

LiDAR Characterization and Localization in Robot@Factory 4.0: A Cost- Effective Approach

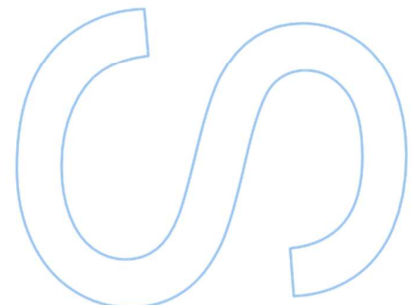
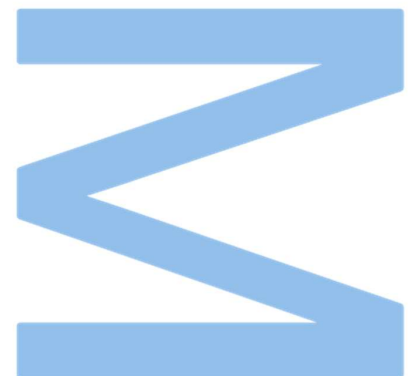
Nuno Duarte Parente

Master in Engineering Physics

Department of Physics and Astronomy

Faculty of Sciences of University of Porto and Faculty of Engineering of
University of Porto

2026



LiDAR Characterization and Localization in Robot@Factory 4.0: A Cost- Effective Approach

Nuno Duarte Parente

Dissertation carried out as part of the Master in Engineering
Physics
Department of Physics and Astronomy
2026

Supervisor

Paulo Costa, Associate Professor, Faculty of Engineering of
University of Porto

Co-supervisor

José Lima, Coordinating Professor, Polytechnic Institute of
Bragança

Acknowledgements

I would like to express my sincere gratitude to everyone who contributed, in any way, to the completion of this dissertation.

First, I would like to thank my supervisors, Paulo Costa and José Lima, for their continuous guidance, patience, availability and time they dedicated throughout this work. Their valuable insights and willingness to clarify every question were instrumental in overcoming the challenges encountered during this project.

I am deeply grateful to my family for their unconditional support and constant encouragement throughout this journey. Their confidence in me gave me the strength to persevere during the most demanding moments.

Finally, I would like to thank my friends for their motivation, companionship and for always being there to help me maintain a healthy balance throughout this process. Their support, positivity and shared moments made this journey both more enjoyable and more rewarding.

To all of you, thank you for your invaluable support.

Resumo

Estimação precisa e fiável da pose é um elemento fundamental para a navegação autónoma de *robots* móveis. Muitas técnicas de localização dependem de sensores dispendiosos ou de algoritmos computacionalmente exigentes, pelo que esta dissertação propõe um sistema de localização simples e de baixo custo, concebido para plataformas com recursos computacionais limitados em ambientes estruturados. Com este fim em vista, a pista da competição Robot@Factory 4.0 é utilizada como ambiente de teste.

O sistema proposto é constituído por três partes principais. Numa primeira fase, dois sensores LiDAR de baixo custo foram caracterizados quanto aos seus protocolos de comunicação, exatidão e precisão das medições de distância e resolução angular. Em seguida, foi desenvolvido um algoritmo de deteção de linhas baseado na segmentação da nuvem de pontos e na análise da linearidade através dos valores próprios da matriz de covariância, permitindo extrair as paredes do ambiente como características lineares. Por fim, foi proposto um algoritmo de localização baseado em associação de características, que associa as paredes detetadas a um mapa de referência conhecido através de duas estratégias complementares: um modo "Localized", computacionalmente eficiente, utilizado quando a estimativa anterior da pose é considerada fiável; e um modo "Lost", mais robusto, destinado à recuperação após erros significativos de localização. As correspondências obtidas são fundidas numa única estimativa da pose recorrendo a um Filtro de Kalman Estendido (EKF) com atualizações sequenciais e validação através da distância de Mahalanobis.

O sistema completo foi implementado num *robot* de tração diferencial controlado por um Raspberry Pi Pico W, explorando a sua arquitetura de dois núcleos para separar a receção em tempo real dos dados LiDAR da estimação de pose e controlo do movimento. O desempenho do sistema foi avaliado através de um conjunto de ensaios experimentais, incluindo a convergência a partir de desvios artificiais da pose, a análise da covariância da estimação ao longo da pista, a estabilidade durante o seguimento de trajetórias simples e complexas, a recuperação num cenário de "kidnapped robot" e a influência da distorção causada pelo movimento do LiDAR para diferentes velocidades de translação e rotação.

Os resultados demonstram que o algoritmo de localização proposto converge de forma rápida e fiável para a pose real do robô, mantendo uma reduzida incerteza durante o movimento e recuperando eficazmente de falhas significativas de localização, com um tempo de execução inferior a 20 ms por varrimento LiDAR. Estes resultados evidenciam

que é possível alcançar uma localização precisa e em tempo real em ambientes estruturados utilizando sensores de baixo custo e algoritmos simples e computacionalmente eficientes.

Palavras-chave: robótica móvel, localização, EKF, LiDAR, optimização, controlo, sistemas de baixo custo, associação de características

Abstract

Accurate and reliable pose estimation is fundamental for autonomous mobile robots. While many localization techniques rely on expensive sensors or computationally demanding algorithms, this dissertation proposes a lightweight, low-cost localization framework designed for platforms with limited computational resources, in structured environments. For this reason, the Robot@Factory 4.0 competition track was used as testing environment.

The proposed framework is built on three stages. First, two low-cost LiDAR sensors were characterized in terms of communication protocol, distance accuracy and precision, and angular resolution. Second, a line detection algorithm is proposed, based on point cloud clustering and linearity analysis using the covariance matrix eigenvalues, allowing the walls of the environment to be extracted as linear features. Third, a feature-based localization algorithm was proposed, matching detected walls to a known reference map through two complementary strategies: a computationally efficient "Localized" mode used when the previous pose estimate is reliable, and a more robust "Lost" mode used for recovery from large pose errors. Matched walls are fused into a single pose estimate using an Extended Kalman Filter with sequential updates, with Mahalanobis distance gates.

The complete system was implemented on a differential-drive robot controlled by a Raspberry Pi Pico W, using its dual-core architecture to separate time-critical LiDAR reception from pose estimation and motion control. The framework was evaluated through a comprehensive set of experimental tests: convergence from artificially induced pose deviations, pose estimation covariance across the track, trajectory-following stability over simple and complex laps, recovery from a "kidnapped robot" scenario and the influence of LiDAR motion distortion at different translational and rotational velocities.

The results show that the proposed localization algorithm is capable of converging rapidly and reliably to the robot's true pose, maintaining low uncertainty during motion and recovering from significant localization failures, all while execution under 20 ms per scan. These results demonstrate that accurate, real-time localization can be achieved in structured environments using inexpensive sensors and simple, computationally efficient algorithms.

Keywords: mobile robotics, localization, EKF, LiDAR, optimization, control, low-cost systems, feature matching

Table of Contents

List of Tables	ix
List of Figures	x
List of Abbreviations	xiii
1. Introduction	1
1.1. Background	2
1.2. Motivation and Objectives	3
1.3. Contributions	4
1.4. Structure	4
2. State of the Art	5
2.1. Sensors in mobile robotics	5
2.2. Mobile robot localization	6
2.2.1. Localization in known environments	7
2.2.1.1. Scan-Matching	7
2.2.1.2. Feature-Matching	8
2.3. EKF	8
2.4. Line detection	9
3. Robotic Platform	11
3.1. Kinematic Model	12
3.1.1. Wheel Linear Velocity	12
3.1.2. Robot Motion	13
3.2. Odometry Estimation	14
3.2.1. Robot translation and rotation	14
3.2.2. Position integration	15
3.2.3. Justification of the Midpoint Integration Scheme	16
3.3. Communication Architecture	16
3.3.1. Protocol Format	17
3.3.2. Output Buffer	17
3.3.3. Sending Commands and Tokens	17
3.3.4. Receiving and Parsing Commands	17
3.3.5. Command Registry	18
3.3.6. File Persistence and Logging	19
3.3.7. Data Processing on the Computer	19

3.4. Firmware Architecture.....	20
3.4.1. Odometry measurement and update	21
3.4.1.1. "PicoEncoder" Library.....	21
3.4.1.2. "Robot" Library	22
3.4.1.3. "Pico4Drive" Library.....	23
3.5. Motion Control.....	24
3.5.1. Implementation of the kinematic model	24
3.5.1.1. Control State Machine	24
3.5.2. DC Motor Characterization	25
3.5.2.1. Dead zone	26
3.5.2.2. PID.....	27
3.5.3. Motion procedure	29
3.5.4. Trajectory control law	30
3.5.5. Top-hat velocity filter	31
3.5.5.1. Implementation	32
3.5.5.2. Velocity robustness	33
3.5.6. Control parameters	34
3.5.7. "Set theta" Mode	34
3.5.8. "Go to point" Mode	36
3.5.9. "Follow Line" Mode	37
3.5.10. "Follow Circle" Mode	39
3.5.11. "Do Laps" Mode	41
3.5.12. "Complex Laps" Mode.....	42
3.6. Platform cost estimation	44
4. LiDAR Characterization.....	45
4.1. LiDAR Ranging Sensors.....	45
4.1.1. HLS-LFCD2.....	46
4.1.2. Delta-2A	48
4.2. LiDAR Data Reception.....	50
4.3. Processing LiDAR Data	51
4.4. Experimental Setup	53
4.5. Results : HLS-LFCD2.....	55
4.6. Results : Delta-2A	57
4.6.1. Distance Calibration.....	57
4.6.2. Voltage Calibration.....	58
4.7. LiDAR comparison.....	59

5.	Line Detection Algorithm	61
5.1.	Methodology	61
5.1.1.	Clustering	61
5.1.2.	Cluster Linearity	62
5.1.3.	Non-validated clusters	62
5.1.4.	Storing walls	63
5.2.	Experimental Setup	63
5.3.	Results : HLS-LFCD2	64
5.4.	Results : Delta-2A	65
5.5.	LiDAR comparison	66
6.	Localization Algorithm	67
6.1.	Methodology	67
6.1.1.	Length Filtering	67
6.1.2.	Matching: "Localized" Mode	68
6.1.2.1.	Number of matches required	69
6.1.3.	Matching: "Lost" Mode	70
6.1.3.1.	Perpendicular distance	70
6.1.3.2.	Relative Comparisons	71
6.1.3.3.	Pair with opposite types	71
6.1.3.4.	Pair with the same type	72
6.1.4.	EKF	72
6.1.5.	Implementation	73
6.2.	Experimental Setup	73
6.2.1.	Convergence test	74
6.2.2.	Covariance test	74
6.2.3.	Simple laps test	76
6.2.4.	Complex laps test	76
6.2.5.	Kidnapping test	76
6.3.	Results	77
6.3.1.	Convergence test	77
6.3.2.	Covariance test	80
6.3.3.	Simple laps test	80
6.3.3.1.	Trajectory Following	80
6.3.3.2.	Matching	82
6.3.3.3.	Execution time	84
6.3.4.	Complex laps test	85
6.3.4.1.	Execution time	86
6.3.5.	Kidnapped robot test	87

7. Spatio-Temporal Registration	89
7.1. Experimental Setup	89
7.1.1. Translation Test	89
7.1.2. Rotation Test.....	89
7.2. Results	90
7.2.1. Translation Test	90
7.2.2. Rotation Test.....	91
8. Conclusion.....	93
8.1. Limitations and Future Work.....	94
Bibliography	94
A. Appendix.....	99
A.1. Algorithm Pseudocodes	99
A.1.1. LiDAR data buffering.....	99
A.1.2. LiDAR data processing	100
A.1.3. Line detection	102
A.1.4. Localization: Localized mode	103
A.1.5. Localization: Lost mode	104
A.1.6. Complete Localization algorithm.....	105
A.2. Relevant figures	107

List of Tables

1.1. Positions, lengths and orientations of machine and warehouse walls. The reference frame has its origin in the center of the track and orientation is measured relatively to the positive X axis.	2
3.1. States of <code>main_fsm_t</code> used in this thesis.	25
3.2. PID parameters tuned for each robot configuration depending on the LiDAR sensor used.	28
3.3. Summary of control parameters used across all motion modes.	34
3.4. Definition of the segments of the simple rectangular trajectory. Motion modes are labeled as: 1 - Go to Point, 2 - Follow Line, 3 - Follow Circle.	42
3.5. Definition of the segments of the complex trajectory. Motion modes are labeled as: 1 - Go to Point, 2 - Follow Line, 3 - Follow Circle.	43
3.6. Estimated cost of the robotic platforms.	44
4.1. Summary of the LiDAR data packet structure.	47
4.2. Summary of the Delta-2A LiDAR data packet structure. The payload starts after the payload length is indicated, from byte 7 up to (but not including) the checksum bytes.	49

List of Figures

1.1. Computational simulation of the RAF4 official track. Many ArUco markers are scattered across the track and can be used for localization.	2
1.2. Distances and geometric proportions of the RAF4 track.	2
3.1. Communication and hardware architecture of the experimental platform implemented.	11
3.2. Robots used in the experimental evaluation of the proposed framework, with the two LiDAR configurations. The robots purposely have the exact same structure and dimensions, to more evenly compare their results.	11
3.3. Kinematic model of a differential-drive robot. The midpoint of the wheel axle, shown in red, is considered the robot's reference point.....	12
3.4. Diagram representing a wheel's angular and linear velocities. Here, i represents an arbitrary wheel.	12
3.5. Pico4Drive board used in development. This board facilitates the control of the DC Motors and odometry estimation.	23
3.6. Schematic of the motion control cycle, performed in every control frame. The "picoEncoder" library is used to measure the wheels' velocities. The odometry equations are applied in "robot.cpp" to obtain the robot's estimated velocities v, ω . The "Pico4Drive" library is used to apply the wheel voltages.	24
3.7. Dead zone analysis performed for both motors. Notably, a dead zone is evident in red.....	26
3.8. Results measured with the defined PID controller. Using the robot's state 102, the angular velocity of each wheel was defined. Then, the estimated angular velocity and voltage to the wheel's motor were recorded.	28
3.9. Results measured with the defined PID controller. Using the robot's state 102, the angular velocity of each wheel was defined. Then, the estimated angular velocity and voltage to the wheel's motor were recorded.	29
3.10 General profile of a tophat velocity filter. In this example, $D_{\text{ramp}} = 0.2$ m, $v_{\text{nom}} = 1$ m/s and $v_{\text{min}} = 0.1$ m/s.	31
3.11 Diagram showing the variables used by calculations in "Go to point" mode and how the angular error is visualized.	36
3.12 Diagram showing the variables used by calculations in "Follow Line" mode and how each error is visualized.	37
3.13 Diagram showing the variables used by calculations in "Follow Circle" mode and how each error is visualized.	39
3.14 Representation of the simple trajectory planned.	41
3.15 Representation of the simple trajectory planned.	43
4.1. Experimental setup used to calibrate the two LiDAR sensors tested.....	54
4.2. Error and Standard deviation measured without calibration. The yellow and orange bands represent the precision or accuracy indicated in the datasheet for distances below and above 50 cm, respectively. Since the datasheet defines precision as a 2σ interval (corresponding to a 95% confidence level assuming normally distributed data), the plotted thresholds are shown as half of the indicated precision values to allow direct comparison with the measured standard deviation (σ).	55

4.3. Distance measurement error and standard deviation after calibration as functions of the real tested distances. The region marked in red indicates the distance interval with significantly higher error and variability, revealing reduced measurement accuracy and precision in that range.....	56
4.4. Error and Standard deviation measured with the Delta-2A without calibration. No expected precision values were provided by the datasheet.....	57
4.5. Evolution of four main quantities as the Delta-2A DC motor voltage is increased.....	58
4.6. Comparison between the two parameters chosen to decide the optimal voltage value for the Delta-2A DC motor.....	59
4.7. Comparison of the Error and Standard deviation obtained with both LiDAR sensors.....	60
5.1. Schematic representing the functioning of the line detection algorithm.....	61
5.2. Comparison of the Average and Standard deviation of the wall length errors.....	64
5.3. Comparison of the Average and Standard deviation of the wall length errors.....	65
5.4. Comparison of the Average and Standard deviation of the wall length errors for both LiDAR models.....	66
6.1. Schematic representating the localization algorithm. It is performed for each LiDAR scan measured.....	67
6.2. Figure displaying the tested Positions 1,2 and 3, as well as the 8 pose deviation directions applied.....	74
6.3. Map of the RAF4 track, with the points used in the covariance test marked.....	75
6.4. Exponential decay of the pose estimation error for different initial deviation magnitudes in the $(1, 0)$ direction. The localization algorithm successfully converged for all tested deviation magnitudes at both positions, at different rates.....	77
6.5. Exponential decay of the pose estimation error of the robot in Position 2, with varying deviation intensities in the $(\sqrt{2}/2, -\sqrt{2}/2)$ direction. At higher magnitudes, the pose estimation converged to the wrong pose, making the error stabilize at high values....	78
6.6. Trajectory described by consecutive EKF updates, as the robot tries to recover from the induced pose deviation. The robot is in Position 2.....	79
6.7. Polar plots representing the convergence time τ estimated by exponential fitting in each test run. In the polar plot, the angle indicates the direction in which the pose deviation was induced, with the radii indicating the magnitude of the deviation induced. The point's colors indicate the τ estimated, crosses mark where the EKF did not converge to the correct pose.....	79
6.8. Variance fields recorded for each pose estimation routine. A square root scale was used to better visualize all plots. Data was recorded in the places marked with black dots, the heatmap was generated by performing linear interpolation.....	80
6.9. Poses estimated over the course of one simple lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.....	81
6.10 Estimated trajectory and uncertainty ellipses computed (95% confidence). The arrows indicate the direction of motion, that coincides with the increase of the ellipses' size.....	82
6.11 Plots showing the points (blue) where each wall (red) was observed and used in matching. This data was obtained by combining the data of the ten laps performed..	83
6.12 Execution time for each component of the localization algorithm, recorded over the duration of ten simple laps.....	84
6.13 Poses estimated over the course of one complex lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.....	85

6.14	Estimated trajectory and uncertainty ellipses computed (95% confidence). The arrows indicate the direction of motion, that coincides with the increase of the ellipses' size.	86
6.15	Execution time of each component of the localization framework, over the course of ten complex laps.	87
6.16	Trajectory estimated by the robot during the kidnapped robot test. The robot was picked up in the Capture point and quickly placed in the Release point. The trajectory is performed clockwise.	87
7.1.	Trajectories estimated over the ten laps performed at each linear velocity tested. When starting the trajectory at 300 mm/s, the robot suffered a strong deviation, visible as the isolated grey line.	90
7.2.	Percentage of frames with EKF corrections during the laps performed at each velocity. Only scans taken while the robot was in motion were considered, in order to properly isolate the skewing phenomenon.	91
7.3.	Percentage of frames in which EKF updates were performed, for each of the three positions tested, as a function of angular velocity.	91
A.1.	Exponential converge parameters A, τ for all pose deviations applied with the robot in Position 1, represented in Figure 6.2. Each plot corresponds to the parameters estimated for each deviation magnitude along a certain direction. When the amplitude does not match the ideal amplitude value for that magnitude, the data from the first scan of the convergence may be missing or some convergence had already happened.	108
A.2.	Exponential converge parameters A, τ for all pose deviations applied with the robot in Position 2, represented in Figure 6.2. Each plot corresponds to the parameters estimated for each deviation magnitude along a certain direction. When the amplitude does not match the ideal amplitude value for that magnitude, the data from the first scan of the convergence may be missing or some convergence had already happened.	109
A.3.	Exponential converge parameters A, τ for all pose deviations applied with the robot in Position 3, represented in Figure 6.2. Each plot corresponds to the parameters estimated for each deviation magnitude along a certain direction. When the amplitude does not match the ideal amplitude value for that magnitude, the data from the first scan of the convergence may be missing or some convergence had already happened.	110
A.4.	Poses estimated over the course of one simple lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.	111
A.5.	Poses estimated over the course of one complex lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.	112

List of Abbreviations

- CCD** Charge-coupled device. 6
- CMOS** Complementary Metal-Oxide-Semiconductor. 6
- DC** Direct Current. xi, 11, 23, 25, 26, 28, 45, 48, 54, 58, 59
- EKF** Extended Kalman Filter. xi, xii, 1, 4, 8, 9, 21, 67, 70, 72–76, 78, 79, 81, 82, 84, 85, 88–91, 93, 94, 111, 112
- FSM** Finite State Machine. 24, 25, 41
- GPIO** General-Purpose Input/Output. 20, 24
- GPS** Ground Positioning System. 1, 5
- ICP** Iterative Closest Point. 7
- INS** Inertial Navigation System. 5
- LiDAR** Light Detection and Ranging. x, xi, 1–9, 20, 21, 24, 44–46, 50–54, 57, 60, 61, 65–67, 73, 76, 80, 81, 84, 89–91, 93, 94
- MCL** Monte Carlo Localization. 7
- PCA** Principal Component Analysis. 63
- PI** Proportional-Integral controller. 27
- PID** Proportional-Integral-Derivative controller. x, 20, 27–29
- PIO** Programmable Input/Output. 21, 22
- PWM** Pulse Width Modulation. 24, 48
- RAF4** Robot@Factory 4.0. x, xi, 1–4, 24, 53, 63, 65, 67, 70, 75, 80, 92
- RANSAC** Random Sample Consensus. 9
- SLAM** Simultaneous Localization and Mapping. 6, 94

UART Universal Asynchronous Receiver-Transmitter. 11, 46, 48, 50, 53

UDP User Datagram Protocol. 16, 17, 19, 21

UKF Unscented Kalman Filter. 9

1. Introduction

Mobile robotics is an increasingly important field of research. Robots of this kind are widely used in warehouses and industrial facilities, where they are used to transport payloads autonomously. To perform these tasks effectively, an accurate and reliable estimation of the robot's pose is essential.

There are multiple ways to localize a robot within its environment. The robot's motion can be tracked using wheel encoders or inertial sensors, while its heading can be estimated using gyroscopes. The Ground Positioning System (GPS), together with compasses, provides a suitable option for outdoor environments. Finally, range sensors and computer vision represent software-based solutions for estimating the robot's pose. Multiple methods can be combined to achieve more accurate, robust estimation using an Extended Kalman Filter (EKF).

In the specific context of industrial robots, GPS and compasses are often unsuitable due to indoor operation and electromagnetic interference. Furthermore, industrial environments often contain many straight walls and corridors, which make line-based localization approaches particularly effective. This enables pose estimation using inexpensive ranging sensors. By extracting linear features from ranging data and matching them to a known map of the environment, the robot's pose can be estimated in a simple and computationally efficient way.

This dissertation proposes a localization framework consisting of Light Detection and Ranging (LiDAR) characterization, line detection in LiDAR data, and feature-based localization. The proposed work is designed for low-cost robotic platforms, aiming at computational efficiency and algorithmic simplicity. The system was implemented and tested in the context of the Robot@Factory 4.0 (RAF4) competition [1], where an industrial environment is simulated on a reduced scale.

Beyond its industrial applications, the proposed framework also has educational value. Due to its low computational requirements and use of inexpensive hardware, it can be used as a didactic platform for teaching robotics, localization and feature extraction concepts.

1.1. Background

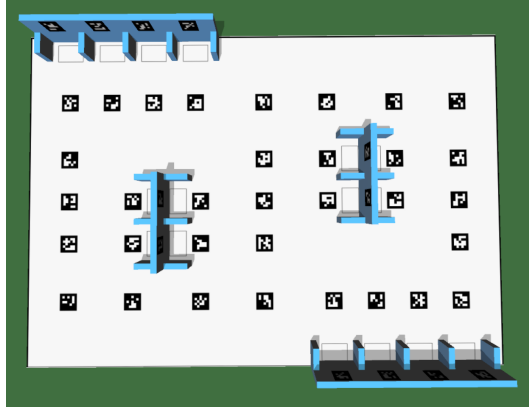


Figure 1.1: Computational simulation of the RAF4 official track. Many ArUco markers are scattered across the track and can be used for localization.

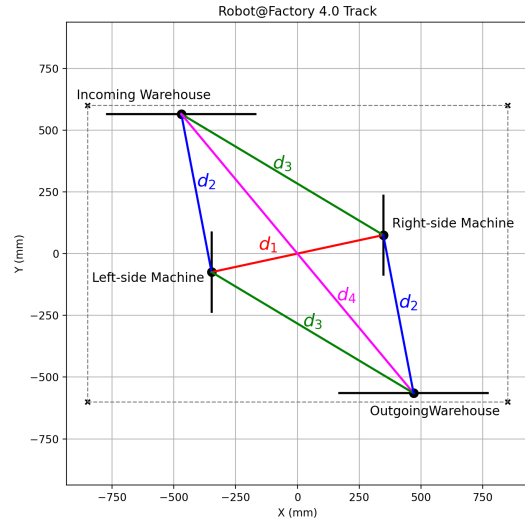


Figure 1.2: Distances and geometric proportions of the RAF4 track.

The proposed algorithms were implemented and tested in the context of the RAF4 competition. Figure 1.1 shows a computational simulation of the official track layout [1]. Vertical walls represent machines, while horizontal walls represent warehouses. The incoming warehouse is in the top left corner and the outgoing warehouse is in the bottom right corner. There are two machine zones: left-side and right-side, with two machines in each. A box is placed on the left side of a machine and retrieved as a "processed" box on the right side. The two warehouses and the two machine zones form four walls, shown in the figure. For simplicity, these walls will be referenced as "warehouse walls" and "machine walls", respectively.

The competition consists of three phases, in which autonomous robots must transport boxes between machines and warehouses, sometimes having to deliver and retrieve boxes in multiple machines.

Type	Name	Center X (mm)	Center Y (mm)	Length (mm)	Orientation (°)
Machines	Left-Side	-347.5	-75	32	90
	Right-Side	347.5	75	32	90
Warehouses	Incoming	-470	565	62	0
	Outcoming	470	-565	62	0

Table 1.1: Positions, lengths and orientations of machine and warehouse walls. The reference frame has its origin in the center of the track and orientation is measured relatively to the positive X axis.

The walls are tall enough that they can be detected by a LiDAR sensor, capturing the lines necessary for pose estimation. With this goal, special attention should be given

to the geometrical properties of the track, outlined in Figure 1.2. The layout has a 180 degree symmetry and machine and warehouse walls consist of two pairs of parallel lines of the same length, placed in symmetrical positions shown in Table 1.1. Additionally, the distance between a pair of wall centers can only take one of four values: d_1 to d_4 . These characteristics play a central role in the proposed localization algorithm. When necessary, the machines will be called by the names shown in this section: incoming warehouse, outgoing warehouse, left-side machines and right-side machines.

1.2. Motivation and Objectives

Autonomous localization is currently one of the most important problems in mobile robotics. As will be explored in chapter 2, many localization techniques exist, although several of them require expensive sensors, complex algorithms or high computational power. For these reasons, most of these methods may not be suitable for low-cost robotic platforms or educational environments.

The RAF4 competition presents an environment specially suited for studying line detection, feature-based localization methods as the track contains multiple linear structures with known geometrical properties. All of this enables pose estimation with simple LiDAR measurements and computationally efficient algorithms. Moreover, the symmetry and repetitive nature of the environment introduce challenges regarding feature association and ambiguity resolution, which can represent problems often found in industrial environments.

The main objective of this dissertation is the development of a lightweight localization framework for indoor mobile robots operating in previously known structured environments.

The specific objectives of this work are:

- Characterize the performance of low-cost LiDAR sensors, comparing different sensors;
- Develop an efficient algorithm for line extraction from LiDAR data, evaluating its performance with different LiDAR sensors;
- Implement a feature-based localization method using geometric matching between detected lines and a reference map;

- Evaluate the accuracy and robustness of the proposed localization framework in experimental tests in the official RAF4 track.

1.3. Contributions

From following these objectives, this work makes the following contributions to the field of low-cost mobile robot localization:

- A comparative characterization of two low-cost LiDAR sensors;
- An efficient line detection algorithm based on clustering and linearity analysis;
- A dual-mode feature-matching localization framework implemented with an EKF;
- An extensive experimental evaluation, including convergence, covariance and performance over long sequences of operation.

1.4. Structure

Due to the interconnected nature of the topics presented in this dissertation, it follows an unusual structure. First, the LiDAR characterization is presented, including the methodology and results. Next, the line detection algorithm, which builds upon the LiDAR characterization, is introduced along with the experimental tests performed and their results. Afterward, the localization algorithm is explored, evolving from the line detection stage, and its experimental results are presented. To conclude, the main findings are summarized.

Associated Publication

This dissertation is associated with a paper accepted for presentation at the Controlo 2026 conference. The conference paper was developed in parallel with this dissertation and presents an earlier and more concise version of the work.

Although the dissertation and the conference paper naturally share the same scientific background, objectives, methodology and results, the dissertation was written independently and does not constitute a reproduction or an adaptation of the conference paper.

The conference paper was written by Nuno Parente, who was responsible for the implementation, experimental work, data analysis and manuscript writing. The co-authors, Prof. Paulo Costa and Prof. José Lima, contributed through scientific supervision, discussion of the methodology and critical revision of the manuscript.

2. State of the Art

Robot localization in structured indoor environments is a widely studied problem in mobile robotics. This section introduces some relevant work on localization and line detection.

2.1. Sensors in mobile robotics

Aqel et al. [2] presents a review of various techniques of visual odometry, while Liu et al. [3] provides an extended review of "Sensing Technologies for Indoor Autonomous Mobile Robots". Some of the types of sensors studied in these articles will be briefly explored below.

The simplest and most accessible option is wheel odometry, using wheel encoders that track the wheel's revolutions. However, these sensors suffer from drift and inherent error.

A commonly used and highly effective localization technology is GPS, in which a receiver estimates its position using signals transmitted by at least four satellites in orbit. These systems can achieve high accuracy, but they are generally unsuitable for indoor environments, where satellite signals are often weak or unavailable.

Another option that is commonly used is Inertial Navigation System (INS). These systems use rotation sensors (like gyroscopes) and motion sensors (like accelerometers) to estimate the robot's angular velocity and acceleration, respectively. Through integration, the robot's position and orientation can be estimated relatively to its original state. However, this solution is also affected by drift: motion measurements contain errors, which are exacerbated by mathematical integration.

These methods are often inexpensive and efficient, but tend to accumulate error. For this reason, most systems combine them with additional, more complex sensors, especially those that sense the environment surrounding the robot. One example of this is ranging sensors: they allow distance measurements between the robot and its surrounding environment, which enable pose estimation and even mapping. Ultrasonic sensors rely on sound emission and reflection to measure distance. They are versatile, but are relatively imprecise and unreliable at long distances due to signal degradation. LiDAR sensors contain laser beams and use their reflections to measure distance. They usually have higher sample rates and greater precision, being one of the most used options in indoor

localization problems. The biggest limitation of LiDAR is the inability to measure in environments with glass or reflective surfaces, which can cause unexpected behavior in the laser beams.

The review also explores vision sensors, based on image capture with Charge-coupled device (CCD) or Complementary Metal-Oxide-Semiconductor (CMOS) sensor. The paper explores how captured images can be undistorted and aligned to enable feature matching. These sensors, besides localization, enable complex 3D mapping of entire environments. However, they often carry the highest computational effort, which requires more expensive and powerful systems.

In conclusion, there are multiple types of sensors, with varying accuracy, simplicity, financial and computational cost. Overall, the articles consulted indicate that LiDAR is a consistent and promising technology that allows efficient measurements.

2.2. Mobile robot localization

A significant portion of the literature focuses on Simultaneous Localization and Mapping (SLAM) [4]. This problem involves a robot operating in an initially unknown environment. Using various types of sensors, the robot builds a map of its surroundings while simultaneously estimating its own position and motion. Whenever it revisits previously mapped regions, a loop closure happens: revisiting a region increases the confidence in the pose estimation, which reduces the estimation error.

Karto SLAM [5] is a 2D SLAM approach in which the problem is formulated as a pose graph. Each node in the graph represents a robot pose, while edges represent spatial constraints derived from LiDAR scan matching and odometry measurements. As the robot moves, the graph is incrementally built with new poses and constraints being added. When a loop closure happens, a complete optimization of the graph is performed. Karto SLAM produces consistent maps even over long trajectories, making it suitable for larger indoor environments.

Hector SLAM [6–8], is a LiDAR-based SLAM method that relies primarily on matching of consecutive high-frequency LiDAR scans. It estimates the robot pose by aligning incoming laser scans with an occupancy grid map using iterative optimization techniques. This technique does not have a strong reliance on odometry data, making it suitable for systems with unreliable odometry. As it does not keep and optimize a graph, this method tends to be more effective and less computationally expensive.

Xie et al. [9] presents an alternative solution, that used position-encoded landmarks, instead of ranging point clouds. This method uses LiDAR scans to detect landmarks that encode 4-bit data about their relative position. This enables localization without the need of high-density, high-accuracy LiDAR scans and respective scan-matching. However, this approach requires previous preparation, including assembly and placement of the landmarks.

2.2.1 Localization in known environments

2.2.1.1 Scan-Matching

Scan matching methods estimate the robot pose by aligning the current sensor observation with a reference map of the environment. Unlike particle-filter based approaches, which evaluate many possible poses, scan-matching searches for the transformation that best explains the observed scan.

Ying and Sun [10] proposed an improved Monte Carlo Localization (MCL) method that incorporates an optimized Iterative Closest Point (ICP) algorithm. The method uses ICP to estimate the relative pose between the current LiDAR observation and the reference scan, providing a more accurate measurement for the MCL process. The experimental results show that the combination improves localization accuracy when compared to MCL or ICP-based localization alone. This work illustrates how scan matching and probabilistic localization can be combined, with the particle filter providing robustness to uncertainty and ICP providing accurate local pose refinement.

Filothéou [11] proposed a different approach to 2D LiDAR localization based on correspondence - free scan-to-map matching. The method assumes a known 2D map and compares the LiDAR scan acquired by the robot with a virtual scan generated from the map at a candidate robot pose. Instead of establishing correspondences between individual measurements in the two scans, the method evaluates the similarity between the scans. The approach is particularly suited to 2D LiDAR localization in environments where establishing reliable point correspondences may be difficult.

Although scan-matching approaches can provide accurate pose estimates, their computational cost can become significant when processing high resolution scans, with high numbers of points. Then, methods employing more compact representations of the environment must be studied.

2.2.1.2 Feature-Matching

Feature-based localization determines certain characteristic geometric features of the environment and uses them to estimate the robot's pose, by comparing the features with a known map. This reduces the amount of information involved in the matching process while retaining essential geometric characteristics relevant for pose estimation.

Hendrikx et al. [12] proposes a feature-based localization method using 2D LiDAR and a geometric model of the environment. The local observations are first combined into local hypotheses, which are then matched against the features present in the known map. Multiple data-association hypotheses are maintained and validated through a registration step, allowing the method to handle correspondence ambiguities. The experimental results demonstrated that the proposed feature-based approach could achieve robust localization while maintaining a relatively small number of hypotheses.

Wang et al. [13] proposed a lightweight LiDAR-based localization system using artificial landmarks. The method identifies reflective landmarks in the sensor measurements. The detected landmarks are matched with the known configuration of landmarks in the environment to determine the robot pose. The system incorporates landmark extraction, motion correction and a matching procedure optimized to enable real-time operation at high linear and angular velocities. Experimental results demonstrated centimeter-level localization accuracy while maintaining a lightweight computational workload. However, this approach fundamentally relies on artificial landmarks placed in the environment, which limits its applicability.

These studies demonstrate that feature-based localization can provide an effective alternative to processing complete LiDAR scans. Retaining only geometric information relevant for localization, these methods can significantly reduce the amount of data processed while still providing accurate and robust pose estimates. This approach was chosen for this work, where a good trade-off between cost and accuracy is fundamental.

2.3. EKF

The EKF [14] is a widely used Gaussian filtering method for estimating the state of nonlinear systems. It extends the standard Kalman Filter by locally linearizing the nonlinear motion and measurement models around the current state estimate. In mobile robot localization, the EKF is specially used to fuse odometry estimates with measurements from the environment, providing a pose estimate and its covariance or uncertainty.

An alternative to the EKF for nonlinear state estimation is the Unscented Kalman Filter (UKF) [15]. Instead of locally linearizing the nonlinear process and measurement models, the UKF propagates a set of deterministically selected sigma points through the nonlinear functions, which generates a higher-order approximation of the state distribution. This can provide advantages when the system exhibits strong nonlinearities. However, these advantages come at an increased computational cost, since multiple sigma points have to be propagated at each filtering step, whereas the EKF requires only the evaluation of the models and their Jacobians.

Another important aspect of EKF-based localization is the validation of measurements before they are used to update the state estimate. When a measured feature is compared to where a feature of the reference map should be observed, their difference is the innovation. It is affected both by measurement noise and uncertainty in the predicted state, so an Euclidean distance can be insufficient to determine whether a correspondence is plausible. The Mahalanobis distance [16, 17] accounts for this uncertainty by weighting the innovation with the state estimate covariance, and measures the statistical compatibility between an observation and a predicted measurement. Assuming normally distributed data, the squared Mahalanobis distance follows a chi-squared distribution, allowing a statistical threshold to be defined at a given confidence level. This gating prevents measurements that are inconsistent with the current state estimate and its uncertainty from influencing the filter.

2.4. Line detection

Nguyen et al. [18] presents and compares several approaches for line detection in LiDAR data, including Split-Merge, Incremental line detection, Random Sample Consensus (RANSAC), Linear Regression and the Hough Transform. Split-Merge methods initially divide the LiDAR scan into smaller segments and iteratively merge segments when they can be adequately represented by a common line. Incremental methods, process the measurements sequentially, extending one line as new points are added and starting a new line when the measurements no longer satisfy the required criteria. RANSAC repeatedly selects subsets of measurements and determines which points are consistent with a linear model. Finally, the Hough Transform maps the LiDAR points into a parameter space, where lines can be identified by detecting accumulations of votes corresponding to potential line parameters.

The methods evaluated in Nguyen et al. [18] provide different approaches to extracting linear features from LiDAR measurements, but their computational requirements vary considerably. Nevertheless, even the most efficient approaches involve processing and evaluating multiple measurements to identify and validate line segments.

For this work's context, where line detection is executed continuously on a resource-constrained robotic platform, reducing the computational cost of feature extraction is fundamental. Therefore, a simpler line-detection method was developed.

3. Robotic Platform

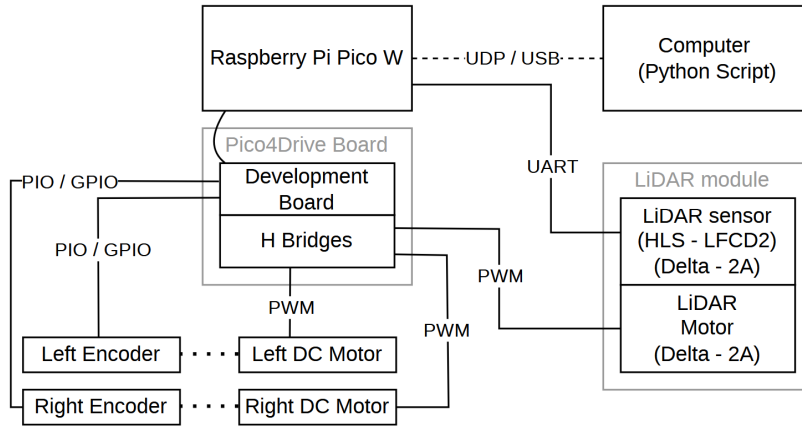
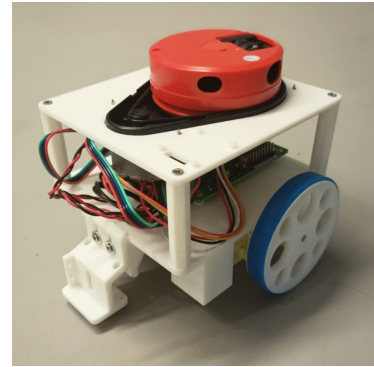


Figure 3.1: Communication and hardware architecture of the experimental platform implemented.



(a) HLS-LFCD2 LiDAR



(b) Delta-2A LiDAR

Figure 3.2: Robots used in the experimental evaluation of the proposed framework, with the two LiDAR configurations. The robots purposely have the exact same structure and dimensions, to more evenly compare their results.

To align with the goal of developing an efficient and cost-effective approach to mobile robot localization, the robots used in this work follow a simple architecture. A Raspberry Pi Pico W is mounted on a Pico4Drive development board, which includes integrated Direct Current (DC) motor drivers. The DC motors used have built-in quadrature encoders [19] and are connected directly to the board. Finally, the LiDAR is also connected to the board and interfaces via Universal Asynchronous Receiver-Transmitter (UART) communication.

This section provides a detailed explanation of the control framework used for the robot, as well as the communication architecture between the robot and the computer.

3.1. Kinematic Model

Both robots developed in this work are differential-drive mobile robots. This type of architecture has two independently actuated wheels placed along the same axis.

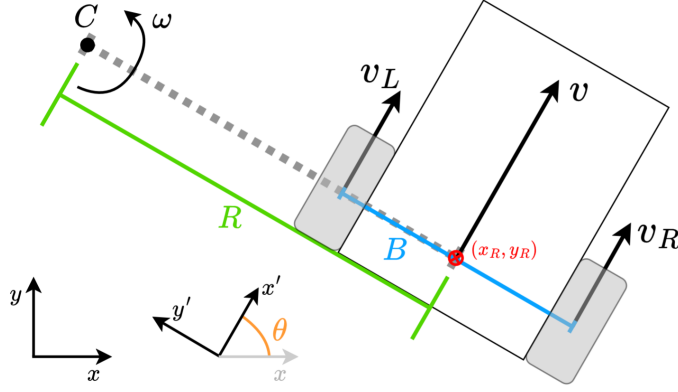


Figure 3.3: Kinematic model of a differential-drive robot. The midpoint of the wheel axle, shown in red, is considered the robot's reference point

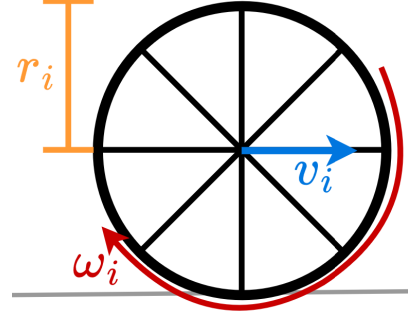


Figure 3.4: Diagram representing a wheel's angular and linear velocities. Here, i represents an arbitrary wheel.

In this model, the robot has two wheels of radius r , separated by a distance B , which corresponds to the track width of the robot. The robot's position is described in a global reference frame (x, y) , where the midpoint of the wheel axle is located at (x_R, y_R) . This will be considered as the robot's position. The robot's orientation is represented by the heading angle θ , measured relative to the x -axis of the global frame.

A body-fixed reference frame (x', y') is also considered, relative to the robot, with its origin located at the midpoint of the wheel axle. The x' -axis points in the forward direction of motion, while the y' -axis is perpendicular to it. The relationship between the body-fixed and global frames is defined by the heading angle θ , as illustrated in Figure 3.3.

Not depicted in the figures, a PTFE slider is located at the front of the robot (as seen in Figure 3.2), enabling smooth motion.

3.1.1 Wheel Linear Velocity

The motion of a differential-drive robot is completely determined by the angular velocities of its wheels. To relate the wheel angular velocity ω_i to its linear velocity v_i , it is necessary to assume that the wheel rolls without slipping.

Under this assumption, the linear velocity of the point of contact between the wheel and the ground is null with respect to the ground. The velocity of this point results from the

combination of the translational motion of the wheel center, v_i , and the tangential velocity generated by the rotation of the wheel, $r_i\omega_i$. Therefore,

$$v_{\text{contact}} = v_i - r_i\omega_i. \quad (3.1)$$

By enforcing the no-slip condition, $v_{\text{contact}} = 0$, this equation yields:

$$v_i = r_i\omega_i. \quad (3.2)$$

Therefore, the linear velocity of the wheel center is directly proportional to the wheel radius and angular velocity. From this point, this linear velocity will be considered as the wheel's velocity. In the robots used in this work, both wheels have the same radius, r .

3.1.2 Robot Motion

The angular velocity of a wheel, ω_i , can be measured with wheel encoders. By using Equation 3.2, the velocity of the wheel can be easily estimated. In each moment, a differential robot with encoders can estimate the linear velocity of the left and right wheel, v_L and v_R respectively.

The robot motion can be seen as the superposition of two components: linear and rotational. When both wheels move at the same velocity, the robot undergoes pure translation. Considering the wheels are at the same distance to the wheel axis center, the robot's linear velocity is given by the average of the wheel velocities:

$$v = \frac{v_R + v_L}{2} \quad (3.3)$$

The angular velocity of the robot is determined by the difference between the wheel velocities. If the wheel velocities differ, one wheel travels a greater distance than the other, causing the robot to rotate. The magnitude of this rotation is proportional to the velocity difference $v_R - v_L$ and inversely proportional to the wheel distance B . Therefore, the robot angular velocity is given by

$$\omega = \frac{v_R - v_L}{B}. \quad (3.4)$$

In this work, positive angular velocities are considered as counterclockwise rotations. Consequently, when $v_R > v_L$, the robot rotates counterclockwise and $\omega > 0$, defining the ordering of the terms in the numerator.

Equations 3.3 and 3.4 make it possible to estimate the robot's motion by measuring each wheel's angular velocity with wheel encoders.

3.2. Odometry Estimation

3.2.1 Robot translation and rotation

Odometry is the process of estimating the robot pose using measurements from different sensors. In this work, odometry was performed using solely wheel encoders. By integrating the wheel displacements (and corresponding velocities) over time, it is possible to obtain an estimate of the robot position and orientation relative to an initial reference frame.

In a sampling interval Δt , a wheel rotating with a velocity ω_i changes its orientation by an angle: $\Delta\phi_i = \omega_i\Delta t$. For a wheel of radius r_i , the distance Δs_i traveled in this interval is given by:

$$\Delta s_i = r_i\Delta\phi_i = r_i\omega_i\Delta t \quad (3.5)$$

Applying the same reasoning presented in subsection 3.1.2 to the left and right wheels, the translational and rotational displacements of the robot during a sampling interval can be obtained. The translational displacement is proportional to the average wheel displacement, whereas the rotational displacement is proportional to the difference between the wheel displacements.

$$\Delta s = \frac{\Delta s_R + \Delta s_L}{2} = \frac{v_R + v_L}{2}\Delta t = v\Delta t \quad (3.6)$$

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{B} = \frac{v_R - v_L}{B}\Delta t = \omega\Delta t \quad (3.7)$$

Therefore, the robot's translation and rotation in a sampling frame Δt can be estimated by measuring each wheel's rotation.

3.2.2 Position integration

During motions, the robot is considered as moving in a 2D plane, following a simple kinematic model:

$$\dot{x} = v \cos \theta \quad (3.8)$$

$$\dot{y} = v \sin \theta \quad (3.9)$$

$$\dot{\theta} = \omega \quad (3.10)$$

These equations describe the continuous-time evolution of the robot pose, although wheel encoder measurements are only available at discrete time instants. Therefore, an exact analytical integration of the system is not possible in practice and numerical integration methods must be used to estimate the continuous motion from sampled data.

Encoders are used to estimate the robot's motion in each sampling interval Δt . In an instant $t = k\Delta t$, the robot has a pose (x_k, y_k, θ_k) . During the following sampling interval, the robot's translation and rotation is measured as $\Delta s, \Delta \theta$. Using this information, the robot's position in the following instant $t' = (k+1)\Delta t$ can be predicted:

$$x(t') = x_{k+1} = x_k + \Delta s \cos \left(\theta_k + \frac{\Delta \theta}{2} \right) = x_k + v_k \Delta t \cos \left(\theta_k + \frac{\omega_k \Delta t}{2} \right) \quad (3.11)$$

$$y(t') = y_{k+1} = y_k + \Delta s \sin \left(\theta_k + \frac{\Delta \theta}{2} \right) = y_k + v_k \Delta t \sin \left(\theta_k + \frac{\omega_k \Delta t}{2} \right) \quad (3.12)$$

$$\theta(t') = \theta_{k+1} = \theta_k + \Delta \theta = \theta_k + \omega_k \Delta t \quad (3.13)$$

In practice, this is performed by accumulating the translation and rotation at each instant. In the end, the position estimated in a frame t is an integration of the robot's motion relative to the starting pose (x_0, y_0, θ_0) :

$$x(t) = x_k = x_0 + \sum_{i=0}^k \Delta s_i \cos \left(\theta_i + \frac{\Delta \theta_i}{2} \right) = x_0 + \Delta t \sum_{i=0}^k v_i \cos \left(\theta_i + \frac{\omega_i \Delta t}{2} \right) \quad (3.14)$$

$$y(t) = y_k = y_0 + \sum_{i=0}^k \Delta s_i \sin \left(\theta_i + \frac{\Delta \theta_i}{2} \right) = y_0 + \Delta t \sum_{i=0}^k v_i \sin \left(\theta_i + \frac{\omega_i \Delta t}{2} \right) \quad (3.15)$$

$$\theta(t) = \theta_k = \theta_0 + \sum_{i=0}^k \Delta \theta_i = \theta_0 + \Delta t \sum_{i=0}^k \omega_i \quad (3.16)$$

Odometry provides a simple and computationally efficient method for pose estimation, that can be calculated in real time. However, its accuracy degrades with time due to wheel slip, encoder quantization and wheel radius uncertainty. In each instant, a small

error is generated and they are integrated over time, so the pose estimate accumulates drift and is not reliable in long-term localization. For this reason, in this work, an external sensing solution will be explored, with the goal of canceling out the odometry drift.

3.2.3 Justification of the Midpoint Integration Scheme

The position equations use the robot orientation at the midpoint of the sampling interval, $\theta_k + \Delta\theta/2$, rather than the initial orientation θ_k . In other words, a Midpoint integration scheme was used in place of Forward Euler integration. This choice can be justified through a local error analysis. Over a sampling interval Δt , the true heading evolves as $\theta(\tau) = \theta_k + \omega_k \tau + O(\tau^2)$, with $\tau \in [0, \Delta t]$.

Forward Euler assumes $\theta(\tau) \approx \theta_k$ throughout the interval, ignoring the robot's rotation. This introduces a local truncation error of $O(\Delta t^2)$ per step, giving first-order global accuracy.

The midpoint scheme evaluates the heading at $\theta_k + \omega_k \Delta t / 2$, approximating the average orientation over the interval. This is the classical midpoint quadrature rule, which reduces the local error to $O(\Delta t^3)$ and yields second-order global accuracy. In other words, a midpoint integration scheme accumulates less error, with a small increase in computational effort.

Geometrically, if v_k, ω_k are constant during an interval Δt , the true path is a circular arc and the line joining its endpoints has exactly the direction $\theta_k + \Delta\theta_k/2$. The midpoint scheme therefore reproduces the true displacement exactly in this case, while Forward Euler systematically underestimates the turn, specially biasing the pose estimate when significant rotation occurs during a sampling interval.

3.3. Communication Architecture

Before exploring the motion control of the robot, this section briefly presents the communication architecture of the system used. The robot is connected to the same wireless network as a computer and they communicate via User Datagram Protocol (UDP). A serial port connection could have been used, but was not chosen during this work.

Communication between the microcontroller and the external system is handled by the `gchannels` [20] C++ file, which allows for different types of information (data, commands and log messages) to be transmitted over the same link without mixing contents.

3.3.1 Protocol Format

Commands sent in this channel follow a simple text-based format:

```
COMMAND PARAMETER;
```

where `COMMAND` is a text identifier of the command and `PARAMETER` can be a string or a numeric value (both are sent as text). A command can be followed by *tokens*, single numerical values terminated by `;` used to add values to a command.

3.3.2 Output Buffer

To minimize the overhead of sending multiple individual transmissions, bytes are accumulated in a buffer (`out_buffer`) before being sent. The function `send_char` appends a byte to this buffer and triggers `flush()` once the buffer is full. `flush()` uses a callback function provided by the user (`send_buffer`) to transmit the buffered bytes (either via `Serial.write` or `UDP`) and resets the buffer. This callback-based design separates the protocol from the transmission mechanism, enabling higher flexibility.

3.3.3 Sending Commands and Tokens

Several helper functions are present in `gchannels.cpp` [20] and have the `send_char` function as their basis. Two notable options are:

- `send_command(command, value)` transmits a command following the command text format. Two variations of this function exist, to operate with `value` as a string or a float.
- `send_token(value)` transmits a single parameter terminated by `;`. This function also has two variations, for `value` as a string or a float.

In both cases, the command is sent to the outgoing buffer using `send_char`, one character at a time.

3.3.4 Receiving and Parsing Commands

Bytes received are processed one at a time by the function `process_char`, which implements a small state machine with three states: `gs_wait_for_data`, `gs_reading_command`, and `gs_reading_token`. The functioning of this function is explained below:

- If the system is idle and an alphabetic character is received, it is start of a **command**, being part of the command label. The character is saved to a reading buffer.
- If the system is idle and a number or minus sign is received, it starts reading a **token**, as no alphabetic part is present. The character is saved to a reading buffer.
- If a backspace character is received, the last character of the buffer is erased. This functionality enables real-time Serial communication, but it was not used in this work.
- When a line terminator (LF, CR or ;) is received, the reading buffer is parsed:
 - For a `command`, the first space character separates the command label from its parameter. The command name is stored as `last_command` for potential tokens, and the parameter is parsed as text and as a float.
 - For a `token`, the value is associated with the previously received command (`last_command`), and an internal index (`frame.index`) is incremented to identify its position among multiple tokens.
- In both cases, the resulting `command_frame_t` structure is passed to a user-supplied callback function `process_command` that applies the command as desired.

3.3.5 Command Registry

This library provides a command registry to simplify the integration of the communication protocol with the robot's internal variables. Each entry is defined as a `command_item_t` that has a command label and a pointer to a variable with the command value (`float*`, `int*` or `char*`). Each command has three flags:

- `send` indicates whether the command label and its value should be transmitted periodically
- `recv` indicates if the command can update its value when it is received
- `sparse_send` decides whether the command belongs to a low-priority group that is transmitted less frequently
- `store` indicates whether the command value should be saved to persistent storage for later use

Commands are registered via various `register_command` functions, that manage different types of command values. Each command name is hashed using the MurmurOAAT [21], an implementation of Murmur Hash. The hash is stored along with the command.

When a command is received, `process_read_command` is used to process it using the Murmur hash. First, the hash of the received command is computed and compared directly to every registered command's hash. When a match is found, a direct string comparison confirms it. The variable containing the command value is updated with the received command's value, if indicated by its `recv` flag.

3.3.6 File Persistence and Logging

With `gchannels` as its base, a second C++ file `file_gchannels.cpp` provides file transmission and configuration using the `LittleFS` file system.

The `send_file` function transmits the contents of a stored file through the communication channel.

The `save_commands` function iterates over the command registry and writes the commands marked as persistent (with the `store` flag) to the configuration file, one command per line, in the format referenced above. The commands that are saved to file form a system configuration that persists across reboots, after being loaded by `load_commands`.

The `load_commands` function performs the reverse process of `save_commands`: it reads the configuration file byte by byte and processes each byte with `process_char`. That is, the configuration file is interpreted as if it was a list of commands received over the communication channel. In this way, the same parser and the same command registry can be reused for both persistent configuration and real-time communication.

3.3.7 Data Processing on the Computer

During testing and debugging, the `send_command` function is used to transmit all relevant data from the robot to the computer via UDP. The data is sent in the format `COMMAND VALUE;` as previously referenced.

On the computer side, a Python script was used to receive, interpret and visualize the data in real time. This script emulates the functioning of `process_char`. The processed data is then stored in text files using a structured format that can be easily imported using the NumPy library.

Afterwards, the data is easily accessible in other codes for analysis and study. This approach provides a practical and efficient method for visualizing and storing experimental data simultaneously. Once stored, the data can be readily accessed by other Python programs for visualization, analysis, performance evaluation and further study of the robot's behavior.

Python was chosen for the data analysis tools due to its extensive scientific computing ecosystem and the author's prior experience with the language, which allowed rapid development, testing and maintenance of the analysis scripts.

3.4. Firmware Architecture

The robot's firmware was developed in C++ using the PlatformIO ecosystem with the Arduino framework for RP2040. PlatformIO was used as a VS Code extension, to handle dependency management, compilation, binary upload via USB and real-time serial monitoring for debugging.

As is standard in microcontroller programming, the coordination of all system components is performed in the `main.cpp`. The `setup` function initializes General-Purpose Input/Output (GPIO) pins, the motor driver module, Serial and WiFi/UDP communications. Most commands are registered in this function, as detailed in subsection 3.3.5. Persistent configuration parameters are loaded from the LittleFS file system and the wheel controller Proportional–Integral–Derivative controller (PID) parameters are applied at this stage.

The RP2040 microcontroller has two independent cores and this architecture was exploited to separate time-critical LiDAR data acquisition from heavier processing tasks. Core 1 is dedicated exclusively to receiving the continuous byte stream from the LiDAR sensor, where any processing delay could cause bytes to be dropped or a lap transition to be missed. Core 0 handles all command parsing, odometry estimation, motion control and EKF updates. This separation is particularly important during development and debugging, where a large amount of debug commands can be frequently sent, in Core 0. Without this separation, a burst of debug traffic on Core 0 could stall the LiDAR reception loop long enough to corrupt a scan frame. Synchronization between the two cores is achieved through a shared lap buffer and the `lap_to_go` flag.

The `loop` function, in Core 0, has two distinct versions. In every iteration, it receives external UDP and Serial data, processing any received commands immediately. Additionally, when Core 1 signals that a complete LiDAR scan is available (via the `lap_to_go` flag),

the LiDAR data is processed with the `lidar.cpp` library and snapshotted for EKF update. Every 40 ms (25 Hz), a control cycle is performed: encoder readings are performed, odometry is updated, the motion control equations are computed and motor voltages signals are updated. If a new LiDAR lap was processed during this cycle, a full diagnostic packet including point cloud, extracted line segments, EKF covariance matrix, and execution times can be transmitted over Serial/ UDP for further analysis in a computer.

The RP2040's Core 1 is employed for real-time LIDAR data reception. The `loop1` function receives bytes from Serial2 as soon as they arrive, feeding them into a lap buffer, using the `lidar_buffer.cpp` library. When this library detects a complete lap, the raw byte stream is copied to a buffer shared between the cores and the `lap_to_go` flag is set to `true`, but only if Core 0 has already consumed the previous lap. This dual-core architecture separates the time-critical byte stream from the more costly processing tasks on Core 0, avoiding delays in receiving data.

The process of constantly accumulating LiDAR data in a buffer is performed by the `lidar_buffer` library. The processing of binary raw LiDAR data into (x, y) points in the robot's reference frame is executed by the `lidar` library. Both libraries will be thoroughly explained in chapter 4.

3.4.1 Odometry measurement and update

3.4.1.1 "PicoEncoder" Library

The first step in estimating the robot's motion through odometry is measuring the wheels' velocities using encoders. Often, rotational velocity is measured using encoders and counting the number of pulses in a certain interval. To facilitate this process, the `picoencoder` library [22] was used. It utilizes the Raspberry Pi Pico's Programmable Input/Output (PIO) to perform encoder counting and decoding, removing the need for a CPU interruption.

In the `main.cpp`, an array of `PicoEncoder` class objects is defined, one for each encoder. Then, in the `setup`, this library's `begin` function initializes the PIO for each encoder using its pin in the Pico4Drive development board.

In each iteration of the `loop` function, the library's `update` function implements a quadrature decoder optimized for the RP2040 microcontroller. Quadrature encoders often use two channels (A and B) shifted by 90° , generating four distinct electrical phases in each

physical pulse. The `PicoEncoder` algorithm uses this structure to perform temporal interpolation between encoder steps. Instead of relying only on pulse counting, which results in a discrete scale, the `update` function measures the time elapsed since the last state transition detected. This way, a continuous position estimation is obtained. The following equation is used for this interpolation:

$$P(t) = P_{\text{trans}} + \dot{P}_{\text{est}} \cdot (t - t_{\text{trans}}) \quad (3.17)$$

where P_{trans} and t_{trans} represent the position and timestamp of the last detected transition and $\dot{P}_{\text{est}}$ is the estimated velocity. To prevent integration drift and numerical errors, the estimated position is bounded by the physical constraints of the current quadrature phase:

$$P_{\text{low}} \leq P(t) \leq P_{\text{high}} \quad (3.18)$$

Where $P_{\text{low}}, P_{\text{high}}$ are computed in the `update` function. They correspond to the lower and upper spatial limits of the current quadrature phase, mapped according to the calibrated resolution of the encoder.

3.4.1.2 "Robot" Library

This library is used to define and update the odometry estimations of the robot. First, a `robot_t` structure is initialized in the `main.cpp`, containing information such as the robot's geometry (track width and wheel radius), angular and linear velocity and encoder measurements.

Then, in each `loop` iteration the `odometry` function of the library is executed. As detailed in the `PicoEncoder.cpp` [22] file comments, the encoder speed is estimated as the number of sub-steps per second. Within the library's architecture, each physical quadrature phase change is subdivided into 64 sub-steps (a 2^6 scaling is performed via a `step << 6` bitwise shift) and applied to high-resolution timing data from the `PIO`. To convert this virtual velocity into the physical angular velocity of the wheel (ω_{ie}), the encoder estimated velocity is used in Equation 3.19. Before the equation, the velocities are divided by 64 in the `main.cpp` to account for the bitwise shift performed.

$$\omega_{ie} = \text{enc}_i \cdot \frac{2\pi}{64 \cdot N_{\text{phases}} \cdot k_{\text{poles}} \cdot k_{\text{gear}}} \quad (3.19)$$

Where enc_i is the encoder velocity estimate. The denominator represents the total number of virtual sub-steps contained within a single complete revolution of the wheel (which covers 2π radians). The term "64" is applied in the `main.cpp` and was explained above. The rest are defined as:

- $N_{\text{phases}} = 4$, is the number of phases present in a quadrature signal.
- $k_{\text{poles}} = 8$, is the number of physical poles present in the encoders.
- $k_{\text{gear}} = 120$ is the gear ratio of the DC motors employed.

The equation normalizes the virtual displacement tracked by the RP2040 into standard SI kinematic units (rad/s). Naturally, Equation 3.2 is used to estimate the wheel's linear velocity v_{ie} .

This function also uses Equation 3.4 and Equation 3.3 to estimate the robot's velocities. In the original version of this library, this function also updates the robot pose estimation. However, this calculation is performed by the proposed localization algorithm, using the robot's velocities.

3.4.1.3 "Pico4Drive" Library

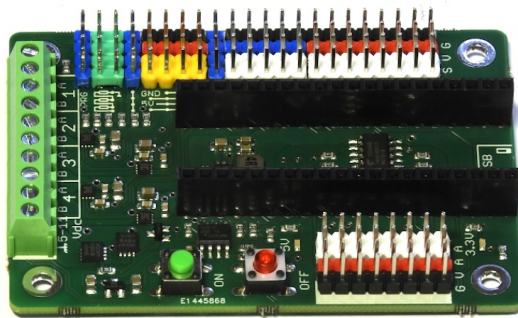


Figure 3.5: Pico4Drive board used in development. This board facilitates the control of the DC Motors and odometry estimation.

As mentioned above, in both robots the microcontroller is placed in a Pico4Drive development board. Therefore, in the architecture explained, whenever DC motors are controlled

the `pico4drive` library is used [23]. In the `setup` function, the `init` function of this library is called to define the GPIO pins and the SerialTiny communication in the board.

In every iteration of the `loop` function, the `update` function of the library is executed to update the "On" button state and to read the battery voltage. This updated voltage is then used to define the Pulse Width Modulation (PWM) for each motor's required voltage

3.5. Motion Control

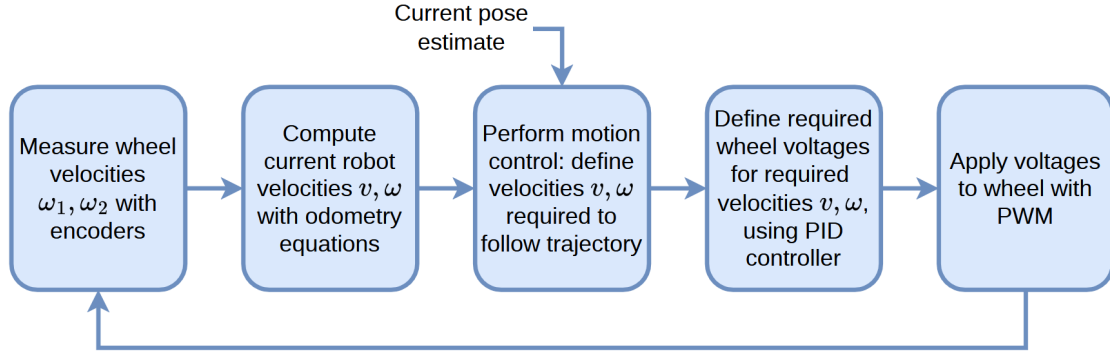


Figure 3.6: Schematic of the motion control cycle, performed in every control frame. The "picoEncoder" library is used to measure the wheels' velocities. The odometry equations are applied in "robot.cpp" to obtain the robot's estimated velocities v, ω . The "Pico4Drive" library is used to apply the wheel voltages.

This subsection will explore how the kinematic model was employed in controlling the robot. The complete motion control cycle is represented in Figure 3.6, summarizing previous sections. This section will particularly focus on the motion control, that uses the current pose estimate and the robot's velocities to compute the velocities required to follow a certain trajectory. Multiple motion modes were defined, with a common motion procedure.

3.5.1 Implementation of the kinematic model

3.5.1.1 Control State Machine

A Finite State Machine (FSM), `main_fsm_t`, is used to control the robot. The FSM implements two functions: `next_state_rules` defines the transitions between states, and `state_actions_rules` defines the robot's behavior in each state. This work focuses only on LiDAR characterization and the proposed localization algorithm so only the states used are introduced below. Nevertheless, this FSM could be used to control the robot during the RAF4 competition.

State	Name	Behavior
ts_set_theta	Orient robot	Rotates in place until a target heading is reached
ts_goto_xy	Go to point	Navigates to coordinates (x, y)
ts_follow_line	Follow line	Follows a defined line segment
ts_follow_circle	Follow arc	Follows a circular trajectory
ts_follow_track	Follow track	Chains multiple trajectory segments
ts_do_laps	Lap mode	Executes N laps around a rectangle $(\pm A, \pm B)$; transitions to state 200 when finished
ts_do_laps_complex	Complex laps mode	Executes N laps with complex geometry, employing all motion modes
201	Direct voltage	Applies $u_{1,req}$ and $u_{2,req}$ directly to the motors
102	Remote PID control	Enables PID mode with remotely sent angular velocities ω_1, ω_2

Table 3.1: States of `main_fsm_t` used in this thesis.

State 200 acts as a safe stop (zero voltage), used as the initial state and as a return state after any sequence. In all motion states (those starting with "ts"), `control_mode` is set to `cm_kinematics` and trajectory logic is delegated to the `trajectories` library, a library that contains each motion mode's function, that will be detailed below.

As this FSM is used in each frame, each motion state simply executes its mode function. This allows the functions to continuously follow the robot's motion and update the required velocities accordingly.

3.5.2 DC Motor Characterization

A characterization of the DC motors was performed to better control the robot's motion. The tests were performed with the robot on the ground, under the same conditions experienced during normal operation. This was done to account for the effects of the robot's weight, wheel-ground friction and other mechanical resistances that influence the motor

response. The characterization was carried out only on the robot used in those experiments. Two main characteristics were investigated: the motor dead zone and the dynamic response.

3.5.2.1 Dead zone

A DC motor does not necessarily start rotating when a voltage is applied to its terminals. Due to static friction, mechanical resistance and the characteristics of the motor itself, a minimum voltage is required to produce a non-zero angular velocity. The region where the applied voltage does not move the motor is the motor's dead zone. Characterizing this region avoids applying voltages that generate no movement.

To characterize the dead zone, several voltage values were applied to each motor. For each voltage, the angular velocity was measured after the response had settled and the steady-state value was used to characterize the relationship between applied voltage and motor speed.

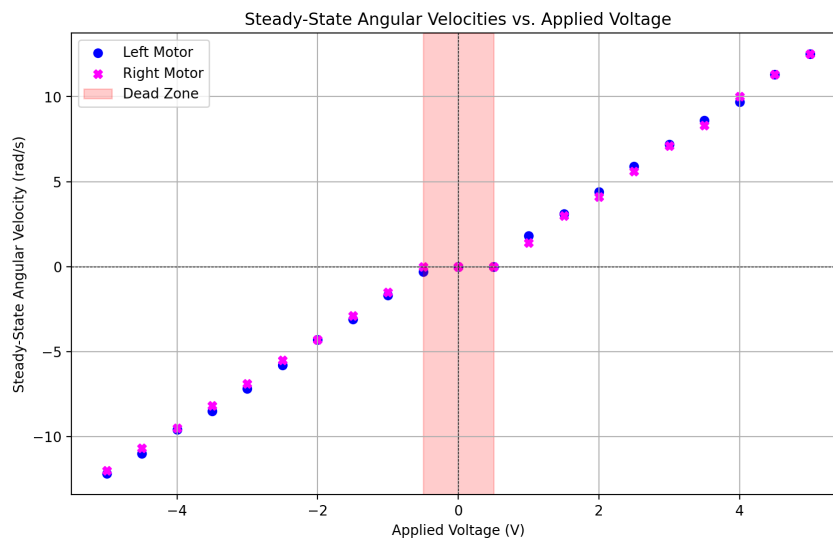


Figure 3.7: Dead zone analysis performed for both motors. Notably, a dead zone is evident in red.

The resulting steady-state angular velocity is shown in Figure 3.7. A clear dead zone can be observed around zero voltage, with the measured angular velocity remaining close to zero for voltages between -0.5 V and 0.5 V. Outside this region, the angular velocity increases linearly with the magnitude of the applied voltage. The response is approximately symmetric with the sign of the voltage, which indicates similar behavior in forward and reverse directions.

3.5.2.2 PID

Multiple motion modes were implemented, each defining a certain trajectory for the robot to follow. In every control frame, the robot's pose and velocities are estimated. Based on these estimates, the required linear and angular velocities of the robot, v_{req} and ω_{req} , are computed. This computation is done by the corresponding mode function in `trajectory.cpp` and will be detailed below.

In the `robot` library, the `calcMotorsVoltage` function starts by computing the angular velocity required by each wheel. This process entirely depends on the `control_mode`:

- When the control mode is `cm_kinematics`, the robot is performing a trajectory and the angular velocities are computed by inverting Equation 3.3 and Equation 3.4. In this case, the velocities computed are $\omega_{1,ref}$ and $\omega_{2,ref}$, as these values are the reference or goal of the PID controller.
- When the control mode is `cm_pid`, this function does not use the required robot velocities v_{req} , ω_{req} . Instead, it used values $\omega_{1,req}$ and $\omega_{2,req}$ received as commands as reference values for the PID. This mode is only used in state 102 for testing of the PID performance.
- When the control mode is `cm_voltage`, this function does not calculate any required angular velocity. Instead, it applies values $u_{1,req}$ and $u_{2,req}$, received as commands, directly as wheel voltages. This mode is used in motor testing states, such as 201.

A Proportional-Integral controller (PI) controller was implemented to control the angular velocities by adjusting the motor voltages. The controller parameters were obtained using a standard first-order model and Lambda tuning procedure. With the motors in open loop, their angular velocity response to a voltage step can be approximated by:

$$\omega(t) = A(1 - e^{-t/\tau}) \quad (3.20)$$

Where A is the steady-state angular velocity gain per unit of applied voltage.

Figure 3.8 shows each wheel's response to a voltage step. The voltage, as expected follows a step function, while the angular velocity seems to follow Equation 3.20. Notably, the right motor seems to have a slower response. This analysis was performed with

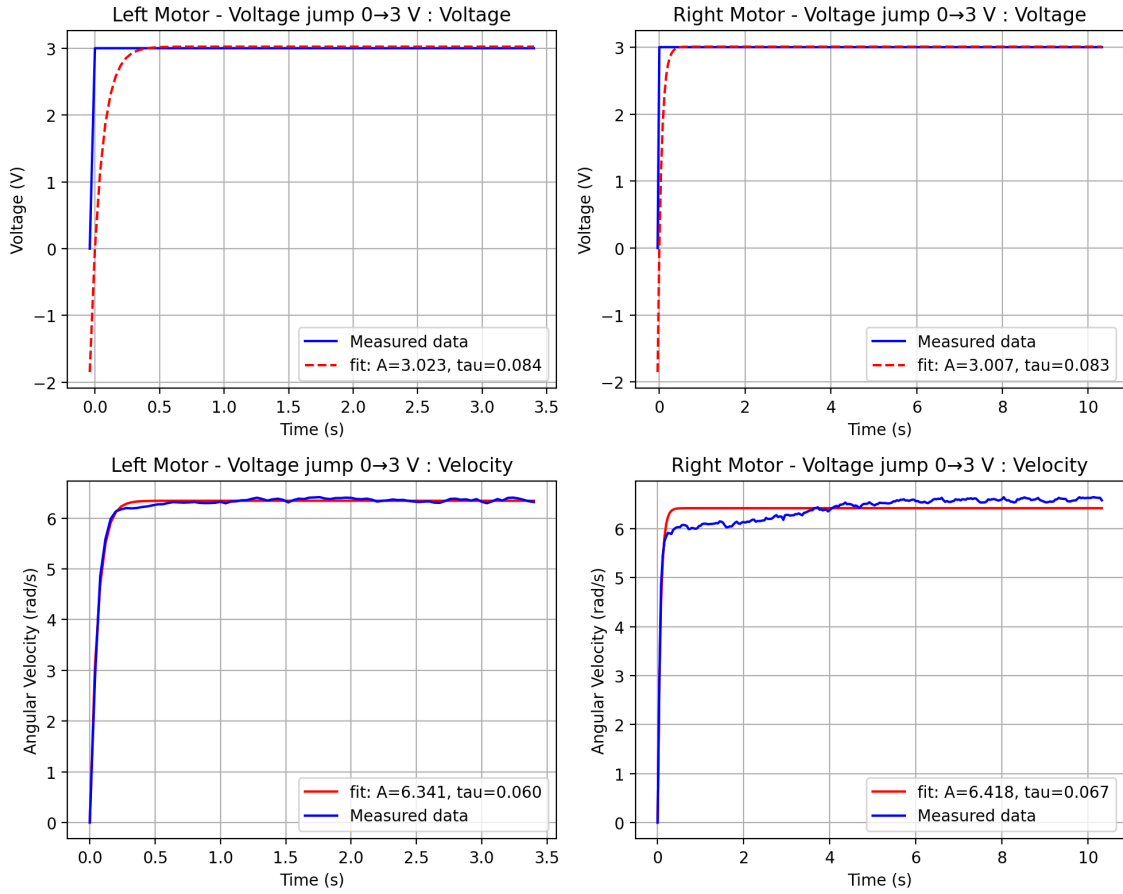


Figure 3.8: Results measured with the defined PID controller. Using the robot's state 102, the angular velocity of each wheel was defined. Then, the estimated angular velocity and voltage to the wheel's motor were recorded.

various voltages steps. The velocity gains and time constants of all exponential fits were averages and used to compute the PI controller gains:

$$K_p = \frac{\tau}{A\tau_c} \quad ; \quad K_i = \frac{K_p}{\tau} \quad (3.21)$$

The control time constant τ_c is the desired closed-loop time constant for the controller. That is, the controller defined by these constants must converge according to τ_c . In this work, a control time constant of 100 ms was defined. The same controller gains were applied for both wheels, assuming identical DC motors.

Robot	LiDAR	Kc	Ki	Kf	Kd	Kfd
Robot 1	HLS-LFCD2	0.254	2.337	0	0	0
Robot 2	Delta-2A / Mijia	0.290	2.500	0	0	0

Table 3.2: PID parameters tuned for each robot configuration depending on the LiDAR sensor used.

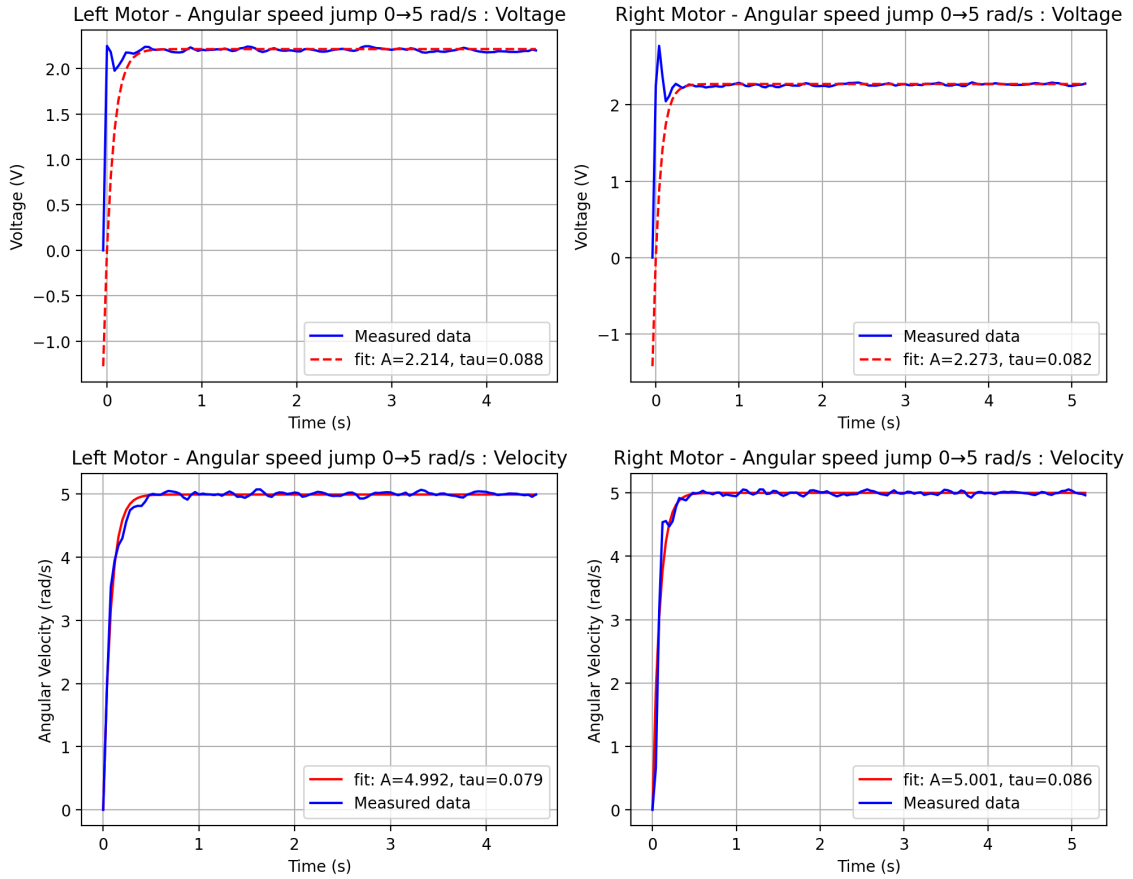


Figure 3.9: Results measured with the defined PID controller. Using the robot's state 102, the angular velocity of each wheel was defined. Then, the estimated angular velocity and voltage to the wheel's motor were recorded.

Figure 3.9 displays the performance of the wheel controller defined. The controller reaches the desired angular velocity in 100 ms, as desired. The low control time constant was chosen to achieve more responsive and smoother robot motion. However, as Figure 3.9 clearly shows, this rapid convergence results in voltage peaks, which would make this solution less suitable for industrial applications where long-term reliability and component stress would be of greater concern. Nevertheless, for the purposes of this work, the increased responsiveness was considered more important, as the PID controller is intended for a research-oriented, low-cost robotic platform rather than an industrial system.

3.5.3 Motion procedure

The motion modes "Follow line" and "Follow circle" share a common control procedure, executed every control cycle:

1. **Initial heading correction.** Before moving, if the robot's heading error relative to the planned trajectory is above a threshold `max_start_angle_error`, it rotates in place

until the error falls below this threshold. This prevents erratic trajectories caused by large initial angular offsets.

2. **Gain scaling.** The normal correction gain k_n is scaled inversely with the nominal velocity v_{nom} , so that the trajectory correction is consistent across operating speeds.
3. **Velocity computation.** The mode's geometry and the robot's current pose are used to compute the required linear and angular velocities v_{req} and ω_{req} . These are then used by the `robot` library, which controls the motors.
4. **Smooth braking.** As the robot approaches its target, a top-hat velocity filter reduces v_{req} and ω_{req} , ensuring smooth deceleration.
5. **Stopping.** At the end of the top-hat filter, when the remaining distance indicates that the target point was reached, the robot is ordered to stop. There, it awaits further instructions.

During motion, all modes except *Set Theta* continuously monitor the heading error. If it exceeds `max_ang_error`, the robot performs an emergency stop and rotates in place to correct its heading before continuing.

The values used were `max_ang_error` = 45° and `max_start_angle_error` = 15° . These thresholds were chosen so that the initial alignment is relatively strict, ensuring accurate orientation before motion begins. At the same time, the emergency stop threshold is permissive, allowing the robot to tolerate larger orientation errors without interrupting its motion.

The "Go to point" mode follows the same general structure but adapts it to its specific context and needs, as described in its respective section.

3.5.4 Trajectory control law

Each motion mode is controlled by a proportional error controller expressed in two orthogonal error components of the trajectory. The two scalar errors used are computed at each instant: the *heading error*, e_θ , is the angular difference between the robot's current heading and the heading tangent to the path, as well as the *normal error*, e_n , is the signed lateral deviation from the path.

The angular velocity is what can correct both these errors. Therefore, it is computed as a weighted sum of them:

$$\omega_{\text{req}} = k_{\theta} e_{\theta} + k_{n,\text{eff}} e_n \quad (3.22)$$

As expected, k_{θ} penalizes heading misalignment and $k_{n,\text{eff}}$ penalizes lateral drift. The required linear velocity v_{req} is set to the nominal velocity v_{nom} and subject to the top-hat braking filter described below. Circular arc following presents an exception, where an additional term v_{nom}/R is added to ω_{req} to account for the curvature of the path, of radius R . The definitions of e_{θ} and e_n depend on the motion mode and will be explored in detail below.

3.5.5 Top-hat velocity filter

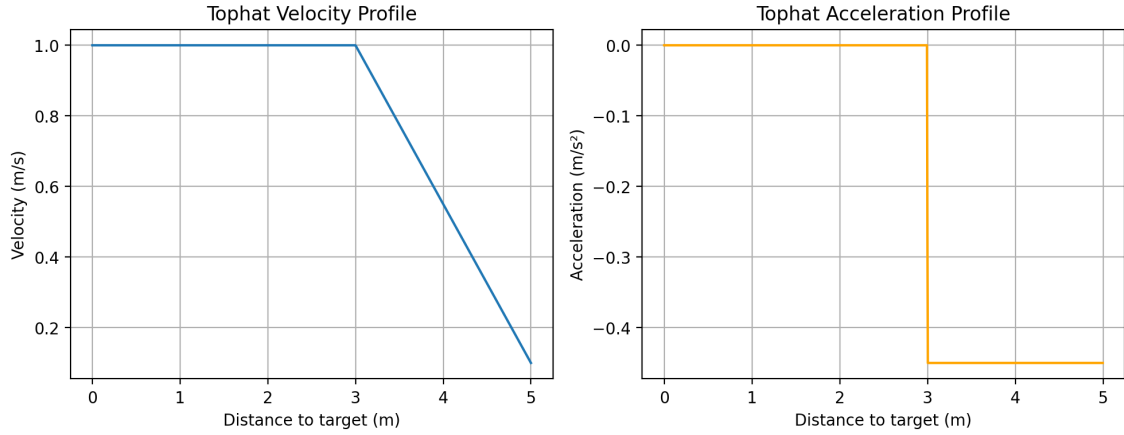


Figure 3.10: General profile of a tophat velocity filter. In this example, $D_{\text{ramp}} = 0.2$ m, $v_{\text{nom}} = 1$ m/s and $v_{\text{min}} = 0.1$ m/s.

A top-hat velocity filter is applied when a certain trajectory segment approaches its end-point to produce controlled, consistent deceleration. Instead of changing the robot's required velocity directly from its nominal value to zero, the filter gradually decreases the required velocity.

For linear motion, the top-hat filter depends on the remaining distance to the target, d . This filter can be defined as follows:

$$v(d) = v_{\text{min}} + \frac{\max(\min(d, D_{\text{ramp}}), 0)}{D_{\text{ramp}}} \cdot (v_{\text{nom}} - v_{\text{min}}) \quad (3.23)$$

$$= v_{\text{min}} + t \cdot (v_{\text{nom}} - v_{\text{min}}) \quad , \quad t \in [0, 1] \quad (3.24)$$

Where D_{ramp} defines the braking distance, while v_{nom} and v_{min} are the nominal and minimal velocities. From these equations, it is clear that the required velocity remains equal to the nominal value until reaching the braking region, where it decreases linearly until the minimum velocity.

3.5.5.1 Implementation

In practice, d was not used in the implementation of the filter. If the robot has accumulated lateral error, the direct distance to target (d) may never fall below a certain threshold, even if the robot passes by the target. For this reason, this metric was replaced by d_{path} , the remaining distance along the path. Its calculation depends on the trajectory taken, but it will always reach zero when the robot passes by the target. Furthermore, this variable will always decrease as the robot approaches the target, even if oscillations occur. By using it in the top-hat filter, the robot will always decelerate as it reaches P_f .

During linear motion, the robot can only reach a finite deceleration, causing it to drift after the required velocity reaches zero, potentially overshooting the target. To compensate for this effect, a stopping margin d_{stop} was defined. To account for this phenomenon, both the braking distance and the remaining distance must be adjusted. The top-hat then uses the effective values of these metrics:

$$d_{\text{eff}} = d_{\text{path}} - d_{\text{stop}} \quad (3.25)$$

$$D_{\text{ramp,eff}} = D_{\text{ramp}} - d_{\text{stop}} \quad (3.26)$$

This simple transformation shifts the stopping point backwards, compensating for the robot's drifting and overshooting. In other words, the robot will effectively start braking when $d_{\text{path}} = D_{\text{ramp}}$ and the required velocity will be null when $d_{\text{path}} = d_{\text{stop}}$, so the robot should be stopped on the target, when $d_{\text{path}} = 0$. The effective remaining distance d_{eff} is then used to decide if the robot has reached its target and must stop, which happens when $d_{\text{eff}} < 0$.

The top-hat filter is evaluated by computing d_{eff} and $D_{\text{ramp,eff}}$ from the current d_{path} and d_{stop} , then directly applying:

$$v_{\text{req}} = v_{\text{min}} + \frac{\max(0, \min(d_{\text{eff}}, D_{\text{ramp,eff}}))}{D_{\text{ramp,eff}}} \cdot (v_{\text{nom}} - v_{\text{min}}) \quad (3.27)$$

The braking distance and stopping margin naturally depend on the nominal velocity, as will be explored in subsubsection 3.5.5.2. The minimum linear velocity was kept constant:

$$v_{\min} = 0.05 \text{ m/s}$$

3.5.5.2 Velocity robustness

A design goal of the controller implemented was to achieve consistent trajectory-following performance across all implemented motion modes and over a wide range of operating velocities, while keeping the implementation and its tuning simple. Therefore, a single set of parameters is used for all cases, with a small subset being scaled with the nominal linear velocity. This ensures that the same control strategy is functional without having to retune. The system was evaluated with linear velocities in the range 0.10 – 0.35 m/s.

To ensure that trajectory following behavior remains consistent across different nominal velocities, the lateral correction gain is scaled with the nominal linear velocity of the trajectory, v_{nom} . Without this scaling, a gain tuned for a given velocity could produce overly aggressive corrections at higher speeds and insufficient corrections at lower speeds. An inversely proportional relationship was chosen:

$$k_{n,\text{eff}} = \frac{k_n}{s}, \quad s = \frac{v_{\text{nom}}/v_{\text{ref}}}{2} + 0.5 \quad (3.28)$$

where $v_{\text{ref}} = 0.15 \text{ m/s}$ is the reference velocity at which the control gains were tuned, while s is the speed scale factor. Note that the relationship is not strictly inversely proportional, the form used softens the scaling at low speeds and prevents excessive gain amplification.

No scaling is applied to the angular gain, k_{θ} , as the initial angular correction step of the trajectory makes it unnecessary. Furthermore, the lateral correction term is much more likely to produce deviations and oscillations, so properly controlling it is essential.

The braking parameters of the top-hat linear velocity filter were also adjusted with the nominal velocity, ensuring that the braking zone grows with speed. This keeps the temporal profile of deceleration approximately constant regardless of the operating velocity. The following relationships were defined empirically:

$$D_{\text{ramp}} = |v_{\text{nom}}| \quad (\text{m}) \quad (3.29)$$

$$d_{\text{stop}} = 0.01 + 0.04 \cdot \frac{|v_{\text{nom}}|}{v_{\text{ref}}} \quad (\text{m}) \quad (3.30)$$

3.5.6 Control parameters

Parameter	Description	Notes	Value
k_n	Lateral correction gain	Scaled with v_{nom}	2.50
k_θ	Angular correction gain	Constant	0.75
v_{nom}	Nominal linear velocity	Motion-dependent input	Variable
v_{min}	Minimum linear velocity	Constant	0.05 m/s
D_{ramp}	Braking start distance	Scaled with v_{nom}	$ v_{\text{nom}} $
d_{stop}	Stopping margin	Scaled with v_{nom}	Equation 3.30

Table 3.3: Summary of control parameters used across all motion modes.

To summarize, all implemented motion modes share a common set of base control parameters, listed in Table 3.3. This unified parameterization guarantees consistent behavior across all trajectory types and velocities, without requiring retuning.

Among these parameters, three are scaled with the nominal linear velocity v_{nom} : the lateral correction gain k_n , the braking distance D_{ramp} , and the stopping margin d_{stop} . Together, D_{ramp} and d_{stop} determine the effective braking region. The robot begins decelerating at $d_{\text{path}} = D_{\text{ramp}}$ and receives a null velocity command at $d_{\text{path}} = d_{\text{stop}}$, with the residual momentum carrying it to the target. All remaining parameters are kept constant.

This results in a compact and simple control structure in which velocity-dependent scaling is applied only to k_n , D_{ramp} , and d_{stop} , while all other parameters remain fixed regardless of operating conditions.

3.5.7 "Set theta" Mode

This motion mode is used to change the robot's heading while stationary. As mentioned, this is the only motion mode that does not follow section 3.5, as there is no linear motion happening. This mode compares the robot's heading estimate θ_e to the target heading θ_f and the signed heading error is computed:

$$e_\theta = \theta_f - \theta_e \quad (3.31)$$

An error proportional controller is used to compute the required velocities:

$$\begin{cases} v_{\text{req}} = 0 \\ \omega_{\text{req}} = k_{\theta} \cdot e_{\theta} \end{cases}$$

To avoid overshooting, a top-hat filter analogous to Equation 3.5.5.1 is applied to the angular velocity:

$$\omega_{\text{req}}(\theta) = \omega_{\text{min}} + \frac{\max(\min(\theta, \Theta_{\text{ramp}}), 0)}{\Theta_{\text{ramp}}} \cdot (k_{\theta} \cdot e_{\theta} - \omega_{\text{min}}) \quad (3.32)$$

Here, θ represents the remaining angle to the target heading, Θ_{ramp} defines the angular deceleration range and ω_{min} is the minimum angular velocity. These parameters were kept constant:

$$\Theta_{\text{ramp}} = 60^{\circ} \quad ; \quad \omega_{\text{min}} = 5^{\circ}/\text{s} \quad (3.33)$$

These required velocities are computed in each control frame, until the angular error is below 3° .

This trajectory mode is intentionally simple, as it is used in the initial alignment of every other motion mode. Its primary objective is to provide reliable and predictable aligning behavior, while being easy to understand and debug. The `set_theta` function is implemented with a boolean output: when the angular error is below 3° it returns `true`, otherwise it returns `false`. This capability is used by the other motion modes, that rely on the `set_theta` function to check and perform alignment.

3.5.8 "Go to point" Mode

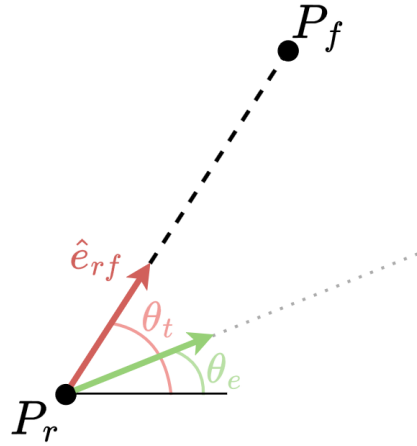


Figure 3.11: Diagram showing the variables used by calculations in "Go to point" mode and how the angular error is visualized.

This mode was designed to make small and precise position adjustments, where reaching the final point is more important than following a certain trajectory. The robot's estimated position P_r is defined as the start, from where the robot aims to reach a target point P_f . Contrary to other modes, the target heading θ_t is recomputed in each control frame, as the robot's position changes:

$$e_\theta = \theta_t - \theta_e \quad ; \quad \theta_t = \text{atan2}(P_{f,y} - y_e, P_{f,x} - x_e) \quad (3.34)$$

Since heading is the sole control variable in this mode (there is no lateral error correction), precise alignment with θ_t is essential throughout the maneuver. As will be explored below, the other motion modes have constant values of θ_t , so large angular error during motion indicates that the robot has deviated from the intended trajectory. In this mode, elevated heading errors may simply reflect the natural shift in θ_t as the robot moves.

For this reason, this mode contains alterations to the motion control procedure in subsection 3.5.3. Like in other modes, `set_theta` mode is used for initial alignment and for mid-motion correction. However, it keeps the "Set theta" mode's standard stopping threshold of 3 error for initial alignment, instead of the `max_start_angle_error` threshold used in the other modes. This results in tighter initial alignment, which is essential as this mode only uses heading for control.

This mode implements the top-hat filter using the direct distance from the robot to the target point, as there is no trajectory to measure the remaining distance on. As explained above, when this distance falls below the stop margin d_{stop} , the robot should be moving with minimum velocity and is ordered to stop completely.

3.5.9 "Follow Line" Mode

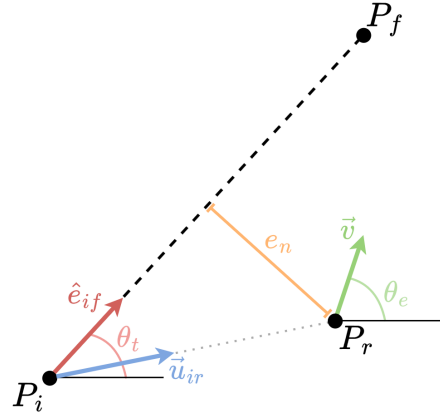


Figure 3.12: Diagram showing the variables used by calculations in "Follow Line" mode and how each error is visualized.

In this mode, the robot's motion aims to follow a line between an initial and final points, P_i and P_f , respectively. In this mode, the robot is not guaranteed to start at P_i , so the lateral error is considered. This error consists of the perpendicular distance between the robot's pose P_r and the trajectory line. Following the control law in subsection 3.5.4, the lateral error term will make the angular velocity turn the robot towards the line, while the angular term will make the robot move parallel to the line. An ideal tuning of the gains would enable optimal convergence from the robot's starting position to the line. Inefficient tuning can result in passing the target point with elevated lateral error or oscillating around the line. The gains used in testing (listed in Table 3.3) were reached empirically and allow convergence with minimal oscillation.

As referenced above, this mode follows the motion procedure in subsection 3.5.3. This mode's geometry uses only the coordinates of P_i and P_f , so the target heading θ_t is constant and given by:

$$e_\theta = \theta_t - \theta_e \quad ; \quad \theta_t = \arctan \left(\frac{P_{f,y} - P_{i,y}}{P_{f,x} - P_{i,x}} \right) \quad (3.35)$$

The unit vector indicating the trajectory's direction can be described by:

$$\hat{e}_{if} = \frac{(P_{f,x} - P_{i,x} ; P_{f,y} - P_{i,y})}{\sqrt{(P_{f,x} - P_{i,x})^2 + (P_{f,y} - P_{i,y})^2}} \quad (3.36)$$

The vector indicating the robot's position relative to P_i can be easily computed:

$$\vec{u}_{ir} = (P_{r,x} - P_{i,x} ; P_{r,y} - P_{i,y}) \quad (3.37)$$

Considering these two vectors, the lateral error can be easily computed as a cross product magnitude:

$$e_n = \|\hat{e}_{if} \times \vec{u}_{ir}\| = e_{if,y}u_{ir,x} - e_{if,x}u_{ir,y} \quad (3.38)$$

After computing the lateral and angular error, the required angular velocity can be estimated with Equation 3.22.

By using the dot product, the distance traveled along the path can be computed:

$$d_{\text{travel}} = |\hat{e}_{if} \cdot \vec{u}_{ir}| = e_{if,x}u_{ir,x} + e_{if,y}u_{ir,y} \quad (3.39)$$

Therefore, the remaining distance along the path, essential for the top-hat filter, can be calculated:

$$d_{\text{path}} = \sqrt{(P_{f,x} - P_{i,x})^2 + (P_{f,y} - P_{i,y})^2} - d_{\text{travel}} \quad (3.40)$$

The top-hat filter is implemented with this distance. As mentioned in subsection 3.5.5, when this distance is below the stop margin, the robot is ordered to stop.

Although this motion mode includes both angular and lateral error correction terms and is capable of recovering from an initial pose error, an initial heading alignment step is still performed, which may seem redundant or appear to reduce the purpose of the controller. However, the correction terms in the motion procedure are most effective at compensating for small deviations accumulated over time, during motion. If the robot starts with a large initial heading error, it may experience slow or oscillatory convergence before reaching the planned trajectory. Therefore, the initial alignment step ensures that the controller starts from a well-conditioned and predictable state, resulting in smoother and more reliable trajectory tracking.

3.5.10 "Follow Circle" Mode

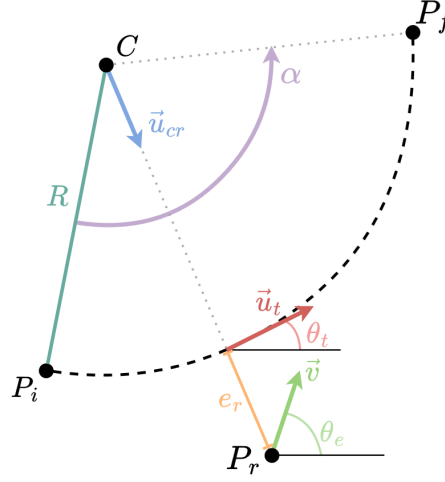


Figure 3.13: Diagram showing the variables used by calculations in "Follow Circle" mode and how each error is visualized.

In this mode, the robot is commanded to follow an arc. The arc starts at an initial point P_i , has center C and describes an angle α . In this mode, the robot is not guaranteed to start at P_i , so the lateral error is considered. This error consists of the perpendicular distance between the robot's pose P_r and the trajectory line, which in a circular trajectory is simply the radial error. Following the control law in subsection 3.5.4, the lateral error term will make the angular velocity turn the robot towards the arc, while the angular term will make the robot move parallel to the arc. The gains used in testing (listed in Table 3.3) were achieved empirically and resulted in convergence with minimal oscillation.

This mode follows the motion procedure in subsection 3.5.3. The mode's geometry is defined using the coordinates of P_i and C , and the arc angle α . In this kind of trajectory, the lateral error is simply the signed difference between the trajectory radius and the robot's distance to C , if this distance is low then the robot is following a circular trajectory with the correct radius.

$$e_n = \sqrt{(P_{r,x} - C_x)^2 + (P_{r,y} - C_y)^2} - R \quad (3.41)$$

$$R = \sqrt{(P_{i,x} - C_x)^2 + (P_{i,y} - C_y)^2} \quad (3.42)$$

The vector indicating the robot's position relative to the trajectory center can be defined as:

$$\vec{u}_{cr} = \begin{pmatrix} P_{r,x} - C_x \\ P_{r,y} - C_y \end{pmatrix} \quad (3.43)$$

Defining $\vec{u}_t$ as a 90 rotation of $\vec{u}_{cr}$, it is a vector tangential to the trajectory along the $C \rightarrow P_r$ line, as represented in Figure 3.13. The direction of the rotation has to be the same as the arc, so it must match the sign of α :

$$\vec{u}_t = R_\beta \vec{u}_{cr} = \begin{pmatrix} u_{cr,x} - \text{sign}(\alpha)u_{cr,y} \\ \text{sign}(\alpha)u_{cr,x} + u_{cr,y} \end{pmatrix} \quad (3.44)$$

$$\beta = \text{sign}(\alpha) \frac{\pi}{2} \quad (3.45)$$

Finally, the angular error can be easily computed with the components of $\vec{u}_t$:

$$e_\theta = \theta_t - \theta_e \quad ; \quad \theta_t = \arctan2(u_{t,y}; u_{t,x}) \quad (3.46)$$

To implement the error proportional controller, Equation 3.22 is used, with an alteration. In circular motion, even if the lateral and angular errors are null, the angular velocity of the robot has to make it follow a trajectory with radius R . The circular motion and lateral correction terms must match signs and follow α , so that the correct direction is always used. Therefore, the required angular velocity for this mode is given by:

$$\omega_{req} = \text{sign}(\alpha) \cdot \left(\frac{v_{nom}}{R} + k_{n,eff} e_n \right) + k_\theta e_\theta \quad (3.47)$$

Like the "Follow line" mode, the linear velocity follows a top-hat filter that uses the remaining trajectory along the path. However, this means that the remaining arc length must be computed. For this, it is necessary to compare the angle from the center to the start point with the angle from the center to the robot's current position:

$$\theta_{start} = \arctan2(P_{i,x} - C_x; P_{i,y} - C_y) \quad ; \quad \theta_r = \arctan2(P_{r,x} - C_x; P_{r,y} - C_y) \quad (3.48)$$

The planned arc trajectory describes an angle α , starting from P_i . The done and remaining angles can be estimated:

$$\alpha_{done} = \theta_r - \theta_{start} \quad ; \quad \alpha_{remain} = \alpha - \alpha_{done}$$

Finally, the remaining arc length can be easily computed:

$$d_{\text{path}} = \alpha_{\text{remain}} \cdot R$$

The top-hat filter is then implemented using the distance d_{path} . As mentioned in subsection 3.5.5, when this distance is below the stop margin, the robot is ordered to stop, and it should be stopped on the target. Again, the initial heading alignment step is performed, because the correction terms are most effective at compensating for small deviations accumulated during motion. The alignment step ensures that the controller starts from a well-conditioned state.

3.5.11 "Do Laps" Mode

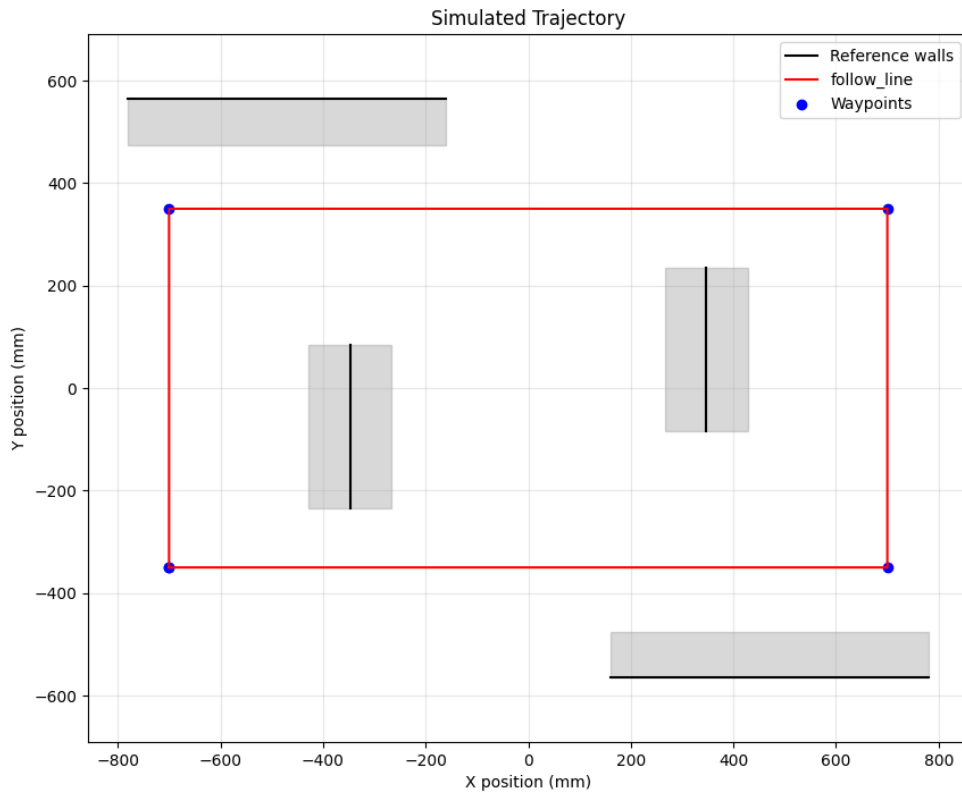


Figure 3.14: Representation of the simple trajectory planned.

To properly test and evaluate the proposed localization algorithm, two testing trajectories were developed and implemented as motion modes in the control FSM. They will be explored in the following sections.

The "Do Laps" mode commands the robot to perform 10 rectangular laps around the track. The four corners of the rectangle coincide with four of the ArUco markers on track, making it easy to visually check if the robot is performing a decent trajectory during testing. Each lap consists of four straight segments, implemented using the "follow line" mode:

Seg.	Mode	$P_{i,x}$	$P_{i,y}$	$P_{f,x}$	$P_{f,y}$	C_x	C_y	α (rad)
0	2	-0.700	-0.350	-0.700	0.350	0	0	0
1	2	-0.700	0.350	0.700	0.350	0	0	0
2	2	0.700	0.350	0.700	-0.350	0	0	0
3	2	0.700	-0.350	-0.700	-0.350	0	0	0

Table 3.4: Definition of the segments of the simple rectangular trajectory. Motion modes are labeled as: 1 - Go to Point, 2 - Follow Line, 3 - Follow Circle.

Transitions between segments are done with two variables of the trajectories library: the `finished` flag and the `vertex_counter`. The `finished` flag is set by `follow_line` when the effective remaining distance, d_{eff} , falls below zero (indicating the robot has reached the target point). However, as a precaution, the transition to the next segment is only triggered when `finished` is set and the robot's estimated linear velocity has fallen below 0.02 m/s. This condition prevents erratic behavior from commanding a new trajectory while still in motion.

When these conditions are met and the segment is considered completed, `vertex_counter` is incremented, moving to the next segment. Both `finished` and `initial_align_done` are reset, so the robot performs the initial heading alignment step at the start of every segment, as described in subsection 3.5.3. When the `vertex_counter` wraps back to zero, a completed lap is recorded using `lap_counter`, and the cycle is restarted.

3.5.12 "Complex Laps" Mode

This mode implements a more demanding trajectory, designed to test three motion modes in a single lap. The simple rectangular trajectory, which only uses "follow line", while this circuit combines "go to point", "follow line" and "follow circle" segments, making it a more extensive test of the motion control. The 11 segments of the trajectory are defined in Table 3.5:

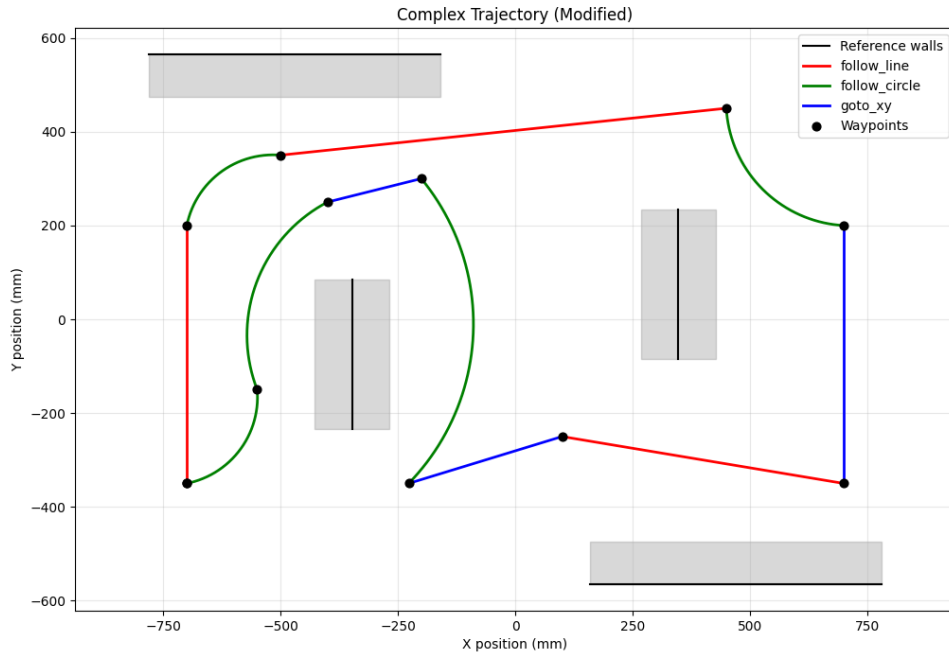


Figure 3.15: Representation of the simple trajectory planned.

Seg.	Mode	$P_{i,x}$	$P_{i,y}$	$P_{f,x}$	$P_{f,y}$	C_x	C_y	α (rad)
0	2	-0.700	-0.350	-0.700	0.200	0	0	0
1	3	-0.700	0.200	-0.500	0.350	-0.516	0.163	-1.4577
2	2	-0.500	0.350	0.450	0.450	0	0	0
3	3	0.450	0.450	0.700	0.200	0.715	0.465	1.4595
4	1	0.700	0.200	0.700	-0.350	0	0	0
5	2	0.700	-0.350	0.100	-0.250	0	0	0
6	1	0.100	-0.250	-0.227	-0.350	0	0	0
7	3	-0.227	-0.350	-0.200	0.300	-0.577	-0.010	1.4595
8	1	-0.200	0.300	-0.400	0.250	0	0	0
9	3	-0.400	0.250	-0.550	-0.150	-0.251	-0.034	1.4577
10	3	-0.550	-0.150	-0.700	-0.350	-0.737	-0.166	-1.4577

Table 3.5: Definition of the segments of the complex trajectory. Motion modes are labeled as: 1 - Go to Point, 2 - Follow Line, 3 - Follow Circle.

The transition logic is identical to that of the "Do Laps" mode: the `finished` flag and the null velocity condition control transitions between segments, with `initial_align_done` being reset at each transition to ensure the robot performs the initial heading alignment

in every segment. The `vertex_counter` wraps back to zero after the last segment, a lap is counted and the circuit is restarted.

The trajectory was designed to cover a larger area of the arena, ensuring that the localization algorithm is evaluated in different regions and headings. The "follow circle" segments introduce circular motion that combines the radial and angular correction with the top-hat braking along the arc. Meanwhile, the "go to point" segments test the precision of the point-to-point positioning. By combining straight, curved and point-to-point segments, this trajectory aims to provide an extensive evaluation of the interaction between the motion controller and the localization algorithm, testing tracking performance, localization robustness and the error accumulation over long distances.

3.6. Platform cost estimation

Table 3.6: Estimated cost of the robotic platforms.

Component	Robot 1	Robot 2	Total
Raspberry Pi Pico W [24]	6.42 €	6.42 €	12.84 €
Pico4Drive board [25]	19.90 €	19.90 €	39.80 €
DC motors with encoders [26]	23.00 €	23.00 €	46.00 €
HLS-LFCD2 [27]	30.00 €	–	30.00 €
Delta-2A [28]	–	25.99 €	25.99 €
18650 batteries [29]	9.20 €	9.20 €	18.40 €
Battery power supply [30]	1.90 €	1.90 €	3.80 €
Total	70.52 €	66.51 €	137.03 €

The cost of the robotic platforms used is summarized in Table 3.6. The estimated hardware cost of Robot 1 (with a HLS-LFCD2 LiDAR) is 70.52 € and 66.51 € for Robot 2 (with a Delta-2A LiDAR), resulting in a total cost of 137.03 € for the platforms developed in this work. The robots' chassis were manufactured using 3D printing and were not included in this estimation, as no direct cost could be reliably estimated.

Overall, the estimated hardware costs are low, constituting a cost-effective platform for mobile robotics. The remainder of this work has the goal of investigating whether a platform built from readily available and inexpensive components can provide the sensing and computational capabilities required for robot localization and navigation. The performance of the proposed platforms is evaluated experimentally in the following chapters.

4. LiDAR Characterization

Before defining any localization algorithm, the work was started by studying and understanding the sensor used: LiDAR. This section has the objective of explaining LiDAR sensors and the two particular sensors used.

4.1. LiDAR Ranging Sensors

LiDAR stands for "Light Detection and Ranging". As the name suggests, these are active-range sensors that operate by emitting modulated laser pulses and detecting the light reflected from objects in the surrounding environment. The fundamental measurement quantity is the *time of flight*, Δt , the time passed between pulse emission and the detection of its reflection. Using it, the distance from the sensor to the reflecting surface is computed:

$$d = \frac{1}{2} c \Delta t \quad (4.1)$$

where $c \approx 3 \times 10^8$ m/s is the speed of light in air. The factor of $\frac{1}{2}$ accounts for the fact that the time of flight encompasses the round-trip propagation of the pulse. Given the high propagation speed of light, LiDAR sensors are capable of high-accuracy distance measurements at elevated measurement rates.

A key advantage of LiDAR over passive optical sensors, like cameras, is its independence from ambient illumination conditions. Because the sensor emits light at a precisely defined wavelength, the photo-detector can be filtered to reject ambient light and isolate the reflected light. This enables reliable operation across a wide range of lighting conditions.

Many applications employ *spinning* LiDAR sensors, in which the laser-detector pair is continuously rotated by a DC motor while remaining centered on a vertical axis. At sufficiently high rotational speeds, given the high measurement rates of typical LiDAR sensors, it is possible to obtain dense, accurate planar scans of the environment in real time. Some variants stack several emitter-detector pairs vertically, with different angles, enabling three-dimensional point cloud acquisition.

Despite their advantages, LiDAR sensors are subject to some physical limitations. Propagation through certain atmospheric conditions, such as rain, fog or snow, causes dispersion and attenuation of the laser pulses, which degrades the accuracy and maximum

operating distance of the sensor. Similarly, surfaces with low reflectivity or strong absorption at the sensor's operating wavelength will produce weak return signals, which increases measurement uncertainty. In spinning LiDAR units, rapid platform motion during a scan introduces *motion distortion*: since each range measurement is acquired at a different sensor pose, the resulting point cloud is geometrically inconsistent unless a motion compensation scheme is applied.

This work aims at proposing an accessible, efficient localization algorithm. To better evaluate available options, two different LiDAR models were tested: HLS-LFCD2 and Delta-2A. The selected models are low-cost 2D spinning LiDARs, which are commonly employed in indoor mobile robotics due to their favorable cost-to-performance relationship. Evaluating sensors from this category is particularly relevant for cost-sensitive robotic platforms. In the following sections, the two models' specifications and communication protocols will be explored.

4.1.1 HLS-LFCD2

The first sensor studied is the HLS-LFCD2, also known as LDS 1.5, manufactured by HL Data Storage. This sensor is a 2D spinning LiDAR based on time of flight. Because of its compact design and reduced cost, it has been adopted in academic robotic platforms, such as the TurtleBot3 LDS-01 implementation.

The HLS-LFCD2 communicates through a UART interface connected to the Raspberry Pi Pico (in the Serial2 peripheral), with a baud rate of 230400 Hz. The sensor outputs a continuous stream of data packets, each corresponding to a 6° range, with a constant packet length of 42 bytes. This simplifies the data processing, since the frame boundaries are predetermined. Furthermore, the LiDAR's rotation speed cannot be controlled, as it is defined by the sensor to ensure a constant scanning rate and angular resolution.

Each data packet follows the same structure. It starts with a header containing the sync byte, the angle index and the rotation speed, followed by 6 data blocks and ends with 2 checksum bytes. This structure is detailed in Table 4.1 [31].

Byte(s)	Length	Field	Description
0	1	Sync	Frame synchronization byte (0xFA).
1	1	Angle index	Index of the first measurement in the packet.
2-3	2	Rotational speed	LiDAR rotational speed (in RPM).
4-39	6×6	Measurements	Six measurement blocks are transmitted per packet. Each block corresponds to a range measurement and contains: Intensity (2 bytes), Distance (2 bytes), Reserved (2 bytes).
40-41	2	Checksum	Packet checksum used for error detection.

Table 4.1: Summary of the LiDAR data packet structure.

Notes This protocol follows a little-endian order. Therefore, in every quantity that takes two bytes, the least significant is sent first.

The angle index of the packet is not the starting angle. Instead, that is given by:

$$\theta_0 = (\text{Angle index} - 0xA0) \times 6 \quad (4.2)$$

where θ_0 is presented in degrees. Each data block corresponds to a measurement angle, in increments of 1 degree. The 6 data blocks have indices from 0 to 5, making the packet cover 6 degrees. The value 0xA0 is subtracted from the angle index to normalize it, so each lap starts with the angle 0°.

The intensity parameter indicates the intensity of the reflected light detected and can be used to define the uncertainty of the measurements.

Checksum Bytes 40 and 41 both carry the expected checksum values. A packet is considered valid if the value obtained from the checksum equation matches either of the two received checksum bytes.

Therefore, the checksum for this protocol is performed by summing all data bytes from byte 0 to byte 39. Since the sum may exceed one byte, only the least significant byte is retained by taking the result of $\text{mod } 256$. This byte is subtracted from 0xFF and the resulting value compared to bytes 40 and 41.

$$\text{checksum} = 0xFF - \left(\sum_{k=0}^{39} \text{packet}[k] \right) \bmod 256 \quad (4.3)$$

Specifications According to the model's datasheet [31], it has a measurement distance range of [0.12, 3.5] m. The scan angular resolution is 1° , something that is enforced in the packet structure. This sensor has a sampling rate of 1.8 kHz. With 2520 bytes per lap, this yields a scanning rate of 5 Hz.

When it comes to accuracy and precision, two distinct behavior regimes are expected:

- Distances up to 50 cm: accuracy of 15 mm, precision of 10 mm
- Distances above 50 cm: accuracy of 5%, precision of 3.5%

In the datasheet, precision is defined as the average of the maximum and minimum measured distances. Assuming a confidence level of 95% and normally distributed data, this definition of precision corresponds to 2σ , with σ being the standard deviation of the data.

4.1.2 Delta-2A

The second sensor explored is the Delta-2A, manufactured for robotic vacuum cleaners. Similarly to the HLS-LFCD2, it is a 2D spinning time of flight LiDAR intended for indoor navigation and mapping. This sensor is manufactured by 3irobotix / Shenzhen Shanchuan Robot Co. and its datasheet [32] was only found as a PDF file written in the original language, Chinese.

The Delta-2A LiDAR also uses a UART interface via the Raspberry Pi Pico Serial2, with a baud rate of 115200 Hz. In contrast to the other model, this sensor uses a variable-length structure, where each packet begins with a header containing various information. This header defines the number of bytes in the payload, which corresponds to an angular sector of approximately 22.5° . As a result, packet parsing requires dynamic length decoding, increasing the complexity of the reading process compared to fixed-length protocols. Further increasing this model's variability, its rotational speed can be easily controlled by the microcontroller, by adjusting the DC motor driver voltage or PWM.

Byte(s)	Length	Field	Description
0	1	Sync	Frame synchronization byte (0xAA).
1–2	2	Length	Number of bytes from sync up to (but not including) the checksum.
3	1	Protocol	Protocol version identifier (0x01).
4	1	Type	Packet type identifier.
5	1	Command	Command byte.
6-7	2	Payload length	Number of bytes in the payload (N_{bytes}), little-endian.
8	1	Rotational speed	LiDAR rotational speed: multiply by 0.05 to obtain RPS, or by 3 to obtain RPM.
9-10	2	Offset angle	Angular offset of the frame, in units of 0.01° .
11-12	2	Start angle	Angle of the first measurement in the packet, in units of 0.01° .
13-(13+3N-1)	$3 \times N$	Measurements	N measurement blocks are transmitted per packet, where $N = (\text{Payload length} - 5)/3$. Each block corresponds to a range measurement and contains: Signal quality (1 byte), Distance MSB (1 byte), Distance LSB (1 byte).
last 2	2	Checksum	Packet checksum used for error detection.

Table 4.2: Summary of the Delta-2A LiDAR data packet structure. The payload starts after the payload length is indicated, from byte 7 up to (but not including) the checksum bytes.

Notes This protocol follows a big-endian order, that is, in every quantity that takes two bytes, the most significant is sent first.

The command byte indicates whether the packet has useful data. If this byte has the value 0xAD, the data is useful, otherwise the byte has the value 0xAE.

Measurement angle and distance As two bytes are used, the packet contains the start angle itself, in a reduced scale. Considering that a packet covers 22.5° , the measurement with index $i \in \{0, \dots, N - 1\}$ will correspond to the angle:

$$\theta_i = \theta_0 + 22.5 \times \frac{i}{N} \quad (4.4)$$

The distance of a certain block, in millimeters, is obtained by combining the most and least significant bytes and dividing the result by 4.

Checksum The checksum is computed by summing all bytes from the sync byte to (but not including) the checksum bytes, retaining only the least significant 16 bits of the accumulated sum:

$$\text{checksum} = \left(\sum_{k=0}^{L-1} \text{packet}[k] \right) \bmod 65536 \quad (4.5)$$

where L is the number of bytes preceding the checksum field. A packet is considered valid if the computed value matches the received checksum exactly.

Specifications According to the model's datasheet, it has a measurement distance range of $[0.15, 8.0]$ m. This sensor has a sampling rate of 5 kHz. As the LiDAR's rotational speed is variable, angular resolution and scanning rate are variable. The datasheet indicates that the scan frequency lies in the $4 - 10$ Hz range, while angular resolution is about $0.3 - 0.8^\circ$.

In the technical document, the expected accuracy is indicated as 1% over the entire measurement range. No expected precision is provided.

4.2. LiDAR Data Reception

The raw byte streams described above for both LiDAR models are continuously transmitted to the microcontroller through a UART interface and processed by a dedicated buffering library, `lidar_buffer`, which is introduced in this section. As mentioned in section 3.4, the Raspberry Pi Pico's Core 1 is responsible for continuously reading incoming LiDAR data, using this library.

This library defines the `LapBuffer` structure. Its central element is the main buffer, `buf`, a byte array that accumulates all raw packets belonging to a single LiDAR rotation. An index `write_ind` tracks the next write position in `buf` and is reset to zero when a complete lap is processed. A secondary buffer `pkt_buf` is used to assemble each packet as each byte is received, before it is transferred to `buf`. The index `pkt_idx` serves as the write index for `pkt_buf` and indicates how many bytes have already been received in the current packet. It is reset to `-1` to signal that no packet is currently being assembled. Once the

expected packet length `pkt_len` is known, a packet is considered complete when `pkt_idx = pkt_len`. The complete packet is validated and copied into `buf`.

The library introduces the `lap_buf_feed` function, that is designed to be called whenever a byte is received. This function performs a bit-level state machine, using the `LapBuffer` variables and gradually filling `buf`.

Synchronization is ensured by waiting for the sensor-specific sync byte, `0xFA` for the HLS-LFCD2 and `0xAA` for the Delta-2A. These bytes can appear in data blocks, so the checksum and angle validity checks are used to reject any possible false synchronizations.

Packet length determination differs between the two sensors. The HLS-LFCD2 has a fixed packet length of 42 bytes, so `pkt_len` is defined immediately when a sync byte is detected. The Delta-2A encodes the packet length in bytes 1–2 of the packet header, so `pkt_len` can only be determined once at least 3 bytes have been accumulated. An explicit range check discards packets whose length field is implausible (below 8 or above 200 bytes). As that might mean that the wrong sync byte was used, `pkt_idx` is reset to `-1` and synchronization is restarted.

Lap detection is performed by comparing the start angle of each completed packet with the previous one. A decrease of more than 20° indicates a wrap-around from $\sim 360^\circ$ to 0° , marking the end of one lap and the beginning of the next. The threshold of 20° was chosen to be large enough to tolerate the angular span of a single packet (6° for the HLS-LFCD2 and 22.5° for the Delta-2A), while remaining small enough to avoid false detections. The packet where a lap wraparound is detected belongs to the new rotation, as the angle has already decreased. However, it is still appended to the buffer before `lap_complete` is set.

If the main buffer does not have sufficient space to accommodate an incoming packet, the packet is discarded and the angle tracking is reset. This prevents buffer overflow at the cost of losing one packet.

4.3. Processing LiDAR Data

To process the LiDAR data, the `process_lap()` function was implemented. It converts raw bytes stored in the lap buffer into Cartesian point clouds and detected line segments. The line detection algorithm will be introduced in the next chapter; this section will introduce the LiDAR data processing. The function is designed to handle both LiDAR sensors studied. Independent of the model, the function follows this structure: it clears the output arrays, iterates over the buffer to decode and validate individual range measurements,

projects valid measurements into the robot frame, stores them into the output arrays, and invokes the `find_lines()` algorithm.

The `angular_resolution` of the Delta-2A LiDAR is estimated in this function, as the sensor's spinning velocity is not fixed. As packets are parsed, two accumulators are updated: `angle_span_total` increments 22.5° per packet and `point_total` which counts the points. Only packets where the gap between its start angle and that of the previous packet is below 5° contribute to the accumulators, excluding possible discontinuities which would degrade the estimation. At the end of the scan, the resolution is computed as `angular_resolution = angle_span_total/valid_point_total`, obtaining the mean degrees per sample.

Coordinate projection. Each valid range measurement d at the LiDAR angle θ is projected into Cartesian coordinates using lookup tables for sine and cosine, pre-computed at a resolution of 0.25° to reduce run time by avoiding floating-point trigonometry. Empirically, this step decreased the run time of this function by several milliseconds. The projection formula differs between sensors.

For the HLS-LFCD2, the projection is

$$x = d \cos \alpha \quad , \quad y = -d \sin \alpha \quad (4.6)$$

The sensor spins clockwise, whereas the trigonometric convention used in the lookup tables and in mathematical equations in this system assumes counter-clockwise angles. Swapping the y signal restores a standard right-handed frame.

The Delta-2A also rotates clockwise, but its measurement angles place its zero-angle reference rotated 90 relative to the HLS-LFCD2 and to the robot reference frame used in this work. As a result, the axes are swapped but not negated:

$$x = d \sin \alpha \quad , \quad y = d \cos \alpha \quad (4.7)$$

Both formulations produce equivalent right-handed Cartesian frames in the robot's coordinate system: the X axis aligns with the robot's forward axis.

Delta-2A: Point count per packet. Each Delta-2A packet carries a variable number of range measurements N , computed as

$$N = \frac{\text{pkt_len} - 13}{3}. \quad (4.8)$$

This equation differs from that in Table 4.2, because it uses `pkt_len` in place of Payload Length. The offset is 13, because `pkt_len` contains the payload length and the 8-byte header, with the first measurement byte beginning at byte 13.

Delta-2A: Angular offset correction. The start angle obtained from each packet is corrected by subtracting a fixed offset of $+4$. Without this correction, the point cloud is rotated relative to the arena walls, causing the wall-matching step to produce a systematic heading error in the pose estimate. The offset was determined experimentally by comparing the reconstructed map against the known arena geometry.

Range gating. Points with $d < 0.02$ m are considered noise, and those with $d > 3.00$ m are discarded to only consider measurements within the competition track.

Repeating the parsing of the lap buffer. The HLS-LFCD2 version of the function directly reads and processes the raw byte data by going one packet at a time. The Delta-2A branch re-parses the byte stream: the `lap_buf_feed` function operates under strong timing constraints and stores bytes as they arrive over UART, doing only the essential processing of the data. Determining and storing the packet framing in this function would require more complex synchronization and risk losing bytes due to lack of overhead at high baud rates. Instead, this overhead is spent once per lap inside `process_lap()`, where timing is not critical.

4.4. Experimental Setup

The robot was placed directly facing the center of a 32 cm wide white wall. This length was chosen, because it is the length of a machine wall in the RAF4 track. The robot's position was manually adjusted to measured, known distances along a line in the ground, perpendicular to the wall. This setup is represented in Figure 4.1

For each distance tested, the Python script was executed on a computer to receive the robot's distance measurements. Two hundred LiDAR distance measurements at the

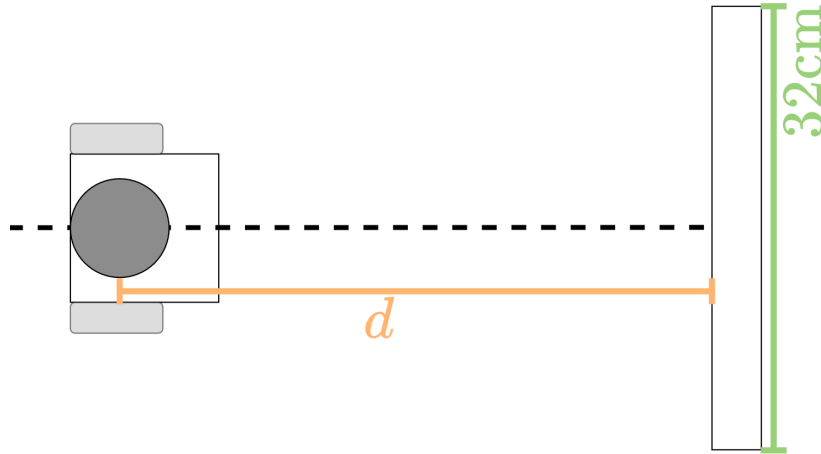


Figure 4.1: Experimental setup used to calibrate the two LiDAR sensors tested.

angle 0° (directly in front of the robot) were recorded. This process was done at distances from 10 to 150 cm (in 2 cm steps).

The collected data were then analyzed: the measurement error was computed, and a linear calibration function was fitted when necessary. The calibration function is applied to each distance measurement in the `process_lap` function. The entire calibration is repeated to validate the calibration function.

This experimental setup is simple and intuitive. Using a line in the ground ensures the correct distance to the wall and the alignment with the center of the wall. However, ensuring that the robot is facing exactly the center of the wall is much more difficult and these heading errors will introduce distance measurement errors that increase with the distance to the wall.

As mentioned before, the voltage to the Delta-2A DC motor can be easily adjusted by the robot. Therefore, it is relevant to better understand the sensor's behavior at different rotational speeds. The robot with this sensor was placed in a location without any object in a 3 meter range. With the robot stationary, the motor's voltage was changed from 2.8 V up to 4.8 V, in increments of 0.1 V. Above and below this boundaries, the rotational speed does not allow proper measurements and no ranging data is transmitted. For each voltage tested, the following quantities are stored for two hundred LiDAR scans: rotational velocity, signal quality, angular resolution and scan intervals. The objective of this test is to determine the optimal value for the DC motor's voltage.

4.5. Results : HLS-LFCD2

After storing two hundred ranging measurements at each test distance, the values are averaged and their standard deviation computed. The figure below displays how the distance measurement error and standard deviation evolve with distance to the wall.

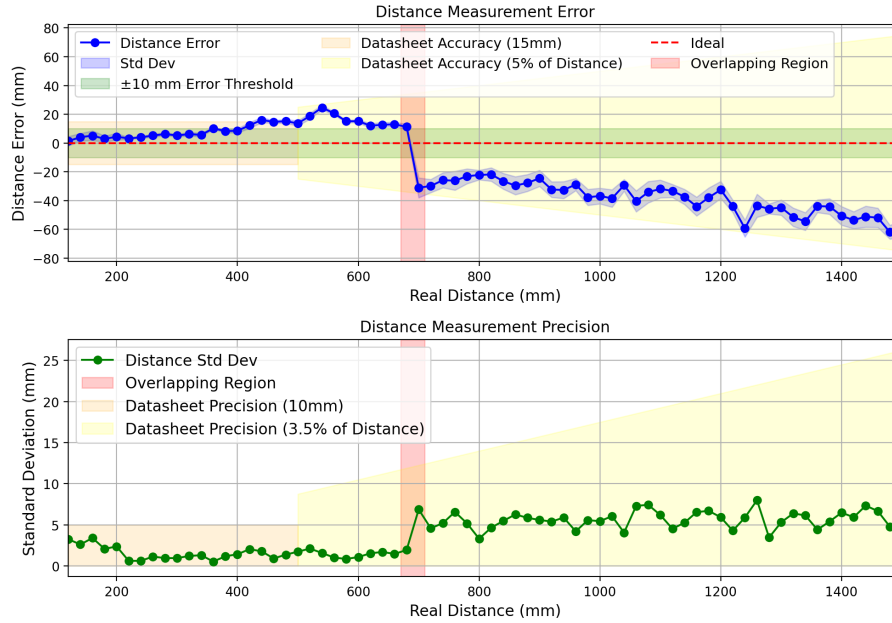


Figure 4.2: Error and Standard deviation measured without calibration. The yellow and orange bands represent the precision or accuracy indicated in the datasheet for distances below and above 50 cm, respectively. Since the datasheet defines precision as a 2σ interval (corresponding to a 95% confidence level assuming normally distributed data), the plotted thresholds are shown as half of the indicated precision values to allow direct comparison with the measured standard deviation (σ).

The behavior described in the datasheet is partially observed, as the accuracy and precision remain within the predicted bounds. However, a more pronounced and notable transition is observed at around 70 cm: for distances below 70 cm, the sensor overestimates the distance; above 70 cm, it underestimates it. The standard deviation tends to increase gradually, but also presents a noticeable jump in this region, marked red in the figure.

The region presents an overlapping behavior, where a measured distance d may correspond to two real distances, which prevents the use of a single linear calibration function. To minimize the maximum calibration error, the data was fitted with a linear calibration

function with two branches separated at the center of the overlapping region. The calibration function, with D being the uncalibrated measurement, is defined in meters as:

$$d = \begin{cases} 0.967949D + 0.0025 & , D < 0.690 \\ 1.018437D + 0.0141 & , D \geq 0.690 \end{cases} \quad (4.9)$$

The calibration function was implemented in the robot and applied to each distance measurement. The experimental procedure was repeated, as explained in section 4.4. Its results are represented in Figure 4.3.



Figure 4.3: Distance measurement error and standard deviation after calibration as functions of the real tested distances. The region marked in red indicates the distance interval with significantly higher error and variability, revealing reduced measurement accuracy and precision in that range.

It is immediately evident that the calibration reduced the overall error, falling below 1 centimeter. However, a problematic zone becomes evident in the [67,73] cm range, where both accuracy and precision degrade significantly. This happens because this interval corresponds to the transition region between the two branches of the calibration function. In this region, consecutive measurements of the same real distance may fluctuate around the boundary value used to select the calibration branch. As a result, some of those measurements are processed using one calibration branch, while others are processed using the other. This random switching introduces large artificial fluctuations in the calibrated distance, increasing error and standard deviation.

It should be noted that the two-regime behavior of the LiDAR used will always introduce difficulties. More complex models could reduce the maximum error, but were deemed unnecessary for the application targeted.

4.6. Results : Delta-2A

4.6.1 Distance Calibration

After completing all measurements, the ranging measurements for each testing distance are averaged. Figure 4.4 displays the measured error and standard deviation at the testing distances.

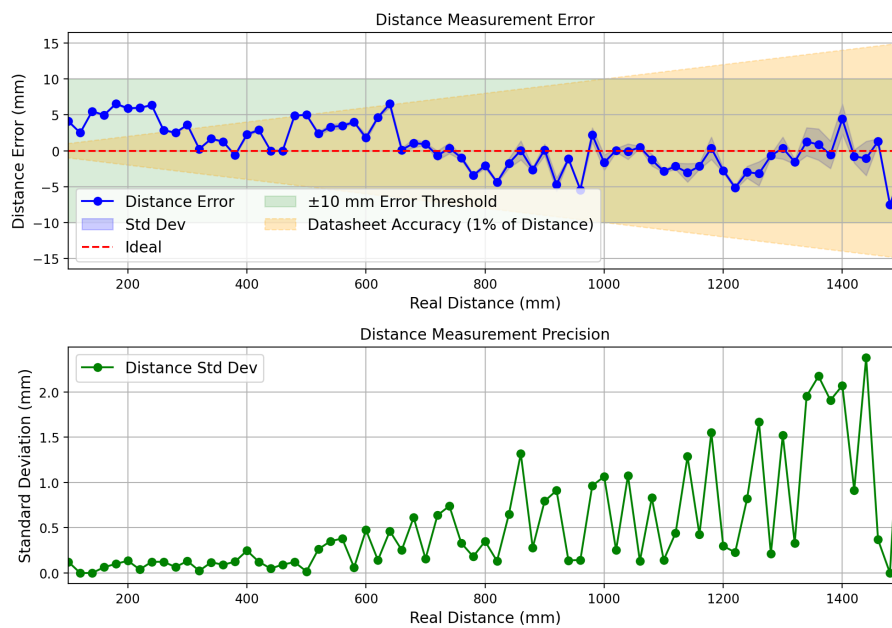


Figure 4.4: Error and Standard deviation measured with the Delta-2A without calibration. No expected precision values were provided by the datasheet.

Once again, the behavior predicted by the datasheet is mostly observed; the measurement lies below the 1% threshold for most of the testing range, only exceeding it at distances below 30 cm. Nevertheless, the absolute error remains below 1 centimeter in the entire range, showing an accurate and reliable performance.

The standard deviation plot shows a general increase with distance, which is expected. Although this increase is not monotonic, the standard deviation's upper boundary and average level both increase with distance, which suggests that the sensor noise grows with range. Nevertheless, this quantity mostly remains below 2mm.

Given the high accuracy and precision achieved by this sensor, no calibration function was fitted. It would be nearly constant with a small offset, introducing unnecessary complexity without providing reasonable benefit.

4.6.2 Voltage Calibration

Figure 4.5 summarizes the effect of various motor supply voltage on four parameters.

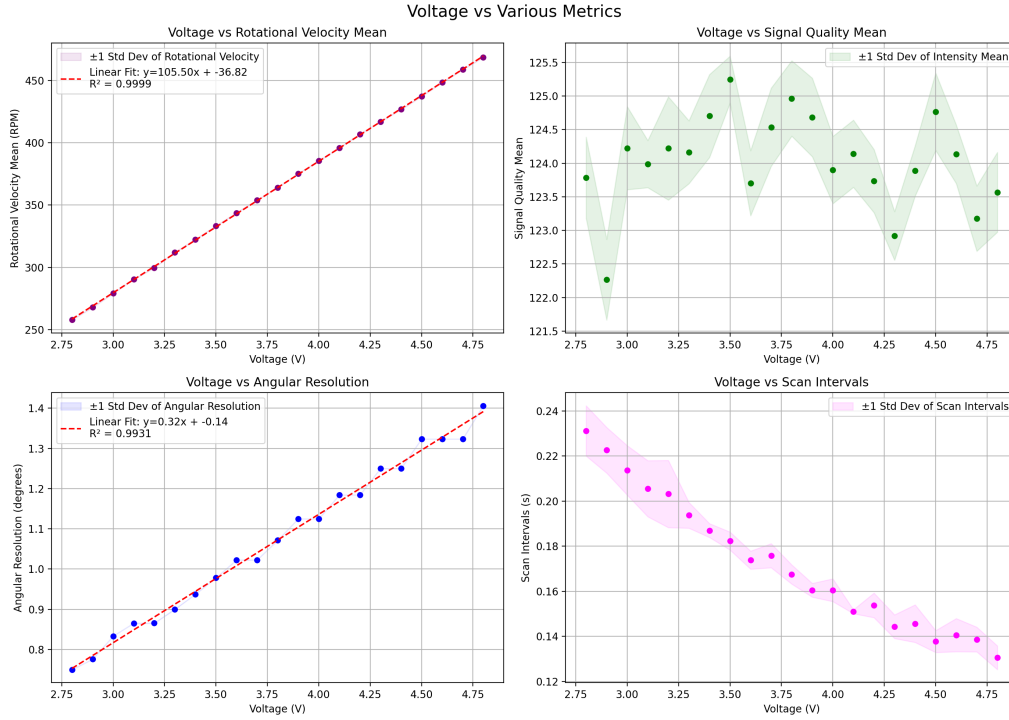


Figure 4.5: Evolution of four main quantities as the Delta-2A DC motor voltage is increased.

Rotational speed increases linearly with voltage ($R^2 = 0.9999$), evolving with $\omega \approx 106.75 U - 44.63$ (RPM). This confirms the DC motor behaves as expected across the tested range of [2.8,4.8] V. Throughout the testing range, the standard deviation measured for this velocity remains around 0.3 RPM, indicating high stability and predictable behavior.

The angular resolution, as expected, degrades proportionally with the voltage and rotational speed ($R^2 = 0.9931$). Higher speed, with a constant sample rate, means that fewer points per degree are acquired. At 3.6 V, the resolution is approximately 1.0° , rising to 1.4° at 4.8 V. This behavior follows the linear tendency consistently, but a discretization is evident at voltages above 3.5 V.

The scan interval decreases with voltage and lies below 0.2 s above 3.4V, which is expected as the rotation speed increases.

Signal quality does not show a clear trend with voltage, remaining roughly constant around 140–146 a.u. There is a significant amount of variability, indicating that motor speed does not meaningfully affect measurement intensity inside this range.

In the context of localization of a mobile robot, it was decided that the most relevant parameters are the angular resolution and the scan intervals. They define a fundamental trade-off, between high point density in a scan and elevated scan rate.

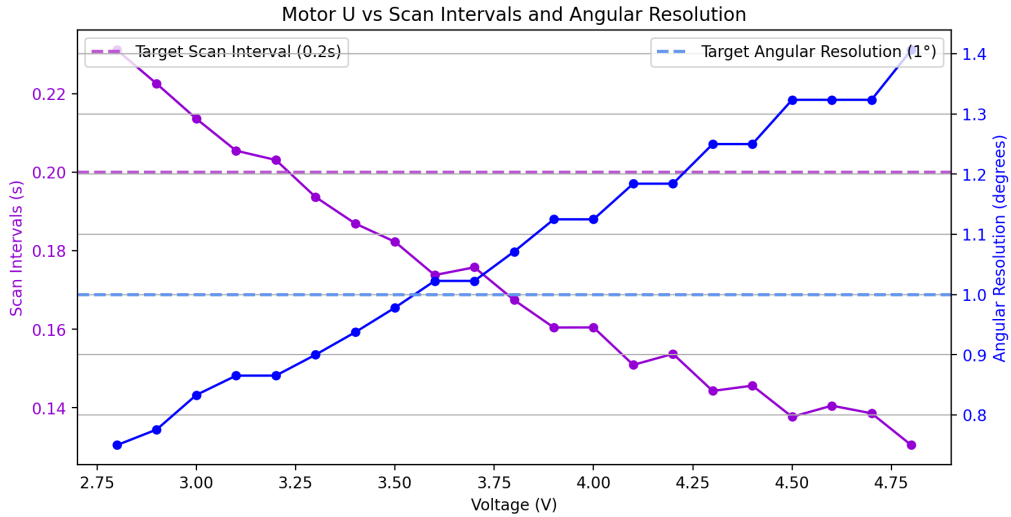


Figure 4.6: Comparison between the two parameters chosen to decide the optimal voltage value for the Delta-2A DC motor.

Based on these observations, a supply voltage of 3.4 V offers a favorable trade-off, achieving better scan rate and angular resolution than the HLS-LFCD2.

4.7. LiDAR comparison

Finally, to properly compare the performance of both sensors, their errors and standard deviations are shown in the figure below.

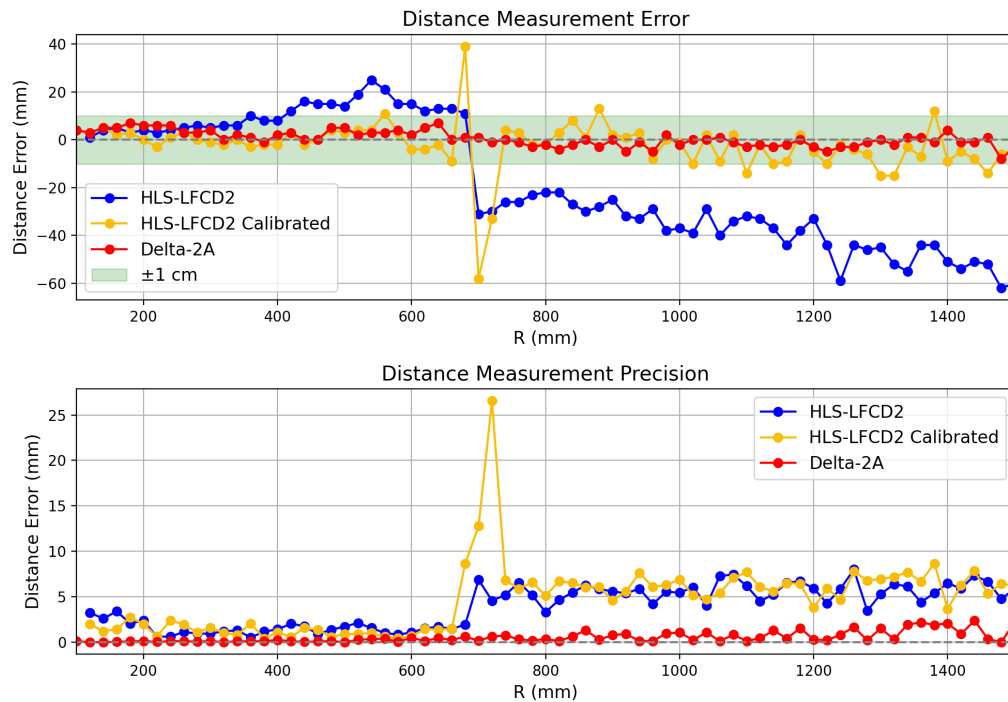


Figure 4.7: Comparison of the Error and Standard deviation obtained with both LiDAR sensors.

It is immediately evident that the Delta-2A consistently performs better than the HLS-LFCD2, with higher accuracy and precision in the entire testing range. Directly comparing data from HLS-LFCD2 before and after calibration emphasizes that the calibration reduced the overall error, while keeping the same precision profile, apart from the problematic zone where increased variability was measured.

5. Line Detection Algorithm

Having calibrated the LiDAR sensors, it is now time to use their data to understand the robot's surrounding environment. Considering the track's geometric, organized and known layout, a line detection algorithm was chosen. By comparing where two walls were detected and with which orientation, the robot's pose can be estimated. This section presents the line detection algorithm developed.

5.1. Methodology

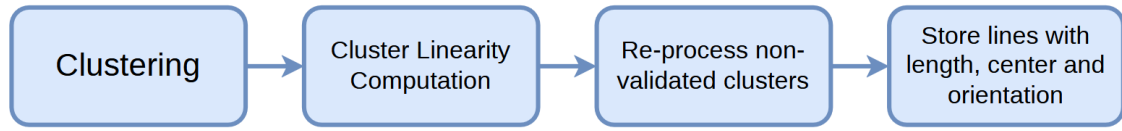


Figure 5.1: Schematic representing the functioning of the line detection algorithm.

Figure 5.1 displays the main steps of the proposed line detection algorithm, which is applied to LiDAR measured 2D point clouds, that represent the robot's surroundings at each frame. The LiDAR data processing algorithm in section 4.3 converts the buffer with raw byte data into arrays with the valid point's Cartesian coordinates. As it is mentioned in the algorithm's section, it ends by feeding these arrays to the `find_lines` algorithm. The proposed algorithm uses these coordinates to detect walls and their centers, lengths and orientations.

5.1.1 Clustering

In ranging data measured radially from a single point, walls visible to the sensor appear as clusters of points arranged in straight lines. Given this insight, the algorithm starts by dividing the point cloud into clusters. This is done by analyzing pairs of consecutive points in the point cloud. Two points belong to different clusters when the euclidean distance between them is above a certain threshold.

The clustering threshold was defined using the fact that two LiDAR beams are separated by approximately a fixed angular resolution. Considering a LiDAR sensor with an angular resolution of δ , two consecutive beams will diverge by a distance of $d \tan(\delta)$ after propagating a distance d . To ensure robustness with noisy data, a sensibility parameter τ was included:

$$\text{Cluster Threshold} = d \tan(\delta) \cdot \tau \quad (5.1)$$

For each pair of consecutive points, the clustering threshold is computed with d as the mean of the points' radial distances to the LiDAR. This is done to ensure the same distance and resolution for both points, and the approximation is reasonably accurate if the points lie on the same line.

A clustering method avoids analyzing large amounts of points and checking if they belong to a line, something that is often done by incremental line detection algorithms. Furthermore, isolated points are also discarded by not being grouped to any cluster.

5.1.2 Cluster Linearity

After dividing the point cloud into clusters, the 2×2 covariance matrix of each cluster is defined by computing the set's mean and variation. Using its eigenvalues, the linearity of the cluster is determined with the following equation[33]:

$$\text{Linearity} = \frac{\lambda_1 - \lambda_2}{\lambda_1} \quad (5.2)$$

Where λ_1 is the largest eigenvalue of the covariance matrix and λ_2 the smallest.

This criterion was chosen due to its high consistency and robustness to noise. Computing the linearity of the entire cluster provides information on whether it is a line, while being a relatively simple operation. Clusters whose linearity is above a threshold ε are considered as valid lines and stored for the localization algorithm.

5.1.3 Non-validated clusters

To accurately localize the robot, it is imperative that all visible walls are used, so non-validated clusters must be analyzed. Multiple factors may cause a cluster to have a linearity value below the threshold ε . It may include wall corners and noisy boundary points, it may be non-linear or be too noisy overall.

Among these factors, the most likely to be useful is the presence of a corner or noisy line boundary, where at least one valid line can be contained. Then, the algorithm calculates the local linearity at each point in the cluster using $2n$ neighboring points, n on each side. Here, n is a parameter that controls the sensitivity of local linearity analysis.

As each point is analyzed, the value and index of the point with the minimum linearity are updated. After scanning the whole cluster, it is divided into two subclusters, along the minimum linearity point. Consistently, this point corresponds to a corner or the endpoint of a line. Afterward, each subcluster is analyzed on its own, as explained in Section 5.1.2.

This step can involve tens of local linearity computations per cluster, representing a reasonably expensive process. However, this is only performed one time and useful information can be retrieved. No subcluster is subdivided a second time.

Admittedly, this step is often unnecessary in the context of the RAF4 competition, where walls are well separated and unobstructed. Nevertheless, in the event of any object being close to the track borders, this process would prevent erratic detection of the warehouse walls.

All parameters and tolerances used in this algorithm were optimized by trial and error. They were adjusted until the robot could reliably detect walls from multiple distances and directions.

5.1.4 Storing walls

Clusters that are validated as walls are stored for the localization algorithm. Each valid cluster is fitted using 2D Principal Component Analysis (PCA). This fitting method was chosen as it relies on the already programmed computation of the covariance matrix and its eigenvalues. Using the matrix eigenvectors, the first and last points of the cluster are projected onto the principal component direction. Using these projected points, the length, center and orientation of the fitted line are calculated. The wall's orientation is measured between the wall and the X-axis and normalized to the $[-\pi, \pi]$ rad interval.

5.2. Experimental Setup

In order to understand the performance of the Line Detection Algorithm, the procedure explained in section 4.4 was modified and repeated. The robot is, placed on top a line facing the center of white wall with 32 cm length. In each distance tested, two hundred measurements of the wall's length were stored.

Once again, while the robot's distance to the center of the wall is measured and controlled, the robot's heading cannot be easily adjusted visually. For this reason, only the wall length estimation was tested, as it is independent on the robot's position and heading. The wall center coordinates and wall orientation estimates were not explicitly tested.

However, these three parameters are fundamental to the localization algorithm, so they will be indirectly evaluated in section 6.3. The wall length is the only parameter independent from the robot's state, so it has an essential role in the localization algorithm. For this reason, the goal of this section is to define an appropriate tolerance for wall detection.

5.3. Results : HLS-LFCD2

The analysis method was the same as in chapter 4. For each test distance, the wall length error was calculated for each sample. Then, the average and standard deviation of the errors in each test distance were computed. The figure below displays the results observed.

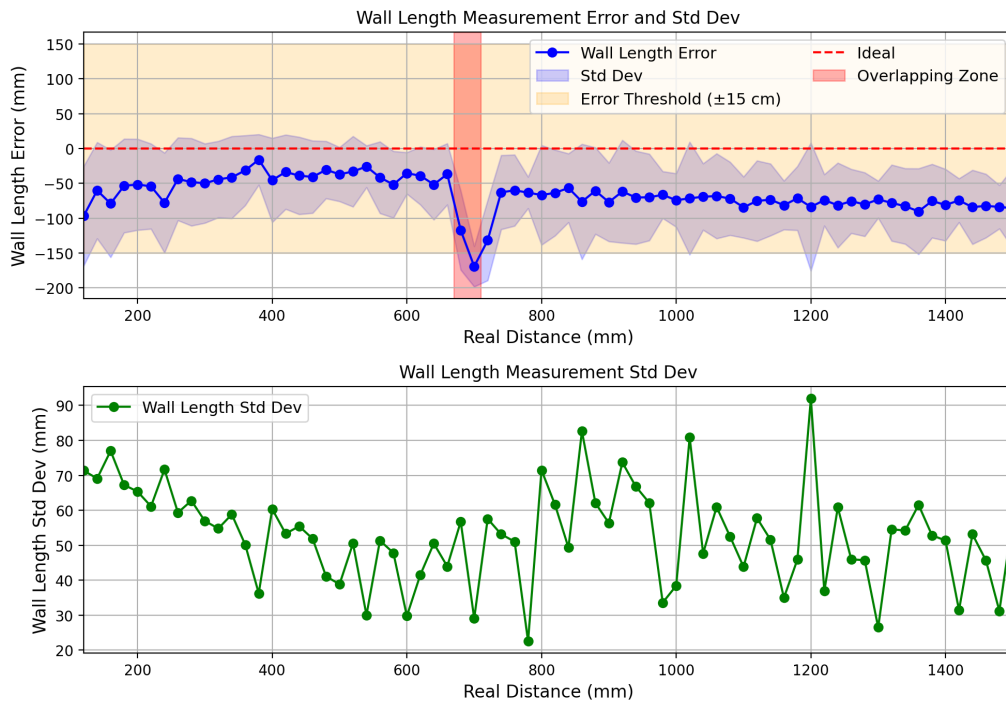


Figure 5.2: Comparison of the Average and Standard deviation of the wall length errors.

The estimation error average stabilizes around 5 cm (15% error). However, at several distances the error greatly surpasses that value, sometimes reaching 10 cm (30% error). In chapter 4, this sensor's overlapping region degraded the distance measurements and their precision. Naturally, the region also degrades wall length estimation, causing the highest error measured, at over 15 cm (45% error).

Another relevant aspect to point out is the errors' high standard deviation. This strong instability, coupled with the elevated error, makes it difficult to define a fixed wall length tolerance for the localization algorithm. The only value tolerance that can accommodate the errors across the entire range is shown in the figure: 15 cm. However, this value is

too high compared to the walls present in the RAF4 track: machines measure 32 cm, while warehouses have 62 cm of length.

5.4. Results : Delta-2A

The following figure presents the results obtained from the analysis of the wall length data:

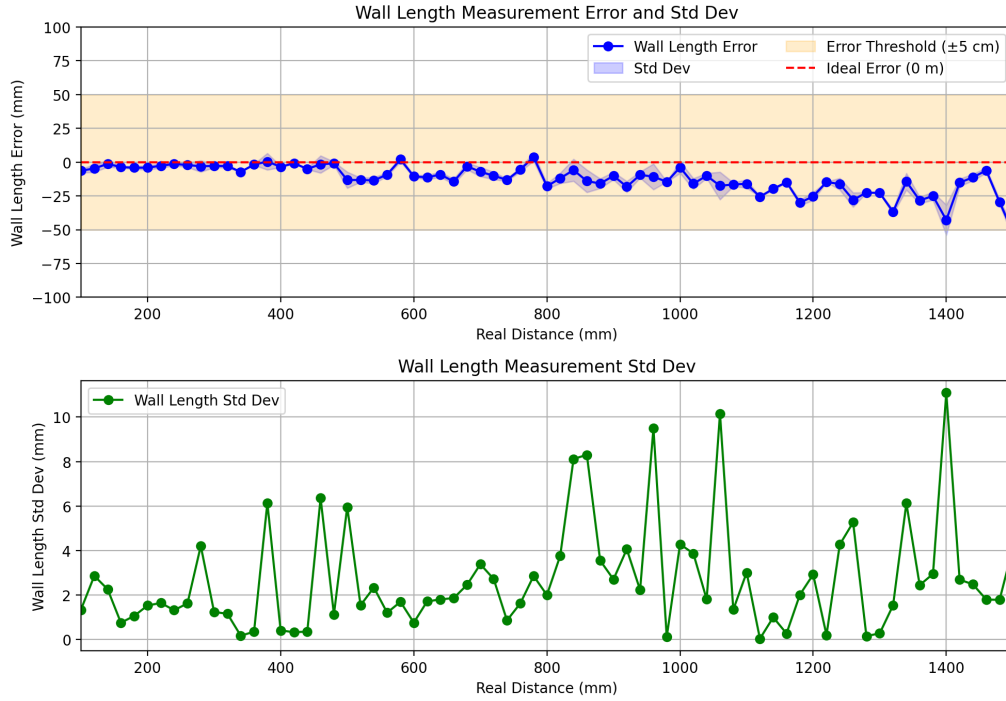


Figure 5.3: Comparison of the Average and Standard deviation of the wall length errors.

The wall length estimation average error gradually grows in absolute value with the distance. At short ranges, at least one LiDAR beam intersects the wall very close to each corner, so the wall endpoints can be estimated accurately. As the distance increases, the fixed angular resolution of the sensor causes the spacing between consecutive beams to grow. Consequently, the wall corners are no longer hit by any beam, increasing the endpoint and wall length estimation errors. Nevertheless, the error remains below 5 cm for distances up to 1.5 m. Therefore, a wall length tolerance of 5 cm can be adopted.

5.5. LiDAR comparison

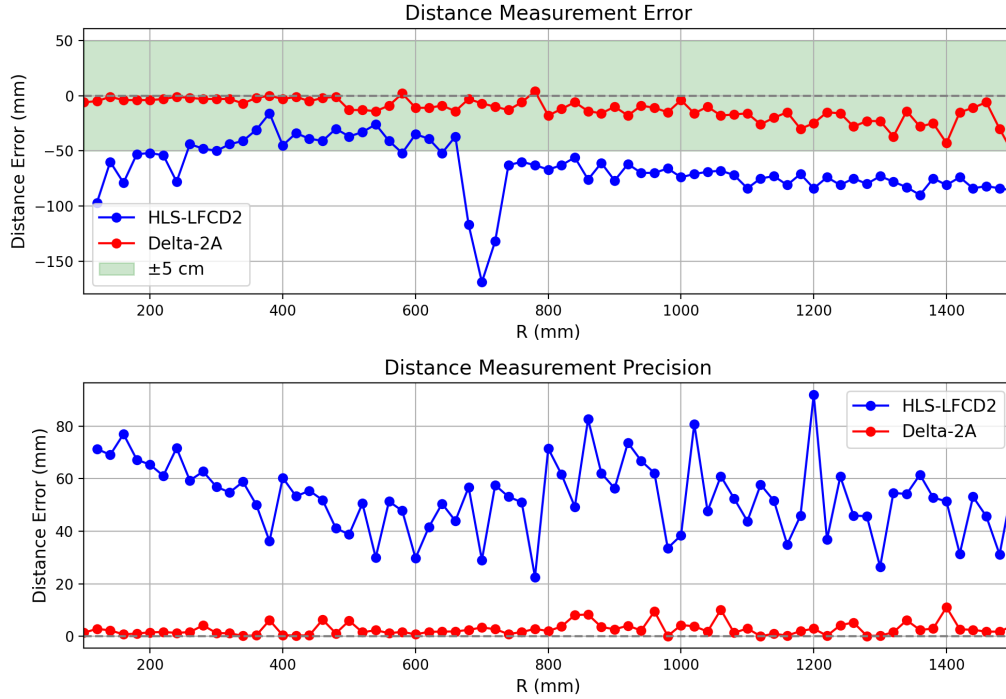


Figure 5.4: Comparison of the Average and Standard deviation of the wall length errors for both LiDAR models.

Like in section 4.7, the Delta-2A LiDAR consistently outperforms the HLS-LFCD2. Both the wall length error and standard deviation are substantially lower. In particular, the wall length standard deviation of the Delta-2A is typically an order of magnitude smaller, which indicates significantly higher measurement stability and repeatability.

Overall, the results of both comparisons performed identify the Delta-2A as the superior and more reliable model. Although its communication protocol requires a more complex reading and parsing process, it justifies that effort with higher performance. Moreover, its wall length measuring performance makes it possible to adopt a fixed wall length tolerance of 5 cm. In contrast, the HLS-LFCD2 would require a tolerance about three times larger.

In conclusion, both LiDAR models demonstrated that the proposed algorithms works, with varying levels of success. As both models used precisely the same tolerances and code, the variability in results appears to be the result of the sensors' different ranging performances, as the results are consistent with those in section 4.7.

6. Localization Algorithm

Using the line detection algorithm as tested base, this sections proposes an efficient and optimized localization algorithm based on feature matching of linear features.

6.1. Methodology

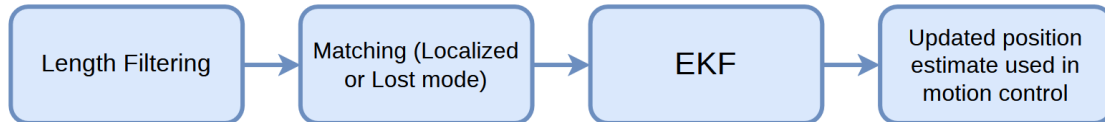


Figure 6.1: Schematic representing the localization algorithm. It is performed for each LiDAR scan measured.

Figure 6.1 illustrates the main steps of the proposed localization algorithm. It starts with the lines detected in a LiDAR scan, by the localization algorithm. Then it follows a feature matching approach, where detected walls are matched to reference walls in the track map. The resulting matches are fed to an EKF, that can easily fuse all wall matches performed in a LiDAR scan into a single pose estimation. This algorithm was defined and optimized for the RAF4 competition, but the reasoning presented can be applied to various environments that can be represented as detectable linear features.

This section describes the complete localization algorithm with detail, explaining every step performed. This algorithm is executed in the main loop of the `main.cpp`, after the LiDAR raw data is processed and lines are detected. From this point, it operates directly with the linear features detected, namely the wall centers, lengths and orientations.

The first stage of the algorithm consists in matching the observed walls with the reference map. To improve robustness, two matching modes were implemented: *Localized* and *Lost*. The *Localized* mode is used when the robot has a reliable pose estimate, while the *Lost* mode is activated when the robot's estimated pose is no longer trustworthy, after a localization failure or a "kidnapped robot" event.

6.1.1 Length Filtering

In this step, each detected wall is assigned a type according to its length: "machine", "warehouse" or "invalid". Each wall's length is compared to machine and warehouse wall lengths. If the difference is below the tolerances defined in chapter 5, the walls are

considered to be the same type. This is the reason why the HLS-LFCD2's tolerance of 15 cm is problematic: it enables many mismatching of wall types.

Only "machine" and "warehouse" walls will be considered by the algorithm from this point onward, "invalid" walls are ignored. Additionally, walls may later be assigned a fourth type: "matched", to avoid matching the same detected wall with multiple reference walls.

Both matching modes operate on the detected walls relying on their wall types, determined only once, in this stage. This stage is performed to increase computational efficiency: walls that are invalid are never utilized again and a valid detected wall cannot match with a reference wall of opposing type. In other words, this process prevents many unnecessary computations and comparisons.

6.1.2 Matching: "Localized" Mode

This matching mode is computationally efficient, but relies heavily on the previous pose estimation performed. For this reason, this mode is applied when the pose estimation process is working properly and the robot's pose estimate is considered reliable.

The mode compares each reference wall r to every detected wall i of the same type. First, the orientations of the two walls are compared using the cross product. Matching walls should have the same orientation, so pairs whose cross product exceeds a threshold $PROD_TOL$ are rejected. This allows early exit from "Localized" mode, avoiding significant error accumulation and unnecessary comparisons.

The distance between wall centers d_{ir} is computed using the previous pose estimation (x_e, y_e, θ_e) to estimate the detected wall's center in the global reference frame:

$$\begin{cases} x_{i,global} = x_e + C_{x,i} \cos(\theta_e) - C_{y,i} \sin(\theta_e) \\ y_{i,global} = y_e + C_{x,i} \sin(\theta_e) + C_{y,i} \cos(\theta_e) \end{cases}$$

where $(C_{x,i}, C_{y,i})$ are the coordinates of the detected wall's center, in the robot reference frame. Then, naturally, the reference wall's center coordinates (x_r, y_r) are used to calculate d_{ir} :

$$d_{ir} = \sqrt{(x_{i,global} - x_r)^2 + (y_{i,global} - y_r)^2} \quad (6.1)$$

Then, a distance check is performed. Pairs whose distance between centers is above a threshold MAX_CENTER_DIST are discarded. If a pair has the matching orientation and

types but an elevated center distance, they are likely swapped. For example, this result would be observed if the reference wall is the incoming warehouse and the detected wall is the outgoing warehouse. This check prevents making a bad match that would cause a 180° rotation of the reference frame when estimating the pose.

If both checks are passed, the following cost function is computed for each wall pair:

$$\text{Cost} = w_d d_{ir} + w_a \|\mathbf{t}_r \times \mathbf{t}_i\| \quad (6.2)$$

Where w_d, w_a are the weights attributed to distances and angles in the cost function. d_{ir} is the distance between the position of the reference wall's center and the estimated position of the detected wall center in the global reference frame. The second term of the cost function is the cross product between the tangent vectors of the two walls, given by the sine of the difference of their orientations, that was already computed in the orientation check.

For each reference wall, the detected wall with minimum cost is selected if its cost is below a threshold `MAX_COST_COMBINED` and its type becomes "matched".

As a final note, the order in which reference walls are analyzed can affect the resulting matches. This happens because, in certain cases, more than one detected wall may fall within tolerance for a given reference wall. For example, consider a situation where the right machine wall corresponds to detected wall $i + 1$, and the left machine wall to detected wall i . If the right machine wall is processed first and detected wall i happens to fall within its tolerances, the algorithm may incorrectly match it to this reference wall, leaving wall $i + 1$ to then be (also incorrectly) matched to the left machine wall. This kind of swap was not explicitly prevented in the implementation, as it did not pose a problem during practical testing.

6.1.2.1 Number of matches required

If fewer than 2 matches are made after evaluating every wall, the "Localized" mode does not return any correspondences. If this happens, the robot is considered lost. In this case, the previous pose estimation is deemed unreliable, and the robot switches to the "Lost" mode.

This requirement of at least two matches follows from the geometry wall-based localization. A single match provides two measurements: the perpendicular distance (which is

explained in subsection 6.1.3.1) and its orientation. The perpendicular distance to vertical walls constrains the robot's x coordinate and to horizontal walls constrains the robot's y . The orientation constrains the heading estimate. But a single wall always leaves one of these undetermined: the coordinate in the wall's direction, which leaves space for translation in that axis. As a result, a single match is insufficient for full pose estimation, since the robot could be anywhere along a line parallel to the wall, always remaining consistent with the measurement.

Two non-parallel walls resolve this ambiguity. Each wall constrains one coordinate, so they define the robot's position. The presence of two orientations constrains the heading with redundancy.

Two parallel walls provide more information than a single wall, but still fall short of a full pose estimate. Their perpendicular distances constrain the same coordinate axis, and their orientations are equal, so the coordinate along the walls' direction remains undetermined. However, discarding one of the two matches would be wasteful: redundant distance measurements to the same axis improve the reliability of the estimate of that coordinate. For this reason, both matches are kept and used, with the missing coordinate resolved by trusting the current heading estimate.

Nevertheless, single matches could still be employed, using them to update only the heading and one position coordinate, rather than being discarded entirely. This approach was rejected, as it would introduce additional complexity into the EKF implementation, going against this work's goal of defining a simple and intuitive localization framework. Furthermore, in the RAF4 track, there are only a few positions where less than two walls should be observed.

6.1.3 Matching: "Lost" Mode

This mode relies on the previous position estimate, but tolerates large errors, at the cost of higher computational load. The robot heading estimate is not used, as it is less reliable.

6.1.3.1 Perpendicular distance

Before describing the proposed algorithm, an important quantity must be introduced: the perpendicular distance from the robot to a certain wall. A valid wall i is detected with an orientation α_i and with its center in the position $(C_{x,i}, C_{y,i})$, both in the LiDAR's reference

frame. This quantity is the shortest distance from the robot to the infinite line containing the wall, given by:

$$d_{\text{perp}} = |C_{x,i} \sin \alpha_i - C_{y,i} \cos \alpha_i| \quad (6.3)$$

This metric was employed because it has no dependence on the robot's heading estimate. Moreover, it is robust to partial wall observations and constrains the robot's position relative to the wall.

6.1.3.2 Relative Comparisons

The matching process begins by comparing each valid detected wall i with all other valid walls j . For each pair, three criteria are compared: the Euclidean distance between wall centers, the wall type compatibility and the cross product between orientations. The distance between wall centers is pose-independent and can be directly compared with known distances between reference walls, as shown in Figure 1.2. Additionally, the perpendicular distance to each wall is computed and used as a geometric metric for disambiguation. The wall lengths are not compared in this mode, they define the wall types that reliably encode that information.

The objective of this mode is to match each detected wall pair with a reference wall pair. This enables matching with less knowledge by establishing geometric relationships. A detected pair is rejected if their orientations or the distance between centers does not match any reference wall pair with the same type combination. This approach relies on the low probability of observing a wall pair with a valid geometric configuration that is not represented in the reference map.

When a valid pair is detected, the index of each wall is stored in a four-element array, where each element corresponds to a reference wall.

6.1.3.3 Pair with opposite types

Pairs of walls with different types provide the most reliable matching scenario, as they must be orthogonal (the walls are a machine and a warehouse). For such pairs, the perpendicular distances to each wall constrain the robot position independently along each axis: machine walls constrain the x-coordinate, while warehouse walls constrain the y-coordinate.

However, since a perpendicular distance is unsigned, each wall corresponds to two possible positions, one on each side of it: if a vertical wall is at a perpendicular distance of X meters, then the robot is X meters to the right or left of the wall. The current position estimate is used to solve this ambiguity, the sign of the difference between the estimate coordinate (x or y) and the reference wall's coordinate predicted with that perpendicular distance indicates on which side the robot must be. Because this sign is determined by the coarse position estimate rather than the heading, the disambiguation remains independent of the heading estimation.

6.1.3.4 Pair with the same type

When two walls of the same type are detected, there is less information available: the walls must be parallel so the perpendicular distances only constrain one coordinate (X for machines or Y for warehouses).

The perpendicular distances are compared with the values expected for each reference wall, from the current position estimate. For each candidate assignment, a cost is computed as the sum of the absolute differences between the observed and predicted perpendicular distance. The assignment with the lowest cost is selected. This approach differentiates parallel walls without using the robot heading estimate.

6.1.4 EKF

Finally, an EKF is implemented using the array of matched wall pairs to update the pose estimate. This stage is implemented in two passes.

The first pass computes the Mahalanobis distance for each match, using the same predicted covariance matrix for all matches, obtained in the previous pose estimation. This means that the Mahalanobis innovation gate is applied uniformly, with every match being independently compared to the same uncertainty estimation. Each match's Mahalanobis distance is compared to the innovation gate `gate_mah2` and marked as invalid if it surpasses the gate. In the second pass, only matches that were accepted by the innovation gate are used to update the state and covariance estimations. To achieve higher efficiency, the corrections are applied sequentially. This two-pass design prevents single bad matches from degrading the filter's performance.

The Q matrix and Mahalanobis threshold is adjusted according to the matching mode used, to represent the confidence placed in the pose estimation in each mode. The measurement covariance matrix R is adjusted according to the LiDAR model, to account for their different precision performance, as well as the matching mode used, to adjust the ratio between Q and R to optimize the EKF performance. Additionally, when using the HLS-LFCD2, this matrix is adjusted with the distance from the LiDAR center to the wall's center, to represent the two-regime behavior of this sensor. The exact values of this matrices were scaled by trial and error, until predictable and stable motion and pose estimation were achieved.

As a final note, all tolerances and parameters used in this algorithm were defined by trial and error. They were adjusted in practice, until the robot's pose estimations were stable and no bad matches were approved.

6.1.5 Implementation

In the robot's firmware, this algorithm is implemented with a single function `find_pose`, which is executed in every control frame. It implements all the stages detailed above. Additionally, this function also performs odometry updates when a complete LiDAR scan is unavailable, being the only point of the system responsible for pose estimation.

The proposed localization algorithm combines computational efficiency with various failure protections. The two-mode matching strategy allows the system to operate reliably under normal conditions, while having the capability to recover from pose estimation failure without requiring re-initialization. The length-based filtering stage reduces the number of walls to search significantly. The sequential EKF update strategy, combined with Mahalanobis gating, ensures that only geometrically consistent observations contribute to the pose estimate, which limits the amount of updates and the impact of erratic detections. Furthermore, the adaptive tuning of both Q and R matrices according to the matching mode and LiDAR model helps the filter reflect the quality of available information.

6.2. Experimental Setup

The testing process for this algorithm was only performed with the robot mounted with a Delta-2A LiDAR. This was done to assess the best results attainable with the means available. If the HLS-LFCD2 were used in these tests, there would be higher ranging error and lower precision, with much higher wall length tolerances. This would lead to more erratic matches and, overall, a worse localization performance.

Many tests were developed, with the goal of evaluating main aspects of this algorithm: EKF convergence capability, pose estimation covariance, performance in motion over long distances, execution time and a "kidnapped robot" test.

6.2.1 Convergence test

This test consisted of placing the robot in three representative points of the map and testing its convergence capability. The commands implemented in the communication architecture allow the user to directly update the robot's pose estimate. Using that, an artificial error was introduced to robot's pose in 8 directions, spaced 45° apart:

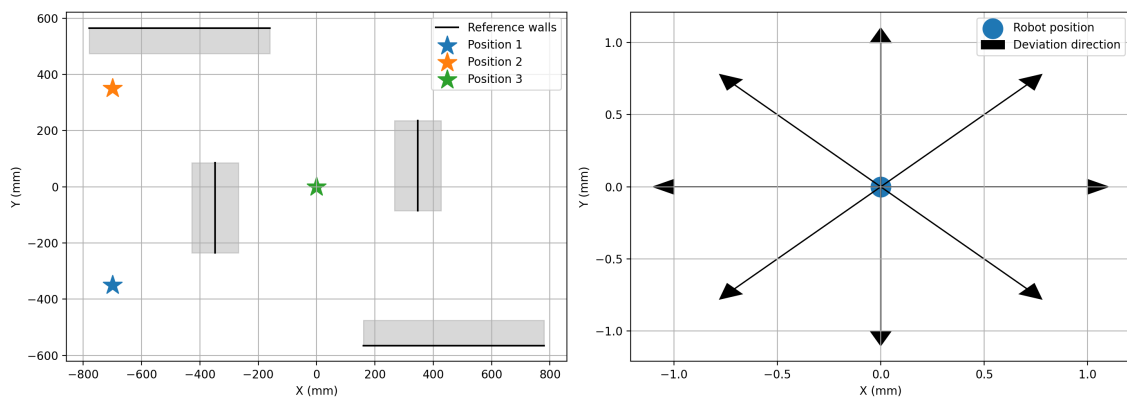


Figure 6.2: Figure displaying the tested Positions 1,2 and 3, as well as the 8 pose deviation directions applied.

In each direction, the deviation was implemented in 10 magnitudes, ranging from 10 cm to 1 m. Then, the robot's pose estimate over time was saved to the computer via the Python script for a duration of 2 seconds. With this data, the EKF's convergence profile is obtained at different directions and intensities. This test aims to characterize how quickly the pose estimate converges, and whether it converges to the correct pose.

It should be noted that, in the implementation of this test, the Mahalanobis innovation gates were significantly relaxed, to accept every pose update that the algorithm was able to compute. In a real deployment, this configuration would not be used, as it could lead to erratic pose updates. This test evaluates the full operating range of the algorithm, even though these extreme deviations would not normally occur during regular operation.

6.2.2 Covariance test

In this test, a map of 55 locations was defined:

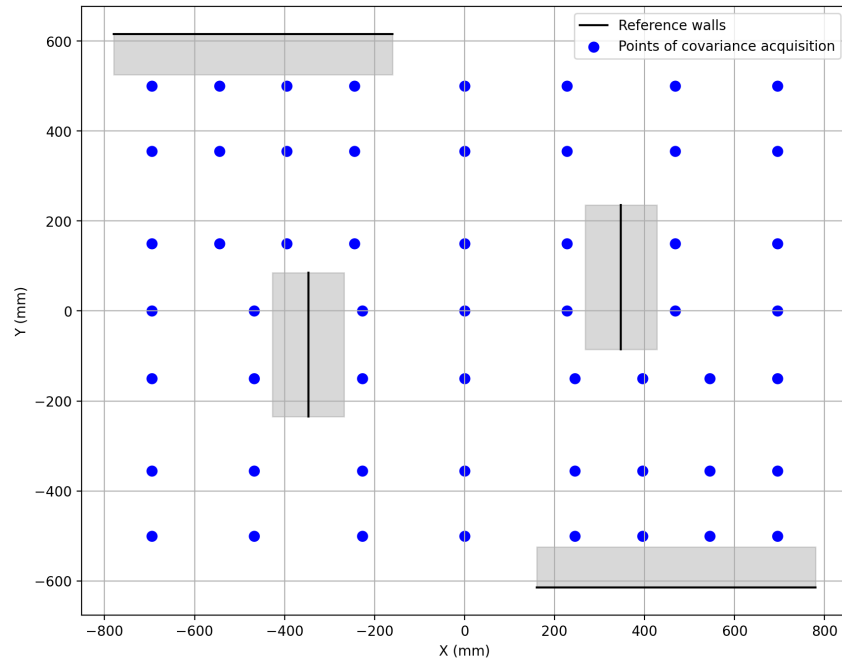


Figure 6.3: Map of the RAF4 track, with the points used in the covariance test marked.

The robot was placed in each of these locations. For each, a gross estimate of its pose was provided. Then, three different pose estimation routines were tested:

- EKF pose estimation, demonstrating the expected covariance of pose estimation with the robot at rest
- "Localized" mode pose estimation. It uses the EKF pose estimation as its base and applies matched walls directly into a pose estimation (using their orientations and perpendicular distances)
- "Lost" mode pose estimation. It uses the gross pose estimate provided as a reference point to solve ambiguity. Similar to "found" estimates, it converts matches directly to a pose estimate.

Pose estimations by "Lost" and "Localized" modes are not normally performed in isolation like in this test. This part of the test serves to evaluate the two modes' consistency and stability.

Using the python script, 50 estimations of the pose for each routine were recorded. The goal of this test is to generate covariance maps of the track, displaying where each routine varied the most.

6.2.3 Simple laps test

This test evaluates the robot's stability over long periods of time. In this test, the robot is simply commanded to execute the motion mode "Do Laps", explained in subsection 3.5.11. These laps are rectangular, with only straight segments. Using the Python script, the pose estimation, matching information and execution times were received by the python script for the 10 laps this motion mode controls.

With the pose estimation data, the stability of the trajectory perceived by the robot can be analyzed. The evolution of the EKF uncertainty matrix can be studied in the shape of its uncertainty ellipse. Finally, the execution time data transmitted sent is the time taken to execute each step of the localization framework: LiDAR data processing, line detection, wall matching and EKF update, which can be studied.

6.2.4 Complex laps test

This test is performed and analyzed in the same way as the Simple Laps test. The only difference is in the trajectory itself, as this test employs the "Complex Laps" motion mode, detailed in subsection 3.5.12.

This test evaluates the localization algorithm over long distances, with multiple rotations and arcs, testing all the motion modes implemented.

6.2.5 Kidnapping test

The Convergence and Covariance tests already evaluate the algorithm's capability to estimate its pose starting from a wrong estimate. This test complements them, by performing a "kidnapping" of the robot while in motion. The robot is commanded to follow the "Do Laps" motion mode. While the robot performs a lap, it is manually moved to a different point of the track without being provided any indication. The robot was moved as fast as possible, without moving its wheels or rotating it. During this process, the pose estimation data is recorded by the Python script.

In other words, this is a test of the kidnapped robot problem. Because no routine was developed to make the robot stop and return to a predetermined place in case of it being lost, it will keep moving. There also aren't any measures to explicitly avoid walls. Therefore, in this test, it is tested where the robot estimates it was placed, and it is observed what it does in that case.

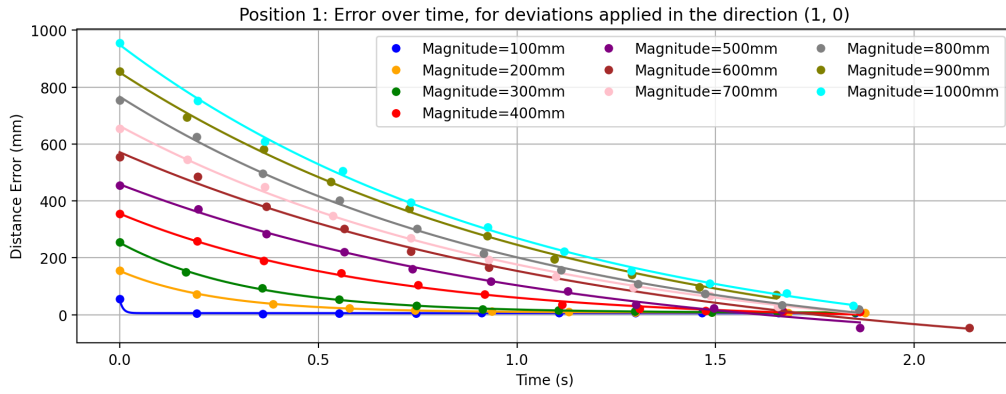
6.3. Results

6.3.1 Convergence test

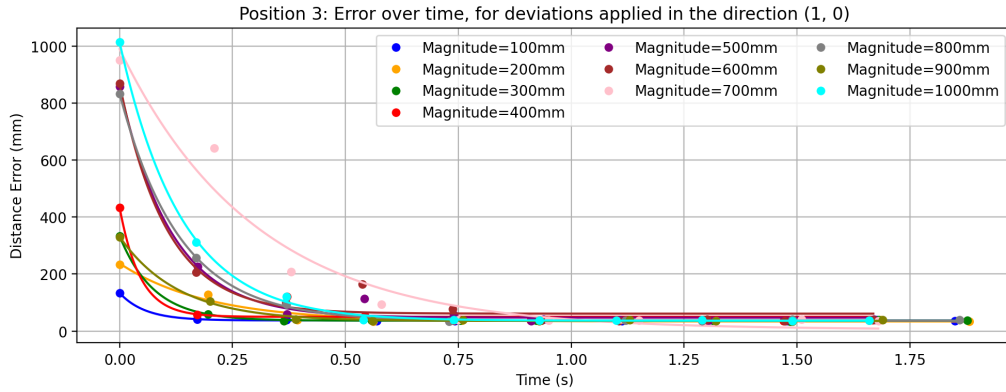
To analyze the poses recorded in this test, the error of the predicted pose was computed in each deviation direction over time, for each deviation magnitude. Then, for each decay, an exponential function was fitted:

$$e(t) = A \exp(-t/\tau) + C \quad (6.4)$$

Where A is the initial deviation error and τ is the convergence time constant, that indicates the time taken for the error to decay to $1/e$ of its initial value.



(a) Position 1.



(b) Position 3.

Figure 6.4: Exponential decay of the pose estimation error for different initial deviation magnitudes in the $(1,0)$ direction. The localization algorithm successfully converged for all tested deviation magnitudes at both positions, at different rates.

In each position tested, the direction and magnitude of the induced pose deviation define the convergence process. Figure 6.4 displays two ideal scenarios, where the pose always converges to the robot's true position, even with magnitudes of 1 meter. The curves with

the highest convergence times do not necessarily indicate a problem with the algorithm: in a real-world scenario the error would never accumulate to such extents, since the system continuously updates the pose.

In Figure 6.4, a clear comparison between the convergence in positions 1 and 3 is provided. Both cases present deviations in the same direction, where the pose estimate always converges to the true value. However, a significant difference is observed in the convergence time, with the position converging in significantly less time in position 3. What separates these positions is the number of walls observed: two walls in position 1, four walls in position 3. As a result, in position 3, each scan generates four EKF updates, which accelerates the convergence process.

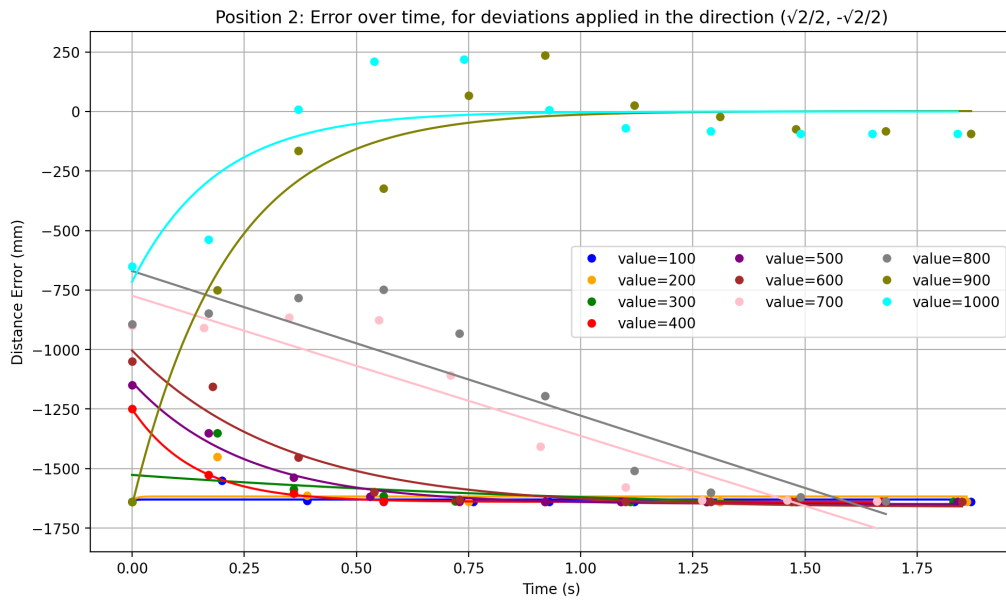


Figure 6.5: Exponential decay of the pose estimation error of the robot in Position 2, with varying deviation intensities in the $(\sqrt{2}/2, -\sqrt{2}/2)$ direction. At higher magnitudes, the pose estimation converged to the wrong pose, making the error stabilize at high values.

In contrast, in some testing conditions, as in Figure 6.5, the limitations of the localization algorithm can be observed. At 70 cm and 80 cm, the error follows an approximately parabolic curve: it increases slightly, before slowly converging to zero. At 90 cm and 100 cm, the estimation error converges exponentially, but stabilizes at the wrong position.

Beyond a certain magnitude, the localization algorithm has to rely solely on the "Lost" mode. Then, in certain deviation directions, high magnitudes will place the robot on the other side of the track. Because of the track's 180° symmetry, the algorithm converges

to the pose that is symmetrical to the true one, because it represents a locally lower-cost solution given the available wall observations and deviated pose.

Several convergence trajectories can be followed to observe this phenomenon.

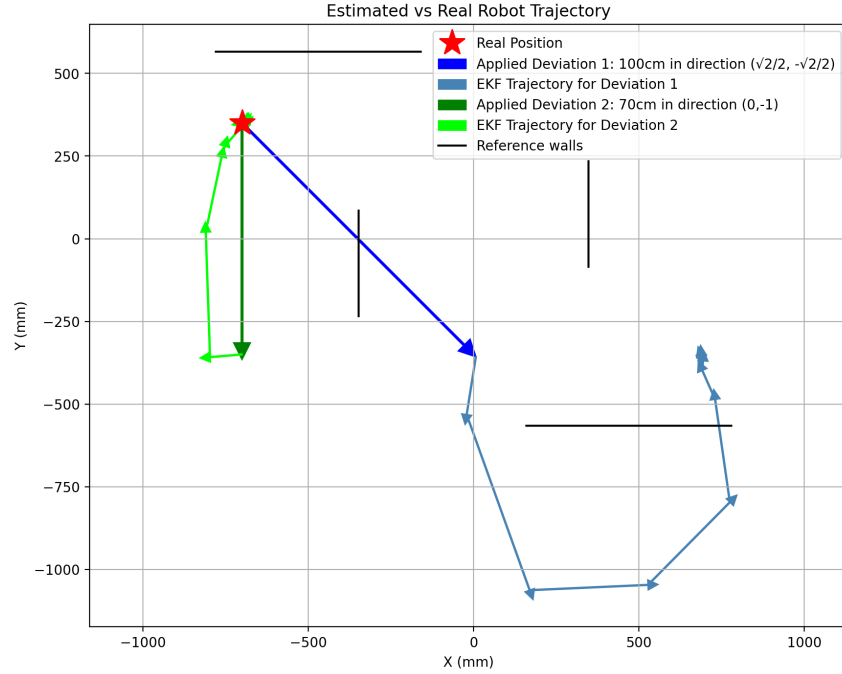


Figure 6.6: Trajectory described by consecutive EKF updates, as the robot tries to recover from the induced pose deviation. The robot is in Position 2.

All the convergence times and exponential amplitudes estimated from the fits performed are plotted in Figures A.1-???. Finally, to help in visualizing this complex test, polar plots were used in Figure 6.7.

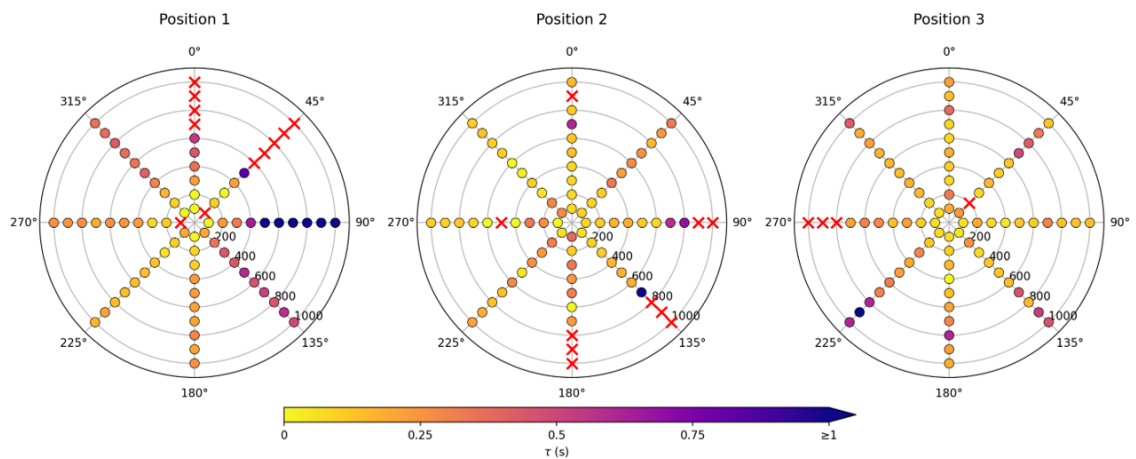


Figure 6.7: Polar plots representing the convergence time τ estimated by exponential fitting in each test run. In the polar plot, the angle indicates the direction in which the pose deviation was induced, with the radii indicating the magnitude of the deviation induced. The point's colors indicate the τ estimated, crosses mark where the EKF did not converge to the correct pose.

In this plot, it is clear the great majority of runs performed resulted in convergence with τ under 0.5 s. Most of the cases where convergence failed happened at the highest deviation magnitudes.

Overall, this test demonstrated the algorithm's capability to rapidly estimate and converge to the robot's true position. Most cases where the convergence fails are when the symmetry of the track causes the estimate to switch sides.

6.3.2 Covariance test

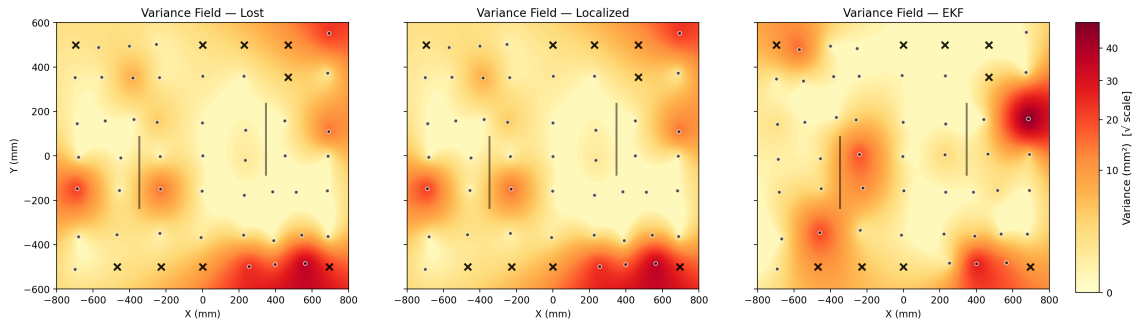


Figure 6.8: Variance fields recorded for each pose estimation routine. A square root scale was used to better visualize all plots. Data was recorded in the places marked with black dots, the heatmap was generated by performing linear interpolation.

This heatmap shows that the pose estimation variance is relatively low in most of the track, always below 50 mm^2 . The black crosses indicate places where no pose estimation was performed. These marks are almost symmetric, which is expected since the RAF4 track has 180° symmetry.

Furthermore, the heatmaps for the matching modes are identical, which is desired. After all, in each point, the robot can only observe the same walls. No covariance pattern seems evident and no heatmap displays significantly higher values, which indicates that the algorithm is properly implemented, providing stable estimations. The algorithm's behavior is further enhanced by the Delta-2A LiDAR's high precision.

6.3.3 Simple laps test

6.3.3.1 Trajectory Following

In practice, the robot was observed as following a clear, consistent path in the track. The four corners of the track are points with ArUco markers on the ground, which facilitated visual observation of the track following performance. Nevertheless, the data recorded

shows the robot's estimated trajectory. From its point of view, it followed the trajectory closely, with only slight lateral deviations, that were corrected before reaching a corner. The estimated trajectory during one lap is represented in Figure 6.9. Figure A.4 shows a representation of the ten laps overlapped.

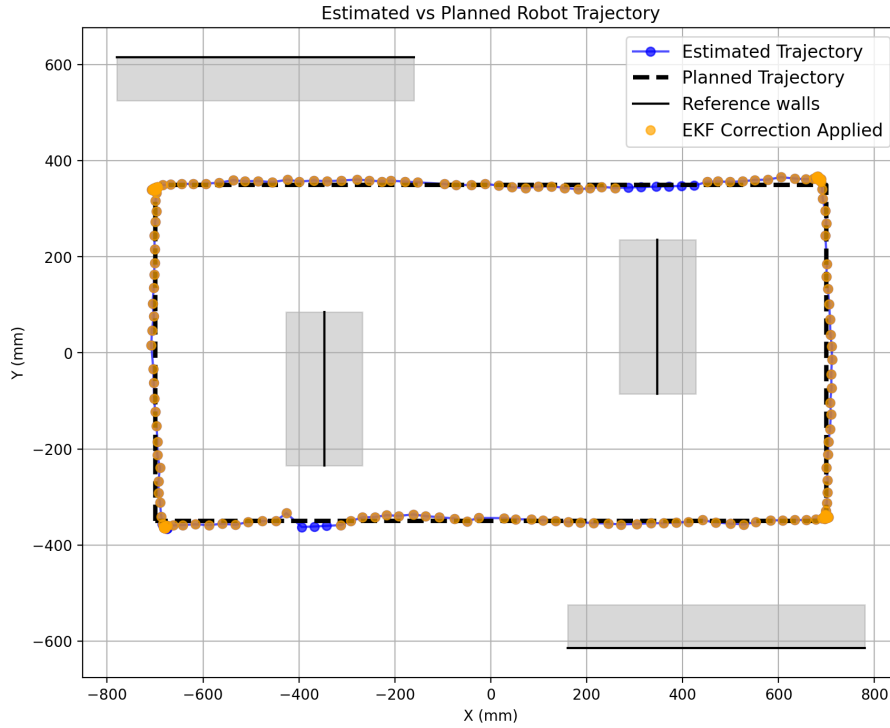


Figure 6.9: Poses estimated over the course of one simple lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.

In Figure 6.9, the blue points show where no EKF correction could be performed. In these regions, the machine above or below the robot partially obstructs the view of the warehouse in the opposite side of the track. As the machine is directly below or above the robot, it also cannot be detected in LiDAR data. That way, only one wall can be observed: the machine on the other side of the track. Physically, no EKF update can be performed in these regions using only this algorithm.

When a system relies on EKF for pose estimation and no observations are performed, the filter represents the lack of information as an increase in uncertainty. To verify that the EKF implemented behaves as expected, the estimated covariance matrix was recorded at each point along the trajectory.

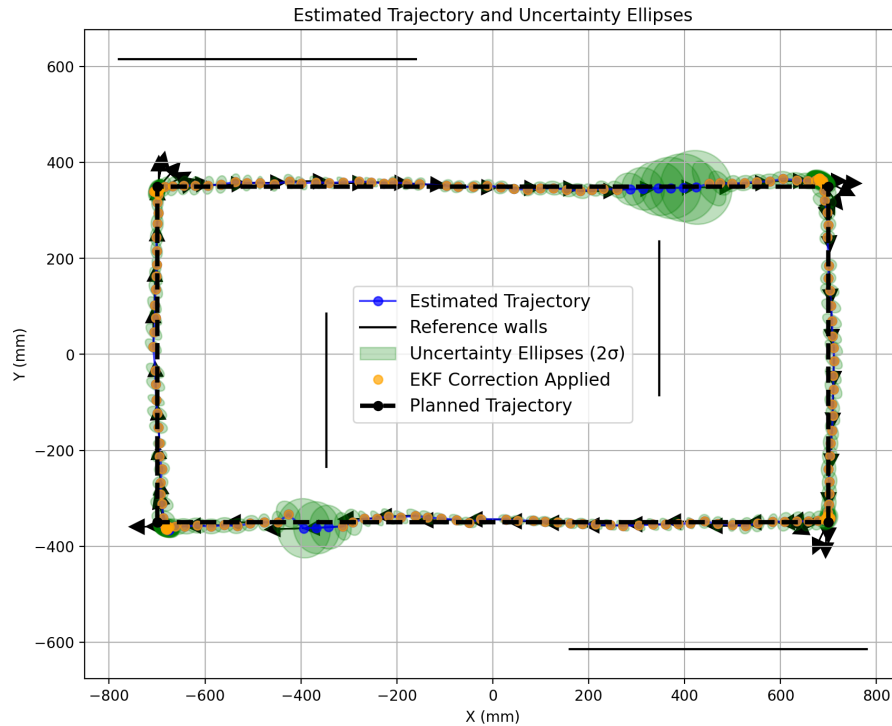


Figure 6.10: Estimated trajectory and uncertainty ellipses computed (95% confidence). The arrows indicate the direction of motion, that coincides with the increase of the ellipses' size.

These ellipses were computed using the eigenvalues and eigenvectors of the EKF estimated covariance matrix. As expected, the ellipses and the robot's pose uncertainty increase gradually when no updates are performed. When the robot exits a problematic region and an EKF update is carried out, the system instantly recovers and the uncertainty collapses.

6.3.3.2 Matching

The matching mode used in each execution of the algorithm is among the recorded data. Out of all the frames recorded over 10 laps, 95.99% of them used the "Localized" mode, with "Lost" only being used in 0.04%. The remaining frames could not perform EKF update, using the "invalid" mode. This significant difference does not necessarily indicate a problem with the cost tolerances in the "Localized" mode. Instead, these percentages indicate that, during every lap, the localization framework was able to estimate its pose with enough consistency and precision that the robot never got lost, so the efficient and simple "Localized" mode was enough.

To better evaluate the performance and uniformity of the "Localized" matching mode, Figure 6.11 shows where each wall was observed.

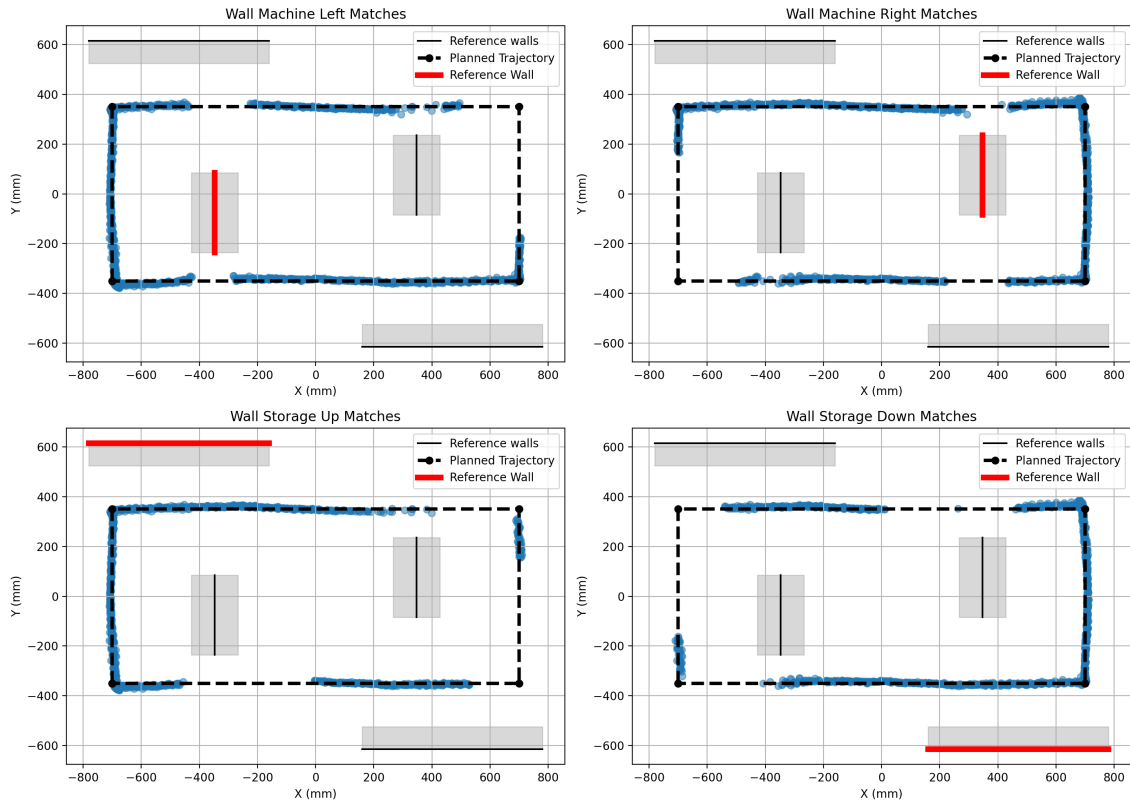


Figure 6.11: Plots showing the points (blue) where each wall (red) was observed and used in matching. This data was obtained by combining the data of the ten laps performed.

These plots show almost the exact theoretical distributions expected. Each wall is observed where it would be expected and the point maps of symmetrical walls are also symmetrical.

6.3.3.3 Execution time

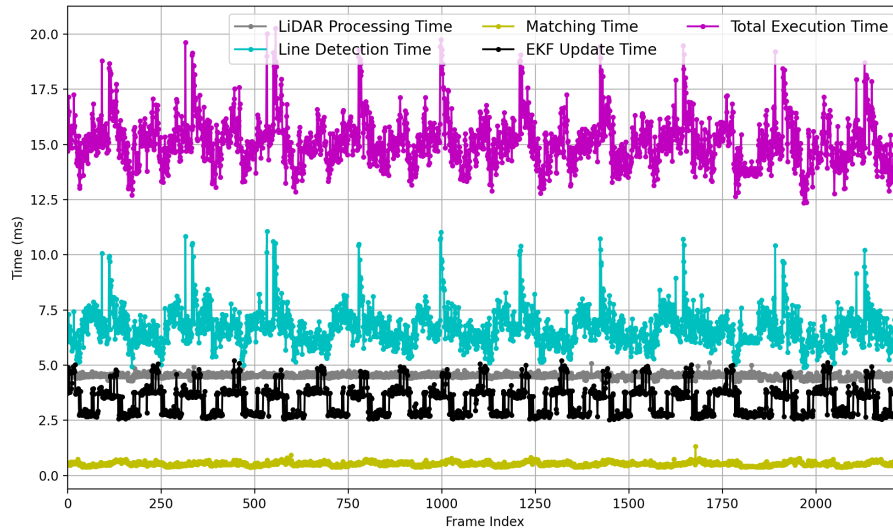


Figure 6.12: Execution time for each component of the localization algorithm, recorded over the duration of ten simple laps.

In the figure above, it immediately becomes clear the execution time follows the trajectory's periodicity. This is an expected behavior, as the same amount of walls and points should be observed in a certain point of the trajectory, in every lap.

The line detection algorithm, due to its many linearity computations performed for every point cluster detected, is the biggest component of the algorithm's execution time, at about 7 ms per scan. Then, the LiDAR processing and the EKF update both take around 4 ms to be executed. The LiDAR processing needs to perform various simple computations, while the EKF update has to perform matrix multiplication and inversion up to four times. Finally, the matching algorithm is the smallest portion of the execution time, at around 1 ms.

Overall, all the data analyzed in this test indicates that the localization algorithm is capable of commanding a robot through a simple, stable trajectory without ever losing itself. The full localization framework stack takes less than 20 ms to be executed.

6.3.4 Complex laps test

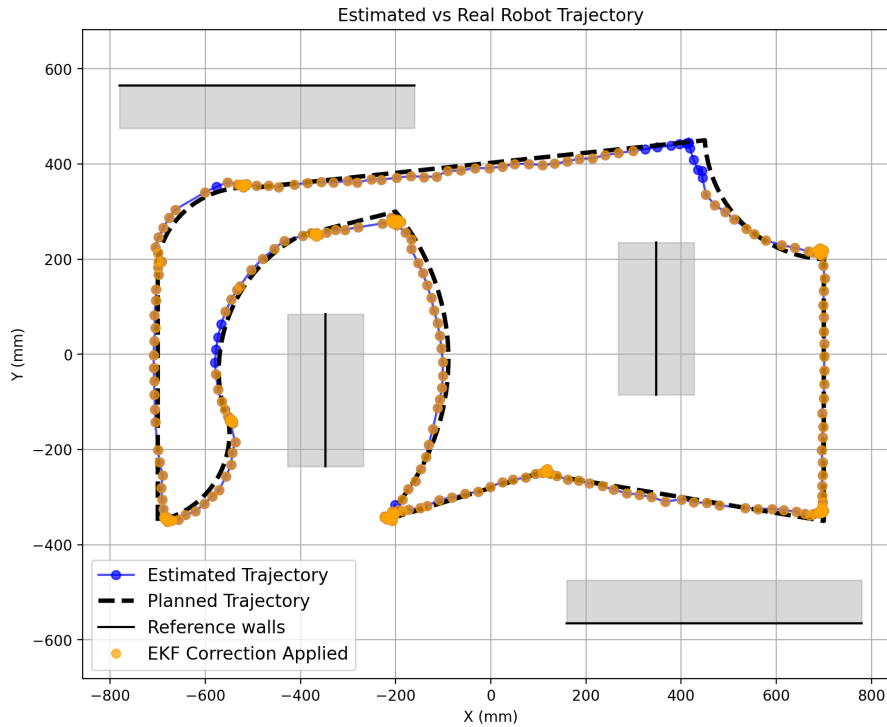


Figure 6.13: Poses estimated over the course of one complex lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.

Figure 6.13 indicates that all motion modes are properly implemented. The robot can successfully follow a 11-step trajectory without losing itself or accumulating drift. A map with all ten laps overlapped is provided in Figure A.5.

During this test, the matching modes used were recorded. The "Localized" mode was used in 91% of all frames, with "Lost" mode being utilized in zero frames. This result, once again, indicates that the robot can maintain elevated confidence levels, while staying close to a planned trajectory.

Following the analysis done for the simple trajectory, the estimated covariance matrix was recorded throughout this test. In Figure 6.14, the computed uncertainty ellipses are represented. In this trajectory, the uncertainty explodes in a point on the upper right part of the track. In this point, the robot stops, rotates itself and starts an arc segment down. While the robot is stopped or rotating in this region, the uncertainty is gradually increasing at every scan, resulting in very elevated uncertainty when leaving this point. However, the trajectory also shows that, even after achieving elevated uncertainty, the localization algorithm is able to recover and finish the trajectory steadily.

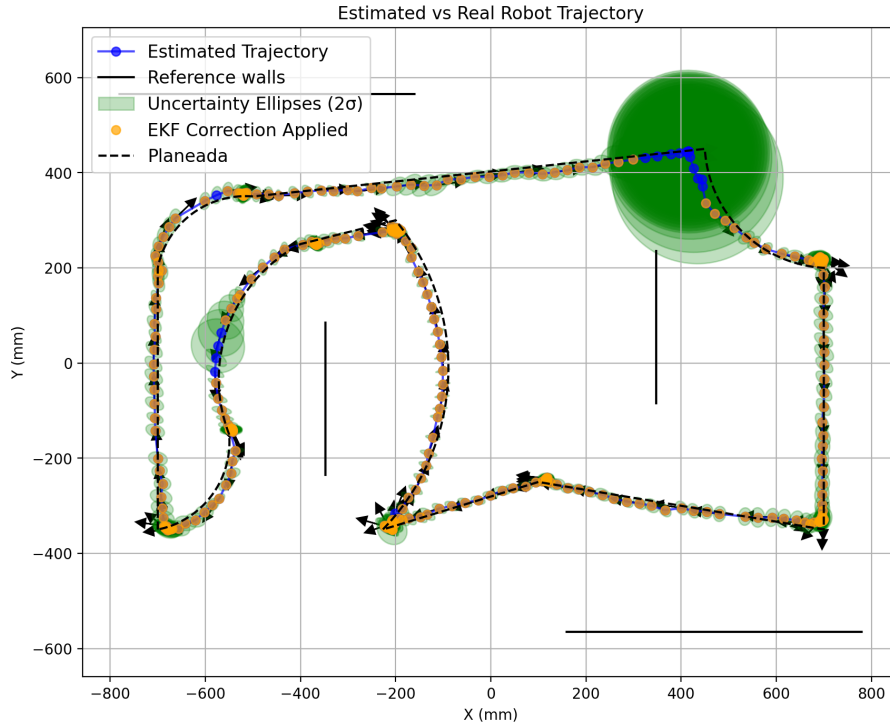


Figure 6.14: Estimated trajectory and uncertainty ellipses computed (95% confidence). The arrows indicate the direction of motion, that coincides with the increase of the ellipses' size.

For this test, the matching data will not be explored. The greatest objective of analysing that data in the simple track was to understand the track and its different zones of visibility. In this test, a very irregular and complex track will follow, so the matching data cannot easily provide useful information about the track or the robot's motion.

6.3.4.1 Execution time

Finally, following subsection 6.3.3, the execution time of each component of the localization framework was measured and plotted. Again, the periodicity of the laps followed is clearly evident. The execution time of each component stays roughly the same as in Figure 6.12, indicating the algorithm's robustness to the robot's motion.

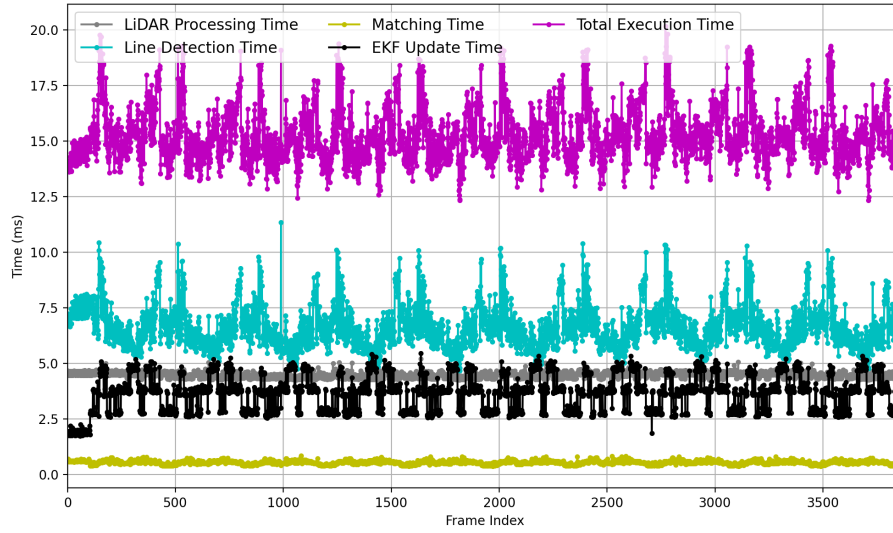


Figure 6.15: Execution time of each component of the localization framework, over the course of ten complex laps.

6.3.5 Kidnapped robot test

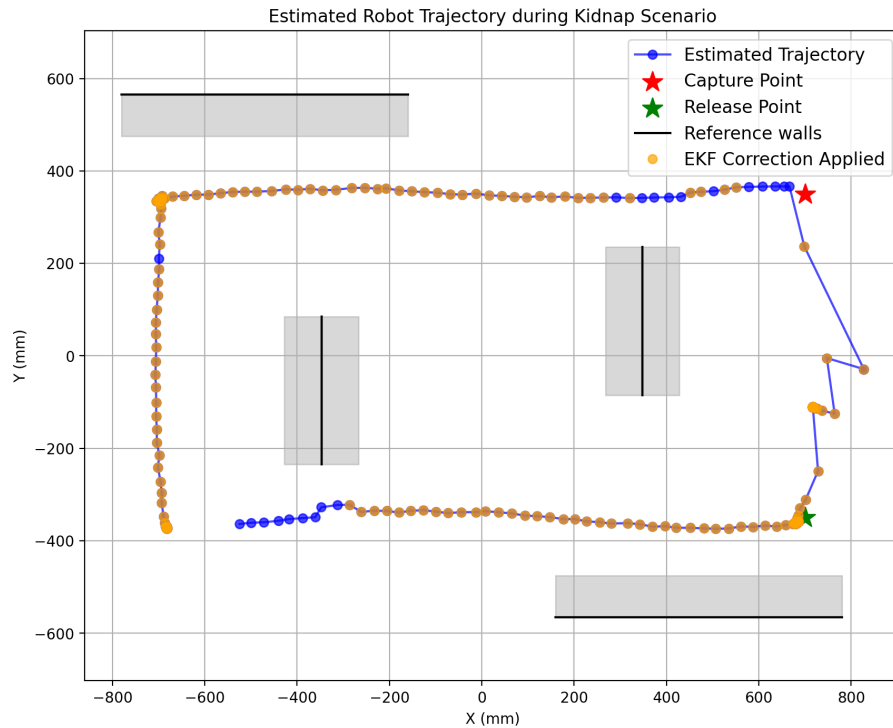


Figure 6.16: Trajectory estimated by the robot during the kidnapped robot test. The robot was picked up in the Capture point and quickly placed in the Release point. The trajectory is performed clockwise.

Figure 6.16 displays the robot's perceived trajectory during the kidnapped robot test. It was picked up before it completely stopped in the Capture point. When the robot was placed in the Release point, it was finishing its rotation and about to move forward and downwards. However, during the forced motion between the two points, no match was

performed and the EKF's uncertainty increased. Consequently, the Mahalanobis gates became progressively more permissive until, eventually the walls observed in the Release point could be matched. From that point, a sequence of aggressive position corrections was performed until the pose estimation converged to the correct values. The robot then correctly moved towards the left side of the track, following the planned trajectory.

Since no collision-avoidance mechanism is implemented, this test effectively evaluates the algorithm's ability to increase its permissibility and rapidly converge to an unknown pose. It demonstrates that the robot can accurately estimate its pose while in motion, recovering from a substantial localization error. Moreover, the fact that walls at the Release point were successfully matched shows that the robot correctly transitioned into the "Lost" matching mode when the "Localized" mode failed.

The outcome of this test would be similar if the robot was placed in the same region it was picked up from, regardless of any change applied to its heading. However, relocating it to the opposite side of the track could expose the system to the track's symmetry, causing the robot to converge to the pose symmetric to its true one and subsequently follow a mirrored trajectory. Naturally, this is something that should ever happen in a practical context.

7. Spatio-Temporal Registration

When a robot has a LiDAR mounted to it to perform ranging measurements while in motion, the motion of the sensor during each scan can cause distortion in the ranging data. For example, if the robot is moving in pure translation at 1 m/s and one LiDAR scan takes 200 ms, then the first and last measurements of the scan were taken at positions 20 cm apart. This phenomenon must be considered in the context of this work, as low-cost LiDAR sensors are used while the robot is in motion and rotating, but the localization algorithm always assumes that a given scan was performed entirely from the same position.

The process of countering this phenomenon is called Spatio-Temporal Registration. In general terms, position and velocity measurements are recorded throughout the course of each scan. Then, when processing the LiDAR data, the velocity data is integrated to estimate the robot's pose at the moment each point of the scan was measured. With this information, the complete scan can be transformed to represent a scan taken from a single point on the track.

This chapter, rather than implementing this feature, aims at determining if such functionality would be required or even beneficial to the system's performance.

7.1. Experimental Setup

Two tests were performed: one with the robot moving at various translational velocities, and another with the robot rotating in place at different angular velocities.

7.1.1 Translation Test

In this test, the robot was commanded to perform the "Do Laps" motion mode at different nominal linear velocities, ranging from 150 mm/s up to 350 mm/s. The pose estimation data was recorded during these laps, and the percentage of frames in which an EKF correction was performed was used as the success metric.

7.1.2 Rotation Test

The robot was placed in the three characteristic positions used in subsection 6.2.1. In each position, the robot was commanded to rotate in place for 150 LiDAR scans, at

angular velocities up to 3 rad/s. The percentage of frames with EKF corrections was again used as the success metric. Since the robot remained stationary in place throughout the test, this percentage should ideally stay constant. Any significant decrease in the percentage would indicate that the LiDAR data is being skewed by the rotational motion.

7.2. Results

7.2.1 Translation Test

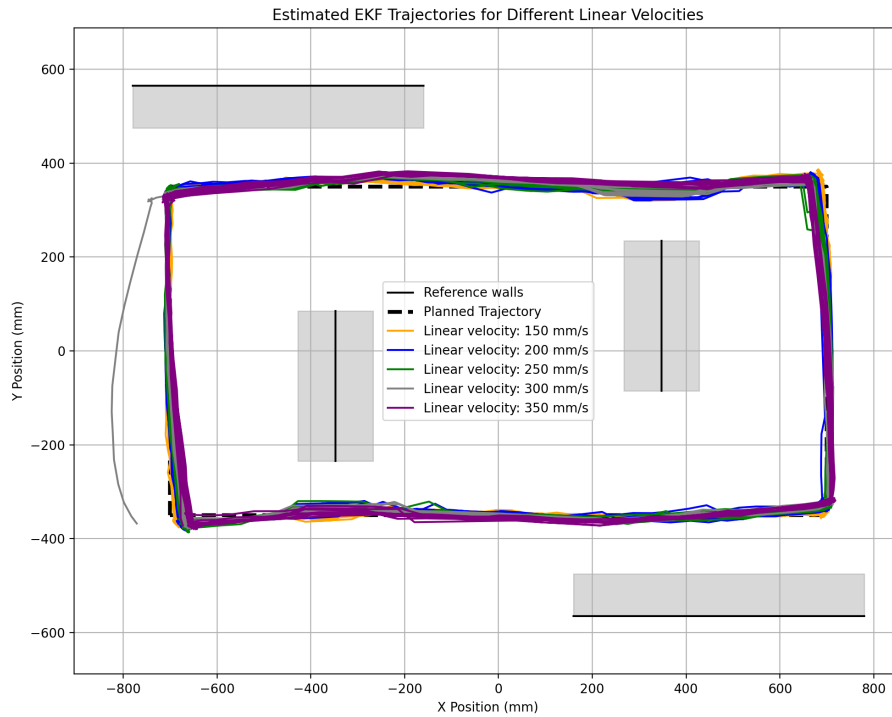


Figure 7.1: Trajectories estimated over the ten laps performed at each linear velocity tested. When starting the trajectory at 300 mm/s, the robot suffered a strong deviation, visible as the isolated grey line.

Figure 7.1 shows the poses estimated over the course of all the laps performed in this test. Notably, the trajectory remains consistent across all speeds tested. This indicates that the velocity robustness measures implemented were effective, with adequately chosen tolerances. Admittedly, the lateral correction gain could be further tuned to reduce the slight oscillations observed, which prevent the trajectories from being perfectly rectangular.

As shown in Figure 7.2, the correction percentages remain roughly constant at around 90% across all tested speeds. This indicates that, within the tested speed range, the LiDAR data is not noticeably affected by skewing. Although skewing is expected to become more significant at higher speeds, further testing was not possible due to a limitation of the robotic platform. At higher commanded speeds, the large torque applied to the wheels

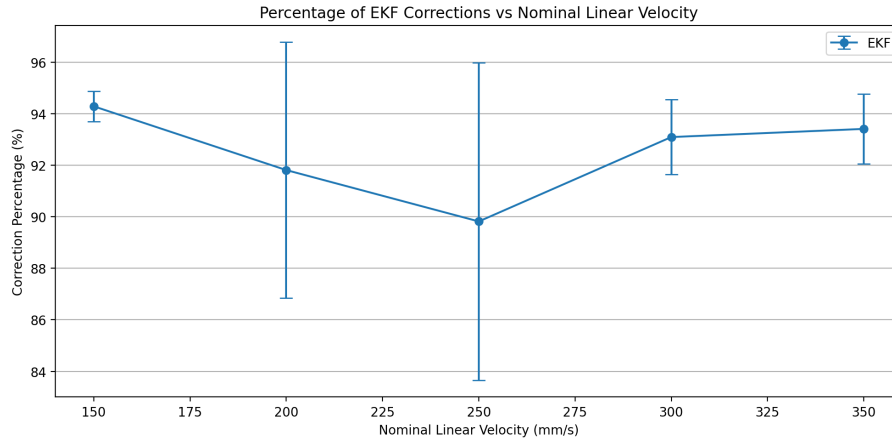


Figure 7.2: Percentage of frames with EKF corrections during the laps performed at each velocity. Only scans taken while the robot was in motion were considered, in order to properly isolate the skewing phenomenon.

during the initial acceleration caused the front of the robot to lift off the ground, compromising its stability and repeatability. While this behavior could likely be mitigated by introducing a smoother acceleration profile, the motion controller used for these experiments intentionally employed an instantaneous speed command, and therefore no higher-speed tests were conducted.

7.2.2 Rotation Test

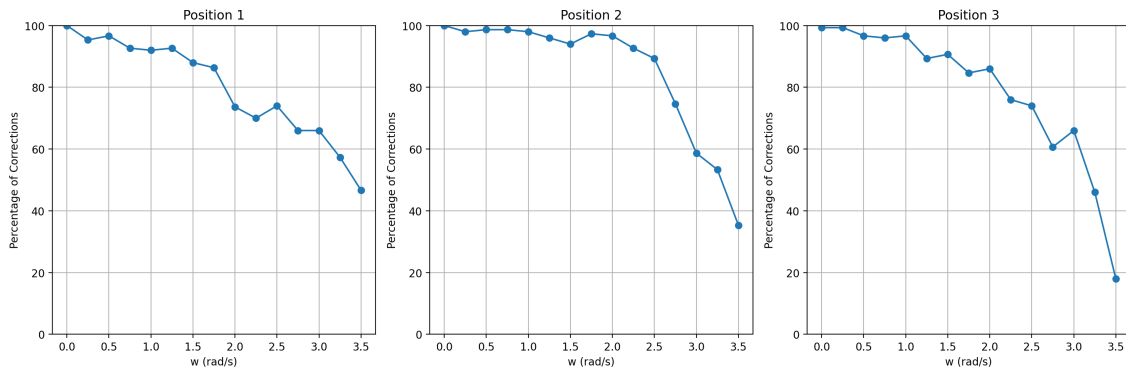


Figure 7.3: Percentage of frames in which EKF updates were performed, for each of the three positions tested, as a function of angular velocity.

This test demonstrated that rotational motion is considerably more problematic for LiDAR skewing than translational motion. Angular velocities of only 3 rad/s were enough to reduce the correction percentage to as low as 60% in some cases. It should be noted that all curves start at 100%, since the three positions chosen allow detection of two or more walls simultaneously. As the robot rotates faster, these walls appear increasingly distorted in the scan, eventually exceeding the tolerances of the line detection algorithm

and causing the localization update to fail. Nevertheless, the correction percentage never dropped to zero, with only one of the three positions falling below 20%.

In conclusion, both tests indicate that, under the range of motion conditions typically experienced on the RAF4 track, no spatio-temporal registration algorithm is required. The Delta-2A's high precision and angular resolution provide sufficient scan quality even at the higher translational and rotational velocities tested.

8. Conclusion

This dissertation set out to develop a lightweight, computationally efficient localization framework for low-cost mobile robots operating in structured indoor environments, with the RAF4 competition track serving as a representative case study.

The LiDAR characterization performed in this work provided a detailed comparison between two low-cost sensors, the HLS-LFCD2 and the Delta-2A. The results showed that the Delta-2A consistently offers superior distance accuracy, precision and angular resolution, at the cost of a more complex communication protocol.

The proposed line detection algorithm demonstrated that a simple clustering and linearity-based approach is sufficient to reliably extract linear wall features from LiDAR point clouds, even in the presence of noise, partial wall observations and non-linear clusters. The additional local linearity analysis performed on non-validated clusters further improved robustness in cases involving corners or wall boundaries, without introducing significant computational cost.

The localization algorithm itself, combining a length-based filtering stage, two matching strategies and a Mahalanobis-gated EKF update, was shown to be efficient and robust. The "Localized" mode allowed for fast pose tracking under normal operating conditions, while the "Lost" mode enabled recovery from large pose errors and even a simulated kidnapped robot scenario, without requiring manual re-initialization. Importantly, the complete localization pipeline, comprising LiDAR data processing, line detection, wall matching and the EKF update, executes in under 20 ms per scan. As a LiDAR scan takes around 200 ms and a control loop is performed every 40 ms, these execution times are completely within the real-time requirements of the application tested.

The experimental evaluation included convergence, covariance, trajectory-following and kidnapped robot tests, that consistently confirmed the stability and robustness of the proposed algorithm. The system was shown to converge rapidly to the correct pose from a wide range of induced deviations, to maintain low estimation uncertainty throughout typical operating trajectories and to recover its pose estimate after abrupt relocation. The only systematic failure identified, convergence to the pose symmetric to the true one, is a direct and expected consequence of the track's inherent 180°geometric symmetry. Finally, the spatio-temporal registration study demonstrated that, within the velocity ranges

achievable by the robotic platform used, motion-induced distortion of the LiDAR scans does not meaningfully degrade localization performance and no spatio-temporal registration algorithm is required for this application.

8.1. Limitations and Future Work

Despite these results, some limitations of the current implementation should be acknowledged. The reliance on a structured environment with known, symmetric linear features means that the algorithm cannot be directly applied to unmapped environments. This is not a shortcoming of the algorithm, having a known map to exploit was necessary to achieve high computational efficiency and simplicity. Extending the framework with SLAM capabilities would widen its applicability, but would depart from the simplistic, low-cost philosophy of this work.

Furthermore, the higher-velocity regime of the translation spatio-temporal registration test could not be fully explored due to mechanical limitations of the testing platform, but a more capable motion controller with a smoother acceleration profile would enable this boundary to be better tested.

Future work could also explore the use of other low-cost sensors, such as inertial measurement units, to further improve the robustness of the odometry-based prediction step of the EKF, particularly when no LiDAR corrections can be performed. Finally, given the framework's computational efficiency and reliance on inexpensive hardware, future work could formally evaluate this system as an educational tool, determining its effectiveness in teaching core robotics concepts such as sensor characterization, feature extraction and probabilistic state estimation.

Overall, this dissertation demonstrates that reliable, real-time localization in structured environments does not require expensive sensors or computationally intensive algorithms. By combining sensor characterization, an efficient feature extraction pipeline and a robust matching strategy, the proposed framework achieves accurate and resilient pose estimation on a low-cost robotic platform, fulfilling the objectives set out at the beginning of this work.

References

- [1] P. Costa, “Robot at factory github repository,” <https://github.com/P33a/RobotAtFactory>, last accessed: 2026-04-27. [Cited on pages 1 and 2.]
- [2] M. O. A. Aqel, M. H. Marhaban, and M. I. Saripan, “Review of visual odometry: types, approaches, challenges, and applications,” *SpringerPlus*, vol. 5, 2016. [Online]. Available: <https://link.springer.com/article/10.1186/s40064-016-3573-7> [Cited on page 5.]
- [3] Y. Liu, S. Wang, Y. Xie, T. Xiong, and M. Wu, “A review of sensing technologies for indoor autonomous mobile robots,” *Sensors*, vol. 24, no. 4, 2024. [Online]. Available: <https://www.mdpi.com/1424-8220/24/4/1222> [Cited on page 5.]
- [4] S. Thrun, *Simultaneous Localization and Mapping*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 13–41. [Online]. Available: https://doi.org/10.1007/978-3-540-75388-9_3 [Cited on page 6.]
- [5] X. S. Le, L. Fabresse, N. Bouraqadi, and G. Lozenguez, “Evaluation of out-of-the-box ros 2d slams for autonomous exploration of unknown indoor environments,” in *Intelligent Robotics and Applications*, Z. Chen, A. Mendes, Y. Yan, and S. Chen, Eds. Cham: Springer International Publishing, 2018, pp. 283–296. [Cited on page 6.]
- [6] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf, “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. [Cited on page 6.]
- [7] S. Kohlbrecher, J. Meyer, T. Graber, K. Petersen, U. Klingauf, and O. von Stryk, “Hector open source modules for autonomous mapping and navigation with rescue robots,” in *RoboCup 2013: Robot World Cup XVII*, S. Behnke, M. Veloso, A. Visser, and R. Xiong, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 624–631.
- [8] S. Nagla, “2d hector slam of indoor mobile robot using 2d lidar,” in *2020 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS)*, 2020, pp. 1–4. [Cited on page 6.]

- [9] W. Xie, T. Ren, C. J. Xue, J.-M. Wu, and N. Guan, "Lidar localization using position-encoded landmarks without point cloud maps," *Journal of Systems Architecture*, vol. 168, p. 103556, 2025. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1383762125002280> [Cited on page 7.]
- [10] W. Ying and S. Sun, "An improved monte carlo localization using optimized iterative closest point for mobile robots," *Cognitive Computation and Systems*, vol. 4, no. 1, pp. 20–30, 2022. [Online]. Available: <https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/ccs2.12040> [Cited on page 7.]
- [11] A. Filotheou, "Correspondenceless scan-to-map-scan matching of homororiented 2d scans for mobile robot localisation," *Robotics and Autonomous Systems*, vol. 149, p. 103957, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0921889021002323> [Cited on page 7.]
- [12] R. W. M. Hendrikx, H. Bruyninckx, J. Elfring, and M. J. G. Van De Molengraft, "Local-to-global hypotheses for robust robot localization," *Frontiers in Robotics and AI*, vol. Volume 9 - 2022, 2022. [Online]. Available: <https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2022.887261> [Cited on page 8.]
- [13] S. Wang, X. Chen, G. Ding, Y. Li, W. Xu, Q. Zhao, Y. Gong, and Q. Song, "A lightweight localization strategy for lidar-guided autonomous robots with artificial landmarks," *Sensors*, vol. 21, no. 13, 2021. [Online]. Available: <https://www.mdpi.com/1424-8220/21/13/4479> [Cited on page 8.]
- [14] J. Speyer and W. Chung, "Stochastic processes, estimation, and control," 01 2008. [Cited on page 8.]
- [15] S. Julier and J. Uhlmann, "Unscented filtering and nonlinear estimation," *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401–422, 2004. [Cited on page 9.]
- [16] P. C. Mahalanobis, "On the generalized distance in statistics," *Proceedings of the National Institute of Sciences of India*, vol. 2, no. 1, pp. 49–55, 1936. [Cited on page 9.]
- [17] Y. Bar-Shalom, X. R. Li, and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation*. New York: John Wiley & Sons, 2001. [Cited on page 9.]

- [18] V. Nguyen *et al.*, “A comparison of line extraction algorithms using 2d laser rangefinder for indoor mobile robotics,” in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 1929–1934. [Cited on pages 9 and 10.]
- [19] DFRobot, “Tt motor with encoder (6v 160rpm 120:1),” <https://wiki.dfrobot.com/fit0450/>, last accessed: 2026-06-28. [Cited on page 11.]
- [20] P. Costa, “Comrobot - pico demo, gchannels code,” https://github.com/P33a/ComRobot/blob/master/pico_demo/src/gchannels.cpp, last accessed: 2026-06-20. [Cited on pages 16 and 17.]
- [21] “Smhasher,” <https://github.com/aappleby/smhasher/blob/master/src/Hashes.cpp>, last accessed: 2026-06-04. [Cited on page 19.]
- [22] P. Marques, P. Pereira, and P. Costa, “Sub-step quadrature encoder,” <https://github.com/pmarques-dev/PicoEncoder>, 2022, last accessed: 2026-06-20. [Cited on pages 21 and 22.]
- [23] P. Costa, “Pico4drive arduino library,” <https://github.com/P33a/Pico4DriveLib>, last accessed: 2026-06-18. [Cited on page 24.]
- [24] botnroll. Raspberry pi pico w. Price: EUR 6.42 per unit, accessed August 1, 2026. [Online]. Available: <https://www.botnroll.com/en/controller-boards/4466-raspberry-pi-pico-w.html> [Cited on page 44.]
- [25] ——. Pico4drive - placa de desenvolvimento para pi pico. Price: EUR 19.90 per unit, accessed August 1, 2026. [Online]. Available: <https://www.botnroll.com/pt/shields-hats/4722-pico4drive-placa-de-desenvolvimento-para-pi-pico.html> [Cited on page 44.]
- [26] ——. Micro motor dc com encoder e redutora 120:1 - 6v 160rpm. Price: EUR 11.50 per unit, accessed August 1, 2026. [Online]. Available: <https://www.botnroll.com/pt/motores-dc/1645-micro-motor-dc-com-encoder-e-redutora-1201-6v-160rpm.html> [Cited on page 44.]
- [27] AliExpress. Hls-lfcd2 aspirador de pó sensor distância a laser (...). Price: EUR 30.93 per unit, accessed August 1, 2026. [Online]. Available: <https://pt.aliexpress.com/item/1005003909442436.html?gatewayAdapt=glo2bra> [Cited on page 44.]

- [28] Makers Pet. 3irobotix delta-2a lidar distance sensor (used/refurb). Price: EUR 25.99 per unit, accessed August 1, 2026. [Online]. Available: <https://makerspet.com/product/3irobotix-delta-2a-lidar-distance-sensor-used-refurb/> [Cited on page 44.]
- [29] botnroll. Pilha li-ion 18650 3,7v 2600mah. Price: EUR 4.60 per unit, accessed August 1, 2026. [Online]. Available: <https://www.botnroll.com/pt/18650/2300-pilha-li-ion-mr18650-37v-2550mah.html> [Cited on page 44.]
- [30] ——. Suporte para 2 pilhas 18650 p/ pcb. Price: EUR 1.90 per unit, accessed August 1, 2026. [Online]. Available: <https://www.botnroll.com/pt/18650/5399-suporte-para-2-pilhas-18650-p-pcb.html> [Cited on page 44.]
- [31] H-L Data Storage, “Lds1.5 h-l data storage (model: Hls-lfcd2) technical datasheet,” https://emanual.robotis.com/assets/docs/LDS_Basic_Specification.pdf, 2017, last accessed: 2026-04-27. [Cited on pages 46 and 48.]
- [32] Shenzhen Shanchuan Robot Co., Ltd., *Delta-2A Communication Protocol (Delta-2A □□□□)*, Shenzhen Shanchuan Robot Co., Ltd., Jan. 2019, communication protocol manual for the Delta-2A LiDAR. Original document in Chinese. [Cited on page 48.]
- [33] M. Weinmann *et al.*, “Semantic 3d scene interpretation: A framework combining optimal neighborhood size selection with relevant features,” *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. II-3, pp. 181–188, 2014. [Cited on page 62.]

A. Appendix

A.1. Algorithm Pseudocodes

A.1.1 LiDAR data buffering

Require: Incoming byte b , lap buffer object lb

Ensure: Received packets accumulated in $lb.buf$; $lb.lap_complete$ set when a full rotation is detected

```

1: if  $lb.buf$  was just cleared and no packet is being assembled then
2:   Reset angle tracking:  $last\_angle \leftarrow -1$ 
3: end if
4: if no sync byte found yet then
5:   if  $b$  matches the sensor-specific sync byte then
6:     Store  $b$  as first byte of  $pkt\_buf$ ; begin packet accumulation
7:   end if
8:   return ▷ discard all bytes until sync
9: end if
10: Append  $b$  to  $pkt\_buf$ 
11: if sensor is Delta-2A and at least 3 bytes have been received then
12:   Extract packet length from bytes 1–2 of  $pkt\_buf$ 
13:   if length is outside valid range  $[8, 200]$  bytes then
14:     Discard packet; reset parser
15:   return
16: end if
17: else if sensor is HLS-LFCD2 then
18:   Set expected packet length to 42 bytes
19: end if
20: if expected packet length is known and  $pkt\_buf$  has  $pkt\_len$  bytes then
21:   Extract start angle  $\theta_0$  from packet header
22:   if  $\theta_0$  is invalid then
23:     Discard packet; reset parser ▷ prevents false sync
24:   return
25: end if
26: if  $last\_angle \geq 0$  and  $last\_angle - \theta_0 > 20^\circ$  then ▷ wrap-around
27:   Append  $pkt\_buf$  to  $lb.buf$ 

```

```

28:     lb.lap_complete ← true
29:     last_angle ←  $\theta_0$                                 ▷ next lap starts from this angle
30:     Reset parser
31:     return
32: end if
33: last_angle ←  $\theta_0$ 
34: if lb.buf has sufficient space then
35:     Append pkt_buf to lb.buf
36: else
37:     Discard packet; reset angle tracking                ▷ prevent buffer overflow
38: end if
39: Reset parser for next packet
40: end if

```

A.1.2 LiDAR data processing

Require: Raw lap bytes `lap.bytes`, number of packets `num_packets`, sensor type `lidar_type`

Ensure: Arrays `lap.x`, `lap.y` of valid Cartesian points; detected lines in `lines`

```

1: Clear all lap data arrays and line results
2:  $t_{\text{start}} \leftarrow \text{micros}()$ 
3: if sensor is HLS-LFCD2 then
4:     for each packet  $i \in \{0, \dots, \text{num\_packets} - 1\}$  do
5:          $\text{pkt} \leftarrow \text{lap.bytes}[i \times \text{LIDAR\_MAX\_PKT\_BYTES}]$ 
6:         if not check_sum_hls(pkt) then
7:             Mark all 6 points of packet  $i$  as invalid
8:             continue
9:         end if
10:        for each measurement  $j \in \{0, \dots, 5\}$  do
11:             $\theta \leftarrow (\text{pkt}[1] - 0xA0) \times 6 + j$ 
12:            if  $\theta \notin [0, 360)$  then
13:                Mark point as invalid; continue
14:            end if
15:            Extract intensity and distance  $d$  from packet bytes
16:            Apply distance calibration
17:             $(x, y) \leftarrow d \cdot (\cos \theta, -\sin \theta)$                 ▷  $y$  negated: LiDAR rotates clockwise
18:            if  $d < 0.02 \text{ m}$  or  $d > 3.00 \text{ m}$  or  $(x, y) \approx (0, 0)$  then

```

```

19:         Mark point as invalid
20:     else
21:         Mark point as valid
22:     end if
23: end for
24: end for
25: else if sensor is Delta-2A then
26:      $i \leftarrow 0$ ; point_counter  $\leftarrow 0$ 
27:     while  $i < \text{total\_bytes}$  do
28:         if lap.bytes[ $i$ ]  $\neq 0xAA$  then
29:              $i += 1$ ; continue
30:         end if
31:         Extract packet length pkt_len from bytes  $i+1, i+2$ 
32:         if pkt_len outside valid range or packet exceeds buffer bounds then
33:              $i += 1$ ; continue
34:         end if
35:         if not check_sum_delta(pkt, pkt_len) then
36:              $i += \text{pkt\_len} + 2$ ; continue  $\triangleright$  pkt_len does not count the 2 checksum
            bytes
37:         end if
38:         Extract start angle  $\theta_0$  from bytes 11–12
39:         Apply angular offset correction
40:          $N \leftarrow (\text{pkt\_len} - 13) / 3$ 
41:         if  $N \leq 0$  or  $N > 100$  then
42:              $i += 1$ ; continue
43:         end if
44:         Increment total angle span by  $22.5^\circ$ 
45:         Initialize packet valid point count
46:         for each measurement  $j \in \{0, \dots, N - 1\}$  do
47:              $\theta \leftarrow \theta_0 + j \times 22.5^\circ / N$ 
48:             Extract signal quality and distance  $d$  from 3-byte measurement block
49:              $(x, y) \leftarrow d \cdot (\sin \theta, \cos \theta)$   $\triangleright$  swap coordinate axis
50:             if  $d < 0.02 \text{ m}$  or  $d > 3.00 \text{ m}$  or  $(x, y) \approx (0, 0)$  then
51:                 Mark point as invalid

```

```

52:         else
53:             Mark point as valid; point_counter += 1
54:             Increment packet valid point count
55:         end if
56:     end for
57:     Increment total valid point with packet valid point count
58:      $i += \text{pkt\_len} + 2$ 
59: end while
60: Compute effective angular resolution = total angle span / total valid points
61: end if
62: Collect all valid points into lap.x, lap.y
63: process_time ← micros() - t_start
64: find_lines(lap.x, lap.y, lines)
65: lines_time ← micros() - t_start

```

A.1.3 Line detection

Require: *LidarData*, *TolClusters*, *MinPoints*, *LinearityThreshold*, *CornerLinearityThreshold*

Ensure: *Centroids*, *Lengths*, *Orientations*

```

1: Initialize empty arrays Centroids, Lengths, Orientations
2: for each pair of consecutive points  $(i, i + 1)$  in LidarData do
3:      $d \leftarrow$  mean distance to robot
4:      $C_{th} \leftarrow$  cluster threshold using  $d$ , TolClusters
5:     if  $\text{dist}(i, i + 1) > C_{th}$  then
6:         close cluster if  $\text{size} \geq \text{MinPoints}$ 
7:     else
8:         add point to cluster or start new cluster
9:     end if
10: end for
11: for each cluster do
12:     compute centroid,  $S_{xx}$ ,  $S_{yy}$ ,  $S_{xy}$  and linearity
13:     if linearity > LinearityThreshold then
14:         project first and last points onto 2D PCA principal direction
15:         compute center, length, orientation with projected points
16:         store results
17:     else

```

```

18:     compute local linearity in each point, using 2N points neighborhood
19:     find point with minimum local linearity
20:     if minimum linearity  $\leq$  CornerLinearityThreshold then
21:         split cluster into  $Sub_1, Sub_2$  along minimum linearity point
22:         compute linearity of each subcluster
23:         if linearity of Subcluster  $>$  LinearityThreshold then
24:             project first and last points onto 2D PCA principal direction
25:             compute center, length, orientation using projected points
26:             store results
27:         end if
28:     end if
29: end if
30: end for
31: return Centroids, Lengths, Orientations

```

A.1.4 Localization: Localized mode

Require: Estimated pose (x, y, θ) , classified wall types, detected LiDAR lines, reference

walls $r \in \{\text{mLEFT}, \text{mRIGHT}, \text{sUP}, \text{sDOWN}\}$, MAX_CENTER_DIST, MAX_COST_COMBINED

Ensure: Matched wall indices *ind_matches*, count of valid matches

```

1: count_valid_matches  $\leftarrow$  0
2: for each reference wall  $r$  do
3:     min_cost  $\leftarrow$   $\infty$ ; best_match  $\leftarrow$  -1
4:     for each detected wall  $i$  do
5:         if type of wall  $i \neq$  type of reference wall  $r$  then
6:             reject; continue
7:         end if
8:         compute expected orientation = ref. angle $r$  -  $\theta$ 
9:         compute orientation cross product cross = sin(measured - expected)
10:        if |cross|  $>$  PROD_TOL then
11:            reject; continue
12:        end if
13:        convert detected wall center to global frame using current pose  $(x, y, \theta)$ 
14:        compute center distance pos_dist between global detected center and refer-
            ence wall center
15:        if pos_dist  $>$  MAX_CENTER_DIST then

```

```

16:         reject; continue
17:     end if
18:     compute cost =  $w_d \cdot \text{pos\_dist} + w_a \cdot |\text{cross}|$ 
19:     if cost < min_cost then
20:         min_cost  $\leftarrow$  cost; best_match  $\leftarrow i$ 
21:     end if
22: end for
23: if min_cost < MAX_COST_COMBINED then
24:     mark wall best_match as matched  $\triangleright$  prevents reuse by other reference walls
25:     ind_matches[r]  $\leftarrow$  best_match
26:     count_valid_matches += 1
27: end if
28: end for
29: if count_valid_matches < 2 then
30:     reset all entries of ind_matches to -1
31: end if
32: return ind_matches

```

A.1.5 Localization: Lost mode

Require: Previous pose (x, y) , classified wall types, detected LiDAR lines, reference wall map

Ensure: Matched wall indices ind_matches

```

1: if number of valid detected walls < 2 then
2:     return
3: end if
4: for each pair of valid detected walls  $(i, j)$ ,  $i \neq j$  do
5:     compute center distance  $d_{ij}$  between walls  $i$  and  $j$ 
6:     compute orientation cross product cross =  $|\sin(\alpha_i - \alpha_j)|$  with the orientations
       normalized to  $[-\pi, \pi]$ 
7:     compute perpendicular distances  $d_{\perp,i}$ ,  $d_{\perp,j}$  from the robot to each wall
8:     if walls have different types and cross  $\approx$  CROSS_OPP then  $\triangleright$  orthogonal pair:
       one machine, one warehouse
9:         assign  $d_{\text{machine}}$ ,  $d_{\text{warehouse}}$  from  $d_{\perp,i}$ ,  $d_{\perp,j}$  according to wall type
10:        compute candidate machine positions  $x_{mL}$ ,  $x_{mR}$  using  $d_{\text{machine}}$  and the sign of
        $(x_{\text{robot}} - x_{\text{ref}})$ 

```

```

11:     compute candidate warehouse positions  $y_{sU}$ ,  $y_{sD}$  using  $d_{\text{warehouse}}$ 
12:     if  $d_{ij} \approx D\_MACH\_WARE\_SAME$  then
13:         compute distance from previous pose to candidate  $(x_{mL}, y_{sU})$  and to  $(x_{mR}, y_{sD})$ 
14:         select the closer candidate; assign matches to mLEFT/sUP or mRIGHT/sDOWN
        accordingly
15:     else if  $d_{ij} \approx D\_MACH\_WARE\_OPP$  then
16:         compute distance from previous pose to candidate  $(x_{mL}, y_{sD})$  and to  $(x_{mR}, y_{sU})$ 
17:         select the closer candidate; assign matches to mLEFT/sDOWN or mRIGHT/sUP
        accordingly
18:     end if
19:     else if walls have the same type and cross  $\approx$  CROSS_SAME then      ▷ parallel
        pair: two machines or two warehouses
20:     if  $d_{ij} \approx D\_MACH\_MACH$  then
21:         compute robot's distance to reference mLEFT and mRIGHT walls
22:         compute cost of assigning  $(i, j) \rightarrow (mLEFT, mRIGHT)$  and  $(j, i) \rightarrow (mLEFT,$ 
        mRIGHT), using absolute perpendicular-distance differences
23:         select minimum-cost assignment
24:     else if  $d_{ij} \approx D\_STOR\_STOR$  then
25:         compute robot's distance to reference sUP and sDOWN walls
26:         compute cost of assigning  $(i, j) \rightarrow (sUP, sDOWN)$  and  $(j, i) \rightarrow (sUP, sDOWN),$ 
        using absolute perpendicular-distance differences
27:         select minimum-cost assignment
28:     end if
29: end if
30: end for
31: return ind_matches

```

A.1.6 Complete Localization algorithm

Require: Current pose (x, y, θ) and covariance P , robot velocities (v, w) , timestep Δt ,
reference walls, LiDAR line data

Ensure: Updated robot pose and covariance

- 1: **if** LiDAR lap not completed **then**
- 2: reset matching information and mode
- 3: compute Jacobian F_x and predict state with motion model
- 4: $P \leftarrow F_x P F_x^\top + Q_{\text{odom}}$

```

5:   return
6: end if
7: reset lapDone flag
8: classify each detected wall as machine, warehouse or invalid, based on its length
   and LiDAR-specific tolerances
9: initialize all matches as INVALID
10: attempt matching using Localized mode (subsection 0), using the current pose esti-
    mate
11: if no matches found then
12:   attempt matching using Lost mode (subsection 0)
13:   if no matches found then
14:     set localization mode to INVALID
15:   else
16:     set localization mode to LOST
17:   end if
18: else
19:   set localization mode to LOCALIZED
20: end if
21: if mode is INVALID then
22:   compute Jacobian  $F_x$  and predict state with motion model
23:    $P \leftarrow F_x P F_x^\top + Q_{\text{odom}}$ 
24:   return
25: end if
26: configure process noise  $Q$  and innovation gate  $\text{gate\_mah2}$  according to the mode
27: compute Jacobian  $F_x$  and predicted state  $(x_0, y_0, \theta_0)$  with the motion model
28:  $P_{\text{pred}} \leftarrow F_x P F_x^\top + Q$ 

```

▷ **Pass 1 – Mahalanobis gating against the pure prediction**

```

29: for each matched reference wall  $r$  do
30:   compute measured distance and angle  $z_{\text{med}}$  from the matched LiDAR line
31:   compute predicted measurement  $z_{\text{pred}}$  using the predicted state  $(x_0, y_0, \theta_0)$ 
32:   compute innovation  $\nu = z_{\text{med}} - z_{\text{pred}}$ , normalizing the angular component
33:   configure measurement noise  $R$  according to mode, LiDAR model and measured
       distance
34:   compute observation Jacobian  $H_r$  using the predicted state

```

```

35:   compute innovation covariance  $S_r = H_r P_{\text{pred}} H_r^\top + R$ 
36:   if  $S_r$  is singular then
37:     reject this matched wall
38:     continue
39:   end if
40:   compute squared Mahalanobis distance  $d_r^2 = \nu^\top S_r^{-1} \nu$ 
41:   if  $d_r^2 \leq \text{gate\_mah2}$  then
42:     mark wall  $r$  as valid; store  $H_r, S_r^{-1}$ 
43:   else
44:     reject this matched wall
45:   end if
46: end for

      ▷ Pass 2 – sequential EKF update using only gated matches
47: for each valid matched wall  $r$  do
48:   recompute predicted measurement  $z_{\text{pred}}$  using the current updated state  $(x_0, y_0, \theta_0)$ 
49:   recompute innovation  $\nu = z_{\text{med}} - z_{\text{pred}}$ , normalizing the angular component
50:   reconfigure measurement noise  $R$  (mode- and distance-dependent)
51:   compute Kalman gain  $K_r = P_{\text{pred}} H_r^\top S_r^{-1}$ 
52:    $(x_0, y_0, \theta_0) \leftarrow (x_0, y_0, \theta_0) + K_r \nu$ ; normalize  $\theta_0$ 
53:    $P_{\text{pred}} \leftarrow (I - K_r H_r) P_{\text{pred}}$ 
54:   mark pose as corrected
55: end for
56:  $(x, y, \theta) \leftarrow (x_0, y_0, \theta_0)$ ;  $P \leftarrow P_{\text{pred}}$ 
57: return updated pose and covariance

```

A.2. Relevant figures

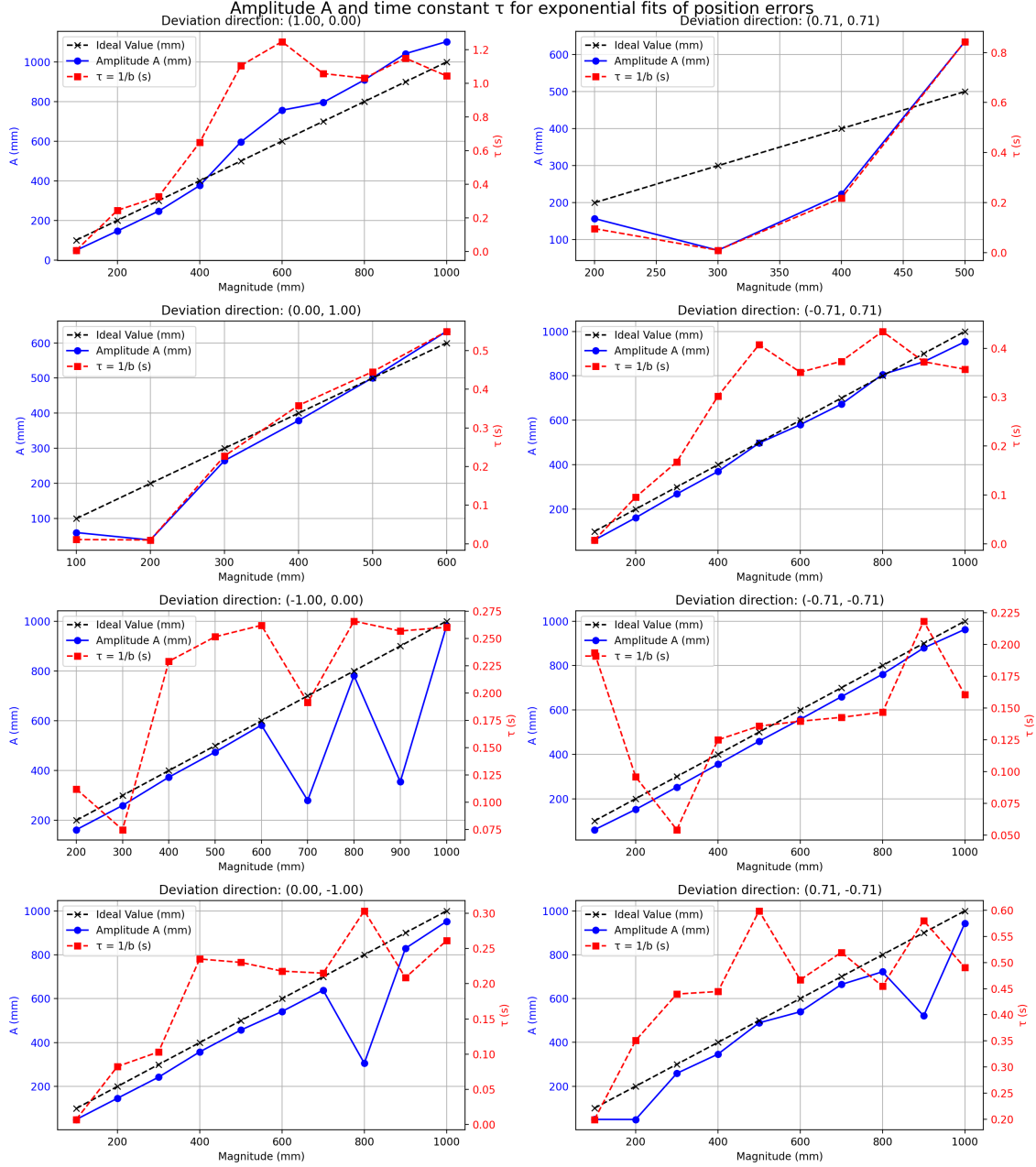


Figure A.1: Exponential converge parameters A , τ for all pose deviations applied with the robot in Position 1, represented in Figure 6.2. Each plot corresponds to the parameters estimated for each deviation magnitude along a certain direction. When the amplitude does not match the ideal amplitude value for that magnitude, the data from the first scan of the convergence may be missing or some convergence had already happened.

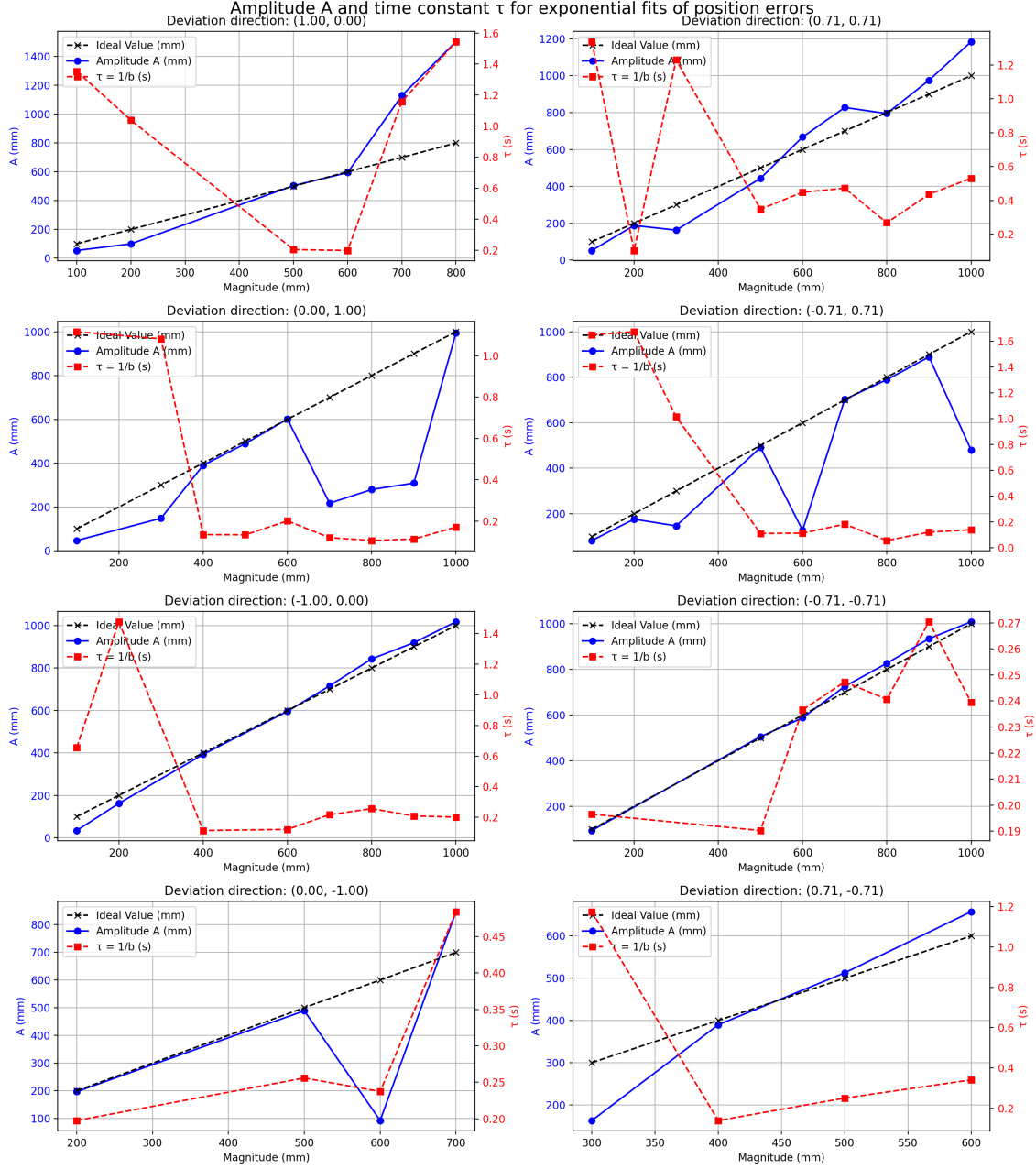


Figure A.2: Exponential converge parameters A , τ for all pose deviations applied with the robot in Position 2, represented in Figure 6.2. Each plot corresponds to the parameters estimated for each deviation magnitude along a certain direction. When the amplitude does not match the ideal amplitude value for that magnitude, the data from the first scan of the convergence may be missing or some convergence had already happened.

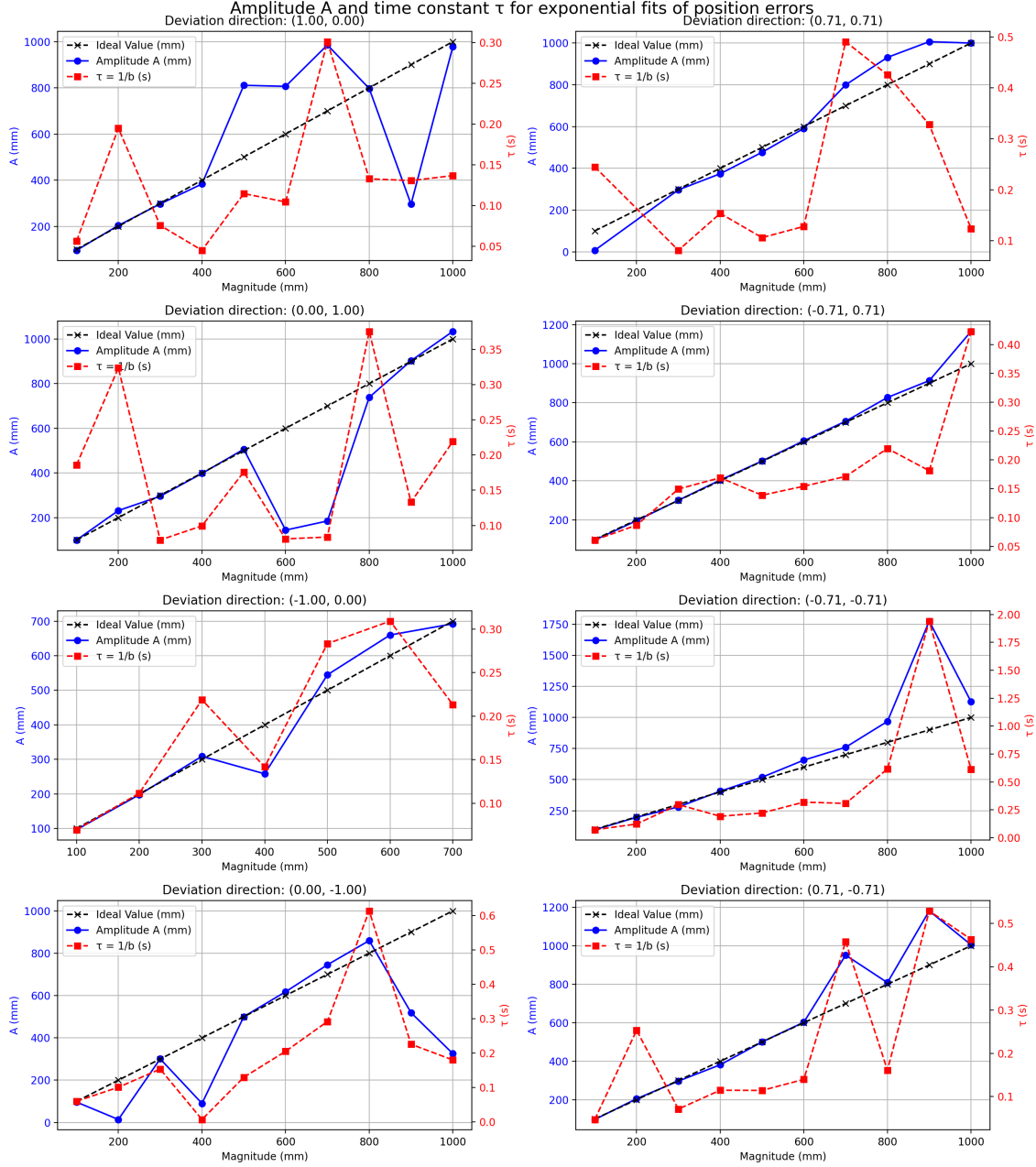


Figure A.3: Exponential converge parameters A , τ for all pose deviations applied with the robot in Position 3, represented in Figure 6.2. Each plot corresponds to the parameters estimated for each deviation magnitude along a certain direction. When the amplitude does not match the ideal amplitude value for that magnitude, the data from the first scan of the convergence may be missing or some convergence had already happened.

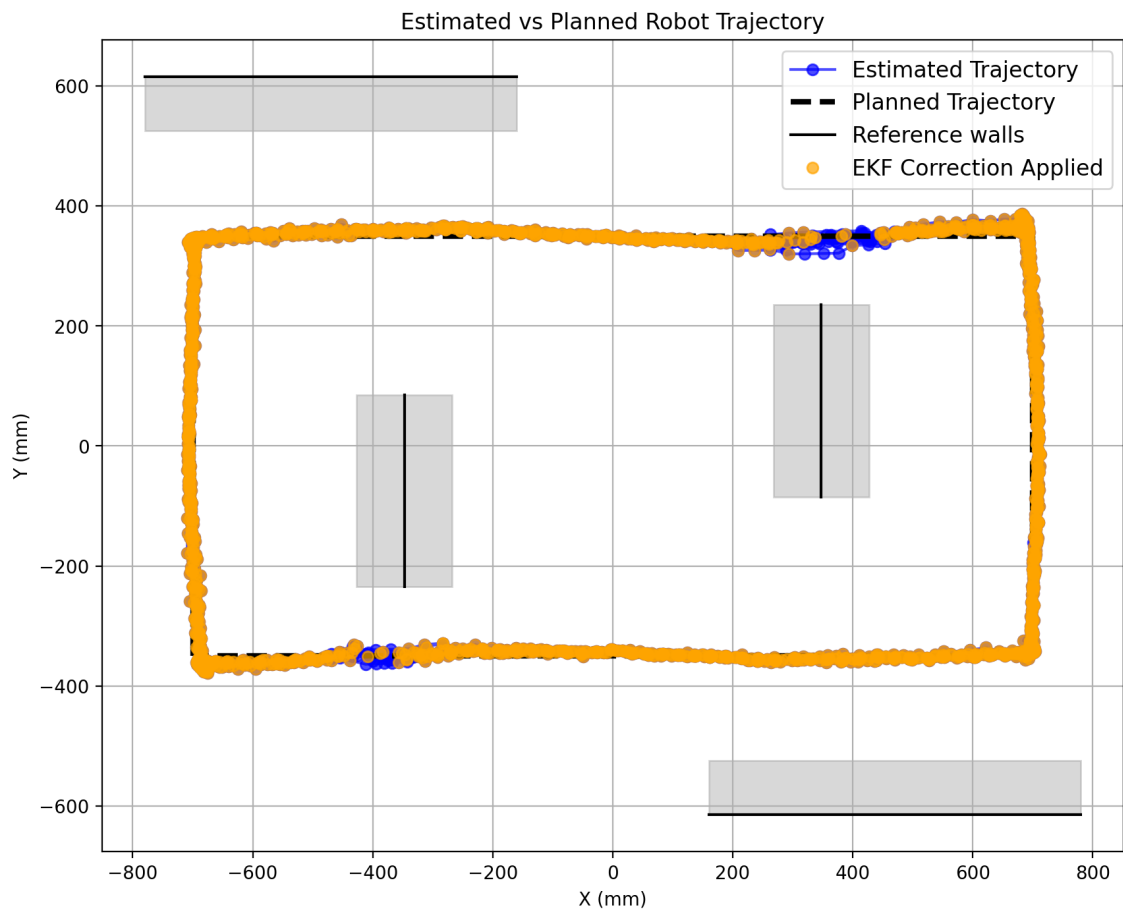


Figure A.4: Poses estimated over the course of one simple lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.

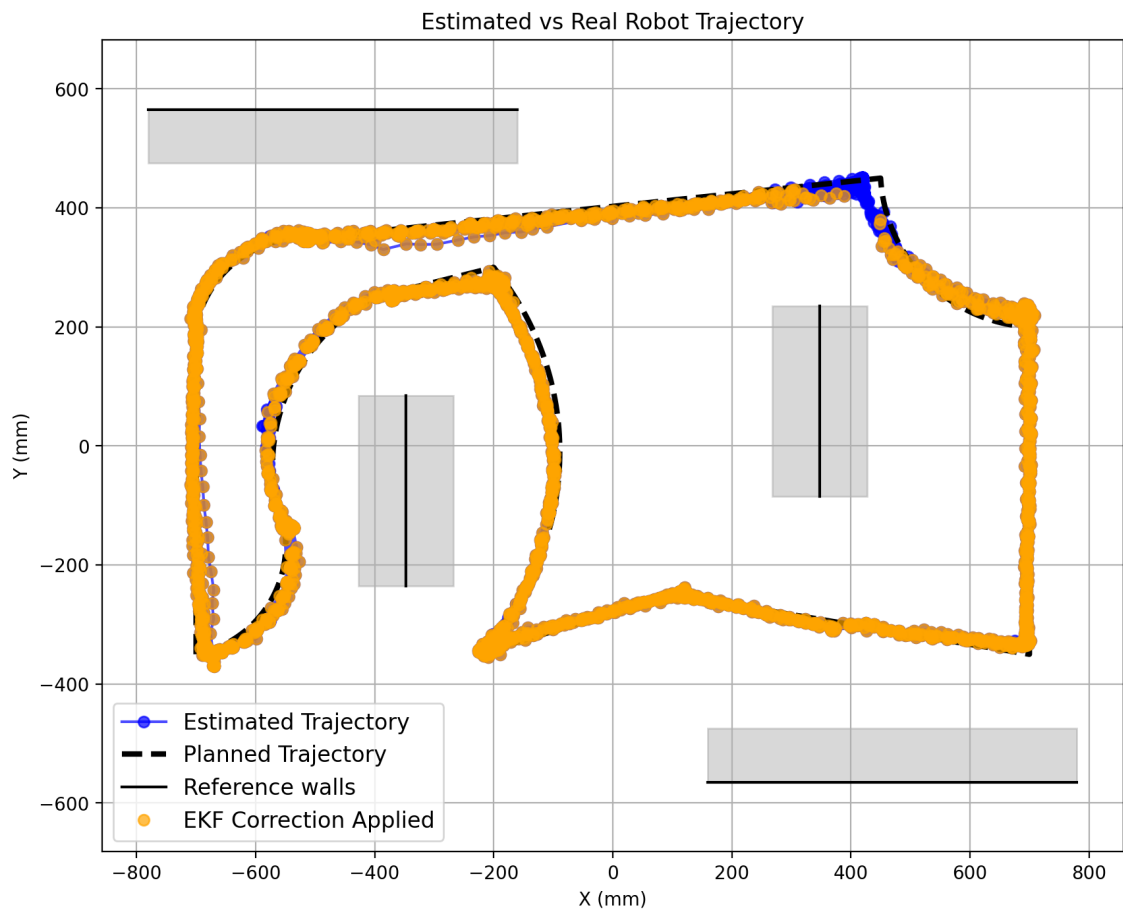


Figure A.5: Poses estimated over the course of one complex lap of the track. Orange dots indicate where 2 or more walls were observed and an EKF update was performed.