POLITECNICO DI TORINO

MASTER DEGREE THESIS

Department or Control and Computer Engineering

Mechatronic Engineering

a.y. 2023/2024

Politecnico
di Torino

Enhacement of 2D lidar-based SLAM in an accelerator housing using fiducial markers

Supervisors:

Marcello CHIABERGE
(*Politecnico di Torino*)

Thomas T.THAYER
(*SLAC National Accelerator Laboratory*)

Candidate:

Valeria PIGNATARO

July 2024

POLITECNICO DI TORINO

Abstract

Department or Control and Computer Engineering

Mechatronic Engineering

Enhacement of 2D lidar-based SLAM in an accelerator housing using fiducial markers

by Valeria PIGNATARO

Mobile robots are known for their flexibility and autonomy, being able to execute different tasks in dynamic and non-structured environments. Regardless of the robot's aim, the main thing underlying the correct execution of whatever task is their ability to move autonomously in environments that might be known or not. For this reason, great attention has been paid to develop efficient algorithms of localization and mapping, in particular when it come to environments where the most common GPS-tracking systems can not be used. Nowadays, mapping the surrounding environment and localizing a robot inside the map created can be done simultaneously using SLAM (Simultaneous localization and mapping) algorithms.

The aim of the thesis project is the implementation of a mapping and localization system inside an indoor repetitive environment. Due to the particular structure of the environment, navigation performances using standard localization algoritms such as Monte Carlo particle filter do not have optimal performances. For this reason the standard localization approach has been enforces using Arucos, a set of fiducial marker whose position is known. Tests have been performed using a real robot and the testing environment is the accelerator housing of SLAC (Stanford linear accelerator laboratory). Thanks to the usage of the fiducial markers, safe and reliable navigation can be performed, avoiding collisions and obstacles.

Contents

Abstract **i**

1 Introduction **1**
1.1 Thesis objective . . . 1
1.2 Robot . . . 2
1.2.1 Hardware . . . 2
1.2.2 Software . . . 3
1.3 Thesis structure . . . 4

2 Background **6**
2.1 Transformations and transformation matrices . . . 6
2.1.1 Pose of a rigid body: . . . 6
2.1.2 Rotation matrix . . . 7
2.1.3 Parametrizations of rotation: . . . 9
ZYZ Angles . . . 10
RPY angles . . . 11
Angle-Axis . . . 12
Quaternion . . . 13
2.1.4 Homogeneous Transformation: . . . 14
2.2 ROS - Robotic Operating System . . . 14
2.2.1 Graph Structure: . . . 16
2.2.2 ROS commands: . . . 16
2.2.3 Transform Tree: . . . 17
2.2.4 Coordinate Frame for Mobile Platforms: . . . 18
2.2.5 Rviz: . . . 19
2.3 Camera Model . . . 20
2.3.1 Pinhole camera model . . . 20
2.3.2 Camera calibration and pose computation: . . . 20

3 State of art **23**
3.1 SLAM . . . 23
3.1.1 Related concepts . . . 23
3.1.2 Sensor-based classification . . . 24
3.1.3 ROS open source packages for SLAM . . . 25
3.2 Monte Carlo Localization . . . 26
3.3 Fiducial Markers . . . 27

4 Algorithms and methods **29**
4.1 Sensors calibration . . . 29
4.1.1 IMU . . . 29
4.1.2 Cameras . . . 29
4.2 Pose estimation with Aruco markers . . . 30

4.2.1 Reference frames involved: . . . 30
4.2.2 Reference frame transformations: . . . 30
From Map to Marker . . . 30
From Marker to Camera . . . 31
From camera to base_link . . . 32
4.3 Variance estimation . . . 32
4.3.1 IMU and wheels encoders . . . 32
4.3.2 LiDAR . . . 32
4.3.3 Markers . . . 33
Data collection . . . 33
Interpolation . . . 35
Error propagation . . . 35
4.4 Odometry . . . 36
4.5 SLAM . . . 37
4.6 Localization . . . 38
4.6.1 Without Markers . . . 38
4.6.2 With Markers . . . 38
4.6.3 TF tree . . . 39

5 Implementation and Experiments 41
5.1 Tools and Technologies . . . 41
5.1.1 Programming languages . . . 41
5.1.2 Packages and Libraries . . . 41
5.1.3 Instrumentation . . . 41
5.2 Experimentation and Results . . . 42
5.2.1 Data collection . . . 42
5.2.2 SLAM testing . . . 51
5.2.3 Navigation testing . . . 52

6 Discussion and conclusions 56
6.1 Challenges with real robots . . . 56
6.1.1 Verifiability Issues . . . 56
6.1.2 Robot failures and real-time limitations . . . 56
6.2 Possible Improvements . . . 57
6.2.1 Markers new features . . . 57
6.2.2 Autonomous driving . . . 58
6.3 Conclusions . . . 58

Bibliography 59

Acknowledgements 62

List of Figures

1.1 Accelerator housing . . . 2
1.2 SLAC Robot . . . 4

2.1 Rigid Body . . . 7
2.2 Rotation example . . . 8
2.3 Composition of Rotation about current axes . . . 9
2.4 Composition of Rotation about fixed axes . . . 10
2.5 Euler Angles . . . 11
2.6 RPY Angles . . . 12
2.7 ROS Structure . . . 15
2.8 ROSGraph . . . 17
2.9 ROS frames convention . . . 19
2.10 Pinhole Camera Model . . . 21

3.1 Markers . . . 28

4.1 Camera-Marker frames . . . 31
4.2 Positions of the marker during data collection . . . 33
4.3 Area Normalization . . . 34
4.4 Bilinear Interpolation . . . 35
4.5 SLAM structure . . . 37
4.6 Navigation algorithm . . . 39
4.7 Transformation tree . . . 40

5.1 IMU not calibrated . . . 43
5.2 IMU calibrated . . . 43
5.3 Wheel encoders . . . 44
5.4 Local EKF . . . 45
5.5 Aruco data 1 . . . 46
5.6 Aruco data 2 . . . 47
5.7 Yaw variance . . . 48
5.8 Distance variance . . . 49
5.9 Offset variance . . . 49
5.10 Position estimation with Aruco . . . 50
5.11 LTU map . . . 51
5.12 AMCL initialization . . . 52
5.13 AMCL ambiguity . . . 53
5.14 Rviz visualization of the wrong result . . . 53
5.15 Rviz visualization of the correct result . . . 54
5.16 Result anlysis with Aruco, EKF and amcl . . . 55
5.17 Result analysis with Aruco and EKF . . . 55

List of Tables

4.1 IMU Operation modes . . . 29
4.2 Inputs to the Extended Kalman filter publishing the *odom* → *base_link* transform . . . 37
4.3 Inputs to the Extended Kalman filter publishing the *map* → *odom* transform . . . 38

5.1 Front camera calibration parameter . . . 42
5.2 Rear camera calibration parameter . . . 42
5.3 Right camera calibration parameter . . . 42
5.4 Left camera calibration parameter . . . 42

List of Abbreviations

COTS	**C**ommercial **O**f **T**he **S**helf
CPU	**C**entral **P**rocessing **U**nit
CV	**C**omputer **V**ision
EKF	**E**xtenbded **K**alman **F**ilter
FOV	**F**ield **O**f **V**iew
GPS	**G**lobal **P**ositioning **S**ystem
LTU	**L**inac **T**o **U**ndulator
ICP	**I**terative **C**losest **P**oint
IMU	**I**nertial **M**easurement **U**nit
LiDAR	**Li**dar **D**etection **A**nd **R**anging
PF	**P**article **F**ilter
PnP	**P**erspective-**n**-**P**oint
RANSAC	**Ran**dom **Sa**mple **C**onsensus
REP	**R**OS **E**nhacement **P**roposal
RGB	**R**ed **G**reen **B**lue
RPY	**R**oll, **P**itch, **Y**aw
SLAC	**S**tanford **L**inear **A**ccelerator **C**enter
SLAM	**S**imultaneous **L**ocalization **A**nd **M**apping
SSH	**S**ecure, **Sh**ell
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter

List of Symbols

Symbol	Name	Unit
$R_k(\theta)$	Rotation about k axis by angle θ	-
A	Intrinsic camera matrix	-
σ^2	Variance	-
S_β	Detected area of the marker	px
S_n	Normalized area of the marker	px

Chapter 1

Introduction

1.1 Thesis objective

Nowadays robots are used in a broad diversity of applications. If, on one hand, there is a whole category of static robots used to accomplish repetitive actions, on the other hand there is another category that is gaining interest: the one of mobile robots. Mobile robots can move autonomously in the space and can be really versatile [1].

These robots have become fundamental when it comes to industrial processes, in particular when talking about warehouse and industry logistic. These robots are also used in other fields like the medic and domestic ones, space and underwater exploration an many others.

Autonomous mobile robots have the advantage of being able to navigate in the space and execute tasks without a constant human supervision. This can be done thanks to a whole set of sensors that allow robots to sense the surrounding environment in real-time and act accordingly, avoiding obstacles and making decisions autonomously, adapting themselves to dynamic environments.

These robots are used to explore places that can not be reached by humans or for surveillance of already known environments, executing task such as diagnostic, data collection and material handling.

The characteristic underlying every action that these robots can accomplish is their ability to move autonomously and safely. To achieve this autonomy robots need to map the surrounding environment and be able to navigate inside the map they have created. When it comes to outdoor environment the problem of localization is usually solved using GPS systems, but when there is an indoor navigation to be accomplished or any other kind of GPS-deprived environments, a whole set of sensors is used to help the robot localize itself within the surrounding environment.

Localization and mapping of the surrounding space has been a main theme for the robotic research for many years and it still is, leading to the development of many algorithms.

Nowadays, localization and mapping are two problems that can be solved simultaneously thank to a category of algorithms called SLAM. Once the map has been created, the robot can navigate inside the map everlasting. This second problem is easier and is a localization problem.

This thesis is done in collaboration with SLAC [2]. Accelerator housings are harsh environments for humans due to the high level of radiations when the accelerator is active. For this reason monitoring and troubleshooting operations are accomplished

by autonomous robots that can work in the accelerator housing even when the beam is on. It is very important that tasks are done in the safest way, avoiding any kind of collision with instruments and equipment.

This thesis presents the work that has been done to make the robot available in the laboratory able to navigate inside the accelerator housing safely. In particular, due to the repetitive structure of the indoor environment, see figure **1.1**, the localization inside the acceleration could not be achieved using standard methods. For this reason, a set of fiducial markers called Aruco was used to find a workaround and solve the localization problem.

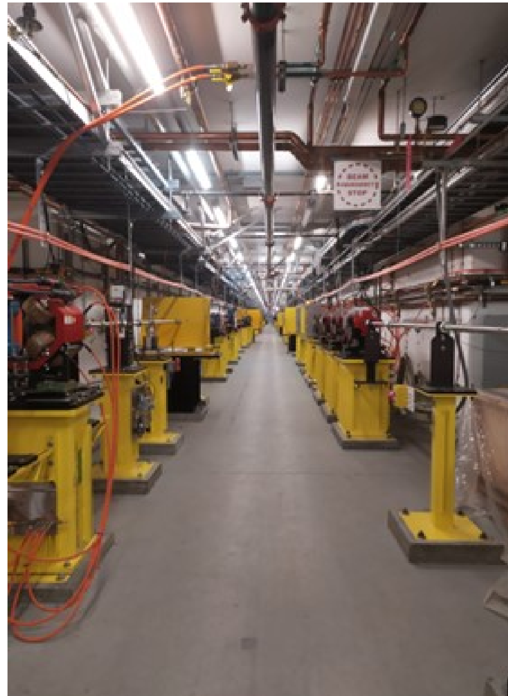

FIGURE 1.1: Accelerator housing and its repetitive environment made of pedestals placed all along the corridor

1.2 Robot

The robot available at SLAC was designed from COTS components and uses the open source framework ROS [3]. It is a remotely controlled robot and is equipped with sensors helpful both for navigation or diagnostic. In figure **1.2** robot's appearance is shown.

1.2.1 Hardware

The main components of the robot's hardware are [4]:

- **Robot Chassis:** The robot chassis is a *Rover Robotics 4WD Rover Pro*, a differential drive ground robot. The two front wheels are driven by brushless motors, while the rear two are connected by belts that synchronize their movement whit the front wheels.

- **Battery:** The robot is powered by a 294Wh lithium-ion battery pack and operates between 12V to 16.8V with over-current protection. The battery life depends on the devices on board and the task performed, but can last up to about 12h while stationary. The battery can be charged using charging docks that can be placed anywhere.
- **Onboard Computers:** To control the robot there is a Nvidia Jetson Tx2 embedded computer dated 2017. Compared to newer technologies this computer is outdated, but anyway it is powerful enough to accomplish the main tasks the robot is built for.
- **Scissor lift:** Mounted on top of the chassis there's a scissor lift that can be adjusted at different heights to align with the accelerator components. On the scissor lift different sensors are installed to perform useful monitoring.
- **Sensors:** Sensors can be divided in two categories:
  - **Navigation:**
    - *Velodyne VLP-16 Puck Lite*: is a 3D LiDAR sensor providing distance data of robot's surrounding. It can measure distances up to 100m and its field of view is 360° with a 2° angular resolution.
    - RGB Cameras : the cameras are 4, one facing each direction. the resolution of the image is 640x360 pixels.
    - IMU: The sensor is a *Bosh Sensortec BNO-055*. It includes an accelerometer, a gyroscope and a magnetometer. It provides linear acceleration, angular velocity and orientation information along the three axes.
    - Limit Switches: Are used for collision detection. The robot is equipped with 8 limit switches, 4 on the front and 4 on the back. They are placed behind transparent bumper and anytime one of them is triggered the robot's e-stop is activated.
  - **Data Acquisition:**
    - Radiation measurement : The *Canberra EcoGamma-G* environmental gamma monitor measures the radiation levels in the accelerator housing.
    - Thermal camera : This sensor is a *Mosaic Core S304NP*. Provides a 320x240 pixel resolution image with calibrated temperature information. Is used to detect components in the accelerator housing that might experience overheating.
    - Extra Camera: Is a *Razor Kiyo* RGB camera. It ss an high resolution camera that acquires 1920x1080 pixel resolution images. It has an integrated microphone and a LED light.

1.2.2 Software

The Jetson TX2 runs Ubuntu 20.04 and ROS is installed. To control the robot remotely, Ubuntu and ROS are set up to provide SSH connection. While the robot collects data and information, a remote computer displays the information. The remote computer serves also for the operator to send commands to the robot, such as

moving it backward and forward, turning left or right, rising or lowering the scissor lift.

FIGURE 1.2: Robot supplied by SLAC

1.3 Thesis structure

The thesis is structured in further five chapters:

Chapter 2 - *Background*:
This chapter provides an overview of key concepts necessary for the correct understanding of this work, with a focus on reference frames and transformation matrices, monocular camera models, perspective projection and ROS basic concepts.

Chapter 3 - *State-of-art*:
This chapter delves into the main localization and mapping approaches for indoor and outdoor environments, focusing on the different SLAM approaches adopted by far for indoor application. The Monte Carlo Localization algorithm is presented as a standard navigation algorithm in a pre-built map. After that, an overview of fiducial markers is done.

Chapter 4 - *Algorithms and Methods*:
This chapter delves into the methodology's core algorithms and techniques. It describes how the sensors have been calibrated, then the chapter presents the math behind the pose estimation retrieved from the detection of fiducial markers and how the covariance of the estimation has been computed. Finally, the ROS packages used for the mapping and navigation processes are presented and an explanation of why and how fiducial markers where used is given.

Chapter 5 - *Implementation and Experiments*:
The chapter presents the results and performances of the method used, highlighting how it improves the standard approach. It also outlines possible further implementation of other features that might be helpful for a better navigation and user experience.

Chapter 6 - *Discussion and Conclusion*: An overview of the challenges encountered during the work is presented, highlighting the limitations and struggles of working with a real robot. Finally conclusions about the work done are drawn.

Chapter 2

Background

2.1 Transformations and transformation matrices

The creation of multiple coordinate systems to capture the locations and orientations of rigid objects, as well as the transformations between these coordinate systems, are important subjects in robot motion. Indeed, the geometry of rigid motions and three-dimensional space is fundamental to all facets of robotic operation.

2.1.1 Pose of a rigid body:

To define a rigid body within a space we need to know its position and orientation, or better, its pose with respect to a predefined reference frame. When it comes to rigid body it is common practice to attach a reference frame to the center of gravity of the body (or another relevant point) and express its unit vectors with respect to another reference frame [5] [6].

The position of a point p' with respect to the coordinate frame *O-xyz* is expressed by:

$$\mathbf{p'} = p'_x\mathbf{x} + p'_y\mathbf{y} + p'_z\mathbf{z} \tag{2.1}$$

Where p'_x, p'_y, p'_z are the components of the vector $\mathbf{p'} \in R^3$ along the axes **x**, **y** and **z** of the *O-xyz* reference frame.

The position of point **p′** can be compactly written in vector form as :

$$\mathbf{p'} = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} \tag{2.2}$$

To express the position of a rigid body we need a set of three equations, describing the position of the three axis of the reference frame attached to it.

$$\begin{aligned} \mathbf{x'} &= x'_x\mathbf{x} + x'_y\mathbf{y} + x'_z\mathbf{z} \\ \mathbf{y'} &= y'_x\mathbf{x} + y'_y\mathbf{y} + y'_z\mathbf{z} \\ \mathbf{z'} &= z'_x\mathbf{x} + z'_y\mathbf{y} + z'_z\mathbf{z} \end{aligned} \tag{2.3}$$

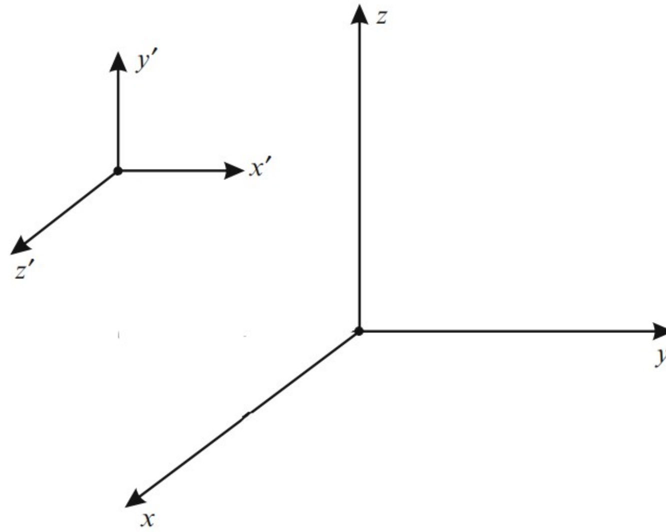

FIGURE 2.1: Position of a rigid body represented by the attached frame *O-x′y′z′* with respect to the reference frame *O-xyz*

2.1.2 Rotation matrix

The set of equations in 2.3 can be rewritten in compact form as:

$$\mathbf{R} = \begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x'_x & y'_x & z'_x \\ x'_y & y'_y & z'_y \\ x'_z & y'_z & z'_z \end{bmatrix} \tag{2.4}$$

This matrix is called *Rotation Matrix* and is peculiarity is that it is an *orthogonal* matrix. This means that all rows and columns are mutually orthogonal and with unit norm. This leads to the following statements:

$$\mathbf{R^T R} = \mathbf{I_3} \tag{2.5}$$

$$\mathbf{R^T} = \mathbf{R}^{-1} \tag{2.6}$$

Where $\mathbf{I}^3$ is the identity matrix in $\mathbf{R}^3$.

Rotation matrices are used to express rotation about any axis. When the rotation is not around one of the three standard axis, the rotation matrix can be decomposed, so that the rotation can be expressed as a combination of rotations about the x, y and z axes.

If a reference frame *O-xyz* is rotated counter-clockwise by an angle θ around the x, the y or the z axes the associated rotation matrices are:

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{2.7}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.8}$$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

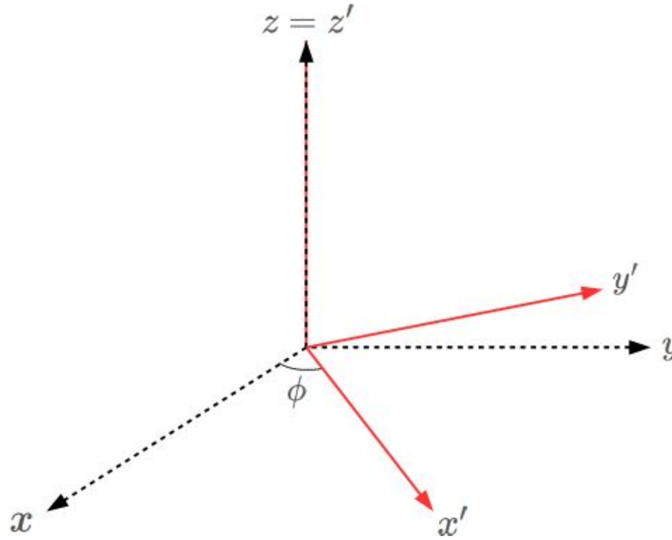

FIGURE 2.2: Example of rotation about z axis [7]

An important feature of rotation matrices is that :

$$\mathbf{R}_k(-\theta) = \mathbf{R}_k^T(\theta) \tag{2.10}$$

Consider a point p and two reference frames O-xyz and O-$x'y'z'$, one rotated with respect to the other around a generic axis of rotation. The point p can be represented in both reference frames as:

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \; or \; \mathbf{p'} = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}$$

Referring to 2.4 we can derive that :

$$\mathbf{p} = \mathbf{R}\mathbf{p'} \tag{2.11}$$

And from 2.6 we obtain that:

$$\mathbf{p}' = \mathbf{R}^\mathbf{T}\mathbf{p} \tag{2.12}$$

Thus, the matrix **R** can be used both to represent the orientation of a frame with respect to another and to transform the coordinates of a point from one frame to the

other. Moreover, as for a point, the matrix **R** can be use to express the rotation of a vector in the space.

Suppose now to have three reference frames with a common origin O, $O - x_0y_0z_0$, $O - x_1y_1z_1$ and $O - x_2, y_2, z_2$. The position of a point p in the space can be expressed in each reference frame as p^0 , p^1 and p^2. Since a rotation matrix represents the rotational transformation from one frame of reference to another, we can use the notation $\mathbf{R_1^0}$ to represent the rotation from frame $O - x_1y_1z_1$ to frame $O - x_0y_0z_0$. In this way the following relationship held:

$$p^0 = R_1^0 p^1$$
$$p^1 = R_2^1 p^2$$
$$p^0 = R_2^0 p^2$$

It follows that:

$$p^0 = R_1^0 R_2^1 p^2$$

This means that to transform the coordinate of a point from frame $O - x_2y_2z_2$ to $O - x_0y_0z_0$ we can pass through the middle frame $O - x_1y_1z_1$.

$$R_2^0 = R_1^0 R_2^1 \quad (2.13)$$

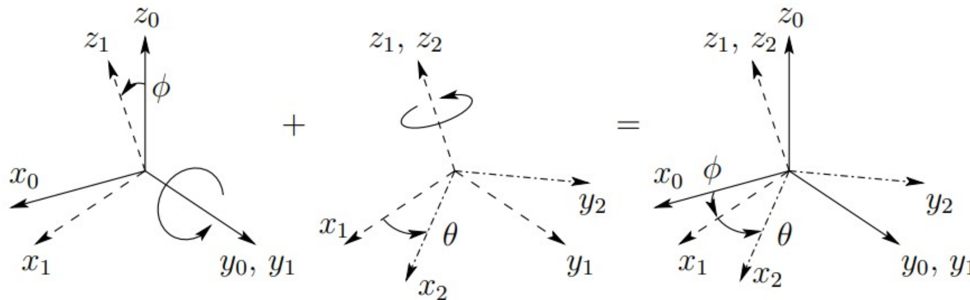

FIGURE 2.3: Composition of rotation about current axes [5]

Thus, the whole rotation can be decomposed into two smaller rotations. We can also interpret the relation 2.13 as if, starting from frame O_0 we rotate according to $\mathbf{R_0^1}$ to align with frame O_1 and then, once aligned, we rotate again according to $\mathbf{R_2^1}$. Anyway, what we are doing is rotating about the current frame. The order of multiplication is really important since matrix multiplication is not commutative.

If it is necessary to perform consecutive rotation keeping the original reference frame as rotation reference, the order of multiplication must be inverted!

2.1.3 Parametrizations of rotation:

Using the composition of rotations, every kind of rotation can be represented using matrices. Anyway, using matrices means involving 9 parameters. But, since a rigid body can have at most three degrees of freedom when it comes to rotations, only 3

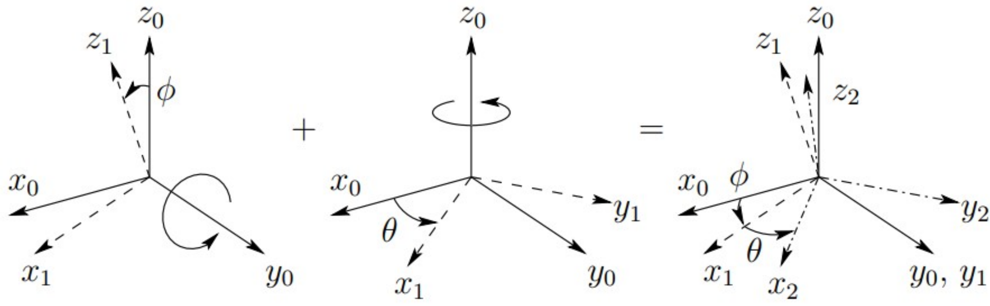

FIGURE 2.4: Composition of rotation about fixed axes [5]

parameters are needed, so rotation matrices are not the most efficient way of representing rotations. Many other equivalent representations exist and they can be used as preferred.

ZYZ Angles

The rotation described by the *ZYZ Angles* is a composition of three elementary rotations about current frames.

- Rotation by angle ϕ about axis z
- Rotation by angle θ about current axis y
- Rotation by angle ψ about current axis z

$$\mathbf{R}(\mathbf{\Phi}) = \mathbf{R}_z(\phi)\mathbf{R}'_y(\theta)\mathbf{R}''_z(\psi) = \begin{bmatrix} c_\phi c_\theta c_\psi - s_\phi s_\psi & -c_\phi c_\theta s\psi - s_\phi c_\psi & c_\phi s_\theta \\ s_\phi c_\theta c_\psi + c_\phi s_\psi & -s_\phi c_\theta s\psi + c_\phi c_\psi & s_\phi s_\theta \\ -s_\theta c_\psi & s_\theta s\psi & c_\theta \end{bmatrix} \tag{2.14}$$

If, given the following rotation matrix

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.15}$$

the *ZYZ angles* representation is needed, the formulas to use are:

$$\begin{aligned} \phi &= Atan2(r_{23}, r_{13}) \\ \theta &= Atan2(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}) \\ \psi &= Atan2(r_{32}, -r_{31}) \end{aligned} \tag{2.16}$$

The square root limits the value of θ to $(0,\pi)$. There is a singularity problem when $s_\theta = 0$. In this case ϕ and ψ cannot be determined, their sum or difference can though.

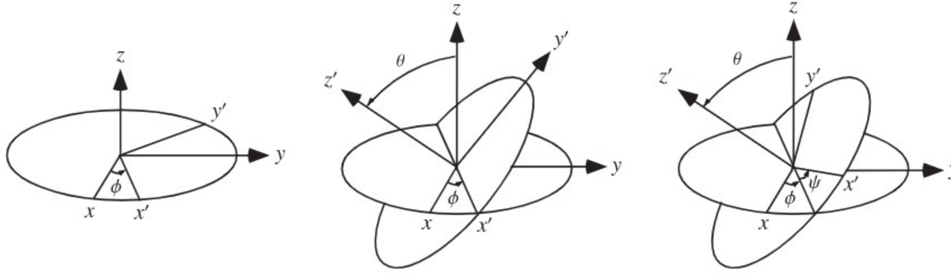

FIGURE 2.5: Representation of Euler Angles [8]

RPY angles

The rotation described by the *RPY Anlges* is a composition of three elementary rotations about fixed frames.

- Rotation by angle ψ about fixed axis x - Roll
- Rotation by angle θ about fixed axis y - Pitch
- Rotation by angle ϕ about fixed axis z - Yaw

The composition of these rotations gives :

$$\mathbf{R}(\mathbf{\Phi}) = \mathbf{R}_z(\phi)\mathbf{R}_y(\theta)\mathbf{R}_x(\psi) = \begin{bmatrix} c_\phi c_\theta & c_\phi s_\theta s_\psi - s_\phi c_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ s_\phi c_\theta & s_\phi s_\theta s_\psi + c_\phi c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi \\ -s_\theta & c_{\theta s}\psi & c_\theta c_\psi \end{bmatrix} \tag{2.17}$$

If, given the following rotation matrix

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.18}$$

the *RPY angles* representation is needed, the formulas to use are:

$$\begin{aligned} \phi &= Atan2(r_{21}, r_{11}) \\ \theta &= Atan2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}) \\ \psi &= Atan2(r_{32}, -r_{33}) \end{aligned} \tag{2.19}$$

The square root limits the value of θ to $(-\pi/2, \pi/2)$. There is a singularity problem when $c_\theta = 0$. In this case ϕ and ψ cannot be determined, their sum or difference can though.

RPY and ZYZ Angles use the minimum amount of parameters to represent rotations, but they are both subject to singularities. This two type of representation go under the name of *Euler Angles*.

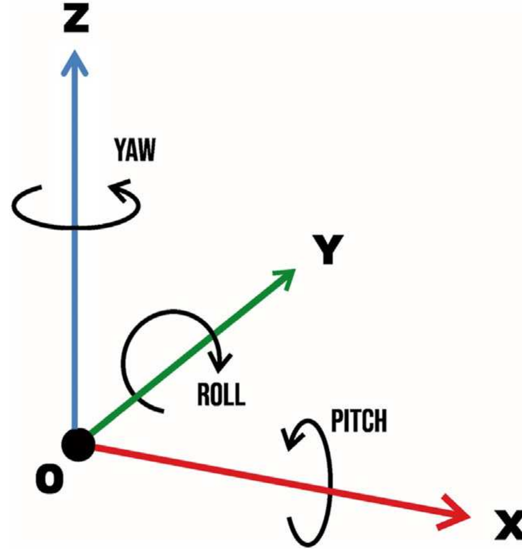

FIGURE 2.6: Representation of RPY Angles [9]

Angle-Axis

The angle-axis representation uses four parameter to represent a rotation by angle θ around an axis of rotation defined by the vector $\boldsymbol{r}$=$[r_x r_y r_z]^T$

The corresponding rotation matrix is:

$$\mathbf{R}(\mathbf{\Phi}) = \begin{bmatrix} r_x^2(1-c_\theta)+c_\theta & r_xr_y(1-c_\theta)-r_zs_\theta & r_xr_z(1-c_\theta)+r_ys_\theta \\ r_xr_y(1-c_\theta)+r_zs_\theta & r_y^2(a-c_\theta)+c_\theta & r_yr_z(1-c_\theta)-r_xs_\theta \\ r_xr_z(1-c_\theta)-r_ys_\theta & r_yr_z(1-c_\theta)+r_xs_\theta & r_z^2(1-c_\theta)+c_\theta \end{bmatrix} \tag{2.20}$$

This representation is not unique since $\mathbf{R}(-\theta,-\mathbf{r}) = \mathbf{R}(\theta,\mathbf{r})$

If, given the following rotation matrix

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.21}$$

the *Angle-Axis* representation is needed, the formulas to use are:

$$\theta = \cos^-1\frac{r_{11}+r_{22}+r_{33}-1}{2} \tag{2.22}$$

$$r = \frac{1}{2\sin\theta}\begin{bmatrix} r_{32}-r_{23} \\ r_{13}-r_{31} \\ r_{21}-r_{12} \end{bmatrix} \tag{2.23}$$

$$r_x^2 + r_y^2 + r_z^2 = 1 \tag{2.24}$$

When $\sin\theta = 0$ a singularity occurs.

Quaternion

Quaternions are another way of representing rotations with four parameters. A quaternion can be represented in the form $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ where a, b, c, d are real numbers, while $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are the basis vectors.

Knowing the angle-axis representation of a rotation (θ, r_x, r_y, r_z) it is possible to convert the representation into a quaternion (q_0, q_1, q_2, q_3) using the following formulas:

$$q_0 = \cos\frac{\theta}{2}$$
$$q_1 = r_x \sin\frac{\theta}{2}$$
$$q_2 = r_y \sin\frac{\theta}{2}$$
$$q_3 = r_z \sin\frac{\theta}{2}$$

That can be written as $q = \cos\frac{\theta}{2} + \mathbf{r}\sin\frac{\theta}{2}$

If, given the following rotation matrix

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.25}$$

the *Angle-Axis* representation is needed, the formulas to use are:

$$q_0 = \frac{1}{2}\sqrt{r_{11} + r_{22} + r_{33} + 1} \tag{2.26}$$

$$\begin{bmatrix} q1 \\ q2 \\ q3 \end{bmatrix} = \begin{bmatrix} sgn(r_{32} - r_{23})\sqrt{r_{11} - r_{22} - r_{33} + 1} \\ sgn(r_{13} - r_{31})\sqrt{r_{22} - r_{33} - r_{11} + 1} \\ sgn(r_{21} - r_{12})\sqrt{r_{33} - r_{11} - r_{22} + 1} \end{bmatrix} \tag{2.27}$$

for $\theta \in [-\pi, \pi]$.
Quaternion multiplication correspond to rotation matrix multiplication and, as for matrices, is not commutative. Multiplication is defined in this way :

$$\mathbf{t} = \mathbf{rs}$$

$$(t_0, t_1, t_2, t_3) = (r_0, r_1, r_2, r_3) \times (s_0, s_1, s_2, s_3)$$
$$t_0 = r_0 s_0 - r_1 s_1 - r_2 s_2 - r_3 s_3$$
$$t_1 = r_0 s_1 - r_1 s_0 - r_2 s_3 - r_3 s_2$$
$$t_2 = r_0 s_2 - r_1 s_3 - r_2 s_0 - r_3 s_1$$
$$t_3 = r_0 s_3 - r_1 s_2 - r_2 s_1 - r_3 s_0$$

The quaternion corresponding to $\mathbf{R}^{-1}$ is equivalent to $(q0, -q1, -q2, -q3)$. All rotation quaternion must be unit quaternions, that means that |q| = 1

2.1.4 Homogeneous Transformation:

By now, we extensively addressed the problem of describing the rotation of a frame with respect to the other, supposing thet the two frames have origin in the same position. As we have seen, reference frames can be attached to rigid bodies to describe their position in the space, so talking about the position of a frame with respect to the other is actually the same as talking about the pose of a rigid body. If we have a point $\boldsymbol{p}$ in the space this point can be described as $\mathbf{p^0}$ with respect to the reference frame $O - x_0 y_0 z_0$ and as $\mathbf{p^1}$ with respect to the reference frame $O - x_1 y_1 z_1$. Defining $\mathbf{o_1^0}$ the vector that describes the origin of frame O_1 with respect to O_0 and $\mathbf{R_1^0}$ the rotation of frame O_1 with respect to O_0 we can say that :

$$\mathbf{p^0} = \mathbf{o_1^0} + \mathbf{R_1^0 p^1} \tag{2.28}$$

$$\mathbf{p^1} = -\mathbf{R_0^1 o_1^0} + \mathbf{R_1^0 p^0} \tag{2.29}$$

In order to have a compact notation, called *homogeneous representation* the vector $\mathbf{p}$ needs to be extended to $\tilde{\mathbf{p}}$.

$$\tilde{\mathbf{p}} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \tag{2.30}$$

Formula 2.28 can be rewritten in compact form using:

$$\mathbf{T_1^0} = \begin{bmatrix} \mathbf{R_1^0} & \mathbf{o_1^0} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.31}$$

In this way 2.28 becomes :

$$\tilde{\mathbf{p}}^\mathbf{0} = \mathbf{T_1^0} \tilde{\mathbf{p^1}} \tag{2.32}$$

And 2.29 becomes :

$$\tilde{\mathbf{p}}^\mathbf{1} = \mathbf{T_0^1} \tilde{\mathbf{p}}^\mathbf{0} = (\mathbf{T_1^0})^{-1} \tilde{\mathbf{p^0}} \tag{2.33}$$

Where

$$\mathbf{T_0^1} = \begin{bmatrix} \mathbf{R_1^0}^T & -\mathbf{R_1^0 o_1^0} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.34}$$

For this matrix, called *homogeneous transformation matrix* the orthogonality does not hold.

2.2 ROS - Robotic Operating System

ROS is an open source framework for robot programming. It's a meta-operating system operating as a middle-ware, functioning as an intermediate layer between the Operating System and robotic applications. It provides a diversity of functionality such as hardware abstraction, package management, device drivers, tools and libraries, processes management, data logging, and much more.

Even though the acronym stands for Robotic Operating System, ROS is not an operating system. ROS is usually targeted to work with Ubuntu, but other operating systems such as Windows and MacOS are supported to varying degrees.

Low latency in robotics application is an important feature, ROS is not a real time operating system though. For this reason a second version of ROS, ROS2 has been developed in recent years to address this and other issues.

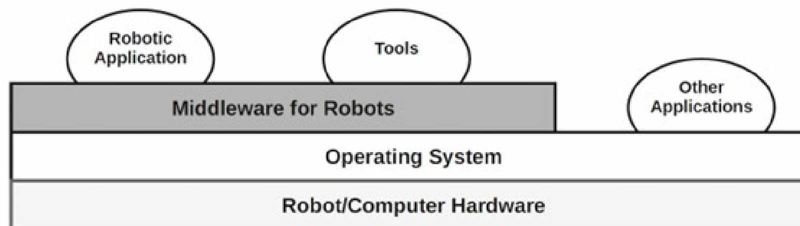

FIGURE 2.7: ROS structure [10]

The key feature of ROS are [11]:

- **Open Source** : Together with the ROS platform, also the majority of codes written in ROS are open source. Being open source is one of the key idea behind ROS philosophy. In this way, users can use packages and tools to create the main structure of their own robot is a small amount of time, focusing on the key feature of their new application using an existing foundation.
- **Multi-Lingual:** The officially supported programming languages are C++ and Python, which are the mostly used in robotic applications. However, ROS modules can be written in any language for which a client library exist. However, in some cases it might be easier to add support for a different language wrapping an existing library.
- **Flexibility:** ROS programs do not need to run on the same system or architecture. Multiple devices, each of them with their own architecture are all connected to each other.
- **Peer-to-Peer:** Some robotic systems carry multiple computers on board and have also offboard computers running computation-intense tasks. All these computers are connected through a peer-to-peer topology. Since the system is usually divided into independent modules, many message routes are entirely contained inside a subnet. For this reason running a central server either onboard or offboard would lead to unnecessary traffic.
- **Tools-based:** The complexity of a robot using ROS is simplified by the usage of different small tools that can be built and ran independently. The decision of using separate modules instead of a monolithic development makes the structure more manageable by the user.
- **Thin:** Drivers and algorithms do not need any ROS dependencies. Libraries can be wrapped so that they can communicate with other ROS modules. The use of CMake builds the code module by module inside the source code tree. This makes the code reusable and easier to extract.
- **Rapid testing:** The modular structure makes the debugging easier: a new node can run alongside well debugged nodes. In this way, the user can easily isolate

the behaviour of the new node and test it. Moreover testing can be time consuming and a robot may not always be available. Working with ROS allows to record and play sensor data or any other kind of message type using the command *rosbag*. These recordings can be replayed anytime to do intensive tests using always the same collection of data without the need of having the physical robot.

2.2.1 Graph Structure:

The peer-to-peer structure is easier to picture if ROS is saw as a graph structure with nodes and arcs. On top of everything there is a main node called ROS Master that manages the peer-to-peer communication between nodes. To do this, nodes need to register at startup with the ROS Master.
The fundamental concepts of the ROS implementation are:

- **Nodes:**
  A running instance of a ROS program is called 'Node'. Each node is a piece of code that is compiled, executed and managed individually. Every node has a name and is registered to the ROS Master. Nodes can have the same name if they are under different *namespaces* or an additional random identifier can be assigned to make it unique. A node can send and receive information or requests for action to/from other nodes. Nodes can send and receive messages to/from another node using topics, can provide services for other nodes or retrieve data from a common database called parameter server.
- **Messages:**
  A message is a strictly typed data structure. Messages can be of a broad variety of types and can be also user-costumized. Typically messages represents sensor data, commands, state information or anything else.
- **Topics:**
  Topics are buses that drive messages from one node to the other. The name of the topic must be unique within the *namespace* as well as nodes. To send a message to a topic, the node needs to publish on that topic, and if a node wants to receive messages from a topic, it must subscribe to it. Usually, one topic has one publisher and *n* subscribers, but it's not mandatory and more than one node can publish on the same topic.
- **Services:** When a node wants to broadcast a message, topics are used. But when it comes to one-to-one communication, services are more appropriate. A service is a synchronous procedure call that allows a node to call a function that executes in another node. They are useful for functions that are needed occasionally. Only one node can advertise a particular service.

2.2.2 ROS commands:

- **roscore:** It is the command used to initialize the ROS master and the ROS environment. The master is the node that manages communication between nodes. When roscore is run the environment variable ROS_MASTER_URI is set to *http://hostname:11311* by default. This means that there is a running instance of roscore running on port 11311 on a host called hostname. This variable can be

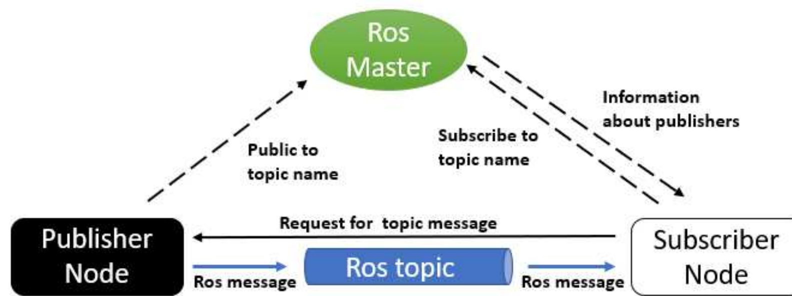

FIGURE 2.8: Basic ROS graph with a simple publisher and subscriber [12]

modified, changing the port and the hostname so that multiple ROS systems can coexist on a single network.

When a new node is fired up, ROS_MASTER_URI variable gives information about where to find the master node so that the new node can register to it, telling which messages it provides and which it would like to subscribe to. The master provides the information needed to form a peer-to-peer communication.

roscore provides also the *parameter server* where data structures are stored and made available to nodes. The parameter server can be accessed with the *rosparam* command.

- **rosrun:** ROS is organized in packages that are collections of resources built together. The command *rosrun* search for the package location in the file system and passes any argument called by the command line. The syntax is the following:

  rosrun package_name [ARGS]

- **roslaunch:** A robotic system can consist of hundreds of nodes. Running *rosrun* for each node would not be handy. *roslaunch* is a tool that allows the launching of more than one node simultaneously, setting parameter for each one. The syntax is the same of rosrun, but a file name is given instead of the node name. The file must be a *.launch* file, that is a XML code that describes all the nodes that need to be launched together with their name, their parameters, and their remapped topics. Running this command will initialize automatically a ROS master.

- **rostopic:** It is used to inspect topics and messages. It has different sub commands such as list, that prints the list of the active topics; echo, that prints messages to screen; info, that displays information about the message type and who is publishing/subscribing to that topic.

2.2.3 Transform Tree:

A robot is usually composed of different pieces assembled together. For example, a robot might be running on 4 wheels, might have a sensor mounted on top, might have a robotic arm or whatever. Tracking the spatial relationship between robot's components but also the relation between the robot and the environment is fundamental to accomplish many tasks.

The problem is addressed by the *tf* [13] package. The *tf* package constructs a dynamic transform tree that relates all frames of reference in the system. Any node can publish information about some transform through the */tf* topic and any node can subscribe to this topic to retrieve information. The message inside the *tf* topic contains a string with the name of the two reference frames we are transforming and the time associated with the transform, together with the transform vector. An important rule of the transformation tree is that each node of the tree can have only one parent, creating, as a matter of fact, a tree structure.

2.2.4 Coordinate Frame for Mobile Platforms:

ROS creators have written a series of REP to provide information to the ROS community, such as naming conventions or description of new features.

REP-103 [14] and REP-105 [15] address the problem of coordinate and naming conventions when it comes to coordinate frames for mobile platform.

- **Axis orientation:** In relation to a body the standard is:
  - x forward
  - y left
  - z up
- **Suffix Frames:** In the case of cameras there is an 'optical' reference frame that uses a different orientation convention:
  - z forward
  - x right
  - y down
- **Rotation Representation:** The convention depends on which representation is chosen.

  Quaternion:
  - Compact representation
  - No singularities

  Rotation matrix:
  - No singularities

  Fixed RPY:
  - No ambiguity on order
  - Used for angular velocities

  Euler Angles:
  - Usually discouraged

  By the right end rule, yaw increases when the body rotates counterclockwise.
- **Covariance Representation:**
  - 3 Dimensional: 3x3 row major matrix in x,y,z

- 6 Dimensional: 6x6 x, y, z, rotation about x axis, y axis, z axis

- **Coordinate frames:** When it comes to Mobile Robots the reference frame naming convention are:
  - *base_link:* This coordinate frame is attached to the robot base in any arbitrary position, even though is usually placed in the center.
  - *odom:* Is a world fixed frame. The pose of the robot in this frame is continuous, even though can drift over time from the real pose. For this reason this frame is useful for short-term global reference, but not for long-term navigation.
  - *map:* Is a world fixed frame. The pose of a mobile robot in this reference frame is not continuous, this means that discrete jumps can occur over time. Usually, a localization component computes the transform of the mobile base in the *map* frame, eliminating the drift that can occur in the *odom* frame. For this reason, the *map* frame is useful as a long-term reference.
  - *earth:* This frame is also referred as *world* frame and it is present only if there is more the one map. For each map a complete transformation tree from *map* to *base_link* frame must be defined.

By convention the *map* frame is the parent of *odom* and *odom* is the parent of *base_link*. The transform *odom* $\rightarrow$ *base_link* is broadcast by only one odometry source. The transform *map* $\rightarrow$ *base_link* is broadcast by a localization source that actually publishes the *map* $\rightarrow$ *odom* transform, since *base_link* can have only one parent in the transformation tree.

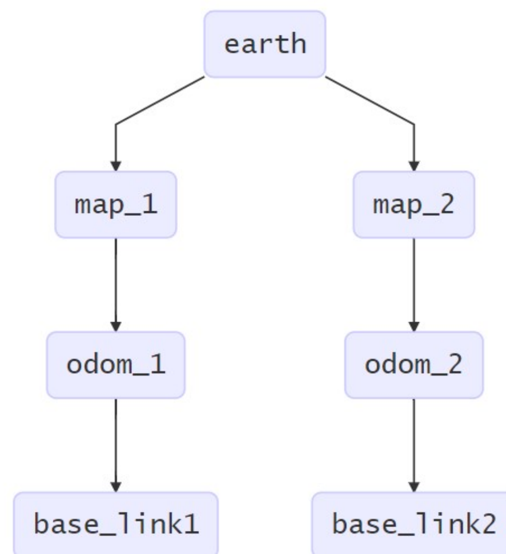

FIGURE 2.9: Standard transformation tree for mobile robots [15]

2.2.5 Rviz:

Rviz [16] is a powerful visualization tool. It subscribes to the topics and the transformation tree and allows the visualization of data. Thanks to the most common

plugins it is possible to visualize grid maps, robot models, the transformation tree, images viewed form cameras and point clouds collected by LiDARs. In this way the user can have a view of what the robot is seeing, sensing and doing. Moreover Rviz has a nice user interface the allow the user to set the initial pose of a localization system or to set a goal pose.

2.3 Camera Model

Cameras are one of the most used sensors in robotics since they are the most human way to record the environment around the robot. As every other sensor, a model for the camera is needed and different models exist. The most common is the *Pinhole camera model*.

2.3.1 Pinhole camera model

The *pinhole camera model* is a monocular camera model. It simplifies the complex model of a real camera approximating it to a single point in space. No lenses are included in the model and this leads to no distortion model. It's the easiest way of representing the relation between a true three dimensional object in the space and its two dimensional projection onto the image. A model for the camera and the estimation of the parameters that characterize the model are fundamental to calibrate the camera and use the model for reconstructing 3D objects.

The ingredients of this model are

- **Optical center:** Is the point where the camera is placed in the model and where all the rays of light converge.
- **Image plane:** Is the virtual plane where the object's projection lies.
- **Optical axis:** Is the Z axis of the camera model pointing forward.
- **Focal length:** The focal length is the shortest distance between the image plane and the optical center.
- **Principal point:** Is the intersection of the image plane and the optical axis.

2.3.2 Camera calibration and pose computation:

The function that transforms a 3D point $\mathbf{P_c}$ in the camera coordinate into pixel coordinates $\mathbf{p} = (\mathbf{u}, \mathbf{v})$ is called perspective transformation. Without taking into consideration any distortion the simplified transformation is [18][19]:

$$s\mathbf{p} = \mathbf{A}\mathbf{P_c} \tag{2.35}$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{2.36}$$

Where $\mathbf{A}$ is the so-called *intrinsic camera matrix* and $\mathbf{P_c}$ is a whatever point in the camera space, that can be represented in the world frame using an appropriate roto-translation matrix.

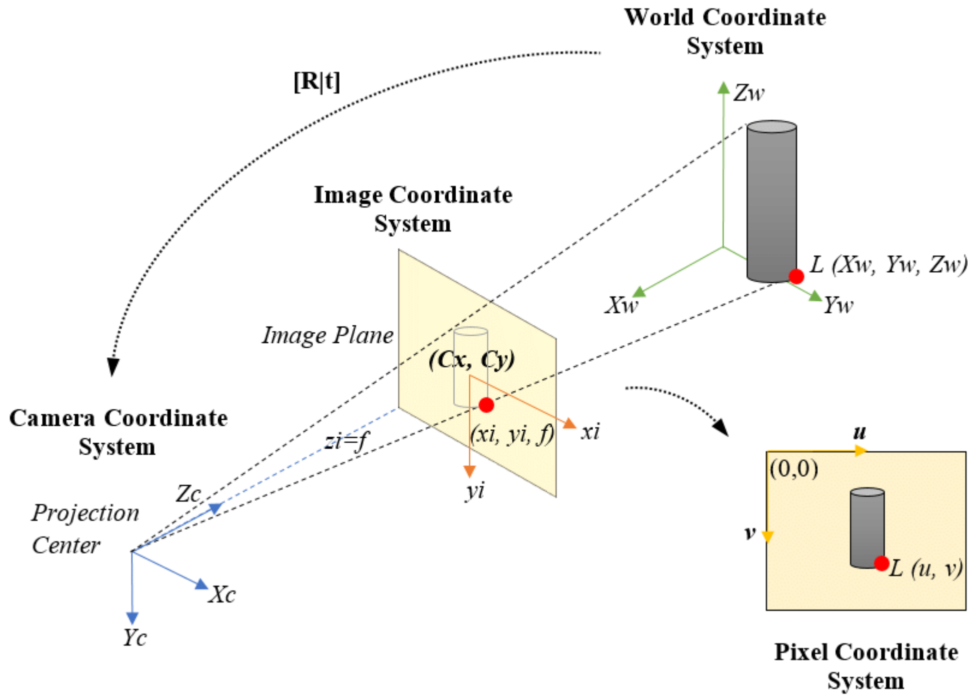

FIGURE 2.10: Pinhole Camera Model [17]

Moreover we have that the transformation that maps 3D point into 2D points in the image plane and in normalized camera coordinates is:

$$Z_c \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{2.37}$$

Combining the three formulas we obtain that :

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.38}$$

If $Z_c \neq 0$ we obtain:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x X_c / Z_c + c_x \\ f_y Y_c / Z_c + c_y \end{bmatrix} \tag{2.39}$$

The pinhole mode actually does not take into account any kind of distortion, but in reality lenses of the cameras usually have two type of distortion: radial and tangential. Radial distortion makes straight line far from the center of the camera appear curved. Tangential distortion is introduced because the image plane and the lens are not aligned.

To remove both distortion from the image a twelve-parameter model $(k_1, k_2, k_3, k_4, k_5, k_6, p_1, p_2, s_1, s_2, s_3, s_4)$ is used:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x x'' + c_x \\ f_y y'' + c_y \end{bmatrix} \tag{2.40}$$

Where

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} x' \frac{(1+k_1 r^2 + k_2 r^4 + k_3 r^6)}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x'y' + p_2(r^2 + 2x'^2) + s_1 r^2 + s_2 r^4 \\ y' \frac{(1+k_1 r^2 + k_2 r^4 + k_3 r^6)}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_2 x'y' + p_1(r^2 + 2y'^2) + s_3 r^2 + s_4 r^4 \end{bmatrix} \tag{2.41}$$

And $r^2 = x'^2 + y'^2$.

To estimate the set of parameters necessary to perform the transformation the camera must be calibrated. Different tools exist to perform calibrations of cameras, such as the one offered by the open-source library *OpenCV* or *Matlab*. The *OpenCV* calibration method uses a simplified model with only five parameters: $(k_1, k_2, k_3, p_1, p_2)$

Once the camera is calibrated it is possible to use this pinhole camera model to estimate the pose of objects captured by the camera. To do this, the *OpenCV* library is helpful again. The library offers different algorithm to solve the problem of pose computation.

Chapter 3

State of art

3.1 SLAM

SLAM problem is the problem of building a map of the environment and simultaneously localize the robot inside this map. Before the advent of SLAM algorithms the problems of localization and mapping where faced separately but now the SLAM approach is the one used. Since measurements coming from sensors are affected by noise, all SLAM algorithms are based on probability [1].

Initially the probabilistic models were based on Bayes filters such as Extended Kalman or Particle Filters, then graph based approaches arrived, overcoming some weaknesses of filter-based approaches. To understand how SLAM works, few related concepts need to be introduced:

3.1.1 Related concepts

- **Grid map**

  Grid maps are a way to represent the surrounding environment in 2D. Te map is subdivided in different cells and to each one a value is associated. Each cell is a binary random variable, but usually the values are three: 1,0,-1. The value assigned is the one with higher probability. When the value associated with the cell is 1, that means that there is an obstacle that can be an object or a wall, then, if it is 0, that means that the space is empty and the robot can navigate that space, if it is -1 that mean that the space is unexplored and there is no information about it. [20]

- **Filters**

  When it comes to SLAM algorithms two kind of filter are used: Extended Kalman Filter and Particle Filter.

  - **Extended Kalman Filter:**

    The Extended Kalman Filter is the non-linear version of the standard Kalman Filter.
    The algorithm takes into account a series of measurements together with their covariance, over time. Measurements can come from different sensors and the result is obtained by a joint probability distribution. The covariances are supposed to be zero mean multivariate Gaussian distributions.

The algorithm is developed in two steps: Prediction and Update phase.
In the prediction phase the filter predicts an estimate of the state variables based on how the system has behaved up to that time.
Then, once a new set of measurements arrive, the estimate is compared with the predicted estimate and the result is updated using a weighted average.

EKF is not an optimal estimator, because it is not linear and if the initial estimate is wrong or the model is not correct the result can diverge [21].

– **Particle Filter:**

Particle Filter are also know as *sequential Monte Carlo*. It is used in non-linear problems to find an approximate solution when information coming from sensors are partial and corrupted by noise.

The result comes from the computation of *a posteriori* distribution, then, this distribution is used to generate a set of samples together with their likelihood that are called *particles.*

When the probability of a particle goes to low, the particle is replaced with another one closer to the particles with higher likelihood. This kind of filter does not have good performances when it comes to high dimensional systems [22].

- **Dead Reckoning**

  The SLAM algorithm make use of data coming from other sensors to estimate the position of the robot, incorporating estimates of speed and time. The estimated position is with respect to a starting point, so we can say that it is a local, but not global estimation, since the position is not estimated with respect to a map. Since there is no global reference and no way to correct error over time, this method is not suitable for long term navigation [23]. Usually, to make the estimate more accurate the estimation is made from more than one sensor and the measurements coming from all these sensors are generally fused together through an Extended Kalman Filter.

3.1.2 Sensor-based classification

The only way robots can perceive the environment is through sensors. According to what sensors are available the way robots can interact with the environment, store information and process information is different. When speaking about SLAM the two sensor that can help a robot to store information about surrounding dimensional space are cameras and LiDARs.
For this reasons SLAM can be divided in two main categories [24][25]:

- **Visual-SLAM**

  Visual SLAM uses cameras to extract features from the surrounding environment. Cameras can be monocular, stereo or RGB-D cameras: for each of them different algorithms exist. In some cases features are extracted from images and sequential images are compared to see how these features move in the images to extract velocity information. Another approach is to compare the whole images without extracting any feature. In the case of RGB-D camera the information about the depth is really important for computing distances.

Usually, image processing is computational expensive and, more important, the precision of this methods really depends on how many features are present in the images and the resolution of the camera. Another disadvantage is that cameras do not work with in the darkness or if there is dust.

- **Lidar-SLAM**

  LiDAR sensors are more expensive than cameras, but the higher price is compensated by a really higher accuracy in distance computation. LiDARs work by sending beams of light in different directions and measuring the time it takes for the beam to be reflected by an object and come back. In this way distances are determined. 2D or 3D Lidar exists, ranging 360° degree or less. All these distance measurements put together form a so-called *pointcloud* that is used by SLAM algorithms to estimate robot's pose. Usually the algorithm used is a scan-matching approach that consists in comparing two consecutive scans of the surrounding environment and from the comparison determine the transformation that makes the two scans match as much as possible.

  Unlike cameras, LiDARs can work in the darkness or in presence of dust. The disadvantage is that this sensor requires more processing power than cameras and the algorithms might be slower.

3.1.3 ROS open source packages for SLAM

- **Gmapping (2007)**

  Gmapping toolbox [26] is based on a Particle Filter. The result of the filter is a set of particles that stand for possible robot's poses. The a posteriori distribution is computed taking into account the likelihood of the pose computed with scan-matching procedure together with odometry information coming from other sensors.

  Due to the usage of the particle filter, it struggles in large environments. In complex and intricate environments the toolbox succeed but only if a increased number of particles are use, leading to a higher computation cost.

- **Karto (2010)**

  slam_karto toolbox [27] Is a Graph-based SLAM algorithm, structured in the same way as Cartographer. The map is updated every time a new node is added to the graph. At every node registration an optimal map is computed and when the robot goes back to a previous position a loop closure algorithm is provided. The algorithm is based on the decomposition of Cholesky matrices to minimize the error.

- **Hector_mapping (2011)**

  The Hector_mapping toolbox [28] is a LiDAR-based approach and does not make use of odometry information coming from other sensors because it uses the Newton-Gauss method. For this reason it is necessary for LiDAR's data to arrive at very high speed. It can be used in small scenarios where large loops do not exist, because it does not provide any explicit algorithm for loop closure. When the environment is big it struggles to succeed.

If there is no reliable source of odometry data, it can be a good choice. Since it does not use any odometry information it can accumulate drift over time which can lead to inaccurate map generation.

- **Google Cartographer (2016)**

  Google Cartographer [29] is a graph-based SLAM. The graph is composed of nodes and edges, where nodes are saved position of the robot and links are the movement between each position. It has a front-end working to scan match the LiDAR with the map that has been built and to trace the trajectory and a back-end that checks for loop closures using the Ceres Solver.

  Requires more data storage and faster CPU than the other packages, but shows better results. Has a localization node if a map already exists and provides data serialization for storing maps. Unfortunately, Google abandoned the project and nowadays there is no maintenance and support for this package.

- **Slam_toolbox (2019)**

  The slam_toolbox [30] offers two different types of nodes, built from two different code sources: synchronous and asynchronous. Despite the other packages this one offers a variety of useful tools such as the ability of merging together different maps created in different sessions, multi session mapping, lifelong mapping and pose-graph manipulation. Nowadays is the most used package for SLAM and was integrated in the *Navigation2* project. It can serialize a mapping session and deserialize it later for further mapping. Has a Rviz plugin to assist the user doing operations such as manual pose graph manipulation to close challenging loops.

  Synchronous mapping keeps a buffer of the measurements. It is suited when the quality of the map is important or when doing offline processing. Asynchronous mode will process new data only when the last one is completed, this leads to lagging in real-time if there are difficult loop-closures. The localization node has a temporary buffer of measurement that are matched against the pose-graph computed while mapping. The new measurements are added to the original pose-graph only temporarily.

  According to the presentation paper, it is 10x faster then Karto thanks to multithread processes. The sparse Pose adjustment was replaced with google Ceres, that is faster and has more flexible optimization settings and other technical improvements.

3.2 Monte Carlo Localization

Once the map has been built the robot can navigate inside it. This problem is different from SLAM because the map is now available so the name for this new problem is simply localization. To solve this problem the standard approach when a LiDAR is available is the Adaptive Monte Carlo algorithm that is implemented in the *amcl* [31] open source package for ROS.

When the algorithm starts, the particles indicating the belief of where the robot can be are spread uniformly over the entire map. As the robot senses the environment and find characteristic features importance factors are assigned to each particle. The result is incorporated with the robot motion sensed by other sensors, contributing to

giving importance to some particles with respect to others. The feature extraction is done through scan-matching: at each time sampling, the scan made by the LiDar is compared with the map available.

The accuracy of the estimation can be seen looking at how many particles are concentrated in the same point.

To overcome the problem of kidnapping or wrong result, a set of particle is add every now and then in the probability distribution to simulate the remote probability that the robot was moved by a human operator or that the estimated position is wrong. The algorithm keeps track of long and short term likelihood of the measurement and during the resampling process new particles are added depending on whether the short-term likelihood is higher or lower then the long term likelihood [32].

3.3 Fiducial Markers

Fiducial markers are commonly used in robotics for labeling and localization when natural features such as doors, corners and other objects are not sufficient and unique. They are cheap and easy to use since they can be recognized using a simple monocular camera [33][34].

A fiducial markers has a simple and unique pattern that comes together with a fast algorithm for recognition. The pattern can go from simple circles to more sophisticated bar codes and can be of different shapes and colours. Black and white markers are easier to recognize.

Thank to these markers it is possible to estimate, through some geometric transformations the position of the robot with respect to the markers. With the algorithms developed until today, such as PnP methods, it is important that at least four relevant point in the marker can be extracted. For this reason, a squared shape is the most reasonable, also because it makes easier the computation due to the symmetry.

Detect a marker and estimate the position of the robot with respect to it needs few steps: first of all the marker needs to be recognized, for this purpose the algorithm focuses on finding the right shape using some threshold algorithms, then searches for a pattern inside the shape. If the pattern found matches one of the pattern stored in a library, the marker is recognized. After, the pose of the robot is computed taking into account the four relevant points of the marker using various transformations.

The disadvantages of this method and the reasons why they are not the main localization method is that the algorithm is prone to error due to different factors such as:

- Pose Ambiguity: Due to 3D-2D projection in some cases the pose of the marker cannot be determined uniquely since the true and the flipped orientation are indistinguishable. This problem is due to the quality of the camera and how many big the marker is inside the image.
- Occlusion: When an object is between the camera and the marker the recognition of the pattern is more difficult leading to no recognition or, worse, wrong recognition. The same problem arises also with poor lighting condition.

There are different kind of packages that provide fiducial markers, most of them are open source and compatible with ROS. The most famous outside the robot field are doubtlessly QR-codes, but there are actually many other kind of fiducial markers

that are easier to detect and more reliable. Among the variety of packages available some of the most used are Arucos, AprilTag and ARTag.

FIGURE 3.1: Examples of fiducial markers

Chapter 4

Algorithms and methods

4.1 Sensors calibration

When using real instruments, calibration is always necessary to avoid systematic error in measurements. Calibration needs to be done only once or, in case of instrument degradation, every now and then.
In this project, the sensors that needed prior calibration were the IMU and the four cameras.

4.1.1 IMU

The IMU used in this project is a sensor that fuses together measurements coming from an accelerometer, a gyroscope and a magnetometer. It can work in different configurations:

Operation mode	Accelerometer	Gyroscope	Magnetometer
IMU	X	X	
COMPASS	X		X
M4G	X		X
NDOF_FMC_OFF	X	X	X
NDOF	X	X	X

TABLE 4.1: IMU Operation modes

This IMU has a ROS library that works as a driver for the instrument using UART communication. In addition, the library provides a user-friendly calibration interface [35].

Using IMU mode, only accelerometer and gyroscope need to be calibrated.
To calibrate the gyroscope the device needs to be placed in a stable position for few seconds.
To calibrate the accelerator the device needs to be positioned in six different stable positions perpendicular to the three orthonormal axes for few seconds each.

4.1.2 Cameras

The robot is equipped with four USB cameras. For each one of them calibration has been performed using the *camera_calibration* [36] node in ROS. To calibrate a camera the standard procedure is to print a checkerboard pattern of known dimensions. In

this case a 8x6 checkerboard with 30mm squares was used. Then, using the camera, image messages are published over a topic in ROS and the *camera_calibration* node is launched. To get a good calibration the checkerboard has to be moved around the camera in different positions and orientations. As the checkerboard is moved, the interface displays the amount of calibration achieved. Once satisfied with the result the *CALIBRATE* button can be pressed and the camera parameters are saved.

4.2 Pose estimation with Aruco markers

Among the various markers available, the decision fell on the Aruco library. The reason behind this decision is that Arucos are included inside the *OpenCV* [37] library, one of the most used open source library for image processing and computer vision. The methods offered in this library make the marker detection possible with only few lines of code. Moreover detection performance in terms of CPU usage are one of the best and also detection accuracy and pose estimation are proved to be resilient to distances and high angles of rotation [33][34]. Using the *OpenCV* library markers can be detected easily.

4.2.1 Reference frames involved:

Once the marker has been detected it is necessary to estimate the position of the camera with respect to the marker and then the position of the robot with respect to the world frame.

To do this we need to transform among different reference frames:

- **Marker Frame:** The marker's reference frame has its origin in the center of the marker. The x axis points right, the y axis points up and the z axis points out.
- **Camera Frame:** The camera reference frame follows the REP103. So it has the x axis pointing right, the y axis pointing down and the z axis pointing forward.
- **Robot Frame:** The robot frame is called *base_link* in accordance with REP105 conventions. The x axis point forward, the y axis points left, the z axis points up.
- **Map Frame:** The map frame has its origin where the map started to be created. The direction of the axis is in accordance to the *base_link* frame when the robot was spawned in the map during the map creation, so x pointing forward, y pointing left and z pointing up.

4.2.2 Reference frame transformations:

From Map to Marker

The pose of the marker inside the map needs to be measured manually. Once a marker position is placed inside the map, information regarding the position and orientation, together with the id number and the square size, are stored inside a *.yaml* file. When markers detection is needed, the parameters are loaded as *rosparams*.

Since the four markers are facing each direction of the robot, attaching markers to walls was the smartest option, so markers are rotated with respect to the map around z by an angle θ and can be placed everywhere in the map, so there is also a translation

vector $\mathbf{t} = [t_x, t_y, t_z]$. Now to orient the axis as described in the previous section a rotation by 90° about x and then y is performed:

$$\mathbf{T}_{\mathbf{marker}}^{\mathbf{world}} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & t_x \\ \sin\theta & \cos\theta & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

From Marker to Camera

As explained in 2.3 it is possible to estimate the pose of a camera just looking at the position of a known object inside the image. In this case, the known object is the marker. About it we know the side length of the square and its position in the map.

Knowing the length of the side it is possible to derive the distance between the camera and the marker, and since we know that it is a square, we can compute the relative pose of the camera and the marker.

To do this the PnP algorithm is used. This algorithm is provided by the *OpenCV* library and has different methods to perform it, according to the situation. When the object to estimate is a square of known dimension the *SOLVEPNP_IPPE_SQUARE* is the most suitable one.

The necessary information are the intrinsic camera parameters, the dimension of the side of the square and the pixels position of the four corners of the marker inside the image.

The result is given with a rotation matrix and a translation vector representing the rotation and translation of the marker with respect to the camera.

$$\mathbf{T}_{\mathbf{marker}}^{\mathbf{camera}} = \begin{bmatrix} \mathbf{R}_{\mathbf{marker}}^{\mathbf{camera}} & \mathbf{t}_{\mathbf{marker}}^{\mathbf{camera}} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.2}$$

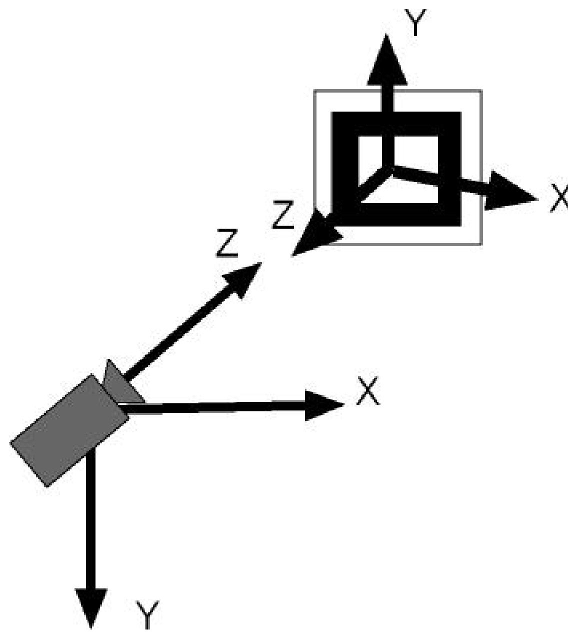

FIGURE 4.1: Reference frames orientation of camera and marker [38]

This matrix needs to be inverted to obtain the pose of the camera with respect to the marker.

$$\mathbf{T}_{\mathbf{camera}}^{\mathbf{marker}} = \mathbf{T}_{\mathbf{marker}}^{\mathbf{camera}\,-1} \tag{4.3}$$

From camera to base_link

This transformation depends on the camera that is detecting the marker. To compute automatically the transformation the *tf* package helps providing the *lookupTranform* method that looks at the transformation tree and gives as result the rotation matrix and the translation vector. Anyway, the rotation is a simple 90° rotation around the x and then another rotation about the z axis. The amount of rotation ϕ around the z axis depends on which camera is in action.

$$\mathbf{T}_{\mathbf{base_link}}^{\mathbf{camera}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 & c_x \\ \sin\phi & \cos\phi & 0 & c_y \\ 0 & 0 & 1 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

Where $\mathbf{c} = [c_x, c_y, c_z]$ is the distance between the camera and the base_link.

As a result the position of the robot with respect to the map, given a know position of a marker in the map frame can be achieved multiplying together the following transformation matrix:

$$\mathbf{T}_{\mathbf{base_link}}^{\mathbf{world}} = \mathbf{T}_{\mathbf{marker}}^{\mathbf{world}} \cdot \mathbf{T}_{\mathbf{camera}}^{\mathbf{marker}} \cdot \mathbf{T}_{\mathbf{base_link}}^{\mathbf{camera}} \tag{4.5}$$

4.3 Variance estimation

4.3.1 IMU and wheels encoders

Every measurement acquired by any sensor is affected by error. The estimation of this error is fundamental to have information about the accuracy of the final result.

For the IMU and the wheel encoders, measurements of speed and acceleration were taken with the robot slowly moving. The variance was estimated using the standard formula:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \overline{x})}{N} \tag{4.6}$$

Where N is the numbers of samples and $\overline{x}$ is the true value.

4.3.2 LiDAR

The LiDAR does not measure velocity or acceleration directly. The only thing that a LiDAR does is giving sets of 3D-point cloud and 2D scans of the surrounding environment. From these scans it possible to derive information about velocity through scan-matching algorithms. The algorithm used is the one implemented in the *rf2o* [39] ROS open-source package. The source code has its own method to estimate measurement variance, so no work was needed.

4.3.3 Markers

Data collection

For the pose estimation retrieved from the marker detection the work is different. In this case, the measurement taken is the position of the camera with respect to the marker and then, from that measurement, the position of the robot is derived. What can be computed directly is the variance of the measurement taken. To have the uncertainty about the robot position the theory about *propagation of uncertainty* has to be taken into account.

First of all, variance of the direct measurement must be computed. What affects the precision of the estimation is the distance and the angulation of the marker with respect to the camera. Following the work done in [40] an experiment was run to understand how distance e relative orientation affect the measurement.

For the experiment, an Aruco marker of 180mm size was mounted on a stepper motor and rotated from -80° to 80° with a step of around 3°. The marker was placed at different distances spanning from 0.4m to more than 2.5m and for each position and orientation 200 images were taken with one of the four cameras available. Most of this positions were exactly in front of the camera but also off-center positions were tested. The 200 images for each pose are used to compute the variance of the measurement according to 4.6. The whole experiment was repeated three times.

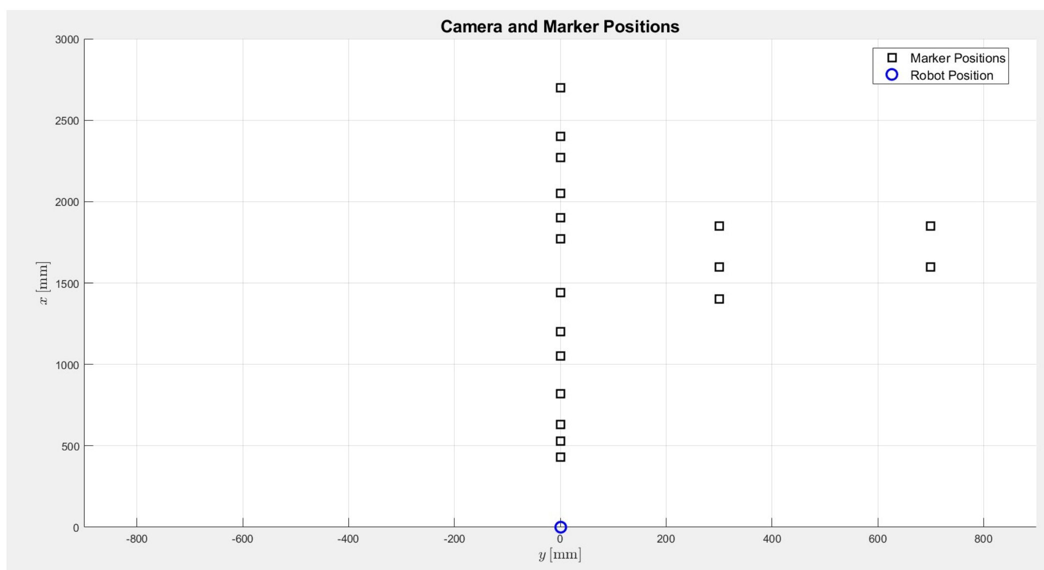

FIGURE 4.2: Positions of the marker during data collection

The accuracy of the estimation depends on the distance between the marker and the camera, but actually it is not always accurate. On this note, further analysis were made. After a certain distance and for some angles, the estimated position of the marker shows to be flickering between the right position and the symmetric one. Indeed this is a huge problem and a workaround to avoid wrong estimations was necessary.

The idea was to compute accurately the variance for each position and orientation and build a model of the variance, in this way, every time a marker is detected the

variance can be computed and if it happens to be too high, the measurement is discarded. Moreover, in case multiple markers are detected at the same time, only the one with lower variance will be taken in consideration.

The accuracy depends on the angle and the dimension of the marker in the image, so the key factor is not the distance itself, but the number of pixels representing the area of the marker in the image. If the model was dependent on the area instead of the distance, it would be possible to use markers of different sizes, as preferred. The detected area of the marker depends not only on the distance but also on the orientation of the marker, for this reason a formula to normalized the detected area is used:

$$S_n = \frac{(x^2 - a^2 \sin^2 \beta)^2}{x^3(x \cos \beta + y \sin \beta)} S_\beta \tag{4.7}$$

Where a is the half of the square side, x (l in the figure), y and β are the marker coordinates and yaw with respect to the camera, S_β is the number of pixels in the undistorted image displaying the marker and S_n is the normalized area.

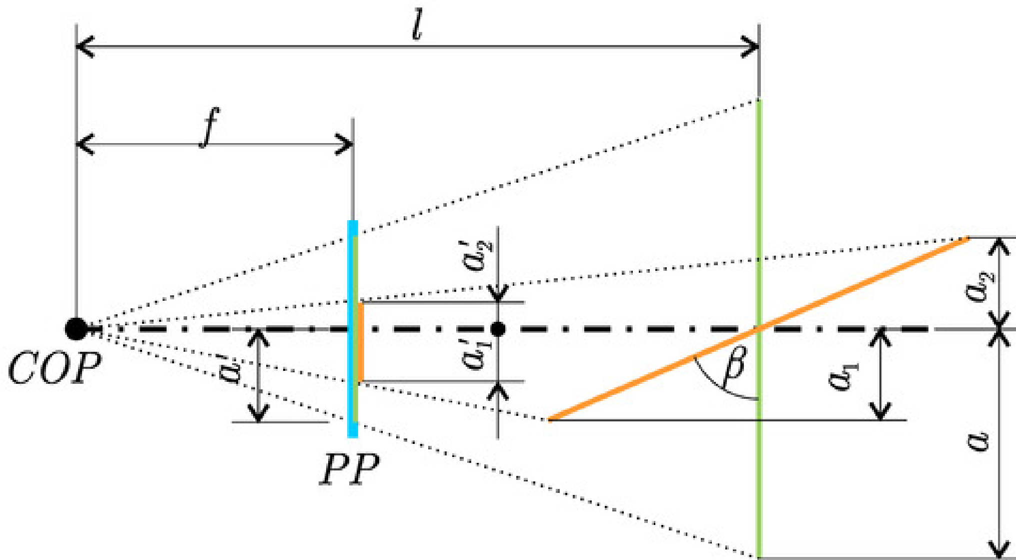

FIGURE 4.3: Rotated and equivalent marker detected by the camera on the projection plane [40]

The variance of the estimation of yaw, x and y with respect to the camera $\sigma^c_{m_\beta}\sigma^c_{m_x}\sigma^c_{m_y}$ are estimated depending on the detected area and the angle. The offset dependence was removed because data showed that a y-offset does not affect the variance. Variances can be written in compact notation as :

$$\mathbf{\Sigma^c_m} = \begin{bmatrix} \sigma^c_{m_x} & 0 & 0 \\ 0 & \sigma^c_{m_y} & 0 \\ 0 & 0 & \sigma^c_{m_\beta} \end{bmatrix} \tag{4.8}$$

This covariance matrix is made of functions depending on the detected area and the angle of the marker in the image. A fitting non-linear function was searched using the Levenberg-Marquardt algorithm, but the result was not successful.

Interpolation

Since no function could be found, an alternative solution was implemented: the data collected during the experiment were put together in a look-up table. Every time a marker is detected, the online variance of the measurement is computed interpolating the data in the look-up table. The interpolation performed is a bilinear interpolation over a surface determined by the four closest data collected. To perform the interpolation the algorithm is pretty simple: having four point $P_{00}, P_{10}, P_{01}, P_{11}$ and the function to interpolate as $f(x,y) \approx a_{00} + a_{10}x + a_{01}y + a_{11}xy$ the problem can be solved as:

$$\begin{bmatrix} 1 & x_1 & y_1 & x_1y_1 \\ 1 & x_1 & y_2 & x_1y_2 \\ 1 & x_2 & y_1 & x_2y_1 \\ 1 & x_2 & y_2 & x_2y_2 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{bmatrix} = \begin{bmatrix} f(P_{00}) \\ f(P_{10}) \\ f(P_{01}) \\ f(P_{11}) \end{bmatrix} \tag{4.9}$$

Yielding to the result:

$$\begin{bmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{bmatrix} = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2y_2 & -x_2y_1 & -x_1y_2 & x_1y_1 \\ -y_2 & y_1 & y_2 & -y_1 \\ -x_2 & x_2 & x_1 & -x_1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{bmatrix} \tag{4.10}$$

Using the estimated parameter we can compute $f(x,y)$ and obtain the value of $\sigma^c_{m_\beta}\sigma^c_{m_x}\sigma^c_{m_y}$ in real time.

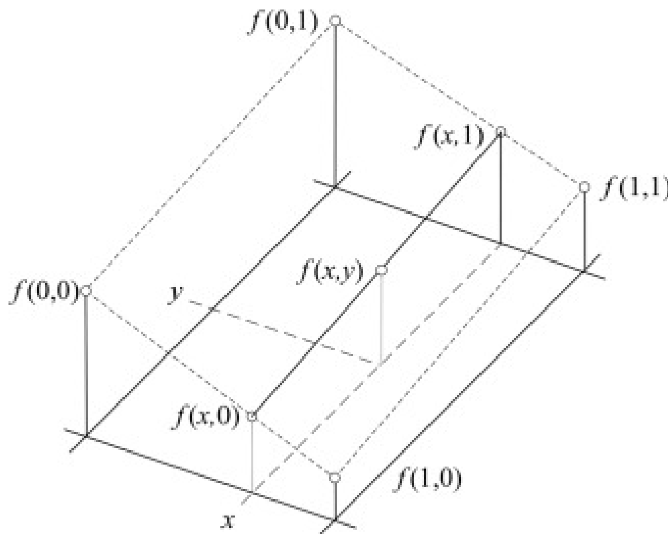

FIGURE 4.4: Example of bilinear interpolation in 3D [41]

Error propagation

Another source of variance in the measurements is the human inaccuracy when placing markers in the designated spot. This affects the precision of the position of the marker with respect to the map frame. A matrix of the same form of 4.8 can be written:

$$\Sigma_{\mathbf{m}}^{\mathbf{map}} = \begin{bmatrix} \sigma_{m_x}^{map} & 0 & 0 \\ 0 & \sigma_{m_y}^{map} & 0 \\ 0 & 0 & \sigma_{m_\beta}^{map} \end{bmatrix} \tag{4.11}$$

Using the formula 4.3 it is possible to compute the x and y coordinates of the robot with respect to the map frame, r_x^{map} and r_y^{map}. The other data needed is the angle of the robot, that can be computed as $r_\beta^{map} = m_\beta^{map} - m_\beta^c$ where the last two parameter are the yaws of the marker with respect to the map and the camera frame. This can be written with the notation $\mathbf{r^{map}} = f(\mathbf{m^c}, \mathbf{m^{map}})$. The resulting formulas will depend on the position of the marker with respect to the world and the position of the marker with respect to the robot and we just derived the two corresponding covariance matrices.

Finally, we can apply the formula for the propagation of the uncertainty:

$$\Sigma_{\mathbf{r^{map}}} = \frac{\partial f(\mathbf{m^c}, \mathbf{m^{map}})}{\partial \mathbf{m^c}} \Sigma_{\mathbf{m^c}} \frac{\partial f(\mathbf{m^c}, \mathbf{m^{map}})}{\partial \mathbf{m^c}}^T + \frac{\partial f(\mathbf{m^c}, \mathbf{m^{map}})}{\partial \mathbf{m^{map}}} \Sigma_{\mathbf{m^{map}}} \frac{\partial f(\mathbf{m^c}, \mathbf{m^{map}})}{\partial \mathbf{m^{map}}}^T \tag{4.12}$$

The result is the estimation of the covariance of x,y and yaw of the robot in the map frame.

4.4 Odometry

Once all the sensors are calibrated and the variance of the measurements is estimated, it is common practice to fuse the measurements through an Extended Kalman Filter. The reason is that taking into account measurements coming from different sensors is way better then relying only on one. This helps to overcome problems that can arise from momentary sensors failure or break down. The EKF is implemented by the *robot_localization* [42] package. This package gives the possibility to use all the sensors wanted and configure them individually.

The state variables are $(x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \ddot{x}, \ddot{y}, \ddot{z})$. Each sensor contributes to the estimation of any of these variables, but the filter works also if data coming from sensors are not continuous, allowing a partial update of the state vector. In general, at least information about $\dot{x}, \dot{y}$ and *pitch* should be given to the filter.

A first EKF filter was used to provide the $odom \rightarrow base_link$ transform fusing together measurements coming from wheel encoders, IMU and LiDAR with the following configuration (1=active , 0=disabled) :

As it is possible to notice, the markers are not included for the odometry calculation. This happens because the robot's position computed using the markers is referenced to the map frame and not the odometry frame. Moreover, to estimate the position of the robot using the marker, we need to know the position of the marker inside the map, so we first need to build the map.

Trying to estimate the marker position by hand before building the map, and then using the marker to help estimating the robot's position to create the map led to no result: the measurement retrieved from wheels, IMU and LiDAR are giving velocity and acceleration estimation, while the marker are giving precise position estimation.

Configuration Vector

Sensor	x	y	z	ϕ	θ	ψ	$\dot{x}$	$\dot{y}$	$\dot{z}$	$\dot{\phi}$	$\dot{\theta}$	$\dot{\psi}$	$\ddot{x}$	$\ddot{y}$	$\ddot{z}$
Encoders	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
IMU	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
rf2o	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0

TABLE 4.2: Inputs to the Extended Kalman filter publishing the *odom* → *base_link* transform

Using only velocity and acceleration ensure the odometry frame to move smoothly, allowing the mapping toolbox to create a map; but using the markers leads to jumps in the position estimation and this leads to map segmentation.

4.5 SLAM

Among all the packages listed in chapter 3 the *slam_toolbox* was the one chosen. Is the newest one and allows mapping in multiple section, has two different mapping method and a user-friendly plugin for Rviz to help with the mapping, together with all the features already mentioned in the previous chapter.

The node used for mapping is the *sync_slam_toolbox_node* and it is used with the default parameter suggested in the documentation. To work, the only information the toolbox needs to know are the transformations *LiDAR* → *base_link* and the *odom* → *base_link*. Both of them are sent as messages in the *tf* topic. The topics published by the toolbox are the map and the position of the robot as *map* → *odom* transform. The position of the robot is estimated from the odometry correcting it using a scan matching algorithm.

The built map can be saved in two ways: using the *map_server* [43] package the map is saved as *.pgm* format and using the *slam_toolbox* the serialized file is saved for possible further mapping.

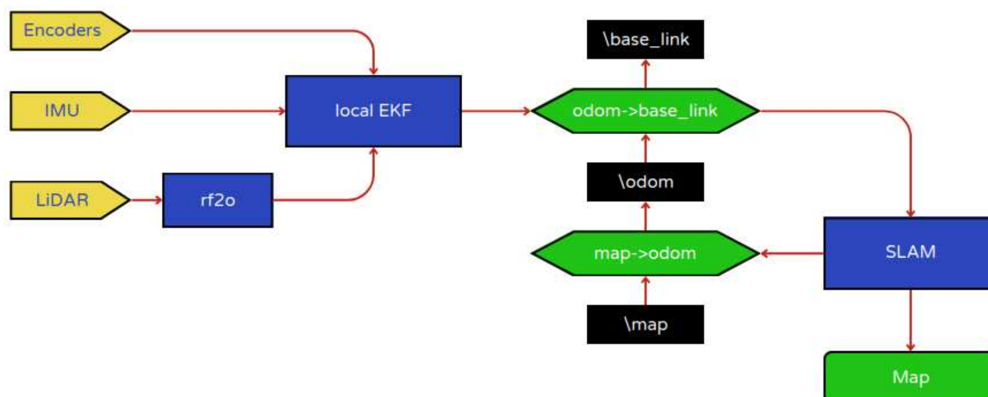

FIGURE 4.5: Graph of the algorithm structure during SLAM

4.6 Localization

Once the map is created, it can be used for navigating the robot without using the SLAM algorithm every time, using less CPU. Anyway another way for computing the $map \rightarrow odom$ needs to be chosen.

4.6.1 Without Markers

Typically, the standard procedure for robot navigation is to use the *amcl* package. This package uses the LiDAR to perform scan-matching and implement the Monte Carlo Localization algorithm. It takes as inputs the *tf* transform between $odom \rightarrow base_link$, the 2D scans coming from the LiDAR and a map. The algorithm works comparing the real time scan of the environment with the 2D map, searching for the best matches all over it.

To start working it is necessary to give a starting position to the algorithm, that will help to converge to the right result. In the case no initial estimation can be given because there is no idea of where the robot is, it is possible to use the *global_localization* service. Using it, the particles will be spread all over the map, and the algorithm will understand autonomously the estimated position. Convergence in this way is slower and could actually lead to wrong result, but it is way better to use this option instead of giving a wrong initial estimate.

The published topics are the $map \rightarrow odom$ transform, the pose of the robot and, together with it, a cloud of particles indicating other possible poses of the robot.

A second Extend Kalman Filter fusing the result of *amcl* and other sensors publishes the *tf* transform $map \rightarrow odom$. Since the transform is better to be published by only one node, the transform published by the *amcl* package is discarded. The inputs to this second filter are displayed in table 4.3 :

Configuration Vector

Sensor	x	y	z	ϕ	θ	ψ	$\dot{x}$	$\dot{y}$	$\dot{z}$	$\dot{\phi}$	$\dot{\theta}$	$\dot{\psi}$	$\ddot{x}$	$\ddot{y}$	$\ddot{z}$
Encoders	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
IMU	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
rf2o	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0
amcl	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0

TABLE 4.3: Inputs to the Extended Kalman filter publishing the $map \rightarrow odom$ transform

4.6.2 With Markers

Localization with *amcl* was not successful due to the repetitive structure of the accelerator housing. To be exact, problems arise when no initial position can be given. In this case, the algorithm gets lost in the repetitive structure of the tunnel without being able to localize the robot in the correct position, moreover, it takes a long time to converge and this is really dangerous because until the right position is found collisions can happen anytime. Even worse is the convergence to a wrong position.

To overcome this problem markers were placed evenly all around the tunnel. When the robot is turned on it has no idea about its position, but has soon as an Aruco is detected, position is estimated correctly. This position information is given as input to *amcl* as initial position estimate. From that point, the Monte Carlo algorithm, together with all the sensors fused together with the second EKF continues with the localization. Every now and then, when a new marker is found, the filter is reset as a preventative measure in case of kidnapping.

If the LiDAR is not available or out of work, localization can be performed using other sensors and the markers, without the Monte Carlo algorithm. Anyway, other sensors are not so accurate, so between one Aruco and the next one position estimation might be inaccurate, thus, in this case markers should be placed closer all over the map.

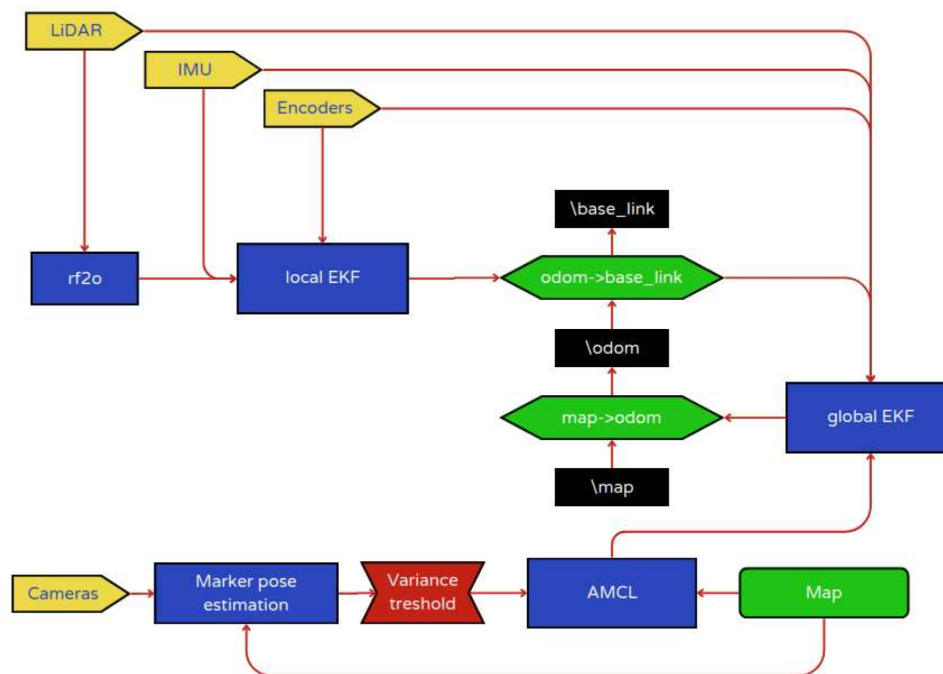

FIGURE 4.6: Graph of the algorithm structure during navigation

4.6.3 TF tree

To give a better visualization of how the robot is structured and how the transformation tree is made, a simplified version of the tf tree is shown below. In the graph only the transformation necessary for the understanding of this work are presented, even though the robot is way more complex than this, and other sensors, as mentioned in Chapter 1, are mounted on it. *chassis_link* and *upper_link* represent pieces of the chassis, that consist of more pieces assembled.

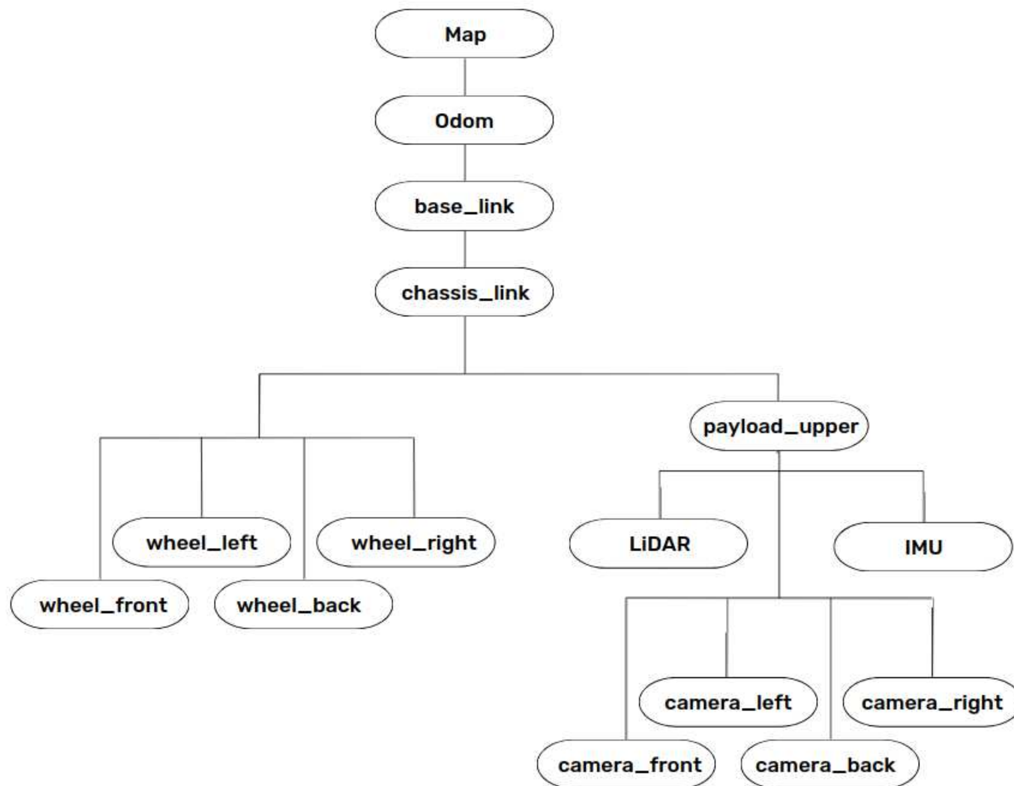

FIGURE 4.7: Simplified tf tree of the robot

Chapter 5

Implementation and Experiments

5.1 Tools and Technologies

For this research, a screening of the available tools and technologies was made to understand what would best support the analysis and implementation. This sections provides an overview of the packages and libraries used, the instrumentation employed and the programming languages supporting the work.

5.1.1 Programming languages

The main programming language used is *Python*. It was chosen for its simplicity and wide variety of open-source libraries available. Moreover, is the supported language for ROS together with *C++*. No code was directly written in *C++*, but this language is largely used in other nodes of the robot. Another programming language that is worth to mention is *XML* to write launch files.

5.1.2 Packages and Libraries

- **Numpy**

  Numpy [44] is a library for Python that implements support for multi dimensional arrays and matrices. It also provides implementation of mathematical functions to operate on them. Most of the libraries in python depend on Numpy-like arrays or matrices, so using it makes the work easier in terms of compatibility.

- **OpenCV**

  Is a library used for computer vision and image processing application in real-time. Is written in C++ but there are wrapper and language bindings for Python. The library offers a module for Aruco detection and is compatible with the Numpy library contributing to a smooth and cohesive workflow.

- **MatplotLib**

  MatplotLib [45] is a Python library for plotting static or animated 2D or 3D data plotting and visualization. It offers a lot of customization options and various plot types. Moreover is perfectly integrated with the Numpy library.

5.1.3 Instrumentation

The instrumentation available in the laboratory consist of two twin robots as described in 1.2 and a laptop running Ubuntu 20.04 with ROS mounted on it. For the

data collection during the experiment described in 4.3.3 the stepper motor was set up using an Arduino Uno. An holder for sticking the marker on a flat surface rotated by the stepper motor was designed ad-hoc and 3D printed.

5.2 Experimentation and Results

5.2.1 Data collection

First of all as reported in Chapter 4, camera and IMU needed to be calibrated. Parameter of the four calibrated camera are reported in the following tables:

Intrinsic parameters	Values
f_x	622
f_y	621
c_x	334
c_y	220
k_1	-0.4104
k_2	0.1795
k_3	0.0087
p_1	0.0023
p_2	0

TABLE 5.1: Front camera calibration parameter

Intrinsic parameters	Values
f_x	651
f_y	656
c_x	340
c_y	194
k_1	-0.4239
k_2	0.1091
k_3	0.0041
p_1	0.0068
p_2	0

TABLE 5.2: Rear camera calibration parameter

Intrinsic parameters	Values
f_x	642
f_y	642
c_x	331
c_y	179
k_1	-0.4339
k_2	0.2421
k_3	0.0016
p_1	0.0015
p_2	0

TABLE 5.3: Right camera calibration parameter

Intrinsic parameters	Values
f_x	641
f_y	621
c_x	346
c_y	193
k_1	-0.4259
k_2	0.2264
k_3	0.0017
p_1	0.0027
p_2	0

TABLE 5.4: Left camera calibration parameter

Also the IMU was calibrated and comparison between the values before and after calibration with the robot standing still shows that an offset of 0.005 was successfully removed for acceleration measurement in the x direction. For what regards the angular velocity, results were pretty good also before calibration.

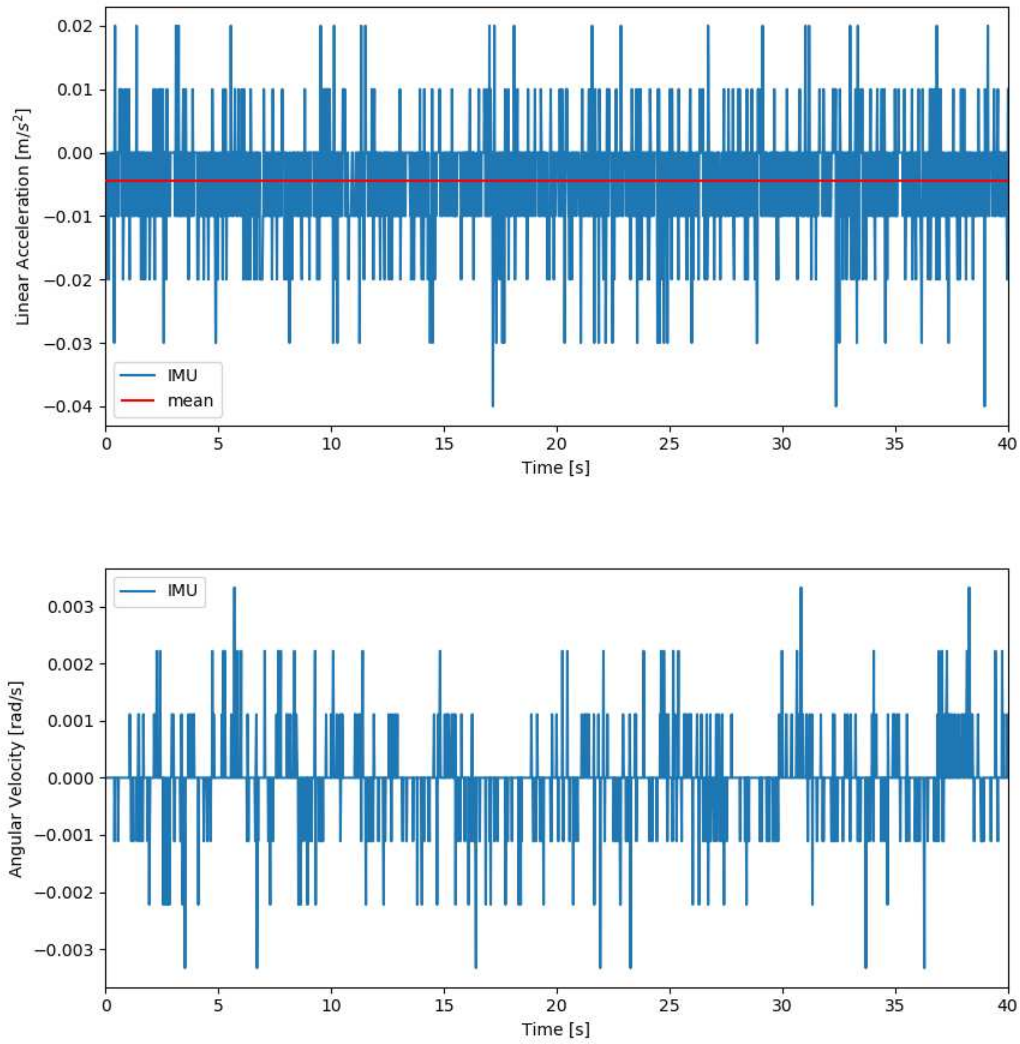

FIGURE 5.1: Values of linear acceleration on the x axis and angular velocity about the z axis BEFORE calibration with the robot standing still

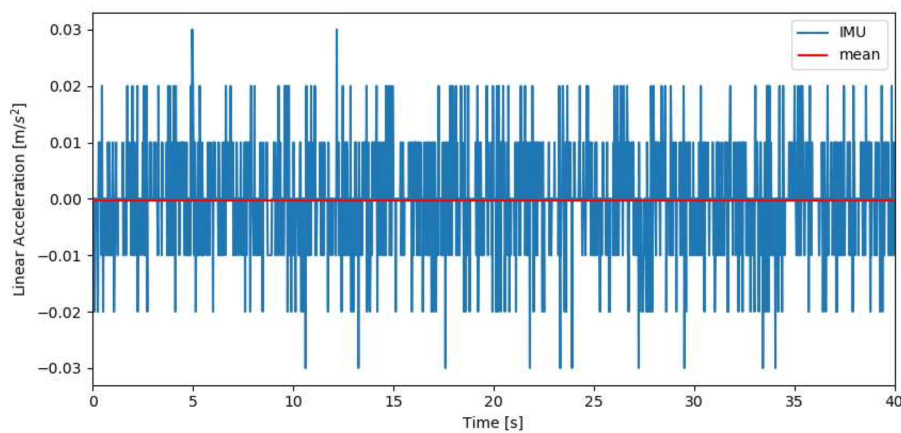

FIGURE 5.2: Values of linear acceleration on the x axis AFTER calibration with the robot standing still

While trying to perform SLAM, the odometry performed poor results. Diving deep into the possible causes, it came out that the encoders of the wheel where measuring wrong results. The plot below shows the measurement of the encoders against the input given as command to the robot. As it is possible to see, the encoders are jumping between right measurement and completely wrong measurement.

Looking carefully, it is possible to notice that when the linear velocity is measured 0m/s an angular velocity is measured. The reason behind this wrong result is that the encoders on the wheels are not acquiring correctly all the data: when both the encoders are measuring a value the sensors measure a linear velocity, if only one of them is measuring something or the two values are different then there is a angular velocity too. In the plot below the robot was going straight, so this behaviour probably means that the two encoders are not synchronised.

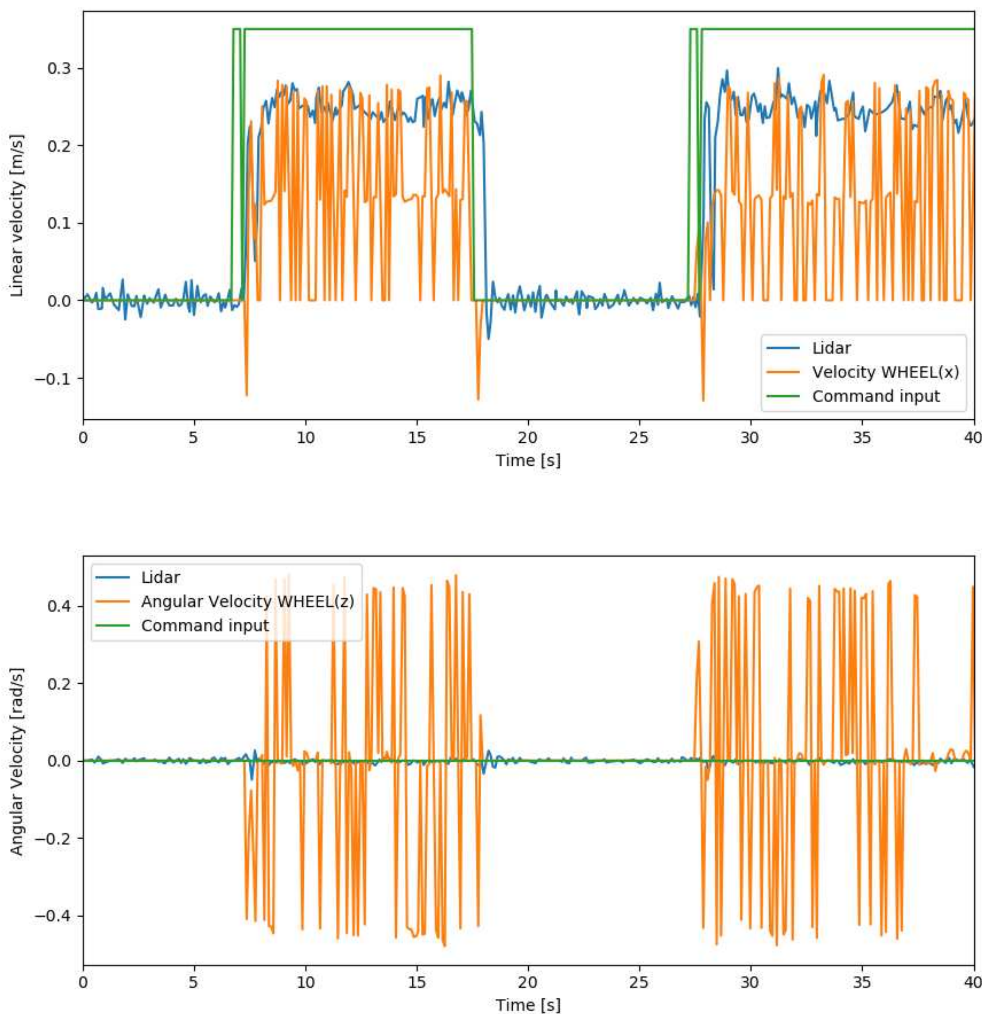

FIGURE 5.3: Faulting behaviour of wheel encoders

Repairing the encoders was not possible in a short period of time, so they were not used.

Once all the variances of the sensors were estimated, the measurements acquired are fused together in a first local Kalman filter. As it is possible to see from Figure 5.4 the values registered from the sensors are lower then the commands sent as input,

this is due to some sort of low power in the motor or a faulty loop closure in the PID. Except for this thing, the value of the EKF fuse correctly the data coming from the other sensors w8th just a little delay. In the case of the linear velocity, also the acceleration coming from the IMU is taken into account even if not displayed.

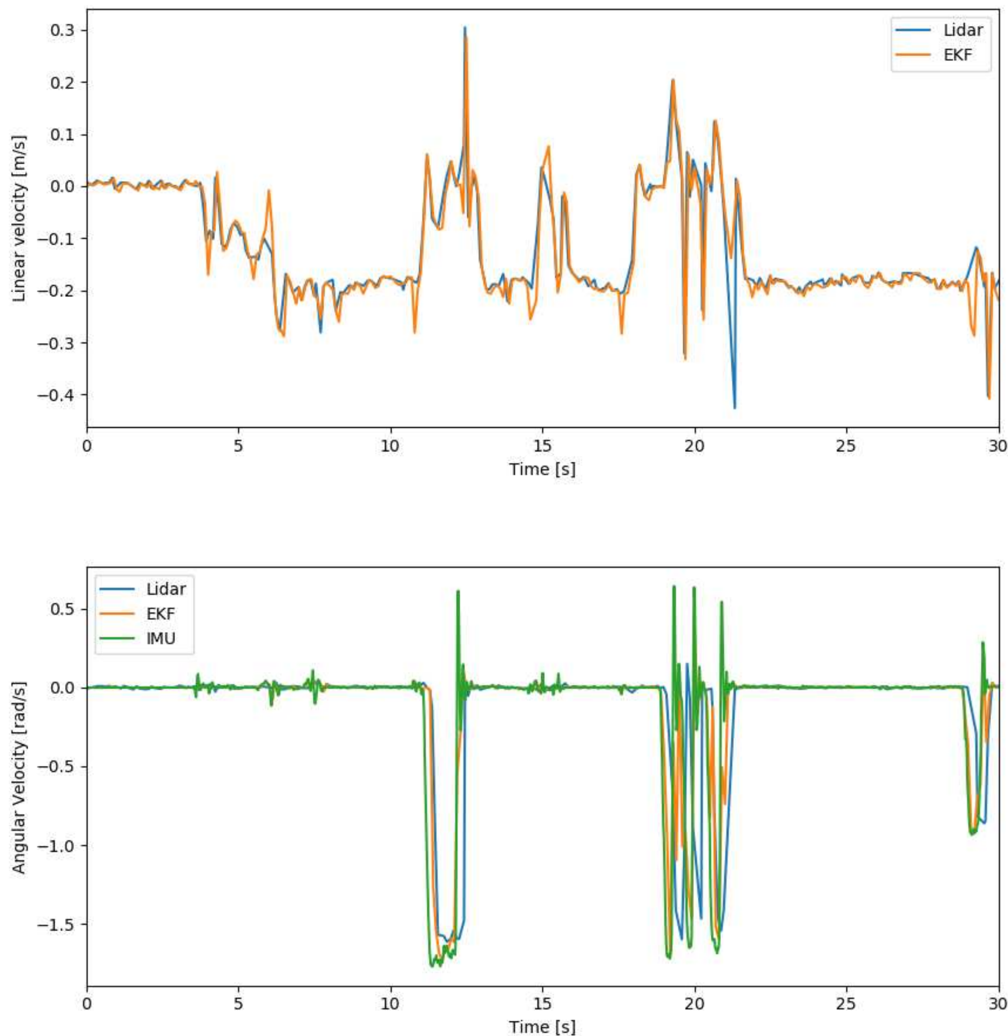

FIGURE 5.4: Extend Kalman Filter, LiDAR and IMU and command velocity data for odometry estimation

With regards to markers and their detection multiple images were collected to estimate the accuracy and variance of the estimation. Images were taken in different positions and for each position the marker was rotated from 0° to 80° with a step of 5°. For each angle 200 images were collected and position was estimated from each image.

In the two following graphs, as examples, it is possible to see the data collected for the 200 images when the robot was in front of the camera (so no offset along the Y axis) rotate of 40° and distant 0.800m and 2.400 m. When the robot is closer to the camera, the position estimated is accurate to the real values, but when the robot is further it is possible to see from 5.6 that the angle detected jumps from 40° and -40°. At that distance, an error like that means a lot of centimeter of error, so it is

not acceptable. The farther the marker is from the camera, the more this faulting behaviour appears.

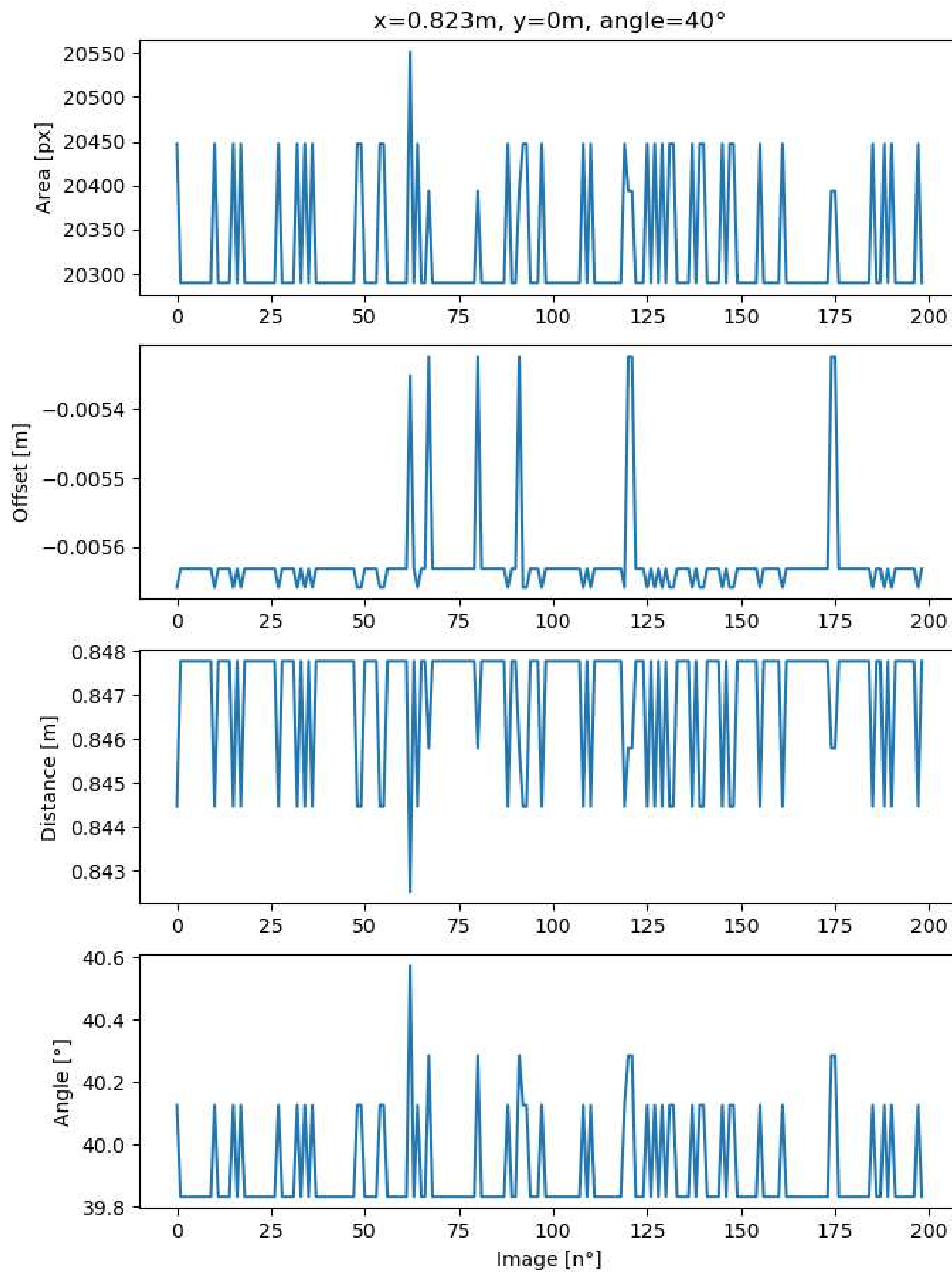

FIGURE 5.5: Area normalized, Offest, Distance and Angle detected when x=0.823, y=0, angle=40°

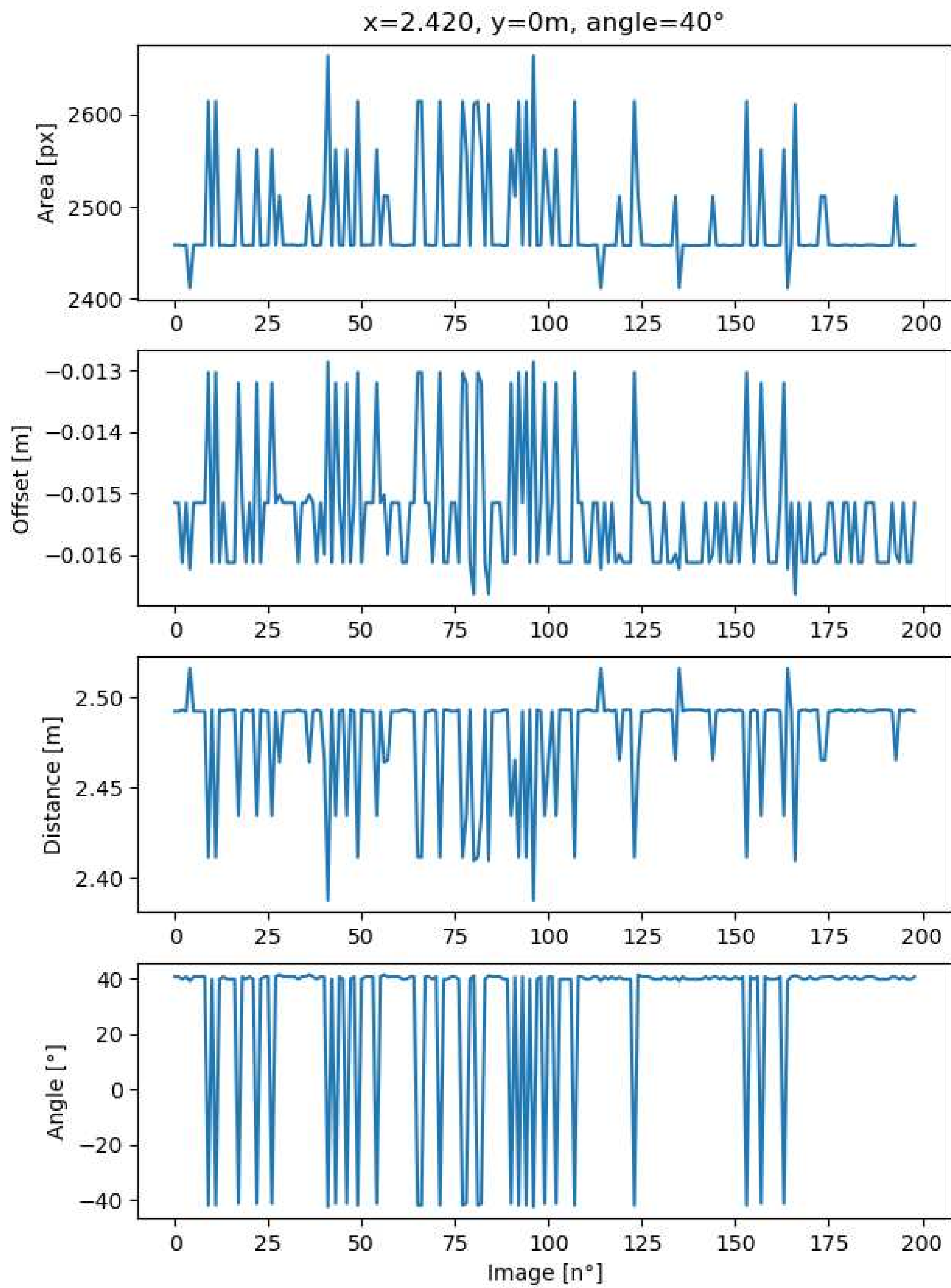

FIGURE 5.6: Area normalized, Offest, Distance and Angle detected when x=2.420, y=0, angle=40°

To have a clear idea of where the variance is higher and not acceptable, variance was computed for each set of images and all the variances were put together to form a 3D graph, dependent on the angle and the distance (in term of normalized area). The variance was estimated not only for the angle, but also for the measured distance in meters and the position of the marker along the y axis. In this way, an accurate pattern of how the variance behave was made.

The experiment was repeated three time and a mean of the variances across the three experiments was computed. For an intuitive understanding of the graphs, a color code was used: the darker the color, the higher the variance.

As it is possible to see from the below figures, the highest variance is generally associated with the angle measurement, that due to some geometric reasons affects not only the estimated orientation of the robot, but also its position.

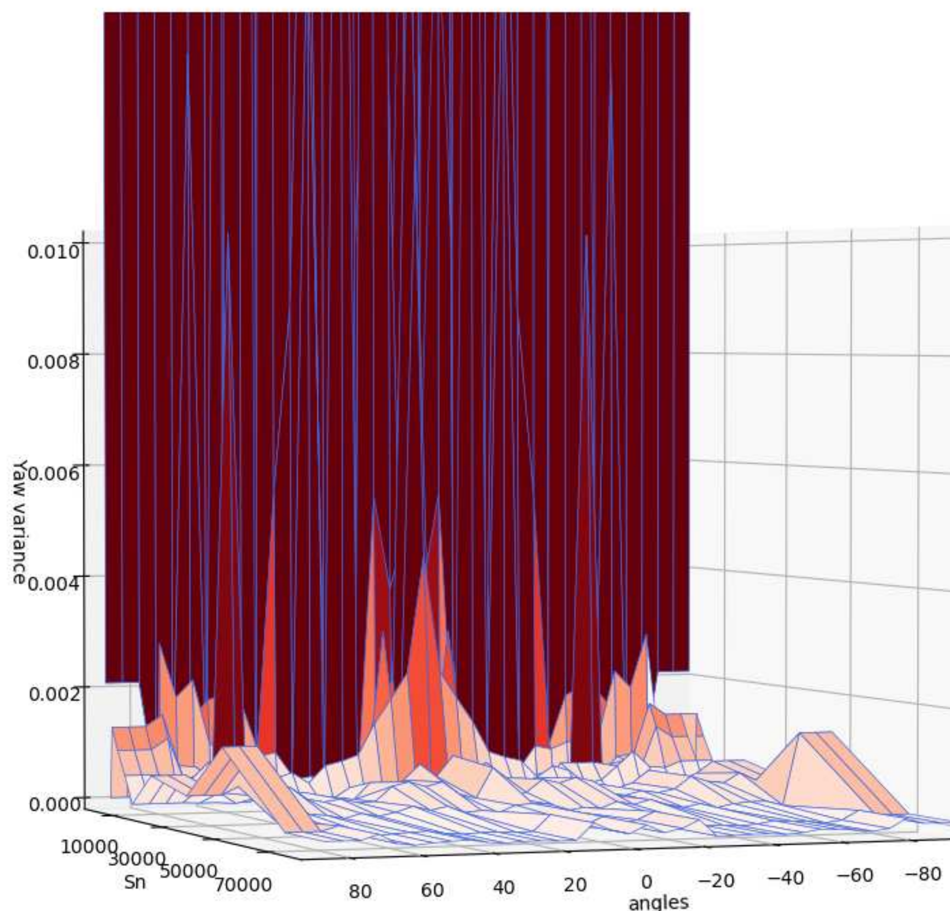

FIGURE 5.7: Values of Yaw variance depending on marker's detected area and orientation

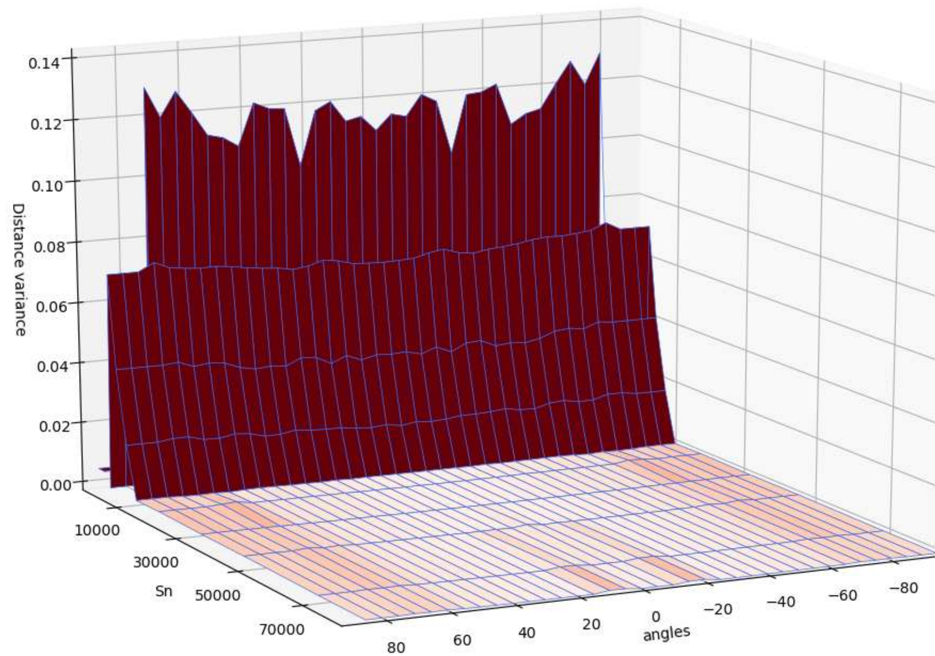

FIGURE 5.8: Values of Distance variance depending on marker's detected area and orientation

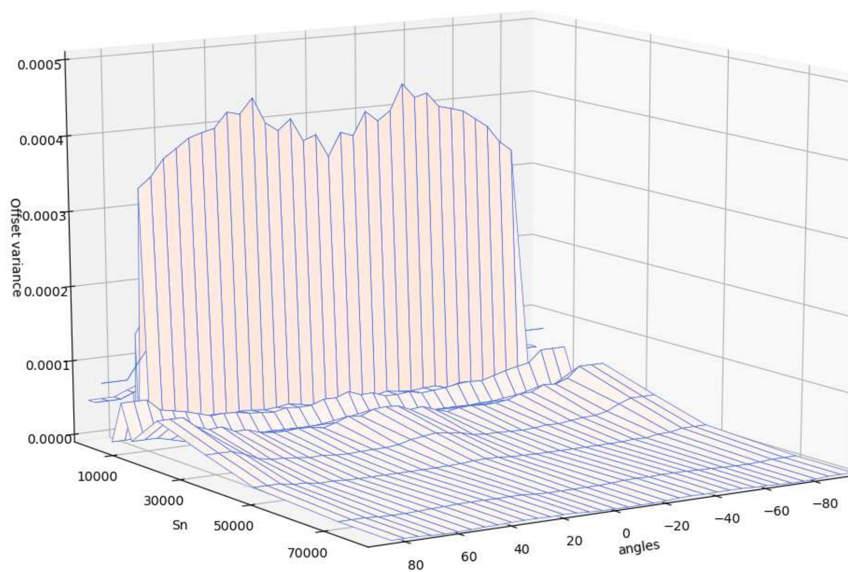

FIGURE 5.9: Values of Offset variance depending on marker's detected area and orientation

Once the variance of the robot position in the map has been computed using the propagation of uncertainty, it was possible to publish the position information together with the covariance through a topic sending *PoseWithCovarianceStamped* messages. In figure 5.10 an Aruco is detected by the right camera and the result of the pose estimation is displayed in Rviz, giving with a lilac ellipse the uncertainty about the position, and with the yellow cone the orientation variance is shown. As it is possible to see, the estimation is pretty accurate.

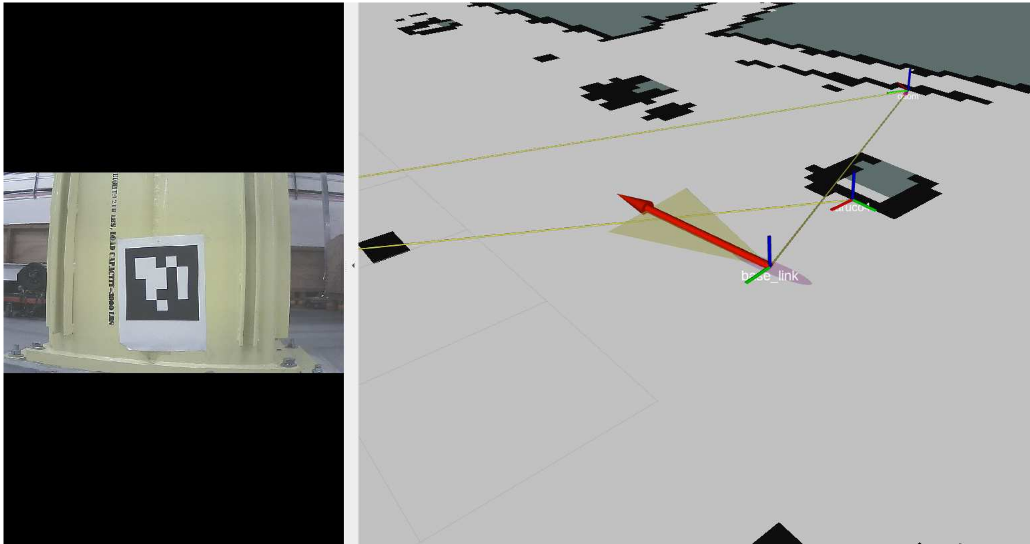

FIGURE 5.10: Position of the robot based on the Aruco detection

5.2.2 SLAM testing

In figure 5.11 the result of the SLAM session in the accelerator housing is shown.

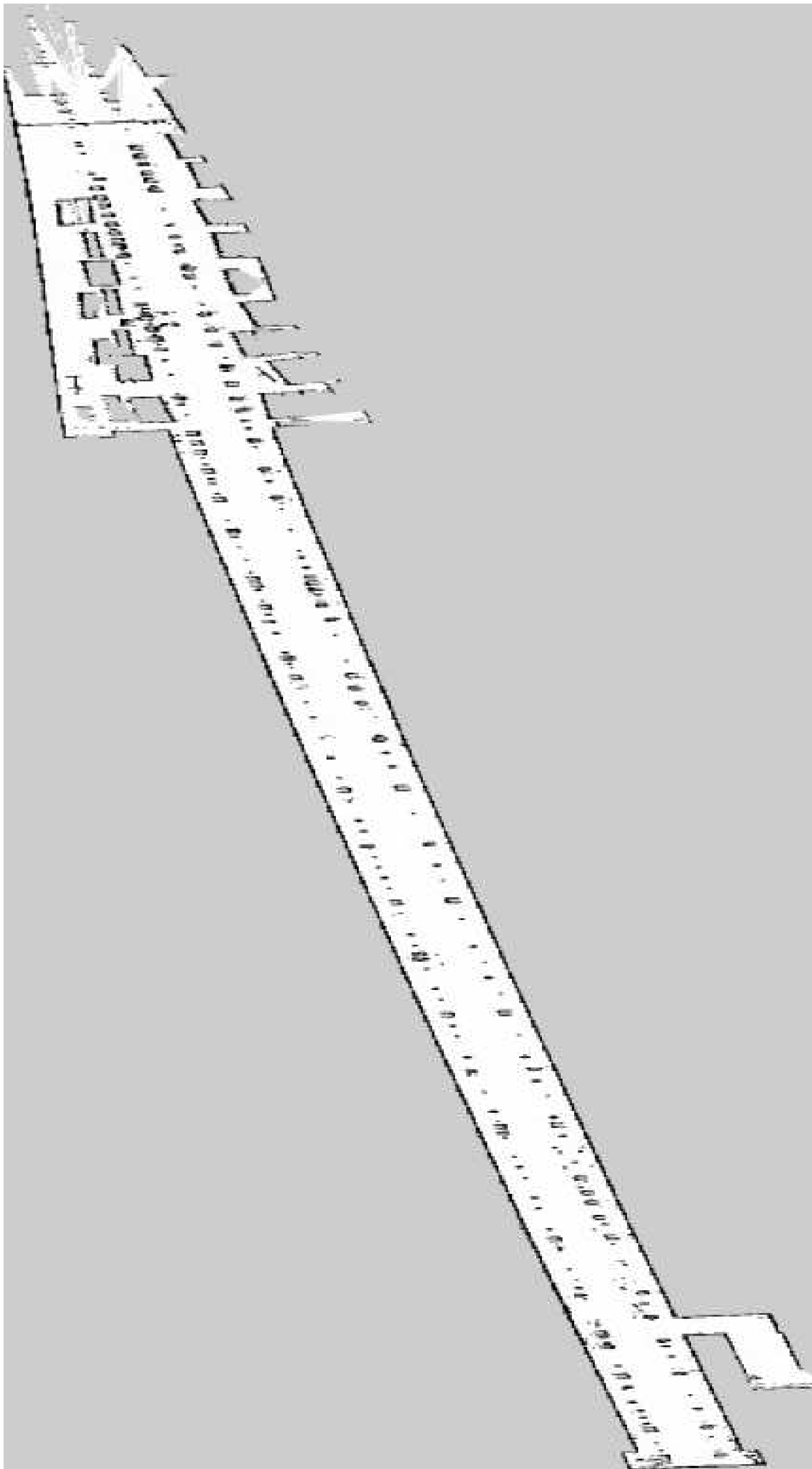

FIGURE 5.11: Map of the accelerator housing (LTU)

As it is possible to see, the SLAM session was successful. The accelerator housing was mapped completely. The top of the map may appears as if the map was done wrong, but actually there was a mesh fence dividing two sections of the accelerator, so it was possible for the Lidar to see over fence, but access was not permitted. The tunnel in the map appears to be a little bit skewed, probably due to some drifts in the odometry estimation that is inevitable when it comes to long environment like this. Anyway, the skewness of the map is low and it does not affect the accuracy of the localization.

5.2.3 Navigation testing

When the first navigation test was performed, only the local Kalman filter and the *amcl* package were used. The goal was to achieve a correct position starting from an unknown position, so no information about initial position was given to the Monte Carlo localization algorithm. This means that at the beginning particles are spread all over the map as shown in figure 5.12

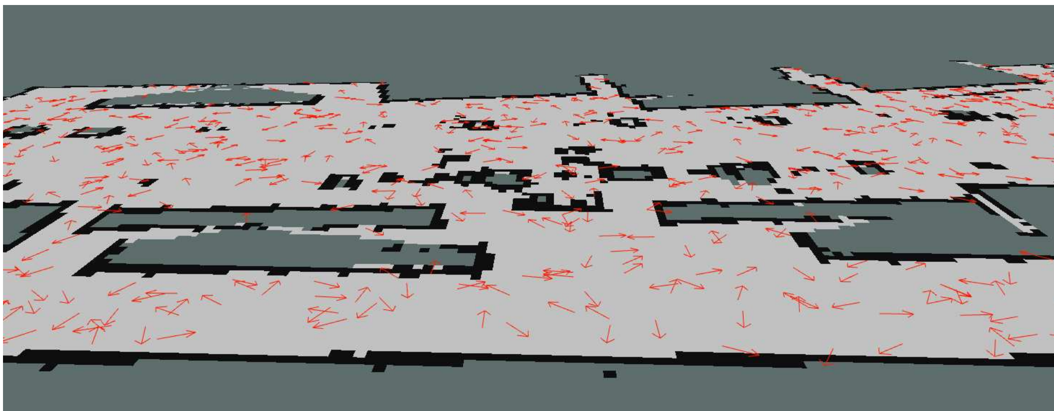

FIGURE 5.12: AMCL particles at the beginning

In normal condition, the *amcl* package is reported to show great results. But during the testing session inside the accelerator housing, the results were not good as expected.

When starting the navigation session, if a good estimate of the initial position is given, the localization filter converges fast to a correct position and the robot can navigate safely inside the tunnel.

If no initial estimate is given, the particles of the filter, initially spread all over the map, are gradually grouped together. The algorithm struggles to converge to a result, forming different groups of particles in the map. During this period navigation is obviously a problem, since there is no certainty of where the robot is. Finally, when the particles are all grouped together, the result is wrong.

The reason behind this faulty behavior is the repetitive structure of the accelerator housing: the filter works using a scan-matching approach against the map, and since the map is all similar, the probability associated with each scan are all close to each other. In figure 5.13 and 5.14 the initial ambiguity and the final wrong result of one of the testing session are shown.

It is possible to understand that the result is wrong because together with the result of the particle filter, also the real-time particle cloud scanned by the LiDAR is

available. The particle cloud is showing the truth of what is around the robot in that precise moment, so that one is the ground truth. Since the particle cloud is not matching the map, the position the filter is estimating is necessarily wrong.

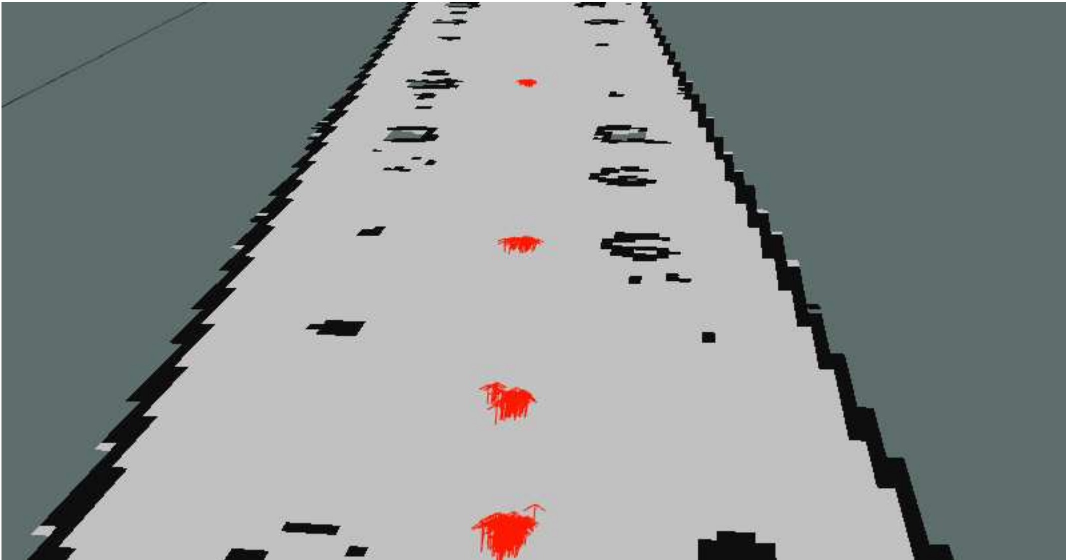

FIGURE 5.13: AMCL particles in the middle of the testing session

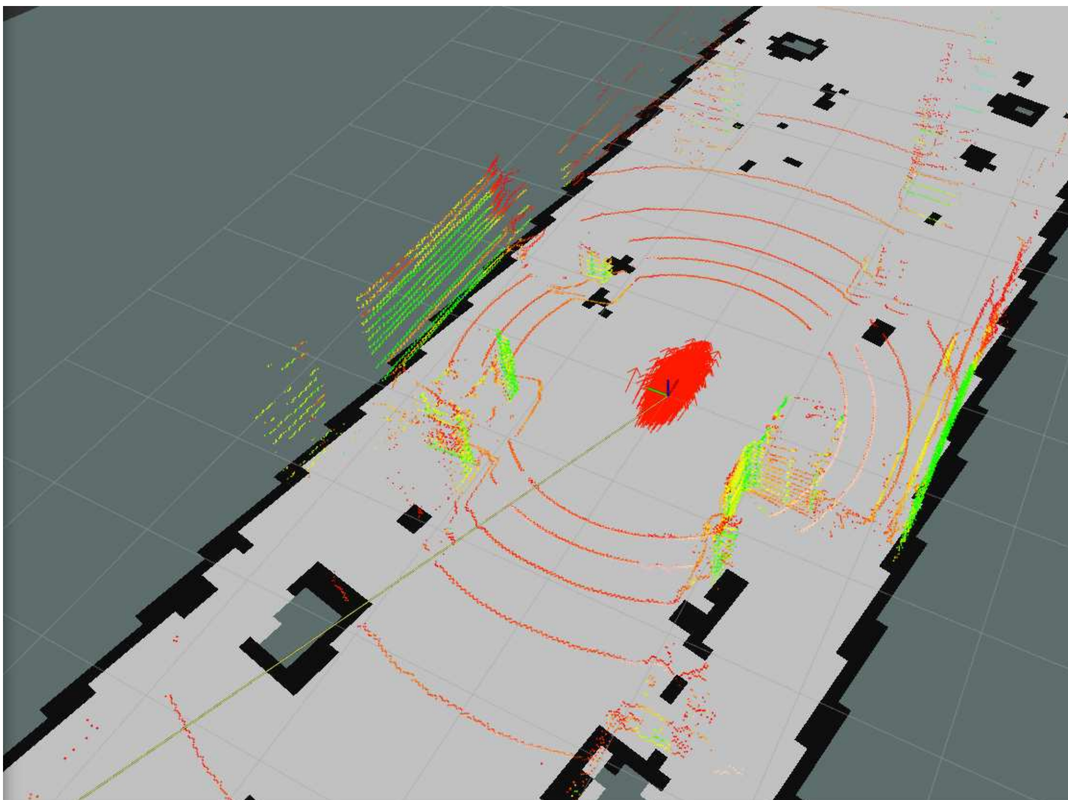

FIGURE 5.14: AMCL particles showing wrong result compared to the point cloud of the LiDAR

Using the markers completely solved the problem: when the robot starts navigating inside the map, it has no idea of where it is, but as soon as a marker is detected by one of the cameras, the position of the particle filter is reset according to the marker detection.

Finally, the result of the *amcl* is fused together with the other sensors available using a second global Kalman Filter. In Figure 5.15 the Rviz visualization of the robot navigating inside the map is shown. It is possible to see how the real-time point cloud is matching the map, and from the lilac circle it is possible to see that the variance of the estimation is really low.

In addiction to the graphical visualization it is possible to check the correctness of the results plotting the values of x,y and yaw of the robot according to the markers, the *amcl* package and the Kalman filter, as shown in figure 5.16 In the graph is shown how, with the first detected marker, the position of the robot is corrected. Then after the first correction, the localization work fine, and the marker are confirming the pose of the robot.

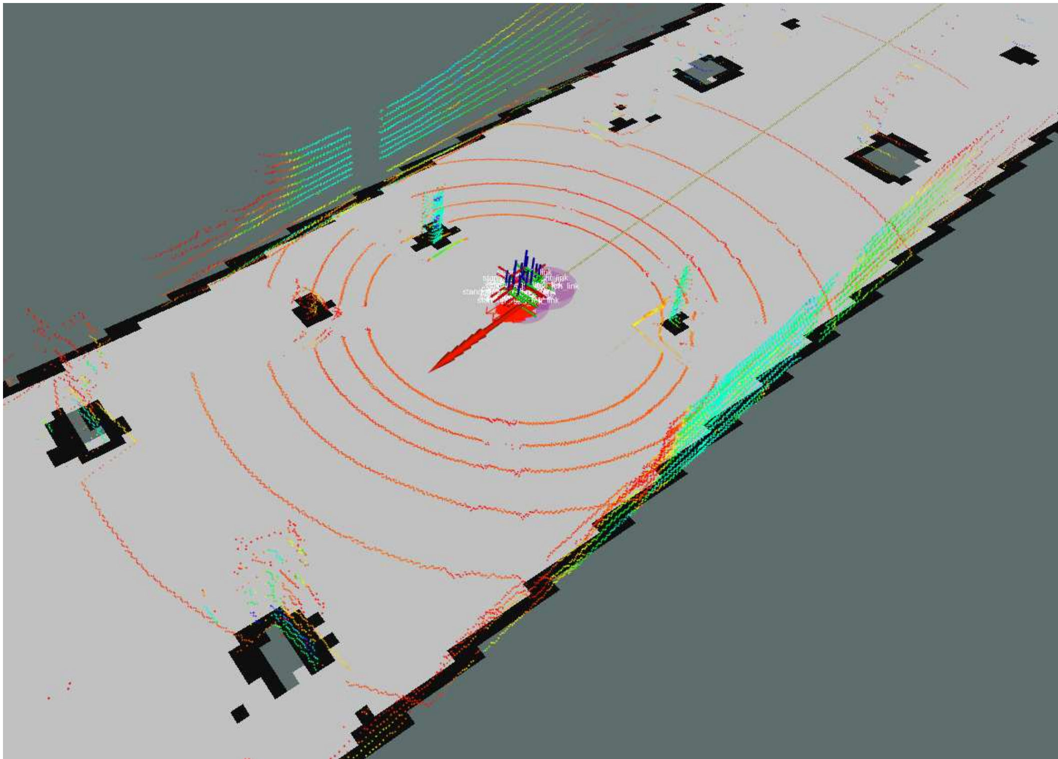

FIGURE 5.15: Rviz visualization of the navigation algorithm

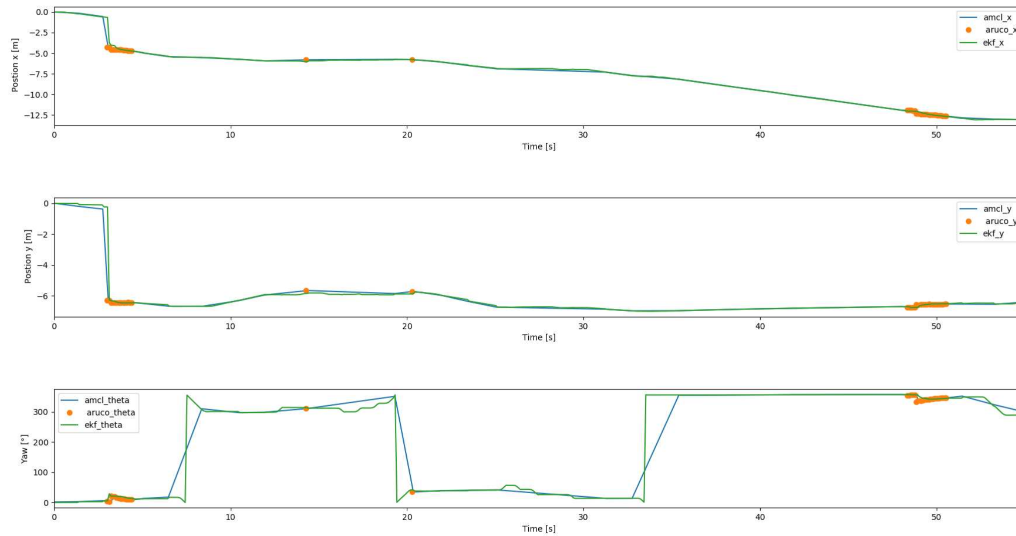

FIGURE 5.16: Plot of x,y,yaw of the robot in the map frame when using Arucos and *amcl* fused in the EKF together with other sensors

Another test was performed to see if it was possible to remove the *amcl* package and use only the fiducial markers. To do that, *amcl* was removed from the algorithm and the position estimate coming from the marker detection was used as input to the EKF, fusing the result with the *rf2o* algorithm and the IMU measurement. Result is shown in figure 5.17.

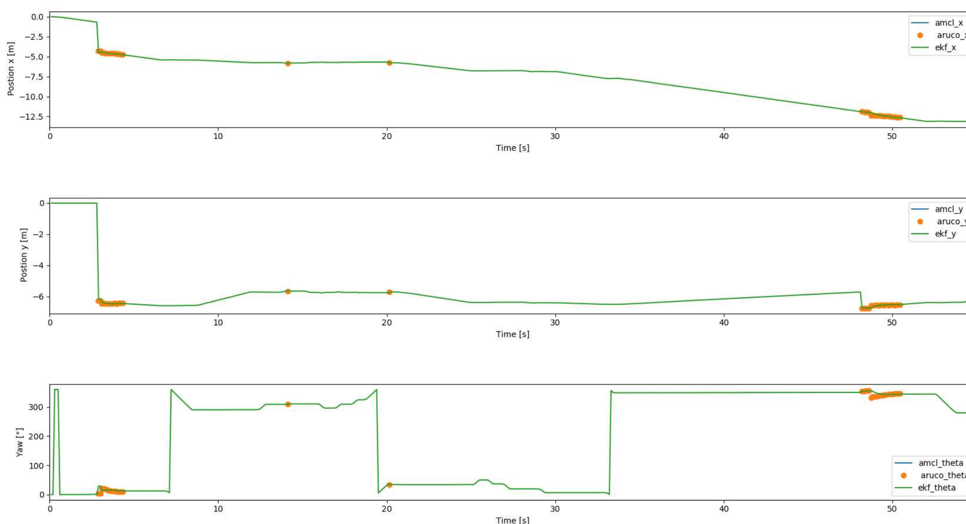

FIGURE 5.17: Plot of x,y,yaw of the robot in the map frame when using Arucos fused in the EKF together with other sensors

The results are pretty similar to the ones in 5.16 but looking closely at the y coordinate it is possible to see that around time=48s , when the last marker is detected, there is a jump in the y coordinate. The jump is of more than 1m and it is a lot considering that the width of the corridor is on average 3m. This amount of error cannot be accepted since it could lead to collision, so using both the *amcl* package and the markers leads to the best result.

Chapter 6

Discussion and conclusions

6.1 Challenges with real robots

Dealing with real robot instead of using simulations means encountering different sources of problems: the first one has to do with the verifiability of the results and correctness of the algorithm, another one is due to having to deal with a real robot that is made of circuits, firmware and pieces of software working all together to make the robot work. Any of them can fail anytime and to go on with the project is necessary to solve different side problems that can arise.

6.1.1 Verifiability Issues

To verify the accuracy and correctness of an algorithm is often necessary to have a ground truth to refer to. In outdoor environment satellite tracking systems are the one used as reference, being the most accurate in most of the cases. In indoor environment GPS can not be used and finding another reliable tracking system to compare the result might be hard.

A solution for indoor environment might be using an external camera to track the position of the robot directly inside the environment and use that position as a ground truth reference to compare the results obtained by the sensors mounted on the robot. In this case, an external camera could not be used and establishing the ground truth was a problem.

For the mapping, no accuracy estimation could be done. For localization, the best way to be sure whether the robot is in the correct estimated position or not is to check it using Rviz: using Rviz it is possible to visually check if the pre-built map is actually matching the real-time point cloud detected by the LiDAR sensor. A real-time 2D map can also be created and this can be overlapped with the pre-built one to see if they are matching. Of course this can give an approximate idea of the correctness of the result, but it does not provide an accuracy metric.

6.1.2 Robot failures and real-time limitations

Robots are complex objects made of mechanical components, electrical circuits, operating systems and pieces of code running and anyone of these parts of the robot can fail or have some sort of problem that need to be solved in order to have a perfectly working robot.

When using real robots it is perfectly normal to face setbacks of different nature and the robot used was not an exception: during the project the charging circuit got

broken in one of the robot and a twin robot was used since then. Another problem was that the network connection between the robot and the laptop used to drive to robot was failing all of a sudden without reporting any kind of error. It took two weeks to troubleshoot the issue and understand if it was an electrical or a software problem. It ended up being just a wrong set-up during the update of the Ubuntu operating system, that was making the Network connection unstable. As already mentioned, another issue was that the encoders of the wheels were not collecting data as expected and they could not be used.

Together with these issues also the limited computational speed of the processor had to be taken into account: mapping the environment is a real computationally expensive procedure and to map everything it was easier to record all the data and process them at a later time using the laptop.

6.2 Possible Improvements

Due to the limited time available to work at the project, some side tasks were left undone. The main goal was a safe navigation inside the acceleration and it was accomplished, but there is room for further improvements that can make the navigation system more feature rich and autonomous.

6.2.1 Markers new features

At SLAC there is more than one accelerator and the longest one is long more than 3km. This accelerator is subdivided in different sections and it would be helpful in the future if the robot could work in more than just one of them. To this date, only a sector of the accelerator housing was mapped. When the navigation node starts in ROS, a map of the sector is loaded together with the position of the markers inside the map.

Once that more then one sector will be mapped it will be possible to choose every time which map needs to be loaded together with the markers position, anyway this can be done only when firing up the localization and navigation node. If a change of map needs to be done in real time while the robot is operating, it would be possible, but markers positions would not be loaded together with the map. The problem can be easily solved using a service in ROS, removing the *rosparams* referring to the old markers position and loading the new set of positions every time the map is changed.

Another interesting feature that might be added is to recognize what map should be loaded just by detecting a marker. In this way, the operator driving remotely the robot would not need to manually load the map.

For what concerns the navigation, by now the markers are used every time they are detected to reset the Monte Carlo localization algorithm, but as it possible to see from 5.16, the *amcl* output, fused together with the measurement coming from the other sensors, is sufficient to estimate the correct position once it is correctly initialized, so resetting it every time is not wrong, but probably useless.

For this reason a good practice would be to use the markers only when necessary, at the beginning of the navigation or when the robot is manually moved by a human operator. Also in this case, a service might be useful to accomplish the task.

6.2.2 Autonomous driving

A standard practice once you have a map and a robot that can accurately estimate its own position inside that map is to make the robot able to navigate autonomously without the tele-operation of a human operator. In this way the robot can execute long term operation of surveillance overnight and get back to the charging station once the operation is done.

To do this, a package in ROS called *move_base* exist and it helps with setting the robot's position goal and gives feedback about whether the task is accomplished or not.

6.3 Conclusions

This projects presents the work that was done on a real robot to implement a mapping and localization algorithm, overcoming the difficulties arisen by the repetitive structure of the environment. The algorithm is implemented using the most known ROS packages usually used for this kind of situations, enhancing the localization algorithm using a marker-based approach. If the mapping process didn't present any particular issue, the localization encountered some problems leading to ambiguity in the estimation, if not a completely wrong result. The usage of fiducial markers inside the algorithm as a way to initialize the particle filter efficiently solves the problem and the markers can also be used as a checkpoint throughout the navigation, providing a correct estimation of the position and orientation of the robot inside the map and ensuring a safe navigation and avoidance of the obstacles.

Bibliography

[1] Andrew Yarovoi and Yong Kwon Cho. "Review of simultaneous localization and mapping (SLAM) for construction robotics applications". In: *Automation in Construction* 162 (2024), p. 105344. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2024.105344. URL: https://www.sciencedirect.com/science/article/pii/S0926580524000803.

[2] *SLAC National Accelerator Laboratory*. 2024. URL: https://www6.slac.stanford.edu/ (visited on 07/11/2024).

[3] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org.

[4] Thomas C. Thayer, Maria Alessandra Montironi, and Alessandro Ratti. "ROAM: A Remotely Operated Accelerator Monitor". In: *2022 18th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*. 2022, pp. 1–6. DOI: 10.1109/MESA55290.2022.10004449.

[5] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley select coursepack. Wiley, 2005. ISBN: 9780471765790. URL: https://books.google.it/books?id=muCMAAAACAAJ.

[6] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 1846286417.

[7] *How to derive rotation matrix by Euler angles*. 2022. URL: https://semath.info/src/euler-angle.html (visited on 07/11/2024).

[8] *The Cylindrical Keyword*. 2016. URL: https://www.svibs.com/resources/ARTeMIS_Modal_Help/The_Cylindrical_Keyword.htm (visited on 07/11/2024).

[9] Flavio Ferraz et al. "A comparative study of the accuracy between two computer-aided surgical simulation methods in virtual surgical planning". In: *Journal of Cranio-Maxillofacial Surgery* 49 (Dec. 2020). DOI: 10.1016/j.jcms.2020.12.002.

[10] Atte Rantanen. "Robot Operating System: overview and case study". MA thesis. University of Turku, 2024.

[11] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: vol. 3. Jan. 2009.

[12] Robert JohnJul. *Hands-On Introduction to Robot Operating System(ROS)*. 2020. URL: https://master-engineer.com/2020/11/05/writing-ros-node/ (visited on 07/11/2024).

[13] Tully Foote. "tf: The transform library". In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, pp. 1–6. DOI: 10.1109/TePRA.2013.6556373. URL: https://wiki.ros.org/tf.

[14] Purvis Foote. *Standard Units of Measure and Coordinate Conventions*. 2010. URL: https://www.ros.org/reps/rep-0103.html#copyright (visited on 07/11/2024).

[15] Wim Meeussen. *Coordinate Frames for Mobile Platforms*. 2010. URL: https://www.ros.org/reps/rep-0105.html (visited on 07/11/2024).

[16] HyeongRyeol Kam et al. "RViz: a toolkit for real domain data visualization". In: *Telecommunication Systems* 60 (Oct. 2015), pp. 1–9. DOI: 10.1007/s11235-015-0034-5. URL: https://wiki.ros.org/rviz.

[17] Luis Ortiz, Luiz Gonçalves, and Elizabeth Cabrera. *A Generic Approach for Error Estimation of Depth Data from (Stereo and RGB-D) 3D Sensors*. May 2017. DOI: 10.20944/preprints201705.0170.v1.

[18] *Camera Calibration and 3D Reconstruction*. 2024. URL: https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html (visited on 07/11/2024).

[19] *Perspective-n-Point (PnP) pose computation*. 2024. URL: https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html (visited on 07/11/2024).

[20] Bayu Kanugrahan Luknanto. "A Review of 2D SLAM Algorithms on ROS". MA thesis. Politecnico di Milano, 2019.

[21] Wikipedia contributors. *Extended Kalman filter — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-July-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Extended_Kalman_filter&oldid=1224606263.

[22] Wikipedia contributors. *Particle filter — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-July-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Particle_filter&oldid=1218031285.

[23] Wikipedia contributors. *Dead reckoning — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-July-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Dead_reckoning&oldid=1230006947.

[24] César Debeunne and Damien Vivet. "A Review of Visual-LiDAR Fusion based Simultaneous Localization and Mapping". In: *Sensors* 20.7 (2020). ISSN: 1424-8220. DOI: 10.3390/s20072068. URL: https://www.mdpi.com/1424-8220/20/7/2068.

[25] *LiDAR SLAM vs Visual SLAM: Which is Better?* 2023. URL: https://eu.hookii.com/it/blogs/robot-lawn-mowers/laser-slam-vs-visual-slam-which-is-better (visited on 07/11/2024).

[26] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters". In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 34–46. DOI: 10.1109/TRO.2006.889486. URL: https://wiki.ros.org/gmapping.

[27] *slam_karto*. 2019. URL: https://wiki.ros.org/slam_karto (visited on 07/11/2024).

[28] Stefan Kohlbrecher et al. "A flexible and scalable SLAM system with full 3D motion estimation". In: *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*. 2011, pp. 155–160. DOI: 10.1109/SSRR.2011.6106777. URL: https://wiki.ros.org/hector_slam.

[29] *Github cartographer*. 2016. URL: https://github.com/cartographer-project/cartographer (visited on 07/11/2024).

[30] Steve Macenski and Ivona Jambrecic. "SLAM Toolbox: SLAM for the dynamic world". In: *Journal of Open Source Software* 6 (May 2021), p. 2783. DOI: 10.21105/joss.02783. URL: https://wiki.ros.org/slam_toolbox.

[31] *amcl*. 2020. URL: https://wiki.ros.org/amcl (visited on 07/11/2024).

[32] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Intelligent Robotics and Autonomous Agents series. MIT Press, 2005. ISBN: 9780262201629. URL: https://books.google.it/books?id=2Zn6AQAAQBAJ.

[33] David Jurado et al. "Planar fiducial markers: a comparative study". In: *Virtual Reality* 27 (Feb. 2023), pp. 1–17. DOI: 10.1007/s10055-023-00772-5.

[34] Michail Kalaitzakis et al. "Fiducial Markers for Pose Estimation: Overview, Applications and Experimental Comparison of the ARTag, AprilTag, ArUco

and STag Markers". In: *Journal of Intelligent and Robotic Systems* 101 (Apr. 2021). DOI: 10.1007/s10846-020-01307-9.

[35] *ros_imu_bno055*. 2020. URL: https://wiki.ros.org/ros_imu_bno055 (visited on 07/11/2024).

[36] *camera*$_c$*alibration*. 2020. URL: https://wiki.ros.org/camera_calibration (visited on 07/11/2024).

[37] *Detection of ArUco Markers*. 2024. URL: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html (visited on 07/11/2024).

[38] Yazmin Villegas-Hernandez and Federico Guedea-Elizalde. "Marker's position estimation under uncontrolled environment for augmented reality". In: *International Journal on Interactive Design and Manufacturing (IJIDeM)* 11 (Aug. 2017). DOI: 10.1007/s12008-016-0356-x.

[39] *rf20*. 2016. URL: http://wiki.ros.org/rf2o (visited on 07/11/2024).

[40] Roman Adámek et al. "Analytical Models for Pose Estimate Variance of Planar Fiducial Markers for Mobile Robot Localisation". In: *Sensors* 23.12 (2023). ISSN: 1424-8220. DOI: 10.3390/s23125746. URL: https://www.mdpi.com/1424-8220/23/12/5746.

[41] Kenneth R. Castleman. "Chapter Four - Geometric Transformations". In: *Microscope Image Processing (Second Edition)*. Ed. by Fatima A. Merchant and Kenneth R. Castleman. Second Edition. Academic Press, 2023, pp. 47–54. ISBN: 978-0-12-821049-9. DOI: https://doi.org/10.1016/B978-0-12-821049-9.00005-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128210499000058.

[42] *robot_localization*. 2024. URL: https://github.com/cra-ros-pkg/robot_localization (visited on 07/11/2024).

[43] *map_server*. 2024. URL: https://wiki.ros.org/map_server (visited on 07/11/2024).

[44] *Numpy libray*. 2024. URL: https://numpy.org/ (visited on 07/11/2024).

[45] *Matplotlib libray*. 2024. URL: https://matplotlib.org/stable/ (visited on 07/11/2024).

Acknowledgements

I would like to reserve this final section to say thank you to all the people who helped me through this journey and with the completion of this thesis.

I must start with my mum and dad, who always supported me and made me believe that I could achieve any goal I set for myself. I wouldn't be here if it weren't for them.

A special thanks goes to my two supervisors: professor Marcello Chiaberge and Thomas T. Thayer. Despite their many commitments, they always tried to give their utmost to help and guide me in my thesis work, making it possible to achieve all the results.

I must also mention Maurizio and Daniele for giving me enormous emotional support and helping me during my time at SLAC. Moreover, I want to say thank you to all my colleagues and superiors for always being nice to me and making me feel welcome. A special thanks goes to Alessandro Ratti that contribuited to make all of this possible.

Finally, a huge thanks and hug go to all my friends: those I've known all my life and those I've met during my university journey. Thank you for cheering me on and making me feel your support.

I am deeply grateful to everyone who has been part of my life; your support has been invaluable.

Valeria