Uppsala University

Project report

Projekt i inbyggda system — 1TE721

Indoor navigation of an autonomous car based on LIDAR SLAM positioning and ROS

George Alhoush[1] Nils Axelsson[2]

October-December, 2023

UPPSALA
UNIVERSITET

[1]E–mail: George.Alhoush.3713@student.uu.se
[2]E–mail: Nils.Axelsson.9385@student.uu.se

Abstract

This project presents the final phase of the development of an autonomous Remote-Controlled (RC) car. The primary objective of this project is to transform a commercial RC car chassis into a fully autonomous car which can avoid obstacles and navigate to a predetermined position.

To achieve this goal, a Raspberry Pi and three different sensor types are used. A Light Detection and Ranging (LIDAR) sensor is used for mapping and positioning, an Inertial Measurement Unit (IMU) is used for acceleration measurements and ultrasonic sensors are used to detect obstacles not present in the map.

The main software framework used is the Robot Operating System (ROS), which is leveraged to enable sensor-to-actuation functionalities and distributed computing. Matlab is also used for sensor fusion using a Kalman filter and to implement the autonomous navigation algorithm using tools from its navigation toolbox.

After overcoming initial challenges, both hardware and software related, we were ultimately successful in creating a car which can navigate fully autonomously to a goal point and back, assuming that it can turn freely. It is also able to detect and navigate around unexpected obstacles in its path.

This was achieved by first creating an occupancy grid map of the working area using a Simultaneous Localisation and Mapping (SLAM) algorithm on a collected series of LIDAR scans. A path from the starting point to the target point is then found in this grid using a Probabilistic Road Map (PRM) algorithm. The path is then followed using a Pure Pursuit controller which is fed current position and heading estimates, creating a closed loop system. A set of three ultrasonic sensors in the front of the car allow it to detect obstacles and to re-plan the path in order to navigate around them.

For future work, it would be of interest to see whether a similar but visual SLAM-based system can be implemented. A system based on a camera input might have the advantage of being able to incorporate both obstacle detection and recognition, as well as live 3D mapping of the environment using only one sensor input, as opposed to the current 2D system which relies on LIDAR and ultrasonic sensors for positioning and obstacle detection.

Contents

1 Introduction **4**
- 1.1 Background . . . 4
- 1.2 Purpose and objective of the project . . . 5
- 1.3 Project specifications . . . 5
- 1.4 Work distribution and planning . . . 5
- 1.5 Preliminary time schedule and grading criteria . . . 6

2 Theory **8**
- 2.1 ROS . . . 8
  - 2.1.1 ROS nodes . . . 8
- 2.2 LIDAR sensors . . . 9
- 2.3 Ultrasound sensors . . . 10
- 2.4 Simultaneous Localisation and Mapping (SLAM) . . . 11
- 2.5 Localisation with IMU sensors and LIDAR-based SLAM . . . 13
  - 2.5.1 State space model for position, velocity and acceleration estimation with a Kalman filter . . . 13
  - 2.5.2 Kalman equations . . . 15
- 2.6 Probability RoadMaps . . . 16
- 2.7 Pure pursuit controller . . . 17
- 2.8 Servo and stepper motors . . . 20
  - 2.8.1 Servo motor control with PWM signals . . . 20
  - 2.8.2 Stepper motor control . . . 22
- 2.9 The obstacle avoidance system . . . 22

3 Implementation **25**
- 3.1 Overview of the system . . . 25
- 3.2 Hardware and components . . . 27
  - 3.2.1 Computing resources . . . 27
  - 3.2.2 The RC car . . . 28
  - 3.2.3 Using the LIDAR sensor . . . 28
  - 3.2.4 Adafruit 9DOF IMU . . . 30
- 3.3 Software and development tools . . . 30
  - 3.3.1 The ROS system . . . 31
  - 3.3.2 Interfacing the LIDAR sensor with ROS and Matlab . . . 32
- 3.4 Hardware implementation . . . 32
- 3.5 Matlab implementation . . . 33

4 Results and discussion **34**
4.1 Overview of results 34
4.1.1 SLAM results and occupancy map 35
4.1.2 Kalman filter results 36
4.1.3 Results of path planning using PRM 37
4.1.4 Pure Pursuit results 39
4.1.5 Obstacle avoidance results 41

5 Discussion **42**
5.1 Challenges and solutions 42
5.2 Conclusions and further work 44

6 References **45**

1 Introduction

This report details the creation of an autonomous remote controlled (RC) car using three different sensor types as well as the Robot Operating System (ROS). The report is organised as follows. Section 1 gives an introduction to the project and contains a list of the hardware used. The relevant aspects of ROS and the tools it provides are presented in section 2, along with the working principles and theory behind the sensors and actuators. The Kalman filter, the path finding algorithm and the path-following controller is also introduced in this section. Section 3 gives an overview of the implementation and contains flowcharts describing the whole system. The results are then presented in section 4, with a following discussion about the results presented in section 5.

1.1 Background

Autonomous navigation systems have many applications, and some functioning systems can be seen in everyday life in the form of self-driving cars and vacuum cleaning robots. Many such applications rely on pre-made maps to generate a path to a specified location, and the navigation to that goal is done by a controller aided by real time sensor data to facilitate obstacle avoidance along the path.

To create a small autonomous system for indoor navigation, one can use a commercial RC car chassis and a single-board computer such as a Raspberry Pi. Pairing the Raspberry Pi with a LIDAR sensor for mapping of the indoor environment, an Inertial Motion Unit (IMU) sensor for motion monitoring and ultrasonic sensors for obstacle detection, it can serve as a brain for the car. Communication between the Raspberry Pi and the other system components can be done via ROS, the Robot Operating System. As such, the single-board computing platform can orchestrate the RC car's autonomous behavior. By leveraging the ROS communication system, additional computing power can easily be added to the system by sharing live sensor data between computers over a WiFi-network.

The intent of this is to implement a self-driving car which can navigate autonomously to a specified location indoors in an indoor environment.

1.2 Purpose and objective of the project

The aim of this 15 credit project is to implement a system which can make the car autonomous. The car will be paired with LIDAR, IMU and ultrasonic sensors, which will also be integrated in a ROS environment. The first part of this project involves hardware setup and actuator control, developing software for sensor data acquisition as well as creating a ROS network for sensor-to-actuator communication. The second part is concerned with path planning, sensor fusion and path following controllers.

The purpose of this project is to get the chance to solve a complex task related to embedded systems engineering.

1.3 Project specifications

The hardware available for the project is as follows:

- A commercial RC car chassis
- Raspberry Pi 4
- LIDAR sensor (RPLIDAR/A1)
- IMU (Adafruit Industries/9-DOF IMU Breakout Board)
- Motor power driver boards (L298N)
- Stepper motor driver (ARCELI A4988)
- Ultrasonic sensors (Adafruit Industries/LLC3942)
- Breadboards, resistors and connectors

1.4 Work distribution and planning

This project can be divided into four main parts:

1. Connecting the sensors and motors to ROS and creating publisher-subscriber communication networks for live sensor data sharing over a WiFi-network.
2. Sensor fusion using a Kalman filter for estimating the current position and acceleration of the car from noisy sensor measurements.

3. Creating a SLAM map of the environment and integrating path planning algorithms as well and path-following controllers into one system.

4. Implementing a live obstacle avoidance system which can automatically re-plan the path to the goal if sudden obstacles appear.

George has been mainly responsible for parts one and four, while Nils has been working on the second and third parts of the project.

1.5 Preliminary time schedule and grading criteria

Week	Work Contents
1 – 3	Create a preliminary plan and research the different approaches to navigation we have identified.
4 – 5	Decide on a navigation approach based on previous research and start implementing it. Look into how the LIDAR measurements can be used to estimate the current position.
5 – 7	Continue working on the navigation system and start implementing the Kalman filter. Try to plot the estimated position in real time on an occupancy grid map in Matlab.
7 – 8	Focus on writing the report and present the whole project including the project progresses in the final phase.

The grading criteria

Grade 3: {tasks finished on a basic level}

1. Studying the theory behind the project, and writing a comprehensive report.

2. Successfully implementing a simple self driving algorithm which allows the car to navigate in a straight line to a specified position and then back to its original position.

Grade 4: {tasks finished at an advanced level} Successfully implementing a more advanced navigation system which allows the car to navigate to a goal in an environment with known obstacles. For example, the navigation system could be either

- (1) A path generating algorithm paired with an advanced path following controller from Matlab.

- (2) The *teb_local_planner* algorithm from ROS, which is capable of advanced navigation and live obstacle avoidance for car-like robots.

- Successfully implementing a Kalman filter for estimating the live position and acceleration of the car while navigating using different sensor measurements. Since the LIDAR measurements can be used to estimate the position of the car in real time, the Kalman filter should use either such measurements or UWB sensor data.

Grade 5: {tasks finished at a challenging level}

1. Successfully implementing a more advanced navigation system where the car can detect and navigate around obstacles not present in the occupancy grid. **Alternatively**

   Use a machine learning algorithm to detect images with instructions such as 'left', 'right' or 'stop', and modify the path accordingly.

2 Theory

This section presents the working principles of the Robot Operating System (ROS), LIDAR sensors, Inertial Measurment Units (IMUs) and ultrasonic sensors. Theory regarding LIDAR SLAM algorithms is also presented, along with the working principles behind servo and stepper motors. A state space model for estimating the position, velocity and acceleration using a Kalman filter with position and acceleration measurements as input is also presented. The basic idea behind the Probability RoadMap (PRM) path generating algorithm, the Pure Pursuit path following controller and a simple algorithm for navigating around obstacles is also presented.

2.1 ROS

The Robot Operating System is a middleware framework designed for robotics applications. It serves as a layer connecting the different components in a typical robotic system, namely sensors, actuators and intelligent control systems [1]. The communication infrastructure ROS provides can be used to share data wirelessly between different computers and robots, allowing for a distributed computing and parallel instruction execution.

Since ROS is open source, code reusability is encouraged and many libraries and tools for common sensors and actuators can be found.

There are two main versions of ROS, the original version called ROS1 and the newer version ROS2. ROS1 is mainly suited for the Linux operating system, while ROS2 is platform independent and can be used with macOS and Windows as well.

2.1.1 ROS nodes

A process running ROS code is called a *node*. Nodes often perform specific tasks, and can share information with other nodes through the ROS messaging infrastructure. Several nodes are often packaged together with specific sensor or actuator libraries and configuration files to create a ROS package.

The ROS code itself is often written in Python or C++ using the `rospy` or `roscpp` libraries, [1]. Using the ROS toolbox, Matlab code can also be used [2]. For simple processes, ROS specific commands written directly to the terminal can also be used.

One of the core features of ROS is the Publish-Subscribe communication system, which allows for distributed computing, messaging and visualization [3]. A ROS node can publish data to information channels called *topics*. Other nodes can also publish to the same topic, or subscribe to it in order to receive the published data. The publishers do not need to know which nodes, if any, who are subscribed to the topic.

Publisher and subscriber nodes can run on different machines, and the wireless data transfer will be handled automatically if the machines are connected to the same WiFI-network. This communication mechanism allows for distributed parallel computing, visualisation and actuation using live sensor data.

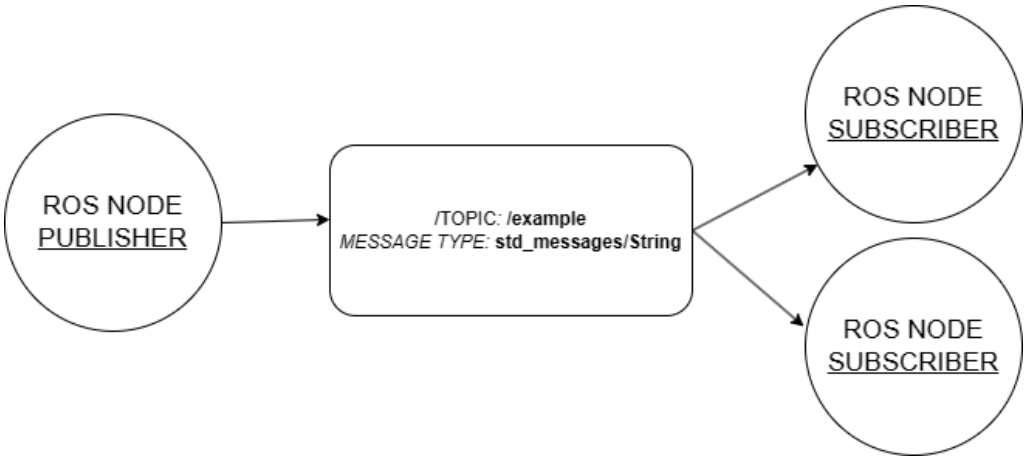


Figure 1: An example of a publisher-subscriber network [2]. A publisher node broadcasts messages of the type **std_messages/String** on the **/example** topic. Two subscriber nodes receive the messages.

2.2 LIDAR sensors

LIDAR, which stands for Light Detection and Ranging, is a sensor type that measures the distance to its surroundings. It works by sending out a targeted laser beam towards a surface and then measuring the time it takes that beam to scatter back to its receiver. Figure 2 shows this working principle. The distance to a surrounding object is calculated using

$$d = \frac{c \cdot t}{2},$$

where d is the distance to the object, c is the speed of light and t is the time it took the light to reach its destination and return to the receiver.

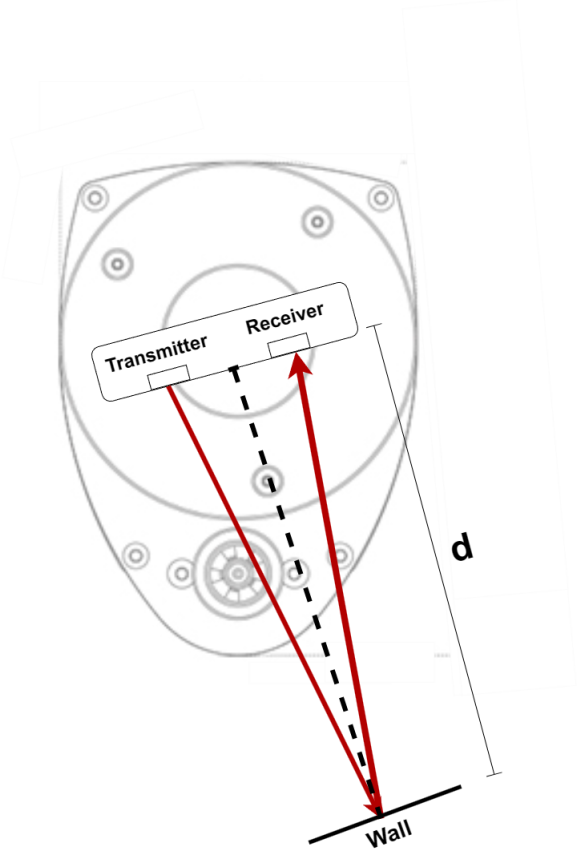


Figure 2: Working principle of the RPLIDAR A1 from Slamtec [4].

2.3 Ultrasound sensors

Ultrasound sensors work in the same way as LIDAR sensors with one difference between them, LIDAR uses light while ultrasound sensor uses sound to measure the distance to an object. It works by sending out a targeted sound wave towards a surface and then measuring the time it takes that wave to reflect back to its receiver. The distance to that object can be calculated using:

$$d = \frac{v_{sound} \cdot t}{2}$$

Similarly to LIDAR, d is the distance to the object, v_{sound} is the speed of sound, and t is the time it takes the sound wave to travel to the object and reflect back.

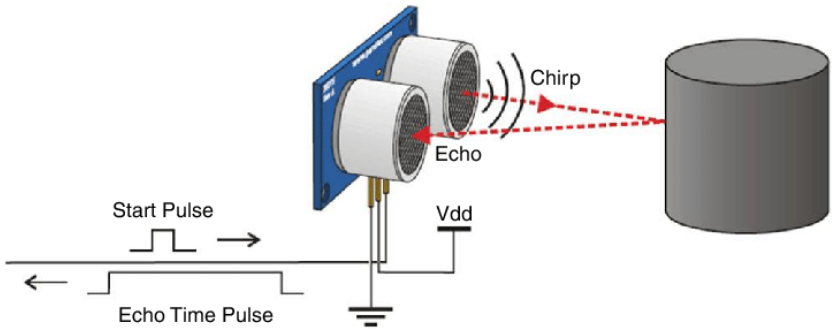


Figure 3: An illustration of how the ultrasonic sensor works [5]

2.4 Simultaneous Localisation and Mapping (SLAM)

One of the key applications of LIDAR sensors other than just live distance measurements is mapping. A common algorithm type for this is called Simultaneous Localisation and Mapping (SLAM). SLAM maps can later be converted into binary occupancy grids which can be used with path finding algorithms.

In essence, SLAM is a probabilistic problem that needs to be solved for each time step k. Assume we have an autonomous car navigating through a workspace equipped with a LIDAR sensor. The sensor takes relative observations of a number of unknown landmarks (obstacles). Lets define the following quantities:

- x_k: The state vector which includes the vehicle's odometry-data such as position and orientation at time k
- u_k: The control signal applied on the vehicle's motor at time $(k-1)$ to drive the vehicle to the position x_k at time k
- m_i: The vector which contains the location data of the i^{th} obstacle that this algorithm tracks.
- z_{ik}: The observation taken by the vehicle's LIDAR sensor of the location of the i^{th} obstacle at time step k

SLAM also requires the definition of the following data sets:

- $X_{0:k} = \{x_0, x_1, x_2, ..., x_k\}$: The history vector of the vehicle location up to and including time step k.
- $U_{0:k} = \{u_1, u_2, ..., u_k\}$: The history vector of the vehicle input control signals up to and including time step k.
- $m = \{m_1, m_2, ..., m_i\}$: The location of all the landmarks in the workspace.
- $Z_{0:k} = \{z_1, z_2, ..., z_k\}$: The history vector of all the observation taken by the vehicle LIDAR sensor of the location of the i^{th} landmarks up to and including time step k.

The main problem to solve in SLAM is to calculate

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) \tag{1}$$

for each time step k. This expression describes the collective probability density of all the landmark locations and vehicle states (at time k), given the recorded observations and control inputs up to and including time k together with the initial state of the vehicle. It is preferable to solve the probability density in equation (1) recursively, because an autonomous car needs to know the location of itself and the location of all the landmarks, in order to avoid any collisions with the them as it drives itself through the workspace to its destination.

The recursive calculation of the joint posterior probability density of all the landmark location and vehicle state $P(x_k, m|Z_{0:k}, U_{0:k}, x_0)$ can be done by using Bayes theorem, but it requires the acquisition of prior information, namely the **state transition model** and an **observation model**.

The observation model describes the probability of making an observation of the landmarks location, given the vehicle location, and the general landmarks location. This can be described by the following probability density:

$$P(z_k|x_k, m) \tag{2}$$

The state transition model, on the other hand, describe how the vehicle state vector x_k will change, given the old state vector x_{k-1} and the input control signal applied in the vehicle's motor at time $(k-1)$. it can generally be described the the following probability density:

$$P(x_k|x_{k-1}, u_k) \tag{3}$$

The state transition model can be considered Markovian, which means x_k will only depend on x_{k-1} and u_k.

After acquiring the result for both Equation (2) and (3), the recursive calculation of Eq. (1) can be completed in two steps, prediction (time update) and correction (measurement update).

Prediction:

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) = \int P(x_k|x_{k-1}, u_k) \times P(x_{k-1}, m|Z_{0:k-1}, U_{0:k-1}, x_0) dx_{k-1} \tag{4}$$

Correction:

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k|x_k, m)P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0)}{P(z_k|Z_{0:k-1}, U_{0:k})} \tag{5}$$

This problem can be also expressed as $P(m|X_{0:k}, Z_{0:k}, U_{0:k})$, but this assumes that the car location is known and deterministic for all time steps k [6].

2.5 Localisation with IMU sensors and LIDAR-based SLAM

Inertial Measurement Units (IMUs) are sensor devices that combine accelerometers, gyroscopes, and magnetometers to measure an object's orientation, angular velocity and acceleration [7]. The SLAM algorithm in Matlab continously outputs the current position and heading of the LIDAR which provides its input. Together with the linear acceleration from the accelerometer on the IMU, their output can be combined in a state space model, which enables position tracking of a robot using a Kalman filter.

2.5.1 State space model for position, velocity and acceleration estimation with a Kalman filter

A state space model for estimation of the position, velocity and acceleration from IMU and LIDAR SLAM measurements is given in [8], where $\boldsymbol{x}$ is the state vector

$$\boldsymbol{x} = \begin{bmatrix} p_x & p_y & v_x & v_y & a_x & a_y \end{bmatrix}^{\mathrm{T}}$$

which after the prediction becomes

$$\boldsymbol{x'} = \begin{bmatrix} p'_x & p'_y & v'_x & v'_y & a'_x & a'_y \end{bmatrix}^{\mathrm{T}}$$

with the motion model

$$\begin{cases} p'_x = & p_x + v_x\Delta t + \frac{1}{2}a_x\Delta t^2 + \frac{1}{6}j_x\Delta t^3 \\ p'_y = & p_y + v_y\Delta t + \frac{1}{2}a_y\Delta t^2 + \frac{1}{6}j_y\Delta t^3 \\ v'_x = & v_x + a_x\Delta t + \frac{1}{2}j_x\Delta t^2 \\ v'_y = & v_y + a_y\Delta t + \frac{1}{2}j_y\Delta t^2 \\ a'_x = & a_x + j_x\Delta t \\ a'_y = & a_y + j_y\Delta t. \end{cases}$$

where Δt is the time between two samples. The state space model is then

$$\begin{cases} \boldsymbol{x'} = \boldsymbol{F}\boldsymbol{x} + \boldsymbol{v} \\ \boldsymbol{z} = \boldsymbol{H}\boldsymbol{x'} + \omega \end{cases} \tag{6}$$

where

$$\boldsymbol{F} = \begin{bmatrix} 1 & 0 & \Delta t & 0 & \frac{\Delta t^2}{2} & 0 \\ 0 & 1 & 0 & \Delta t & 0 & \frac{\Delta t^2}{2} \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

and where $\boldsymbol{v}$ is the jerk (rate of change of acceleration with respect to time) vector modelled as random noise

$$\boldsymbol{v} = \begin{bmatrix} \frac{1}{6}j_x\Delta t^3 \\ \frac{1}{6}j_y\Delta t^3 \\ \frac{1}{2}j_x\Delta t^2 \\ \frac{1}{2}j_y\Delta t^2 \\ j_x\Delta t \\ j_y\Delta t \end{bmatrix} = \begin{bmatrix} \frac{1}{6}\Delta t^3 & 0 \\ 0 & \frac{1}{6}\Delta t^3 \\ \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} j_x \\ j_y \end{bmatrix} = \boldsymbol{G}j.$$

The jerk vector $\boldsymbol{v}$ is seen as zero mean Gaussian noise $\boldsymbol{v} \sim \mathcal{N}(\mu, \boldsymbol{Q}) = \mathcal{N}(0, \boldsymbol{Q})$, where $\boldsymbol{Q}$ is the covariance matrix

$$\boldsymbol{Q} = \mathbb{E}[\boldsymbol{v}\boldsymbol{v}^T] = E[\boldsymbol{G}jj^T\boldsymbol{G}^T] = \boldsymbol{G}\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}\boldsymbol{G}^T. \tag{7}$$

The measurement vector $\boldsymbol{z}$ and the projection matrix $\boldsymbol{H}$ from the current state vector to the measurement space are defined as

$$\boldsymbol{z} = \begin{bmatrix} p_x \\ p_y \\ a_x \\ a_y \end{bmatrix}, \quad \boldsymbol{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The ω therm in the state space model is the measurement uncertainty, with a measurement noise covariance matrix $\boldsymbol{R}$

$$\boldsymbol{R} = \begin{bmatrix} \sigma_{px}^2 & 0 & 0 & 0 \\ 0 & \sigma_{py}^2 & 0 & 0 \\ 0 & 0 & \sigma_{ax}^2 & 0 \\ 0 & 0 & 0 & \sigma_{ay}^2 \end{bmatrix}, \tag{8}$$

This covariance matrix can be adjusted based on the confidence in the measurements. Manufacturers of sensors often include the measurement uncertainty in the data sheet [8].

2.5.2 Kalman equations

The algorithm structure of the Kalman filter can be divided into two parts, “Predictions" and “Updates". The state vector $\boldsymbol{x}$ and covariance matrix $\boldsymbol{P}$ are initialised beforehand.

Updates

$\boldsymbol{x'} =$	$\boldsymbol{Fx}$	(LMMSE prediction of $\boldsymbol{x}$)
$\boldsymbol{P'} =$	$\boldsymbol{FPF}^T + \boldsymbol{Q}$	(Error covariance associated with the LMMSE prediction)

Predictions

$\boldsymbol{y} =$	$\boldsymbol{z} - \boldsymbol{Hx}$	(Innovation - Difference between model & observation)
$\boldsymbol{S} =$	$\boldsymbol{HP'H}^T + \boldsymbol{R}$	(Covariance of innovation)
$\boldsymbol{K} =$	$\boldsymbol{P'H}^T\boldsymbol{S}^{-1}$	(Optimal Kalman gain)
$\boldsymbol{x} =$	$\boldsymbol{x'} + \boldsymbol{Ky}$	(LMMSE filter estimate of $\boldsymbol{x}$)
$\boldsymbol{P} =$	$(\boldsymbol{I} - \boldsymbol{KH})\boldsymbol{P'}$	(Error covariance associated with the LMMSE filter estimate)

2.6 Probability RoadMaps

A probabilistic roadmap (PRM) is a network graph of possible paths in a given map based on free and occupied spaces. The PRM algorithm uses the network of connected nodes to find an obstacle-free path from a start to an end location. In Matlab, the PRM algorithm makes use of two parameters

1. The number of nodes, `NumNodes`.

2. The maximum connection distance between nodes, `ConnectionDistance`

When using the PRM algorithm in Matlab, a `mobileRobotPRM` object randomly generates nodes and creates connections between these nodes. All nodes are first connected, but connections which intersect with obstacles or are too long are subsequently deleted. The shortest path from start to goal is then found along the connected nodes.

The number of nodes and the connection distance can be customized to fit the complexity of the map and the desire to find a more efficient path.

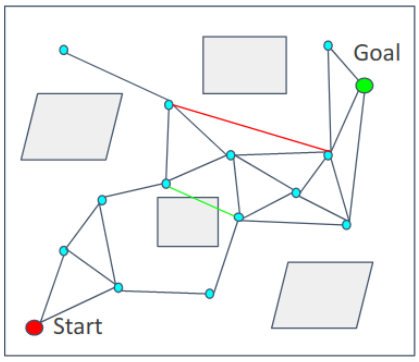


Figure 4: An example of connecting nodes in the creation of a PRM. The red connection is removed because of the long distance, while the green connection is removed because of the blocking obstacle.

By increasing the number of nodes generated, and specifying a shorter maximum distance between them, a more optimal and smoother path can be found. When creating or updating the `mobileRobotPRM` object, the node locations are randomly generated, which can lead to unique paths generated each time even if the start and end locations are the same.

2.7 Pure pursuit controller

Navigating to a predetermined coordinates requires:

1. A path which the car should follow from its starting point (SP) to the target point (TP).

2. A controller which keep the car on the path, and makes adjustments to the motors signals in case the car deviates from its path.

The controller in this case utilises the position data from a Kalman filter and the heading angle of a LIDAR sensor to create a closed loop system where the input to the controller is the measured error between the reference position (on the path) and the measured position. The output from the controller is a steering angle value, which is then converted to the corresponding signal for the servo motor, facilitating movement along the path according to diagram 5. The speed of the car, controlled by the stepper motor, is set to a constant value unless an obstacle is detected.

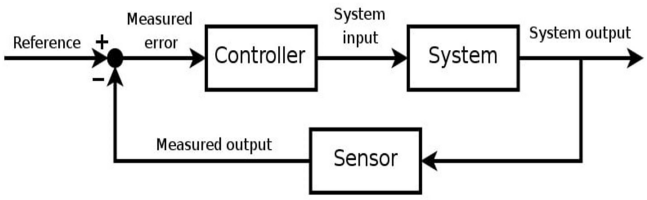


Figure 5: The role of a controller in a feedback controlled system.

There are many controllers for path following, with PID controllers being the most commonly used one due to their simplicity and good performance. Another simple and commonly used controller for path following is the Pure Pursuit controller, which is used in this case.

To understand the theory behind this controller, it is important to note that we choose to model the car as a bicycle for simplicity. In the bicycle kinematic model, there are four main parameters that should be taken into consideration, namely:

- Wheel base length l
- Velocity v

- Steering angle ψ
- Turning radius R

The steering angle for the car can be derived from that model using the following equation [9]:

$$\psi = \arctan\left(\frac{l}{R}\right) \tag{9}$$

On the other hand, one important parameter for the controller is the look ahead distance l_d, which is analogous to the distance to a spot in front of a car that a human driver might look toward to track the roadway.

The controller discretises the given path to several target points (TP) where each TP is one look ahead distance from the car. The controller calculates the steering angle ψ which allows the car to reach the target point. To establish the steering angle, trigonometry and drawing the isosceles triangle in Figure 6 are required. One of the characteristics of isosceles triangle is:

$$\gamma_2 = \gamma_3 \tag{10}$$

But we can as well see that

$$\gamma_3 + \alpha = 90° \quad \Longrightarrow \quad \gamma_2 = \gamma_3 = 90° - \alpha \tag{11}$$

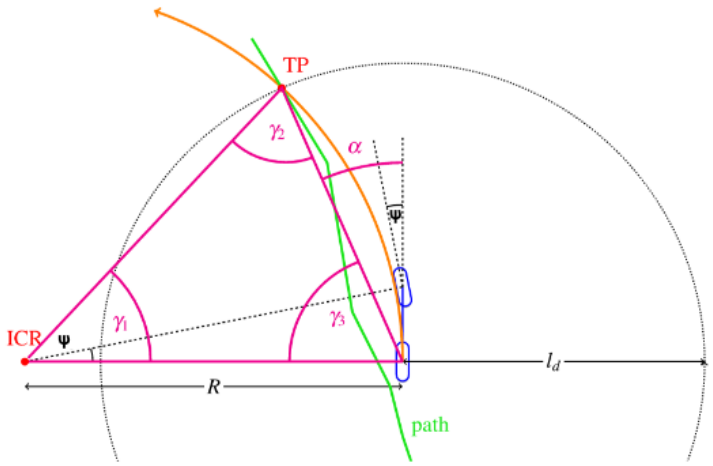


Figure 6: An illustration of a bicycle model with pure pursuit controller [10]

Since the sum of all angles in a triangle is 180°

$$180° = \gamma_1 + \gamma_2 + \gamma_3 = \gamma_1 + (90° - \alpha) + (90° - \alpha) \quad \Longrightarrow \quad \gamma_1 = 2\alpha \tag{12}$$

According to law of sines:

$$\frac{l_d}{\sin \gamma_1} = \frac{R}{\sin \gamma_2} \tag{13}$$

But we know that $\gamma_1 = 2\alpha$ and $\gamma_2 = 90° - \alpha$:

$$\frac{l_d}{\sin 2\alpha} = \frac{R}{\sin 90° - \alpha} \tag{14}$$

Using some trigonometric formulas we get

$$\frac{l_d}{2 \sin \alpha \cos \alpha} = \frac{R}{\cos \alpha} \quad \Longrightarrow \quad R = \frac{l_d}{2 \sin \alpha}. \tag{15}$$

By substituting R in (9), then the steering angle becomes

$$\psi = \arctan\left(\frac{2l \sin \alpha}{l_d}\right). \tag{16}$$

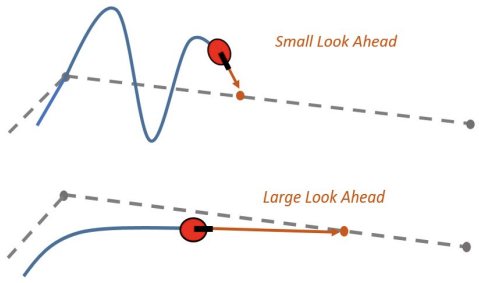


Figure 7: The effect that look ahead distance has on the car movement [11].

Changing the look ahead distance will change how the car follows the path. Choosing a small look ahead distance will compel the car to follow the path even when small deviations are present in the path leading to a lot of adjustments and oscillation left and right. Meanwhile choosing a big look ahead distance will allow the car to ignore some minor detour along the path but it will also take longer time to converges to the path when making a turn.

2.8 Servo and stepper motors

A servo motor is a type of electric motor that is designed for precise control of position and speed. There are both AC and DC servo motors, and the speed and position can be controlled by pulse width modulated (PWM) signals.

Stepper motors are another kind of electric motor which use a rotating magnetic field from coils surrounding the shaft to achieve a rotating motion. What differs stepper motors from other electrical motors is that they have a higher density of magnetic coils, allowing a full rotation to be divided into smaller segments. Each "step" corresponds to the magnetisation of a coil. By energising the coils in a specific sequence, the shaft of the motor can be controlled in a flexible way. If the magnetisation is done at a high frequency, the motion will be smoother. To be able to handle the high frequency output required for smooth operation, certain motor driver boards might be required [12].

2.8.1 Servo motor control with PWM signals

Pulse width modulation (PWM) is a technique used to control the position or speed of servo motors. It involves sending a series of pulses with varying pulse widths to the motor at a fixed frequency. The width of the PWM pulse determines the servo motor's position. A wider pulse corresponds to a larger deviation from the center position, as seen in Figure 8.

Certain pins on microcontrollers or single-board computers, like the Raspberry Pi 4, are usually dedicated to PWM control signals. Python libraries like `pigpio` can be used to specify frequency and pulse width. Example of code that maps a desired angle to the corresponding pulse width and sends it from a RP4 to a servo motor is seen below:

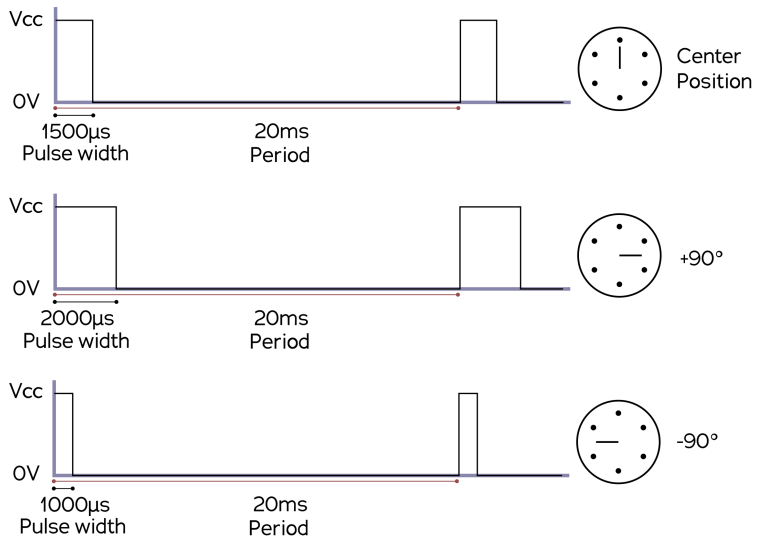


Figure 8: Pulse width modulation for control of a servo motor angle [13].

```
import pigpio

# Define the GPIO pin for the servo motor
servo_pin = 13

# Set up the pigpio library for the servo
pi = pigpio.pi()

# Initialise a servo angle
angle = 90

try:
    while True:

        # Map the angle to the servo's pulse width
        # (500-2500 microseconds)
        pulse_width = int(500 + (angle / 180.0) * 2000)

        # Move the servo to the specified angle
        pi.set_servo_pulsewidth(servo_pin, pulse_width)
```

2.8.2 Stepper motor control

Stepper motors have multiple coils wound around the stator. These coils are energized in a specific sequence to create magnetic fields that interact with the rotor's teeth. To move the motor, the coils are energized in a precise sequence, causing the rotor to rotate in discrete steps. The number of steps per revolution is determined by the motor's design. A stepper motor is easily controlled using a Raspberry Pi and a stepper driver such as TMC2209, by sending signals using the Pi:s STEP and direction (DIR) pins. These signal will be synchronised by the driver and sent to the different coils of the stepper motor to induce a rotational movement of the rotor.

The direction pin (DIR) is instead a binary signal used to control the direction of the rotation (either clockwise or anti-clockwise). The `PiGPIO` library in Python can be used for stepper motor control.

2.9 The obstacle avoidance system

The obstacle avoidance system makes use of three ultrasonic sensors placed on a row at the front of the car (left, center and right). Once an obstacle is detected in front of the car while navigating to a goal, the idea is to stop and plan another path to a temporary goal point (x_t, y_t) located either to the left or to the right of the obstacle. After reaching this temporary position, a new path to the original goal position is calculated and followed. Figure 9 below shows an example of this setting.

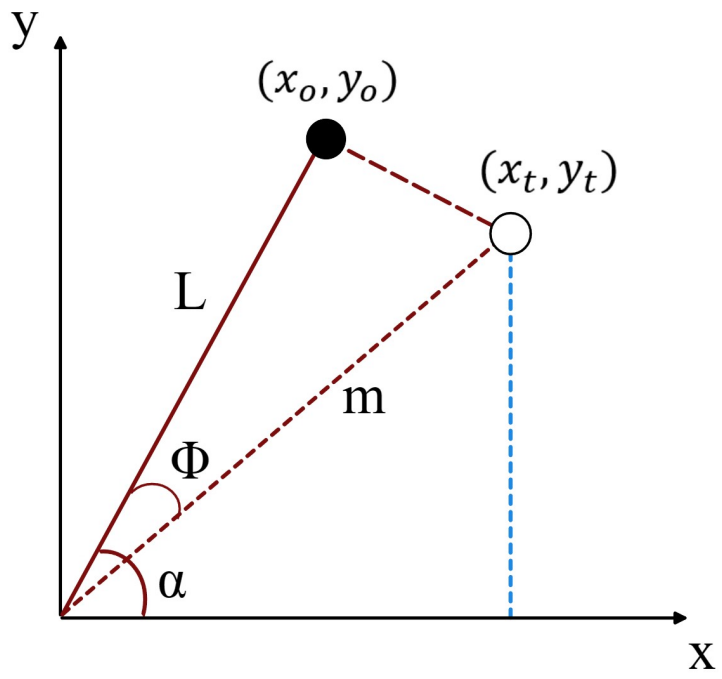


Figure 9: A detected obstacle at position (x_o, y_o) at a distance L from the front of the car. Using the car's orientation α in the map's coordinate system together with the avoidance angle Φ, a temporary goal position (x_t, y_t) is calculated at a distance m from the car.

The orientation angle α of the vehicle is known at all times in the global reference frame from the SLAM algorithm. If the leftmost ultrasonic sensor detects an obstacle at distance L, then the car stops and wants to calculate the position (x_t, y_t) along a line 90 degrees to the right of the line L between the obstacle and the car. Since an infinite number of points lie on this line, the pre-determined avoidance angle Φ is used to pick out the final position so that the car has sufficient clearance around the obstacle. Since the position of the car is also known at all times, (x_t, y_t) can be expressed in terms of its distance to the stopped car, m as

$$\cos(\Phi) = \frac{L}{m} \longrightarrow m = \frac{L}{\cos(\phi)}. \tag{17}$$

The temporary goal position in relation to the current position of the car

can then be calculated as

$$x_t = m\cos(\alpha - \Phi) = \frac{L}{\cos(\Phi)}\cos(\alpha - \Phi), \tag{18}$$

$$y_t = m\sin(\alpha - \Phi) = \frac{L}{\cos(\Phi)}\sin(\alpha - \Phi). \tag{19}$$

Figure 10 shows the temporary goal position (x_t, y_t) calculated using (18) and (19) for different odd multiples of $\alpha = 45$ when $\Phi = 35$ and $L = 1$.

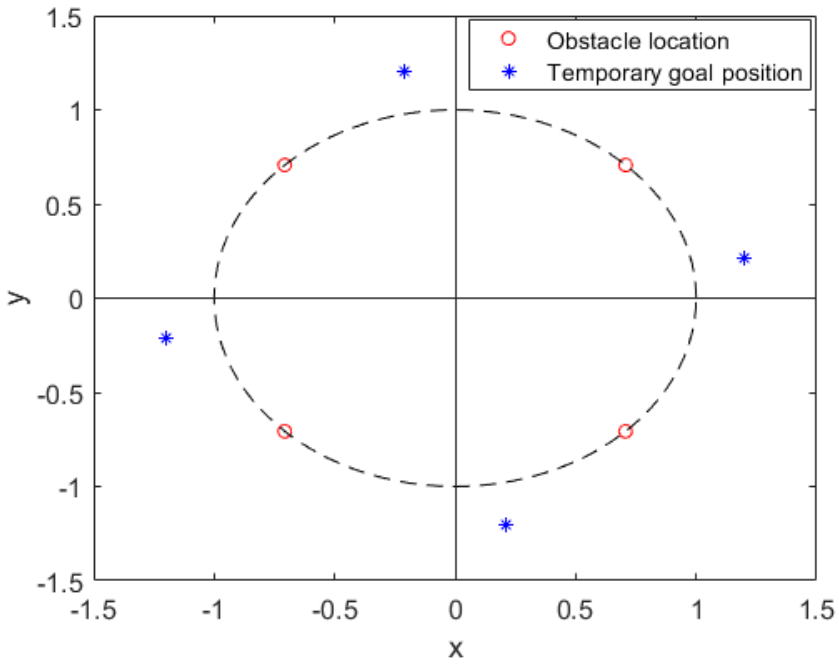


Figure 10: The temporary goal positions (blue) for different obstacle locations (red). In this case α is odd multiples of 45°, $\Phi = 35°$ and $L = 1$.

3 Implementation

3.1 Overview of the system

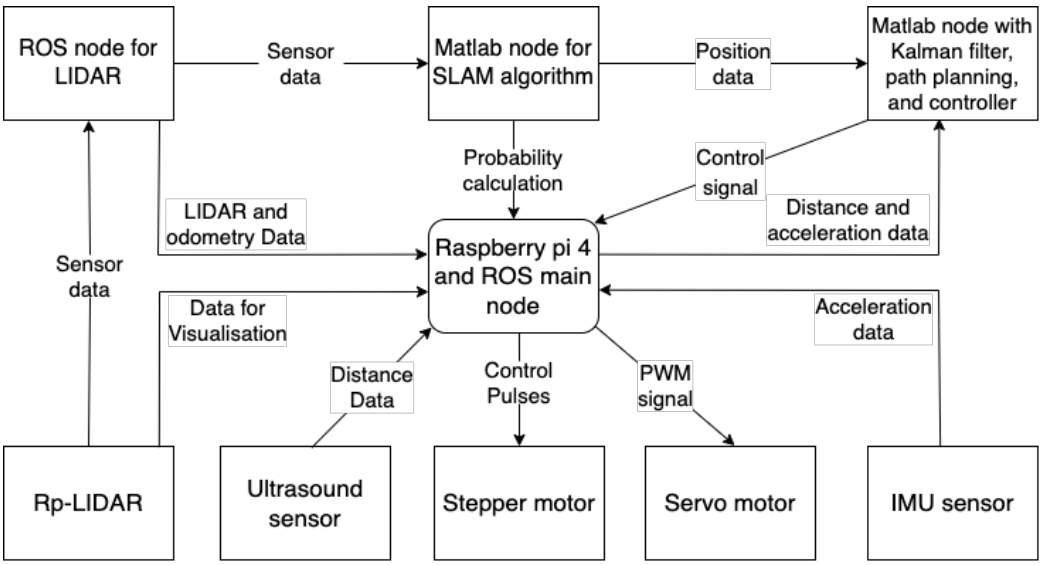


Figure 11: A flowchart of the entire system

As evident from the flowchart in Figure 11, the project can be categorized into two principal domains: software, depicted in the upper section of the flowchart, and hardware, illustrated in the lower section. The Raspberry Pi serves as the central processing unit, bridging these two domains to realise the functionality of the RC car.
The complete schematic of ROS nodes and topics can be seen in Figure 12. It gives an overview of the different nodes that build this project and all the topics that these nodes subscribe to or publish. The ellipses represents the nodes, while the rectangles represents the topics. ROS gather data from the different sensors mounted on the car, and process it in each of the three nodes to the left, namely /**my_imu_node**, /**ultrasound_node**, and /**rplidarNode**. Each of these three nodes publish its processed data using its corresponding topics, /**msg_from_imu**, /**distance** and /**scan** respectively. /**Matlab_global_node** acts a subscriber to all three topics. It takes the processed data as an input, performs the required calculation for path planning and navigation and outputs the steering angle /**Psi** as a topic,

which means /**Matlab_global_node** is both a subscriber and a publisher at the same time. To keep the number of publishers low, several messages other than just the steering angle are published on the /**Psi** topic:

- *Stop*: The stepper motor immediately.
- *Go back*: The stepper motor turns the wheels backwards for a certain duration of time.
- *Go forward*: The stepper motor turns the wheels forward continously.

The *Go forward* message is the default message on the topic together with the steering angle, causing the car to move forward unless other messages are published. Finally, /**angle_controller_node** register the steering angle ψ and send the appropriate signals to both stepper and servo motors to enact that steering angle.

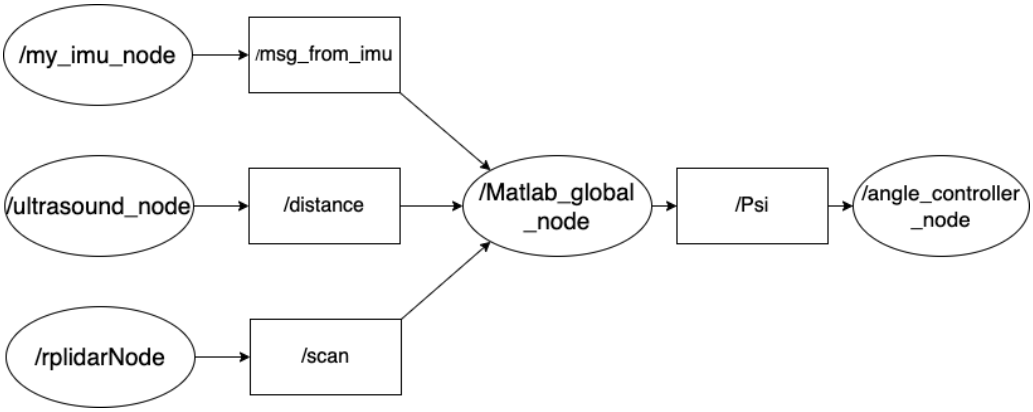


Figure 12: ROS nodes and topics in this project

Finally, it is important to mention that /**Matlab_global_node** takes into consideration the physical limitations of both the model and the motor and adjust the steering angle ψ accordingly. This is evident from the fact that our controller outputs angles between $-45°$ and $+45°$ while the servo motor installed in this car has an operational range between $0°$ and $+40°$, $0°$ coincide with the wheel locked all the way to the left, and $40°$ all the way to the right, /**Matlab_global_node** transforms that angle to a corresponding steering angle ψ in the servo motor's operational range.

3.2 Hardware and components

3.2.1 Computing resources

The main computing resource in this project is a Raspberry Pi 4 (RP4) with the Ubuntu 20.04 Focal Fossa version of the Linux operating system installed. The ROS master node is initialised on the RP4.

Additionally, two laptop computers running Windows and macOS were used. The reasons for using these are twofold:

1. In order to be able to control the Raspberry Pi remotely.

2. Running computationally expensive calculations and visualisation. The RP4 has limited processing power and storage, and being able to run future navigation algorithms and SLAM mapping programs simultaneously is not guaranteed.

The remote control of the RP4 was done through Secure Shell (SSH), which is a network protocol and cryptographic technology used for secure communication between two computers over an unsecured network, such as the internet. SSH allows users to log in to a remote system and execute commands as if they were physically present at the machine's console [14].

Regarding the second reason, having a computer with Matlab installed opens up many opportunities since it offers relevant toolboxes with pre-made, computationally efficient, functionalities. One example of this is the SLAM toolbox, which can both create a SLAM map from LIDAR scans and then export this map as a 2D-occupancy grid [15]. The occupancy grid can then be saved and used for navigation algorithms and visualisation.

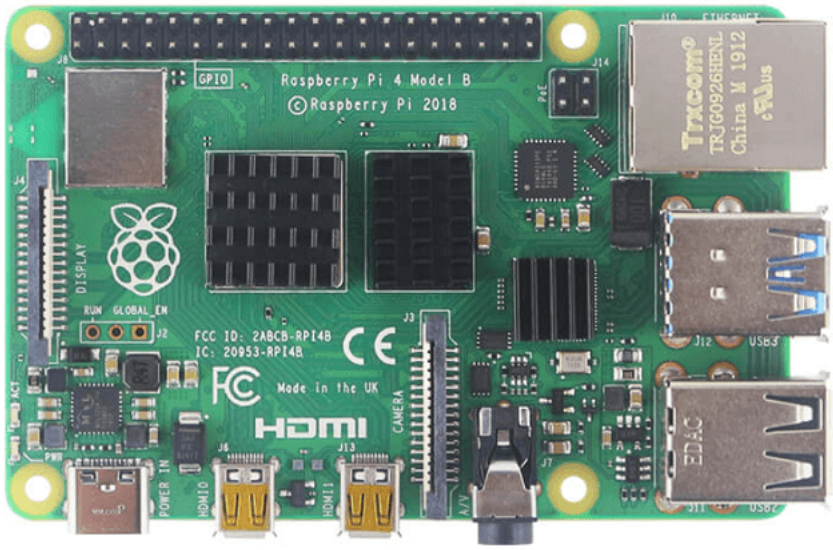


Figure 13: The Raspberry Pi 4 used for controlling the motors, interfacing with sensors and running the ROS master node.

3.2.2 The RC car

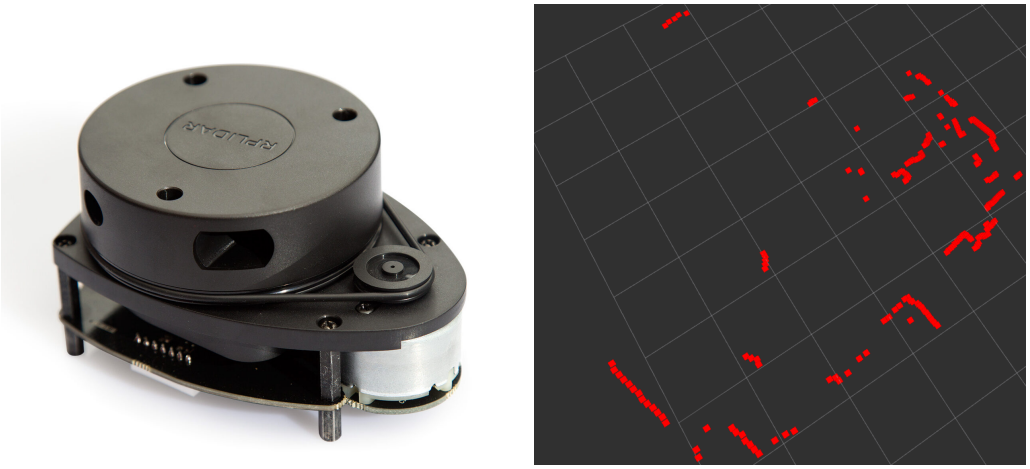
The car in this project is modified RC car with its top and receiver removed, leaving only the chassis . It has an Ackermann steering geometry, meaning that it turns by tilting its front wheels [16]. All four of the wheels are actuated by the same stepper motor, meaning that the car cannot turn in place like a differential drive robot.

For steering, a servo motor is used, while for driving the wheels a stepper motor is used. The cars motors are powered by a 11.1V lithium-polymer (Li-po) battery. The voltage is regulated from 11.1 Volt to an appropriate level for each motor by two L298N motor driver boards.

3.2.3 Using the LIDAR sensor

The LIDAR sensor used in this project is the RPLIDAR A1, which is a 360-degree 2D laser scanner developed by SLAMTEC. It offers a scanning range of up to 6 meters and is suitable for indoor environment scanning and SLAM. The LIDAR system is based on laser triangulation as depicted in Figure 2.

It consists of a range scanner system and a motor which turns the scanner repeatedly. In this way the system is able to perform 360 degree scans [4].



(a) The A1M8 model of RPLIDAR A1 (b) Live LIDAR data in RViz.

Figure 14: (a) The RPLIDAR A1 sensor from Slamtec [4], and (b) live LIDAR data visualized in the ROS environment RViz.

The RPLIDAR A1 outputs sampling data continuously through a communication interface, primarily using a 3.3V-TTL serial port (UART), with the option for USB customization. These data frames contain information about distance and angle values between the object and the scanner [4, p. 6]. Each data frame output contains:

1. Distance (in mm) between the rotating core of the LIDAR and the sampling point.
2. The current heading angle at the time of the measurement, expressed in degrees.
3. A quality level, indicating the reliability of the measurement.
4. A start flag (as a boolean value). This flag indicates the beginning of a new scan.

Host systems have the ability to halt or initialise this data output by communicating through the USB connection.

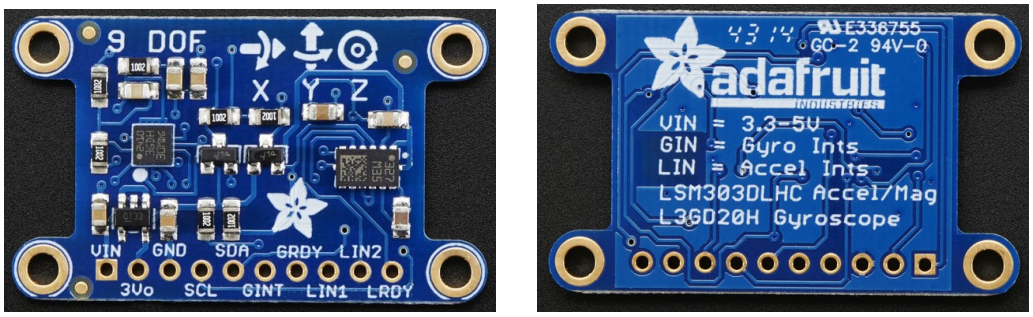
3.2.4 Adafruit 9DOF IMU

The inertial measurement unit (IMU) used in this project is the Adafruit 9-DOF IMU Breakout Board seen in Figure 15. It contains:

- L3DG20H 3-axis gyroscope,
- LSM303 3-axis compass,
- LSM303 3-axis accelerometer.

The gyroscope provides information about the angular velocity, while the magnetic data from the compass provides information about the current orientation. Finally, the accelerometer provides acceleration measurements [7].

All components use the I2C communication protocol [17], allowing for data transfer using the `SDA/SCL` pins. Those pins are connected to its counterpart on the Raspberry Pi, namely 3 & 5.



(a) Top view. (b) Bottom view.

Figure 15: The Adafruit 9-DOF IMU Breakout Board [7].

3.3 Software and development tools

The main operating system for this project is Ubuntu 20.04 (Focal Fossa), and the main reason for this choice is its compatibility with ROS, specifically the ROS1 distribution "Noetic". The operating system was was installed on the RP4's SD card using a Raspberry Pi imager. The same operating system was also downloaded on our laptops using a virtual machine software called "VMware fusion".

3.3.1 The ROS system

This project opted to use ROS1 and adopted the end of life distribution called "Noetic". Maturity and the extensive range of packages that control sensors were the main reasons for deciding which version and distribution this project used. Installing ROS on all devices was accomplished by following instructions found on this website (`http://wiki.ros.org/noetic/Installation/Ubuntu`). Due to the fact that ROS is "language blind", several languages were used to write the codes in each node, but the most commonly used languages in this project were C++ and Python. Certain supplementary files are also included in all the packages. Those files ranges from launch files used to start the package, all the way to YAML & XACRO files to write the configuration parameters in a humanly readable manner.

In a project of this size, several software programs are needed to achieve the task on hand:

- VS-code: The main integrated development environment (IDE) for this project due to the different add-on packages which can handle both Python and C++.
- Git: Git is a version control system that tracks changes done in code. Git is used in this project to clone some of the packages from GitHub to our main repository.
- Thonny: Thonny is another IDE, mainly for Python. It was used during the early phases of this project and to write simple scripts outside of ROS.
- Matlab: The R2021a version was used to access the ROS and SLAM toolboxes.
- Terminal: The terminal itself is not a software program, but it has been used extensively during this project, mainly to establish a SSH connection between the Raspberry Pi and our computers. It has also been used to edit some of the text files, to install some ROS libraries and other needed software. Additionally, it is used to start the ROS main node, launch all packages and check the running topics in our ROS main node. The interaction between the user and the terminal is the single most important task when working with ROS.

3.3.2 Interfacing the LIDAR sensor with ROS and Matlab

To be able to control the LIDAR sensor and interface it with ROS, it was connected to a USB port on the RP4. A node for data retrieval and publishing was then initialised by running the command `roslaunch rplidar_ros view_rplidar.launch` in the terminal with the `rplidar` ROS package installed on the RP4 [18].

With the LIDAR publisher node continuously publishing data to a topic called `/scan`, the data could be easily be accessed and visualised in the ROS visualisation environment RViz. Figure 14b shows an example of this. The live data could also be visualised live in Matlab by creating a `/scan` subscriber.

The LIDAR data was imported through ROS to a Matlab script running a SLAM algorithm. After controlling the car manually to drive and explore the workspace, the final SLAM map was converted with toolbox functionalities to a 2D occupancy grid, which can be used for future navigation algorithms.

3.4 Hardware implementation

To implement the system, the first thing that was done was to connect by SSH to the RP4, which was powered by a 5 V powerbank. The LIDAR sensor was then plugged into the one of its USB ports. After that, the lidar node in ROS was initialised wirelessly, allowing for both visualisation of the live data and for saving it.

The next step was to connect the power from the 11.1 V batteries to the cars motor driver boards. The boards transform the battery voltage to an appropriate value for the motors. Connections between the motor signal pins on the RP4 (PWM, STEP, DIR, EN) and the motors themselves were then wired. In the case of the stepper motor, an ARCELI A4988 stepper motor driver was used to convert the low power RP4 signals into higher-power signals that the stepper motor can use.

Similarly, the IMU and ultrasonic sensors were interfaced with by using two Python scripts. This was done using the `adafruit_lsm303_accel` and `adafruit_l3gd20` libraries in the case of the IMU.

The ultrasonic sensor has four pins, VCC, TRIG, ECHO and GND. The TRIG pin was connected directly to a GPIO pin on the RP4, while the ECHO pin was connected via a two resistor voltage divider to a GPIO pin. The distance values could then be obtained using the `ultrasonic` Python library.

3.5 Matlab implementation

The basic structure of the navigation algorithm in Matlab is seen below:

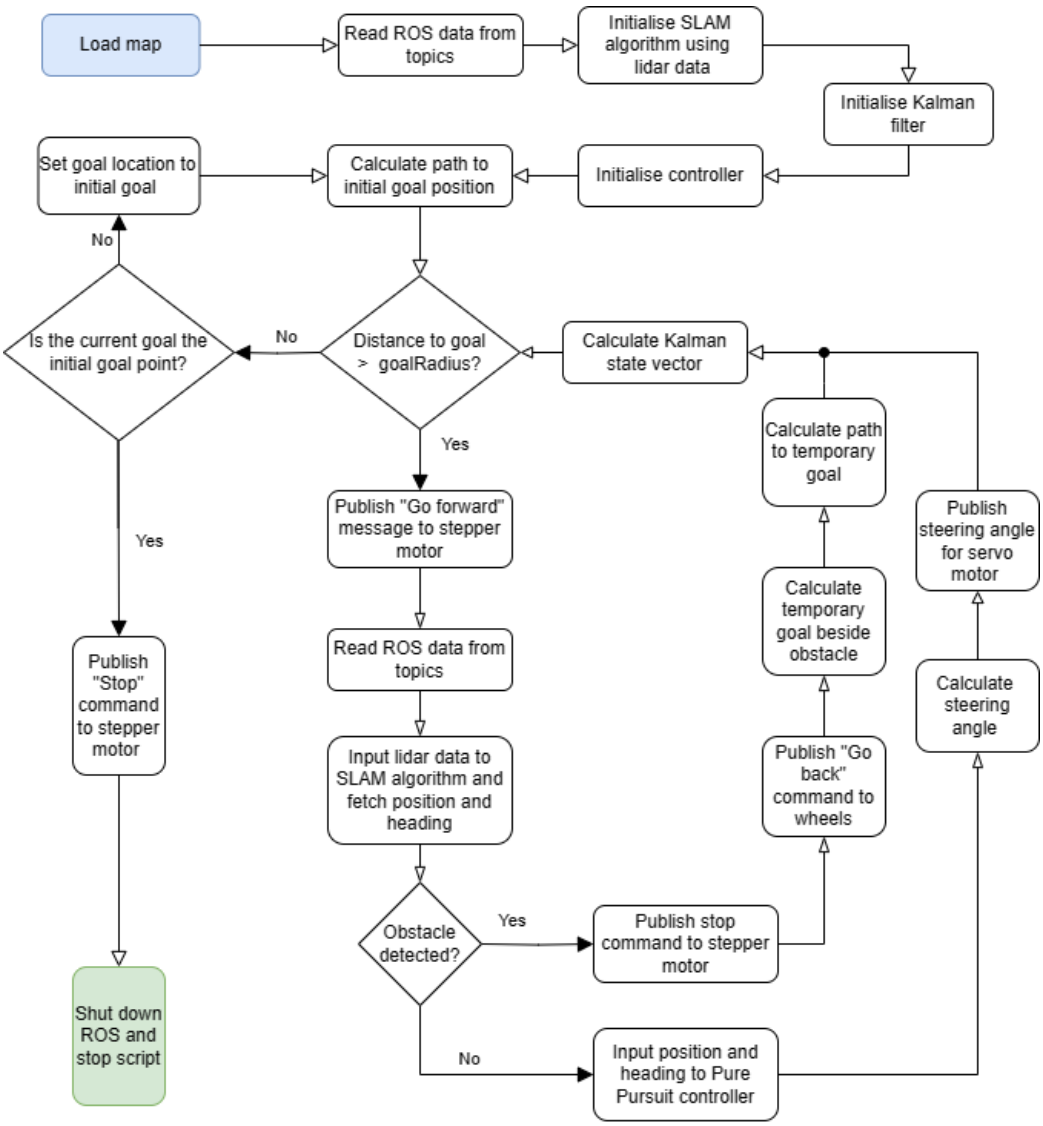
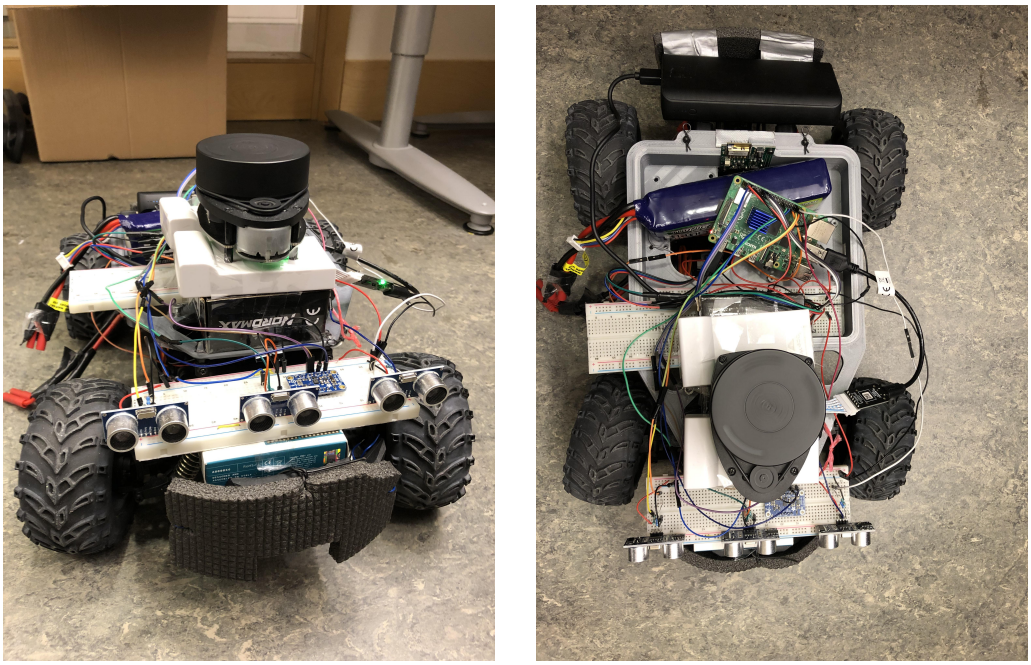


Figure 16: A flowchart of the structure of the Matlab algorithm for path planning, state estimation and path following.

4 Results and discussion

This section presents the results and discusses how well the results fulfilled the specifications. The problems encountered during this project and the solutions to them are also discussed. The final car setup can be seen in Figure 17.



(a) Front view.

(b) Top view.

Figure 17: The RC car with the motor battery, powerbank, LIDAR, IMU, ultrasound sensors and RP4 attached.

4.1 Overview of results

We were successful in creating a fully autonomous car in the sense that it can navigate from a starting point to a predetermined target point and back to its starting position, while avoiding obstacle along the way. This was accomplished by achieving the following points:

- Creating a map of the environments using LIDAR and SLAM.

- Estimating the current position and heading using LIDAR SLAM and a Kalman filter.
- Applying a PRM path planning algorithm which finds a path that the car will follow from the starting point to the target coordinates.
- Implementing a controller to keep the car along the path and makes adjustment in case of deviations.
- Detecting obstacles during the car's journey, and initialising the movement on an alternate avoiding path.

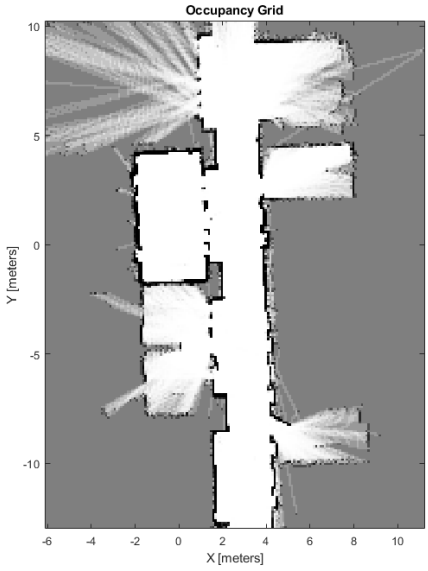
4.1.1 SLAM results and occupancy map

The LIDAR was successfully integrated with ROS, and its real-time scan data was published to a ROS topic, enabling wireless data access and live visualization in Matlab.

The LIDAR data was first used to create an "offline" SLAM map when the car was being manually controlled to drive around the workspace. This SLAM map was then saved and converted to an occupancy grid. The SLAM map and its corresponding occupancy grid can be seen in Figure 18.

During the autonomous navigation, the SLAM algorithm was also used in an "online" setting to obtain estimates of position and heading, while the previously constructed occupancy grid was used for path planning.

(a) SLAM map of the workspace created from LIDAR scans.



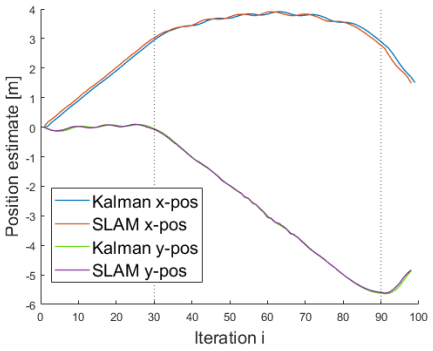
(b) An occupancy grid of the office space.

Figure 18: A SLAM map of the office space and the corresponding occupancy grid. The SLAM estimate of the LIDAR sensor's position over time is seen in blue in (a).

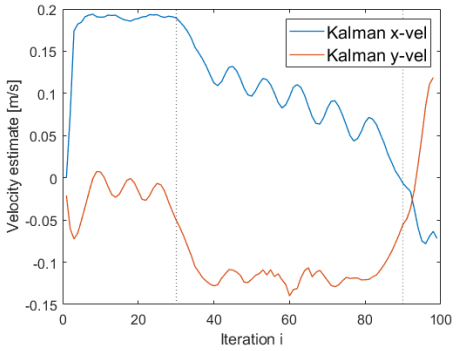
4.1.2 Kalman filter results

As seen in Figures 19 and 20a, we successfuly implementated and integrated a Kalman filter into the navigation algorithm. The filter estimates the position, velocity and acceleration of the car. The acceleration measurements from the IMU are also seen in Figure 20b.

Since the SLAM position estimates were considered highly accurate and reliable, their corresponding standard deviation specified in (8) was set to much lower values than the standard deviations for the noisy IMU measurements. This resulted in that the Kalman filter in practice ignoring the IMU measurements and instead only using the SLAM position estimates and the kinematic model to estimate the acceleration, velocity and position of the car.

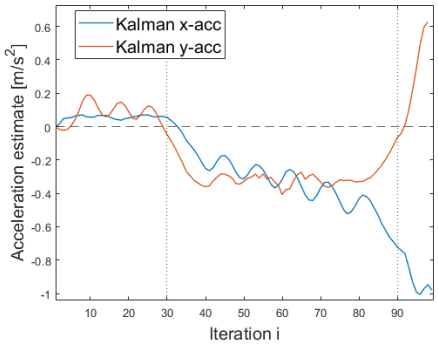


(a) Position estimates from the SLAM algorithm and the Kalman filter.

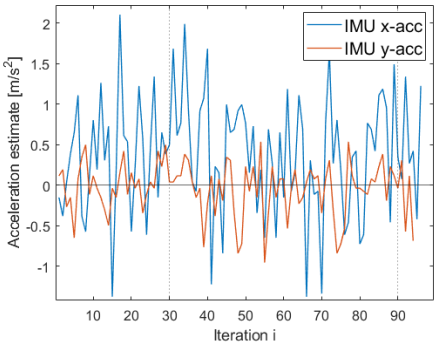


(b) Linear velocity estimates from the Kalman filter

Figure 19: Kalman filter position and velocity estimates from an autonomous navigation run. The LIDAR-SLAM position estimates are also seen in (a).



(a) Linear acceleration estimates from the Kalman filter



(b) Linear acceleration measurements from the accelerometer on the IMU.

Figure 20: Acceleration estimates from the Kalman filter and the IMU.

4.1.3 Results of path planning using PRM

We successfully implemented the PRM algorithm from Matlab into the navigation algorithm and were able to calculate paths between any two non-occupied positions in the occupancy map. The occupied spaces in the occupancy grid was artificially inflated to prevent the PRM algorithm from finding paths which were too close to walls.

The specific parameters chosen for this algorithm are as follows:

1. Number of nodes: 1500
2. Wall inflation: 40 cm
3. Maximum connection distance between the nodes: 1 m

Figure 21 shows a PRM map with inflated walls and a proposed path between two points.

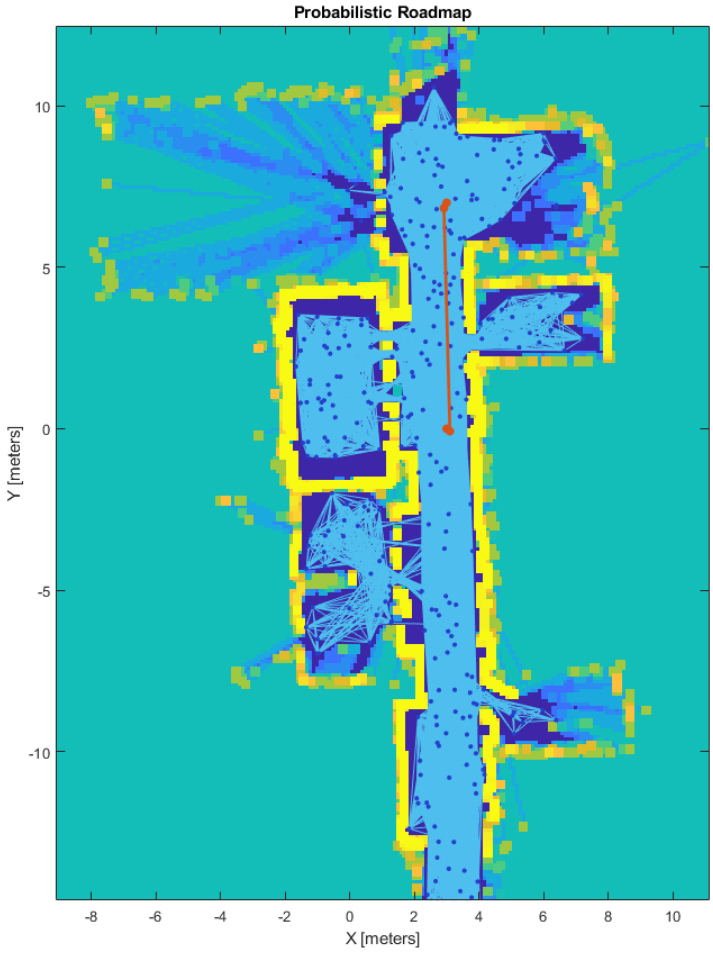


Figure 21: A PRM map of the workspace made using the occupancy grid in Figure 18b. The inflated walls are seen in yellow and a proposed path between two points is seen in orange.

4.1.4 Pure Pursuit results

By using the position and heading estimates of the car as inputs to the Pure Pursuit controller, the necessary steering angle for following a PRM path could be calculated and sent to the servo motor. The main parameters provided to the controller were:

1. Desired linear velocity: 0.15 m/s
2. Max angular velocity: 0.5 rad/s
3. Look ahead distance: 0.65 m

These parameters were adjusted according to the size of the workspace and the complexity of the path.

The controller was able to steer the car along a path to a goal and then steer it back to its starting position. An example of this is seen in Figure 22. It can be seen that the car makes an almost 360° turn before continuing on a linear path towards its starting position. This is because the car is initially oriented in the opposite direction of the path towards the goal.

Figure 22: The planned PRM path (dashed lines) and the car's position over time (blue circles) when navigating autonomously back to its original starting position near (0,0).

In Figure 23a the car can be seen following a complex path from one room to another, passing by two narrow doorways and avoiding walls along the way. Some oscillatory movement due to a short look ahead distance can be seen.

Figure 23: The position of the car as estimated by the Kalman filter (red circles) and the SLAM algorithm (blue circles) when following a complex path between two rooms in the workspace.

4.1.5 Obstacle avoidance results

Applying equations (18)-(19) and the algorithm from section 3.5, the car was able to detect the obstacles lying ahead and implement several changes to its course to avoid them. Figure 24 shows how the car handles two obstacles, depicted as black triangles.

When an obstacle is detected, the car goes backwards for a brief moment and calculates a new and temporary goal position (seen as $*$ in the map) to the side of the obstacle. Which side of the obstacle the new goal lies on depends on which ultrasonic sensor that detected it. The car then navigates to this temporary goal, and when it is reached a new path is calculated and followed to the original goal position.

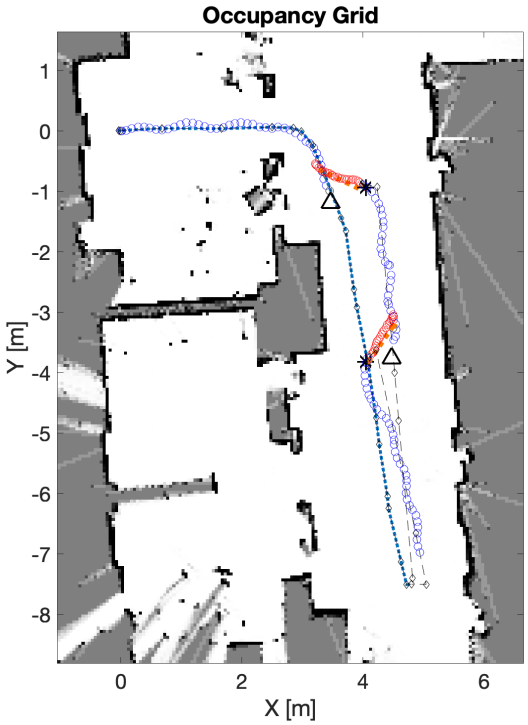


Figure 24: Movement of the car when detecting two obstacles along a path. When an obstacle in front of the car is detected, its position is recorded and displayed on the map as black triangle. A new temporary goal position ($*$) is calculated at a 90 degree angle from the line between the car and the obstacle. The path to this goal is highlighted in red.

5 Discussion

From the results it is clear that we have succeeded with creating a ROS and LIDAR-based system which makes a car able to navigate autonomously to a specified position and back.

The autonomous car works as intended, but it can be improved in several ways. The first improvement that can be made is to use a less noisy IMU. Since the car's acceleration is below the noise level of the IMU at all times, it was deemed unusable and made unnecessary by increasing its the standard deviation in the measurement covariance matrix of the Kalman filter. Using a less noisy IMU might therefore improve performance by making the Kalman filter position estimates more accurate, and thus making the inputs to the navigation system more reliable.

The second improvement which can be made to system is to change the path following controller to perhaps a PID or MPC controller. This might improve path following by making the car follow the path with less overshoot. It might however increase the computational complexity of the controller, making the system slower.

The third improvement which can be made is to use a more physically accurate kinematic model of the car. The current model which is used approximates the car as a bicycle, which works in this case when the car travels at low speeds. It might however not work as well at higher speeds and if a wider car is used.

The fourth and final improvement that can be made is to replace the LIDAR and ultrasonic sensors with a camera and then use camera-based SLAM algorithms to map the surroundings, detect and identify obstacles. This would reduce the number of components in the system and allow for additional capabilities such as object following.

5.1 Challenges and solutions

Numerous challenges were encountered throughout the project, many of which were attributed to a limited understanding of ROS. Some of the problems we faced were:

1. Incompatible packages: Several packages we found online for ROS were either compatible with an older or a more novel distributions of ROS.

2. Inadequate sensor-specific packages: Some of the sensors do not have ROS packages available online. In such cases we had to write our own code to fetch the data from sensors and publish the results to a topic. This was the case for the IMU and for the control of both motors.

3. Misconfigured parameters within packages: Each package encompasses multiple parameters governing sensor behavior. These parameters can vary between sensor models and operational conditions. Addressing these challenges required comprehensive research into individual sensors and adjustments of parameters based on the specific environmental conditions of sensor operation.

These challenges were mostly overcome as we got more familiar ROS and discovered Python packages fitting for our hardware. Some other difficulties not attributed to ROS were the following:

1. Hardware issues: Both motors ceased to work at some point during this project due to faulty connections and burned driver boards. Even when we managed to get them working, the motors started to behave in unpredictable ways when the temperature of the driver boards rose to a certain temperature, making the car not move as intented.

2. Unable to detect obstacles: We started by using one ultrasound sensor, and found it to be inadequate in detecting all obstacles. We upgraded to three ultrasound sensors as seen in Figure 17, but still some of the obstacles were harder to detect than others. This is due to the narrow field of detection these sensors have, as they can only detect obstacles which lay ahead linearly in their filed of sight. A possible solution would be to upgrade to a camera based system for detecting the obstacles.

3. Fine-tuning the parameters: Finding the right parameters for the controller, and the model took a long time to obtain, especially the look ahead distance of the controller. We wanted the car to move linearly without oscillating, but by increasing the look ahead distance, the car failed to navigate around walls since it starts taking too large shortcuts when the path turns.

The inclusion of Matlab with the project, which was initially not planned, allowed us to progress further than expected. The SLAM capabilities that Matlab offers were easier to use and faster than the SLAM functionalities in

ROS, such as `hector_slam`. We also discovered many useful toolbox functionalities which greatly helped us with path planning and implementation of the controller.

5.2 Conclusions and further work

Based on the final results of this project, it can be conclusively stated that the objectives set forth at the project's outset have been successfully realised. We have successfully implemented a fully autonomous car through sensor and actuator fusion using ROS. Ultimately, based on the grading criteria set at the beginning of this project, we believe that we have achieved grade five as stated in the grading critera.

Future work that might improve the system may include developing a better controller, improving obstacle detection capabilities and including a camera to make use of camera-based SLAM algorithms. Some of the hardware components used in this project might require updating as well, most notably the IMU.

6 References

[1] ROS.org, "ROS/Introduction." Available: http://wiki.ros.org/ROS/Introduction, 2018. [Online; accessed 10-October-2023].

[2] Mathworks, "Get Started with ROS." Available: https://se.mathworks.com/help/ros/ug/get-started-with-ros.html, 2023. [Online; accessed 10-October-2023].

[3] ROS.org, "Writing a simple publisher and subscriber." Available: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29, 2018. [Online; Accessed Oct. 09, 2023].

[4] SLAMTEC, "RPLIDAR A1, Introduction and Datasheet." Available: https://www.slamtec.ai/home/rplidar_a1/, 2020. [Online; accessed 10-October-2023].

[5] B. B. Deepak, M. V. Bahubalendruni, and B. B. Biswal, "Development of in-pipe robots for inspection and cleaning tasks: Survey, classification and comparison," *International Journal of Intelligent Unmanned Systems*, vol. 4, pp. 182–210, 2016.

[6] H. Durrant-Whyte and T. Bailey, "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms,"

[7] Adafruit Industries, "Adafruit 9-DOF IMU Breakout - L3GD20H + LSM303." Available: https://www.adafruit.com/product/1714, November 2014. [Online; Accessed Oct. 08, 2023].

[8] C. Liu, T. Kadja, and V. P. Chodavarapu, "Experimental evaluation of sensor fusion of low-cost UWB and IMU for localization under indoor dynamic testing conditions," *Sensors*, vol. 22, no. 21, 2022.

[9] P. Polack, F. Altché, B. d'Andréa Novel, and A. de La Fortelle, "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, pp. 812–818, 2017.

[10] M. Theers and M. Singh, "Pure pursuit."

[11] Mathworks, "Pure Pursuit Controller." Available: https://se.mathworks.com/help/robotics/ug/pure-pursuit-controller.html, 2023. [Online; Accessed Nov. 19, 2023].

[12] Shawn Dietrich, "Servo Motor vs Stepper Motor: Understanding the Differences." Available: https://control.com/technical-articles/servo-motor-vs-stepper-motor-understanding-the-differences/, 2022. [Online; accessed 11-October-2023].

[13] Wikimedia user Hforesti, "Servomotor Timing Diagram." Available: https://commons.wikimedia.org/w/index.php?curid=101951622, 2021. [Online; accessed 11-October-2023].

[14] Wikipedia contributors, "Secure shell — Wikipedia, the free encyclopedia." Available: https://en.wikipedia.org/w/index.php?title=Secure_Shell&oldid=1178959792, 2023. [Online; Accessed Oct. 08, 2023].

[15] Mathworks, "SLAM." Available: https://se.mathworks.com/help/nav/slam.html, 2023. [Online; Accessed Oct. 08, 2023].

[16] Wikipedia contributors, "Ackermann steering geometry — Wikipedia, the free encyclopedia." Available: https://en.wikipedia.org/w/index.php?title=Ackermann_steering_geometry&oldid=1177050788, 2023. [Online; accessed 10-October-2023].

[17] Wikipedia contributors, "I²C — Wikipedia, the free encyclopedia." Available: https://en.wikipedia.org/w/index.php?title=I%C2%B2C&oldid=1178347487, 2023. [Online; Accessed Oct. 08, 2023].

[18] ROS.org, "rplidar — Package summary." Available: http://wiki.ros.org/rplidar, 2019. [Online; accessed 10-October-2023].