CHALMERS
UNIVERSITY OF TECHNOLOGY

Implementation of LiDAR and SLAM on a Small-Scale Autonomous Platform

Investigation of how LiDAR can be used to Simultaneously Localize a Vehicle and Map its Surroundings using SLAM

Master's thesis in Systems, control and mechatronics

DAVID ESPEDALEN
ANTON STIGEMYR HILL

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Implementation of LiDAR and SLAM on a Small-Scale Autonomous Platform

Investigation of how LiDAR can be used to Simultaneously Localize a Vehicle and Map its Surroundings using SLAM

DAVID ESPEDALEN
ANTON STIGEMYR HILL

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2025

Implementation of LiDAR and SLAM on a Small-Scale Autonomous Platform
Investigation of how LiDAR can be used to Simultaneously Localize a Vehicle and Map its Surroundings using SLAM
DAVID ESPEDALEN, ANTON STIGEMYR HILL

Supervisor: Hamid Ebadi, PhD in Computer Science, Infotiv AB
Examiner: Bengt Lennartson, Department of Electrical Engineering, Chalmers University of Technology

Master's Thesis 2025
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image of autonomous platform generation 4

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Implementation of LiDAR and SLAM on a Small-Scale Autonomous Platform
Investigation of how LiDAR can be used to simultaneously localize a vehicle and map its surroundings using SLAM
DAVID ESPEDALEN, ANTON STIGEMYR HILL

Department of Electrical Engineering
Chalmers University of Technology

Abstract

Autonomous driving (AD) technology continues to evolve with goals of enhancing road safety, lowering emissions and increasing transportation efficiency. Modern solutions rely on sensors including GPS, radar, camera and LiDARs to perceive their surroundings and enabling safe and precise navigation. Continued research and development in this field require accessible and adaptable platforms for doing so on.

This thesis, conducted in collaboration with Infotiv AB, builds on two previous master theses and focuses on developing such a platform and expanding it, making the development more available. The autonomous platform 4 (AP4) is a small-scale autonomous platform based on the Ninebot GoKart, which has been implemented with a Raspberry Pi 4b and ROS2 environment.

The focus of this thesis was to add onto the existing platform by integrating a LiDAR and explore sensor fusion possibilities with an IMU, both in simulation and in real-life. By doing this, the work aimed to improve localization and path planning through simultaneous localization and mapping (SLAM) to further improve the AP4. This was done while simultaneously keeping the platform modular and scalable.

A LiDAR and IMU pipeline has been integrated both in simulation and on the physical platform. For evaluation, a local go-kart track was rendered and used as grounds for both cases. The simulations proved that the concept using SLAM with a LiDAR and sensor fusion of an IMU can work for self driving on a small-scale platform. However, the implementation to the physical environment highlights the necessities of well tuned sensors and position estimations.

Keywords: Simultaneous Localization and Mapping, LiDAR, ROS 2, Navigation 2, Autonomous Driving

Acknowledgements

First of all, we would like to thank Infotiv for giving us the opportunity to work on this project during our master's thesis. They have provided us with all the necessary conditions, tools, and support from beginning to end.

We would also like to thank our supervisor at Infotiv, Hamid Ebadi, for his engagement throughout the thesis and for providing us with support, guidance, and valuable insights during the project.

Furthermore, we would like to thank our examiner and supervisor at Chalmers, Bengt Lennartson, for his valuable insights into the subject and for the helpful feedback he provided throughout the thesis.

Lastly, we would also like to thank the staff at Gokartcentralen in Kungälv for kindly allowing us to test and evaluate the go-kart at their track.

David Espedalen, Anton Stigemyr Hill, Gothenburg, June 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AD	Autonomous drive
AP4	Autonomous Platform 4
BC	Behavior Cloning
BFS	Breadth-First Search
CAN	Controller Area Network
CTE	Cross Track Error
DWA	Dynamic Window Approach
E/E	Electrical/Electronic
ECU	Electrical Control Unit
EKF	Extended Kalman Filter
FOV	Field Of View
HG-DAgger	Human Gated Dataset Aggregation
HLC	High-Level Control
HWI	Hardware Interface Low-Level
ICP	Iterative Closest Point
IMU	Inertial Measurement Unit
LiDAR	Light Detection and Ranging
Nav2	Navigation 2
ORB	Oriented FAST and Rotated BRIEF
OS	Operating System
RMSE	Root Mean Squared Error
ROS 2	Robot Operating System 2
RViz	ROS Visualization
SLAM	Simultaneous localization and mapping
SPCU	Steering and Propulsion Control Unit
SSCU	Speed Sensor Control Unit
URDF	Unified Robot Description Format
USB	Universal Serial Bus

Nomenclature

Below is the nomenclature of parameters that have been used throughout this thesis.

Parameters

ϕ	Roll
ψ	Pitch
θ	Yaw
θ_{acc}	Angle theta calculated from the accelerometer
θ_{gyro}	Angle theta calculated from the gyroscope
θ_{yaw}	Angle around the z-axis
α	Weighting factor for complementary filter
ω_z	Angular velocity around the z-axis
a_x	Acceleration in x-direction in the robot frame
a_y	Acceleration in y-direction in the robot frame
a_x^{global}	Acceleration in x-direction in the global frame
a_y^{global}	Acceleration in y-direction in the global frame
v_x	Velocity in x-direction
p_x	Position in x-direction
v_y	Velocity in y-direction
p_y	Position in y-direction
r	Turning radius
δ	Turning angle
L	Length between the front and back wheels
x	X-position
y	Y-position
W	Length between rear wheels

Contents

List of Figures

List of Tables

1
Introduction

In recent years, autonomous drive (AD) technology has become a common topic in the technical world. The goal is to increase the safety in everyday life, whether it is for personal or professional vehicle use. In addition to improving safety, there is also interest in reducing the energy costs and increasing the efficiency of the overall transportation in the world.

Today's AD system are operating by using multiple sensor technologies which includes GPS, radar, cameras and LiDAR. By using these technologies, the system can create a 3D map of the vehicle's environment which for instance includes roads, buildings, other vehicles, traffic lights and road signs. Advanced computer systems analyze sensor data and make real-time decisions about vehicle operations such as continuously adjusting steering, speed, acceleration, and braking based on the vehicles surrounding environment[1].

In recent years, it has become more and more common to take advantage of the fast development of artificial intelligence. By using these advanced models, the system can collect and learn from the surrounding data in order to make better decisions and increase the ability of AD vehicles to drive. More precisely, the vehicle can make decisions without receiving information about every specific situation on the road [2].

1.1 Background

This master's thesis is a project built on two previous master's theses. One, done by Johan Wellander and Arvid Petersén in 2024, and the other by Erik Magnusson and Fredrik Juthe, who did their project in 2023 [3][4]. The project is a collaboration with Infotiv AB, which has developed a small-scale autonomous platform (AP4, Autonomous Platform 4) with the purpose of investigating and testing different autonomous drive technologies. The platform serves as a tool to do research in these areas without the need for expensive equipment and test sites.

The first project, during 2023, aimed to design and implement a modular centralized electrical/electronic (E/E) setup, combined with the necessary hardware and software architecture on the Ninebot go-kart[5]. This in order to make it easy to modify the functionalities and the hardware components in future research projects.

The project made in 2024 aimed to implement AD on the autonomous platform using imitation learning, specifically using Human Gated Dataset Aggregation (HG-DAgger) and Behavior Cloning (BC). In addition to that, it also aimed to investigate whether using a depth camera or incorporating Oriented FAST and Rotated BRIEF (ORB), which is a 2D object recognition system within the field of computer science, could improve the performance of the AD[3].

This project will focus on implementing additional sensors by using LiDAR to provide the system with more data and in that way give the AP4 a clearer picture of the localization and the appearance of the environment[6]. Further work on integrating a simultaneous localization and mapping (SLAM) environment and path planning will also be worked on to advance the usability of the AP4[7][8].

1.2 Related work

Apart from the two previous master's theses on this particular project, there is also a project done by Oscar Sanner in 2023 who builds an autonomous drive robot that can both map and move around in unknown areas using only a LiDAR and a Raspberry Pi [9]. The goal was to test different SLAM algorithms and see how well they perform in terms of position accuracy, map quality, and processing power. The author also tested different path planning methods for the robot to move safely in the environment. The SLAM algorithms were divided into two types, landmark-based and map-based. For landmark-based SLAM, the system uses special features in the environment to figure out where it is. For map-based SLAM, the robot builds a full map by matching LiDAR data using an algorithm called Iterative Closest Point (ICP).

The results showed that it is possible to run SLAM in real-time on a Raspberry Pi. The position accuracy was usually better than 10 cm, and the map error was below 15 cm in most cases. The robot could also follow paths and avoid obstacles using path planning algorithms.

This work shows that hardware like a Raspberry Pi and LiDAR can be used for effective autonomous navigation, and compares different techniques to find the best approach in a limited computing environment.

1.3 Objective

The aim of this thesis is to implement an improved AD algorithm on the small-scale autonomous platform. This will be achieved by integrating a LiDAR into the already existing hardware architecture. Subsequently, the project will focus on developing the AD software within the ROS 2 environment, incorporating the new LiDAR sensor to improve the overall performance of the AD system.

The research questions investigated during the thesis are presented below.

- Will the implementation of a LiDAR improve the existing AD platform?
- How can LiDAR-IMU sensor fusion improve the AD platform?
- Can path planning using memory and SLAM be integrated to enhance the performance?

1.4 Limitations

During the project, access to the go-kart track at Gokartcentralen in Kungälv will be available [10]. Therefore, the testing of the algorithms and the functionality of the AP4 will primarily take place at this location and it is at this location where all the physical testing is done. The project will be based on the existing platform, with the only addition being the LiDAR sensor. Since the platform is built on a Raspberry Pi 4b, the software should be compatible and capable of running on that or on a laptop that is integrated with the platform.

This chapter introduced the project's background, objectives, and limitations, which are the foundation for the thesis. The next chapter builds on this by presenting the theory behind the project so that readers can understand all the technical components and software used for the AP4.

2

Theory

This chapter presents the theoretical background necessary to understand the implementation of the project. It will give an explanation of key concepts that will be expanded on in the AP4. These include sensor technologies, robot localization models, filtering techniques and relevant software frameworks. The topics are introduced in their corresponding implementation order within the project to follow a logical progression.

2.1 Sensors

As part of the theoretical framework, this section provides a brief overview of the principles of each sensor. This includes typical applications and key equations used to derive meaningful information from their raw outputs. Understanding how these sensors work is essential for their integration later in the project.

2.1.1 LiDAR

Light Detection and Ranging (LiDAR) is a sensor that measures the distance to a target by emitting a short laser pulse and recording the time interval between its emission and the detection of the reflected pulse. This enables the creation of a very precise 2-D representation of the environment which makes it suitable for autonomous vehicles to interact with the physical world.[11]

The distance is calculated by using the laser's velocity and the time interval according to the following formula, where c is the speed of light and t is the time interval:

$$distance = \frac{c \cdot t}{2} \tag{2.1}$$

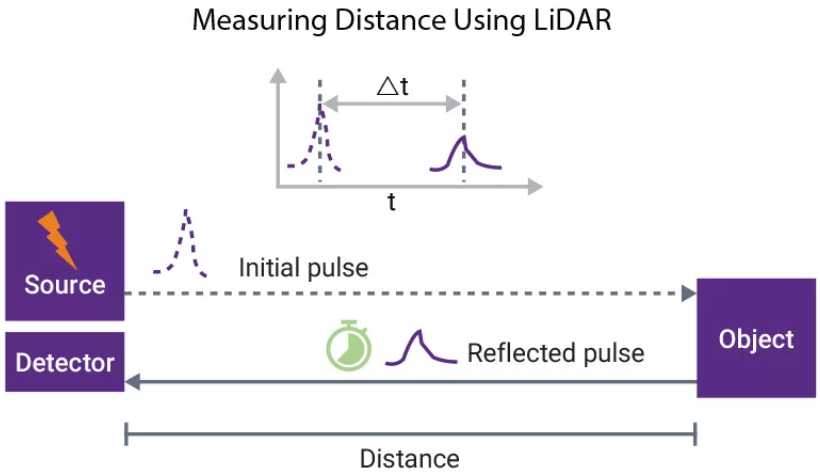


Figure 2.1: Image describing how the LiDAR is calculating the distance to an object. Image source:[12]

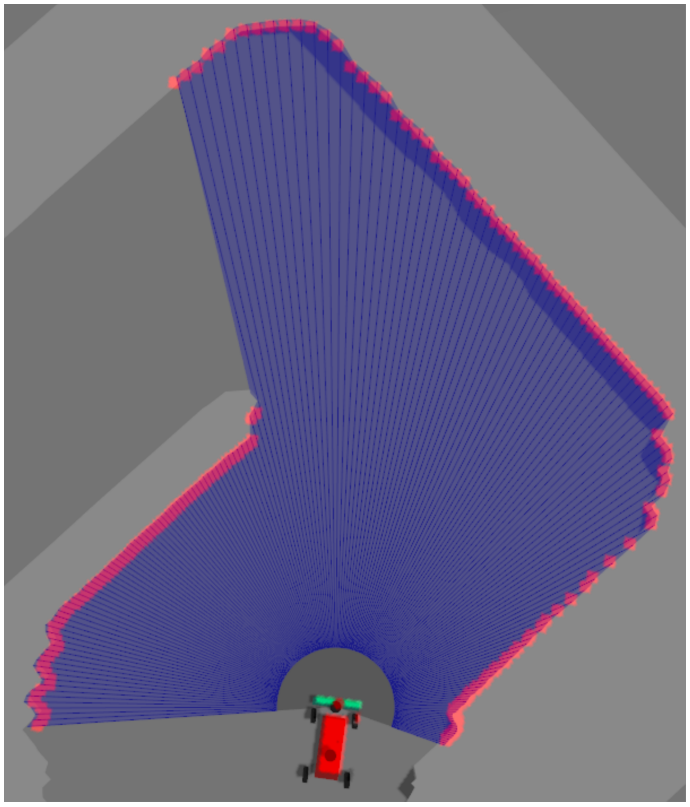


Figure 2.2: Image showing the LiDAR scan points on the AP4 simulation in Gazebo

2.1.2 Speed sensor

To measure the speed of the go-kart, a LM393 speed sensor is used, which can be seen in Figure 2.3 [13]. The LM393 speed sensor is an infrared tranceiver/receiver that can detect whether there is an obstacle or not between them. By mounting a disk with evenly spaced holes on the wheels, the speed sensor can count the number of holes in a set period and thereby calculate the speed of the robot according to the following expressions:

$$rps = \frac{pps}{ppr} \tag{2.2}$$

$$v = rps * 2 * \pi * r \tag{2.3}$$

Equation for rotational speed and velocity:
rps: rotations per second,
pps: pulses per second,
ppr: pulses per revolution.

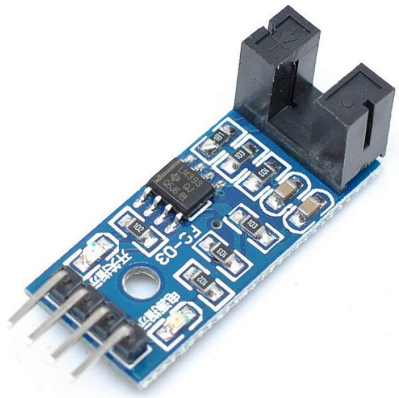


Figure 2.3: LM393 speed sensor. Image source:[14]

2.1.3 IMU

Inertial measurement units (IMUs) are sensors designed to register movement. Often integrated with gyroscopes, accelerometers, and magnetometers, they can capture measurements in 9-degrees of freedom, each in the x, y, and z directions. By combining these sensors, motion and orientation can be estimated.

To estimate the position of the acceleration and velocity measurements, integrating the values of the corresponding amount will provide the wanted information[15].

However, due to the integration, if small errors occur in the measurements, they will accumulate, resulting in a drift of the measurements. This can be accounted for by filtering.

To estimate the yaw, x and y position the gyroscope and accelerometer can be used. For yaw estimation, the following estimation is used:

$$\theta_{acc} = atan2(a_y, a_x) \tag{2.4}$$
$$\theta_{gyro}(t+1) = \theta_{gyro}(t) + \omega_z(t) * \Delta t \tag{2.5}$$
$$\theta_{yaw}(t+1) = \alpha(\theta_{gyro}(t+1)) + (1-\alpha)\theta_{acc}(t) \tag{2.6}$$

This is a complimentary filter and when this is done, the yaw angle can be used to estimate the robot's position in the global frame:

$$a_x^{\text{global}} = a_x \cos(\theta_{yaw}) - a_y \sin(\theta_{yaw}) \tag{2.7}$$
$$a_y^{\text{global}} = a_x \sin(\theta_{yaw}) + a_y \cos(\theta_{yaw}) \tag{2.8}$$
$$v_{x,y}(t+1) = v_{xa,y}(t) + a_{x,y}^{\text{global}} \Delta t \tag{2.9}$$
$$p_{x,y}(t+1) = p_{x,y}(t) + v_{x,y}\Delta t + (a_{x,y}^{\text{global}})/2(\Delta t)^2 \tag{2.10}$$

These positions will, however, be subject to drift, which can be accounted for and used to estimate position[16].

2.2 Odometry

Odometry is the process of measuring and estimating the position and orientation of a robot based on its movement over time. This is done by integrating sensory data as wheel encoders and IMUs, to calculate an estimation of the position. For SLAM, the odometry of the robot is crucial as it provides an estimate of the robot's motion.

2.2.1 Ackermann model

The Ackermann drive system is a system with the drive axle on the back wheels and steering controlled by rotation of the front wheels. The Ackermann model has 3 subdivision, Regular-, Parallel- and Reverse Ackermann drive. These affect the turning radius of the robot differently. To reduce slip in low velocities, both front wheels need to follow different radii that are centered at the same position. If this is not the case, the distance between the paths of the wheels will change, which is not possible for a driving system with a steering axle of fixed length. To make this possible, the regular Ackermann steering geometry is set up to make the inner wheel of a turn always turn more than the outer wheel, accommodating this. If this was not the case, the wheels would follow two untraceable paths, making the go-kart slip.

In reality, a different problem occurs. When turning at high velocities, the robot is subject to the centripetal force, which will focus a greater force on the outer wheel.

This means the robot is more affected by the outer wheels' turning radius than the inner ones. Therefore, in these occasions, the Reverse Ackermann model is used to improve grip in fast-moving robots. An example where this is implemented in real life is formula one cars[17].

To calculate the odometry of an Ackermann drive, a standard simplification is to add an imaginary wheel in the center of the front wheel axle. This can be seen in Figure 2.4. Then by examining the dimension, velocity, heading and center of rotation, the odometry is calculated by the following[18]:

$$r = \frac{L}{\tan(\delta)} \tag{2.11}$$

$$d\theta = \frac{v}{L} \cdot \tan(\delta) \cdot dt \tag{2.12}$$

$$x_{i+1} = x_i + r * (\sin(\theta + d\theta) - \sin(\theta)) \tag{2.13}$$

$$y_{i+1} = y_i - r * (\cos(\theta + d\theta) - \cos(\theta)) \tag{2.14}$$

$$\theta_i = \theta_{i-1} + d\theta \tag{2.15}$$

Equation for Ackermann odometry calculation:

δ: turning angle, r: turning radius,
θ: vehicle direction, L: length between front and back wheels
x, y: 2D cooridnates.

To calculate the angle at which to set the inner and outer front wheels, a modification by the width of the front wheel axle can be adjusted to reach the following relation:

$$\tan(\delta_i) = \frac{L}{r - W/2} \tag{2.16}$$

$$\tan(\delta_o) = \frac{L}{r + W/2} \tag{2.17}$$

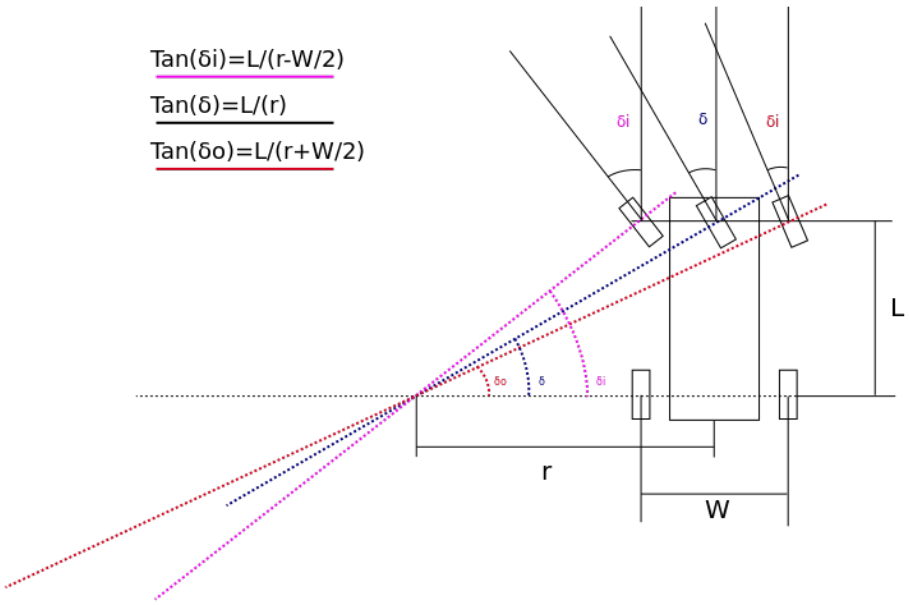


Figure 2.4: Simplification of Ackermann drive odometry

2.3 Filtering

For all data acquisition, small errors will be introduced. Depending on the sensor, the errors vary in mean and variance, which contributes to false errors in calculations further down. These errors can however be taken into consideration by filtering.

2.3.1 Mean and variance

First, the signals' mean and variance need to be gathered. These are calculated by long sets of samples from the sensor and investigating them. By doing this, the measurements are assumed to be normally distributed. The mean and variance are then calculated according to[19][20]:

$$E[x] = \sum_{i}^{n} \frac{x_i}{n} \tag{2.18}$$

$$Var[x] = \sum_{i=1}^{n} \frac{(x_i - E[x])^2}{n} \tag{2.19}$$

2.3.2 Sensor fusion

When the means and variances are gathered, the measurements can be combined with sensor fusion. A typical way to do this for nonlinear systems is to use an extended Kalman filter (EKF). The EKF is an extension to the regular Kalman

filter to handle nonlinearities. When nonlinearities are introduced, the transition and observation models cannot be used to compute the predicted measurements from the predicted states, instead, their corresponding Jacobians are used[21]. Then the EKF is calculated as follows:

Algorithm 1 EKF Prediction Step

Input:	$\hat{\mathbf{x}}_{k-1\|k-1}$, $\mathbf{P}_{k-1\|k-1}$, $\mathbf{u}_{k-1}$
Output:	$\hat{\mathbf{x}}_{k\|k-1}$, $\mathbf{P}_{k\|k-1}$
Predicted state estimate:	$\hat{\mathbf{x}}_{k\|k-1} = f(\hat{\mathbf{x}}_{k-1\|k-1}, \mathbf{u}_{k-1})$
Predicted covariance estimate:	$\mathbf{P}_{k\|k-1} = \mathbf{F}_k \mathbf{P}_{k-1\|k-1} \mathbf{F}_k^\top + \mathbf{Q}_{k-1}$

Algorithm 2 EKF Update Step

Input:	$\hat{\mathbf{x}}_{k\|k-1}$, $\mathbf{P}_{k\|k-1}$, $\mathbf{z}_k$
Output:	$\hat{\mathbf{x}}_{k\|k}$, $\mathbf{P}_{k\|k}$
Innovation (measurement residual):	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - h(\hat{\mathbf{x}}_{k\|k-1})$
Innovation covariance:	$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k\|k-1} \mathbf{H}_k^\top + \mathbf{R}_k$
Kalman gain:	$\mathbf{K}_k = \mathbf{P}_{k\|k-1} \mathbf{H}_k^\top \mathbf{S}_k^{-1}$
Updated state estimate:	$\hat{\mathbf{x}}_{k\|k} = \hat{\mathbf{x}}_{k\|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k$
Updated covariance estimate:	$\mathbf{P}_{k\|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k\|k-1}$

2.4 Software

Implementing a project of this size requires many systems to handle and manage its various tasks. These include standardized development environments, communication protocols between different system components, visualization tools, path planning algorithms and simulation platforms. Using robust open-source variants of these systems allows for faster development without losing the ability to fine-tune system parameters as needed. The following section is an overview of the software tools used in this project and an explanation of their roles within the system architecture.

2.4.1 Robot Operating System 2

ROS 2 is an open-source framework to develop and manage robotic applications. The framework is built in a modular way, which makes it adaptable to robotic projects that have different components and different languages that are supposed to communicate with each other [22].

Communication

Each program or process running within the ROS 2 framework are assigned to nodes, and these nodes can communicate with each other using topics, services, or actions

depending on what type of communication is needed [23].

Topics: The most common communication method which is based on a publish-subscribe model. The topic method enables message passing, allowing nodes to send and receive data without a direct connection. One node can publish messages to a topic and another node can subscribe to the topic and access the data. Topic communication is very useful for updating messages in real-time for example sensor data. By using topics instead of directly sending data through nodes, several processes can subscribe and publish to the same topic without the risk of blocking communication. If a ROS 2 topic is referred to later in the report, it will be referred to as */topic_name*.

Services: A request-response communication model where one node sends a request, and another node replies. This ensures that a task is completed by verification, and allows controlled data passing.

Actions: Designed for long-running tasks where actions combine request-response with periodic feedback and the ability to cancel execution. This makes them useful for complex robotic operations like navigation and manipulation.

RViz

ROS Visualization, RViz, is a 3D visualization tool for ROS. By subscribing to the ROS topics described above, it allows users to visualize sensor data, robot models, maps and planning algorithms from a simulated or real-world environment. This serves as a platform for the user to visualize and interact with the project they are working on [24].

2.4.2 Navigation 2 stack

Navigation 2 stack (Nav2) is a framework for ROS 2 designed for robot's to efficiently move autonomously in a known or unknown environment.

Nav2 provides localization and navigation of the robot and uses different planners and servers in order to achieve this [25].

Localization

Nav2 uses Adaptive Monte Carlo Localization (AMCL) to determine the robot's position in an environment. AMCL is a probabilistic localization algorithm that uses a particle filter to estimate the robot's position based on a known map. It works by comparing the robot sensor data with the map to determine its location[26].

The particle weights indicate how closely their predicted positions match the actual robot position. After each move or sensor update, the particles are re-sampled,

helping the algorithm converge to the correct robot position over time.

Planner Server

The planner server which, is often called the global planner, is computing the global path according to the map and the obstacles within the map. The planner server comes with a multitude of different types of planner plugins, which can be chosen depending on which kind of drive system the robot is using and through which environment it is navigating. The most common planner is the *NavFn Planner*, which is suitable for differential drive robots, which means that the robot turns by using different speeds on its wheels to rotate around its axis. This project is using a go-kart with an Ackermann drive steering. For robots such as that, there is a planner plugin called *SmacPlannerHybrid* which is suitable for robots with Ackermann steering.

The SmacPlannerHybrid A* is a two-step path planning system made for self-driving cars that need to move through tricky, changing environments that have tight spaces and dynamic obstacles which make the robot to need to figure out new paths during the navigation.

The A* algorithm combines the Dijkstra's algorithm and BFS (Breadth-First Search) algorithm in order to find the shortest path from a point A to B. The algorithm is based on the function[27]:

$$f(n) = g(n) + h(n) \tag{2.20}$$

Where $f(n)$ is the estimated total cost from the initial position to the target position via state n, $g(n)$ is the actual cost from the initial position to the state n and $h(n)$ is the estimated cost of the best path from state n to the target state, also known as the heuristic function. The functions $g(n)$ and $h(n)$ are given as:

$$g(n) = g(parent(n)) + cost(parent(n), n) \tag{2.21}$$

$$h(n) = |x_n - x_{goal}| + |y_n - y_{goal}| \tag{2.22}$$

Where *parent* defines the previous state and the state the algorithm goes from to the target state.
The node with the lowest cost, i.e. lowest $f(n)$ is selected for exploration next [27] [28].

The first part of the Hybrid A* planner is based on the classic A* algorithm. The normal A* only works with simple grid points and doesn't take in account how a

real Ackermann steering robot moves. Hybrid A* is different and uses the car's position and direction. This means the paths it finds are ones that the car can actually drive based on it's kinematics. In order to find the best valid suitable path, Hybrid A* uses two kinds of guides called heuristics. One looks at how the car moves but ignores obstacles, and the other looks at the map of obstacles but not how the car moves. Together they provide strong guidance without compromising the admissibility of the path finding.

After Hybrid A* makes the first version of the path, the second part of the planner takes over and tries smoothing and improving the path. Even though the path is drivable, it might have hard or unnatural turns. Because of that, the planner runs an optimization process to make it smoother. This step tries to make the path shorter, keep it away from obstacles, and create smooth curves that the go-kart can drive. To avoid obstacles but still allow the go-kart to go through tight spaces, the planner uses a special Voronoi Field [29]. This works like a path-cost function that will push the car away from walls and objects, but still let it squeeze through narrow spots if needed.

Finally, to make the path even smoother for the car to follow, the planner adds more points between the original ones and smooths those out. This helps avoid sudden sharp turns and creates a more natural driving[29].

The *NavFn planner* works very similar to the *SmacPlannerHybrid A* planner* and is based on the Dijkstra's algorithm with the option to use A*-like approach as well[30]. The big difference between the *NavFn planner* and the *SmacPlannerHybrid A* planner* is that the *NavFn planner* don't have the ability to take the robots kinematics and turning in account which makes it more suitable for differential drive robot's. As well as that the *NavFn planner* doesn't have the ability to smoothen the path which the *SmacPlannerHybrid A* planner* has [31].

The planner server uses a global costmap which provides a 2D grid-based representation of the environment with a value in each cell describing how costly it is for the robot to drive in that specific area of the map. The costmap is generated by the point cloud from the LiDAR that measures and maps the environment. The planner uses the costmap to determine a valid long-term path through the environment.

Controller Server

The controller server can be seen as the local planner which is the server that performs a reactive path planning a few meters ahead, according to the current sensor data. By using the local planner, the robot can avoid obstacles that the global planner doesn't find on the map, such as dynamic obstacles.

The controller server is also the server that controls the robot and its wheels in order to follow the planned path from the global and local planner.

There are several controllers to use in Nav2. However, the most common one is the

DWB Controller. The DWB Controller uses a Dynamic Window Approach (DWA) to determine the robot's short-term path in the environment. The DWA uses the robot's velocity space, where it explores different combinations of linear and angular velocities to determine the robot's next movement. This approach takes in account the robot's physical constraints such as acceleration, steering abilities and the environment which making it suitable for real-time obstacle avoidance and navigation. To ensure feasibility, the DWA only uses velocities that the robot can reach within the next time interval t. These velocities are called admissible velocities and the set of all those velocities is defined by[32]:

$$V_a = \left\{ (v, \omega) \;\middle|\; v \leq \sqrt{2 \cdot \text{distance}(v, \omega) \cdot \dot{v}_b}, \;\; |\omega| \leq \sqrt{2 \cdot \text{distance}(v, \omega) \cdot \dot{\omega}_b} \right\} \quad (2.23)$$

where V_a is the set of all feasible velocity pairs (v, ω) that allow the robot to stop without colliding with an obstacle.

Because the robot cannot change speed instantly, this approach only looks at the speeds it can reach in a short time. This area is called the dynamic window and depends on how fast the robot can accelerate or de-accelerate. Let v_a and ω_a be the robot's current linear and angular velocity and let $\dot{v}$ and $\dot{\omega}$ be the accelerations. In the time interval t, the dynamic window V_d is defined as:

$$V_d = \{(v, \omega) \mid v \in [v_a - \dot{v}t, \; v_a + \dot{v}t] \,, \; \omega \in [\omega_a - \dot{\omega}t, \; \omega_a + \dot{\omega}t]\} \quad (2.24)$$

This shows all the speeds the robot can reach in time t which are the speeds that are considered when avoiding obstacles. Anything outside this area is too far for the robot to reach and will therefore not be taken in account for this time interval.

Let V_s be the set of all possible velocities. The set V_a contains velocities that are safe. The set V_d includes velocities the robot can reach from its current speed within the given time considering its acceleration limits.

The final search space, namely the reduced search space V_r, is the safe space for the robot within the time interval and is the intersection of these three sets as defined below:

$$V_r = V_s \cap V_a \cap V_d \quad (2.25)$$

After applying the constraints on velocity, the search space for the possible velocities is reduced. This reduced space V_r includes only those velocities that are achievable, safe, and avoid collisions with obstacles (See Figure 2.5).

The controller server uses a local costmap which works similar to the global costmap. The controller uses the costmap to determine a valid short-term path through the environment from the sensor data.

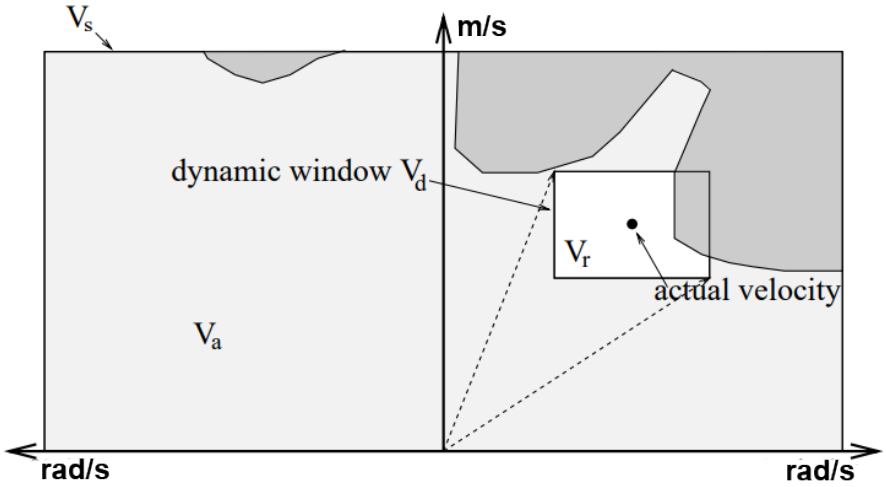


Figure 2.5: Dynamic Window Approach. Image source:[32]

Behavior Tree Navigator Server

The Behavior Tree Navigator Server is the node that calls for the planner server and the controller server to start navigate the robot through the environment. See the Nav2 architecture in Figure 2.6 to get overview of Nav2 and it's different servers and nodes[25].

Behavior Server

The behavior server or, as it also is called, the recovery server is the server that provides the system with a safe and secure navigation. The Behavior tree navigator server is calling the recovery server when the robot gets stuck or can't find a valid path. The recovery server will then make the robot do some pre-defined movements such as turn, reverse or wait to make new conditions for the robot to find a valid path or get realesed from it's stuck position[25].

2.4.3 Docker container

The Docker container is a technology that combines the application, the dependencies and the system library organized in what is called a container. This platform is known as Docker and ensures that the software can work in all environments. The Docker containers are run on the host computer and each container is then running its own isolated operating system (OS) in which we run the application. The Docker containers share the host computers resources which make them more efficient and gives the possibility to run multiple containers at the same time on the same host machine [33][34].

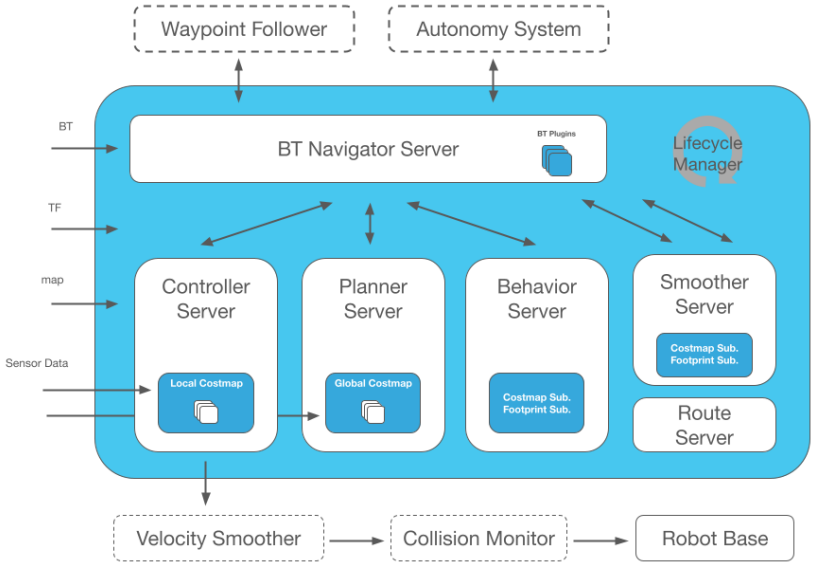


Figure 2.6: Navigation 2 Stack architecture. Image source:[25]

Docker image

A Docker image is an instruction or blueprint that includes all the necessary software to build the Docker container. The Docker image is a static file and contains application code, libraries, tools, dependencies and other files that are needed to make an application run. The Docker image is also immutable, which means that the image cannot be changed after it is built. This makes it very useful for development and testing of new software because there will always be a backup Docker image [35].

2.4.4 Gazebo

Gazebo is a robot simulation software designed to create realistic virtual environments for testing and development of robotic systems. It features a built-in physics engine which can be customized to match real-world conditions, enabling accurate simulations for the developers need. In Gazebo, sensor data and robot behavior are simulated through Gazebo topics which can be bridged to ROS topics allowing configuration between the two.

2.4.5 SLAM

Simultaneous localization and mapping (SLAM) is the problem of constructing a map of an unknown environment while simultaneously estimating the robot's position within that environment. It is done by collecting information about the environment using sensors such as LiDARs or GPS and using algorithms such as particle filters or Kalman filters to approximate the robot's location. This applies Bayesian statistics where sensor readings and motion models are continously updated to produce the best approximation of the robot's position and the surrounding map [7][36].

The process begins with the robot estimating its motion over time using its odometry. These sensors measure how the robot moves which creates a motion model that predicts the robot's new position. Mathematically this can be described as:

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t) + \mathbf{w}_t \tag{2.26}$$

where $\mathbf{x}_t$ is the robot's pose at time t, $\mathbf{x}_{t-1}$ is the previous pose, $\mathbf{u}_t$ is the control input from odometry and $\mathbf{w}_t$ is the motion noise. This prediction is useful but it's not perfect due to cumulative drift.

To correct this drift and refine the robot's understanding of the environment, the robot uses sensors like a LiDAR or GPS to scan its environment. The sensors provides distance measurements to close objects which the system uses to detect obstacles in the environment. These measurements are fed into the observation model which is defined as:

$$\mathbf{z}_t = h(\mathbf{x}_t, \mathbf{m}) + \mathbf{v}_t \tag{2.27}$$

Where $\mathbf{z}_t$ is the actual sensor data at time t, $h(\mathbf{x}, \mathbf{m})$ is the expected measurement given the robot's pose $\mathbf{x_t}$ and the current map $\mathbf{m}$ and $\mathbf{v}_t$ represents the noise. SLAM combines the motion and observation models using a probabilistic estimation. Through a Bayesian update, SLAM estimates the probability of the current pose and map according to:

$$P(\mathbf{x}_t, \mathbf{m} \mid \mathbf{z}_{1:t}, \mathbf{u}_{1:t}) \propto P(\mathbf{z}_t \mid \mathbf{x}_t, \mathbf{m}) \int P(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{u}_t) P(\mathbf{x}_{t-1}, \mathbf{m} \mid \mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1})\, d\mathbf{x}_{t-1} \tag{2.28}$$

This probability distribution is updated as the robot moves and collecting more sensor data which allows it to continuously estimate both its map and its position within that map. By combining odometry or IMU data with sensor data SLAM enables real-time localization and mapping [8][37].

This chapter provided the theory behind the thesis, including all the sensors, the Ackermann steering model, filtering theory and software components. With this knowledge, the following chapter gives a system-level view of the AP4 which includes its hardware and software architecture.

3
System Overview

In this chapter, an overview of the AP4 will be described, including the previous years' contribution to the platform. The Hardware and software components will be explained to familiarize the reader with the platform.

3.1 Hardware design

The hardware in this thesis is based on the Ninebot Go-Kart platform, which serves as the foundation for the autonomous system. In addition to the base platform, hardware components were added during the two previous theses to enable autonomous functionality. The system now includes a High-Level Control (HLC) computer for data processing and a Hardware Interface Low-Level (HWI) computer that interfaces with the various hardware components. Furthermore, a CAN network enables communication between components, and multiple sensors are integrated to receive data from the surrounding environment. The communication between the HWI and HLC platforms are then transported over Ethernet through a router [4].

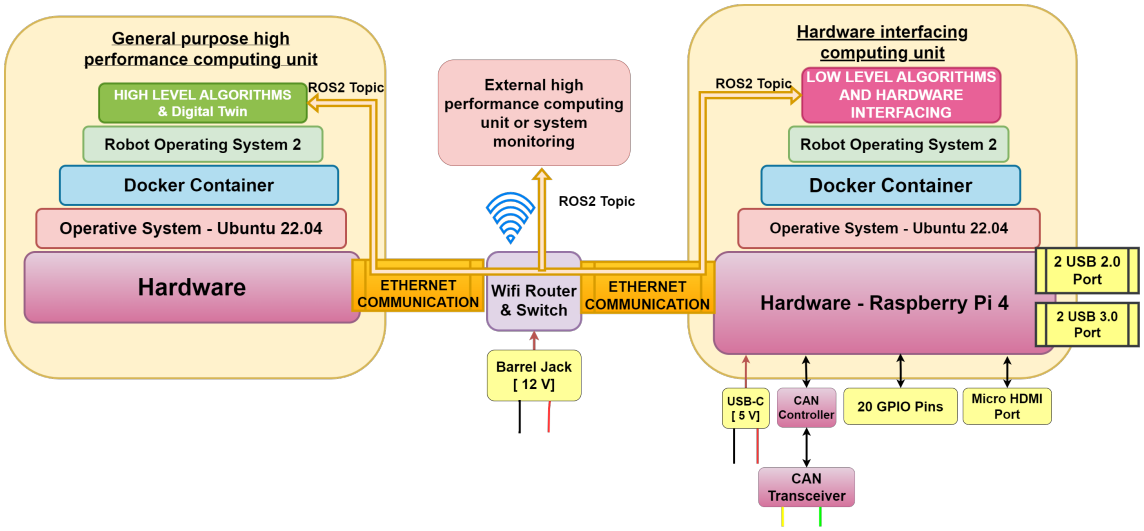


Figure 3.1: Hardware design of AP4 [4]. Reprinted with permission.

3.1.1 Ninebot Go-Kart

As mentioned previously, the autonomous platform is based on the Ninebot go-kart. The Ninebot utilizes a Segway unit that provides propulsion, enabling forward and

backward movement. Additionally, a go-kart kit is mounted on the Segway to enable steering and maneuvering by a driver[5]. To control both steering and propulsion, a centralized electrical/electronic (E/E) architecture has been implemented.

Figure 3.2: Image of the Ninebot Gokart AP4 [4]. Reprinted with permission.

3.1.2 Electrical Control Unit

To handle the the modular use of sensor and controllers, Electrical Control Units (ECUs) are implemented to handle specific tasks. These can then communicate to the HWI over a CAN network, enabling the Raspberry Pi to fetch and send information. There exists multiple ECUs for different tasks as controlling the steering heading and speed, as well as measuring the velocity of the Gokart. The generic structure for an ECU is as follows.

- MicroController The brain of the ECU is the STM32-F103C8T6 microcontroller, often called the Bluepill[38]. It has 37 GPIO pins and 128KB of flash memory, making it suitable for the small-scale control units. The Bluepill is supported by PlatformIO popular frameworks as Arduino can therefore be used for programming [39].

- DC-DC converter Each ECU is built with an LM2596 DC-DC converter that supports input voltages from 4.5-35 V and output voltages from 1.2-40 V[40]. This makes the ECUs compatible with components that have varying voltage specifications.

- CAN controller and transceiver: The MCP2515 CAN bus module and TJA1050 transceiver are implemented to allow SPI communication to the microcontroller[41].

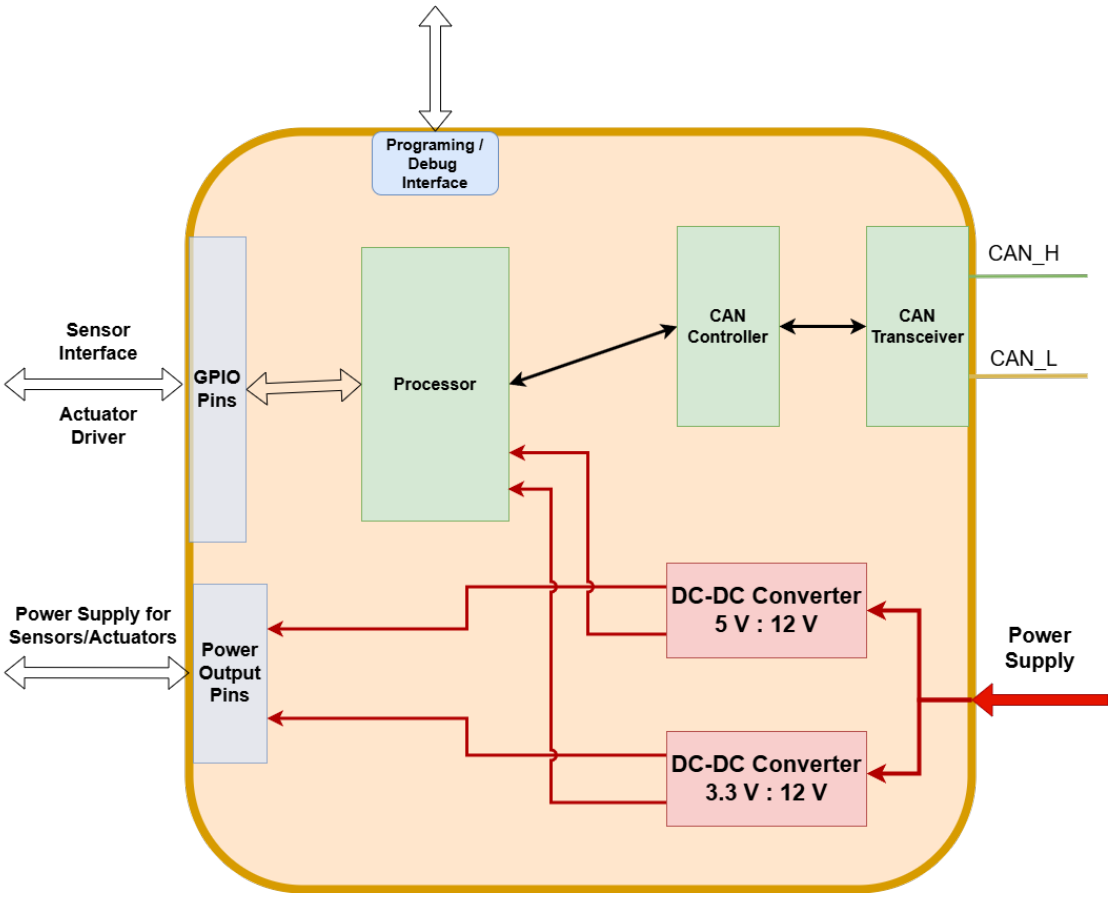


Figure 3.3: Generic ECU hardware setup [4]. Reprinted with permission.

Steering and Propulsion Control Unit

The Steering and Propulsion Control Unit (SPCU) is composed of a Kangaroo x2 Controller + Sabertooth 2x50 DC motor driver which is a closed-loop controller for the steering wheel motor. The current steering angle can be read from the kangaroo card. This is manually set to ±40 degrees[42][43].

The propulsion of the AP4 is controlled by sending analog voltages to the same connections the pedals are connected. This mimics the pressing of the pedals that the Ninebot is designed to operate from.

Speed Sensor Control Unit

The Speed Sensor Control Unit (SSCU) is composed of LM393 speed sensors, which are mounted on rotary plates around the wheels to measure the velocity of the GoKart[44]. This is done by measuring the frequency of pulses captured by the IR

sensor and converting that to a velocity according to the radius of the wheel. This is then averaged over the four wheels.

3.1.3 Hardware-Interface Low-Level Computer

The Hardware-Interface Low-Level Computer (HWI) is responsible for the communication between the ECUs, other sensors and the rest of the physical platform. This is needed since most regular computers do not support a GPIO interface and therefore a Raspberry Pi 4B was chosen. The Raspberry Pi is implemented with the RS485 CAN HAT, which allows the Raspberry Pi to use the CAN communication sent from the ECUs. The desired information is then published to ROS topics which allows different nodes to use the information asynchronously. The HWI is then connected to a router on the Gokart which allows different hardware to connect to the same network via Wifi or cable and communicate with the same ROS topics[45].

3.1.4 High-Level Control Computer

Since the HWI has restricted specifications, another computer can be connected to the same network and perform more computationally intensive operations on the data. This is done by the High-Level Control Computer (HLC). The HLC can be any computer that supports ROS Humble and can run Docker containers. It is on this computer that the self-driving algorithms this thesis will integrate will be developed.

3.1.5 Power module

To power the AP4, a built in power module was implemented. The module consists of a battery, a battery charger and a power supply unit. The battery used is a 12 V lead acid battery and the module is built with a battery out, power in and AP4 out sockets.

3.2 Software design

An autonomous platform like this needs a robust system design where different components and software should communicate with each other. Due to the different purposes of the different software and components, the system is therefore divided into three main software components that run on different machines. The software design is shown in Figure 3.4 where the three different components are a High-Level control computer, a Low-Level Hardware Interfacing computer and an Embedded Hardware Interfacing component.

The communication between these components is managed by ROS 2. ROS 2 enables data sharing between nodes and components by subscribing to or publishing different topics generated from the ROS 2 nodes and the sensors.

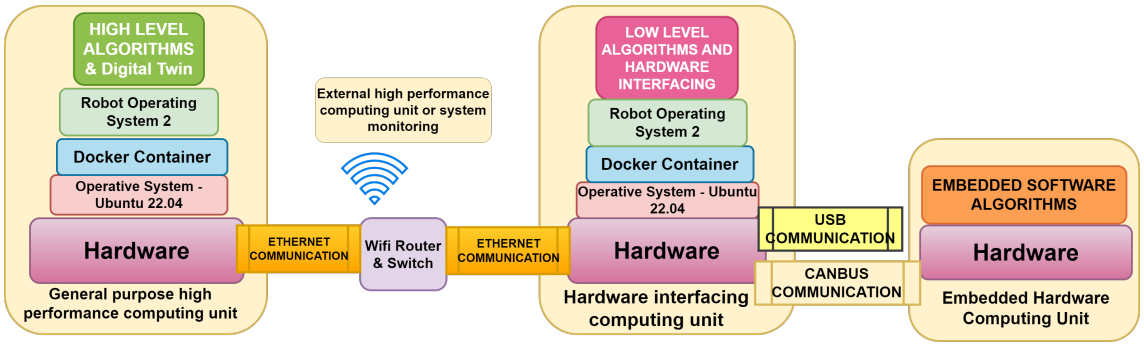


Figure 3.4: An overview of the system for the AP4 [4]. Reprinted with permission.

3.2.1 Embedded Hardware Computing units

The Embedded Hardware Computing Units include ECUs mentioned earlier and other sensors connected to the HWI. Since the specific hardware on these components vary, so does the software and is therefore designed depending on the specific sensor or ECU's purpose. This is then sent to the HWI via USB or through the CAN bus. For the externally connected sensors via USB, the embedded software is taken care for by the manufacturer's PCBs. For the ECUs, the Bluepills are programmed in C++ in the Arduino framework to format sensor readings or control signals.

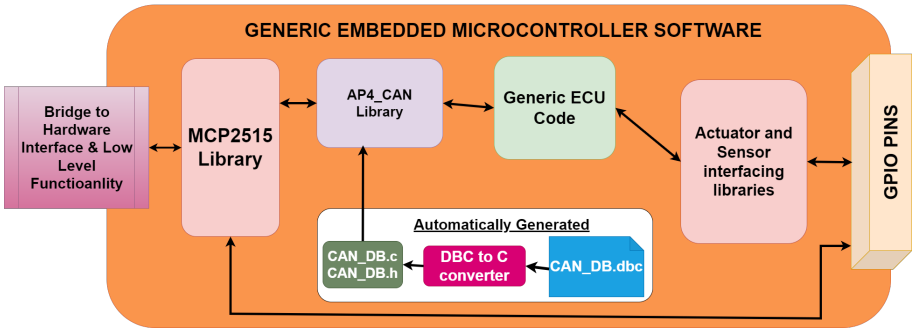


Figure 3.5: Generic ECU software [4]. Reprinted with permission.

3.2.2 Hardware Interface Low-Level Computer

The HWI serves as a bridge between the Embedded Hardware computing units and the High-Level Control Computer. Together with the Embedded Hardware Units, they make the bare bones of the platform, serving as the interface to correlate commands to actions and making it possible to drive and turn the go-kart by sending commands to it. It can be viewed as the nervous system of the platform, connecting everything. It does this by publishing the necessary data from the CAN bus and USBs to ROS 2 topics, making them accessible to computers that are connected to the same network. The Raspberry Pi runs on Ubuntu 22.04, which is the latest version that supports ROS Humble, and sets up a Docker container with the necessary dependencies. It also executes a bash script on startup which runs all the ROS 2 nodes that are needed to drive the go-kart.

3.2.3 High-Level control computer

The High-Level control computer (HLC) is responsible for processes that are more computationally heavy and that are not suited for the Raspberry Pi. Since the HLC is connected to the same network as the HWI, they share ROS 2 topics, making it possible to do the heavier computations on a different computer. The HLC also runs Ubuntu 22.04 to make it able to run ROS Humble.

To make it easier to include all dependencies, a docker container is set up. Since this is where the user is supposed to run the desired algorithms, no bash script is executed to run them on startup. Instead, they are to be launched manually. During 2024, in the Autonomous Driving via Imitation Learning in a Small-Scale Automotive Platform, their autonomous driving algorithms were run on the HLC using mostly Python.

3.2.4 ROS 2

The communication between the HWI and HLC is done through ROS 2. As mentioned earlier, it works by creating nodes that subscribe or publish to topics that both computers have access to. The nodes can then be run asynchronously and threads are assigned to them automatically by the ROS framework. This means no order has to be given to the processes, but they are instead executed when needed.

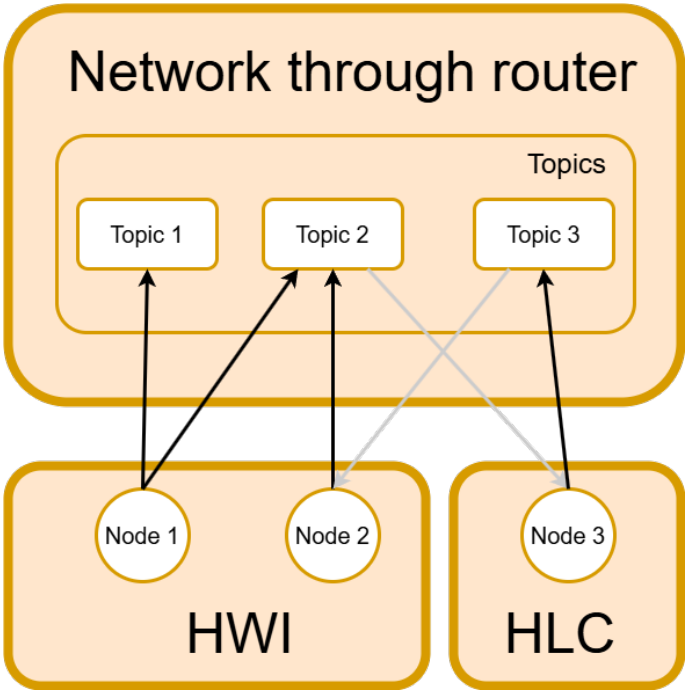


Figure 3.6: Example of ROS nodes communicating through Ethernet, black arrows: publishers, gray arrows: subscribers

This chapter outlined the hardware and software structure of the AP4 and described how the system components work together. With an understanding of the AP4

system architecture, the next chapter will dig into the methodology of this thesis and how the implementation of the simulation and a physical environment is done.

4

Methods

This section describes the design and procedures to implement SLAM on the AP4 project. The methodology followed an iterative approach, which allows for continuous testing of concepts and adjustments if necessary. This process ensures reproducibility by actively working with objectives throughout all stages of development. The repository containing the developed code from this thesis and the previous theses can be found in [46].

4.1 Simulation

To evaluate the objective, a simulation of the go-kart was created. This allowed for early testing and validation of the SLAM implementation in a controlled and repeatable setting. The simulation is built using Gazebo Fortress, which supports ROS communications via topics similar to the architecture used in the physical go-kart. The simulations provide a safe and cost-effective platform for development. This allows for rapid testing of models without the need for a facility that can accommodate the size of the go-kart.

4.1.1 Digital twin

A Digital Twin was developed to replicate the physical go-kart. The foundations for this already existed. However, several modifications were needed to make it function on Gazebo Fortress. The Digital Twin was designed using a URDF (Unified Robot Description Format) file, which defines each component of the go-kart with corresponding visual, inertial, collisional and joint properties. Once all parts were connected, an Ackermann steering controller was configured to translate ROS topics into correct body transformations. The controller ensures a forward velocity command results in the rear wheels accelerating, which in turn accelerates the entire vehicle. Additionally, a LiDAR sensor was integrated using a similar approach.

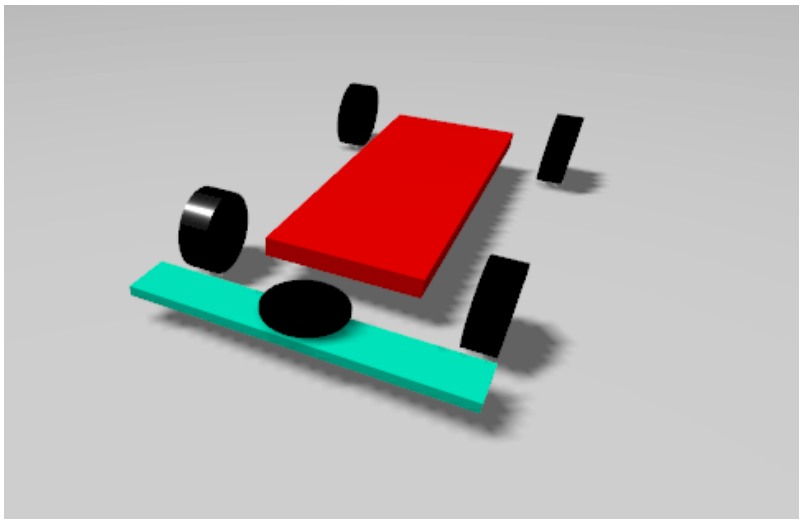


Figure 4.1: The AP4 digital twin

4.1.2 Replica of Gokartcentralen

To test the self-driving algorithms in a realistic setting, a digital replica of Gokartcentralen in Kungälv was created[47]. This was done by importing a map of the track into Blender and creating a 3D-mesh of the railings[48]. To ensure accurate dimensions, the width of different parts of the real track was measured. The measurements were then used to scale the 3D model appropriately, resulting in a full size virtual replica of the track at Gokartcentralen in Kungälv. In Figure 4.2 the replica of Gokartcentralen track is shown.

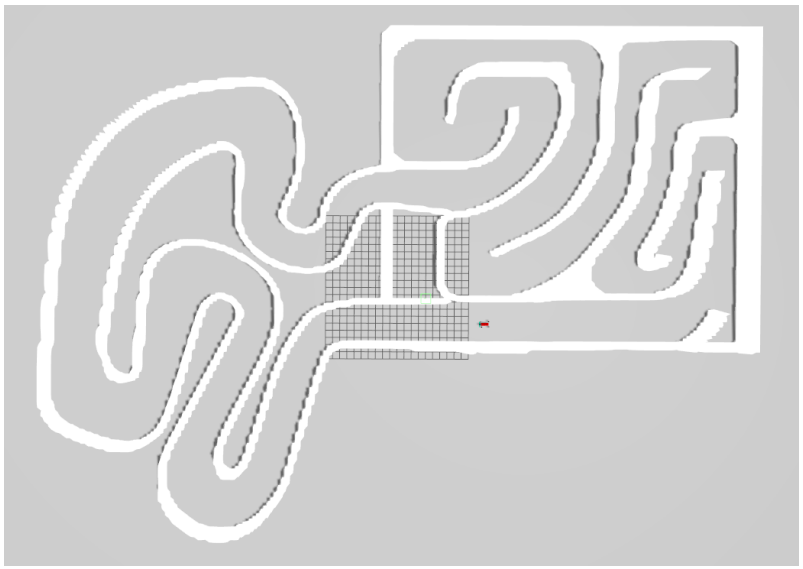


Figure 4.2: The 3D constructed replica of Gokartcentralen track used in Gazebo simulation

4.1.3 Nav2

As described in Chapter 2 Theory, Nav2 is used to navigate the go-kart through the track. Nav2 is a built-in framework in ROS 2 that enables autonomous driving for a robot in a known or unknown environment[25].

Planner Server
In order to achieve a path for the go-kart to navigate through, a path planner needs to be chosen for the Nav2 framework. Nav2 allows a choice between multiple sets of planners which are based on different algorithms and suitable for different kinds of robots. In Table 4.1 some of the planners that Nav2 offers are described and declared which type of robot they are suitable for.

Planner name	**Description**	**Robot type**
NavFn Planner	A navigation function using A* or Dijkstra's expansion, assumes 2D holonomic particle	Differential, Omnidirectional, Legged
SmacPlannerHybrid	A SE2 Hybrid-A* implementation using either Dubin or Reeds-shepp motion models with smoother and multi-resolution query. Cars, car-like, and ackermann vehicles. Kinematically feasible.	**Ackermann**, Differential, Omnidirectional, Legged
SmacPlanner2D	A 2D A* implementation using either 4 or 8 connected neighborhoods with smoother and multi-resolution query	Differential, Omnidirectional, Legged
SmacPlannerLattice	An implementation of State Lattice Planner using pre-generated minimum control sets for kinematically feasible planning with any type of vehicle imaginable. Includes generator script for Ackermann, diff, omni, and legged robots.	Differential, Omnidirectional, Ackermann, Legged, Arbitrary / Custom
ThetaStarPlanner	An implementation of Theta* using either 4 or 8 connected neighborhoods, assumes the robot as a 2D holonomic particle	Differential, Omnidirectional

Table 4.1: Nav2 planners with descriptions and robot type [25].

The planner server is the global planner and is the server that calculates and determines a valid long-term path for the go-kart. In this thesis the *SmacPlannerHybrid* planner has been used because it is most suitable for Ackermann drive robot's and is built on algorithms that enables smooth path planning in tricky and tight spaces which is an environment this project encounter while driving the go-kart on the track at Gokartcentralem. The planner is defined in the Nav2 configuration together with all constraints for the go-kart. Nav2 finally uses the configuration to find a valid global path for the go-kart.

Controller Server
The controller server is the local planner that determines the short-term path and enables dynamic obstacle avoidance that the global planner doesn't detect from the beginning. As well as that, the controller server also controls the wheels of the go-kart itself. In Table 4.2 some of the controllers that Nav2 offers are described and

declared for which type of robot they are suitable.

Controller name	Description	Robot type
DWB Controller	A highly configurable DWA implementation with plugin interfaces	Differential, Omnidirectional, Legged
TEB Controller	A MPC-like controller suitable for ackermann, differential, and holonomic robots.	**Ackermann**, Legged, Omnidirectional, Differential
Regulated Pure Pursuit	A service / industrial robot variation on the pure pursuit algorithm with adaptive features.	**Ackermann**, Legged, Differential
MPPI Controller	A predictive MPC controller with modular & custom cost functions that can accomplish many tasks.	Differential, Omni, **Ackermann**
Rotation Shim Controller	A "shim" controller to rotate to path heading before passing to main controller for tracking.	Differential, Omni, model rotate in place
Graceful Controller	A controller based on a pose-following control law to generate smooth trajectories.	Differential, Omni, Legged
Vector Pursuit Controller	A controller based on the vector pursuit algorithm useful for high speed accurate path tracking.	Differential, **Ackermann**, Legged

Table 4.2: Nav2 controllers with descriptions and robot type [25].

In this project the *DWBLocalPlanner* was used. Even though the *DWBLocalPlanner* is not suitable for Ackermann drive robots, the planner has shown promising results during the project and by testing different controllers, it has appeared that the *DWBLocalPlanner* works the best for the thesis purpose. The DWB controller uses the Dynamic Window Approach that uses all the Gokart's contraints such as acceleration and steering angle which, even though it is not suitable for Ackermann drive, makes it adaptable to the gokart and its kinematics. The controller is defined in the Nav2 configuration where all the constaints for the Gokart is defined as well. Nav2 uses the configuration and all the parameters to find the short-term paths in order to avoid obstacles during the drive around the Gokart track.

4.1.4 SLAM

To enable mapping of the environment and to determine the go-kart's position during the drive, SLAM is used. ROS 2 comes with a built-in toolbox that enables running SLAM inside the ROS 2 framework. In this project SLAM is used to map the unknown area using the LiDAR data in order to get a occupancy grid of the environment and let the robot know where the walls and obstacles are. SLAM is crucial for this project and especially for NAV2 to determine the global and local path for the go-kart by using the generated costmaps from SLAM and the LiDAR data. The SLAM toolbox for ROS 2 is run alongside Nav2 and gives information about the gokart's localization and the environment in terms of an occupancy grid. This includes where the walls and the obstacles are located.

4.1.5 Path selection

Once SLAM and Nav2 is configured and integrated with the AP4 and its sensors, the final step to enable autonomous navigation around the track is to select a path that the go-kart can follow in order to reach its goal. As shown in Figure 4.3, the go-kart is following the green path with a goal positioned far outside the track. Because of this, the go-kart always attempts to reach that goal during the run. When the gokart detects new obstacles and walls from the track, Nav2 will update the path to a new valid path that ensures obstacle avoidance. This procedure continues until the go-kart has driven one full lap around the track and Nav2 can no longer generate a new valid path to the goal. This approach enables the go-kart to autonomously drive around the entire track by simply setting an initial goal path at the beginning of the run.

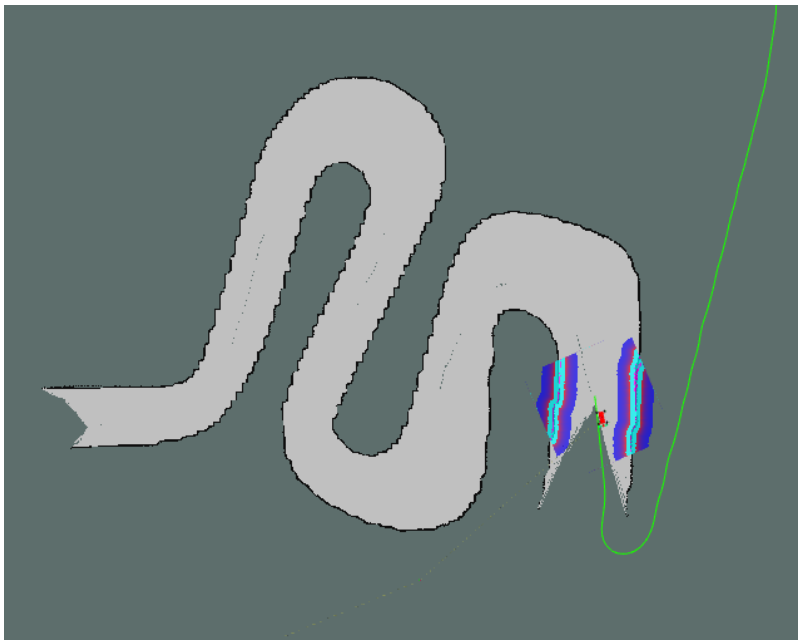


Figure 4.3: Image of the go-kart autonomously navigating and mapping the go-kart track in simulation

4.1.6 Path evaluation

To evaluate how well the go-kart is following the planned path, some form of evaluation criteria was needed. Since the planned path is updated 20 times per second, a way to recalculate the deviation from it at every time step was required. There are several ways to do this, including heading error, which tracks the error in yaw direction, time deviation from an estimated time of arrival and the cross track error (CTE), which calculates the perpendicular distance from the go-kart's current position to the nearest point on the planned path. Since the purpose of this thesis was to use SLAM to map the go-kart's surroundings and not to find the quickest way through a track, it was decided to use the CTE to evaluate how far from the planned path the go-kart is deviating since this results in a measurement reflecting how well the go-kart is following its path[49].

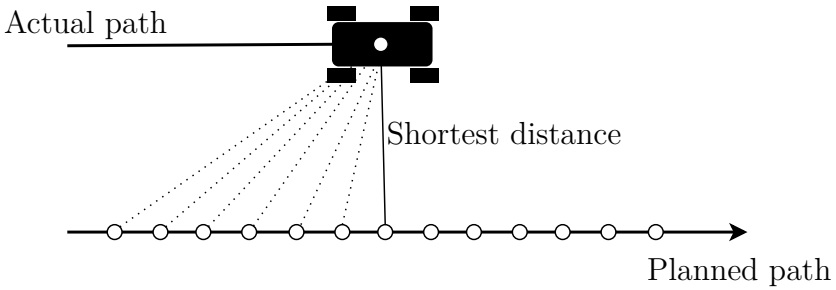


Figure 4.4: Cross Track Error concept

4.2 Implementation of hardware

To achieve the goal of the project, some implementation of new hardware was needed. This chapter include the integration of the new hardware, as well as the changes made to previously existing sensors. The reasoning behind the choice of the different sensors will be explained, as well as the different positioning and calibration of them.

4.2.1 LiDAR

To map the surroundings of the go-kart, a point cloud is needed. This can be collected by the use of a LiDAR. Therefore, the SLAMTEC RP LiDAR A1 was chosen to be integrated into the AP4. The SLAMTEC RP LiDAR A1 is a 360° 2D LiDAR with a maximum scan distance of 12 m. It can be configured with a variable sample rate of a maximum of 8000 points per time sample and a varying turning frequency from 5- 10 Hz[50]. This combined with its built-in USB interface and open source Software Development Kit (SDK) makes it a perfect fit for ROS 2 integration. Different mounting positions were tested for the LiDAR. First, a variable x- and z-direction mounting plate was prototyped to make its position tunable during testing. During testing, however, the lowest position in the z-direction was too high to capture all railings of the track at Gokartcentralen. Therefore, it was decided to mount the LiDAR on the front wing of the go-kart to make sure it is positioned low enough to capture all necessary obstacles. With the LiDAR mounted on the front wing, it was configured to have field of view (FOV) of 360° with 360 points per sample and a turning frequency of 10 hz. The LiDAR is configured to only measure objects that are 1-12 meters away from it. This means that even though the LiDAR is positioned low at the go-kart's front wing and will be blocked by the go-kart itself from behind. The LiDAR will not map those objects because it doesn't take objects that are closer than one meter into account. The LiDAR is connected via USB to the Raspberry Pi and the ROS 2 nodes was integrated in the HWI.

Figure 4.5: Slamtec RP LIDAR A1. Image source:[50]

4.2.2 Speed sensors

When the point cloud is generated, the movement of the go-kart needs to be known to be able to accurately merge newly gathered point clouds into a single map. Therefore the odometry of the go-kart needs to be calculated and transformed to the world frame. In the previous iterations of the AP4 project, speed sensors were integrated to calculate the speed of the go-kart. These are composed of the LM393 IR speed sensor that measures a rotary encoder wheel. This is then processed by a STM32 Bluepill microcontroller in the Speed Sensor ECU. It works by measuring the amount of edges passing through the IR sensor from the encoder wheel during a set time period. This is then sent to the HWI's ROS 2 network and is converted to the distance traveled since the edges per revolution and circumference of the wheels are known.

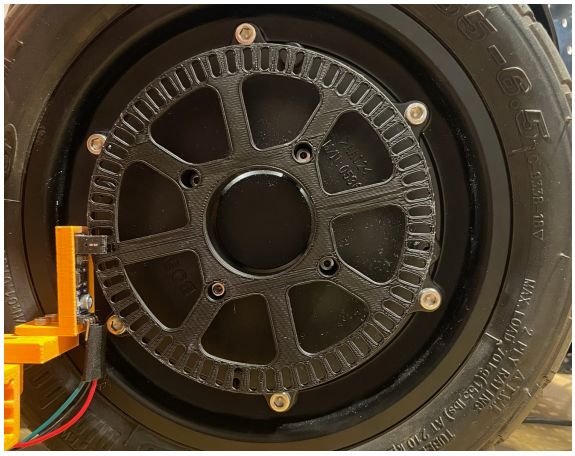


Figure 4.6: Wheel encoder with speed sensor

4.2.3 IMU

With the LiDAR and speed sensors integrated, the necessary data for SLAM is gathered, however, a more accurate estimation of the go-kart's position could be achieved. This is done by the inclusion of the data from the accelerometers and gyroscope on the IMU. In the previous year's master's thesis, a Luxonis OAK-D stereoscopic camera was integrated which includes a built-in IMU that measures accelerations in x-, y- and z-directions and can be seen in Figure 4.7 [3][51]. This data can be fused with the speed sensors, resulting in greater accuracy of the go-kart's position. Since the IMU is capable of measuring the acceleration, it can also determine if the go-kart is moving forward or backward. The camera is mounted on the front plate of the go-kart with a joint around the y-axis to allow for pitch rotation. Since this allows for different setups, an IMU calibration is executed on start-up to calculate the mean and variances of the acceleration data and center it around zero.

Figure 4.7: Luxonis OAK-D stereoscopic camera with built-in IMU. Image source:[51]

4.3 Implementation to Physical Environment

When the simulation was tested and the new hardware integrated, it was time to combine everything in the physical environment. However, some extra steps needed to be added when moving from simulation to practice.

4.3.1 Calculating mean and variance of measurements

When using sensors to estimate the go-kart's position, it is important to validate how accurate they are. Therefore, to measure the accuracy of the sensors, some tests had to be set up to evaluate them.

IMU

When collecting and analyzing the data from the IMU, an unexpected offset was observed. With the camera lying flat on the table and the effect of gravity taken in consideration, an offset of approximately $2m/s^2$ was detected in the x-direction.

This can be seen in Figure 4.8, where the acceleration in the x-direction should have been close to -9.82.

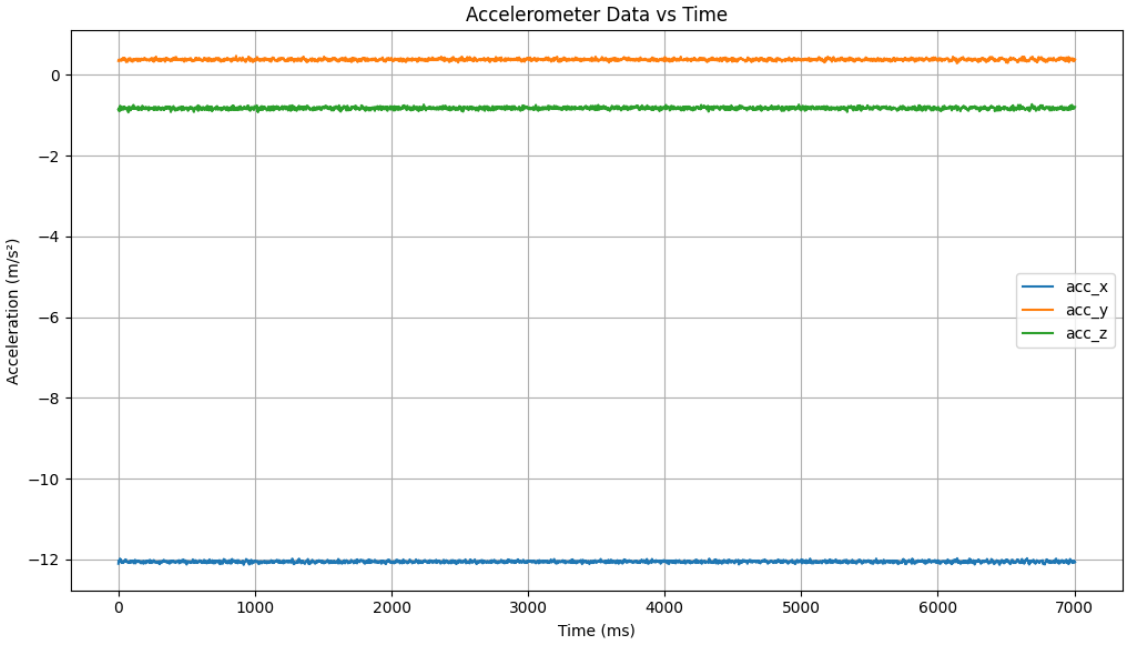


Figure 4.8: Figure of the raw data from the accelerometer

Since it is also possible to change the pitch of the camera on its mounting plate, a script was implemented to calculate the mean and variance of the acceleration and normalize everything around zero on start-up. This is done over five seconds, and once normalized, the individual means and variances are sent to the /imu topic to give accurate updates regardless of how the camera is mounted.

Speed sensor

To calculate the mean and variance of the speed sensors, an attachment was added to the wheels to visualize when a full revolution was completed. Then the wheels were rotated two full revolutions and were compared to the distance of two times the circumference. The results can be seen below:

$$\text{Test with 2 revolutions} = 1.56 \text{ m} \tag{4.1}$$

$$[1,58 \;\; 1,61 \;\; 1,58 \;\; 1,71 \;\; 1,63 \;\; 1,59 \;\; 1,57 \;\; 1,57 \;\; 1,59 \;\; 1,55] \tag{4.2}$$

$$Mean : 1.598m \qquad Var : 0.001836m \tag{4.3}$$

4.3.2 Odometry calculation

When simulating the go-kart in Gazebo, the Ackermann steering controller handles the conversion between the velocity commands into corresponding x- and y-positions in the world frame. Outside of the simulation environment, however, this needed to

be implemented manually.

To achieve this, the odometry equations 2.11-2.15 were used to convert velocity and steering data from the robot frame into x- and y-position in the world frame. This computation then updates the /odom topic with the necessary pose and velocity data, which is later visualized in Rviz.

During testing of the go-kart, it was noticed that the vehicle had an asymmetrical turning behavior when turning fully to the right versus fully to the left. In the reference code, this was set to ±40 degrees for both left and right turns, however, this did not reflect the actual limits of the Ackermann system. Through physical measurements and visualization, the turning angle was observed to be approximately -25 to 15 degrees, with negative degrees referencing a turn to the left. The code was updated to limit the turning angles accordingly. However, if this discrepancy isn't tuned appropriately, it would potentially cause more problems when calculating the odometry on a greater scale. To better estimate the actual turning angles, more testing was conducted to incorporate this discrepancy.

The testing was done by steering the go-kart at its maximum turning angle and making a 90° turn. The resulting motion was then visualized in RViz to identify discrepancies between the actual and calculated paths. When this was done, it could be seen that when the go-kart held full turning to the left and did a 90° turn, it registered a greater turn in RViz. This indicated that the odometry calculations underestimated the turning radius, meaning the turning angle used to estimate the go-kart's path was too large. To correct this, the maximum turning angle was decreased incrementally until the visualized path in RViz closely matched the actual trajectory of the go-kart. This was done for both the left and right turns and the final calibrated values to achieve the alignment can be seen below.

$$\text{Maximum left turning angle} = -30^\circ \tag{4.4}$$

$$\text{Maximum right turning angle} = 20^\circ \tag{4.5}$$

Robot Localization

When the odometry was calculated, it could be used to estimate the go-kart's position. To better this estimation, ROS 2's `robot_localization` package was used. This is a collection of state estimation nodes. The package allows the integration of multiple sensor readings and the configuration of filters to improve the accuracy of the robot's estimated state. For this project, an EKF was chosen with 15 state variables. A corresponding measurement configuration was then created, specifying which sensor readings should be mapped to which state variables. The EKF could also be configured with a process covariance matrix to better reflect confidence levels of the motion model. The following state variables are set in the robot's frame [52].

$$\begin{bmatrix} X & Y & Z & \phi & \theta & \psi & \dot{X} & \dot{Y} & \dot{Z} & \dot{\phi} & \dot{\theta} & \dot{\psi} & \ddot{X} & \ddot{Y} & \ddot{Z} \end{bmatrix}^T \tag{4.6}$$

To use the sensor readings, you define which state variables are available from each sensor topic. The `robot_localization` node then runs the EKF algorithms using the provided measurements. For each input topic, a configuration is specified to indicate which state variables are to be used. This is represented as a list of boolean values. An example for the /odom topic can be seen below, where T=True and F=False signaling which variables to extract:

$$\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} \end{bmatrix}^T \tag{4.7}$$

meaning it will use the x- and y-positions as well as the velocity in the x-direction for prediction and estimation. If more sensors are available, you configure them in the same way, setting True for the measurements you want to use and False for the others. This way, you can fuse any number of sensors, including multiples of the same kind.

This was also done with the /imu topic, however, with the following list of boolean values to use the angular velocities and the linear accelerations:

$$\begin{bmatrix} \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} \end{bmatrix}^T \tag{4.8}$$

During evaluation of both the simulation and the physical environment, two different EKF configurations were tested. The first uses only the /odom topic and the second includes both the /odom and /imu topics. For the EKF with both /odom and /imu data, a process covariance matrix was also added to account for system uncertainties. This was a diagonal matrix with the diagonal elements set to the following values:

$$\begin{bmatrix} 1.0 & 1.0 & 0.001 & 0.3 & 0.3 & 0.01 & 0.5 & 0.5 & 0.1 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \end{bmatrix} \tag{4.9}$$

This chapter explained the implementation steps of the hardware and software in both the simulated and the physical environment. With the setup and method clarified, the next chapter evaluates the performance of the AP4 in the simulation and the physical environment.

5

Results

In this chapter, the results of the project are presented based on the previous chapters and how the implementation of new hardware and new AD algorithms worked out for the Autonomous Platform. The project was divided into two main parts: simulation and real-life testing of the Autonomous Platform. The results in this chapter are based on the go-kart trying to drive one lap around the go-kart track at Gokartcentralen in Kungälv, or driving around the track inside a simulation. In both cases, the go-kart has been tested with and without an integrated IMU.

5.1 Simulation

In this section the outcome and the results for the implemented hardware and algorithms on the simulation are presented.

5.1.1 Evaluation of the drive

By using the implemented algorithms and hardware in the Gazebo simulation, the following results has been generated. This was done by driving the go-kart around the simulated go-kart track using the Gokartcentralen track replica for Gazebo. The results show that the go-kart manages to drive one lap around the track without any collisions or problems with tight turns both with and without IMU. In Figure 5.1 and 5.3, the results for both cases are shown. In addition to that, the mapped areas from the simulations are shown in Figure 5.2 and 5.4. for comparison between the planned and actual path relative to the mapped environment from the LiDAR.

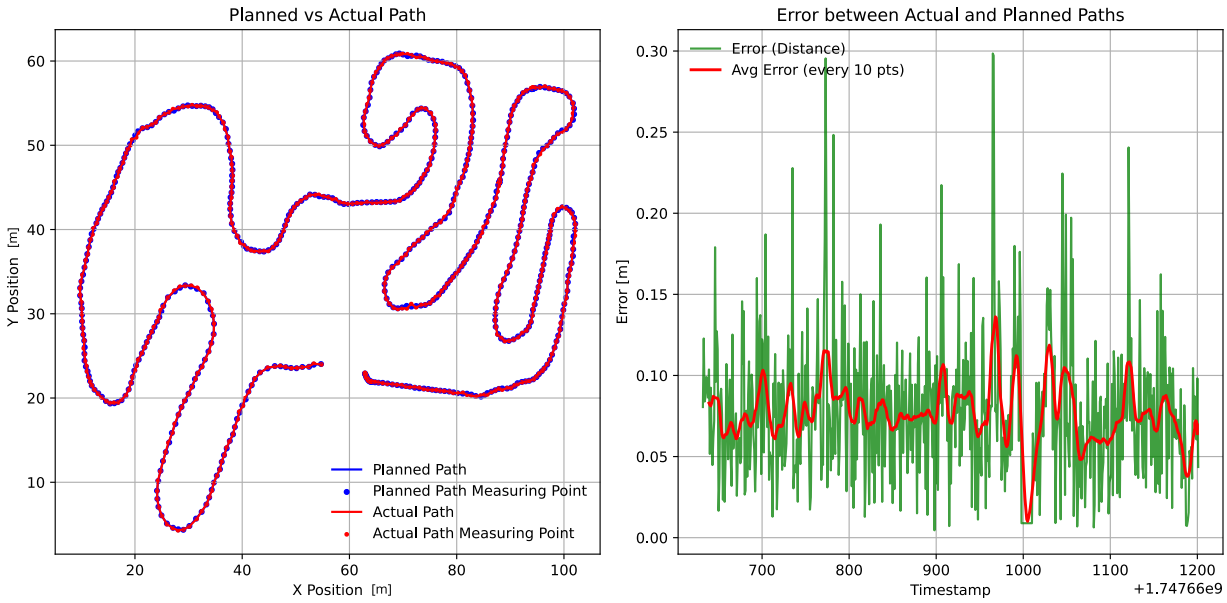


Figure 5.1: Plot over results for the go-kart driving a lap around the track in simulation without IMU

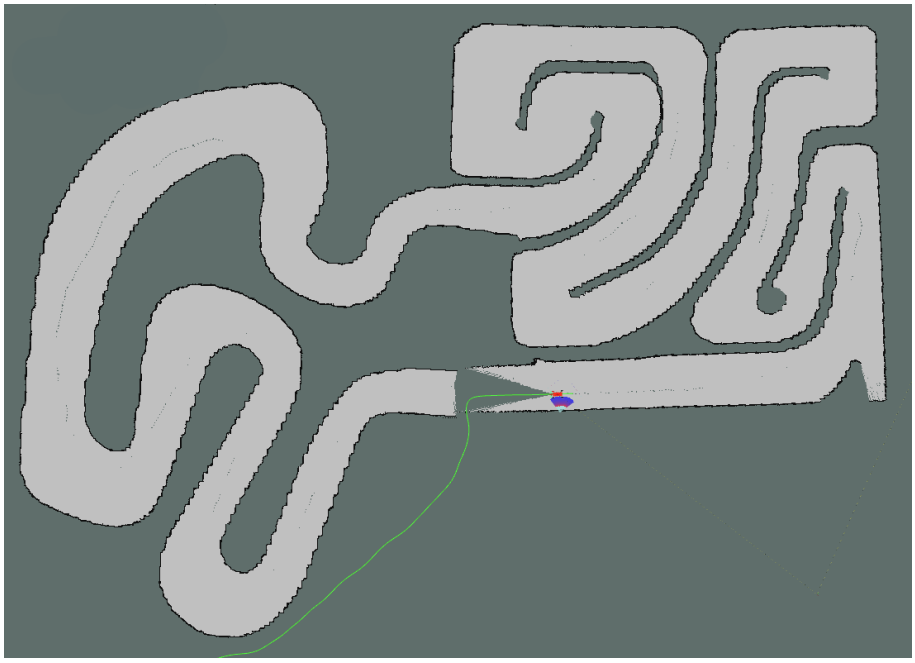


Figure 5.2: Image of the mapped environment in simulation without IMU

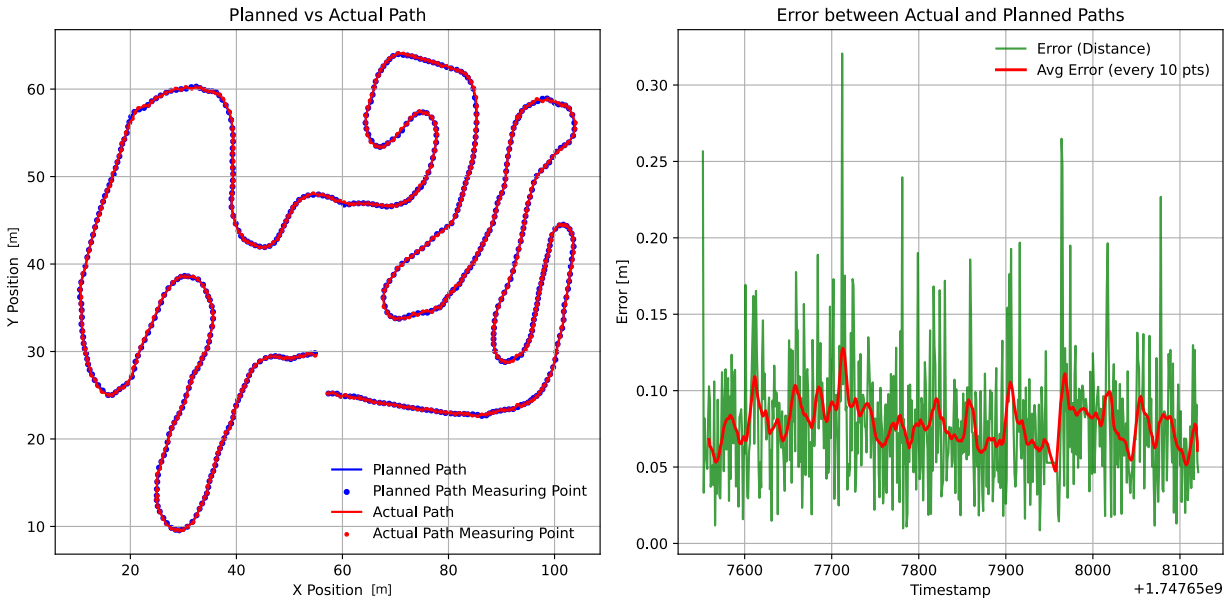


Figure 5.3: Plot over results for the go-kart driving a lap around the track in simulation with IMU

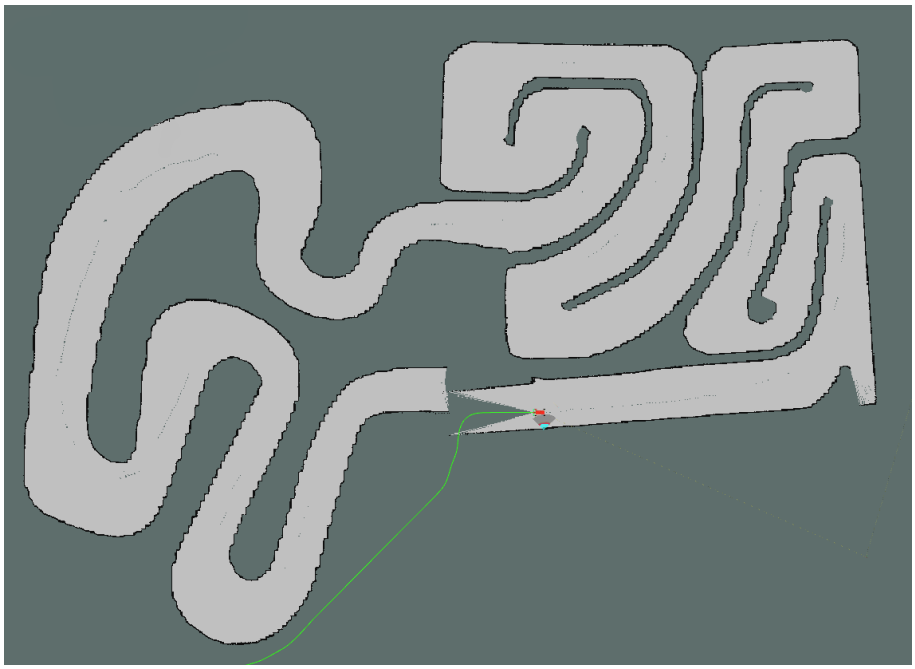


Figure 5.4: Image of the mapped environment in simulation with IMU

5.2 Physical environment

In this section, the results for testing and driving the Autonomous Platform in a real physical environment are presented. The go-kart was tested on the track in Gokartcentralen in Kungälv, and from that, it was evaluated on how good it

managed to drive around the track and how much it differed from the planned path from Nav2.

5.2.1 Evaluation of the drive

The evaluation of the go-kart performance in the physical environment focused on its ability to follow the track at Gokartcentralen by following the planned path generated by Nav2. The go-kart was equipped with the same hardware and software as used in the simulation. The results shown in Figure 5.5 and 5.7 show that the gokart doesn't manage to drive one full lap around the track, but generally follows the planned path given from Nav2. In addition to that, the mapped areas from the physical testing are shown in Figure 5.6 and 5.8 for comparison between the planned and actual path relative to the mapped environment from the LiDAR.

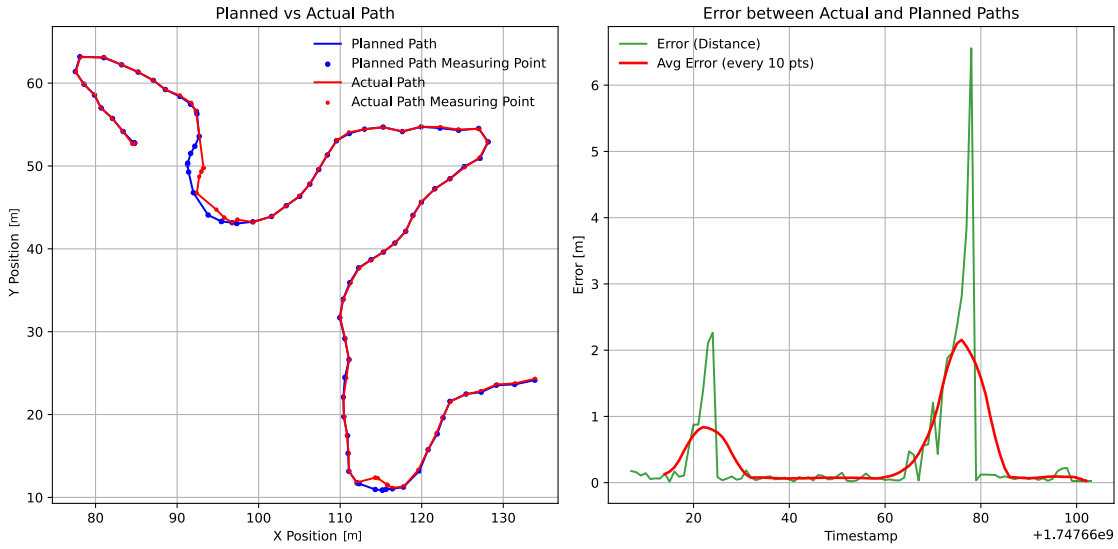
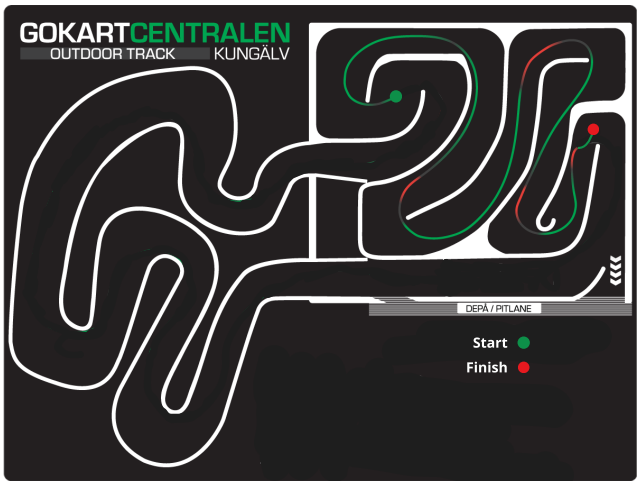


Figure 5.5: Plot over results for the go-kart driving around the track in the physical environment without IMU

(a) Image of the mapped environment in the physical environment without IMU



(b) Image of the actual path the go-kart takes in the physical environment

Figure 5.6: A comparison between the mapped area and the actual path the go-kart is taking

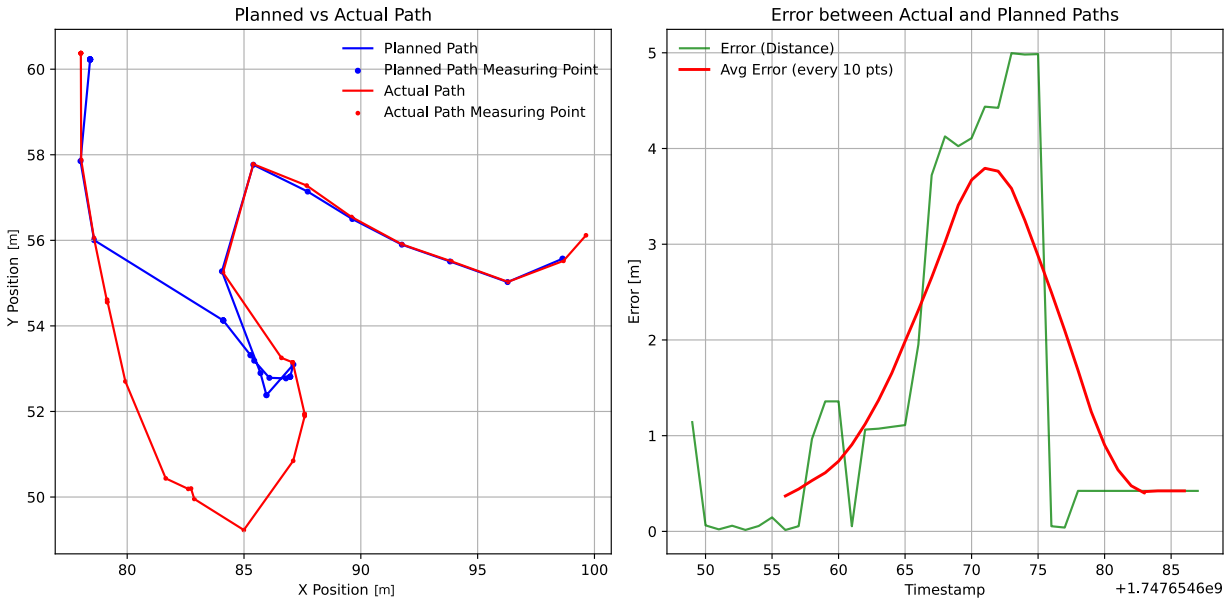
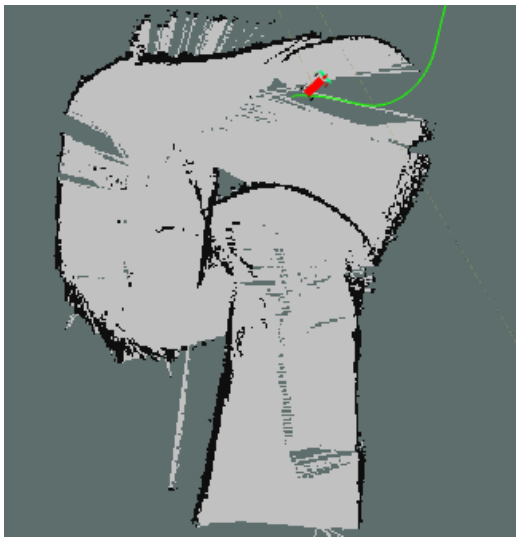
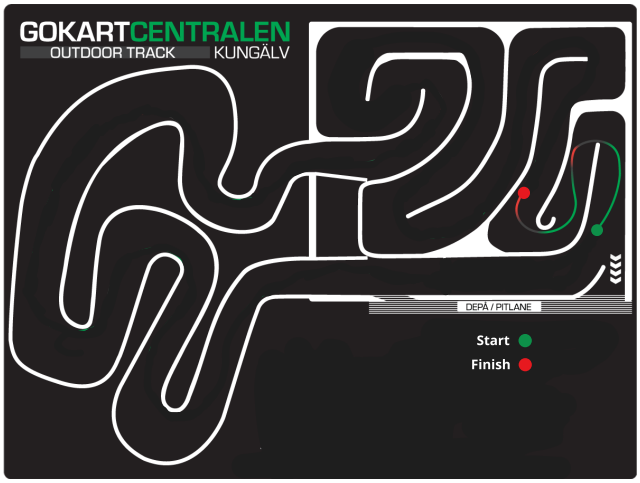


Figure 5.7: Plot over results for the go-kart driving a lap around the track in the physical environment with IMU



(a) Image of the mapped environment in the physical environment with IMU



(b) Image of the actual path the go-kart takes in the physical environment

Figure 5.8: A comparison between the mapped area and the actual path the go-kart is taking

From these four test cases, the Root Mean Square Error (RMSE) has been calculated for each test. These are shown in Table 5.1 below.

Test case	**RMSE**
Simulation without IMU	0.0769 m
Simulation with IMU	0.0786 m
Physical environment without IMU	0.3981 m
Physical environment with IMU	1.2251 m

Table 5.1: Table of Root Mean Square Error for the different test cases

The results presented in this chapter show the performance of the AP4 under various test conditions. The outcomes provide a foundation for the reflection, discussion of the results as well as the further improvements.

6
Discussion

This chapter discusses the main results and findings from the thesis project. It looks at how well the autonomous driving system worked in both the simulation and the physical tests. The chapter also explains what challenges that were faced during the project, how well the methods worked, and what problems were found. In addition, it highlights what was learned from the project and suggests ideas for how the system can be improved in the future to work better in real-life situations.

6.1 Performance of simulation

By looking at the results from the go-kart driving one lap around the track, it can be concluded that the go-kart manages to autonomously drive around the track very well and follows the planned path with an average error of about 10 cm during the drive. The reason why the simulation performs this well is mainly because the go-kart's odometry is updated by the exact position and orientation from the simulation. This is not the case for the physical environment, where these need to be estimated from sensors. By knowing the exact position of the go-kart in every timestamp, the mapping of the environment will be more accurate and the controller can provide the go-kart with valid paths during the run without any concerns. Another thing that makes the implementation better in the simulation than in the physical environment is the LiDAR and its laser scans of the environment. In the simulation, there is no sensor noise that disturbs the data given to SLAM. With a fully correct map of the environment without mismatches, the path planning will be more or less perfect.

Although there are some larger errors during the run, which can be seen in the right plot in Figure 5.1. These errors can occur due to several reasons. One reason can be that the path planner finds a valid path that doesn't fit the go-kart kinematics regarding the steering angle and velocity. Because of this, the controller needs to deviate from the planned path to continue the go-kart's run around the track. When the go-kart is deviating more from the planned path, the error increases in that particular timestamp until a new valid path is being generated. Another factor that can contribute to the larger errors is a timestamp delay. Due to the use of an asynchronous SLAM algorithm, any delay can cause the system to get out of sync. This may result in a mismatch between the timestamps of the actual path and the planned path. As a result, when calculating the error, wrong data points might be compared from the actual path to the planned path. This can inflate the measured

error, making it appear larger than it is.

One thing to reflect on in the simulation is that the new generated paths from the path planner always has their origin in the center of the robot, i.e., the point where the go-kart's actual position is measured from. In that case, for every new generated path, the first measurement point will be exact the same for the actual and the planned path. This might make the results for the go-kart's path following ability slightly biased and not represent how well the go-kart is following the planned path in the long term, since it is always updating from where it is. However, if the go-kart deviates from its planned path, this would still be captured in between updates.

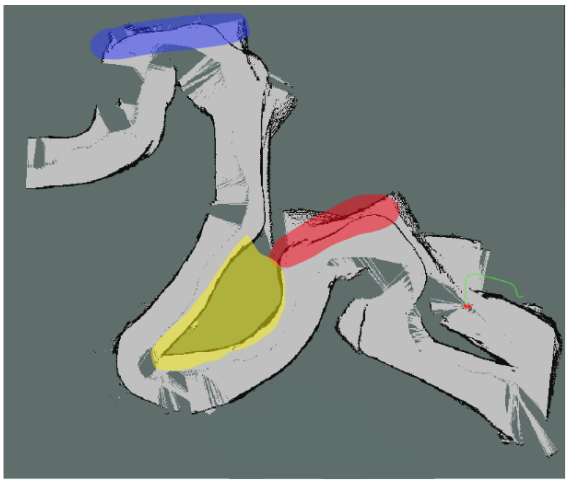
When comparing the EKF with and without IMU, i.e., figure 5.1 and 5.3, it can be seen that there is slightly more error in the IMU case, Figure 5.3. There can be several reasons for this. First of all, the IMU is simulated with added noise to better match a real-life scenario. This will introduce some uncertainty to the estimations, leading to greater error. When comparing the Planned vs Actual Path and the go-kart track, it can also be seen that instead of going in a straight line at the end to the finish line, both the x- and y-position are increasing. This could be because of drift, which is typically for an IMU, since the additional positional data will come from a double integration of the acceleration of the go-kart.

6.2 Performance of physical environment

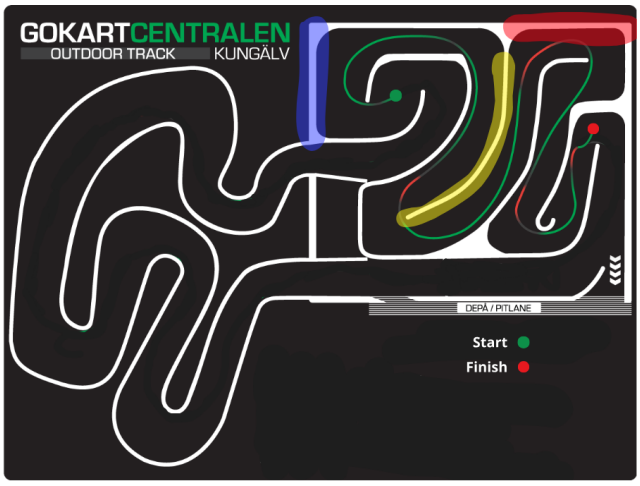
By looking at the results from the physical environment where the go-kart attempts to drive one lap around the track, it is clear that the performance does not reach the same level as in the simulation. In the real-life test, the go-kart only manages to drive half a lap before the go-kart fails to continue. Apart from the go-kart not being able to drive a full lap around the track, the go-kart follows the planned path well with an average error and deviation from the planned path of about 15 cm. As seen in Figure 5.5, some large errors differ a lot from the rest of the result data. This occurs due to several errors and challenges when testing in a real-world scenario. The main reason is because of the difference in maximum turning angle when turning to the left versus turning to the right. Since the go-kart has a maximum turning angle of 15° to the right, it struggles with taking sharp right corners. The spikes in the CTE are from some of these right corners where the go-kart needed help maneuvering, meaning it deviated from its planned path. Another reason why the real environment does not perform as well as the simulation is due to inaccuracies in the odometry position updates. Especially, we noticed that when the go-kart turned 180° in the physical environment, it only registered about a 150° turn in SLAM

This means that the map that is created will not match the actual track at Gokart-centralen. During small runs, this has a small likelihood of affecting the performance of the AD. However, for larger tracks with many turns, the created map might turn back on itself and overwrite parts it should not, since this error in the running angles accumulates over time. The problem this causes and how it occurs can be seen in Figure 6.1, which is an analysis of Figure 5.6. The misalignment is easiest

visualized by the straight walls on the track marked in blue and red, which should be perpendicular but have almost become parallel. It is also possible to see that it sometimes has difficulties matching walls that are the same and wrongly interprets large spaces between them, which is marked in yellow. If the walls are not matched together, the map might box itself in, and when that is the case, the go-kart can't find a valid path to move forward and will come to a halt.



(a) Image of the mapped environment in the physical environment without IMU



(b) Image of the actual path the go-kart takes in the physical environment

Figure 6.1: A comparison between the mapped area and the actual path the go-kart is taking, with highlights marking where they should be similar

When comparing the two EKFs in the physical environment, it can be noted that the case with the IMU is performing much worse. Intuitively, adding more sensors to the EKF should improve the estimation. However, if the IMU data is not correctly calibrated, it might make it worse. During the live visualization in RViz, this could be seen where the heading of the go-kart changed when the IMU data was added. When this happened, the odometry and imu sensor fusion got out of sync, and the go-kart moved in its lateral direction instead of forward. This also contributed to conflicts in the mapping since the /scan topic is referenced in the driving direction of the go-kart.

6.2.1 Odometry performance

In contrast to the simulation, where the go-kart's position is always known from Gazebo, the position in the physical environment is estimated using wheel encoders, which come with cumulative errors. This causes a difference between the real position of the go-kart and the position estimated by the SLAM algorithm, leading to an inaccurate map of the environment. As a result of this, the path planner is no longer aligned with the actual environment, making it difficult for the controller to drive the go-kart along a valid path. This issue becomes more critical the longer

the go-kart drives, since small errors accumulate over time and eventually cause a total mismatch between the real map and the map that SLAM thinks the go-kart has taken.

Another reason for the decreased performance in the physical environment is the go-kart's steering ability. The steering angle, especially in right turns, is very small, making the turning radius large. This makes it very difficult for the go-kart to make sharp right turns and stay within the track and follow the planned path. As the path planner assumes a certain steering ability, the mismatch between expected and actual steering behavior will result in paths that are not physically possible for the go-kart to follow.

When including the IMU in the positioning estimation, both the simulated and real-world estimations become worse, but for different reasons. In the simulated environment, the ground-truth data of the position, velocity and heading of the go-kart is available and used in the /odom topic. Adding additional acceleration and angular velocity measurements from the IMU with simulated noise will make the accuracy worse, however, it should reflect real-life scenarios better.

For the physical environment, the worse performance is not as easily explained. In theory, adding more sensor data should improve the estimation. However, when analyzing the results, unexpected behavior was noticed. When visualizing the go-kart in RViz during the testing, the heading of the go-kart varied more between time steps. This instability is probably because of the angular velocity readings from the IMU's gyroscope. When testing the IMU, its accelerometer was found to be miscalibrated, which led to its recalibration script during startup. Given this issue, it is plausible that the gyroscope also contains undetected errors, contributing to the observed inconsistencies in the heading orientation.

6.2.2 Sensor performance

The LiDAR on the physical go-kart is mounted low on the front wing of the go-kart. Because of its position, it can't detect obstacles behind itself. This causes the go-kart to only be able to match scans that are in front of it. If the go-kart doesn't have a full view of what is happening around itself all the time, it could cause the SLAM algorithm to have a problem matching the scan data to the environment and thereby updating the position of the go-kart in the map. This causes the position in SLAM to be slightly wrong compared to the real environment.

Furthermore, a phenomenon that occurs during driving in the physical environment is the similarity of the track and its obstacles. Since a go-kart track only includes similar walls and no other obstacles, the environment is very similar over the whole track. Due to this, the scan data along the track is very similar, causing the SLAM algorithm to have a problem deciding whether a scan data point is the same point as before or a new one. This can lead to a mismatch between the map that SLAM is providing and the actual physical environment, making the localization of the

go-kart wrong.

The LiDAR sensor in the real environment introduces another error. Unlike in simulation, where the laser scans are perfect without noise, the real LiDAR comes with noise and disturbances such as reflections or changes in the environment. This noise can lead to incorrect scan data being sent to the SLAM algorithm, which can result in mismatches in the map. When the map does not represent the actual environment, the path planning will also be wrong, which decreases the go-kart's ability to navigate along the track.

6.3 Conclusion

The objective of this thesis was to implement improved AD algorithms to the AP4 by integrating a LiDAR into the existing platform and develop new AD algorithms within the ROS 2 environment.

From this project, it can be concluded that the modularity of the autonomous platform makes the integration of new sensors, such as a LiDAR, straightforward and easy to integrate within the ROS 2 network.

Moreover, it can also be concluded that SLAM is an application that is conceptually suitable for the purpose, AD in a small-scale autonomous platform, and performed well in the simulation. However, when testing in the physical environment, it showed some limitations and issues. The sensor data, especially from the wheel encoders and steering angle sensors, were not accurate enough. To improve how well SLAM works in real-world conditions more accurate sensor data are needed.

Regarding the LiDAR-IMU sensor fusion, it can be concluded that it worked well in the simulation and showed that this method can help to improve the go-kart's localization. However, to make it work well in the physical environment, the IMU and the Extended Kalman Filter needs to be carefully adjusted and fine-tuned in order to achieve the expected results.

Furthermore, the integration of SLAM with the built-in framework Nav2 for path planning showed that it can improve the autonomous navigation of the go-kart through the track. Even if the mapped environment provided by SLAM doesn't become representative of the actual environment, the system was still able to avoid obstacles using the real LiDAR data and the local costmap provided by Nav2. This can be proof that the algorithms are adaptable in dynamic and unknown environments.

In summary, this thesis shows that using SLAM with LiDAR and sensor fusion can work well for self-driving on a small-scale platform. The results in the simulations is good, but to get the same performance in the real world, the sensors need to be better and more well-tuned in order to achieve an improved result in the physical environment.

6.4 Future work

This project is built on a modular autonomous platform which enables simple development of new hardware and software for the AP4, such as adding new sensors and developing new AD algorithms. In this section, some of the possible future improvements and work are discussed.

6.4.1 Localization

During this thesis, it has been concluded that the main cause why the go-kart didn't manage to drive as well in the physical environment as in the simulation is because of the go-kart's positioning system. In the simulation, the ground truth is available. In real life, however, the position and orientation vectors need to be estimated at every timestamp. Some relevant and crucial improvement approaches are stated in this section to achieve a more accurate vector update.

Odometry calculation

As described earlier in this chapter, one of the main problems why the go-kart doesn't drive as well in the physical test as in the simulations is that the steering angle of the go-kart is too small for some of the curves on the track. For future work with the AP4, it could be beneficial to increase the steering angle in both directions. It would also be better to make the maximum steering angle the same for both directions. This would make the go-kart be able to follow the same path for both left and right turns. It would also be better to further investigate if the steering angles are the same for the continuous case between the maximum values. This would improve SLAM's ability to achieve an accurate map of the environment, which the path planner can work with.

Performance of IMU

To achieve a better localization estimate of the go-kart, the IMU would need to be better tuned for the AP4 in order to work effectively together with the odometry data from the wheel encoders and the steering angle. Several things can be done to achieve a better performance from the IMU-Odometry sensor fusion. The first thing to do is to verify that all of the IMU data is reasonable. To limit unwanted behavior, it is possible to limit which state variables to use in the `robot localization` EKF configuration. Since the rotations in the roll and pitch directions are not relevant information, these could be disregarded and set to False in the `robot localization` EKF configuration. This would disregard all noise in those directions and possibly improve the localization.

Another thing to look at regarding the IMU is the covariance matrix for the process noise. For future development and improvement, this matrix would probably need to be tuned for this specific project. A well-tuned matrix will probably increase the performance of the IMU in the `robot localization` node.

LiDAR improvements

Another way to improve the localization is to match more LiDAR data. As of now, the LiDAR is only facing forward and the localization is therefore not updating itself against the previously gathered data behind the go-kart. If more LiDARs are implemented, the scans would be able see 360° around the go-kart, which would most likely improve the localization.

6.4.2 LiDAR-Camera sensor fusion

During this thesis, the main focus has been to implement a LiDAR and use its sensor data to autonomously navigate around the go-kart track. One interesting possibility for future development is to investigate whether sensor fusion between the LiDAR and the already existing camera can improve the AD. By combining different types of sensor data for the system, it could give a better understanding of the go-kart's environment. As of now, the SLAM algorithms have some trouble identifying distinctive landmarks on the go-kart track. With edge- and ORB detection, which was investigated in last year's thesis, it might be possible to make it easier for the SLAM algorithm to identify landmarks. The fusion of these sensors might give a good complement to each other, where the camera offers high-resolution color and texture information and the LiDAR provides precise depth and distance[6].

Bibliography

[1] Manzoor Khan Henry Alexander Ignatious Hesham-El-Sayed. *An overview of sensors in Autonomous Vehicles*. Accessed: 2025-05-27. 2021. URL: https://pdf.sciencedirectassets.com/280203/1-s2.0-S1877050921X0021X/1-s2.0-S1877050921025540/main.pdf.

[2] Encyclopedia Britannica. *Autonomous vehicle | Definition, History, & Facts - Britannica*. Accessed: 19-Mar-2025. 2025. URL: https://www.britannica.com/technology/autonomous-vehicle.

[3] J. Wellander & A. Petersén. *Autonomous Driving via Imitation Learning in a Small-Scale Automotive Platform*. URL: https://github.com/infotiv-research/autonomous_platform/blob/main/Master_Thesis2024.pdf.

[4] E. Magnusson & F. Juthe. *Design of a modular centralized E/E and software architecture for a small-scale automotive platform*. URL: https://github.com/infotiv-research/autonomous_platform/blob/main/Master_Thesis2023.pdf.

[5] Segway. *Ninebot Go-Cart PRO*. Accessed: 2025-05-27. 2025. URL: https://se-en.segway.com/products/ninebot-gokart-pro.

[6] OMID KARAMI. *3D Object Detection via LiDARCamera Fusion in Autonomous Driving*. URL: https://his.diva-portal.org/smash/get/diva2:1900805/FULLTEXT01.pdf.

[7] Hongyi Li Jun Zhu and Tao Zhang. *Camera, LiDAR, and IMU Based Multi-Sensor Fusion SLAM: A Survey*. URL: https://www.sciopen.com/article/pdf/10.26599/TST.2023.9010010.pdf?ifPreview=0.

[8] Oscar Sanner. *Autonomous Navigation: LIDAR-based SLAM, Terrain of Technology Explored*. URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9134347&fileOId=9134362.

[9] Oscar Sanner. *Autonomous Navigation: LIDAR-based SLAM, Terrain of Technology Explored*. Accessed: 2025-05-19. 2023. URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9134347&fileOId=9134362.

[10] *Gokartcentralen Kungälv*. URL: https://gokartcentralen.se/info/banan/.

[11] Synopsys. *What is Lidar?* Accessed: 2025-04-04. 2025. URL: https://www.synopsys.com/glossary/what-is-lidar.html#1.

[12] Elecfans. *Vad är LiDAR? Arbetsprinciper och lösningar för LiDAR-systemet*. Accessed: 2025-06-05. 2023. URL: https://www.elecfans.com/d/2346006.html.

[13] Dubai Sensor. *Revving Up Precision: Exploring the World of Wheel Encoders*. Accessed: 2025-05-27. 2023. URL: https://www.dubai-sensor.com/blog/revving-up-precision-exploring-the-world-of-wheel-encoders/.

[14] Amazon. *LM393*. Accessed: 2025-06-05. 2025. URL: https://www.amazon.se/-/en/Youmile-Measurement-Optocoupler-Arduino-Encoder/dp/B0817H9436.

[15] Jouav Unmanned Aircraft Systems. *A Complete Guide to Inertial Measurement Unit*. Accessed: 2025-05-27. 2025. URL: https://www.jouav.com/blog/inertial-measurement-unit.html.

[16] Gordon Wetzsteins. *3-DOF Orientation Tracking with IMUs*. Accessed: 2025-05-27. URL: https://stanford.edu/class/ee267/notes/ee267_notes_imu.pdf.

[17] Daniel Dominguez. *Ackermann Kinematic Modeling for AWS DeepRacer*. Accessed: 2025-05-27. 2020. URL: https://dominguezdaniel.medium.com/aws-deepracer-ackermann-kinematic-model-3e0b32e79923.

[18] Eugen Kaltenegger. *Physical and Graphical Simulation of an Ackermann Steered Vehicle*. Accessed: 2025-05-27. 2016. URL: https://www.auto.tuwien.ac.at/bib/pdf_TR/TR0183.pdf.

[19] Shaun Turney. *Central Limit Theorem | Formula, Definition Examples*. Accessed: 2025-05-27. 2022. URL: https://www.scribbr.com/statistics/central-limit-theorem/.

[20] Janvi Kumari. *What are Mean and Variance of the Normal Distribution?* Accessed: 2025-05-27. 2024. URL: https://www.analyticsvidhya.com/blog/2024/11/mean-and-variance-of-the-normal-distribution/?utm_source=chatgpt.com.

[21] Jeffrey K. Uhlmann Simon J. Julier. *A New Extension of the Kalman Filter to Nonlinear Systems*. Accessed: 2025-05-27. URL: https://www.cs.unc.edu/~welch/kalman/media/pdf/Julier1997_SPIE_KF.pdf.

[22] Robot Operating System 2. *ROS 2 Documentation*. Accessed: 2025-05-27. URL: https://docs.ros.org/en/humble/index.html.

[23] Steve Macenski, Tully Foote, Brian Gerkey, Chris Lalancette and William Woodall. *Robot Operating System 2: Design Architecture, and Uses In The Wild*. 2022. URL: https://arxiv.org/abs/2211.07752.

[24] Robot Operating System 2. *RViz*. Accessed: 2025-05-27. URL: https://docs.ros.org/en/humble/Tutorials/Intermediate/RViz/RViz-Main.html.

[25] Navigation2 Project Contributors. *Navigation2 (NAV2) Documentation*. Accessed: 2025-04-04. 2024. URL: https://docs.nav2.org/index.html.

[26] Robotics Knowledgebase. *Adaptive Monte Carlo Localization*. Accessed: 2025-05-27. 2020. URL: https://roboticsknowledgebase.com/wiki/state-estimation/adaptive-monte-carlo-localization/.

[27] Chao Liang. *Application of A* algorithm in intelligent vehicle path planning*. Accessed: 2025-04-25. 2022. URL: https://www.extrica.com/article/22828.

[28] NILS J. NILSSON PETER E. HART and BERTRAM RAPHAEL. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. Accessed: 2025-04-25. 2022. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4082128.

[29] Dmitri Dolgov et al. "Practical Search Techniques in Path Planning for Autonomous Driving". In: (2008). Used by the Stanford Racing Team's autonomous vehicle, Junior, in the DARPA Urban Challenge. URL: https://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf.

[30] Vaidehi Joshi. *Finding The Shortest Path, With A Little Help From Dijkstra*. Accessed: 2025-05-27. 2017. URL: https://medium.com/basecs/finding-the-shortest-path-with-a-little-help-from-dijkstra-613149fbdc8e.

[31] Adrián Riesco Enrique Martin Manuel Montenegro. *Verification of the ROS NavFn planner using executable specification languages*. Accessed: 2025-05-27. 2023. URL: https://www.sciencedirect.com/science/article/pii/S2352220823000147.

[32] Sebastian Thrunyz Dieter Foxy Wolfram Burgardy. *The Dynamic Window Approach to Collision Avoidance*. Accessed: 2025-05-27. URL: https://www.ri.cmu.edu/pub_files/pub1/fox_dieter_1997_1/fox_dieter_1997_1.pdf.

[33] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. URL: https://books.google.se/books?id=wpYpCwAAQBAJ.

[34] Shivaraj Kengondc Mohammed Mullad Amit Potdara Narayan. *Performance Evaluation of Docker Container and Virtual Machine*. URL: https://www.sciencedirect.com/science/article/pii/S1877050920311315.

[35] Kinza Yasar. *Docker image*. URL: https://www.techtarget.com/searchitoperations/definition/Docker-image.

[36] Quantstart. *Bayesian Statistics: A Beginner's Guide*. Accessed: 2025-05-27. 2022. URL: https://www.quantstart.com/articles/Bayesian-Statistics-A-Beginners-Guide/.

[37] Joan Sol'a. *Simulataneous localization and mapping with the extended Kalman filter d*. URL: https://www.iri.upc.edu/people/jsola/JoanSola/objectes/curs_SLAM/SLAM2D/SLAM%20course.pdf.

[38] STM-32 base. *Blue Pill STM32F103C8T6*. Accessed: 2025-05-27. 2025. URL: https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html.

[39] PlatformIO. *PlatformIO*. Accessed: 2025-05-27. 2025. URL: https://platformio.org/.

[40] PCHButik.se. *LM2596 DC Converter*. Accessed: 2025-05-27. 2025. URL: https://shop.pchbutik.se/sv/volt-dc-dc/226-justerbar-dc-dc-step-down-spannings-omvandlare-lm2596.html.

[41] electro:kit. *CAN-bus modul MCP2515 / TJA1050 SPI*. Accessed: 2025-05-27. 2025. URL: https://www.electrokit.com/can-bus-modul-mcp2515-/-tja1050-spi.

[42] Dimension Engineering. *Kangaroo x2 motion controller*. Accessed: 2025-05-27. 2025. URL: https://www.dimensionengineering.com/products/kangaroo.

[43] Dimension Engineering. *Sabertooth dual 50A motor driver*. Accessed: 2025-05-27. 2025. URL: https://www.dimensionengineering.com/products/sabertooth2x50hv.

[44] Conrad. *LM393 speed sensor*. Accessed: 2025-05-27. 2025. URL: https://www.conrad.se/sv/p/joy-it-sen-speed-utbyggnadsmodul-lamplig-for-enkorstdator-arduino-banana-pi-cubieboard-raspberry-pi-pcduino-1-st-1646891.html.

[45] Raspberry Pi. *Raspberry Pi 4*. Accessed: 2025-05-27. 2025. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[46] Infotiv. *Autonomous Platform*. Accessed: 2025-06-05. 2025. URL: https://github.com/infotiv-research/autonomous_platform.

[47] Gokartcentralen. *Banor*. Accessed: 2025-06-05. 2025. URL: https://gokartcentralen.se/info/banan/.

[48] Blender.org. *Blender*. Accessed: 2025-05-27. 2025. URL: https://www.blender.org/.

[49] Thor I.Fossen Anastasios M.Lekkas. *Minimization of Cross-track and Along-track Errors for Path Tracking of Marine Underactuated Vehicles*. Accessed: 2025-05-27. 2014. URL: https://www.fossen.biz/publications/2014%20Lekkas%20and%20Fossen%20ECC.pdf.

[50] Slamtec. *RPLiDAR A1*. Accessed: 2025-05-27. 2025. URL: https://www.slamtec.com/en/lidar/a1.

[51] Luxonis. *OAK-D*. Accessed: 2025-05-27. 2025. URL: http://shop.luxonis.com/products/oak-d.

[52] ROS 2. *robot localization wiki*. Accessed: 2025-05-27. 2025. URL: https://docs.ros.org/en/melodic/api/robot_localization/html/index.html.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY