trajectory using GFTT, FAST and Canny algorithms are respectively reported in Figure 4.39, 4.40 and 4.41.

In terms of frames, to better understand the next figures, the first trajectory part (from point 1 to point 2) goes (approximatively) from frame 0 to frame 200, the trajectory part from 2 to 3 goes from frame 250 to 300, then from frame 300 to frame 450 we moved the marker from 3 to 4 and finally we came back to the starting point till frame 520.

We notice that errors on Y and Z coordinates are more contained in the first trajectory part, using all the three algorithms, and they increase when the marker is far and goes along the trajectory part from 3 to 4. As we expect after the results obtained analysing the static scenes, Canny algorithm performs worse, having a higher error range, while GFTT and FAST behave similarly.

Regarding orientation errors, we see that using Canny and FAST algorithms, the rotation error values have almost zero-mean, with outliers when the marker is far from the camera (trajectory part from 3 to 4) which mean that the marker pose was wrongly estimated. Using Canny, these outliers are much more frequent than using FAST, and also their value is higher, confirming that Canny algorithm should not be considered in our applications.

GFTT is a particular case: from the figure, we see that the orientation error values are not limited to a straight bandwidth, but the are changing

over time. In fact, the marker pose is correctly estimated in the trajectory beginning (point 1), when the marker is transversal to the camera, but moving from point 1 to 2 and rotating the marker more and more frontal to the camera force the error to slowly increase, because the pose is turning wrong frame after frame. We can notice that the algorithm keeps its stability, varying the error in a very low range, but we see that the pose is more and more incorrectly estimated. In the last path, we notice many outliers, which in this specific case are the correct values, because the motion helped GFTT to vary the pixel locations and not to fall into a static wrong pose. As we said, this fact may be due to the absolute Data Matrix orientation, because *libdmtx* may calculate in wrong way the corners not refined, so the marker pose is badly estimated and GFTT stability keeps the orientation values incorrect, with correct outliers due to motion.

After these considerations, we can confirm the conclusion we drew after our static analysis. All the considered algorithms give better results (i.e., less error ranges) when the marker is closer to the camera objective. Canny algorithm for corners refinement is the worst when computing the marker pose, because it gives very noisy results over the pose variables. GFTT and FAST algorithms introduce less noise, indeed the error is mostly contained in a low range, with exception of some outliers resulting from completely wrong pose estimates. As already said, although GFTT works well if we consider its stability, even with occasional mistakes in pose calculation, in some marker configurations it can happen that the pose estimate is continuously computed in wrong way, providing the Kalman filter incorrect information while processing the SLAM algorithm.

For these reasons, in general FAST algorithm should be employed, because even though it does not provide as much stability as GFTT, it helps to compute the right marker pose mostly of times. The error swings would be partly corrected thanks to the Kalman filter during SLAM algorithm while building the environment map, anyway.

Finally, in Figure 4.42 we plotted the resulting trajectories of both marker (in red) and chessboard (in blue) for all the three vision algorithms, again after translating marker coordinates system to overlap with the chessboard pose, computing Equation 4.18. The trajectories are obviously almost identical in average, with the marker having a less fluid and more noisy motion respect to the chessboard. The highest errors were retrieved on the distance along camera z-axis. We clearly see that Canny algorithm introduces a huge amount of error, following the right trajectory, but with too many

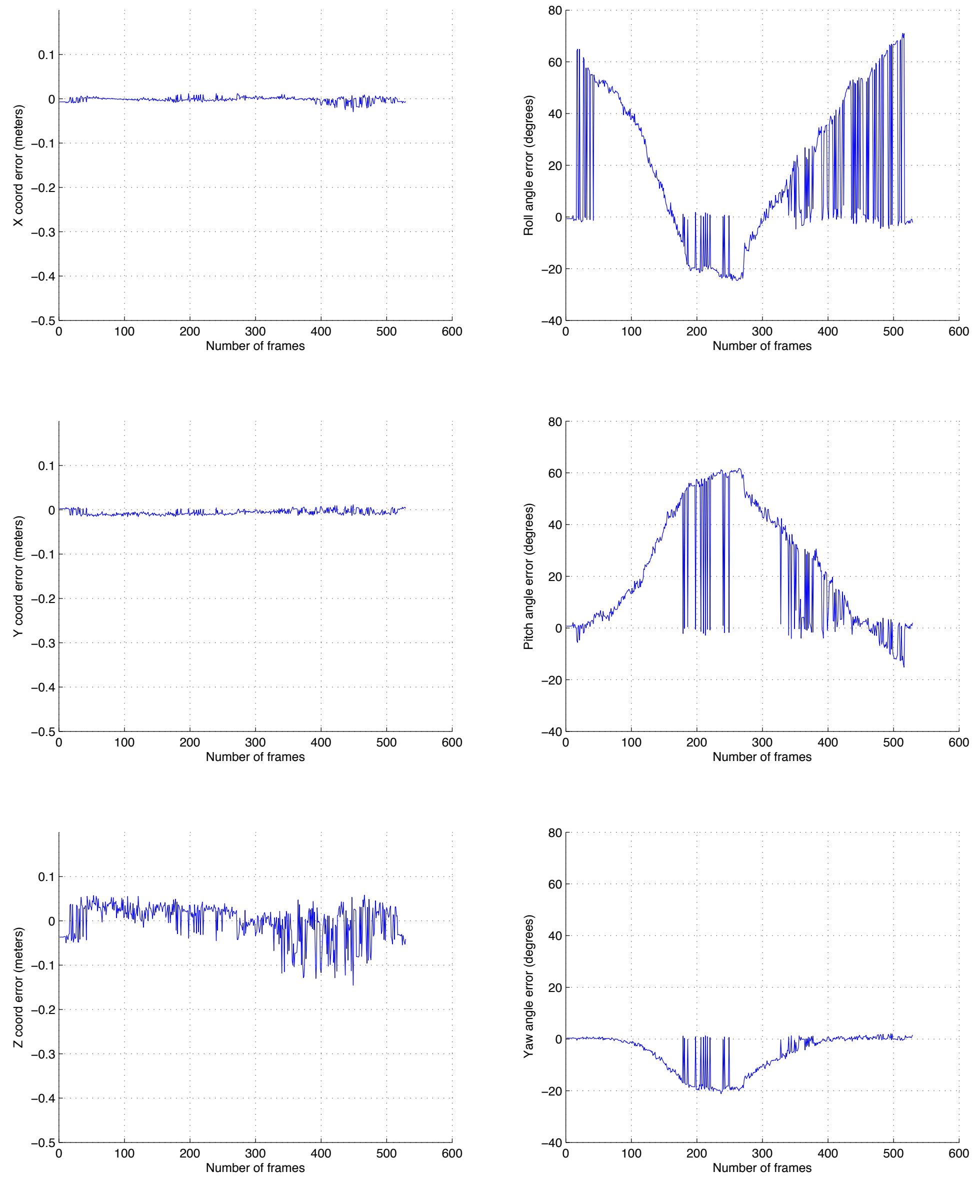


Figure 4.39: Errors between chessboard and marker poses using GFTT method in dynamic scene

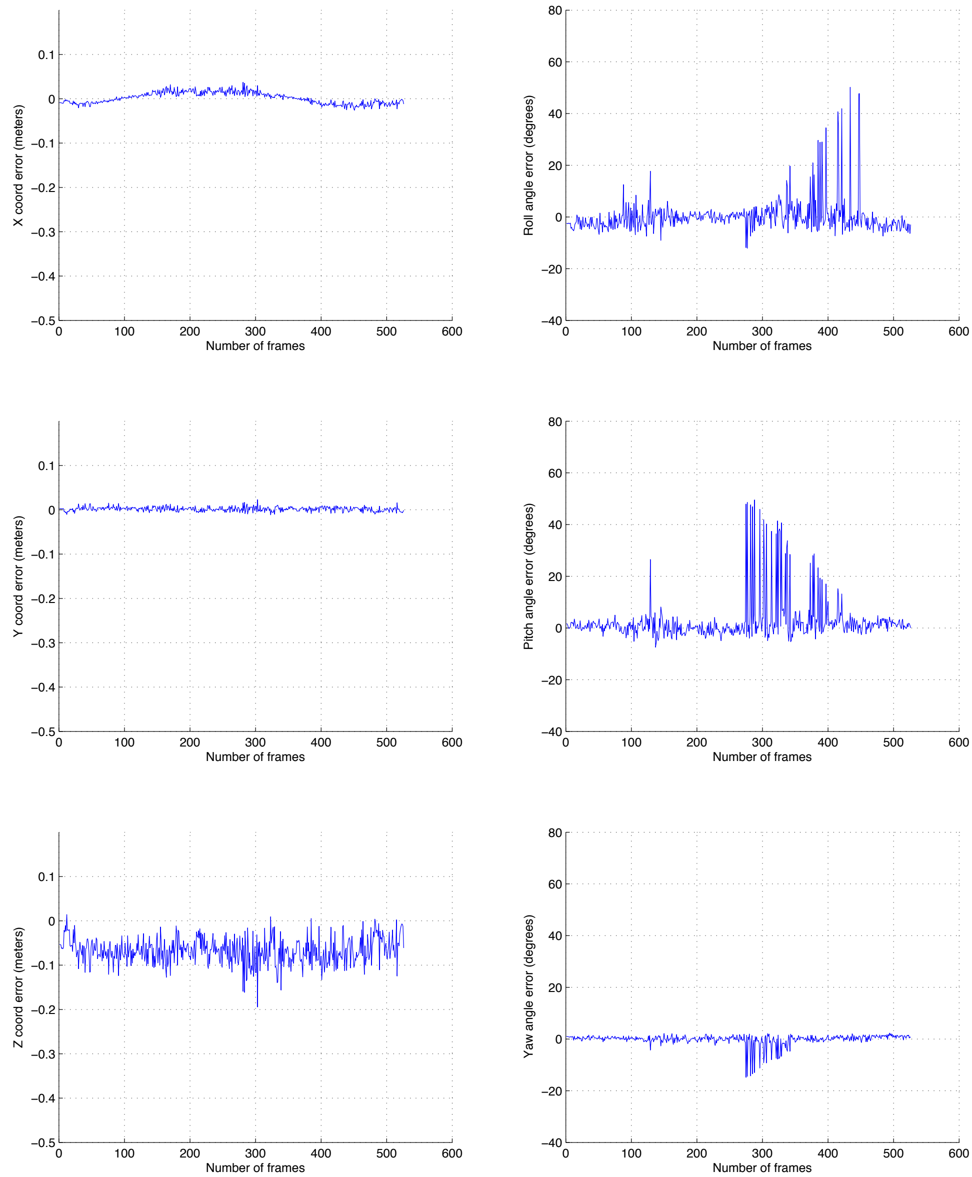


Figure 4.40: Errors between chessboard and marker poses using FAST method in dynamic scene

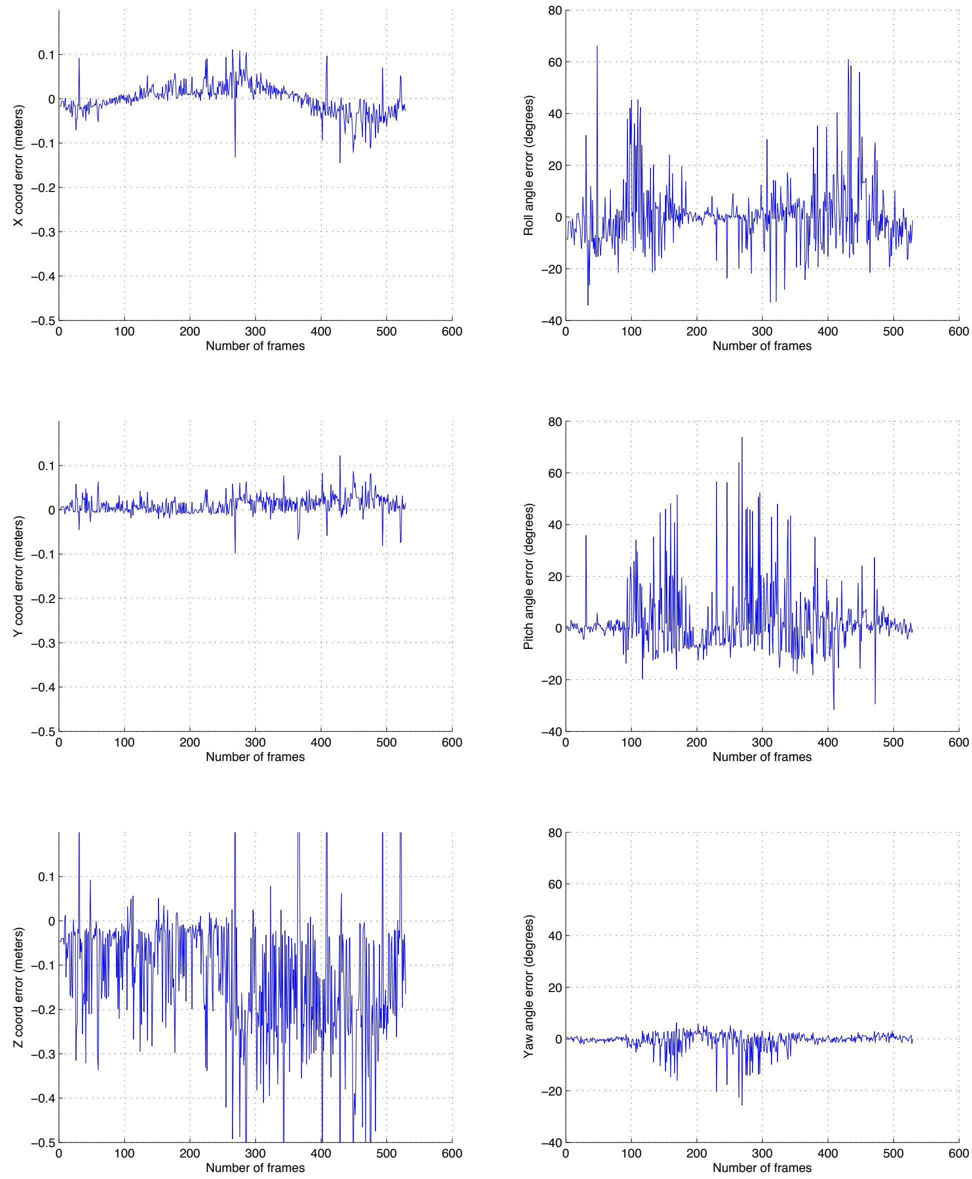
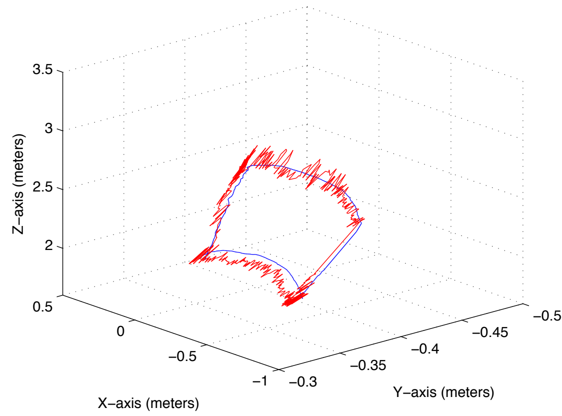
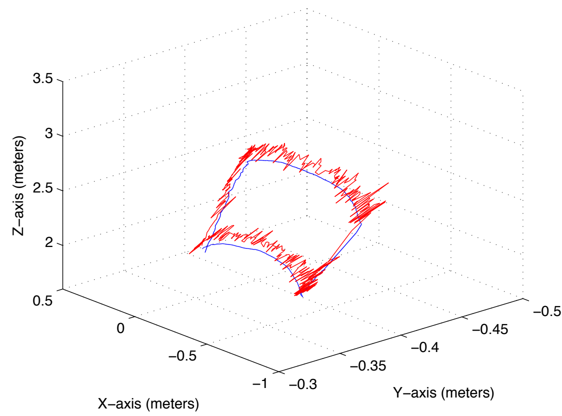


Figure 4.41: Errors between chessboard and marker poses using Canny method in dynamic scene

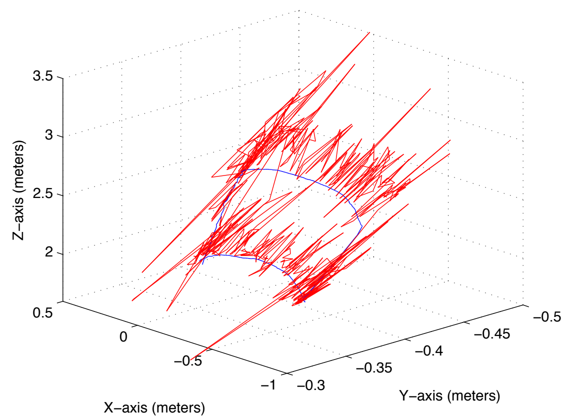
fluctuations, especially over the distance from the camera. FAST and GFTT algorithms, again, give similar results, with a clear trajectory affected by unavoidable errors, so from the point of view of the position errors only, the choice between the two algorithms could be the same. In the figure we do not see the orientation errors (even if we provided the correspondent plots previously), but we already said that FAST algorithm generally is to prefer from this point of view.



(a) GFTT method



(b) FAST method



(c) Canny method

Figure 4.42: Chessboard (blue) and marker (red) trajectories respectively using GFTT, FAST and Canny methods

Chapter 5

AIRVisualMarker library for MoonSLAM

"I am not discouraged, because every wrong attempt discarded is another step forward."

Thomas A. Edison

In this chapter we describe how we integrated our library for detecting visual markers, called AIRVisualMarker, in a EKF-SLAM framework, developed at Artificial Intelligence & Robotics laboratory of Politecnico di Milano. First, we briefly introduce such framework and we talk about the library main structure; then we describe the integration between these two. Finally, we show our experiments and analyse in details most relevant results.

5.1 MoonSLAM

MoonSLAM[1] is a project started on June, 2010, at the Artificial Intelligence & Robotics laboratory [2] (AIRLab) at Politecnico di Milano, with the purpose to create a robust and efficient tool which allows to design, experiment and develop techniques for mobile robots simultaneous localization and mapping (SLAM) in a unique complete framework.

5.1.1 Main features

In the scientific community, there is plenty of projects built to realize SLAM applications; however, many of them are developed to solve specific sub-

[1] http://airlab.elet.polimi.it/index.php/MoonSlam

[2] http://airlab.elet.polimi.it/index.php/AIRWiki

problems or to implement specific algorithms, making use of development environments oriented to easy computation, at the expense of efficiency and real-time performances.

MoonSLAM was created in order to satisfy different characteristics:

- **efficiency**: MoonSLAM is entirely written in C++, this guarantees a better hardware resources management, implementation of more performing algorithms, a better integration with third-party libraries and the possibility to interface the framework with other computation tools (such as MATLAB[3] and Octave[4]);

- **flexibility**: a feature that distinguishes MoonSLAM from other frameworks is the possibility to implement SLAM algorithms without having to specify constrains about the motion model type, the sensor number and type employed, the data acquisition channels, the data association method used and the type of landmark used inside the filter;

- **extensibility**: the modular structure employed in MoonSLAM allows to extend the framework with new functionalities, without having to modify the old ones. Moreover, within a single module it is possible to extend existing classes to add new objects, e.g., feature parametrizations or new motion and camera models;

- **configurability**: another feature of MoonSLAM is represented by the possibility to build SLAM software using three different implementation ways: *coded, configurated* and *hybrid*.

  In coded version, the program is not fed with external configurations, every object is created within the code, which is then compiled and executed: this implementation is rigid, though, and does not allow run-time configuration.

  Configurated implementation, instead, allows the specification of objects and actions to execute through an external configuration file. The program only reads what is reported in such file and executes its instructions. As drawback, the high configurability level increases the configuration file scripting in a significant way.

  Hybrid method is a mix of the two just mentioned: it keeps the biggest part of the algorithm inside the code, but it uses a configuration file to choose the objects to use, joining in this way both configurability and low complexity advantages.

[3] http://www.mathworks.com/products/matlab/index.html

[4] http://www.gnu.org/software/octave/

MoonSLAM can be used for implementing a variety of SLAM algorithms and other useful functionalities:

- monocular, stereo and omnidiretional SLAM
- classic or sub-map based EKF-SLAM
- online SLAM, based on real dataset, and SLAM in simulated environments
- 2D and 3D image processing
- 2D and 3D geometric computations
- Bayesian inference, Kalman filters creation and multivariate Gaussian probability density management
- Bundle Adjustment and Bundle Adjusted SLAM

Thanks to the framework extensibility, it is possible to combine two or more of the cited functionalities to create new ones.

5.1.2 External libraries

MoonSLAM framework integrates third-party GNU GLP3 libraries inside its structure (see Figure 5.1). Between them, the main ones are:

- *OpenCV* libraries[5]: used for 2D image visualization, camera frames acquisition, image features identification and the most common image processing functions.
- *mrpt-gui*, from Mobile Robot Programming Toolkit library[6]: used to create 3D scenes and environment visualizations.
- *Eigen* library[7]: used for matrix computation and linear algebra operations.
- *LibConfig* library[8]: used for configuration files acquisition and decoding.

[5]`http://opencv.willowgarage.com/wiki/`
[6]`http://www.mrpt.org/`
[7]`http://eigen.tuxfamily.org/`
[8]`http://www.hyperrealm.com/libconfig/`

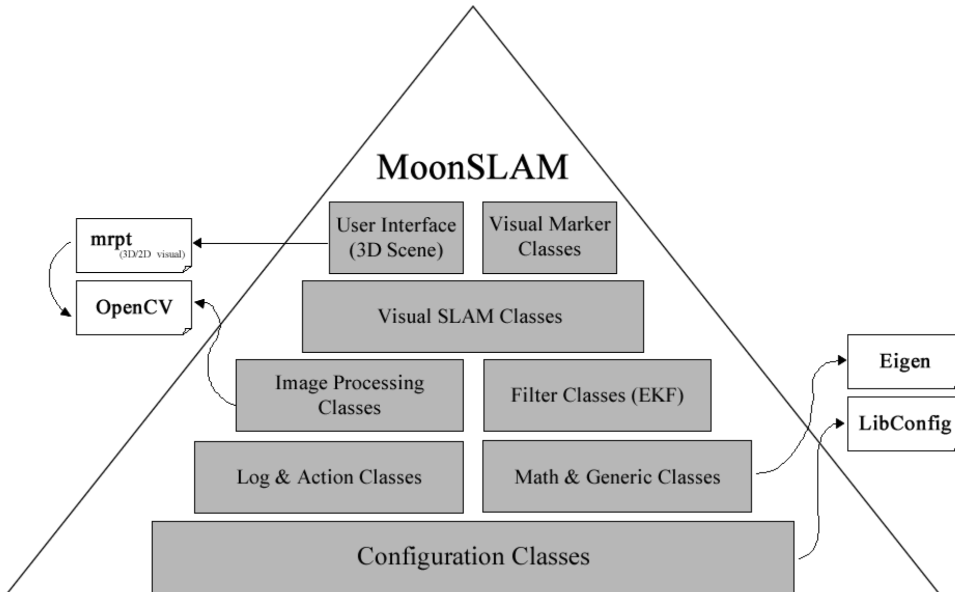


Figure 5.1: MoonSLAM main components and relations with third-party libraries

5.1.3 Framework architecture

MoonSLAM is composed by a collection of classes, which are grouped libraries according to their main functionalities. The main libraries belonging to MoonSLAM and our AIRVisualMarker library are shown in Figure 5.2 with a UML package diagram, explicating dependency relations between them (with oriented arcs). We present now an overview of these already existing libraries, describing their most relevant classes and functionalities.

AIREkfSlam

AIREkfSlam library is one of the main elements of MoonSLAM framework: it contains the main classes needed to build an extended Kalman filter, to handle variables inside the filter itself and to execute the EKF-SLAM prediction and update phases.

The primary class around which the whole library is built around is called *Ekf*: it provides all the functionalities needed to create and manage an extended Kalman filter. The class logic structure is composed by three parts: one containing the dynamic elements (*dynamics*), one containing the static elements (*parameters*) and one containing the observed features (*landmarks*).

The “dynamics” part includes all the variables which estimate the robot physical state, like its position, its attitude and its linear and angular velocity.

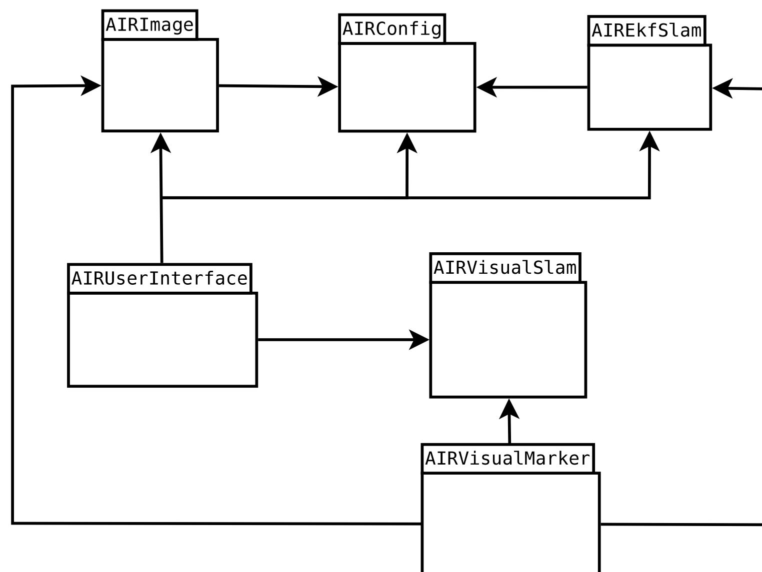


Figure 5.2: MoonSLAM main modules graph and dependencies between them

The “parameters” parts covers all the elements assuming constant values, like camera calibration variables or other values which are not modified during the program execution. All the variables relative to the features observed within the environment are included inside the “landmarks” part.

All the filter elements are represented with random multivariate gaussian variables, whose means and covariances are described by *vars_means* and *vars_covs* variables inside Ekf class (Fig. 5.3). It is important to notice that every element in the filter is an instance of the class *Variables*, which provides methods to access variable mean and covariance: by considering every element as a generic variable makes it possible to include in the filter heterogeneous elements, like landmarks, velocities and positions, and to add to the filter landmarks with different parametrizations or even with different nature (like visual markers). The filter handles in this way only variables means and covariances values, without considering their nature.

Furthermore, Ekf class provides two methods which implements the Kalman filter prediction phase, in which it estimates the robot next state, and the update phase, in which the information about new measurements are used to update the robot state and the landmarks.

In Figure 5.3, the library main components, their structure and their variables are shown.

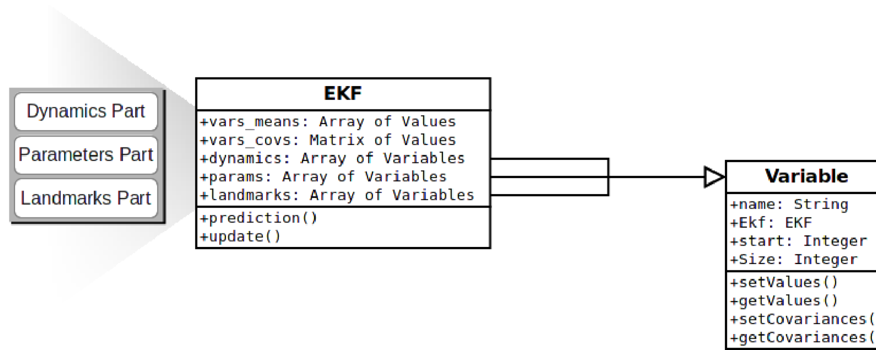


Figure 5.3: AIREkfSlam library main components and variables

AIRImage

AIRImage library is composed by the classes needed to acquire and process datasets for visual SLAM, to identify features within the image and to handle landmarks and their 2D representation.

As explained for AIREkfSlam library, a landmark is just a generic "Variable" inside the Extended Kalman Filter, like robot velocity and position. Unlike these two elements, though, a landmark is an object having also graphical properties and, in case of visual SLAM, is related to a feature in an image through a measurement equation. A landmark is therefore built by two parts, the first one composed by the multidimensional variable whose mean and covariance values are inserted into the Kalman filter, the second one composed by an object containing all the visual information regarding the landmark itself, and stored in a special *LandmarksContainer*. AIRImage library also contains interfaces and classes which allow to build visual landmarks and describe their structure.

Landmarks are managed within the library thanks to special base classes, which allow to create and add a landmark to the filter (*LandmarksCreator* and *LandmarksAdder*), to match a landmark in an image (*LandmarksMatcher*), to delete a landmark from the filter (*LandmarksDeleter*) and to handle the array containing all landmarks (*LandmarksContainer*).

AIRImage library is also composed by classes related to camera models (omnidirectional and prospective), to landmark 2D visualization (*DrawerManager*) and to landmark descriptors management (*DescriptorInterface*).

AIRVisualSlam

AIRVisualSlam library extends many classes from the AIRImage library, giving them a concrete implementation, and contains all the elements needed

to implement image-based SLAM algorithms. The library is composed in the following way:

- a group of classes used to add new landmarks to the filter and another one used to create landmarks measurements; among the supported parametrizations we can cite IS [27] (Inverse Scaling), ISA (Inverse Scaling Anchored), ISDA (Inverse Scaling Double Anchored), UID [6] (Unified Inverse Depth), AHP [37] (Anchored Homogenous Point), FHP [5] (Framed Homogenous Point), 3D (3D Euclidean Point) and FID (Framed Inverse Depth);

- a group of classes *sharedParametrization*, used to adopt the shared version (which means that if two landmarks contain the same information, this is shared) of UID and FHP parametrizations;

- a collection of *MotionModels*, with or without information about robot odometry;

- a group of classes and interfaces (*MapManagement3dPoints* and *MapCreatorInterface*) used to realize and handle 3D maps in a simulated environment.

Moreover, there is a set of support classes, useful for quaternions management, operations between matrices, gaussian variables rototranslations and GPS measurements integrations.

AIRUserInterface

AIRUserInterface library contains all the classes used for 3D scenes representation, scene navigation, landmarks, robot and 3D trajectory visualization and viewport creation of course. This library is a *wrapper* of the external library mrpt-gui.

AIRConfig

AIRConfig library contains the classes needed to decode a given configuration file and consequently create and handle global variables and the required objects. New objects creation process, given a configuration file, is reported in Figure 5.4.

A generic configuration file is divided into three sections: *Global* section contains the file global variables (not code variables), *Create* section contains a list of objects to create together with their initialization parameters, and

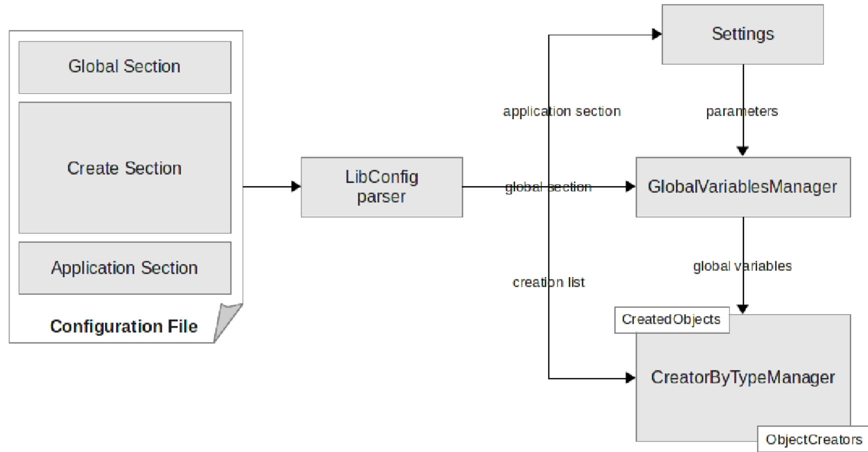


Figure 5.4: Objects creations from external configuration file

the *Application* section which contains a set of parameters used inside the program code.

LibConfig library takes care to read and parse the configuration file, while a special object, called *GlobalVariablesManager*, receives the global variables list and the application parameters and sends them to the *CreatorByTypeManager*, which creates the objects according to what specified from the create section.

5.1.4 EKF-SLAM in MoonSLAM

Here we describe briefly how MoonSLAM implements a classic EKF-SLAM algorithm. The operation flow is shown in Figure 5.5.

1. An *ImageManagerInterface* object (from AIRImage library), given a camera image list as input, returns a pointer to the next frame to process;

2. a *DynamicPredictorBase* (from AIREkfSlam library) computes SLAM prediction step. In particular, the predictor, according to the system motion model, calculates the robot's next state, together with position and velocity uncertainties, and updates the related means and covariances in the filter;

3. a *LandmarksAnalyzer* object belonging to the *LandmarksContainer*, according to the prediction step results, predicts old landmarks position within the map;

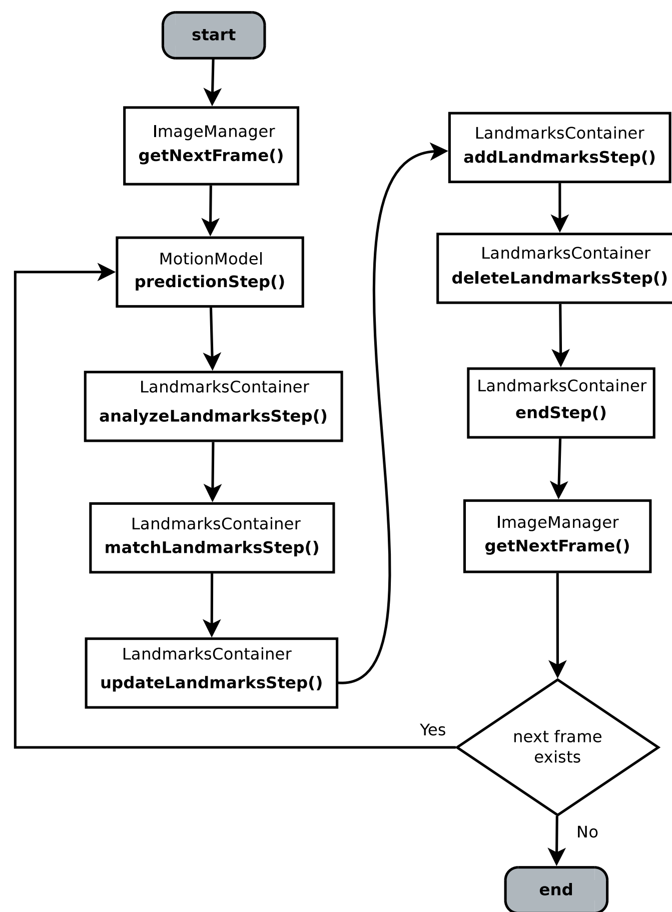


Figure 5.5: Main steps of an EKF-SLAM algorithm in the MoonSLAM framework

4. the *LandmarksContainer*, thanks to a *LandmarksMatcher*, searches for associations between old landmarks and new detected features in the current frame. This process is repeated for each camera available (i.e., one for mono SLAM and two for stereo SLAM);

5. a *LandmarksUpdater* object updates the landmarks positions for which a match was found in the current frame;

6. a *LandmarksAdder* and a *LandmarksCreator* add to the landmarks container (and to the extended Kalman filter) occasional new landmarks detected in the current frame;

7. a *LandmarksDeleter* removes from the container and from the filter potential unnecessary landmarks (i.e., the ones having negative depth

in UID parametrization);

8. finally, map landmarks information related to execution steps are updated.

If a frame is available after the current one, the process starts again from prediction step, otherwise the algorithm terminates.

5.2 AIRVisualMarker library

The main purpose of this thesis was to build a library for visual markers detection to be integrated into the MoonSLAM framework, using methods and information described in previous chapters. We give an overview of the resulting library general structure, describing the principal elements and the dependencies between them; then we talk about markers integration inside the Kalman filter.

5.2.1 Library structure

In Figure 5.6 we reported the library classes and their dependencies. Here is a list with their description:

- *Marker* class: it is the class which represents a marker object. It works as an interface, since it can implements different visual markers, like the ones cited in Chapter 3. In our work, we employed only Data Matrix barcodes, but support for other 2D markers can be easily added. It has a few necessary attributes: *imgPoints* is a vector of 2D float image points, which are the selected marker pixel points (at least four) in the camera frame to compute a Perspective N-Point operation; *modelPoints* is the correspondent vector of 3D float model points; *ID* is a string which identifies in unique way a detected marker; *dim* is a double variable containing the marker 3D size in meters (if the marker allows to encode a large number of characters, being as a Data Matrix, it is possible to store the barcode's dimension inside itself, saving then the correspondent value in the *encodedDim* attribute); two vectors containing the marker pose in camera coordinates system (*frameWRTCam*) and in world coordinates system (*frameWRTWorld*) through attitude quaternions representation; a *MeasureMarkerImagePoints* object (described later) used to compute measurements of the marker image points (i.e., it implements the Kalman filter measurement equations for visual markers). In this class there's also a method

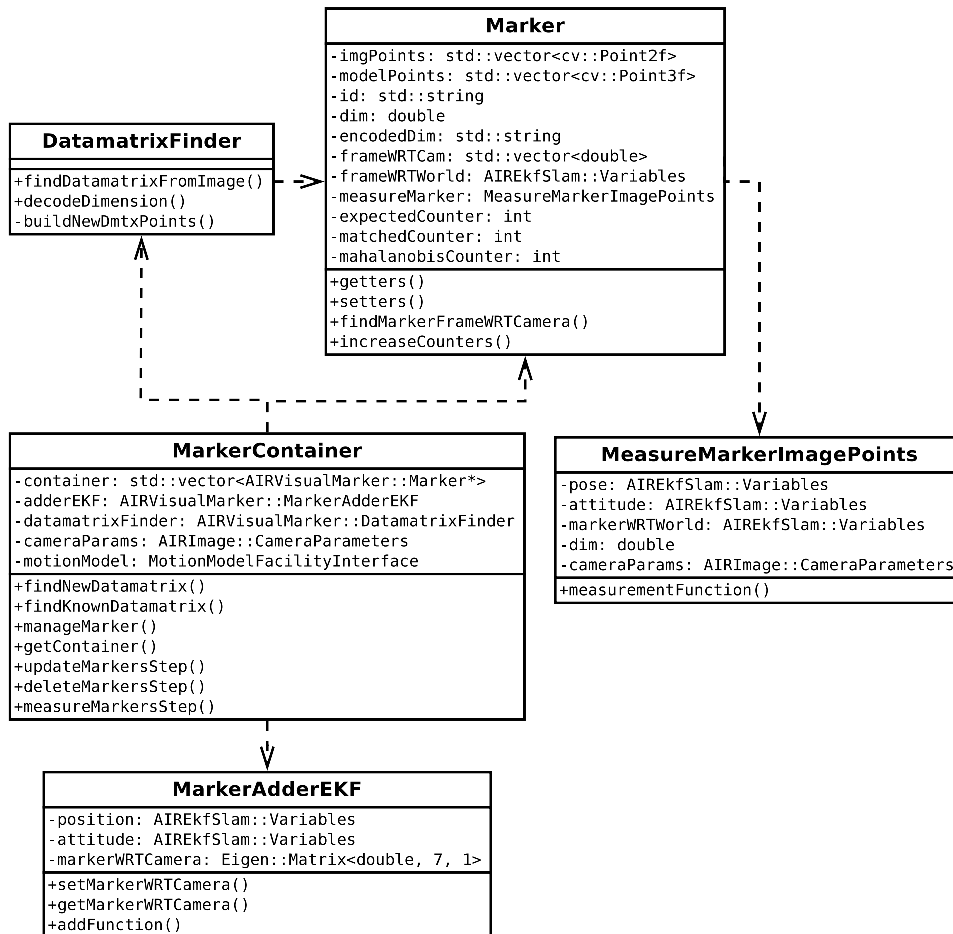


Figure 5.6: Dependencies between classes within AIRVisualMarker library

(*findMarkerFrameWRTCamera*) to compute the marker pose in camera coordinates system, as described in Chapter 5.

- *DatamatrixFinder* class: it contains the specific methods to retrieve, decode (id and dimension size) and refine a Data Matrix barcode, given any camera image, according to the *libdmtx* library and the computer vision techniques explained in details in Chapter 4. In order to add support for other marker typologies, it is necessary to create a similar class where to write specific code to recognize the new marker type, with its specific library.

- *MarkerAdderEKF* class: it is the class that, given the camera pose in world system and the marker pose in camera system, adds a new marker coordinate system with respect to the world frame (i.e., a vec-

tor containing the marker position and attitude) to the Kalman filter, treating it as a generic *Variables* object (this confirms MoonSLAM flexibility in handling heterogeneous objects inside the filter). Please notice that we need to know where the camera is positioned in the world, inferring it from the camera motion model (i.e., from odometry information).

- *MeasureMarkerImagePoints* class: it contains attributes and methods to measure where the marker image points are likely to be located within an image and at which depth from the camera (filter measurement step) after we forecast the camera trajectory according to odometry information from the chosen motion model (filter prediction step). Attributes regarding the camera and marker pose in world coordinates system, marker size and camera intrinsic parameters are needed.

- *MarkerContainer* class: this is the main class in AIRVisualMarker library. Like the *LandmarksContainer* object in MoonSLAM, *MarkerContainer* deals with markers management. It takes care of detecting a marker from a camera image, through *DatamatrixFinder* class or similar, store a pointer to it in a special vector, add it to the Kalman filter (and consequently to the LandmarksContainer) through *AdderMarkerEKF* class, delete it in case of misclassification and handle its image points and depth measurements through *MeasureMarkerImagePoints* class.

In the following sections we describe in more details the most important marker container functions.

5.2.2 Marker detection

As we said, the pure marker detection system is handled by an object from *DatamatrixFinder* class (in case of Data Matrix barcodes) employed in *MarkerContainer*, using the techniques already reported in previous chapters. Specifically, *MarkerContainer* class implements two versions of the marker search: the first one is used to scan the whole camera image in order to find every new occasional marker inside it, exploiting a high searching time-out to lower the probability to miss some Data Matrices. The second version searches for a single marker in the image zone where such marker is expected to be, after the filter predicts and measures its image points on the image frame: in this way it is possible to speed-up the whole process setting a very low time-out for every marker, since *libdmtx* has to scan a very restricted area with respect to the entire image size.

Using this choice, it is possible to plan a program execution calling the first function for marker detection once every K image frames (where K is up to the programmer), to find all the Data Matrix barcodes in the image (and add the new ones to the filter) or to correct the position of a marker within the image in case of wrong prediction, and calling the second version of the scan function on all the frames, to detect in a fast way markers already predicted (filter match step).

Every time a marker is recognized, a new Marker object is created, which stores in its attributes all the information about the barcode: id, dimension size, image points and pose in camera coordinates system.

5.2.3 Adding markers to the filter

The first time a Data Matrix is detected, *MarkerContainer* class takes care of storing a pointer to its object structure inside a container vector, which keeps track of every detected marker. Similarly, it must be added also to the Kalman filter as landmark, so to the *LandmarksContainer*: adding a marker to the filter means that we store inside it a 7-value vector, representing the marker position (first three elements) and attitude (last four elements, in quaternion representation) of the marker in world reference frame.

In order to do such operation we need two variables in the state vector: the camera position in world reference frame, represented with three values as $\mathbf{T}_C^W$, and the camera attitude in world reference frame, represented with four quaternion values, but shown in the following after a transformation to rotation matrix $\mathbf{R}_C^W(att_c)$, easier to understand. These quantities are available from the camera (or robot) motion model, which can be based on physical or visual odometry.

We can obtain the marker pose in world coordinates system ($\mathbf{RT}_M^W$) by composing the camera pose in the world frame with the marker pose in the camera frame, computed by solving the PnP problem and shown as a translation $\mathbf{T}_M^C$ and a rotation matrix $\mathbf{R}_M^C(att_m)$ (for the same reason as before).

The resulting equation is:

$$\begin{aligned}\mathbf{RT}_M^W &= \begin{bmatrix}\mathbf{R}_C^W(att_c) & \mathbf{T}_C^W \\ 0 & 1\end{bmatrix}\begin{bmatrix}\mathbf{R}_M^C(att_m) & \mathbf{T}_M^C \\ 0 & 1\end{bmatrix} = \\ &= \begin{bmatrix}\mathbf{R}_C^W(att_c)\mathbf{R}_M^C(att_m) & \mathbf{R}_C^W(att_c)\mathbf{T}_M^C + \mathbf{T}_C^W \\ 0 & 1\end{bmatrix}\end{aligned} \tag{5.1}$$

So the new marker attitude is given by $\mathbf{R}_C^W(att_c)\mathbf{R}_M^C(att_m)$, after another

transformation from rotation matrix to quaternion representation, while the new marker position is given by the 3-value vector $\mathbf{R}_C^W(att_c)\mathbf{T}_M^C + \mathbf{T}_C^W$.

The filter needs Jacobians of the new marker pose with respect to the state variables (camera position and attitude in world reference frame) and also with respect to the input variables (marker pose in camera reference frame).

5.2.4 Marker image points measurement

Every time the marker container adds a new maker to the filter as landmark, a measurement object (described earlier) for such marker is created: in case of Data Matrix barcodes, this means that we can use this object to measure where the four corners will lay on the next image frame after we have predicted where the camera has moved thanks to motion model information. In other words, we want to find the pose of each image point on the image plane in camera reference frame, assuming we know the camera pose in world coordinates, the marker pose in world coordinates and the selected model points.

As we described in Equation 4.14, each of the four Data Matrix 3D points is represented as $\mathbf{P}_i = [X, Y, 0]^T$, where X and Y are either equal to zero or to the marker edge dimension, resulting in the four marker corners. In order to represent these points in camera reference frame, as specified in Equation 4.13, we need to compose the camera and the marker pose in world coordinates, obtaining the marker pose in camera frame, and then apply the resulting rototranslation to each model point. We dispose of the following quantities:

- the camera attitude in world reference frame, from the motion model and represented as a quaternion. For simpler calculations, we represents such quantity in terms of rotation matrix, indicated as $\mathbf{R}_C^W(att_c)$
- the camera position in world reference frame, represented as a 3-value translation vector $\mathbf{T}_C^W$
- the marker attitude in world reference frame represented as a quaternion $\mathbf{q}_m^w = [w_m, x_m, y_m, z_m]^T$. This quantity must be normalized by the factor $\sqrt{w_m^2 + x_m^2 + y_m^2 + z_m^2}$. As before, we use such quantity in terms of rotation matrix, indicated as $\mathbf{R}_M^W(att_m)$
- the marker position in world reference frame, represented as a 3-value translation vector $\mathbf{T}_M^W$
- the four marker model points $\mathbf{P}_i = [X, Y, 0]^T$

- the matrix of camera intrinsic parameters A and the camera distortion coefficients $k1, k2$

Each marker point in camera coordinates system is then calculated composing the inverse rototranslation of the camera in world coordinates with the rototranslation of the marker in the camera coordinates and multiplying the result for each model point, as it follows:

$$
\begin{aligned}
\mathbf{t}_{\mathbf{P}_i} = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} &= \begin{bmatrix} \mathbf{R}_C^W(att_c)^T & -\mathbf{R}_C^W(att_c)^T\mathbf{T}_C^W \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_M^C(att_m) & \mathbf{T}_M^C \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}_i \\ 1 \end{bmatrix} = \\
&= \begin{bmatrix} \mathbf{R}_C^W(att_c)^T\mathbf{R}_M^C(att_m) & \mathbf{R}_C^W(att_c)^T\mathbf{T}_M^C - \mathbf{R}_C^W(att_c)^T\mathbf{T}_C^W \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}_i \\ 1 \end{bmatrix} = \\
&= \mathbf{R}_C^W(att_c)^T\mathbf{R}_M^C(att_m) \begin{bmatrix} \mathbf{P}_i \\ 1 \end{bmatrix} + \mathbf{R}_C^W(att_c)^T\mathbf{T}_M^C - \mathbf{R}_C^W(att_c)^T\mathbf{T}_C^W = \\
&= \mathbf{R}_C^W(att_c)^T\{\mathbf{R}_M^C(att_m) \begin{bmatrix} \mathbf{P}_i \\ 1 \end{bmatrix} + \mathbf{T}_M^C - \mathbf{T}_C^W\}
\end{aligned} \tag{5.2}
$$

Now, first it is necessary to project the resulting points $\mathbf{t}_{\mathbf{P}_i} = [x_i, y_i, z_i]^T$ to the image plane π, in camera frame, dividing such vector by the z_i component: we obtain the four projected points

$$
\mathbf{p}_{i\pi}{}^C = \begin{bmatrix} x_i/z_i \\ y_i/z_i \\ 1 \end{bmatrix}. \tag{5.3}
$$

Secondly, we need to apply a distortion to the resulting points (through a specific *dist* function, based on the distortion model, and the distortion parameters) and compute a reprojection with the matrix of camera intrinsic parameters A. The final measured 2D marker points on the image plane are then represented by:

$$
\mathbf{p}_i = \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \mathbf{A} \times dist(\mathbf{p}_{i\pi}{}^C, k1, k2). \tag{5.4}
$$

In case, to obtain a more precise measure, we can add to the described measure (the four image points) also the point distances from the camera, which we retrieve calculating the norm of the vector $\mathbf{t}_{\mathbf{P}_i}$, for each of the four points.

$$
d_i = \sqrt{x_i^2 + y_i^2 + z_i^2}. \tag{5.5}
$$

The Kalman filter needs the Jacobians of the new measured points vector with respect to the variables in the state vector, i.e., camera pose and attitude in world reference frame, and with respect to the input variable, i.e., the measurement vector just obtained.

5.2.5 Deleting a marker from the container

MarkerContainer class is also responsible for deleting markers from the container (and from the filter). This operation is due whenever *libdmtx* library does not properly decode a Data Matrix, letting the container to add a potentially wrong marker to the filter (with a wrong id or a wrong encoded dimension, for instance). In this case, such marker should be deleted from the container in order to avoid wrong map building estimates.

Marker class has two extra attributes which keep trace of how many times we expect to see a known marker in the image (*expectedCounter*) after a measurement, and how many time we effectively see the predicted marker (*matchedCounter*). For every image frame, the container checks whether the fraction of number of matched markers over the number of expected markers is lower than a certain threshold. This is equivalent to say that

$$\frac{matchedCounter + K}{expectedCounter + K} < X \tag{5.6}$$

where K is a smoothing parameter adequately chosen in order to avoid improper marker deletions, i.e, when a marker is expected but not detected by *libdmtx* because of image blurring and not because it was previously misclassified, while X is the selected threshold ($0 < X \leq 1$).

Another policy for marker deletion is due to the fact that it can happen that a marker is added to the filter with a wrong pose, because of the explained problems in marker recognition for certain marker configurations, especially using GFTT algorithm to improve the marker corners precision. For this reason, we decided to accept only marker measures under a certain threshold error from the real measures. Since our measures are defined over a multivariate Gaussian distribution, we took into consideration Mahalanobis distance [26] between pixels and depths measured by the filter and ones detected by our library at every frame. The distance of a multivariate vector $\mathbf{x} = [x_1, x_2, \ldots, x_N]^T$ from a group of values with mean $\boldsymbol{\mu} = [\mu_1, \mu_2, \ldots, \mu_N]^T$ and covariance matris $\mathbf{S}$ is defined as:

$$D_M(x) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \mathbf{S}^{-1} (\mathbf{x} - \boldsymbol{\mu})} \tag{5.7}$$

Our Marker class needs another attribute, called *mahalanobisCounter*, which keeps count of how many times the Mahalanobis distance was above a certain value V. We delete a marker from the container when the fraction between the number of times the Mahalanobis distance was higher than V (smoothed by a parameter K) over the number of matched markers, is higher than a threshold X $(0 < X < 1)$.

$$\frac{mahalanobisCounter - K}{matchedCounter} > X \tag{5.8}$$

In this way, whenever a marker is added in wrong way to the filter, if the following measures are closer to the real values, then at one point the marker is deleted from the filter, and added again to the reference world, hopefully with the right pose. Unfortunately, the presented method does not work if the marker pose is very often computed incorrectly, for certain marker configurations (refer to Section 4.5).

5.3 Experiments and results

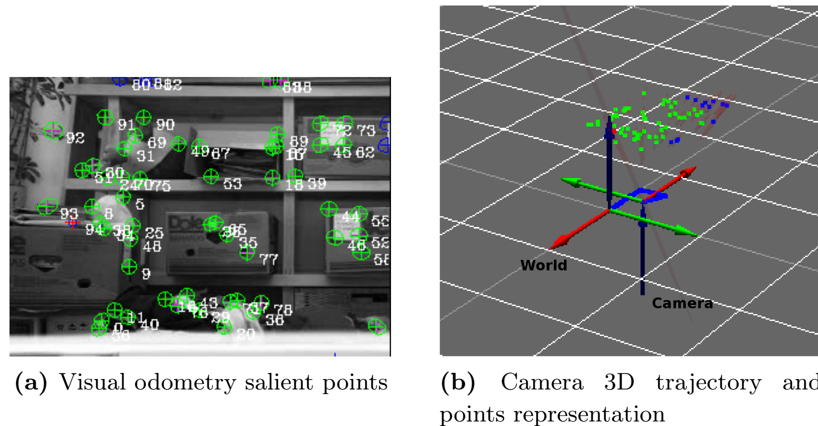
We have performed several experiments to check how SLAM algorithm could work using visual marker as landmarks. We present those experiments now, explaining the policies we employed and the obtained results. The experiments have been done in "batch mode", which means that we used videos to work on, and we did not try how the system works in real-time. Parameters and settings were passed to our program through configuration files, according to AIRConfig library (see MoonSLAM architecture section described earlier).

5.3.1 Motion model

AIRVisualMarker library was designed to work in MoonSLAM framework with any motion model. Since we do not have odometry information from the physical robot, we operate using camera images and visual odometry[30]. Visual odometry is a technique which consists in determining the position and orientation of a robot by analysing the associated camera images. This allows to track the camera (or the robot) trajectory in the world system without using traditional odometry.

In general, visual odometry works as follows:

- acquire input images using a camera (in our application, we used a single camera);



(a) Visual odometry salient points

(b) Camera 3D trajectory and points representation

Figure 5.7: Visual odometry example

- apply image processing techniques for lens distortion removal and image corrections
- detect salient features in the image and match them across frames using correlation, and then construct optical flow field
- check flow field vectors for potential tracking errors and remove outliers
- estimate the camera motion from the optical flow, i.e., using Kalman filter for state estimate distribution maintenance
- periodically repopulate trackpoints to maintain coverage across the image.

In Figure 5.7 we reported an example of visual odometry application: on the right you can see in blue the camera trajectory, based on the green points detected as salient features (remember that the camera is usually oriented following the Z-front, Y-down, X-right convention).

MoonSLAM framework uses all the green points in the image to retrieve the camera motion, but when a point goes out of the frame and it is not matched for a certain number of steps, it is deleted from the filter, so that the algorithm does not store useless points in the filter (which increases the computation time).

The problem of using image features to retrieve information about the camera motion is that, with a single camera, we have no information about the depth of points corresponding to such features. In this way, the features and the camera (or robot) trajectory are computed up to a scale factor which cannot be estimated without the use of a second camera or some knowledge

about the environment. What we aim is to integrate the information given by visual markers (which includes also the distance from the camera objective) to help the Kalman filter to calculate a correct robot trajectory, including the scale, while building a map of the environment containing such markers.

5.3.2 Management policies

We decided to apply certain policies to our application, some of which already mentioned in this chapter, in order to get best results when including visual markers in EKF-SLAM with visual odometry.

One of them concerns the marker detection phase. As explained, *Marker-Container* class allows two functions for Data Matrix detection: one searches for markers over the whole camera image with a long time-out, possibly returning all the markers present before the time expires; the second searches for a marker over the image sub-area where such marker is expected to be (after a filter measurement), with a very short time-out. As a general heuristic, we decided to call the former function every 25 frames, which means every second in our application, since our camera has a frame rate of 25 fps, both to detect new Data Matrices entered in the camera view and to detect old markers which were not correctly predicted within the image. We set a maximum time-out of 150 ms to improve chances of recognizing every marker in the image. The latter, instead, runs during every frame: the marker container calls such function for every Data Matrix expected inside the image, setting a ROI based on the four marker measures points. In this way, we can fix a very short time-out, like 20-30 ms, since the search is processed over a very small area (notice that in most cases the recognition takes only 5-10 ms). With this approach, we managed to match every marker inside the container expected in the image very fast, without missing occasional new markers.

Moreover, we decided to stop searching for a marker whenever its measured points are too close (i.e., 5-10 pixels) to the image borders. In fact, as explained in Chapter 3, Data Matrix barcodes need a white area around them (the quiet zone) to be detected: if the filter measures the marker points on the image edges, but still within the image, the container would go on searching for a marker which cannot be detected, increasing the number of times that such marker is expected in the image, with respect to the number of times it is effectively matched. This can lead the container to delete the marker from the filter, assuming it is a misclassified one.

In this sense, another management policy is about marker deletion from the container according to the general policy in Equation 5.6. If we choose

values like $K = 10$ and $X = 0.5$, we want to match at least 50% of the expected markers, but we allow to miss at most the first 10 detections: if the $11th$ time we do not match the marker, it is considered as misclassified, because $\frac{0+10}{11+10} = 0.476 < 0.5$. With the fast search expedient described above, *libdmtx* manages to match the markers present in the sub-frames most of the times, so we set a threshold $X = 0.7$. As smoothing parameter, we thought $K = 10$ could be a good trade-off: we accept to miss the first 4 detections at most ($\frac{0+10}{5+10} = 0.66 < 0.7$), which likely means that the marker was wrongly decoded.

Again about marker deletion, we had to set the parameters for deleting a marker using Mahalanobis distance indicator (Equation 5.8). We accept Mahalanobis distances under a value V, which represents the value of a *chi-square* distribution with a number of degrees of freedom equal to the considered vector length. With a vector with 12 elements (8 pixel points plus 4 depths), if we want to accept 95% of the measurements we need to set $V = 21.026$. As smoothing parameter, $K = 10$ is a good trade off to smooth the first detections, as explained for the other deletion policy. We delete a marker if the number of wrong estimates over the number of detected markers is over 50%, because this means that the measures are more often very different from the real values than close to them.

One of the most important policies chosen to get the best results during the integration of our library in MoonSLAM, was to help visual odometry feature scaling even before the features were fed to Kalman filter. In fact, we use graphical information from markers in the image to automatically set the depth of new points detected by visual odometry, with a very low variance: the general idea is that if the visual odometry algorithm chooses a point internal to a Data Matrix as a salient feature, we can lower the uncertainty about the depth of that point and set with much more precision its distance from the camera (but represented with its inverse, since MoonSLAM operates with Unified Inverse Depth parametrization). Unfortunately, this step can be done only with new points, not with the ones already in the filter: for this reason, we decided to start executing visual odometry algorithm only after a first marker is detected, to avoid keeping useless information about points inside a marker not yet recognized.

Given all the markers in the container (more specifically, the ones in the image frame), checking if a visual odometry 2D point falls inside the marker perimeter is trivial. More complex is setting its distance as the distance of the marker from the camera. In fact, as we said, a marker is represented as XYZ coordinates system: when we solve the PnP problem and retrieve the

marker pose in camera reference frame, the distance from the camera (i.e., the marker position along z-axis) is just referred to the marker origin (which is the bottom-left corner, in our convention). Instead, the right depth to consider is the one from the visual odometry point to the plane where the marker lays, defined by its X and Y axis.

Let us call ρ the inverse depth we want the visual point to have, $\mathbf{R}_M^C$ the rotation matrix which defines the orientation of a marker in camera frame, $\mathbf{T}_M^C$ the translation of the marker in camera frame (i.e., the bottom-left point coordinates). We know that a geometric plane $\boldsymbol{\pi}$ in $\mathbb{R}^3$ is defined by the equation

$$ax + by + cz + d = 0 \tag{5.9}$$

that is $\boldsymbol{\pi} = [a, b, c, d]^T$. Specifically, $[a, b, c]^T$ is the the vector orthogonal to the plane, which considering the marker plane is represented by the third column of the rotation matrix $\mathbf{R}_M^C$ (we call it $\vec{\boldsymbol{n}} = [n_x, n_y, n_z]^T$). Furthermore, we know that the bottom-left marker point $\mathbf{p} = \mathbf{T}_M^C = [x, y, z]^T$ belongs to the plane:

$$\mathbf{p} \in \boldsymbol{\pi} \Leftrightarrow \boldsymbol{\pi}^T\mathbf{p} = 0 \Leftrightarrow \begin{bmatrix} \vec{\boldsymbol{n}} & d \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = 0 \Leftrightarrow \vec{\boldsymbol{n}}\mathbf{p} + d = 0 \Leftrightarrow d = -\vec{\boldsymbol{n}}\mathbf{p} \tag{5.10}$$

Now we have all the plane parameters and if we want to find the inverse depth of a point $\mathbf{q}$ belonging to the marker plane $\boldsymbol{\pi}$ (inside the marker contour) projected on the normalized image plane (i.e., with focal distance equal to one) on the point $\mathbf{q}' = [x', y', 1]$, first we find the the vector $\mathbf{q}'$ direction ($\vec{\boldsymbol{d}}$) in camera coordinates, secondly we calculate the inverse depth ρ knowing that the point $\mathbf{q} = \frac{1}{\rho}\vec{\boldsymbol{d}}$ belongs to the marker plane, with the following:

$$\vec{\boldsymbol{d}} = \frac{\mathbf{q}'}{\|\mathbf{q}'\|} \tag{5.11}$$

$$\begin{aligned} \begin{bmatrix} \vec{\boldsymbol{n}} & d \end{bmatrix} \begin{bmatrix} \mathbf{q} \\ 1 \end{bmatrix} = 0 \Leftrightarrow \begin{bmatrix} \vec{\boldsymbol{n}} & d \end{bmatrix} \begin{bmatrix} \frac{1}{\rho}\vec{\boldsymbol{d}} \\ 1 \end{bmatrix} = 0 \Leftrightarrow \frac{1}{\rho}\vec{\boldsymbol{n}}\vec{\boldsymbol{d}} + d = 0 \Leftrightarrow \\ \Leftrightarrow \frac{1}{\rho}\vec{\boldsymbol{n}}\vec{\boldsymbol{d}} = -d \Leftrightarrow \rho = -\frac{\vec{\boldsymbol{n}}\vec{\boldsymbol{d}}}{d} \end{aligned} \tag{5.12}$$

In Figure 5.8 we report a graphical explanation of what we just described. $\boldsymbol{\pi}$ is the marker plane, $\boldsymbol{\pi}_N$ is the normalized marker plane and $\boldsymbol{\pi}_C$ is the camera image plane.

Once we obtain the visual odometry point inverse depth from the marker, we can force the filter to set it with very low standard deviation (we set a standard deviation of $2mm^{-1}$, as opposed to the other points standard deviation which can be up to $10cm^{-1}$).

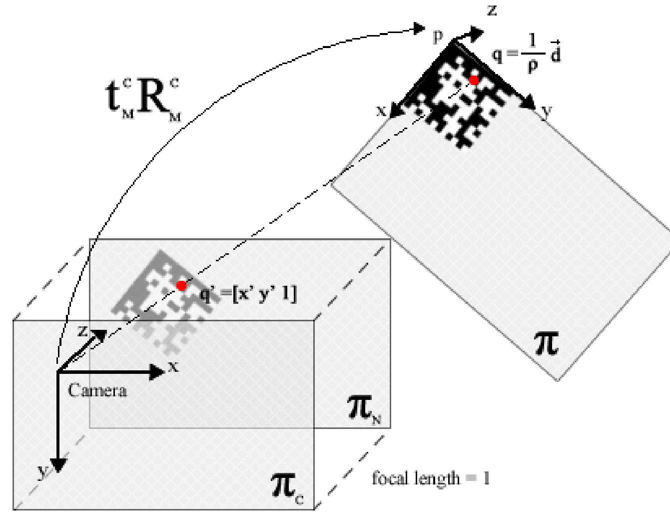


Figure 5.8: Inverse depth initialization

5.3.3 EKF-SLAM with visual markers

Here we describe briefly how EKF-SLAM algorithm is executed by MoonSLAM using a visual odometry motion model, after the integration of visual markers. The algorithm follows the same basic structure already presented in Figure 5.5 and it is graphically reported in Figure 5.9.

1. The *ImageManagerInterface* object, given a camera images list as input, returns a pointer to the next frame to process;

2. the *MarkerContainer* searches for new markers in the image with the “long search”, but no visual odometry is computed until a first marker is detected in the scene. This expedient helps to set the new odometry points detected inside markers at the right depth from the camera;

3. when the first marker is recognized, the *MarkerContainer* adds its pose with respect to the world to the Kalman filter, the world reference frame which in the first detection corresponds to the camera reference frame, because the camera has not moved yet according to the motion model;

4. visual odometry algorithm starts the execution, finding new salient features and initializing the depth of the ones laying inside a marker contour to the right value with low uncertainty, and the depth of the other ones to a value preset in the configuration file.

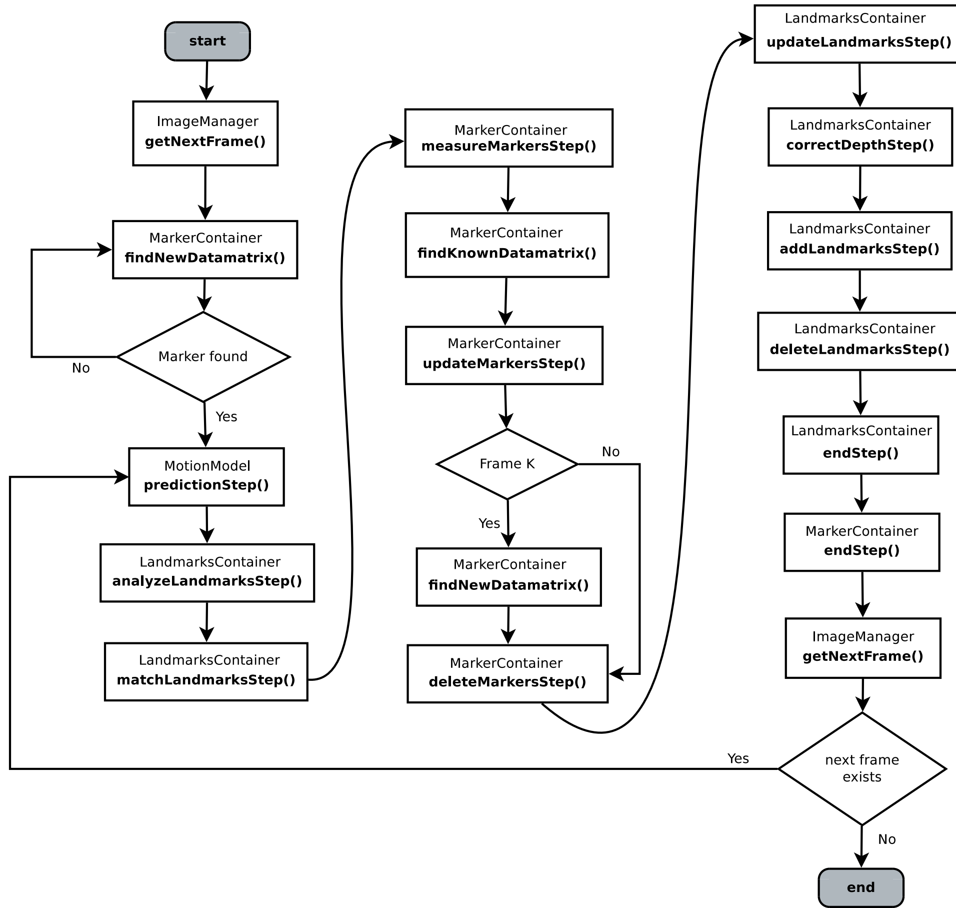


Figure 5.9: Main steps of an EKF-SLAM algorithm in the MoonSLAM framework using visual markers

5. the *DynamicPredictorBase* computes SLAM prediction step and calculates the robot's next state, together with position and velocity uncertainties, and updates the related means and covariances in the filter;

6. the *LandmarksAnalyzer* object, according to the prediction step results, predicts old landmarks position within the map;

7. the *LandmarksMatcher* searches for associations between old landmarks and new detected features in the current frame;

8. the *MarkerContainer* measures the image points and their depths of all the old markers detected till that moment, according to the prediction step results;

9. the *MarkerContainer* searches all the old markers in a sub-area of

the image where the filter previously measured them, using the "short search", and match them if the search gives a positive result;

10. the *MarkerContainer* updates the image positions of the markers for which a match was found in the current frame;

11. once every K frames (K decided a priori) the *MarkerContainer* calls again the "long search", in order to detect and add to the container and the filter new markers not yet recognized, and, in case, to correct the position of old markers wrongly measured during the algorithm;

12. the *MarkerContainer* removes from the filter potentially misclassified markers, whenever their match rate is lower than a established threshold;

13. the *LandmarksUpdater* object updates the landmarks positions for which a match was found in the current frame;

14. the *LandmarksAdder* adds to the landmarks container (and to the extended Kalman filter) occasional new landmarks detected in the current frame. If these new landmarks are positioned within an existing marker contour in the image, their depth from the camera is properly set according to the marker distance from the camera;

15. the *LandmarksDeleter* removes from the container and from the filter potential unnecessary landmarks (i.e., the ones having negative depth in a UID parametrization);

16. finally, map landmarks and markers information related to execution steps are updated.

If a frame is available after the current one, the process starts again from prediction step, otherwise the algorithm terminates.

5.3.4 Results - trajectory scaling

We tested how markers integration in visual odometry using EKF-SLAM techniques would have affected the camera pose in world coordinates. Indeed, as we said, visual odometry cannot provide information about points depth, when a single camera is used, but it builds the camera trajectory and environment map up to a scale factor. The use of visual markers should help the filter to correct this based on their known pose values.

We recorded three videos:

- in the first one we keep the camera in hand, following a trajectory as much as possible linear, moving the camera horizontally toward right (which means along positive values of x-axis) and slightly backward (which means along negative values of z-coordinate); we used only two Data Matrix markers in the scene, positioned at about 1.30 m;
- in the second video, we used a sliding truck on rails to force the camera onto a linear movement. Also in this case, we moved the camera horizontally toward right till the end of the rail (long about 1.21 m), then all the way back to the beginning, using four Data Matrix positioned at about 2.30 m from the camera;
- in the last video, we used the same scene of the second one, but keeping the camera closer to the markers (at about 1.70 m) and moving horizontally toward left (i.e., toward negative values of x-coordinate) till the end of the rail, and then back to the beginning.

In Figure 5.10 you can see the camera trajectories along x, y and z-axis using only visual odometry information (plotted in red) and integrating also information about visual markers (plotted in blue) recorded in the first video. Since we were moving the camera right along the x-axis, we see increasing values of x-coordinate: we notice that while visual odometry estimates a longer trajectory (about 1.90 m) than it was in reality (we moved of about 1 m), visual markers help the filter to scale the distances to the real value (we see that the scaled trajectories is about 0.9 m). We can also clearly see that trajectory along z-axis tends to decrease, due to the fact that we were also moving the camera backward. Of course, holding the camera in hand introduces a lot of noise (minimal unwanted movements), which are clearly visible analysing trajectories along y-axis. Visual odometry tends to amplify such movements because it is based only on image points, so a sudden movement can badly affect the estimated position, while visual markers make the trajectories more smoothed.

In Figure 5.11 we show the same quantities just described, referring to the second scene. Also in this case we can notice that while visual odometry overestimates the distance covered along x-axis (it says that the camera moved about 1.5 m on the right), visual markers scale such distance to a value closer to the real one (1.23 m, on a rail 1.21 m long), before coming back to the initial point. The same thing happens with the trajectories along y-axis and z-axis: there is a slight linear movement along those axis, because the rail on which the truck was sliding was a little bit slanted and not perfectly aligned to the ground, but visual markers correct the camera

trajectory to the scaled value.

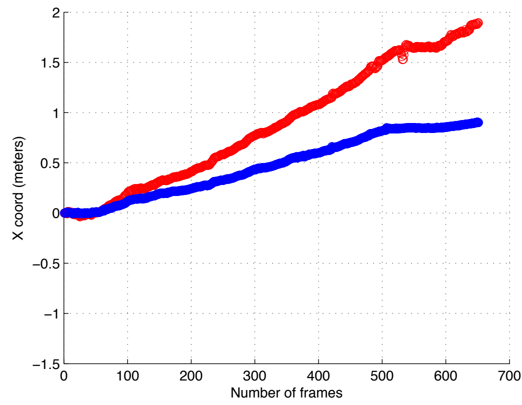
In Figure 5.12 the same results are presented, referred to the third scene, i.e., the second setting but recorded from a closer distance. This time, visual odometry works perfectly in estimating the movement of the camera along x-axis, because it scales it almost equally to the one computed with markers integration, but it can be considered a lucky case, not the general one, as we saw earlier, and can depend from the algorithm initialization. We can draw the same conclusions we made for the other two situations about scaling the y and z-axis.

Results show that integrating visual markers in a visual odometry algorithm adjusts the camera motion to more certain values, by giving more visual information to the system through camera images, at a minor price (essentially, just modifying the environment with the markers). Of course, if we dispose of a physical robot with information about odometry from the wheels, the camera (robot) trajectory could be estimated even more precisely. However, visual markers are still useful as landmarks for the SLAM to build the environment map.

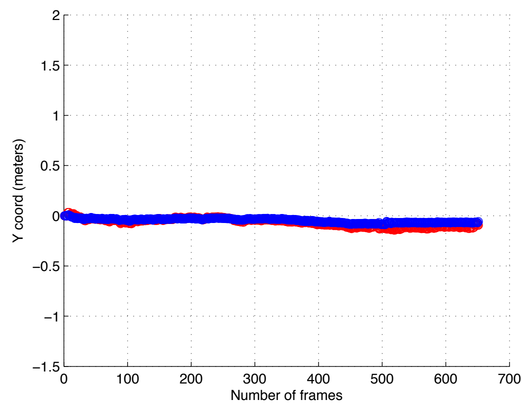
5.3.5 Results - landmarks scaling

Strictly connected to the previous problem, landmarks scaling, which means scaling the features position in the 3D world, can be solved using visual markers in the SLAM algorithm. In fact, like in estimating camera trajectory, the salient features in the image are positioned at an unknown distance from the camera, because it is not possible to compute the image points depth with a single camera, as we said, but only estimate it with a scaling factor. Using visual markers helps the Kalman filter to update the landmark points position in the world, knowing that each marker is positioned at a certain distance and with a certain orientation with respect to the camera. Moreover, as we explained earlier talking about our management policies, we can force the filter to set the depth of the visual odometry points laying inside a Data Matrix to the right value, since we know how far is positioned that marker from the camera.

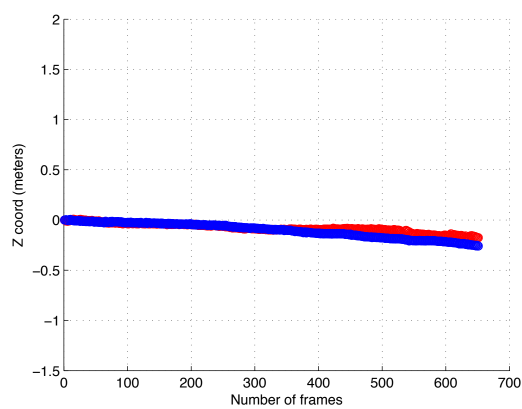
In order to show the differences of SLAM algorithm in landmarks scaling with and without using visual markers, we compare the two situations on the same image frame of our videos (we show only the one with camera in hand and the one with the truck close to the markers, since the scene is the same). To understand the next figures, we remind that a Data Matrix is represented with the x-axis (in red) corresponding to the marker bottom edge and the y-axis (in green) corresponding to the marker left edge: consequently, when



(a) Trajectory along x-axis

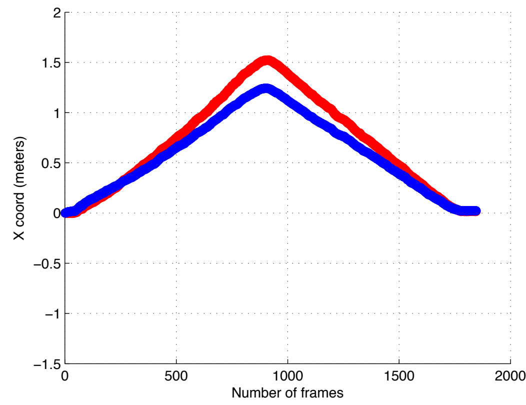


(b) Trajectory along y-axis

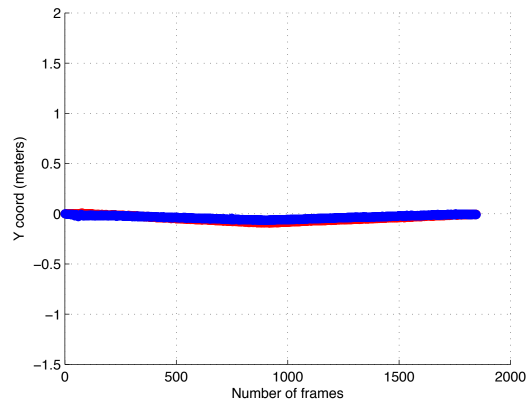


(c) Trajectory along z-axis

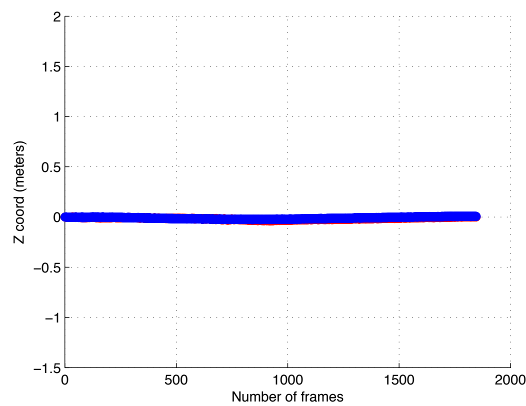
Figure 5.10: Camera trajectory using only visual odometry (in red) and with markers information integration (in blue) - holding camera in hand (first scene)



(a) Trajectory along x-axis

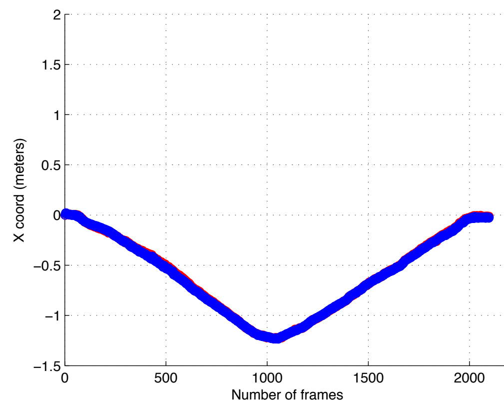


(b) Trajectory along y-axis

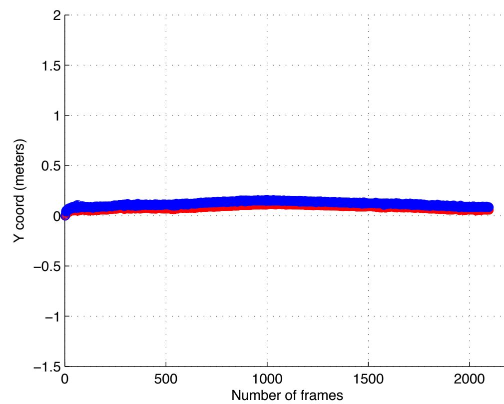


(c) Trajectory along z-axis

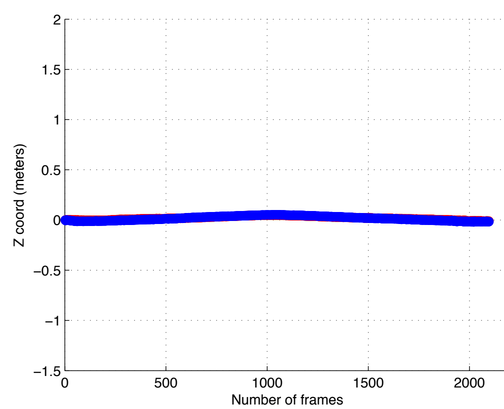
Figure 5.11: Camera trajectory using only visual odometry (in red) and with markers information integration (in blue) - camera far from the scene (second scene)



(a) Trajectory along x-axis



(b) Trajectory along y-axis



(c) Trajectory along z-axis

Figure 5.12: Camera trajectory using only visual odometry (in red) and with markers information integration (in blue) - camera close to the scene (third scene)

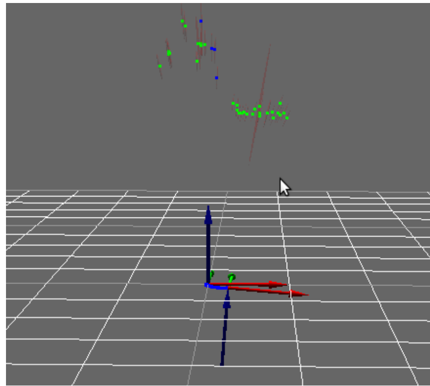
the z-axis (in blue) points downward, it means that the marker is faced down and the camera points upward (along its z-axis).

In Figure 5.13 you can see that without visual markers integration, the landmarks are coherently positioned in the world, but up to a scaling factor, according to the frame we are considering: points close to the marker are grouped together, while other further points are more distant. If we compare this situation with the same one where a marker is detected, we notice that the points laying inside the marker are set exactly on the marker XY plane and all the others are adjusted accordingly to the marker metrics. In particular, the landmarks positioned further in the background would eventually stretch and converge to their position in fast way: you can see that they are computed with an high variance (the red ellipses around them) because they actually are still too close to the camera, so they will be taken up to their real position. Notice also in this figure the differences between the two camera trajectories: it is more “contracted” using markers than with only visual odometry points, confirming the earlier results that the trajectory computed without the markers help is scaled.

In the same way, in Figure 5.14 you can see how all the points are grouped together in the world, without using markers, more or less at the same height. In the same frame (even though visual odometry points are different, because they change from one algorithm execution to another), the visual markers give to the filter more certain information about their position in the world reference. So all the points that are aligned with the markers are compressed on the marker XY planes and set to the right distance from the camera world reference.

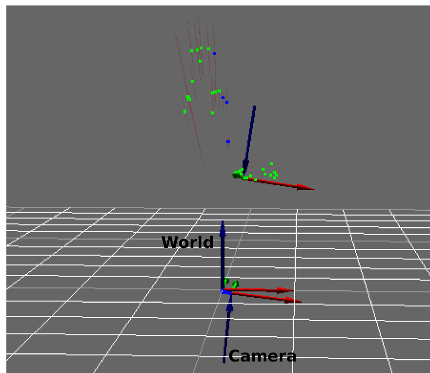
5.3.6 Results - map building

SLAM algorithm, as its name indicates, also takes care to build a map of the environment from landmark observations. Therefore, visual markers represent both landmarks and map elements, which are easy to localize. So, after a marker has been added to the filter it helps the algorithm to adjust the camera trajectory together with all the other observed landmarks, and in the meanwhile its absolute pose is corrected every time such marker is measured and detected again. SLAM algorithm would eventually converge measuring a very high number of times (potentially infinite) the same marker, building in the end a complete map of the explored world, where the robot is able to localize itself and its trajectory is the well defined.



(a) Landmarks detected by visual odometry without markers information

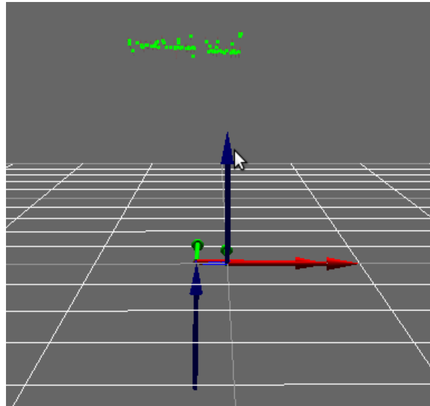
(b) Current frame without marker recognition



(c) Landmarks detected by visual odometry with markers information

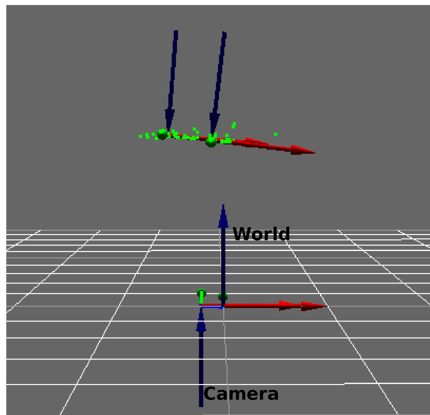
(d) Current frame with marker recognition

Figure 5.13: Visual landmarks scaling - holding camera in hand (first scene)

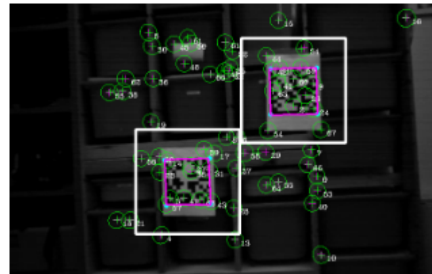


(a) Landmarks detected by visual odometry without markers information

(b) Current frame without marker recognition



(c) Landmarks detected by visual odometry with markers information



(d) Current frame with marker recognition

Figure 5.14: Visual landmarks scaling - camera close to the scene (third scene)

In Figure 5.15 and Figure 5.16 you can see the environment map composed by visual markers, respectively for the first and third scene (which, we repeat, is the same as the second one). The first map is composed only by two markers, aligned between them because they are parallel. They are a little bit slanted because the camera is following a tilted trajectory (the camera was slightly rotated).

Regarding the second map, it is visible how all the markers are aligned between them in the right positions: every time a marker is detected, such map is refined until it converges to a nice result as the one in figure. As you can see, the camera trajectory is well defined respect to the marker positions, so that it is possible for a robot to localize itself within the map.

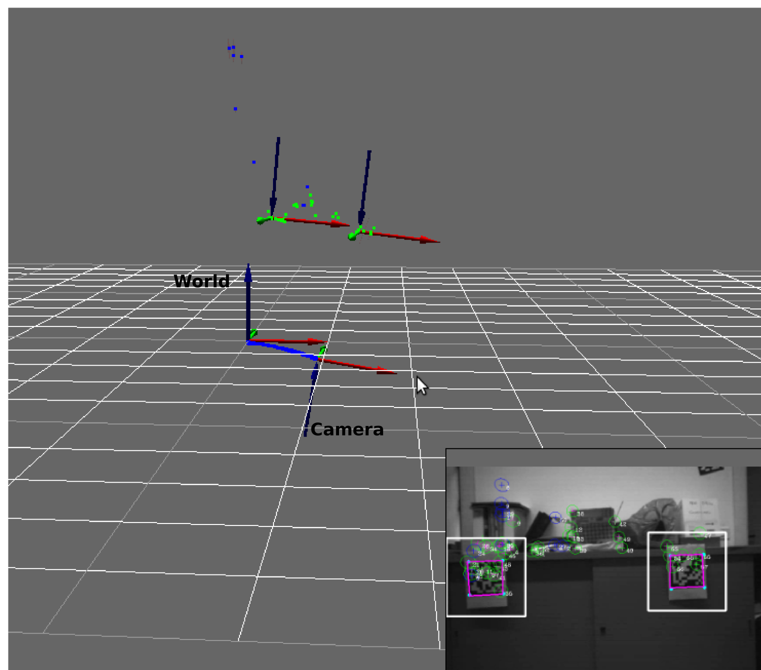


Figure 5.15: Map building - camera in hand (first scene)

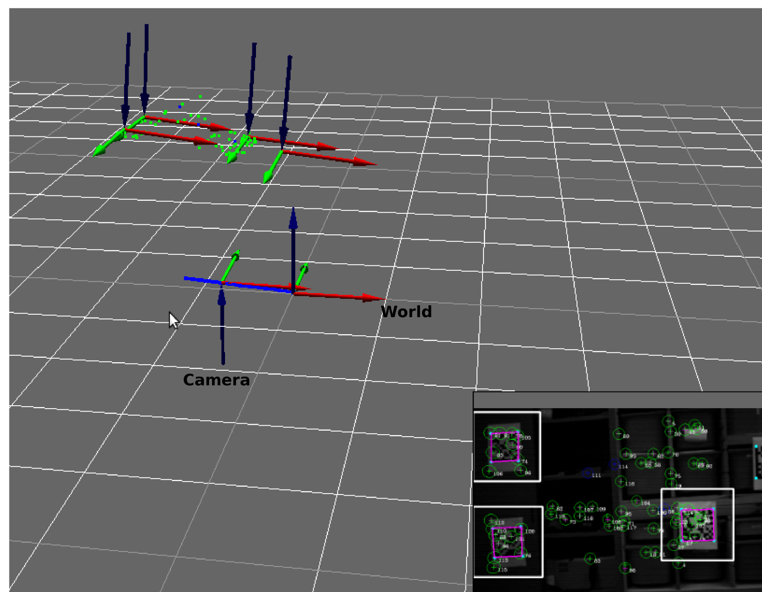


Figure 5.16: Map building - camera close to the scene (third scene)

Chapter 6

Conclusions and future works

"Just because something doesn't do what you planned it to do doesn't mean it's useless"

Thomas A. Edison

In this thesis, we proposed a library for Simultaneous Localization And Mapping (SLAM) algorithm using Data Matrix visual markers as landmarks, within a framework for SLAM developed at Politecnico di Milano, called MoonSLAM. In the presented method we showed how it is possible to detect a Data Matrix, to refine it with computer vision techniques and to integrate it in SLAM as artificial landmark.

First, we introduced Data Matrix markers, describing how to detect them in a 2D image. We discussed three different vision algorithms useful to refine the marker corners in order to retrieve the most possible accurate marker pose estimate in camera coordinates system. We analysed and compared Good Features To Track (GFTT), Features from Accelerated Segment Test (FAST) and Canny with Hough methods, by comparing the marker pose values with a chessboard pose values, for each vision algorithm, in different static and dynamic scenes. We considered the chessboard pose as an accurate estimate of the real pose in camera reference frame, and we based our choices on this assumption. From the results, we discarded Canny algorithm and we generally accepted FAST algorithm, because it estimates in a correct way the marker pose most of times, even if sometimes it produces high errors. GFTT algorithm, instead, is more stable and in general it works better than the other two algorithms, but it may happen that it causes the wrong marker pose to be estimated, for certain marker configurations.

Then, we described our library structure, explaining how it interfaces with MoonSLAM framework, its main classes and functions. We showed

how our library deals with the addition of a marker to the Kalman filter as a landmark, how it is deleted and how the filter computes the measurements on the marker attributes, discussing the management policies we chose to execute the SLAM algorithm in a generic static indoor environment with visual odometry as motion model.

Finally, we reported the main results we obtained thanks to the integration of visual markers in the filter, showing how the metric information given by Data Matrix markers can help the filter to adjust the robot trajectory, localize the camera with less uncertainty, and build a map of the environment in correct way, with the right metrics.

There are still some improvements that can be done to have a better working system. In fact, to compute the perspective N-points process we use only four image points from our Data Matrix marker (which is the minimum number), that is the fastest and the most intuitive choice, because the library for Data Matrix detection (*libdmtx*) already provides data to roughly obtain those points. However, using four points, as we discussed in the thesis, can sometimes lead to marker poses not always correctly estimated. We adopted a series of techniques to smooth this behaviour, like employing computer vision algorithms to refine the marker corners or the policy based on Mahalanobis distance to delete wrong markers, but the base problem persists. This could possibly prevent in some cases SLAM to work properly.

Considering what we said above, future works in this field follow several directives. First of all, it is possible to improve the marker pose estimation process by solving the base problem, which means to employ more than four points to compute the PnP problem. In a Data Matrix, in fact, the internal marker structure is known “a priori” in 3D coordinates, so a possible try-out would be to design a vision detection method to match certain Data Matrix internal corners in a 2D image and use them as image points to solve the PnP problem. The more points are available as image points, the more accurate the PnP function performs. This expedient may solve a big part of the presented problem, because a precise marker pose estimate would avoid the SLAM filter to build a possibly incorrect map of the environment. Along this directive, it is also possible to analyse other different computer vision algorithms or, in case, to improve the existing ones, to refine in a better way the rough estimates of the marker corners returned by *libdmtx* and obtain more accurate image points to compute the marker pose. Alternatively, it is suggested to solve the pose estimation problem “a posteriori”, by identifying wrong poses and not keeping them into consideration when computing

the update phase in SLAM. We proposed a method based on Mahalanobis distance, but other try-outs are plausible.

Another open improvement is to add the support for other visual markers to AIRVisualMarker library. In this thesis we presented Data Matrix barcodes, but we cited other 2D markers, like QRCode and ARToolKitPlus, which may be employed as alternative landmarks, making our library more flexible. A good idea is to implement a detection system working simultaneously with different types of visual marker, maybe encoding heterogeneous information for SLAM system.

The system we implemented was tested using a motion model based on visual odometry. A possible research directive would be to test the functioning employing a physical robot in large environments, using information retrieved by wheels odometry to build a motion model and integrate it with our library. Ground odometry provides more robust information to the system than simple visual odometry, based only on camera images, so it is likely that the SLAM algorithm with visual markers would work very well.

In conclusion, we showed that SLAM can be integrated with visual markers as artificial landmarks to improve the known data association problems present with the use of natural landmarks, by providing the system filter with certain information given by visual markers.

Bibliography

[1] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, August 1997.

[2] Serge Belongie. Rodrigues' rotation formula. From MathWorld - A Wolfram Web Resource.

[3] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library.* 2008.

[4] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8:679–698, June 1986.

[5] Simone Ceriani, Daniele Marzorati, Matteo Matteucci, Davide Migliore, and Domenico G. Sorrenti. On feature parameterization for ekf-based monocular slam. In *In proceedings of 18th World Congress of the International Federation of Automatic Control (IFAC)*, pages 6829–6834, August 2011.

[6] Javier Civera, Andrew J. Davison, and J. M. Martínez Montiel. Unified inverse depth parametrization for monocular slam. In *In Proceedings of Robotics: Science and Systems*, 2006.

[7] Javier Civera, Oscar G Grasa, Andrew J Davison, and J M M Montiel. 1-point ransac for extended kalman filtering : Application to real-time structure from motion and visual odometry. *Journal of Field Robotics*, 27(5):609–631, 2010.

[8] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ICCV '03, pages 1403–, Washington, DC, USA, 2003. IEEE Computer Society.

[9] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. Stanford University, Stanford, California 94301-9010, 2006.

[10] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics & Automation Magazine*, 2006.

[11] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part ii. *IEEE Robotics & Automation Magazine*, 2006.

[12] Mark Fiala. Artag, a fiducial marker system using digital techniques. *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2:590–596, 2005.

[13] Mark Fiala. Comparing artag and artoolkit plus fiducial marker systems. *Haptic Audio Visual Environments and their Applications 2005 IEEE International Workshop on*, 2(20-25):6, 2005.

[14] GS1. *GS1 Data Matrix Applications*, 2011.

[15] Martin Hirzer. Marker detection for augmented reality applications. *Image Rochester NY*, 2008.

[16] P. V. C. Hough. A method and means for recognizing complex patterns. 1962. U.S. Patent No. 3,069,654.

[17] http://ros.org/wiki/datamatrix.

[18] http://www.alliedvisiontec.com/emea/products/cameras/gigabit-ethernet/prosilica gc/gc750.html.

[19] http://www.libdmtx.org/.

[20] R.E. Kalman. A new approach to linear filtering and prediction problems.

[21] I.P.H. Kato and M. Billinghurst. *ARToolkit User Manual, Version 2.33*. Human Interface Technology Lab, University of Washington, 2000.

[22] Gukhwan Kim and Emil. M. Petriu. Fiducial marker indoor localization with artificial neural network. *IEE/ASME International Conference on Advanced Intelligente Mechatronics*, 2010.

[23] Georg Klein and David Murray. Parallel tracking and mapping on a camera phone. In *Proceedings of the 2009 8th IEEE International Symposium on Mixed and Augmented Reality*, ISMAR '09, pages 83–86, Washington, DC, USA, 2009. IEEE Computer Society.

[24] Hyon Lim and Young Sam Lee. Real-time single camera slam using fiducial markers. *Image Processing*, pages 177–182, 2009.

[25] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *AUTONOMOUS ROBOTS*, 4:333–349, 1997.

[26] P.C Mahalanobis. On the generalised distance in statistics. *Proceedings National Institute of Science, India*, 2:49–55, 1936.

[27] Daniele Marzorati, Matteo Matteucci, Davide Migliore, and Domenico G. Sorrenti. On the use of inverse scaling in monocular slam. In *Proceedings of the 2009 IEEE international conference on Robotics and Automation*, ICRA'09, pages 2907–2913, Piscataway, NJ, USA, 2009. IEEE Press.

[28] Alan Mutka, Damjan Miklic, Ivica Draganjac, and S Bogdan. A low cost vision based localization system using fiducial markers. *World*, pages 9528–9533, 2008.

[29] Richard A. Newcombe and Andrew J. Davison. Live dense reconstruction with a single moving camera. In *IEEE Conference on Computer Vision and pattern Recognition*, 2010.

[30] David Nistér, Oleg Naroditsky, and James R. Bergen. Visual odometry for ground vehicle applications. *J. Field Robotics*, 23(1):3–20, 2006.

[31] S. Panzieri, F. Pascucci, R. Setola, and G. Ulivi. A low cost vision based localization system for mobile robots. June 2001. vision.

[32] Long Quan and Zhongdan Lan. Linear n-point camera pose determination. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21:774–780, August 1999.

[33] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. pages 430–443, 2006.

[34] Paul Schmidmayr, Martin Ebner, and Frank Kappe. What is the power behind 2d barcodes. are they the foundation of the revival of print media. *Proceedings of I-Know08 and I-Media08, 6th International Conference on Knowledge Management and New Media Technology.*

[35] Jianbo Shi and Carlo Tomasi. Good features to track. 1993.

[36] Joan Sola, Andre Monin, Michel Devy, and Thomas Lemaire. Undelayed initialization in bearing only SLAM. In *International Conference on Intelligent Robots and Systems*, 2005.

[37] Joan Solà. Consistency of the monocular ekf-slam algorithm for three different landmark parametrizations. In *ICRA*, pages 3513–3518. IEEE, 2010.

[38] Joan Solà, André Monin, and Michel Devy. Bicamslam: Two times mono is more than stereo. In *ICRA*, pages 4795–4800. IEEE, 2007.

[39] Alexandros Stathakis and Emil M. Petriu. Robust pseudo-random fiducial marker for indoor localization. *Proc. ROSE 2011, IEEE Int. Symp. Robotic and Sensor Environments*, 2011.

[40] Christoph Stock, Ulrich Mühlmann, Manmohan Krishna Chandraker, and Axel Pinz. Subpixel corner detection for tracking applications using cmos camera technology. *Pattern Recognition*, (September):191 – 199, 2002.

[41] R. Szeliski. *Computer Vision: Algorithms and Applications.* 2010.

[42] G. Welch and G. Bishop. An introduction to the kalman filter. *University of North Carolina at Chapel Hill, Chapel Hill*, 2006.

[43] Tatsuya Yamada, Takehisa Yairi, Suay Halit Bener, and Kazuo Machida. A study on slam for indoor blimp with visual markers. *Update*, pages 647–652, 2009.

[44] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1998.

Appendix A

Notation: 3D Rototraslations

This chapter is dedicated to the description of the rototranslation transformations used in this thesis.

A 3D rototranslation is a geometric transformation composed by a rotation followed by a translation.

A.1 Rotation

The rotation is the first part of the rototranslation, and it is represented by a rotation matrix or by a vector. The rotation matrix is used to perform the rotations to the points and vectors in 3D space. The reason is that the algebra is straightforward and linear, and that the number of computing operations is minimized. $\mathbf{RT}_B^A$ is the transformation between the two reference frames A and B, which applied to a point p expressed in reference frame B returns p in the reference frame A.

The generic rotation matrix is represented as:

$$\mathbf{RT}_B^A = \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix} \tag{A.1}$$

For instance, a rotation of θ around axis z is represented as:

$$\mathbf{R}_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.2}$$

Two rotation matrices can be composed using the standard matrix product:

$$\mathbf{R}_B^A = \mathbf{R}_C^A \mathbf{R}_A^B \tag{A.3}$$

A rotation matrix has two properties, the first one is that:

$$det(\mathbf{A}) = 1 \tag{A.4}$$

and the second is that the transpose matrix is equal to the inverse matrix

$$\mathbf{R}\mathbf{R}^T = \mathbf{R}^T\mathbf{R} = \mathbf{R}\mathbf{R}^{-1} = \mathbf{R}^{-1}\mathbf{R} = \mathbf{I} \tag{A.5}$$

The quaternion is used to store orientation information in the state vectors. The time evolution equations based on quaternions are continuous and continuously derivable, something that will be very important for further automatic manipulations such as filtering. Because the quaternion q plays such a central role, it will be convenient to use it by default in all frame specifications. A quaternion Q is written in the form

$$\mathbf{q} = [a, b, c, d]^T \in \mathbb{R}^4 \tag{A.6}$$

The conjugated of the quaternion is defined by $\mathbf{q} = [a, -b, -c, -d]^T$ and its norm, where the explicit writing of the multiplication doesn't indicate the use of quaternion algebra.

A.2 Translation

The translation is the second part of a rototranslation and is represented by a vector of three elements. Each elements is the value that represent the translation of each coordinate from the frame A to the frame B.

$$\mathbf{T}_B^A = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{A.7}$$

is the translation from frame B to frame A, and the inverse transformation is:

$$\mathbf{T}_A^B = -\mathbf{T}_B^A \tag{A.8}$$

A.3 Rototranslation

The rototranslation is the composition of a rotation $\mathbf{R}_A^B$ and a translation $\mathbf{T}_A^B$, and is represented by the following matrix:

$$\mathbf{RT}_A^B = [\mathbf{R}_A^B|\mathbf{T}_A^B] = \begin{bmatrix} r_1 & r_2 & r_3 & t_x \\ r_4 & r_5 & r_6 & t_y \\ r_7 & r_8 & r_9 & t_z \end{bmatrix} \tag{A.9}$$

Two rototranslation matrices can not be composed like two rotation matrices because they are not square matrices and is not possible to determine an inverse matrix for a rototranslation matrix. Homogeneous coordinates are used to solve these problems.

A.4 Homogeneous coordinates

Homogeneous coordinates allow several operation between rototranslation matrices. A rototranslation matrix in homogeneous coordinates is written like:

$$\mathbf{RT} = \begin{bmatrix} r_1 & r_2 & r_3 & t_x \\ r_4 & r_5 & r_6 & t_y \\ r_7 & r_8 & r_9 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.10}$$

The last element of the matrix represents a scalar factor that in robotics is always set at 1. This representation is really important because allows composition between rototranslation matrices and also inversion, the determinant of the rototranslation matrices is not equal to one as for rotation matrices.