Exploring the Deployment of an EKF SLAM System on a Mobile Robot Using ROS2

Simen Galland Grønli
Bendik Rolf Stokke

2022-12-19

Summary

The work presented in this report is done as the specialization project of the main profile Robotics and Automation at the Department of Mechanical and Industrial Engineering at NTNU. The report presents an implementation of an Extended Kalman Filter Simultaneous Localization and Mapping (EKF SLAM) algorithm on a mobile robot. The hardware platform used for developing this system is the mobile robot named Turtlebot3. The robot's light detection and ranging (LiDAR) sensor was the input for the implemented software. This software is based on the second release of Robotic Operating System (ROS2).

This report presents a brief introduction about the SLAM problem. Then it explains the relevant preliminaries to understand the methods used in the experiments. The methodology began by validating the initial implementation of the EKF SLAM on a simple python-based simulator. Then the development continued on ROS2, where the ROS2 middleware was then used to control the robot remotely and transmit data to a remote machine in order to produce the experimental results. At last, the results are presented and discussed.

Experimental results have given insight into difficulties that arise when deploying a system to a real-world scenario. By conducting the experiments it is shown challenging to reproduce results from the ideal simulator in a real-world scenario. The results are comparable, but the real-world test showed worse performance regarding localization and mapping. Less reliable LiDAR readings from the irregular surfaces in the real-world, thus less reliable landmark detections, are assumed to be the reason for this.

Finally, it was found that ROS2 was an effective framework for developing and testing new robotics applications quickly and efficiently. As the prebuilt software for the Turtlebot3 was compatible with ROS2, it enabled rapid prototyping of the EKF SLAM system.

Contents

Chapter 1.

Introduction

1.1. Background and motivation

Either one wants to obtain a detailed map of the environment, or one might seek to measure the location of a mobile robot accurately. By approaching these kinds of problems with SLAM, both of these goals can be fulfilled.

SLAM is an active area of research relevant within many fields, such as autonomous mobile robots. SLAM allows for the construction of a map of an environment while also keeping track of the location of a robot within that environment. SLAM can facilitate path-planning algorithms to navigate the robot through the environment without the need for human intervention.

There are three main paradigms for SLAM: Particle Filter SLAM, Graph-based SLAM and Extended Kalman Filter (EKF) SLAM [6, p. 875]. Particle Filter SLAM is a Monte Carlo method that uses a particle filter to simultaneously track the robot's pose and map. Graph-based SLAM represents the map as a graph and uses optimization methods to find the best estimate of the robot's trajectory and the map. EKF SLAM uses an extended Kalman filter to estimate the robot's pose and map simultaneously. Each paradigm has its strengths and weaknesses, and are best suited to different environments and applications. The EKF-based solution was chosen for this project because of our prior knowledge of the Kalman Filter and its relatively simple implementation.

1.2. Problem Description

This project aims to compare an implementation of EKF SLAM in both a simulator and in the real-world. A Turtlebot3 will be used to collect data in the two environments, and the results will be compared in a stylized manner. To obtain

this, a simple simulator was developed to validate the algorithm before the final implementation.

The report covers the process of implementing the EKF SLAM algorithm and to use this in a ROS2 package to solve the SLAM problem. One of our main research goals is to look into the stability of EKF SLAM and LiDAR landmark recognition in a real-world scenario.

Chapter 2.

Preliminaries

2.1. Robotic modelling

2.1.1. Kinematics of a differential drive mobile robot

The kinematics of a system is the mathematical model which represents the dynamics caused by control actions. Mobile robots are primarily restricted to movement in a planar environment. As a result of this, the number of variables needed to represent its configuration is reduced to three. The robot's configuration in Figure 2.1 can be represented by a Cartesian point, (x, y), and an orientation, θ. This representation is commonly known as a pose, described by vector 2.1.

$$\mathbf{x}_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.1}$$

Controlling a robot using velocities is common in many commercial robots. A differential drive is an example of velocity controlled drive system. Velocity motion model $\mathbf{u}_t$ is defined as

$$\mathbf{u}_t = \begin{bmatrix} v_t \\ w_t \end{bmatrix} \tag{2.2}$$

where v_t and w_t is translational and rotational velocities respectively.

A differential drive robot is a mobile robot driven by independently controlled wheels on either side of the robot body. The gap between control and resulting action gets smaller by abstracting inputs to translation and rotation.

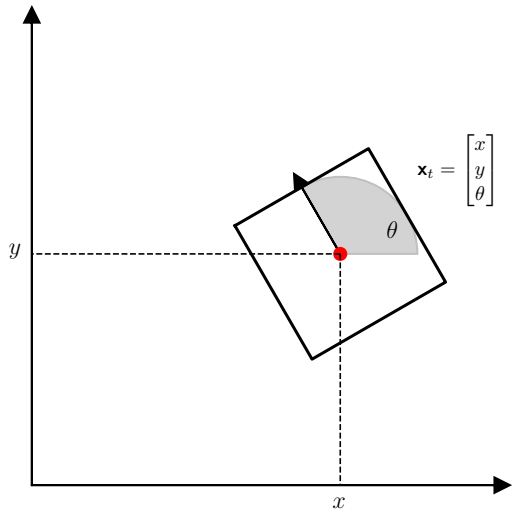

Figure 2.1.: Mobile robot pose in a 2D plane

$$v_t = \frac{v_L + v_R}{2}, \; w_t = \frac{v_L - v_R}{a} \tag{2.3}$$

Equations 2.3 are usually computed from velocities measured from encoders on each of the motors and are directly applicable to the forward kinematic equations

$$\begin{aligned} \dot{x}(t) &= v(t)\cos(\theta(t)) \\ \dot{y}(t) &= v(t)\sin(\theta(t)) \\ \dot{\theta}(t) &= \omega(t) \end{aligned} \tag{2.4}$$

Given the control input from 2.2, it is possible to compute a solution for the odometry, given a known initial pose, using the equations in 2.4. In an ideal world, it is sufficient to know the equations of motion for the differential drive robot and its control input to describe the robot's pose perfectly. However, in the real world, we will notice errors in the motion model, motor controller, and physical model. Essentially, it is impossible to maintain control of the robot's pose solely by dead-reckoning. By gaining knowledge about the uncertainty of

the motion, it is possible to take a probabilistic approach to this issue.

2.1.2. Probabilistic Robots

Probabilistic robotics is a sub-field of robotics that deals with the uncertain nature of information in robotics. It is based on using probability to represent and manipulate uncertain information [8].

Probabilistic robotics takes into account the fact that the results of a control action are never certain due to noise in the system or external factors that need to be accounted for. One advantage of this approach is robustness to changes in the environment and in the system. Probabilistic robotics is particularly effective in robotic navigation, where the robot must deal with uncertainty about its position and the environment's layout.

2.2. Simultaneous Localization and Mapping (SLAM)

SLAM is a problem in robotics where a robotic system acquires a map of its surrounding environment while simultaneously localizing itself relative to this map. The problem is often described with a probabilistic approach; A mobile robot in an unknown environment with uncertain motion will become increasingly challenging to localize as it moves around.

Assuming a robot moves in a map according to control input, and senses its surroundings at regular time intervals. The map, poses, control inputs and measurements are denoted as $\mathbf{m}$, $\mathbf{x}_t$, $\mathbf{u}_t$ and $\mathbf{z}_t$ respectively. Then the SLAM problem is to estimate the current state and the map, given the series of measurements and control inputs up until time t:

$$p(\mathbf{x}_t, \mathbf{m} | \mathbf{z}_{1:t}, \mathbf{u}_{1:t}) \tag{2.5}$$

Figure 2.2 illustrates how these variables are related to each other in the SLAM problem.

It is essential to accurately detect landmarks in SLAM because they provide a known point of reference that the algorithm can use to determine the position and orientation of the robot. Without landmarks, the algorithm would have to rely solely on dead reckoning, which is as mentioned prone to error accumulation. Static landmarks are essential for SLAM because the algorithm needs a known point of reference to determine the robot's position. Therefore, landmark detection is an essential part in a working SLAM system. A Cartesian point represents

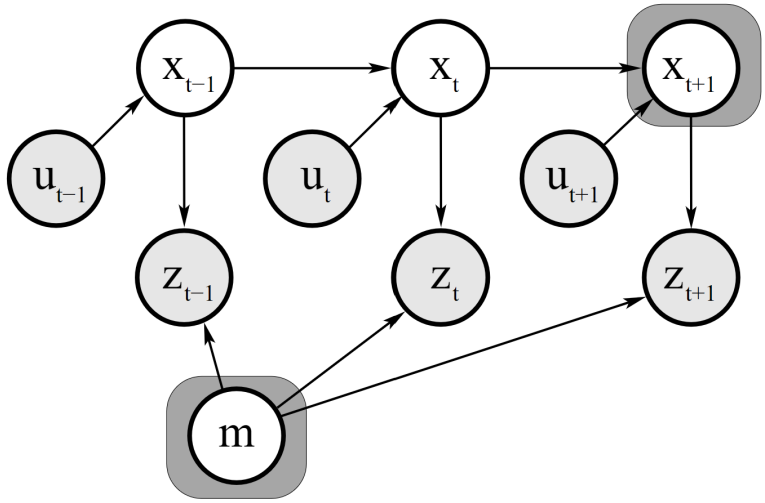

Figure 2.2.: Graphical model of the SLAM problem [8, p. 310]

the configuration of the landmarks as the vector 2.6

$$\mathbf{z}_t = \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.6}$$

2.2.1. The Extended Kalman Filter

The original Kalman filter was presented by Rudolf Kalman in 1960 [4]. This filter was designed for linear systems, but few real-world problems can be modelled as such. Therefore two years later, an extended version of the Kalman filter was developed by NASA [7]. This new model allowed for a nonlinear state space model for the time propagation of the state estimate. This filter has been shown to model mobile robots and drones in real-world scenarios successfully.

2.2.2. Extended Kalman Filter for SLAM

The Extended Kalman Filter (EKF) is a common method used for SLAM, which is mathematically described in algorithm 1. The EKF operates by updating a belief of the robot pose and landmarks in the form of a state and covariance matrix. The EKF predicts the robot's next pose at each time step and updates the state and covariance matrix using a correction step. The EKF has been shown to be effective in SLAM applications. However, the EKF SLAM algorithm has a few limitations and assumptions that must be met in order for it to work

correctly. Firstly, it only works in a feature-based environment where landmarks can be represented by points. This means that the world must be made up of distinguishable landmarks that the algorithm can track. Secondly, there must be a known correspondence of the landmarks between the state vector and the measurement vector. This correspondence must be known a priori for the EKF to work. Finally, the EKF assumes that the system's process and sensor noise is Gaussian-distributed and known.

Assuming that the world is composed of a series of N landmarks or features, each of which can be individually observed and identified, EKF SLAM can create a map by storing the landmarks in a combined state vector $\mathbf{X}_t$ with size $3+2N$ and the associated uncertainties are described in $\mathbf{P}_t$ with size $(3+2N)\times(3+2N)$.

$$\mathbf{X}_t = \begin{bmatrix} \mathbf{x}_t \\ \mathbf{m}_1 \\ \vdots \\ \mathbf{m}_n \end{bmatrix} \mathbf{P}_t = \begin{bmatrix} \Sigma_{\mathbf{x}_t\mathbf{x}_t} & \Sigma_{\mathbf{x}_t\mathbf{m}_1} & \dots & \Sigma_{\mathbf{x}_t\mathbf{m}_n} \\ \Sigma_{\mathbf{m}_1\mathbf{x}_t} & \Sigma_{\mathbf{m}_1\mathbf{m}_1} & \dots & \Sigma_{\mathbf{m}_1\mathbf{m}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{\mathbf{m}_n\mathbf{x}_t} & \Sigma_{\mathbf{m}_n\mathbf{m}_1} & \dots & \Sigma_{\mathbf{m}_n\mathbf{m}_n} \end{bmatrix} \tag{2.7}$$

The EKF SLAM systems belief is represented in 2.7. Here $\mathbf{x}_t$ is the previously mentioned pose of the robot from 2.1 and $\mathbf{m}_n$ contains a point describing a location of a measured landmark.

When initializing the EKF SLAM algorithm, the robot starts at the origin of its reference frame. Additionally, the location of the N landmarks is unknown, and the following initial states and covariance matrix are set.

$$\mathbf{X}_0 = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \end{bmatrix}^T$$
$$\mathbf{P}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \infty & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \infty \end{bmatrix} \tag{2.8}$$

The algorithm of EKF SLAM can be broken down into two steps: state prediction, lines 4-7, and update, lines 8-19 in algorithm 1. In the state prediction step, the Kalman filter predicts the system's current state based on the control input. In the update step, the filter updates the system's state based on measurements.

The state prediction step calculates $\overline{\mathbf{X}}_t$ and $\overline{\mathbf{P}}_t$ by applying the motion update and manipulate the belief of the relevant elements accordingly. [8, p. 313] Further, the update step iterate through every measured landmark, calculates an estimated

Algorithm 1 EKF SLAM Algorithm

Initialize $\mathbf{X}_0$ and $\mathbf{P}_0$

Define covariances of measurement and control input $\mathbf{R}_t, \mathbf{Q}_t$

loop

$$\mathbf{F}_x = \begin{bmatrix} 1 & 0 & 0 & \underbrace{0 \dots 0}_{2N} \\ 0 & 1 & 0 & 0 \dots 0 \\ 0 & 0 & 1 & 0 \dots 0 \end{bmatrix}$$

$$\overline{\mathbf{X}}_t = \mathbf{X}_{t-1} + \mathbf{F}_x^T \begin{bmatrix} -\frac{v_t}{\omega_t}\sin(x_{t-1,\theta}) + \frac{v_t}{\omega_t}\sin(x_{t-1,\theta} + \omega_t \delta t) \\ \frac{v_t}{\omega_t}\cos(x_{t-1,\theta}) - \frac{v_t}{\omega_t}\cos(x_{t-1,\theta} + \omega_t \delta t) \\ \omega_t \delta t \end{bmatrix}$$

$$\mathbf{G}_t = \mathbf{I} + \mathbf{F}_x^T \begin{bmatrix} 0 & 0 & -\frac{v_t}{\omega_t}\cos(x_{t-1,\theta}) + \frac{v_t}{\omega_t}\cos(x_{t-1,\theta} + \omega_t \delta t) \\ 0 & 0 & -\frac{v_t}{\omega_t}\sin(x_{t-1,\theta}) + \frac{v_t}{\omega_t}\sin(x_{t-1,\theta} + \omega_t \delta t) \\ 0 & 0 & 0 \end{bmatrix} \mathbf{F}_x$$

$$\overline{\mathbf{P}}_t = \mathbf{G}_t \mathbf{P}_{t-1} \mathbf{G}_t^T + \mathbf{F}_x^T \mathbf{R}_t \mathbf{F}_x$$

for all observed landmarks, $\mathbf{z}_t^i = \begin{bmatrix} r_t^i & \phi_t^i \end{bmatrix}$ **do**

if landmark is seen before **then**

$$\begin{bmatrix} m_{i,x} \\ m_{i,y} \end{bmatrix} = \begin{bmatrix} m_{t,x} \\ m_{t,y} \end{bmatrix} + \begin{bmatrix} r_t^i \cos(\phi_t^t + \overline{x}_{t,\theta}) \\ r_t^i \sin(\phi_t^t + \overline{x}_{t,\theta}) \end{bmatrix}$$

end if

$$\boldsymbol{\delta} = \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \begin{bmatrix} \overline{x}_{i,x} \\ \overline{x}_{i,x} \end{bmatrix}$$

$$q = \delta^T \delta$$

$$\hat{\mathbf{z}}_t^i = \begin{bmatrix} \sqrt{q} \\ \arctan 2(\delta_x, \delta_x) - \overline{x}_{t,\theta} \end{bmatrix}$$

$$\mathbf{F}_{x,i} = \begin{bmatrix} 1 & 0 & 0 & 0 \dots 0 & 0 & 0 & 0 \dots 0 \\ 0 & 1 & 0 & 0 \dots 0 & 0 & 0 & 0 \dots 0 \\ 0 & 0 & 1 & 0 \dots 0 & 0 & 0 & 0 \dots 0 \\ 0 & 0 & 0 & 0 \dots 0 & 1 & 0 & 0 \dots 0 \\ 0 & 0 & 0 & \underbrace{0 \dots 0}_{\text{2i-2}} & 0 & 1 & \underbrace{0 \dots 0}_{\text{2N-2i}} \end{bmatrix}$$

$$\mathbf{H}_t^i = \frac{1}{q} \begin{bmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \sqrt{q}\delta_x & \sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x & 0 \end{bmatrix} \mathbf{F}_{x,i}$$

$$\mathbf{K}_t^i = \overline{\mathbf{P}}_t \mathbf{H}_t^{iT} (\mathbf{H}_t^i \overline{\mathbf{P}}_t \mathbf{H}_t^{iT} + \mathbf{Q}_t)^{-1}$$

$$\overline{\mathbf{X}}_t = \overline{\mathbf{X}}_t + \mathbf{K}_t^i (z_t^i - \hat{z}_t^i)$$

$$\overline{\mathbf{P}}_t = (\mathbf{I} - \mathbf{K}_t^i \mathbf{H}_t^i) \overline{\mathbf{P}}_t$$

end for

$$\mathbf{X}_t = \overline{\mathbf{X}}_t$$

$$\mathbf{P}_t = \overline{\mathbf{P}}_t$$

return $\mathbf{X}_t, \mathbf{P}_t$

end loop

range-bearing measurement of this landmark and its corresponding Kalman gain. Finally, the EKF updates $\mathbf{X}_t$ and $\mathbf{P}_t$. [8, p. 314]

2.3. Development Environment

2.3.1. ROS2 Concepts

ROS2, the second release of Robot Operating System, is a set of software libraries and tools that help developers to build robot applications. From drivers to state-of-the-art algorithms and with powerful developer tools, ROS2 makes it easy to create compelling robot behaviours for all types of robots. ROS2 is supported on different operative systems and supports a wide range of hardware platforms, including single-board computers, embedded systems, and full-fledged computers. ROS2 is modular and extensible, making it easy to add new functionality. It has a well-defined API that makes it easy to develop new applications. The reason behind its recent broad adaptation for industrial applications is that the system is built on top of industry standards like IDL, DDS and DDSI-RTPS. Additionally, use cases with small embedded systems with real-time capabilities with Quality of Service features are now possible with ROS2. [5]

IDL

IDL is the Interface Description Language, describing the interfaces for passing information in ROS2. Each interface is defined in an IDL file, which is then used to generate code for the various languages supported by ROS2 (C++, Python). The IDL file contains the message definitions and any service or action definitions. The generated code provides the necessary bindings to allow a message to pass between nodes written in different languages.

DDS

ROS2 communication is built on top of the Data Distribution Service (DDS) standard, a middleware used for communication in distributed systems. ROS2 supports multiple implementations of DDS; in ROS terms, this implementation is called a ROS Middleware interface (RMW). Compared to the previous version of ROS, ROS2 is designed to be more modular and scalable. It also has improved support for real-time applications provided by the utilization of DDS.

DDSI-RTPS

The DDS Interoperability Wire Protocol (DDSI-RTPS) is the wire protocol used to define a standard that allows multiple DDS vendors to interoperate. In a ROS2 system, DDSI-RTPS is used to discover and communicate between different nodes reliably. DDS is implemented on UDP, which gives control of the reliability parameters and is easier to use.

ROS DOMAIN ID

ROS2 introduces the concept of a ROS DOMAIN ID, which is used to identify the nodes running on the same network in the same domain. As previously mentioned, the communication middleware, DDS, enables different logical networks to share a physical one. These different logical networks have their own Domain ID. This ID calculates a UDP port for discovering and communicating with other nodes in the same domain.

Quality of Service

ROS2 supports Quality of Service (QoS) features, which allow the user to specify different policies of service for different types of data. By specifying specific policies, ROS2 can be reliable as TCP, fast and efficient as UDP or somewhere in between. Compared with ROS1, which only supports TCPROS/UDPROS, does ROS2 take advantage of the flexibility of DDS.

RViz

RViz is a visualization tool for ROS. It can be used to visualize the state of a robot, including its sensors and actuators, as well as visualize ROS topics, messages, and services. RViz works by subscribing to ROS topics and messages. RViz provides a powerful visualization tool for understanding the state of a ROS system, which can help in the debugging process.

Nodes

The purpose of a node in a ROS system is to process and compute for a defined purpose. Multiple nodes can work together by exchanging data via messages, services or actions.

Topics

Topics act as a communication bus for exchanging messages between nodes. A node can both publish and subscribe on topics. Additionally, a single node can publish and subscribe on multiple topics simultaneously.

Interfaces

Internal communication in ROS typically happens through the predefined interface definition language (IDL). These interfaces could be in the form of messages on topics, services or actions. As a rule of thumb, topics should be used for continuous data streams, services are suitable for occasional procedure calls, and actions are useful for long runnning tasks.

2.3.2. Gazebo

Gazebo is an open-source 3D robotic simulator that can be used with a Turtlebot 3. It can be used to create and test robotic applications in a virtual environment. Gazebo has a wide range of features, including physics simulation, collision detection, and lighting. It can also simulate sensors, such as cameras and LiDAR's.

2.4. Hardware

2.4.1. Turtlebot 3

The Turtlebot 3 is a ROS standard platform robot mainly developed for educational and research applications. It is a collaboration project between Open Robotics and ROBOTIS, as well as other partners. Some of the advantages of the Turtlebot are that it is affordable, is ROS based, and has well-documented accompanying open-source software. The Turtlebot 3 comes in two variants, the Burger and the Waffle Pi.

Turtlebot 3 Waffle Pi Components

The Turtlebot 3 Waffle Pi features multiple useful sensors and components. Its critical components for the use case in this project are the LiDAR and the Raspberry Pi. Its full specifications are listed in section A.2.

LiDAR or light detection and ranging sensor is commonly used on autonomous robots running SLAM. This sensor works similarly to a RADAR, but instead of using radio waves it sends out light. The distance can be calculated by measuring the time it takes for the light to reflect back to the sensor. The LiDAR sensor used on the Turtlebot 3, is of the type LDS-01. This is a 2D sensor capable of measuring distances 360 degrees around itself at a rate of 5Hz. Its full specifications are listed in section A.1.

Raspberry Pi is a single-board computer (SBC) capable of running a complete Linux operating system with a graphical user interface. Due to its small size and low price, it has become popular among hobbyists and researchers. The Turtlebot 3 Waffle Pi uses the Raspberry Pi 3B, which features a 1.2GHz quad-core processor and 1GB of Random Access Memory (RAM).

2.5. Data processing

2.5.1. DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is an unsupervised clustering method which handles outliers [3]. This method was used on the LiDAR measurements in the landmark detection method, explained later in this report. The algorithm needs only two parameters. The neighbourhood size *eps* and the minimum number of points *min_samples*. Figure 2.3 shows how the DBSCAN algorithm has clustered a point cloud into two clusters. The *eps* value is visualized in the figure as grey circles.

The DBSCAN algorithm iterates through all the points and classifies them as either a core point, a border point or an outlier. The red cluster in Figure 2.3 has four core points since all of these points have at least $min\_samples = 3$ number of points within its neighbourhood $eps = 0.7$. The red cluster has two border points, with at least one core point in its neighbourhood, but less than *min_samples*. Lastly are points with no core points in their neighbourhood defined as outliers or noise points, which in the figure is marked as black points.

One of the advantages of the DBSCAN algorithm is that it is not needed to predefine the number of wanted clusters. This makes it makes more versatile and flexible in its applicable use cases. It also handles outliers by filtering out the less dense areas, which can be useful for noisy data.

2.5.2. Least square fitting of circles

Given a two-dimensional point cloud, it is possible to estimate the parameters of the best-fitting circle using the least square fitting method. This is a method that, for example, can be used to find circular objects from the measured points from a LiDAR sensor.

The point cloud consists of N points with cartesian coordinates x_i, y_i. To estimate the circle, it is necessary to find the parameters:

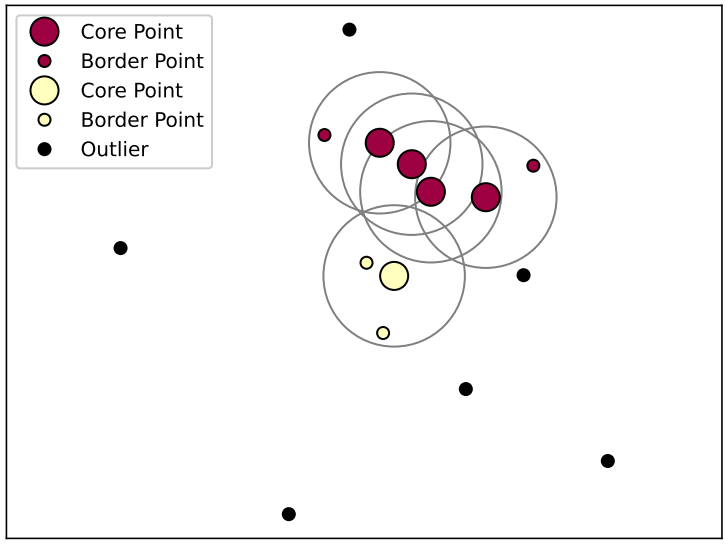

Figure 2.3.: The DBSCAN algorithm. $eps = 0.7$ and $min\_samples = 3$

- x_c - x coordinate of the circle center
- y_c - y coordinate of the circle center
- r - circle radius

Begin with the equation for a circle:

$$(x_i - x_c)^2 + (y_i - y_c)^2 = r^2 \tag{2.9}$$

Rewriting this equation gives:

$$\begin{aligned} x_i^2 + x_c^2 - 2x_i x_c + y_i^2 + y_i^2 - 2y_i y_c &= r^2 \\ 2x_c x_i + 2y_x y_i + r^2 - x_c^2 - y^2 - c &= x_i^2 + y_i^2 \\ a x_i + b y_i + c &= x_i^2 + y_i^2 \end{aligned} \tag{2.10}$$

where

$$\begin{aligned} a &= 2x_c \\ b &= 2y_c \\ c &= r^2 - x_c^2 - y_c^2 \end{aligned} \tag{2.11}$$

To setup the system for all the points in the point cloud, it can be formulated as a matrix equation:

$$\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_n^2 + y_n^2 \end{bmatrix} \tag{2.12}$$

Then the matrices can be defined

$$A = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix}, \quad B = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_n^2 + y_n^2 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{2.13}$$

The equation can then be written as

$$A\mathbf{x} = B \tag{2.14}$$

The optimal solution for $\mathbf{x}$ is then given by moving A to the right hand side by taking its pseudoinverse:

$$\hat{\mathbf{x}} = A^{\dagger} \cdot B \tag{2.15}$$

The last step is then to calculate the circle parameters from a, b and c:

$$\begin{aligned} x_c &= \frac{a}{2} \\ y_c &= \frac{b}{2} \\ r &= \frac{\sqrt{4c + a^2 + b^2}}{2} \end{aligned} \tag{2.16}$$

Having obtained these values, an error parameter can be calculated to determine how well the point cloud fits to the circle.

$$error = \sum_{i=0}^{n} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2} - r_e \tag{2.17}$$

Here is n the number of points in the given point cloud, and (x_i, y_i) is the coordinate of each of these points. (x_c, y_c) is the center coordinate of the estimated circle, and r_e is its estimated radius.

The resulting circle from a given point cloud can then be visualized, as in Figure 2.4.

2.5.3. Mahalanobis distance

Mahalanobis distance is used for unknown correspondence in EKF SLAM to measure the similarity between two observations. This is done by comparing the distance between the two observations and the mean of the observations. The Mahalanobis distance is used to identify which of the two observations is most likely to correspond to the same feature and provide an estimate of the uncertainty in the estimated correspondence. This uncertainty estimate is then used to update the filter's state estimate. Equation 2.18 shows the relationship between the Mahalanobis distance d, position x, measured point μ and the covariance matrix Σ.

$$d^2 = (x - \mu)^T \Sigma^{-1} (x - \mu) \tag{2.18}$$

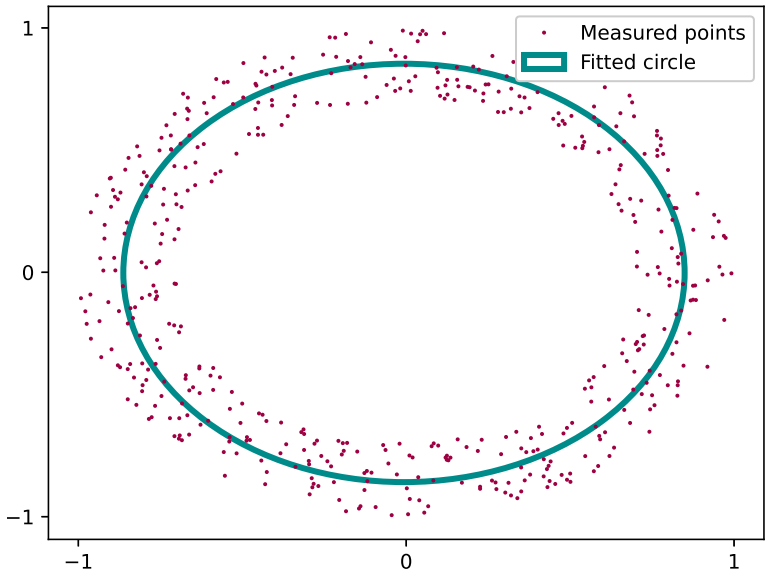

Figure 2.4.: An estimated circle using the least squared fitting method

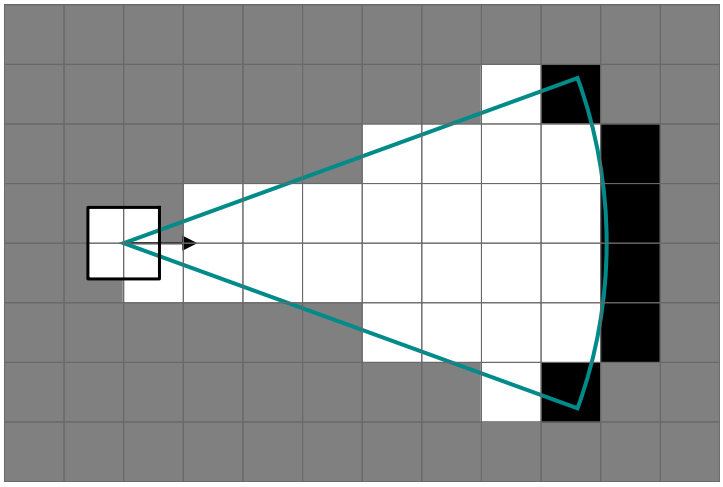

Figure 2.5.: An example of an occupancy grid map generated from a distance sensor.

2.5.4. Occupancy Grid Map

An Occupancy Grid Map is a way of probabilistically representing an environment from noisy sensor data in a two-dimensional map. The map is discretized into a 2D grid where each cell represents the probability of that area being occupied or free space. These cells can vary in size depending on the application, but a cell size of $5x5cm$ is usually sufficient for indoor mobile robots.

An example of an occupancy grid map generated from a LiDAR sensor is shown in Figure 2.5. The square represents the robot, and the cyan-colored outlined sector is the sensor measuring area. The grey cells in the map is set as unknown areas, and will then have a probability value of 0.5, as it is just as likely for it to be occupied as not. The white cells are based on the measurement, free area. Typically this will be some value between 0 and 0.5, based on previous noisy measurements. Lastly is the wall to the right in the figure shown to have a high probability of occupancy, and therefore is set to be a value between 0.5 and 1.

Bresenham line drawing algorithm

When processing LiDAR sensor data to generate an occupancy grid map, it is necessary to discretize the measurements. In practice, this means that it is necessary to draw continuous lines on a discrete map. This is a process often encountered in computer graphics, as it is similar to the process of displaying lines on a computer screen. Bresenham introduced an efficient algorithm for this problem in 1965 [2]. His algorithm uses integer arithmetic to calculate which pixels should be drawn in order to produce a line between two given points. The basic idea behind the algorithm is to choose the integer pixel closest to the line's actual path. This is done by keeping track of the line's error term, which is the difference between the line's actual position and the position of the nearest grid cell. The error term determines which grid cell to turn on or off.

Chapter 3.

Methodology

The initial part of the project consisted of setting up a working EKF SLAM implementation in a simple simulator using Python. The thought behind this is that a more lightweight simulator will make the development and testing more efficient. Also, it makes it possible to isolate code in a way that makes debugging easier. Additionally, it was desired to validate the performance of the EKF SLAM by comparing it against the results of previous academic work.

Since the end goal was to implement the EKF SLAM algorithm in ROS2 and the Turtlebot3, it was also focused on making this simulation run in real-time and isolating the algorithm in a separate node to make it easier to port into the ROS2 framework. The source code for the python simulator and the ROS2 package can be found in the GitHub repository [1].

3.1. Initial Implementation of EKF SLAM in a simple simulator

The goal of this implementation was to get a working demo that could be built upon to extend it to the ROS environment. Therefore it was desirable to split up the code such that the essential part, the EKF SLAM algorithm 1, could be mostly reused when implemented in ROS. Given this case, an object-oriented developing method was chosen.

The simulator was separated into four python classes. A map-, plotting-, robot- and EKF-class. The map and plotting classes are mainly for setup and visualizing and do not affect any of the dynamics of the simulation. The map class is used to place a predefined number of landmarks and keep track of the number of

[1] https://github.com/bendikrs/Project-fall22

landmarks in the map. When a map object is initiated without parameters, the default test map will be created. The plotter class is mainly used to separate all the plotting functions from the actual simulation. It keeps track of the simulated and actual robot trajectories and the figure where all the plots are sent. The plot is updated by sending the relevant states and variables to the *Plotter.plot()* function, which then updates the figure.

The robot class is meant to represent the physical robot and its dynamics. The class contains two main functions, the move and sense function. The move function takes a velocity-based control input $\mathbf{u}_t$ as in Equation 2.2. This input is then propagated using the given time step, which is defined when the class is initiated. The algorithm in 2, shows how this propagation is performed. Moreover, the sense function, described in algorithm 3, takes the true pose of the robot, all landmarks and the covariance matrix of the sensor noise as input. In order to simulate measurement noise, a sample from a normal distribution with zero mean and standard deviation mentioned in [1, p. 20] is added to z_i^r and z_i^θ. A range limit is defined when the class is initiated, and this is used in the if-condition for determining which landmark is in the range of the sensor. As a result, a 2 x N matrix containing all range and bearing measurements is created.

The EKF class contains the EKF-SLAM algorithm. It has methods to predict and update the state of the robot and the landmarks. The methods are all written in a way such that they can be reused in the ROS environment. Algorithm 4 shows how the code is run as a whole, where all the classes work in concert.

Algorithm 2 Move function

$v_t, w_t \leftarrow \mathbf{u}_t$
$x_t, y_t, \theta_t \leftarrow \mathbf{x}_t$
$\mathbf{u}_t = \mathbf{u}_t \cdot t$ ▷ Input is scaled by the timestep
$x_t = x_t + v_t \cdot \cos(\theta_t + w_t)$ ▷ x position is updated
$y_t = y_t + v_t \cdot \sin(\theta_t + w_t)$ ▷ y position is updated
$\theta_t = \theta_t + wrapToPi(w_t)$ ▷ Angular pose is updated
return $\mathbf{x}_t \leftarrow x_t, y_t, \theta_t$

Algorithm 3 Sense function

$x_t, y_t, \theta_t \leftarrow \mathbf{x}_t$
$\mathbf{m} \leftarrow \begin{bmatrix} \mathbf{m}_1 & \mathbf{m}_2 & \dots & \mathbf{m}_n \end{bmatrix}^T$
$\mathbf{Q}_t \leftarrow \mathbf{Q}_t$
for all observed landmarks, **m do**
    $r = \sqrt{(m_i^x - x_t)^2 + (m_i^y - y_t)^2}$
    $\theta = \arctan 2(\frac{m_i^y - y_t}{m_i^x - x_t})$
    if r is within range limit **then**
        $z_i^r = r + rand_r$ ▷ $rand_r \sim \mathcal{N}(0, \mathbf{Q}_t^{\sigma_r})$
        $z_i^\theta = \theta - \theta_t + rand_\theta$ ▷ $rand_\theta \sim \mathcal{N}(0, \mathbf{Q}_t^{\sigma_\theta})$
    end if
end for
return $\mathbf{z}_t \leftarrow x, y, \theta$

Algorithm 4 EKF SLAM Python simulator

$\mathbf{u}_t, \Delta t, \mathbf{R}_t, \mathbf{Q}_t$
landmarks = *map.landmarks*
$\mathbf{X}, \mathbf{P} = \mathbf{X}_o, \mathbf{P}_0$
for all timesteps **do**
    $\mathbf{z}_t = robot.sense(\mathbf{landmarks}, \mathbf{Q}_t)$
    $robot.move(\mathbf{u}_t)$
    $\overline{\mathbf{X}}_t, \overline{\mathbf{P}}_t = ekf.predict(\mathbf{X}_t, \mathbf{u}_t, \mathbf{P}_t, \mathbf{R}_t)$
    $\mathbf{X}_t, \mathbf{P}_t = ekf.update(\overline{\mathbf{X}}_t, \overline{\mathbf{P}}_t, \mathbf{z}_t, \mathbf{Q}_t)$
    $plotter.plot()$
end for

3.2. Implementation of the EKF SLAM in ROS

The objective of developing an EKF SLAM algorithm compatible with ROS2, was to test in simulation environments modelled in Gazebo and compare this to real-world performance. As a result, the assumption of known correspondence between landmarks is no longer valid because the algorithm has no knowledge of the map prior to simulation. The solution for managing the correspondences is further explained in 3.2.3. By leveraging the features of ROS2, the developed package was built to run on an external computer to do all the calculations necessary for the SLAM. Turtlebot 3 acts solely as a node, publishing the relevant messages for the EKF SLAM.

3.2.1. System architecture

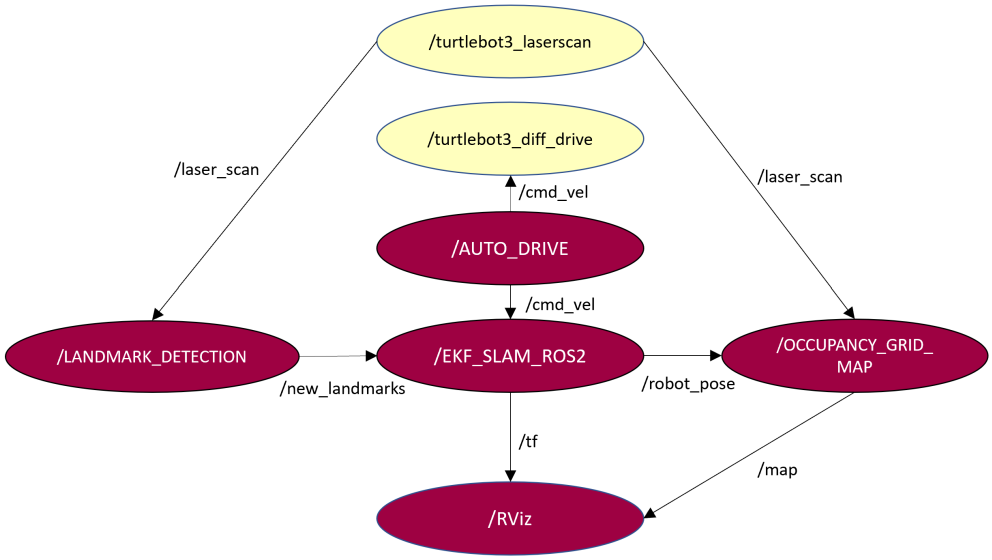

Figure 3.1.: Overview of the participating nodes in the EKF SLAM system

Figure 3.1 portrays the implemented nodes in red, pre-existing nodes in beige and relevant topics between these nodes. The function of all nodes and their dependant topics is explained below.

turtlebot3_laserscan gives the range and bearing data output from the laser readily available on the */laser_scan* topic. A laser scan message is published every 0.2 seconds.

*turtlebot*3_*diff_drive* is the controller which takes the command input, /*cmd_vel*, and converts this into kinematics used for the actuation of the motors.

AUTO_DRIVE is the node that allows the user to initiate circular driving at a constant velocity of the Turtlebot. When this node runs, input is communicated to the ROS2 system on a topic called /*cmd_vel* where the messages are represented as twists.

LANDMARK_DETECTION performs landmark identification based on input from /*laser_scan*.

*EKF_SLAM_ROS*2 run the EKF SLAM algorithm (1) based on information subscribed from /*cmd_vel* and the /*new_landmarks* topics. In order to visualize the results, the combined state vector Equation 2.7 is published on the /*tf* topic. Further, the pose of the TurtleBot is published on the topic /*robot_pose*.

OCCUPANCY_GRID_MAP creates a map of the surrounding environment based on /*laser_scan* and is dependent on /*robot_pose* to transform the map to the world frame. This node publishes the map on the /*map* topic.

RViz visualizes the combined state vector from the EKF and the map from the grid map.

3.2.2. Adapting the system for ROS2

When porting the EKF SLAM to ROS2 there were multiple different factors to take into account. An important difference is that time is predefined and constant in the Python simulator. When the Turtlebot moves around and sends data, the data must be processed before the next time step. The QoS parameters were set to best-effort reliability and reduced queue size, and this resulted in better real-time behaviour.

Further, the Turtlebot's task is only to communicate to an external PC which does all the computing. Separating the workload from the onboard computer to the external PC, the system behaved more reliably. By these measures, factors

such as faulty timing and asynchronous data communication were reduced to a minimum.

3.2.3. Landmark detection

As previously mentioned, the detection and tracking of landmarks are important for SLAM, especially for the EKF SLAM algorithm. This is mainly because it is necessary to know the correspondence for the landmarks between each time step. Therefore a robust detection algorithm is needed to get reliable detections of landmarks. In practice, this is very difficult and seldom viable in real-world applications, so artificial beacons were used as landmarks in this project.

As shown in **??**, our testing environment consists of a grid of cylinders. This setup is described further in the experiments section of the report. Considering the sparsity of the LiDAR measurements, it is not easy to extract useful information about complex shapes from them. Cylindrical shapes are therefore a viable option when designing a beacon for use as landmarks in the EKF SLAM algorithm.

The LDS-01 2D LiDAR on the Turtlebot measures at a sampling rate of 1.8kHz. This corresponds to five full 360° point clouds every second, with an angular resolution of 1°. The LiDAR sensor's full specification can be found in section A.1. These measured point clouds are published as a ROS2 LaserScan message type, containing the range and bearing data. Using these measurements, a landmark detection algorithm can be developed, as shown in algorithm 5.

The means of this algorithm is to find the centre points of any circular landmarks with a specific radius if they are in the LiDAR measuring area. First, the landmark array is initialized. This array is to contain the centre points of any found circular landmark. Then the newest laser-scan message is grabbed on the ROS2 */laserscan* topic. This point cloud is clustered using DBSCAN on line number 3. This is to group the points corresponding to each of the circular landmarks. A more detailed explanation of the DBSCAN algorithm can be found in subsection 2.5.1.

Choosing appropriate parameters for the DBSCAN clustering is essential. Trial and error showed that a neighbourhood size of $eps = 0.1m$ was suitable. Also, a reasonable size of the clusters is needed. This is to avoid misinterpreting small clusters as landmarks if the sensor data is noisy. Therefore a minimum samples value of $min\_samples = 14$ was chosen. The main reasons for using DBSCAN

for this, is its handling of outliers, as well as it has a non-predefined number of clusters. As the LiDAR measurements often are quite noisy, it is important to ignore the bad samples and still retain the good data. Since the DBSCAN algorithm is density based, it can effectively cluster nearby measured points without being affected by noise points.

Having extracted the clusters, a circle fitting algorithm can be executed, as seen on line number 5. The least squares fitting approach is then fitting to search for a circular object of a known radius. This algorithm explained in subsection 2.5.2, outputs four parameters. The x and y coordinates of the centre point of the fitted circle. Also, the radius r of the circle, as well as an error parameter. The error parameter is the sum of the deviations from the estimated circle, for each point, as shown in Equation 2.17. As seen at line number 6, this error is used to filter out the point clouds which are not adequately shaped like a circle. The value of $error = 0.01m$ was found empirically by gradually decreasing it until it became reasonably sensitive. All the relevant parameters used in the testing is presented in Table 4.3.

Additionally, is the estimated radius checked to see if it is close enough to the expected landmark radius. If this is the case, then the centre point of the estimated circle is added to the landmark array **m**. This process is repeated for all the observed clusters, and the landmark array is returned.

Algorithm 5 The landmark detection algorithm

1: **m** = [] ▷ Initialize landmark array
2: $point\_cloud = /laserscan$ ▷ Get latest ROS2 message on /laserscan topic
3: $clusters = DBSCAN(point\_cloud)$
4: **for** cluster in clusters **do**
5: $x, y, r, error$ = LSF(cluster) ▷ Using least squares fitting
6: **if** $error < 0.005m$ and r is within expected values **then**
7: **m** = x, y
8: **end if**
9: **end for**
return m

3.2.4. Landmark data association

As the robot observes different landmarks during operation, it is necessary to keep track of the identity of each landmark for the EKF to work. In this implementation, the EKF_SLAM_ROS2 node addresses this problem by comparing the distance between the newly observed landmarks with the previously observed

ones. The Mahalanobis distance is then used to compute the distance between the existing landmarks, and the new ones. Using the Mahalanobis distance is preferable to just euclidean distance, as it utilizes the known measurement noise and estimated uncertainty to distinguish between observations and minimize false data associations [6, p. 876].

3.2.5. Installation

All software written in this project can be found in the Github repository [2]. A.3 describes the runbook which is also found the the repository.

[2] https://github.com/bendikrs/Project-fall22

Chapter 4.

Results and discussion

Validation of the algorithm was done using our python demo simulator. Then two tests were performed using the ROS2 implementation, one in the Gazebo simulator, and one in real world using the Turtlebot3. Screen capture videos of these experiments are presented for Gazebo here, and for the live demo here. A video of the live demo is also given here.

4.1. Python demo simulator for validating

In order to validate the result, the simulation settings were set up in a similar fashion to the experiment carried out by Barrau and Bonnabel in [1, p. 20]. The robot is then steered to drive at a velocity of $v_t = 1m/s$ and an angular velocity of $w_t = 9°$. This will make the robot run in a circle with a circumference of $40m$ and use 40 seconds per revolution. The maps consist of 20 landmarks placed in a circle outside the path of the robot. The sense function is modelled with the assumption of isotropic noise and a standard deviation of 10 cm. The initial uncertainty of the robots' pose and position of all the landmarks are zero. In algorithm 4, all initialization settings are set before the simulation loop.

Figure 4.1 graphically displays the numerical results given by the implemented Python simulator. The Root Mean Square Error (RMSE) values are calculated by comparing the estimated pose from our EKF SLAM and the ground truth from the ideal motion model. We observe stability and consistent RMSE values. By comparing this result to the result by Barrau and Bonnabel in [1, p. 21], we can see similar behaviour. Therefore we can subsequently validate our result.

Parameter	Value
$\mathbf{X}_0$	$[0, 0, ..., 0]^T_{(3+2N)}$
$\mathbf{P}_0$	$diag([0.1, 0.1, 0.1, 1e6, ..., 1e6)]_{(3+2n,3+2N)}$
$\mathbf{R}_t$	$diag([0.1, 0.1, 0.1])$
$\mathbf{Q}_t$	$diag([0.01, 0.01])$
$\mathbf{u}$	$[1\frac{m}{s}, \frac{\pi}{20}s^{-1}]^T$
t	$1s$
Sense range limit	$5m$

Table 4.1.: The initial values used for the python demo simulator

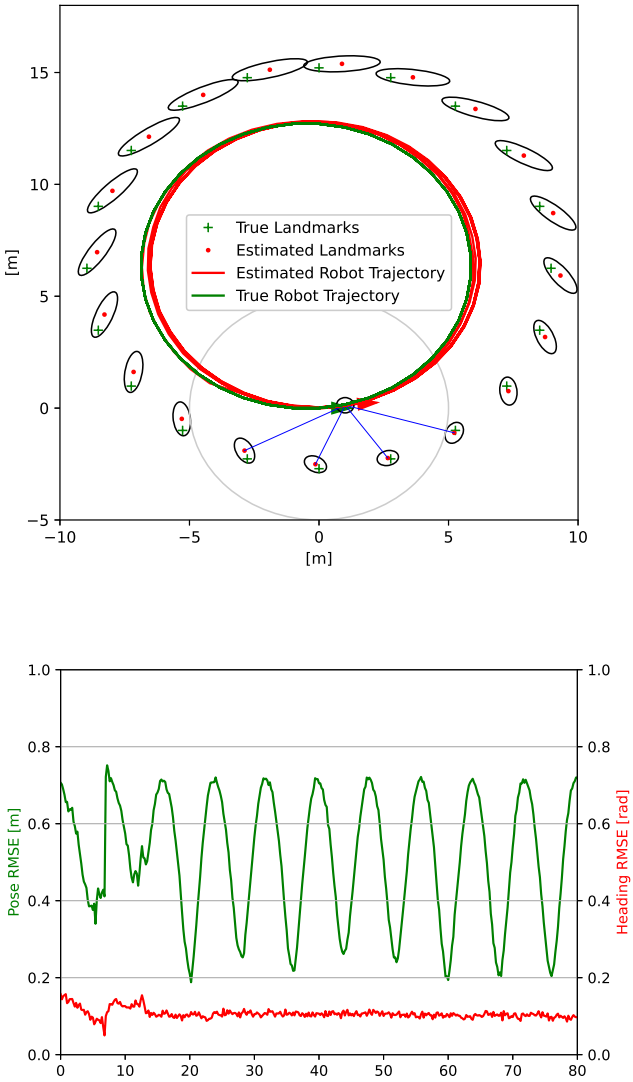

Figure 4.1.: Our EKF SLAM implementation running our python demo simulator. The top plot shows the robot trajectory after ten circular runs. The confidence ellipses shows the possible robot positions within $\pm 1\sigma$. The bottom plot displays both pose and heading root mean square error (RMSE) from the same test run.

4.2. Gazebo simulator and real-world testing

Further testing using the full ROS system can be performed when the algorithm has been validated using the Python simulator. In order to set up the test, an appropriate testing environment needed to be designed.

4.2.1. Design of testing environment

A controlled testing environment had to be made to get a quantifiable and comparable test case. It is desirable to make the physical test environment as similar to the simulator as possible. Therefore a simple and easily reconstructible map was desirable to use. A 2x2.4m rectangle with four landmarks placed around the centre was constructed at the lab. Figure 4.3 shows the layout of the environment.

The final setup of the real-world testing can be seen in the right image from Figure 4.5. The landmarks consist of 3D-printed cones with a radius of $0.08m$ at their widest point. The cones radius is designed to be constant around the height of the LiDAR sensors measuring plane. This is approximately $0.14m$, which is the height of the Turtlebot3, as specified in section A.2.

Figure 4.2.: 3D printed landmark with a radius of $0.08m$.

When testing the use of these landmarks, it was quickly observed that the 3D-printed plastic was too reflective and irregular. This resulted in inaccurate measurements from the LiDAR sensor, which made the captured point clouds unusable for our SLAM system. The solution to this problem was to cover the cones with masking tape. This gave the cones a much more diffuse surface area, which was easier for the LiDAR to measure. A picture of the cones used as landmarks is

shown in Figure 4.2

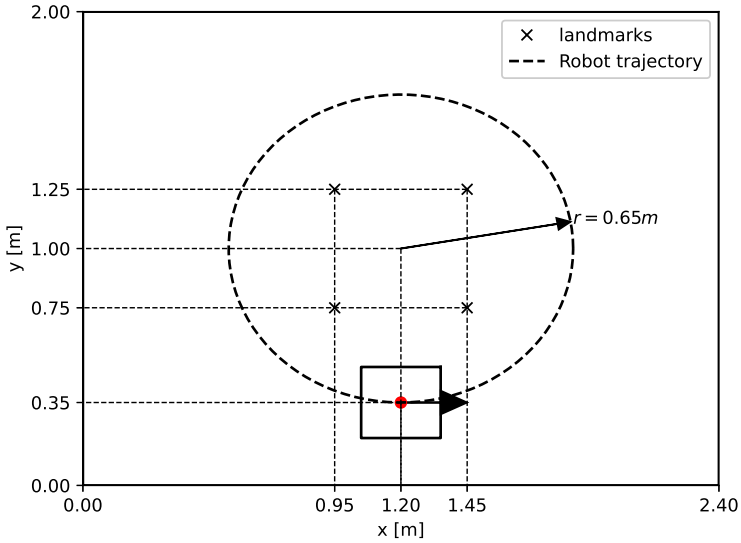

Figure 4.3.: The layout of the map used for both the Gazebo testing and the real-world testing. The robot's starting pose is illustrated with the red dot and arrow.

4.2.2. Initialization of the test case in Gazebo and real-world

Parameter	Value
$\mathbf{X}_0$	$[0,0,0]^T_{(3,1)}$
$\mathbf{P}_0$	$diag([1,1,1]_{(3,3)})$
$\mathbf{R}_t$	$diag([0.1,0.1,0.01])^2$
$\mathbf{Q}_t$	$diag([0.1,0.1])^2$
$\mathbf{u}$	$[0.1\frac{m}{s}, \frac{0.1}{0.7}s^{-1}]^T$
t	$0.2s$
Sense range limit	$3.5m$

Table 4.2.: The initial parameters used in the EKF SLAM algorithm for both the real world and Gazebo testing.

The test case consists of the Turtlebot3, with the implemented ROS2 system, driving around the map described in Figure 4.3 and shown in Figure 4.4 and Figure 4.5. In order to drive in a circular path with $r = 65cm$, the velocity is set to a constant $v_t = 0.1m/s$ and angular velocity of $w_t = \frac{v_t}{r}$. The EKF_SLAM_ROS2 node is initialized with the parameters described in Table 4.2. In this table, the time step is set as a fixed value of $t = 0.2s$. This is the set refresh rate for the laser scan, but the actual time step used in the algorithm is measured between each calculation. This means that the actual time step can vary somewhat between each iteration. The exact same parameters are used in the Gazebo testing as in the real-world testing.

Parameter	Value
landmark_threshold	± 0.2
radius_threshold	$\pm 0.025m$
landmark_circle_error	0.01
landmark_radius	$0.08m$
eps	$0.1m$
min_samples	14

Table 4.3.: The initial parameters used in the landmark detection and association algorithm. Used for both the real world and Gazebo testing.

The parameters related to the landmark detection and association algorithms are presented in Table 4.3. The *landmark_threshold* is the threshold used for determining whether a newly observed landmark should be classified as a new or existing landmark. This threshold is given as the Mahalanobis distance. *radius_threshold* is the threshold used when filtering out the detected circles, based on their radius. *landmark_circle_error* is the highest allowable error from the least square method used for the circle fitting. The *landmark_radius* is the radius for the landmark which the algorithm tries to find. *eps* and *min_samples* refers to the two parameters used for the DBSCAN clustering.

The Gazebo simulation map is replicated to the exact dimensions shown in Figure 4.3. The real-world equivalent map is replicated as accurately as practically possible. This of course counts as a fault node in the comparison of the two tests, but principally the varying landmark locations and map dimensions should not affect the SLAM performance significantly.

4.2.3. SLAM performance comparison

Figure 4.4 and Figure 4.5 displays how the EKF SLAM manages to keep track of its own pose relative to the constructed map in the two different tests. In both cases, the robot has driven four loops around the landmarks and the images are taken after the same amount of run time.

The test run in Gazebo shows a satisfactory performance in keeping track of the robot's pose relative to its surrounding environment. As observed, the estimated pose and the ground truth seem to be aligned, which validates the EKF SLAM algorithm in a simulated Gazebo environment. It shows that the landmark detection algorithm includes some faulty detections. These could maybe be avoided by calibrating the parameters in Table 4.3.

The live test run shows to keep track of its pose relative to its environment. The estimated pose and ground truth seem to be aligned, which indicates that the EKF SLAM algorithm is working properly in a real-world environment. However, more false detections of landmarks are observed in the live test run than in the simulated test run. This could also be improved by the calibration of parameters, but there is an unknown difference between the modeled environment and the real-world environment that could be causing this. Additionally, the occupancy grid map seems to rotate as the robot moves, which could be caused by the asynchronous updates of */robot_pose* topic from the */EKF_SLAM_ROS2* node to */OCCUPANCY_GRID_MAP* node.

As earlier mentioned, was the reflectivity of the landmarks important to get good readings from the LiDAR. The surface of the modelled landmarks in Gazebo is set as completely diffuse, which practically removes this problem from the simulator. Since the LiDAR uses light to get its measurements, it is also possible that the varying light conditions of the tests could have affected the performance. Intense light, from for example a window, can alter the readings. This was observed in the early testing for this project.

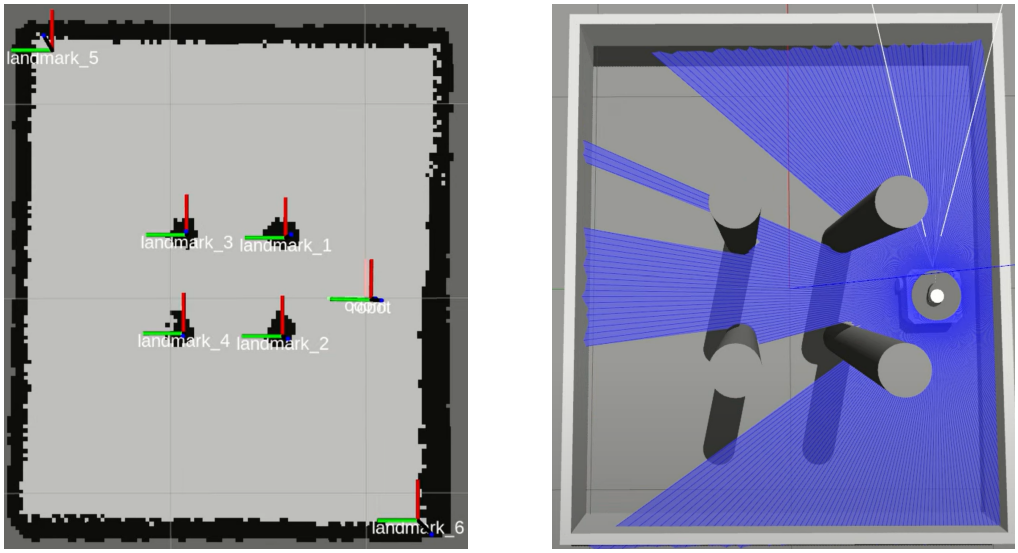

Figure 4.4.: Snap shot of the Gazebo test run. The left image displays the constructed map and the right displays the Gazebo world, at the same time.

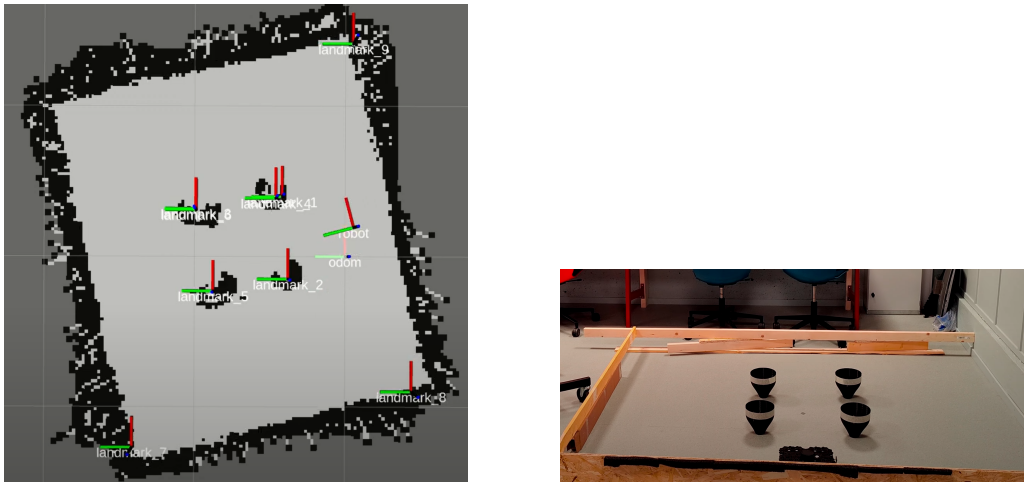

Figure 4.5.: Snap shot of the live test run. The left image displays the constructed map and the right displays the real-world environment, at the same time.

4.2.4. Odometry comparison

Figure 4.6 shows the RMSE values from both the Gazebo simulator test and the real-world test, performed as described in section subsection 4.2.2. These values are calculated by comparing the estimated pose from our EKF SLAM and the ideal circular trajectory from the motion model Equation 2.4. Therefore will these RMSE values measure the combined error from both the open-loop control of the robot and the EKF SLAM pose estimation error. The EKF SLAM performance is thus not directly reflected in these plots. However the plots visualize the difference from the Gazebo simulator motion model, and the actual real-world motion of the robot.

The fact that both the RMSE values increase over time, is as expected. As our reference pose is only based on propagating the input u_t, some deviation is expected. We send the constant input u_t to the robot and assume it will follow our theoretical calculated circular trajectory. As this deviation increases over time, in both the simulation and the real-world test, the RMSE measurement will also increase. Comparing these results to our initial testing using our Python simulator, we can observe a significant difference. As the python simulator is based on a ground truth as reference, the RMSE is observed not to increase over time.

As Figure 4.6 shows, a considerable higher error is present from the real-world demo. The added unknown variables introduced introduces in the real-world test is likely to cause this difference. Factors could be, but not limited to:

- Different wheel diameter on the robot compared to the motion model
- Difference in controlled motor rotation and actual rotation
- Uneven flooring
- Difference in the wheel slip from the simulator and real world

Although the error is higher in real-world testing, it is still shown to vary consistently for both cases in the position RMSE plot. The lower points of the sinusoidal waves are observed to be present when the robot passes its starting point. This is as expected, and shows that the EKF SLAM performance is successfully improved by this loop closure. Based on this one can assume the sinusoidal part of the graphs is caused by the varying performance of the EKF SLAM.

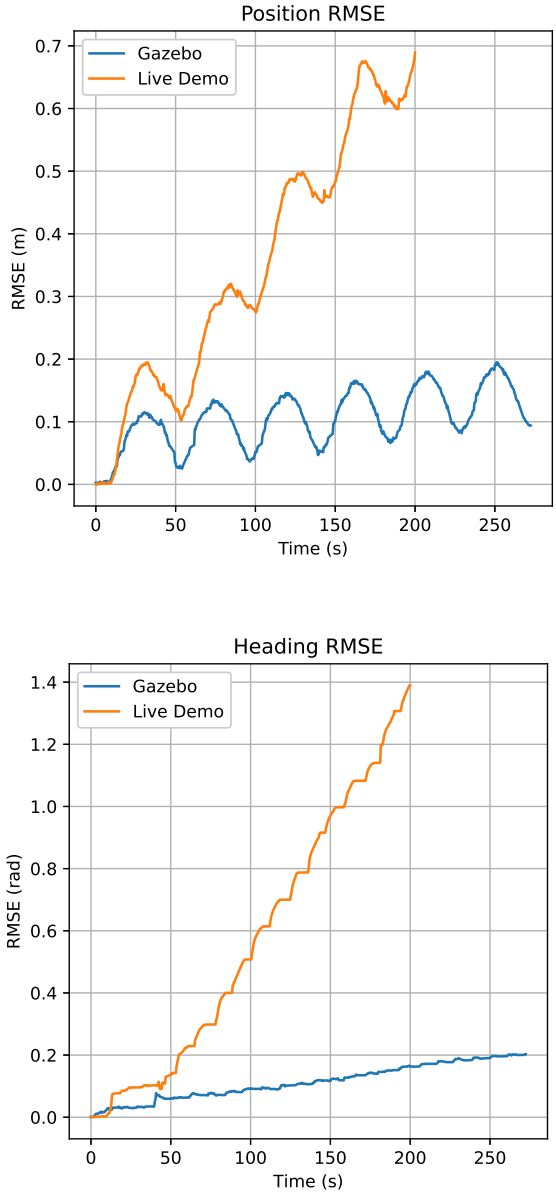

Figure 4.6.: Comparison of the differences between the ideal motion model and the EKF SLAM pose estimation in both Gazebo and in real-world.

Chapter 5.

Future Work

The system implemented in this project can function as a testing framework for experimenting with different SLAM algorithms based on Kalman Filter. Since the ROS2 package is structured modular, only minor changes are needed to use different formulations of the SLAM algorithm. Looking into Invariant EKF could be an interesting variant to compare with. Also, studying the differences in reliability when utilizing the inertial measurement unit available on the Turtlebot.

To test the SLAM performance using RMSE values, a reliable ground truth is needed. This could be solved by using an accurate external tracking method. Optitrack is a company delivering solutions for this purpose. They use cameras and custom tracking markers to determine a robot's pose. This is done with such a high accuracy that it is commonly used for a ground truth reference. The use and implementation of such a system are outside this project's scope.

Chapter 6.

Conclusion

To first run the EKF SLAM algorithm using our simple python simulator was an effective method of validating its correctness. This simplified the further implementation of the system in ROS2, as initially planned. Although the algorithm was validated, it was experienced that landmark detection and data association was shown to be the most critical part of the reliability of the system.

Using the ROS2 middleware, the Turtlebot3 was able to send and receive messages from the remote computer. This enabled the robot to be controlled remotely and for data to be streamed from the robot to the computer. The report also demonstrated an effective and reliable way of remotely controlling robots and streaming data between remote machines and robots.

The results show that the real-world performance of the EKF SLAM works comparable to the simulator testing. It is generally less stable than the controlled simulated environment, but this is as expected as real-world testing introduces multiple unknown variables. A critical factor showed to be the bad LiDAR measurements that came from measuring on irregular surfaces. This added extra inaccuracies to the landmark recognition and overall some worse performance.

References

[1] Axel Barrau and Silvere Bonnabel. "An EKF-SLAM algorithm with consistency properties". In: (2015). DOI: 10.48550/ARXIV.1510.06263.

[2] J. E. Bresenham. "Algorithm for computer control of a digital plotter". In: *IBM Systems Journal* 4.1 (1965), pp. 25–30. ISSN: 0018-8670. DOI: 10.1147/sj.41.0025.

[3] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, Aug. 1996, pp. 226–231.

[4] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". eng. In: *Journal of Basic Engineering* 82.1 (1960), pp. 35–45. ISSN: 0098-2202. DOI: 10.1115/1.3662552.

[5] Steve Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. "Robot Operating System 2: Design, Architecture, and Uses In The Wild". In: *Science Robotics* 7.66 (May 2022). arXiv:2211.07752 [cs], eabm6074. ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074.

[6] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. eng. Berlin: Springer, 2008. ISBN: 9783540303015.

[7] Gerald L. Smith, Stanley F. Schmidt, and Leonard A. McGee. *Application of Statistical Filter Theory to the Optimal Estimation of Position and Velocity on Board a Circumlunar Vehicle*. Tech. rep. NASA-TR-R-135. Jan. 1962. URL: https://ntrs.nasa.gov/citations/20190002215 (visited on 11/27/2022).

[8] Sebastian Thrun. *Probabilistic Robotics*. eng. Intelligent robotics and autonomous agents. Cambridge, Mass: MIT Press, 2005. ISBN: 9780262201629.

Appendix A.

Specifications and setup

A.1. Lidar LDS-01 Specifications

Items	**Specifications**
Operating supply voltage	5V DC ±5%
Light source	Semiconductor Laser Diode(λ=785nm)
LASER safety	IEC60825-1 Class 1
Current consumption	400mA or less (Rush current 1A)
Detection distance	120mm ∼3,500mm
Ambient Light Resistance	10,000 lux or less
Sampling Rate	1.8kHz
Dimensions	69.5(W) X 95.5(D) X 39.5(H)mm
Mass	Under 125g
Distance Range	120 ∼3,500mm
Distance Accuracy (120mm ∼499mm)	±15mm
Distance Accuracy(500mm ∼3,500mm)	±5.0%
Distance Precision(120mm ∼499mm)	±10mm
Distance Precision(500mm ∼3,500mm)	±3.5%
Scan Rate	300±10 rpm
Angular Range	360°
Angular Resolution	1°

A.2. Turtlebot3 Specifications

Items	**Waffle Pi**
Maximum translational velocity	0.26 m/s
Maximum rotational velocity	1.82 rad/s (104.27 deg/s)
Maximum payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1.8kg
Threshold of climbing	10 mm or lower
Expected operating time	2h
Expected charging time	2h 30m
SBC (Single Board Computers)	Raspberry Pi
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	RC-100B + BT-410 Set (Bluetooth 4, BLE)
Actuator	XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01 or LDS-02
Camera	Raspberry Pi Camera Module v2.1
IMU	Gyroscope 3 Axis Accelerometer 3 Axis
Power connectors	3.3V / 800mA 5V / 4A 12V / 1A
Expansion pins	GPIO 18 pins Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
DYNAMIXEL ports	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences
Programmable LEDs	User LED x 4
Status LEDs	Board status LED x 1 Arduino LED x 1 Power LED x 1
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
PC connection	USB
Firmware upgrade	via USB / via JTAG
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

A.3. Setup and installation

Setup ROS2 environment on remote PC

Install Ubuntu 20.04.5 LTS (Focal Fossa) desktop image on the computer you want to run the EKF SLAM on. Ubuntu 20.04

```
# Install ROS2 Foxy on remote PC
wget https://raw.githubusercontent.com/ROBOTIS-GIT/robotis_tools/
    master/install_ros2_foxy.sh
sudo chmod 755 ./install_ros2_foxy.sh
bash ./install_ros2_foxy.sh

# Install Gazebo packages
sudo apt-get install ros-foxy-gazebo-*

# Install TurtleBot3 packages
source ~/.bashrc
sudo apt install ros-foxy-dynamixel-sdk
mkdir -p ~/turtlebot3_ws/src
cd ~/turtlebot3_ws/src/
git clone -b foxy-devel https://github.com/ROBOTIS-GIT/DynamixelSDK.
    git
git clone -b foxy-devel https://github.com/ROBOTIS-GIT/
    turtlebot3_msgs.git
git clone -b foxy-devel https://github.com/ROBOTIS-GIT/turtlebot3.git
git clone -b foxy-devel https://github.com/ROBOTIS-GIT/
    turtlebot3_simulations.git
# Move the ekf_slam_ros2 package from repo into the ~/turtlebot3_ws/
    src/ folder
cd ~/turtlebot3_ws && colcon build --symlink-install
echo 'source ~/turtlebot3_ws/install/setup.bash' >> ~/.bashrc
source ~/.bashrc
```

Setup SBC (Raspberry Pi) and OpenCR

Follow the Robotis e-Manual to setup the physical TurtleBot3.

- SBC Setup
- OpenCR Setup