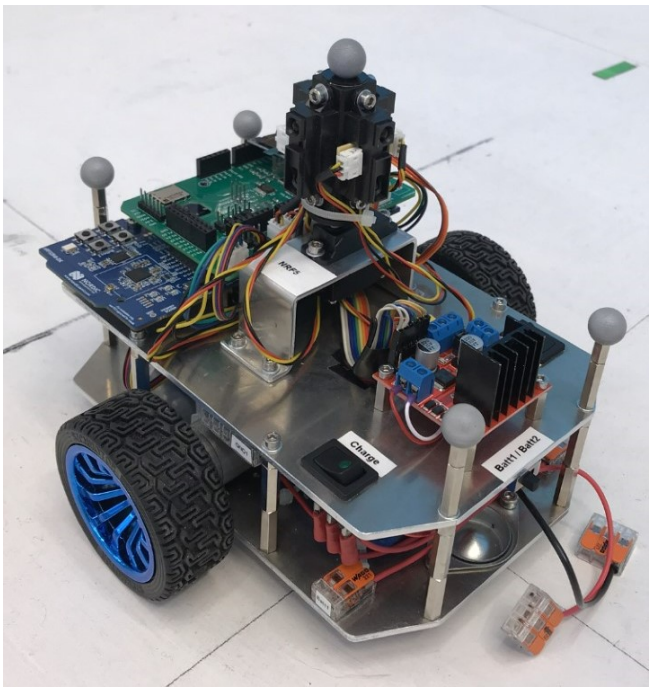
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Thomas Andersen

Sparse IR sensor EKF-SLAM for MQTT-SN/Thread connected robot

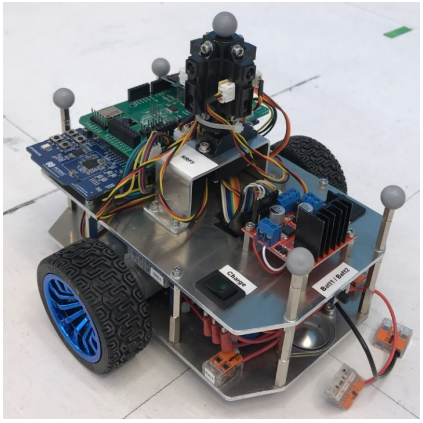
Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
June 2022



NTNU
Kunnskap for en bedre verden

Thomas Andersen

Sparse IR sensor EKF-SLAM for MQTT-SN/Thread connected robot



Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Table of Contents

List of Figures

List of Tables

1 Preface

Concluding 5 years of studies, students who study Cybernetics and Robotics at the Norwegian University of Science and Technology are to deliver a thesis for the 30-credit course *TTK4900 Engineering Cybernetics, Master's Thesis*. This thesis, *Sparse IR sensor EKF-SLAM for MQTT-SN/Thread connected robot*, presents the work done from January to June 2022 and builds upon the work from Andersen (2022) [5].

For working on the project I was given a workstation, a nRF52840-based robot, a Raspberry Pi 3b+, a nRF52840 dongle, and access to the motion tracking system *OptiTrack* in room B333 in Elektrobygget at campus Gløshaugen.

I would like to thank all the personnel at the ITK workshop by Forsøkshallen for helping out with hardware-related issues of the robot. Moreover, I would like to extend my appreciation to my supervisor, Tor Onshus, for always being available for guidance and discussions during work on the project. Last but not least I would like thank all my student colleagues, family, and especially Johanna for all the support.

2 Summary and conclusion

The *robot-project*, also known as the *SLAM-project*, involves student contributions to a system comprising of several robots wirelessly connected to a local server application. The system intends to use the robots to map out an indoor environment. It is a goal that the hardware components involved should be of low-cost in order to motivate smart software solutions and increased accessibility of affordable robotics systems.

This thesis' contributions to the robot-project may be seen as twofold - one part involving embedded software development of the communication system used between robots and the server, and the second part comprising development of a Simultanous Localization and Mapping (SLAM) algorithm for the robots. The main focus has been directed at implementing software for a differential drive robot built by previous student contributions based around the nRF52840 Development Kit ported with FreeRTOS. The software implementations are written in C for the nRF52840 based robot, C++ for minor contributions to the server application and python for testing, logging and plotting results.

In regards to the development of the communication system, a multi-threaded MQTT-SN client was developed for the robot. Furthermore, a Raspberry Pi was configured to run as a MQTT-SN gateway and broker. The developed communication system was extensively used for both debugging, testing and recording results from the state estimation and mapping part of the assignment.

The existing server-robot system before work on this thesis was carried out could be characterized by a master-slave relationship. The robots were responsible for collecting measurements of their surrounding environment, and these measurements along with the robot's pose were sent to the server. The server would command the robots to move to a given target position, and use the information provided by the robot to concurrently build a map and place the robot inside the map. The server application has a simple GUI for observing the robot's position and the incrementally generated map, as well controls for human operators for commanding the robots. In an attempt to remove the master-slave relationship, the robots should be able to operate autonomously (without server interaction) and generate a map by themselves. Thus, rendering the server application only a tool for visualization, data collection and analysis. I.e. the aim is to demote the server application from being a core part of the robot's navigation system.

As a first step towards making the robots of the SLAM-project fully autonomous, a feature-based Extended Kalman Filter (EKF)-SLAM algorithm intended for the nRF52840 robot was developed. The robot utilizes a set of four low-end infrared sensors mounted on top of a rotating tower for exteroceptive sensing. Line segments are used as features, and are used in the state estimation framework provided by the Kalman filter for fusion with odometry and inertial measurements in order to simultaneously provide estimates of the robot's two-dimensional position and heading, and build a map. The line extraction algorithm is based upon the density based clustering method known as DBSCAN, the Iterative Endpoint Filter (IEPF), and minimum-square-error line fitting. The feature extraction algorithm was implemented for the nRF52840 robot, however, due to time constraints of the project the EKF-SLAM algorithm implementation for the robot was not finished. Testing of the SLAM approach was still conducted as the algorithm was developed initially in python. Testing the algorithm was done by creating a dataset utilizing the developed MQTT-SN client to publish data from the robot running the line extraction algorithm whilst navigating through a maze. The results show that the EKF-SLAM algorithm outperforms the pose estimates of the EKF currently running on the nRF52840 robot in the presence of loop-closures. However, the generated map appears slightly skewed and displaced likely due to the simplistic closed-form line uncertainty method used in the line extraction process.

3 Sammendrag og konklusjon

Robot-prosjektet også kjent som *SLAM-prosjektet* er et resultat av flere studentbidrag til et system bestpende av flere roboter med trådløstilkobling til en lokal server applikasjon. Systemets formål er å kartlegge et innendørs område. Det er et mål at maskinutstyr som inngår i prosjektet skal være billig for å motivere smarte programvareløsninger og kunne øke tilgang på rimelige robotsystemer.

Denne avhandlingens bidrag til robot-prosjektet er kan ansees som todelt - en del som omhandler utvikling av programvare i tilknytning til kommunikasjonssystemet brukt mellom robotene og serveren, og en annen del som handler om utvikling av en algoritme for lokalisering og kartlegging for robotene. Hovedfokuset har vært rettet mot utvikling av programvare for en tohjulet robot laget av tidligere studenter som er basert på utviklingskortet nRF52840 portert med FreeRTOS. Programvareløsningene er utviklet i C for nRF52840-roboten, C++ for små bidrag til server applikasjonen og python for testing, logging og plotting av resultater.

I forbindelse med utvikling av kommunikasjonssystemet ble en fler-trådet MQTT-SN klient laget for roboten. Ytterligere ble en Raspberry Pi konfigurert som en MQTT-SN *gateway* og MQTT server. Kommunikasjonssystemet ble tatt i bruk for feilsøking, testing og innhenting av resultater fra tilstandsestimering- og kartleggingsdelen av oppgaven.

Det eksisterende server-robot systemet før arbeid på oppgaven ble utført kunne vært beskrevet som et master-slave forhold. Robotene var ansvarlige for å hente målinger fra omverdenen som ble sendt sammen med robotens posisjon og retning til serveren. Serverens oppgave var å kommandere robotene til et målområde, og bruke informasjonen sendt fra robotene til å bygge et kart og plassere robotene i kartet. Server applikasjonene har et enkelt brukergrensesnitt for å observere robotenes posisjon og kartet, i tillegg til knapper for å manuelt styre robotene. I et forsøk på å fjerne master-slave forholdet bør robotene operere autonomt (uten å måtte forholde seg til serveren) og kunne generere et kart selv. Det er et mål at server applikasjonen kun skal bli brukt som et verktøy for visualisering, datainnhenting og analyse, og ikke fungere som en sentral del av navigasjonssystemet for roboten.

Som et første steg mot å gjøre robotene autonome ble en algoritme basert på et utvidet Kalman Filter utviklet for nRF52840-roboten. Algoritmen detekterer linjestykker som brukes i tilstandsestimeringen for å både kartlegge og korrektere posisjon og retningen til roboten (EKF-SLAM). Roboten bruker fire rimelige infrarødsensorer plassert på toppen av et roterende tårn for å hente data fra omgivelsene sine. Linjedetekterings-algoritmen er basert på grupperingsmetodene DB-SCAN og IEPF, og minste kvadraters metode for lineær regresjon. Linjedetekterings-algoritmen ble implementert for roboten, men EKF-SLAM implementasjonen ble ikke ferdigstilt i tide. Likevel ble implementasjonen testet ettersom den først var implementert i python. For å teste algoritmen ble et dataset laget ved å bruke den utviklede MQTT-SN klienten til å sende data fra roboten samtidig som roboten kjørte rundt en labyrint og detekterte linjestykker. Resultatene tyder på at EKF-SLAM algoritmen er i stand til å estimere posisjon og retning på roboten bedre enn EKF algoritmen som kjører på roboten når roboten gjenkjenner områder den har besøkt tidligere. Kartet som genereres samtidig som roboten estimerer sin egen posisjon og retning stemmer ikke helt overens med de faktiske omgivelsene roboten prøver å kartlegge. Dette skyldes antakeligvis den enkle formen for estimering av linjestykkenes usikkerhet i linjedetektsjons-algoritmen.

4 Introduction

The relevance of autonomous mobile robots for society continuous to grow as advancements in technology make the use-cases for robots extend into a multitude of domains [3]. Mobile robots may be used for tasks such as rescue and recovery missions in hazardous environments, industrial manufacturing, and for consumer-oriented applications such as lawn mowing and home cleaning. Moreover, the rise of Industry 4.0 enables robotics systems to operate more efficiently, provides real-time surveillance and remote control, as well as expanding the list of possible use-cases for autonomous mobile robots[17].

The SLAM-project was first started in 2004 by the Department of Engineering Cybernetics. Previously the project was referred to as the *LEGO-project*, since the robots were based on the LEGO Mindstorms robotics kits. Since then new robots have been built based around the family of Arduino boards, and most lately the nRF52840 development kit.

The goal of the SLAM-project is to make multiple low-end robots cooperate with a local server in order to map out an unknown indoor area. The server application remotely controls the robots to explore their surrounding environment, and receives updates of the robots' location and information regarding the robots' perceived local surroundings.

The main focus of this thesis will be surrounding software development for the nRF52840-based robot. More specifically, building upon the work done in [5] to make the robot less dependent on server interactions by instructing the robot to do the computations involved in generating a map whilst navigating through an unknown area itself.

5 Problem statement

Implementation of a SLAM solution on a nRF52840 DK based robot ported with FreeRTOS connected to a local server for visualization is the main goal of the work on this thesis. More specifically this entails:

- Exploiting the correlations between the range-bearing measurements of the rotating IR-sensor tower with odometry and inertial measurements to minimize pose drift.
- Incrementally building a two-dimensional representation of the environment the robot explores suitable for autonomous operation of the robot without the *need* for communication with the server.
- The SLAM implementation should be extendable to multi-robot SLAM.
- Setting up a robust communication link to the server application.
- Being able to send target positions from the server application to the robot.
- Use the server application to visualize the map made by the robot and its trajectory.

6 Previous work

Every year since the SLAM-project was initiated students have contributed to the project in various ways such as hardware-focused building of the robots, low-level embedded software development of drivers for interfacing the hardware components, high-level embedded software for control, navigation and communication, and various implementations of server-side logic.

Jølsgård (2020) [22] contributed to the SLAM-project by building the nRF52840-based robot resembling the new one used for software development in this thesis. Additionally, Jølsgård adapted the software of the older nRF52832-robot from the work done by Korsnes (2018) [23], Leithe (2019) [25] and Stenset (2020) [47]. These collections of contributions to the SLAM-project form the main part of the software running on the nRF52840-robot encompassing a control system capable of steering the robot to a target position, an inertial navigation system, and software related to server communication.

Later on Håland (2021) [19] improved upon the navigation system of the nRF52832-robot which also was adapted for the nRF52840-robot and serves as a starting point for further software development for the author's specialization project [5]. In [5] a newly built nRF52840-robot closely based of the design by Jølsgård was put into use. During work on the specialization project, the robot's sensor suite was calibrated in collaboration with Lindefjeld (2021) [27], and tuning of the control and navigation system was conducted. Furthermore, a line feature extraction method was developed and implemented for the nRF52840-robot, however, due to server communication issues testing the method was restricted by that the robot had to extract line segments while at rest in order to collect results over a wired connection. The problems related to server communication over Bluetooth is likely caused by multiple access to the shared resources of the communication module. This became apparent during testing in [5] when both point clouds and extracted line segments were tried to be sent concurrently. Moreover, the resulting detected line segments were questionable to be used for a reliable SLAM method.

Two server implementations are currently available for usage in the SLAM-project. The first server was developed in Java in 2016 and is referred to as the Java-server. The current version of the Java-server is a result of the work last conducted by Lien (2017) [26]. Robots communicate with the Java server over Bluetooth Low-Energy (BLE), which was what [5] was trying to use for documenting the results from the line extraction method. Development of the second server implemented in C++ was started in 2019 motivated by that most Cybernetics students at NTNU are more familiar with the C++ programming language over Java in addition to the new communication stack available for the nRF52-series of robots. With the new communication stack the robots form a Thread-based mesh network and communicate with the C++-server over MQTT-SN. A *legacy layer* was designed by Grindvik (2019) [18] which acts as a connectivity chip for older robots to access the Thread network, and Murad (2021) [30] tried to implement an MQTT-SN client for the nRF52840-robot since these robots do not need the additional legacy layer. Murad was not able to successfully create a reliable messaging link using MQTT-SN from the newer robots, instead a CoAP messaging system was implemented which also removed the need for a Thread Border Router. However, the C++-server is not currently able to communicate with the robots over CoAP.

7 System overview from starting point

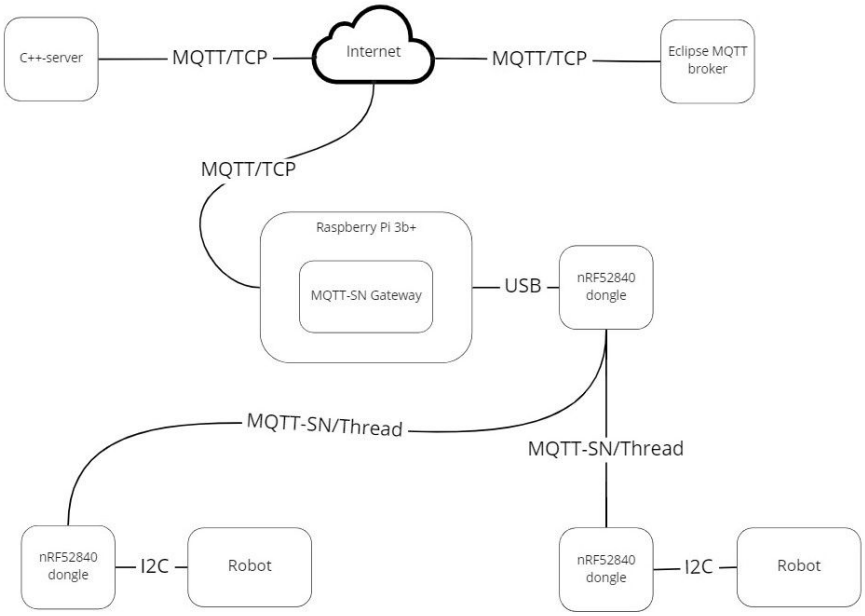


Figure 1: System overview

7.1 Robot hardware

The main hardware components of the robot are briefly outlined in the following sections. Figure 2a-2c are illustrations of the robot from different perspectives with descriptions of the components in use, and its dimensions.

7.1.1 nRF52840 Development Kit

Nordic Semiconductor's nRF52840 Development Kit (DK) board PCA10056 is used for developing embedded applications for the nRF52840 System-on-Chip (SoC) which serves as the main computing module for the robot. The nRF52840 SoC embeds a 32-bit ARM Cortex-M4 central processing unit (CPU). Using the development kit for programming the SoC enables fast prototyping of Bluetooth, Thread, and various other wireless communication applications due to the rich set of peripherals the DK provides. In addition to the 802.15.4 2.4 GHz radio, the board has 64 Mb of external flash memory available, easily accessible GPIO (General Purpose Input/Output) ports for additional peripherals, programmable buttons and LEDs, and power supply options such as both a USB- and Lithium Polymer (Li-Po) battery connector. [33]

7.1.2 Machifit 25GA370 DC motors

The differential drive robot's wheels turn using two separate 12V DC motors from Machifit. Unfortunately there seems to be no known datasheet for the motors. However, the motors are stated to be capable of running at 110 rpm from the website where they were purchased. Additionally, each motor has built-in quadrature encoders for measuring wheel displacement.

7.1.3 L298N Motor Driver

The L298N motor driver board is based on the L298 Dual H-bridge integrated circuit. The purpose of the board is to provide an analog interface for controlling the two DC motors independently. The nRF52840 SoC uses PWM-signal generation as input to the L298N for controlling each wheel's rotational velocity, whilst digital signal inputs to the motor driver board provide the means for controlling the rotation direction. The input PWM-signals are essentially low-pass filtered and boosted by the motor driver board in order to provide a stable 0-12 voltage to each of the motors. [48]

7.1.4 IR sensor tower

On top of the robot there is a set of four 2YA21 Sharp infrared (IR) sensors. The IR sensors are placed radially with respect to each other mounted on a S05NF servo motor to make the set of IR sensors rotate. The sensor tower has a maximum rotation angle of 90°, with a resolution of $\sim 1°$. Each of the IR sensors have a valid measurement range of $0.1 - 0.8$ [m] [42]. One of the IR sensors used in [5] was faulty, and was therefore replaced with an equivalent model.

7.1.5 ICM-20948 Inertial Measurement Unit

The ICM-20948 Inertial Measurement Unit (IMU) from InvenSense is located on the bottom part of the robot's chassis. The IMU consists of a 3-axis MEMS based gyroscope, accelerometer and compass. Additionally the board is equipped with a digital motion processor used for simple filtering of sensor measurements, and power management. The IMU's datasheet [21] claims that it is the world's lowest power 9-axis motion tracking device, and is a good fit for *smartphones, wearable sensors and IoT applications.*

7.1.6 Custom peripheral shield

The custom-made peripheral shield by Jølsgård [22] is attached on the top of the nRF52840 DK. The peripheral shield's primary purpose is to easily connect all peripherals mentioned above to the GPIO headers of the development kit.

7.2 Robot application

The code base at project start-up from [5] consisted of mainly five threads of execution, these were:

- A pose controller. Runs a PID controller for controlling the robot's position and heading to a target position. The control output is fed as speed references to the underlying wheel speed controllers. The pose controller thread is structured as a finite state machine where its current state is dependent on the robot's current pose. When the pose controller thread is in the `moveForward` state, the robot will move in a straight line towards its target position. In this case only the residual distance of the robot's current position and target is used as feedback. When the pose controller is in the `moveClockwise` state, the robot rotates in place in the clockwise direction. Similarly, the robot rotates counter-clockwise whilst at rest when in the `moveCounterClockwise` state. The robot transitions from the `moveForward` state to rotating in place whenever the angle to the target position exceeds a given threshold. Lastly, when the robot arrives at the target position it is stopped and enters the `moveStop` state.
- A speed controller. Runs two PID controllers for controlling each wheel's rotational velocity by providing a suitable varying voltage to the L298N motor driver. Feedback is provided from the built-in encoders of the DC-motors.

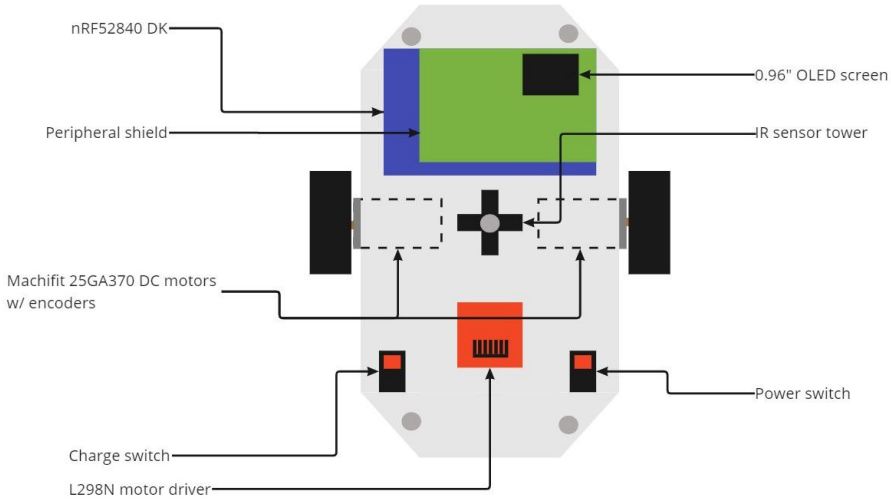
- An estimator. Implements an Extended Kalman Filter (EKF) for pose estimation utilizing measurements from the wheel encoders and IMU. Note that due to the high noise levels of the IMU, the EKF primarly relies on the encoders for pose estimation.
- A sensor tower task. Responsible for actuating the servo motor turning the tower, as well as sampling range-bearing measurements from the four IR sensors. The sensor tower only rotates whilst the robot is stationary. The robot still samples IR sensor measurements while the robot moves, however, the sensor tower does not rotate.
- A mapping task. Runs the line extraction process from [5].
- A communication task. Depending on the configuration from `robot_config.h` handles either Bluetooth communication for use with the Java server, or interfaces an additional onboard nRF52 dongle over I2C for MQTT-SN communication with the C++ server.

The application is built using Nordic Semiconductor's nRF5 SDK version 15.0.

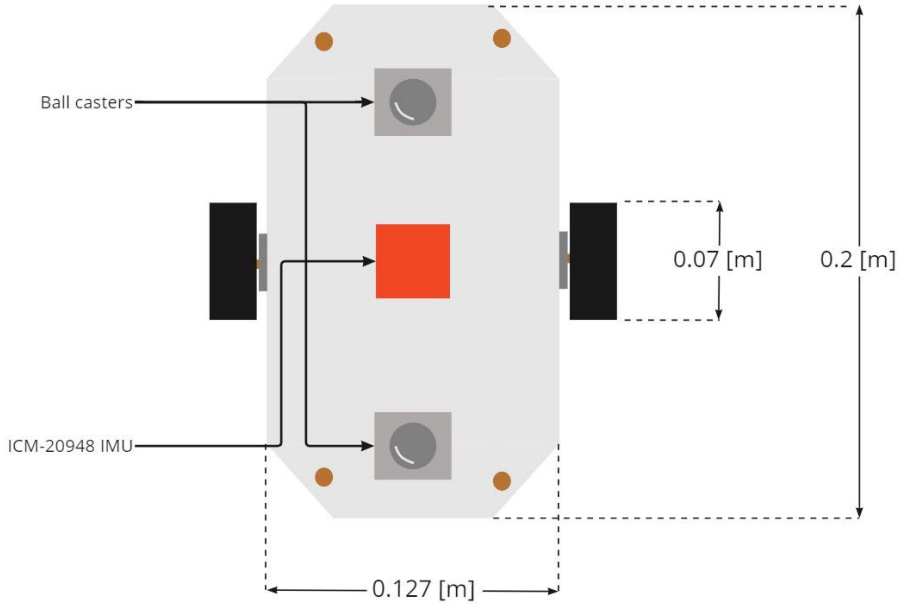
7.3 Server application

The C++-server application is also multi-threaded and its main threads of execution are:

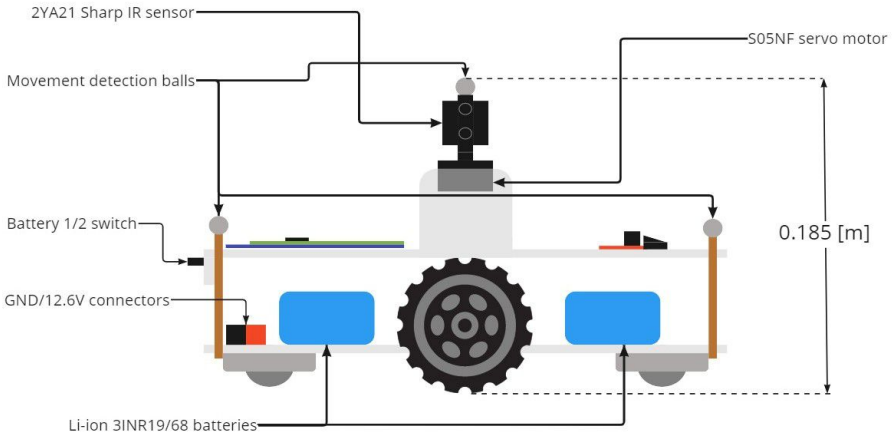
- A renderer for a graphical user interface (GUI). Used to display the robot's location and a map.
- An MQTT client for communicating with robots. The MQTT client transmits and receives messages through a public broker maintained by Eclipse as illustrated in figure 1.
- A RBPF-SLAM solution for simultaneously estimating each robot's pose and generating a map. For more details refer to Mullins (2020) [29].



(a) Robot top view



(b) Robot bottom view



(c) Robot side view

8 Theory

The following sections will first present the reader with theory related to online SLAM, a collection of pose- and map estimation techniques for mobile robotics applications. The SLAM problem will be described with special emphasis on how the problem has been solved by others with similar low-cost hardware as is used for robots of the SLAM-project. Secondly, theory about the nRF52840-robot's communication stack will be presented covering both Thread and MQTT-SN. Lastly, a brief introduction to the real-time operating system FreeRTOS will be given.

8.1 SLAM

Simultaneous Localization and Mapping (SLAM) is an estimation problem at the heart of autonomous mobile robot navigation. A solution to the SLAM problem provides estimates of the robot's relative position to its surrounding environment. Mathematically we may express the estimation problem as finding the *a posteriori* estimate of the probability density function describing the joint pose-landmark vector $\boldsymbol{\nu_k} = \begin{bmatrix} \boldsymbol{x_k^T} & \boldsymbol{m_k^T} \end{bmatrix}^T$ at every discrete time-step k. The pose vector $\boldsymbol{x_k}$ typically holds information on the robot's position and heading, and the landmark vector $\boldsymbol{m_k}$ stores landmarks making out the robot's explored surroundings. Thus, as the name implies, SLAM may be described as the process of a robot concurrently mapping out an *a priori* unknown environment and placing itself in the map being built.

The most common use case for SLAM methods are for autonomous robots in GNSS denied environments, i.e. in situations where the robot does not have any prior knowledge of its local surroundings. Typically the operating environments of SLAM robots either have limited GNSS-signal access, or none at all. Examples of such operating environments are inside buildings, underwater, in mountainous or urban areas with large obstructions, or on another planet than Earth. Even under circumstances when GNSS is commonly available it is not uncommon for autonomous robots to tackle the SLAM problem to increase the navigation system's robustness in case of GNSS-signal loss. For GNSS to work at all there must always be at least three GNSS satellites available, thus rendering GNSS a vulnerable system.

Other factors that motivate for usage of SLAM methods is the generation of a map itself as the robot explores the environment. The map may be used as a core part of other systems of the robot, for instance for path planning and to aid in collision avoidance. Moreover, the map of the environment may serve as a visual tool for human operators.

All SLAM systems depend on the use of *exteroceptive* sensors to capture measurements of the robot's environment. Typically SLAM systems employ ultrasonic sensors, LiDARs, or various types of cameras (these types of systems are commonly referred to as Visual SLAM systems).

A key point to how most SLAM implementations work is that they exploit the inherent correlations between consecutive robot poses and landmarks. Take for instance a robot that gathers odometry measurements from its wheel encoders for measuring the robot's relative displacement from its starting position. Furthermore, the robot is equipped with some kind of exteroceptive sensor for collecting range-bearing measurements of local landmarks. If the robot is able to associate multiple range-bearing readings with the same landmark as the robot moves closer to that landmark, intuitively the robot may increase its belief in its own position and orientation as the odometry and range-bearing measurements correlate with the robot moving towards the landmark. In the case of the robot revisiting an area it has previously explored (and the robot is able to correctly identify that it has been in the same location before) SLAM implementations may try to exploit this match in a way that uses previous correlations to update and correct the entire map. This is commonly referred to as a *loop-closure*. Loop-closures are a key component in the success of SLAM methods, however, they also highlight one of the major vulnerabilities of SLAM. Wrongly claiming that a loop-closure has taken place will for many systems make the robot not able to further function properly and the robot's state estimation will be left unrecoverable.

Most SLAM architectures are characterized by being separable by a front-end and a back-end. The front-end's responsibility is to interface the sensors used by the SLAM method. Mainly this

involves gathering sensor data, and performing any necessary pre-processing steps on the data before the information is fed into the back-end. The back end is responsible for performing the optimization steps required for estimating the system state. At its core the back-end performs *maximum a posteriori* (MAP) estimation, thus seeking to find the pose-landmark vector which minimizes errors in our predictions given a set of measurements. The methods used in the back-end of a SLAM solution are typically divided into two paradigms, these are; recursive filtering approaches, and probabilistic graph methods. The recursive filtering approaches include various flavours of Kalman Filters and particle filters. Graph methods address some of the drawbacks of many of the filtering approaches, for instance they quickly become computationally demanding with a large amount of landmarks. Graph methods try to take advantage of the structure of the SLAM problem with graph optimization techniques.

[9] [11]

8.1.1 Map representation

Various SLAM methods may typically be placed into one of two categories mainly dependent on how the developer of the method chooses to represent the robot's surroundings. These are, feature-based and dense SLAM methods.

Dense SLAM methods are characterised by that there is no need to manipulate measurements from exteroceptive sensors to extract features from the environment. I.e. the measurements from exteroceptive sensors are used directly as landmarks to be used in the estimation problem. Using dense SLAM methods, 2D or 3D occupancy grids are commonly used for representing the map. An occupancy grid is a set of cells with a given resolution and position that may represent a probability or simply a boolean indicating an obstruction in the environment corresponding to that particular cell. On the other hand, feature-based SLAM methods process measurements from exteroceptive sensors in such a way that features of the environment are collected. The most basic features are lines or line segments corresponding to walls of the environment. Other features may be corners, trees, or any object that may be extracted from sensor readings of the surroundings.

8.1.2 Dense vs. feature-based

The main advantage of dense SLAM methods over feature-based methods is that the information loss of not trying to find features from sensor measurements is kept at a minimum. I.e. dense methods do not care about the classification of the sensor data, only the data itself. One can only classify data to a finite set of categories. Thus, there will always be a risk of throwing away parts of the information stream from sensors that might be vital in order to accurately generate a map. Followingly, it should not come as a surprise that dense SLAM methods generally require more storage resources and computational power than feature-based methods since the sensory data has not been aggregated in any way. Therefore constrained embedded devices, such as the nRF52840 robot, are in general not fit for dense SLAM methods. However, feature-based methods still rely on the pre-processing step of feature extraction which also might be troublesome with regards to the extra computational effort required.

8.1.3 SLAM with sparse sensing

Most SLAM implementations rely on exteroceptive sensors that provide accurate and dense measurements. However, less work has been done on SLAM methods where range-bearing measurements are considered sparse with higher noise levels. Similar cases to the sensing system for the nRF52840 robot which uses range-bearing measurements from low-cost IR sensors is even less mentioned in litterature. Using an array of four slowly rotating IR sensors for mapping is not the most conventional use of IR sensors in robotics. More than often the IR sensor's limited sensing range (typically < 1.0 m) and dependence on the reflectance properties of surfaces renders them primarly used for binary obstacle detection and collision avoidance [8] [1].

The research considering sparse sensing SLAM methods is motivated by lowering the cost of the robot, which in turn would make autonomous robots more accessible for consumers and educational institutions. Moreover, many of the sensing systems for robotic SLAM applications may simply not be fit for smaller robots. Smaller robots, such as the nRF52840 based robot, must adhere to more strict constraints regarding the physical size and weight of the equipment, power consumption, and the complexity of neccessary software drivers for extracting data from the sensors. To the author's knowledge, only filtering-based approaches to solving the SLAM problem have previously been used for robot's with a low-end sensor suite and where only sparse exteroceptive measurements are available.

Beever and Huang [7] developed a SLAM method based on a Rao-Blackwellized particle filtering (RBPF) approach with line and line segment features for a similar low-cost sensing system as the nRF52840 robot. Particle filtering is a nonlinear filtering technique based on random sampling. Unlike Kalman filtering approaches, the particle filter does not assume that the estimated state vector is Gaussian distributed. Instead, the particle filter will estimate the state density by iteratively resampling a set of weighted particles representing most likely robot poses and landmarks. Particles that do not match the observed state from odometry and range-bearing measurements are discarded, and new samples are redrawn from the estimated density during the next update. A common problem with the particle filtering approach is that a large number of particles are needed in order to estimate the state with sufficient accuracy. Hence, many particle filtering implementations adopt the Rao-Blackwellized variant which aims to increase the computational efficiency by exploiting that not all system states necessarily require nonlinear models. Rao-Blackwellization essentially boils down to using random sampling for the nonlinear states, and Kalman filtering for the linear states [9]. Beever and Huang's approach is based on collecting multiple sparse scans of the environment whilst the robot moves in order to group the range-bearing readings into what they refer to as *multiscans*. They argue that the tradeoff in the uncertainty of the robot pose whilst it dead-reacons between updates of multiscans is low enough in order to more accurately extract features that are better suited to be used in the estimation process. SLAM updates are only performed after a multiscan has been formed after a given number of robot poses. To further increase efficiency their particle filter implementation does not keep track of each pose between multiscans in order to avoid extracting features for every particle for each pose of the multiscan.

Similarly to Beever and Huang, Yatim and Buniyamin [53] developed a RBPF SLAM algorithm with low-cost IR sensors. However, Yatim and Buniyamin opted for a dense approach and use occupancy grids to represent the map as opposed to a feature-based map. The probability of a cell of the occupancy grid being obstructed is computed by a neural network with the pose of the robot and range-bearing measurements as inputs. The authors advocate for the usage of the machine learning method due to it being hard to establish sufficiently accurate measurement models for IR sensors.

Abrate et al. [1] approached the SLAM problem with low-end IR sensors using an Extended Kalman Filter (EKF) with line features. Mane et al. [28] also use the EKF for pose and map estimation, and also use the same Sharp IR sensors as are in use for the nRF52840-robot. For Gaussian and linear state dynamics the Kalman Filter provides a well-established framework for recursive state estimation, illustrated by the block schematic in figure 3. The Kalman Filter provides online estimates through two important steps; a prediction, and update step. Specific for SLAM, when no information about the robot's local surroundings are available the robot relies on the prediction step based on a motion model describing the robot's kinematics. Predictions are typically computed from the control input, and/or readily available high-frequency measurements. The output of the prediction step is a prediction of the state vector $\boldsymbol{x}_{k|k-1}$ and a state covariance $\boldsymbol{P}_{k|k-1}$ representing the filter's belief in the state at the next time step k, given what is known about the system state at the current time step $k-1$. The Kalman Filter performs an update sequence when information about the environment is available, with the optional step of extracting features from the raw exteroceptive data. The resulting computations of an update sequence gives $\boldsymbol{x}_{k|k}$ and $\boldsymbol{P}_{k|k}$ which ideally should represent the most accurate estimate of the system state and its uncertainty at time step k. The output from the update step may be viewed as a weighted correction step, where one compares an exteroceptive measurement to the filter's prediction. Depending on how the different states and measurements are weighted, the update step will correct the robot's predictions according to what it is currently observing. For linear systems exposed to Gaussian

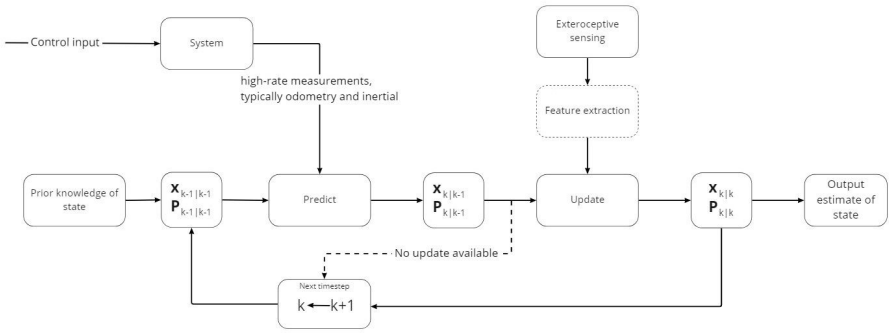


Figure 3: Block schematic for Kalman Filtering framework for SLAM. Adapted from Mane et al. [28]

noise the Kalman Filter will mathematically provide optimal estimates. In the case of systems with nonlinear dynamics, such as the movement of a differential drive robot in 2-dimensional space, it is still possible to use the Kalman filtering framework. However, the nonlinearities of the problem must be handled appropriately. In the case of the EKF the nonlinearities are handled by means of linearization. The EKF will linearize the system's process model and measurement model around the most recent estimate for the prediction step, and the latest prediction for the update step of the filter.

Choi et al. [12] also propose to use the EKF for low-end exteroceptive sensors for solving the SLAM problem and additionally constrain the problem to environments consisting of only parallel or perpendicular walls. Similarly, Yap and Shelton [52] use the orthogonality assumption for their low-cost sonar-based RBPF-SLAM approach.

8.1.4 Feature extraction with sparse sensing

Beevers and Huang's Rao-Blackwellized particle filter SLAM solution was implemented with both line and line segment features. Their feature extraction algorithm initially performs a threshold based clustering of points from a multiscan which are transformed to a global Cartesian frame. The feature extraction scheme followingly performs Iterative Endpoint Filter (IEPF) clustering on each of the previously found clusters. The IEPF clustering is used to group points that belong to the same line. The initially distance based clustering is required since the IEPF scheme expects a set of points that form line segments with connecting endpoints. Thus, the initial clustering step functions as a breakpoint detector. Computation of maximum likelihood line parameters of each IEPF cluster concludes the line extraction process. Using the maximum likelihood method one is able to also compute the covariance of the extracted line feature. For extracting line segments, the same line extraction algorithm is employed, however an additional step of finding the endpoints of the line is required. This was done by orthogonally projecting the Cartesian points onto the line, and letting the pair of projections which make up the longest segment represent the endpoints.

Abrate et al. also extract line features for their SLAM implementation. After a given number of raw range-bearing measurements from the robot's IR sensors are collected their line extraction implementation initially perform a segmentation step. The segmentation step involves grouping the measured points of the environment in a sliding window, and for each group of points a non-linear fitting scheme is employed. This is followed by a merging step where similar lines are merged together, and an information filter is used to compute an expression for each of the extracted line features' uncertainties.

The feature extraction method employed by Choi et al. consists of five steps. The first step entails sampling range-bearing measurements over a number of poses, and computing a set of point cloud means from the raw measurements to reduce both the number of points and noise level. After sorting the points by bearing, the IEPF based clustering method is used to find points that make up straight line segments. In the third step a least square line is computed for each IEPF based

cluster. Since the least squares line fitting strategy is prone to computing inaccurate line segments with respect to the actual environment due to outliers, Choi et al. use a modified version of the Hough Transform to counteract the effect of outliers in addition to estimate the line parameters' uncertainties. Their version of the Hough Transform uses the initial estimates of the line provided by the least square fit to perform a constrained search for line features in Hough space.

8.2 Thread

Thread is an open standard full-stack architecture for wireless IoT applications. Furthermore, Thread may be described as a framework for creating wireless personal area networks. Thread is mostly used in the domain of consumer oriented home automation, however, is also used for industrial applications such as asset monitoring [20].

OpenThread is an open-source implementation of Thread released by Google Nest certified by the Thread Group. OpenThread implements the main part of the communication stack used for creating the MQTT-SN client on the nRF52840 robot.

8.2.1 Stack

As depicted in figure 4, the link- and physical layers of the Thread stack is implemented with IEEE 802.15.4. The physical layer operates at 250 kbps in the 2.4 GHz radio frequency band [4]. The link layer uses CSMA/CA (Carrier-sense multiple access with collision avoidance) as its multiple access method in order for nodes to be able to *sense* the channel's availability and provides link layer re-transmission.

The network layer of the stack is carried out by 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks), a version of the IPv6 protocol designed for constrained battery-powered devices. The network layer enables IP addressed end-to-end packet delivery across networks.

UDP (User Datagram Protocol) serves as the transport layer of the Thread stack. The Thread Specification states that TCP (Transmission Control Protocol) may optionally be used although 6LowPAN-based stacks tend to run into performance issues with the packet flow control overhead of TCP in contrast to UDP [20].

The application layer protocol of the Thread stack is CoAP (Constrained Application Protocol). Similarly to most HTTP(S)-based APIs CoAP follows the principles of the REST model. I.e. application message flow is characterized by a request/response architecture; a CoAP server exposes its resources through a URL and CoAP clients may retrieve and/or manipulate the resources with `GET`-, `POST`-, `DELETE`-, etc. requests. Unlike HTTP, CoAP is designed specifically for embedded IoT devices. For the Thread stack CoAP is used for the control messaging procedures involved in for instance how a node is assigned to take on the Leader role, and for managing IP addresses [43].

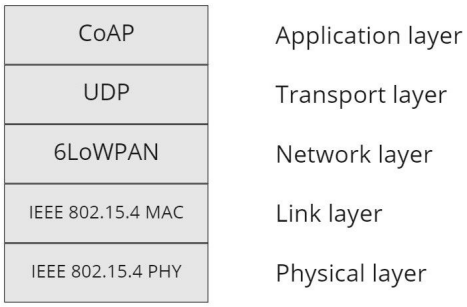


Figure 4: Thread stack

8.2.2 Motivation for Thread

There are several reasons that motivate for usage of Thread for connected embedded applications other than the stack's design emphasis on conforming to the constraints of these types of devices. Firstly, being an IP-based protocol Thread networks are able to seamlessly incorporate into other

IP networks. E.g. battery-powered devices forming a Thread network may easily be monitored by a smart phone or laptop over IEEE 802.11 (Wi-Fi) connected to a local area IP network.

Secondly, Thread is designed to provide reliable end-to-end communication. Thread's reliability is based on the concept of having no single point of failure due to that the devices forming the Thread network are able to take on different roles depending on the current active nodes of the network. I.e. when a certain node fails Thread defines the mechanisms that should take place in order for the network to automatically *heal* and re-configure. For instance reassigning the failed node's role to another (active) node.

Moreover, the Thread stack implements mesh routing in order to further strengthen end-to-end reliability of message delivery. This entails that Thread devices are able to forward messages via other certain types of devices of the network. The mesh forwarding feature of Thread enhances the robustness of the network in situations where a node is not able to directly communicate with another node due to for example signal fading or interference.

Lastly, considering Thread was designed for end-devices that might control various processes such as light switches at home or actuators in industrial plants, security becomes a major consideration. Therefore, the Thread architecture is implemented with security in mind in several layers of its stack. UDP datagrams are secured by DTLS, a variant of TLS, which in the case of the Thread implementation for the nRF52840 robot encrypts datagrams with AES (Advanced Encryption Standard), and provides other cryptographic mechanisms to hinder man-in-the-middle attacks such as manipulating in-flight messages and authentication schemes to verify the sender. The IEEE 802.15.4 protocol also provides security at the link-layer, thus ensuring that frames are secured over each hop in the mesh network on the way to their end destination. Furthermore, as mentioned above Thread may also be described as a framework for deployment of wireless personal area networks, this is especially evident considering the process of a device joining the Thread network referred to as *device commissioning*.

8.2.3 Device types

Devices constituting a Thread network may be divided into two types; routers and end-devices. The main tasks of a router is to forward packets to other devices on the network and serve as a device commissioner when a new node attempts to join the network. End-devices are typically only able to take part in the Thread network via a router, and are not capable of forwarding packets to other devices. Thus, one may label an end-device as a child, whilst the router the end-device communicates with (directly i.e. one hop) is its parent.

There is a special type of router that extends the standard Thread router's specification referred to as a *border router*. The border router is responsible for providing connectivity to other IP-based networks that run on other physical layers than IEEE 802.15.4. The most common use cases for a border router is to unite a Thread network with a LAN/WAN or the internet with Ethernet or Wi-Fi.

End-devices may be further split into four types; Router Eligble End-Devices (REEDs), Full End-Devices (FEDs), Minimal End Devices (MEDs), and Sleepy End Devices (SEDs). REEDs may serve as both a router and end-device. The role the REED takes is dependent on the current topology of the network or network conditions. The transition from a router to an end-device and vice versa is managed automatically by the network. FEDs do not have the ability to take on the role as a router. However, FEDs are normally categorized into the same group of devices as routers and REEDs referred to as Full Thread Devices (FTDs) since these types of nodes are required to always have their radio on, and will maintain IPv6 address mappings in order to link with all neighbouring routers. On the other hand, the last two groups of devices MEDs and SEDs make up the class of devices referred to as Minimal Thread Devices (MTDs). MTDs do not keep track of IPv6 address mappings, thus, they are only able to communicate through a single router. MEDs always have their radio turned on, however, SEDs may turn off their radio to lower their power consumption. Whenever a SED turns on the radio it must poll for awaiting messages from its parent router. [4][51][20]

8.3 MQTT

For a better understanding of the MQTT-SN protocol it is necessary to be familiar with its *parent* protocol MQTT, Message Queuing Telemetry Transport.

8.3.1 Broker

The MQTT protocol is commonly referred to as a publish/subscribe protocol. All MQTT messages are published by clients to a server (not to be confused with the C++ server application), commonly denoted as a broker. The responsibilities of the MQTT broker are to

- manage connection/disconnection of clients,
- listen for published application messages (also referred to as `PUBLISH` messages) from clients,
- process topic subscription/unsubscription requests from clients,
- and distribution of `PUBLISH` messages to clients that have subscribed to the `PUBLISH` message's topic.

8.3.2 Motivation for MQTT

The publish/subscribe characteristic of the MQTT protocol is well suited for networks that require one-to-many distribution of messages which is one of the main reasons for the protocol gaining popularity in the domain of Machine to Machine (M2M) communication and Internet of Things (IoT). These types of applications are often comprised of networks of connected devices which are required to share large amounts of information with other devices in real-time. More than often the client devices comprising these types of networks require either all or only certain parts of the information passed over the network. Moreover, the identity of the devices sending information is not necessarily important compared to the content of the information itself. Furthermore, the protocol's small code footprint, simplicity and low bandwidth requirements due to small header sizes and minimal protocol exchanges between communicating entities makes the MQTT protocol ideal for M2M and IoT applications. [37]

8.3.3 Protocol stack

The Oasis MQTT protocol specification [37] states that MQTT may be run over TCP/IP, or over *any other network that provides ordered, lossless, bi-directional connections.* In other words, the network layer of the MQTT client implementation must be a connection-oriented protocol. The MQTT OSI protocol stack for a client running over TCP/IP with example link- and physical layers is illustrated in figure 5.

Unsurprisingly most *out-of-the-box* client implementations such as the Paho MQTT clients available for both C/C++ and python developed by Eclipse run over TCP/IP. Other implementations such as the FreeRTOS coreMQTT library provides an interface for application developers to specify custom underlying network drivers.

8.3.4 Topics

`PUBLISH` messages of the MQTT protocol are always labeled by a topic name, a series of characters used to match the different subscriptions. At the application layer, the MQTT client will only receive `PUBLISH` messages labeled by a certain topic if the specific topic has already been subscribed to by the client.

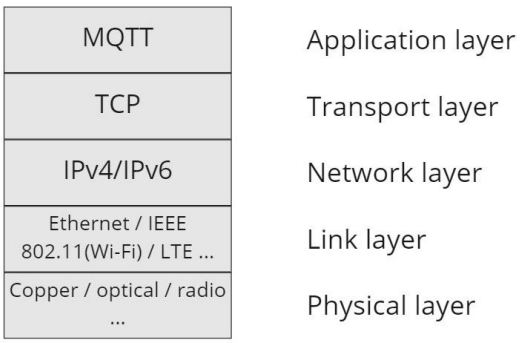


Figure 5: OSI model for MQTT over TCP/IP

Certain characters are reserved in order to handle filtering of topics when used in a topic name. Among the reserved characters are forward slashes ("/") which are used to structure the application messages into several levels. E.g. a set of connected mobile robots in different locations may want to exchange information with one another, whilst a central server aims to record all measurements from the robots in every location to provide insights for human operators. Robots in location A publish their own position to the topic name `robot/A/position` and their current velocity to the topic name `robot/A/velocity`, robots in location B publish messages to `robot/B/position` and `robot/B/velocity`. All robots in location A may subscribe to the topic `robot/A` in order receive updates on the whereabouts and velocities of other robots in the same area to aid collision avoidance. On the other hand, the central server would subscribe to the topic name `robot` in order to gather information from all of the robots in both location A and B.

8.3.5 Quality of Service

The MQTT protocol features three different types of application messages which dictate the level of reliability of message delivery referred to as Quality of Service (QoS).

QoS 0 `PUBLISH` messages are the least reliable types of messages. The MQTT protocol guarantees *at most once* delivery for QoS 0 messages, i.e., similar to the UDP protocol no application layer confirmation of successful delivery is required for these message types. The *at most once* guarantee for QoS 0 message delivery might sound overly pessimistic given the MQTT protocol's requirement of a *lossless* underlying transport service. However, a TCP packet being acknowledged at the transport layer does not necessarily make its way up to the application layer since the QoS guarantee holds for end-to-end distribution of MQTT messages (most of which are distributed to another client through a broker). E.g. a client publishing a QoS 0 message on a given topic may not necessarily be delivered to all clients subscribing to the given topic if the MQTT broker crashes before transmitting to all subscribing clients.

QoS 1 guarantees *at least once* message delivery, in this case a `PUBACK` message will be sent from the broker back to the client who sent the application message once the broker has received a `PUBACK` message from all subscribers. Note that QoS 1 messages may be duplicates of previously sent messages.

QoS 2 is the most reliable application message type as it guarantees *exactly once delivery* of messages. In order to guarantee that a message is only delivered once a process similar to a TCP 3-way handshake is required, obviously consuming the most time and network bandwidth compared to the other QoS-levels.

8.4 MQTT-SN

MQTT-SN, MQTT for Sensor Networks, is the application layer protocol used by the nRF52840-based robot for communicating with the C++ server. The following sections will give an overview of elements of the MQTT-SN protocol which are specifically important to know for understanding the implementation of the MQTT-SN client to be run on the robot.

8.4.1 Motivation for MQTT-SN

The MQTT-SN protocol was designed to resemble MQTT whilst conforming to the constraints of low-cost, battery-powered embedded devices. These types of devices are commonly characterized by their limited computational power and storage resources. Furthermore, MQTT-SN is adapted for usage in wireless networks. Wireless communication systems will always be effected by signal strength fading with distance, and disturbances due to interference tends to be more pronounced than in wired networks.

8.4.2 MQTT-SN protocol stack

Contrary to MQTT, the MQTT-SN protocol does not require a connection-oriented transport layer. Therefore, UDP is commonly used as the transport layer of the protocol stack. The smaller overhead of the UDP packets and connection-less properties of the protocol makes it more suitable for constrained devices and networks. For instance, consider the delivery of a QoS 0 MQTT application message. Although no guarantee is made for message delivery, TCP being a lossless and connection-oriented protocol will (if necessary) set up the required TCP connection, and follow the TCP packet control flow. For many systems that depend on a wireless network with many connected devices TCP packets would put an unnecessary large toll on the load capacity of the network. An illustration showing the MQTT-SN protocol stack run over Thread is given in figure 6.

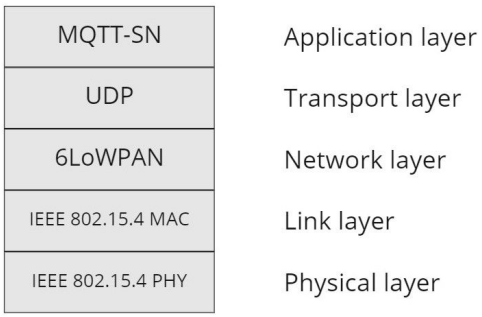


Figure 6: OSI model of MQTT-SN/Thread protocol stack

8.4.3 Architecture

In addition to the broker-client relationship of MQTT, MQTT-SN features *gateways* as part of the communication system's architecture. The purpose of a MQTT-SN gateway is to make devices on entirely different underlying network protocols capable of communicating through the same broker. This enables for instance a MQTT-SN client apart of a Thread network able to communicate with any IP-based network (e.g. internet) through a gateway as if both clients were apart of a network with the same underlying data-transfer services. In fact, any network which supports bi-directional traffic should be able to support MQTT-SN provided there exists a supporting gateway [46]. Figure 7 illustrates how the gateway functions as a translator between MQTT-SN and MQTT, thus *joining* two networks with different underlying protocols into the same network.

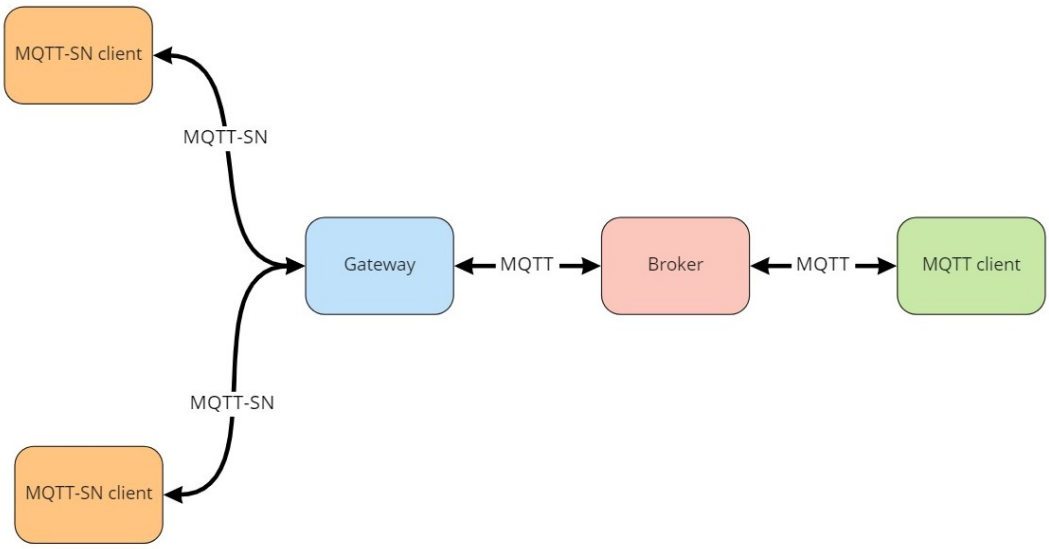


Figure 7: Illustration of MQTT/MQTT-SN communication architecture

8.4.4 Gateway discovery

Before a MQTT-SN client is able to connect to a broker a gateway discovery procedure must take place. After a gateway has successfully connected to a broker, the gateway will start to periodically broadcast a `ADVERTISE` message on the network to advertise to any MQTT-SN clients about its presence. When a MQTT-SN client discovers a gateway by receiving a `ADVERTISE` message, the client responds with a `GWINFO` message thereby confirming the client-gateway connection.

Following the standards of the MQTT-SN version 1.2 protocol specification [46], `ADVERTISE` messages should be sent relatively infrequently in order to prevent congesting the network. Therefore, in order to speed up the gateway discovery process the client may optionally broadcast a `SEARCHGW` message. When a gateway discovers a new client it also responds with a `GWINFO` message.

8.4.5 Topic registration

Unlike MQTT, the MQTT-SN protocol specifies that topics must be registered before any application messages may be published on the given topic. The purpose of topic registration is to increase the available space for the shortened message payloads, and to limit usage of network bandwidth.

Instead of transmitting a series of characters representing a topic name, a two-byte *topic id* is added to a `PUBLISH` message. Therefore, in order for the gateway to be able to translate MQTT application message to MQTT-SN (and *vice versa*) the MQTT-SN client must exchange the topic name string for a topic id before publishing or subscribing to a topic.

For a client to register a topic name the client must start the topic registration procedure by transmitting a `REGISTER` message to the gateway. In response to the reception of a `REGISTER` message the gateway will assign a topic id to the received topic name. The topic id is sent back to the client in a `REGACK` message. When the client receives the `REGACK` message the client is free to publish and subscribe to the given topic using the topic id. Note that a MQTT-SN client's

publish procedure is exactly the same as how the MQTT protocol states to publish messages for all QoS-levels.

8.5 FreeRTOS

The nRF52840 is ported with the Real-Time Operating System FreeRTOS. FreeRTOS is specifically designed for embedded applications that use microcontrollers with real-time constraints. At its core FreeRTOS implements a real-time scheduler which enables single-core processors to run multi-threaded applications.

FreeRTOS refers to each thread of execution as a *task*. The robot application is structured as a set of individual tasks assigned with different responsibilities for making the robot function as intended. Only one task is allowed to execute at the same time. Each task at any point in time will be in one of four states, managed by the scheduler. A task is in the `RUNNING` state when the task is executing. When a task is ready to be run, however, another task is currently in the `RUNNING` state with higher or equal priority to the task waiting to be run the task is said to be in the `READY` state. A task is in the `BLOCKED` state whenever the task is haulted in order to wait for a timer to expire, typically in conjunction with waiting for a an external event to take place, e.g. receiving information from another task. When a task is blocked, the scheduler decides which other task should switch to the `RUNNING` state. Additionally, a task may be in the `SUSPENDED` state. Unlike entering the `BLOCKED` state caused by waiting for an event to take place, a task will only be suspended if explicitly told so (either by the task itself, or another task). Similarly, to transition from the `SUSPENDED` state to the `READY` state the suspended task must also explicitly be resumed from another task. Tasks are assigned different priorities pre-runtime in order for the scheduler to choose which tasks should be run at a given time. Tasks with higher priorities are prioritized for execution over tasks with lower priorities. Moreover, each task is allocated its own stack memory when the task is created. Following the standards of FreeRTOS a task should never return (may of course be suspended from further execution by another task).

FreeRTOS exposes a rich set of API functions which may be used as building blocks for application development. For inter-task communication FreeRTOS provides software objects such as queues, notifications and message buffers. Synchronization primitives such as semaphores and mutexes are provided by FreeRTOS. Additionally, API's for memory management, task- and scheduler control, and software timers are readily available. [6]

9 Implementation

All software for the robot is implemented using Nordic Semiconductor's nRF5 SDK for Thread and Zigbee version 4.1.0. Segger Embedded Studio for ARM version 3.34a was used as the IDE for writing and compiling the source code.

The following two sections will describe how the multi-threaded MQTT-SN client was implemented for the robot application and how the Raspberry Pi was configured to serve as a Thread Border Router and broker. This is followed by a section describing a new line extraction method implemented for the nRF52840-robot, and lastly a line-feature based EKF-SLAM approach.

9.1 MQTT-SN client for nRF52840 robot

The MQTT-SN client implementation is based on Nordic Semiconductor's *Thread MQTT-SN client example* [36], and *FreeRTOS CoAP server example* [32]. Both of Nordic Semiconductor's examples utilize the Openthread API as an interface to the network's transport layer. The MQTT-SN client example demonstrates the usage of the MQTT-SN client by toggling one of the on-board LEDs of the nRF52840 DK when a publish message has been received on a certain topic. Similarly, the CoAP server example demonstrates toggling of LEDs as a response to receiving CoAP messages, and is also implemented with FreeRTOS. Since the MQTT-SN client example was not configured with any RTOS, the FreeRTOS CoAP server example was used as a starting point for the implementation. More specifically, the FreeRTOS task which runs the Openthread stack of the CoAP server example was taken into use, whilst the single threaded MQTT-SN client provided by Nordic Semiconductor's example was implemented in another task with an interface for other FreeRTOS tasks of the robot application.

9.1.1 Thread stack task

The thread stack task is responsible for initializing the thread interface, and drives the underlying thread stack of the MQTT-SN protocol. The implementation of the Openthread stack task can be seen in listing 1.

```
1  void thread_stack_task(void * arg)
2  {
3      UNUSED_PARAMETER(arg);
4
5      thread_instance_init();
6
7      // Notify MQTT-SN task
8      UNUSED_RETURN_VALUE(xTaskNotifyGive(mqttsn_task_handle));
9
10     while (1)
11     {
12         thread_process();
13
14         UNUSED_RETURN_VALUE(ulTaskNotifyTake(pdTRUE, portMAX_DELAY));
15     }
16 }
```

Listing 1: Thread stack task

The function `thread_instance_init()` is responsible for initializing the the thread stack task for the client to operate as a REED. This can be broken down into three main steps. Firstly, the function initializes all OpenThread drivers through a call to `otSysInit()` from the pre-compiled Openthread library. Secondly, the 802.15.4 embedded radio on the nRF52840 SoC is initialized for thread communication. Lastly, the Mbed TLS library with hardware acceleration provided by the SoC's Cryptocell module is initialized for multi-threaded usage. The Mbed TLS library provides the application programmer with a set of functions for performing cryptographic operations and is used to ensure both integrity and confidentiality of messages over the thread network. As many of the cryptographic operations are computationally demanding there is a dedicated piece of

hardware on the nRF52840 SoC able to handle these operations, e.g. AES encryption/decryption. After initialization the thread stack task notifies the MQTT-SN task to start its initialization procedure.

In the main-loop of the thread stack task the function `thread_process()` is called. This function wraps the Openthread API functions `otTaskletsProcess()` and `otSysProcessDrivers()` and is responsible for processing any pending tasks of the thread stack. After processing all tasks in its internal task queue the thread stack (FreeRTOS-)task will wait for the task queue to be filled up again before continuing execution.

9.1.2 Starting point for multi-threaded MQTT-SN client

The MQTT-SN client provided by Nordic Semiconductor's example runs as a single-threaded (bare-metal) application. Lower level serialization/deserialization, i.e. the process of adding/parsing the required MQTT-SN headers of a specific MQTT-SN packet in order to comply to the MQTT-SN specification, is provided by the source code from the *Eclipse Paho MQTT-SN C/C++ client for Embedded Platforms* repository [13]. Outgoing and incoming serialized MQTT-SN packets are sent and received using the Openthread UDP API [50].

The MQTT-SN client implementation is event-driven by the usage of Nordic Semiconductor's *Application Timer* SDK library [34]. The Application Timer library is used for managing multiple software timers based on one of the nRF52840 SoC's real-time counters (RTC1). The MQTT-SN client relies on software-based interrupts triggered by timeouts from a timer initialized by the library. In this way the MQTT-SN client may schedule asynchronous events to fulfill operations such as holding the connection to the broker alive by periodically scheduling a `PINGREQ` message, or retransmitting a lost `Qos=1` MQTT-SN packet after not having received the expected `PUBACK` message corresponding to the previously sent MQTT-SN packet.

The MQTT-SN client utilizes an internal packet queue for reliable delivery of `Qos=1 PUBLISH` packets, subscribe-, and topic registration requests. The packet queue temporarily stores unacknowledged packets that have been sent to the broker which are deleted upon reception of the packet's corresponding acknowledgement-type message (`PUBACK`, `SUBACK` or `REGACK`) for a given message id.

9.1.3 MQTT-SN task

The main responsibilities of the MQTT-SN task are:

- Initializing the MQTT-SN client, and dependencies of the client such as data structures and queues.
- Gateway discovery.
- Keeping the connection to the broker active.
- Topic registration.
- Topic subscription.
- Handling reception of messages for subscribed topics, and distributing these messages to the appropriate task.
- Dequeuing internal messages from other publishing tasks from an outgoing message queue, and publishing these messages on registered topics to the network.

In contrast to the previous implementation of server communication over Bluetooth, the MQTT-SN task may run at a lower priority than the other tasks of the robot application. This is a clear step forwards in terms of making the robot autonomous without the need for server interactions in

order to successfully navigate and map its surroundings. For instance, previous implementations of several FreeRTOS tasks of the robot would have to wait for establishing a connection with the server in order to start execution. This example underlines the previously strong coupling between the robot and server. It should be a goal that the robot is able to operate on its own making the server only serve as an interface for human operators, thus removing the master-slave relationship of the server-robot system. Moreover, the downgraded priority of the main communication task reflects the removal of the robot's dependency of the server, thus tasks such as the pose controller and estimator may be granted higher priorities.

9.1.4 Adapting MQTT-SN client for FreeRTOS

Since the MQTT-SN client provided by Nordic Semiconductor runs in a bare-metal environment, the client had to be adapted to work in a multi-threaded FreeRTOS application in order for existing work on the robot project to be taken into use.

Making the MQTT-SN client compatible for multi-threaded usage with FreeRTOS was solved by first implementing an own thread of execution for the MQTT-SN client itself referred to as the *MQTT-SN task*. Other FreeRTOS tasks interface the MQTT-SN task primarily through the FreeRTOS queue management API. Figure 8 illustrates how other tasks of the robot application communicate with the MQTT-SN task. Thick arrows in figure 8 represent shared queue objects between a specific task and the MQTT-SN task. Tick arrows pointing out from a task illustrate a task placing an instance of a data structure in the corresponding queue designated for publishing or subscribing to a certain topic. The data structure instances are passed by copy through each queue, and match the MQTT-SN payload format for the given topic. Be aware that a given queue may be used for any topic, the only requirement is that the datastructures passed through the same queue are the same. Passing the queued data structures by copy ensures that tasks publishing or subscribing at different rates than the MQTT-SN task is able to read or write to the collection of queues does not overwrite MQTT-SN packets in-transit between tasks.

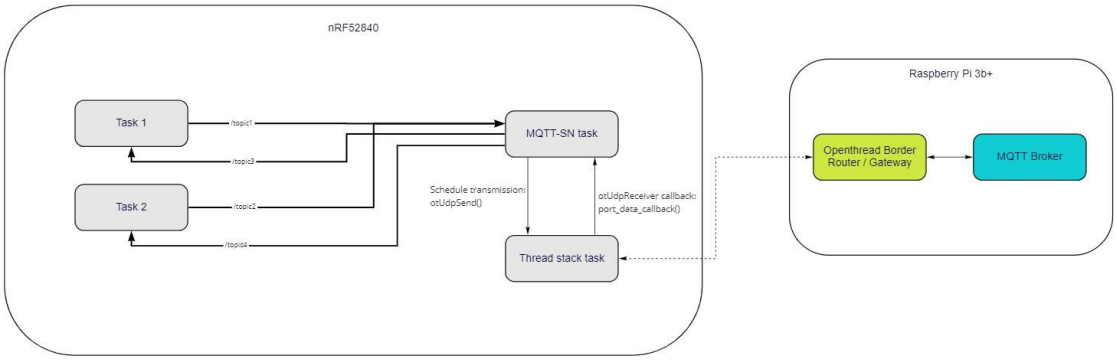


Figure 8: Illustration of MQTT-SN task interface

Secondly, in order to maintain the event-driven software architecture of the bare-metal MQTT-SN client Nordic Semiconductor's App Timer library was replaced by a FreeRTOS compatible version of the library. The libaries are implemented with a shared interface, however, the FreeRTOS version of the App Timer library serves as a wrapper for the FreeRTOS Software Timer Management API. Using FreeRTOS software timers one may schedule the execution of a callback function either periodically, or at some set time in the future. Unlike using one of the nRF52840 SoC's hardware timers directly, the software timers are controlled by the FreeRTOS kernel. Moreover, the software timers execute in the context of a timer service task. Thus, in order to ensure that the timing requirements of the MQTT-SN protocol are satisfied, the timer service task must have the highest priority. The priority of the timer service task is set by the `configTIMER_TASK_PRIORITY` macro in `FreeRTOSConfig.h`. Not setting the priority of the timer service task appropriately will for instance lead to the MQTT-SN client not being able to reply with a `PINGRESP` message fast enough after having received a `PINGREQ` message from the broker. This will make the broker believe the MQTT-SN client is not active anymore. It was also necessary to increase the timer service task's

pre-allocated stack size in order to accommodate for the increased stack usage of the MQTT-SN task's timer callback.

Thirdly, one of the precompiled openthread drivers (specifically the driver for the IEEE 802.15.4 radio) uses hardware timer 1 on the nRF52840 SoC. The encoder driver for the robot also initially used timer 1 for counting high-level voltage pulses as one of the wheels of the robot rotates. Therefore peripheral resources of the nRF52840 had to be swapped around in order to avoid the collision in peripheral usage. Table 1 summarizes which peripherals are in use specific for the robot application. Additional requirements for nRF52840 Thread support are listed in [41].

Peripheral	**Used by**	**Comment**
TIMER0	Sensor tower servo	PWM-signal generation.
TIMER1	IEEE 802.15.4 radio driver	Part of Thread protocol stack (6LoWPAN).
TIMER2	Motor	PWM-signal generation.
TIMER3	Left wheel encoder	Counts encoder ticks/pulses.
TIMER4	Right wheel encoder	Counts encoder ticks/pulses.
SAADC channel 0-3	IR sensor x4	Successive approximation analog-to-digital converter. For sampling range to objects.
RTC1	FreeRTOS scheduler	Real-time counter 1. For the FreeRTOS kernel's ability to decide when to resume/suspend and switch between execution of tasks.

Table 1: Robot application peripherals

9.1.5 MQTT-SN task interface

Other FreeRTOS tasks publish messages through a set of shared queues. For convenience, functions for queuing these messages were made. All publish functions intended to be used by other tasks follow the same logical structure, however, not all message payloads are defined by the same data structure. As an example, given in listing 2 is the function used for publishing data structures of type `mqttsn_line_msg_t`. The only difference between the different publish functions is the payload data structure, and followingly the outgoing message queue it is appended to.

```
typedef struct mqttsn_line_msg {
  uint8_t identifier;      // 1 byte
  int16_t xdelta;          // 2 bytes
  int16_t ydelta;          // 2 bytes
  int16_t thetadelta;      // 2 bytes
  coordinate_t startPoint; // 4 bytes
  coordinate_t endPoint;   // 4 bytes
} __attribute__((packed)) mqttsn_line_msg_t; // __attribute__((packed)) to
    explicitly tell the compiler to not add any padding

uint32_t publish_line(char* topic_name, mqttsn_line_msg_t payload, uint8_t
    payload_size, uint8_t qos, uint16_t msg_id) {
  if (!mqttsn_client_is_connected()) {
    return NRF_ERROR_BUSY;
  }

  mqttsn_line_msg_queue_element_t queue_element;
  queue_element.msg_id = msg_id;
  queue_element.qos = qos;
  queue_element.payload_size = payload_size;
  queue_element.payload = payload;
  queue_element.topic_id = get_topic_id(topic_name);
  if (queue_element.topic_id == NULL) {
    return NRF_ERROR_NULL;
  }

```

```
25   if (mqttsn_outgoing_line_message_queue != NULL && xQueueSend(
       mqttsn_outgoing_line_message_queue, &queue_element, 10) != pdPASS) {
26
27     NRF_LOG_ERROR("Failed to post mqttsn message to outgoing message queue");
28     return NRF_ERROR_NULL;
29
30   }
31   return NRF_SUCCESS;
32
33 }
```

Listing 2: MQTT-SN Publish line

When the client receives a publish message from the broker, similarly, the MQTT-SN task will distribute the message to other tasks through an inter-task queue specific for the format of the received message. For instance, the controller task may poll new target positions sent from the server by reading from the designated queue as seen in listing 3. Note that the application programmer must be aware which data structure is expected for a given topic.

```
1
2 typedef struct mqttsn_target_msg {
3   uint8_t identifier;
4   int16_t target_x;
5   int16_t target_y;
6 } __attribute__((packed)) mqttsn_target_msg_t;
7
8 if (xQueueReceive(get_queue_handle("v2/server/NRF_5/cmd"), &target_msg, (TickType_t
    ) 0) == pdTRUE) {
9     float x_target = target_msg.target_x;
10    float y_target = target_msg.target_y;
11    // Set new target position ...
12 }
```

Listing 3: MQTT-SN Receive target position

9.1.6 MQTT-SN topic and payload formats for server communication

Stenset (2020) designed the newest payload and topic formats of MQTT-SN messages sendt to the C++ server. With the introduction of the work carried out in this thesis involving line segment detection, a new message type for sending line segments was developed. In order to maintain consistency, the topic and payload format of the new line messages were designed to fit into the framework developed by Stenset.

For completeness, the MQTT-SN payloads and topics designed by Stenset are repeated in table 2-4. The message format for detected line segments sent from the robot to the C++-server is given in table 5. All payloads are little-endian formatted, and start with a 1-byte message code intended for the receiver for identifying the payload format. This is the way different messages are separated from each other by the C++-server. Note that applications that use MQTT(-SN) for communication typically separate different message payloads from each other by topic. However, the MQTT client running for the C++ server differentiates between received message formats with the message code, and therefore extending the C++ server to be capable of receiving the new payload format for line segments was easier to implement when using the additional 1-byte message code. Moreover, this allows for backward-compatibility with older robots that use the legacy layer developed by Grindvik (2019) [18] for communication with the C++-server.

The C++-server MQTT client subscribes to the topic `v2/robot`, and expects all received messages from robots to be published to a topic with the following format: `v2/robot/<ROBOT ID>/<RESOURCE>`. The first layer of the topic name corresponds to the message format version (currently at version 2). The second layer of the message format represents that the message was sent from a robot. The third layer is expected to be a unique string used for identifying a particular robot, for our case this is `NRF_5`. The last layer of the topic format describes the contents of the payload. As stated in the previous paragraph, the C++-server does not separate messages by topic, therefore for messages published to the C++-server the last layer's purpose is only for the developer's clarity.

This does not mean that the last layer may be omitted, this will cause the C++-server to reject the message. For the robot used during development, line segments may be published to the topic `v2/robot/NRF_5/line` with payload format as stated in table 5 in order for the C++-server to receive the message. The C++-server will discover a robot after a valid message has been received, and the user will be prompted with a GUI for selecting the robot's initial pose. After the user has confirmed the robot's initial pose, the C++-server publishes the message given in table 2 on the topic `v2/server/<ROBOT ID>/init`. Target positions sent from the server following the format given in table 3 are published on the topic `v2/server/<ROBOT ID>/cmd`.

Byte #	**Low/High byte**	**Parameter**
1	Low	Message code
2 3	Low High	Initial robot x position [mm]
6 7	Low High	Initial robot y position [mm]
8 9	Low High	Initial robot heading [°]

Table 2: MQTT-SN payload format for initial robot pose sent from server to robot developed by Stenset (2020)

Byte #	**Low/High byte**	**Parameter**
1	Low	Message code
2 3	Low High	Robot target x-coordinate [mm]
6 7	Low High	Robot target y-coordinate [,m]

Table 3: MQTT-SN payload format for target coordinates sent from server to robot developed by Stenset (2020).

Byte #	Low/High byte	Parameter
1	Low	Message code
2 3	Low High	Robot x position change [mm]
6 7	Low High	Robot y position change [mm]
8 9	Low High	Robot heading change [°]
10 11	Low High	IR sensor 1 x-coordinate
12 13	Low High	IR sensor 1 y-coordinate [mm]
14 15	Low High	IR sensor 2 x-coordinate [mm]
16 17	Low High	IR sensor 2 y-coordinate [mm]
18 19	Low High	IR sensor 3 x-coordinate [mm]
20 21	Low High	IR sensor 3 y-coordinate [mm]
22 23	Low High	IR sensor 4 x-coordinate [mm]
24	Low	Valid detection

Table 4: MQTT-SN payload format sent from robot to server for robot pose updates and objects detected by the four IR sensors developed by Stenset (2020).

Byte #	Low/High byte	Parameter
1	Low	Message code
2 3	Low High	Robot x position [mm]
6 7	Low High	Robot y position [mm]
8 9	Low High	Robot heading [°]
10 11	Low High	x-coordinate of line start point [mm]
12 13	Low High	y-coordinate of line start point [mm]
14 15	Low High	x-coordinate of line end point [mm]
16 17	Low High	y-coordinate of line end point [mm]

Table 5: MQTT-SN payload format sent from robot to server for robot pose updates and detected line segments.

9.1.7 Bug-fixing provided MQTT-SN client

The provided bear-metal MQTT-SN client provided by Nordic Semiconductor was not able to register multiple topics. Diving into the source code of the client revealed that MQTT-SN `REGACK` messages were dequeued from the client's internal received packet queue before reading the contents of the packet. Since dequeuing the packet also would deallocate the memory for the given packet reading the contents would result in undefined behaviour. I.e. it would in many circumstances appear to work as intended since the same region of memory would not have been altered, however, when registering multiple topics the problem became apparent. Since the 2-byte topic id would be lost, the client would receive error messages from the gateway when publishing to the topic corresponding to the erroneous topic id. The problem was fixed by ensuring that the `REGACK` packet was dequeued after reading the contents of the packet.

9.1.8 QoS 0

The MQTT-SN client was initially only capable of publishing with `QoS` $= 1$. Implementing `QoS` $= 0$ is straight forward as the client will not be expecting any acknowledgement of message delivery there is no need to buffer transmitted `QoS` $= 0$ messages. Since the underlying paho implementation for serializing MQTT-SN messages supports all MQTT-SN QoS levels, the only requirement for the application layer is to check the QoS level of the message. If the QoS level is 0, then simply publish the message and return. In order to add support for `QoS` $= 2$ one would have to implement an event handler for `PUBREC` packets as an acknowledgement for the publish which would deallocate the reference to the published message, followed by transmitting a `PUBREL` to signal the broker that it also may delete its locally stored copy of the initial publish. Furthermore an event handler for `PUBCOMP` packets must be implemented as an acknowledgement for completing the *exactly once delivery* process.

```
uint32_t mqttsn_packet_sender_publish(mqttsn_client_t * p_client, mqttsn_topic_t *
    p_topic, const uint8_t * payload, uint16_t payload_len, uint8_t qos) {
  uint32_t err_code = NRF_SUCCESS;

  unsigned char dup = 0;
  unsigned char retained = 0;
  // qos=-1 and qos=2 is currently not supported

  uint32_t packet_len = MQTTSN_PACKET_PUBLISH_LENGTH + payload_len;
  uint8_t * p_data = nrf_malloc(packet_len);
  uint8_t * p_packet_copy = NULL;

  do {
    if (p_data == NULL) {
      err_code = NRF_ERROR_NO_MEM;
      NRF_LOG_ERROR("PUBLISH message cannot be allocated\r\n");
      break;
    }

    MQTTSN_topicid topic;
    memset( & topic, 0, sizeof(MQTTSN_topicid));
    topic.type = MQTTSN_TOPIC_TYPE_NORMAL;
    topic.data.id = p_topic -> topic_id;

    uint16_t datalen = MQTTSNSerialize_publish(p_data,
      packet_len,
      dup,
      qos,
      retained,
      next_packet_id_get(p_client),
      topic,
      (uint8_t * ) payload,
      payload_len);
    if (datalen == 0) {
      err_code = NRF_ERROR_INVALID_PARAM;
      break;
    }

```

```
38      if (qos == 1) {
39        // Prepare retransmission packet in case of packet loss.
40        p_packet_copy = nrf_malloc(datalen);
41        if (p_packet_copy == NULL) {
42          err_code = NRF_ERROR_NO_MEM;
43          break;
44        }
45
46        memcpy(p_packet_copy, p_data, datalen);
47
48        mqttsn_packet_t retransmission_packet;
49        memset( & retransmission_packet, 0, sizeof(mqttsn_packet_t));
50        retransmission_packet.retransmission_cnt = MQTTSN_DEFAULT_RETRANSMISSION_CNT;
51        retransmission_packet.p_data = p_packet_copy;
52        retransmission_packet.len = datalen;
53        retransmission_packet.id = p_client -> message_id;
54        retransmission_packet.timeout =
55        mqttsn_platform_timer_set_in_ms(MQTTSN_DEFAULT_RETRANSMISSION_TIME_IN_MS);
56        retransmission_packet.topic = * p_topic;
57
58        if (mqttsn_packet_fifo_elem_add(p_client, & retransmission_packet) !=
      NRF_SUCCESS) {
59          err_code = NRF_ERROR_NO_MEM;
60          break;
61        }
62
63        if (mqttsn_client_timeout_schedule(p_client) != NRF_SUCCESS) {
64          uint32_t fifo_dequeue_rc = mqttsn_packet_fifo_elem_dequeue(p_client,
65            p_client -> message_id,
66            MQTTSN_MESSAGE_ID);
67          ASSERT(fifo_dequeue_rc == NRF_SUCCESS);
68          err_code = NRF_ERROR_INTERNAL;
69          break;
70        }
71      } else if (qos < 0 || qos > 1) {
72        NRF_LOG_WARNING("Publish with qos %d is not supported", qos);
73      }
74
75      err_code = mqttsn_packet_sender_send(p_client, & (p_client -> gateway_info.addr
      ), p_data, datalen);
76
77    } while (0);
78
79    if (p_data) {
80      nrf_free(p_data);
81    }
82
83    if (p_packet_copy && err_code != NRF_SUCCESS) {
84      nrf_free(p_packet_copy);
85    }
86
87    return err_code;
88  }
```

Listing 4: MQTT-SN client publish

9.1.9 MQTT-SN task initialization

The initialization sequence of the MQTT-SN task may be described by the flow diagram in figure 9. When the MQTT-SN task is notified after initialization of the Thread stack task, an MQTT-SN client instance is initialized by the MQTT-SN task. The initialization of the client instance involves creating a Openthread network port and initializing a timer used for scheduling events with the App Timer library. After the MQTT-SN client instance has been initialized it will always start in the `DISCONNECTED` state, and will immediately broadcast a `SEARCHGW` message and transition to the `SEARCHING FOR GATEWAY` state. The MQTT-SN task will not continue execution until it has received a direct-to-task notification using the FreeRTOS API function `ulTaskNotifyTake` (serving as a lightweight binary semaphore).

All following transitions are executed in the context of the timer task which periodically is scheduled to execute the event handlers (callbacks). In the event of receiving a `GWINFO` message from the gateway, the client transitions to the `GATEWAY FOUND` state and transmits a `CONNECT` message. When the timer task discovers that a `CONNACK` message has been received from the broker the client will start the topic registration sequence.

In order for topics to be registered they must be added to a list of topics before the robot application is run. The list holds the `mqttsn_topic_t` data structures given in listing 5, which holds the information that ties a topic name (string) to a numeric topic id. The topic id should be initalized to `NULL`, and the topic name must conform to the MQTT-SN protocol specification for topic names. One-by-one the client will exchange the topic names for topic ids with the gateway. When a `REGACK` has been received for each registration the MQTT-SN task will be woken up by a notification using `xTaskNotifyGive`.

```
typedef struct mqttsn_topic_t {
    const uint8_t * p_topic_name; /**< Topic name. */
    uint16_t        topic_id;     /**< Topic ID. */
} mqttsn_topic_t;
```

Listing 5: MQTT-SN Topic data structure

After topic registration, the client will perform a subscribe sequence. All topics which the client wishes to subscribe to must be added to a subscription list. The subscription list holds pointers to topics of the topic list. Iterating through the subscription list, each topic which have been given a topic id (!=`NULL`) will be attempted to be subscribed to.

Initialization of the MQTT-SN task is considered finished whenever the client has been able to successfully connect to the broker and executed the registration process. After initialization other FreeRTOS tasks of the robot application are able to queue messages to one of the MQTT-SN task's outgoing message queues.

9.1.10 Reconnection

The client will hold the connection to the broker alive by transmitting `PINGREQ` messages to the broker, this is also scheduled to run in the context of the timer task. In the event of not receiving a `PINGRESP` reply for a third `PINGREQ` within a set period of time (in our case approximately 8 seconds), the timer task will transmit a `DISCONNECT` message and transition to the initial `DISCONNECTED` state from figure 9. It is then up to the MQTT-SN task to start the initialization sequence again. The same reconnection process will also be triggered for not receiving a `PUBACK` for the third time of retransmitting the same `PUBLISH` with `QoS` $= 1$. If the robot application only uses `QoS` $= 0$ messages, the client will only be able to reconnect after not having received a `PINGREQ`.

9.2 MQTT-SN gateway and broker

All communication goes through the MQTT-SN gateway and broker. The MQTT-SN gateway and broker both run on a Raspberry Pi 3b+ connected to a local network. Placed in one of the

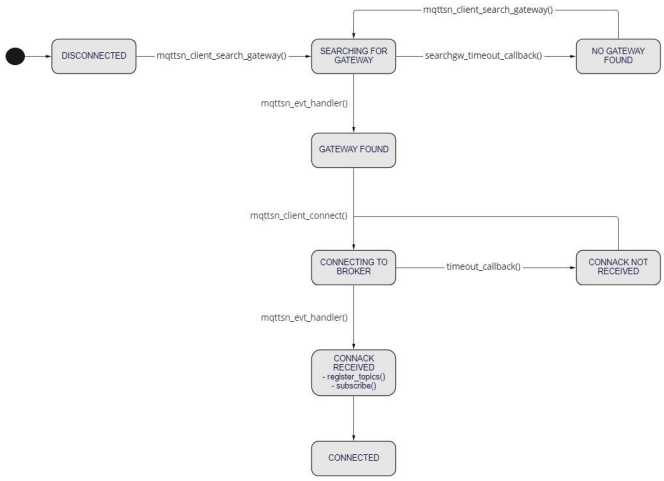


Figure 9: MQTT-SN task initialization flow diagram

Raspberry Pi's USB ports is a nRF52840 dongle. The dongle was flashed with Nordic Semiconductor's NCP (Network Co-Processor) firmware in order to serve as a connectivity chip utilizing the dongle's IEEE 802.15.4 embedded radio for taking part in the Thread network [35]. The Raspberry Pi was flashed with Nordic Semiconductor's debian based Thread Border Router image. The image includes software for the Eclipse Paho MQTT-SN gateway and its dependencies. The nRF52840 dongle together with the Raspberry Pi serve as the Thread Border router as depicted in figure 10.

A Mosquitto MQTT broker (version 2.0.14) was installed on the Raspberry Pi. The MQTT-SN gateway needs to know the IP address of the MQTT broker in order for the gateway to be able to listen for MQTT messages to be forwarded to the Thread network and forward messages from the Thread network to the broker. Since, the broker is run locally on the Raspberry Pi one must set the `BrokerName` entry in the gateway's configuration file `paho-mqtt-sn-gateway.conf` to the loopback address `127.0.0.1`. Furthermore, the Mosquitto broker needs to be supplied with a configuration file which makes the broker listen for connections on port `1883`, and allow anonymous connections.

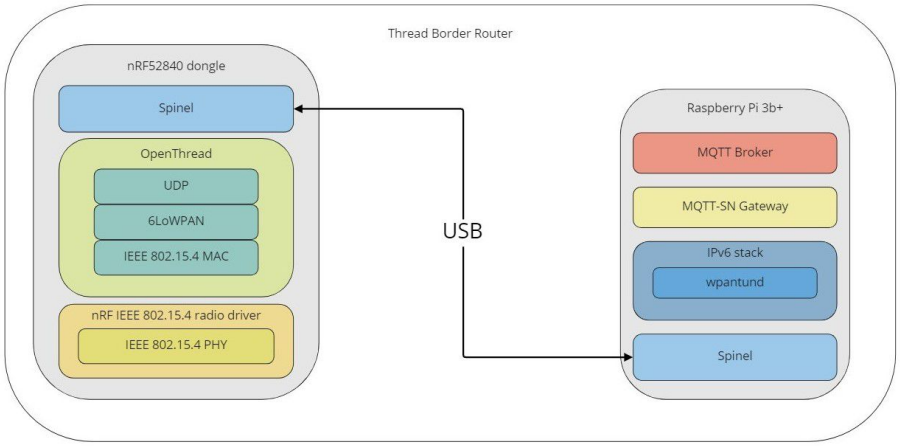


Figure 10: Illustration of a Thread Border Router with a Thread Network Co-Processor architecture. Adapted from [49]

.

9.3 Line segment extraction

The new line segment extraction algorithm implemented for the nRF52840 robot consists of a series of filtering steps, eventually outputting a set of line segments along with a measure of uncertainty of the extracted line parameters. The steps of the line extraction algorithm are illustrated in figure 11.

The first step in the line extraction scheme is a data collection (and preprocessing) step. In this step IR-range bearing measurements are sampled together with the current estimated pose of the robot in order to store multiple points of the environment in the global frame over several robot poses. Adopting a multiscan approach is a necessity in order to extract line features of the environment from measurements from the four IR sensors of the robot. Extraction of line segments commences either when *enough* points of the environment have been collected, the robot transitions from moving in a straight line, or turning around its own z-axis, or, whenever the robot has completed a full 360° scan of the environment by rotating the sensor tower 90° whilst the robot is not moving. When points of the robot's surroundings have been collected, and one of the aforementioned conditions to start processing the collected IR measurements have been met, the line extraction algorithm starts its second step. In the second step the clustering method commonly used for unsupervised machine learning applications known as *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) is performed on the buffered point-measurements of the environment. This is followed by a line segmentation step using *Iterative Endpoint Filter* (IEPF) based clustering to find the points which make up individual line segments. The fourth step of the line extraction scheme involves fitting the IEPF clustered points to a line. Up until the fifth step of the line extraction process, all computations have been done individually on the set of points stemming from one IR sensor. Thus, we may expect that several of the lines extracted so far actually are part the same line feature, therefore, a recursive merging of line segments is conducted in the fifth step in order to attempt finding line segments from the union of points collected from the neighbour IR sensors. The line parameters found after this step along with a measure of their uncertainty, function as measurements for the EKF-SLAM implementation.

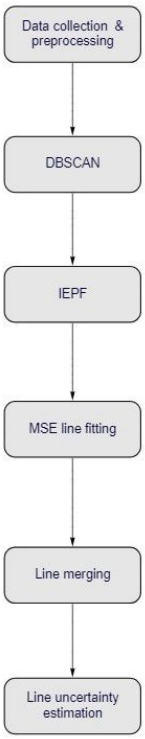


Figure 11: Line extraction steps

The line extraction algorithm runs in its own FreeRTOS task referred to as the *mapping task* given in pseudo code in algorithm 1. The task exposes an inter-task FIFO queue for the *sensor tower*

task to append IR sensor measurements and robot poses to as the robot explores the environment. The following sections will go into depth on each step of the extraction process.

Algorithm 1 Mapping task

```
procedure MAPPING_TASK
    minRange ← 100                                          ▷ Minimum IR sensor range [mm]
    maxRange ← 800                                          ▷ Maximum IR sensor range [mm]

    ϵ ← radius for cluster expansion from point for DBSCAN [mm]    ▷ DBSCAN parameters
    minPts ← minimum number of points for DBSCAN cluster

    T ← distance from middle point to projection point on line between endpoints for IEPF ▷ IEPF parameter

    α ← Angle threshold for line merging [rad]              ▷ Line merging parameters
    β ← Distance threshold for normal line [mm]
    ζ ← Distance threshold between point to line segment [mm]

    PB1 ← []                                                ▷ Initialize point buffers
    PB2 ← []
    PB3 ← []
    PB4 ← []
    PB ← [PB1, PB2, PB3, PB4]
    LB ← []                                                 ▷ Initialize line buffer
    while True do
        if received measurement from sensor tower task then
            updatePointBuffers(PB, measurement, minRange, maxRange)    ▷ Transform range-bearing to {G}, and append valid points to respective point buffer
        end if
        if obtained mutex then                              ▷ Trigger line extraction
            for each point buffer i do
                clusters ← DBSCAN(PB[i], ϵ, minPts, euclidean)    ▷ DBSCAN clustering
                lineClusters ← []                           ▷ initialize output for IEPF
                IEPF(clusters, lineClusters, T)             ▷ IEPF clustering
                lines ← MSE_line_fit(lineClusters)          ▷ MSE line fitting
                Append lines to LB
            end for
            merge_linebuffer(LB, α, β, ζ)
            for each line in LB do
                Compute covariance matrix according to eq. 19
            end for
            Clear PB and LB
            release mutex
        end if                  ▷ Line segments and their uncertainty are now stored in LB
    end while
end procedure
```

9.3.1 Data collection and preprocessing

Similarly to other line extraction implementations with sparse and noisy range-bearing measurements[7][12][24][53], enough measurements have to be collected before it would be beneficial to start the line extraction process. The approach of sampling measurements over multiple robot poses solves the problem of sparsity of exteroceptive measurements, and reduces the impact of noise in the line extraction process.

The mapping task receives four IR measurements (one from each sensor) along with the current estimate of the robot's pose from the sensor tower task. The reason behind collecting both pose

and IR measurements from the sensor tower task was for fast development given the pre-existing code base. In previous implementations of the sensor tower task, the task was responsible for also sending the raw IR sensor measurements to the server along with the current pose estimates from the EKF of the estimator task. I.e. synchronization primitives were already implemented in order to correctly line up pose estimates from the estimator task with the IR sensor readings from the sensor tower task. This makes it easy to send measurements from each of the four IR sensors with synchronized pose estimates to the mapping task.

The FreeRTOS queue management API [6, p. 102-147] was used to implement inter-task communication between the sensor tower task and the mapping task. The data structure which is sent from the sensor tower task with the FreeRTOS API function `xQueueSendToBack` is given in listing 6. Each element in the queue is queued by copy (not by reference) in order to avoid the sensor tower task overwriting data as writing and then reading from the queue does not necessarily happen sequentially.

```
typedef struct ir_measurement {
  uint8_t servo_angle;
  uint16_t measurements[4];
  float x;
  float y;
  float theta
} ir_measurement_t;
```

Listing 6: Data structure sent from sensor tower task to mapping task

On the receiving side of the queue, the mapping task will extract the range-bearing measurements from each received `ir_measurement_t` and transform the measurement given in the robot's sensor frame to the the global frame using (1) and (2). (1) transforms the IR sensor range-bearing measurement from the sensor frame $\{S_i\}$ to the robot's body frame $\{B\}$, d_i denotes the range, s is length from the origin of the $\{B\}$-frame to the origin of the $\{S_i\}$-frame,and β_i is the rotation of the $\{S_i\}$-frame relative to the $\{B\}$-frame. For radially placed IR sensors we have that β_i = `servo_angle` $\times i\frac{\pi}{2}+$ `theta` for IR sensor index $i = [0, ..., 3]$, where `servo_angle` rotates with respect to the body frame of the robot in the interval $[0, \frac{\pi}{2}]$. Δx_i and Δy_i is the x-axis and y-axis translations from the origin of sensor frame $\{S_i\}$ to the origin of the robot's body frame $\{B\}$ for sensor index i.

$$\mathbf{P^B} = \begin{bmatrix} x^B \\ y^B \end{bmatrix} = \begin{bmatrix} (d_i + s)\cos(\beta_i) + \Delta x_i \\ d_i + s)\sin(\beta_i) + \Delta y_i \end{bmatrix}, \; i = [0, \; ..., \; 3] \tag{1}$$

After transforming the point from $\{S_i\}$ to $\{B\}$, we transform the point to the global Cartesian frame $\{G\}$ by applying the transformation given in (2), where (x, y) is the position and θ is the heading of the robot with respect to $\{G\}$. It is necessary to do the line segment extraction computations in the global frame of reference since the robot samples range-bearing measurements over multiple poses.

$$\mathbf{P^G} = \begin{bmatrix} x^B\cos(\theta) - y^B\sin(\theta) + x \\ x^B\sin(\theta) + y^B\cos(\theta) + y \end{bmatrix} \tag{2}$$

Only measurements that are within the valid measurement range of the IR sensor ($100 < d_i < 800)[mm]$ are appended to a fixed-size point buffer. Each IR sensor is allocated one fixed-size point buffer each for buffering of measurements until the conditions for starting line extraction are satisfied. Storing point measurements in separate buffers (one for each IR sensor) will make segmentation easier in the following steps of the line extraction task.

9.3.2 Triggering line segment extraction

The triggering of the line segment extraction process, i.e. starting step 2 (DBSCAN) of figure 11 is one of the responsibilities of the sensor tower task. In order for line segments to be extracted

successfully it is important that the triggering of the process happens at the right moment. The reason for this is that the line segment extraction algorithm expects points of the environment to be buffered sorted in order of the point's bearing. One could of course initially perform a sorting algorithm on the buffered points before proceeding to the feature extraction computations. However, in an effort to speed up the extraction process we would rather exploit that we know the servo angle of the sensor tower and the order of placement of the IR sensors. Thus, as long as the robot does not maneuver in such a manner that changes the heading of the robot whilst sampling range-bearing measurements we know that the buffered points will be stored in order by their bearing. It does not matter in which bearing-direction they are stored, only that they are sorted. The requirement for having the points sorted by bearing is to be able to easily find endpoints of line segments. For a set of sorted points measured by one IR sensor we know that the first point placed in the buffer and the last point placed in the buffer correspond to endpoints of line segments. This should not be understood as the endpoints of the final extracted line segments, but as the endpoints of a set of points which might form multiple line segments in the view of one IR sensor as illustrated in figure 12.

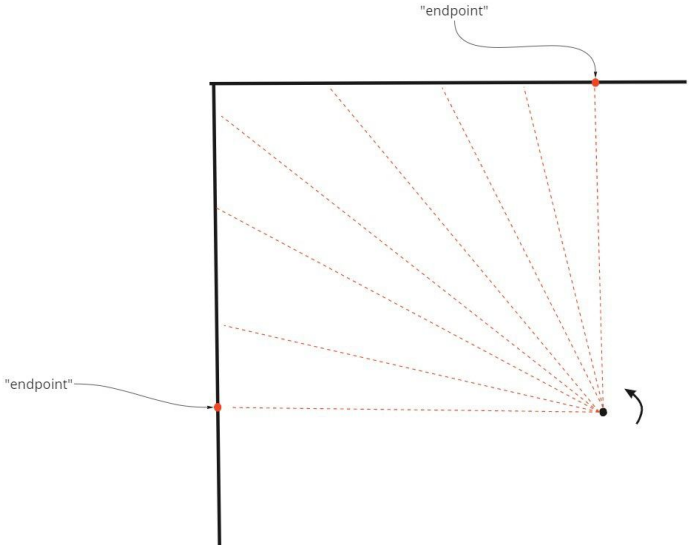


Figure 12: *Endpoints* as seen from the view of one rotating IR sensor

For triggering the extraction process a FreeRTOS mutex (mutual exclusion) object must be released by the sensor tower task. The mapping task must hold the mutex in order to proceed from buffering points to extracting line segments. Whilst the mapping task performs the computations for extracting line segments the sensor tower task should not be permitted to queue more range-bearing measurements, therefore, the sensor tower task will not have access to the inter-task queue before holding the mutex again. This is required in order to synchronize between the sets of sorted buffered points. Albeit, most of the time triggering the extraction process would not require synchronizing access to the queue since the extraction algorithm generally will always finish before the next set of sorted points are placed on the queue due to the motion characteristics of the robot - it either moves in a straight line, stays idle, or rotates in place without quick transitions from one state to the other.

The sensor tower task releases the mutex in order for line segment extraction to commence when the robot transitions from one *move-state* to the other, i.e `moveStop`, `moveClockwise`, `moveCounterClockwise` or `moveForward`. The sensor tower task will also release the mutex when changing the direction of rotation of the sensor tower whilst the robot is in the `moveStop` state. The mapping task may also initiate the extraction process itself when either one of its point buffers are full, which typically happens as the robot is moving straight forward over a longer distance. The only difference here from the triggering of the extraction process in [5] is the use of the mutex instead of the direct-to-task notification.

9.3.3 Density Based Spatial Clustering of Applications with Noise

The second step of the line extraction algorithm is to perform the density based clustering algorithm known as DBSCAN once for each of the four IR sensor point buffers. DBSCAN attempts to cluster measured points of the environment in such a way that each cluster of points distinguish separate partial or full contours of an object. Furthermore, DBSCAN may be used to classify certain points as noise, these points are therefore removed from further processing in the line extraction algorithm. In other words, DBSCAN can be considered a breakpoint detector with the additional advantage of filtering out the most noisy point measurements.

DBSCAN uses two parameters to classify clusters of points, ϵ and `minPts`. The threshold `minPts` denotes the minimum number of points within the ϵ-neighborhood points. The distance measure between points may also be considered a parameter, in our case the euclidean distance was used. If the number of points within the ϵ-neighborhood of a point is larger than `minPts`, then that point is considered to be a *core point*. All points within the ϵ-neighborhood are labeled to belong to the same cluster. Expanding the cluster, i.e. searching for more core points is carried out by querying the distance to neighbors of the neighbors of the point as seen in the pseudo code given in the `expandCluster` procedure in algorithm 2. Again, if the queried point has a sufficient amount of neighboring points within the radius of ϵ, then these points also belong to the same cluster. Eventually during the expansion of a cluster, DBSCAN will reach a point where the number of neighbors is less than `minPts`, these types of points are referred to as *border points*. Border points also belong to the set of points of the expanding cluster, however, we do not continue the recursion of expanding the cluster from these points. Any points that are neither core points or border points are classified as noise since these points have less than `minPts` in their ϵ-neighborhood. [14] [40]

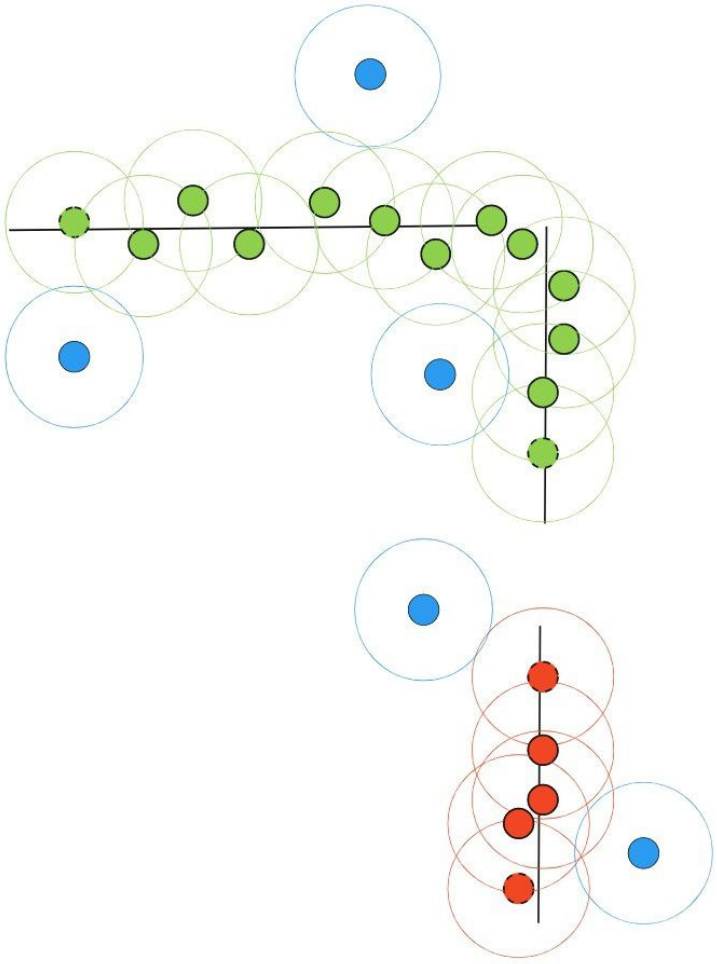


Figure 13: DBSCAN cluster model

Figure 13 illustrates principally how DBSCAN clusters a set of points from an IR scan of an environment with multiple line segments. In figure 13 DBSCAN has found two clusters marked with red and green points. The circles around each point illustrates the ϵ-parameter. Points within these clusters with solid borders are core points, and points with dotted borders are border points. The blue points in figure 13 are classified as noise.

9.3.4 Iterative Endpoint Filter based clustering

After clustering the points of an IR sensor's point buffer using DBSCAN, the next step in the line extraction scheme is to perform iterative endpoint filter based clustering on each DBSCAN cluster. The goal of IEPF based clustering is to cluster the input points into clusters that form straight line segments.

The IEPF based clustering method works by first finding the endpoints of set of input points. Since the points were already placed in sorted order by bearing in the fixed-size point buffer from step one of the line extraction algorithm (and is maintained in order after clustering with DBSCAN) there is no need to sort the points as is done in [12]. I.e. the endpoints of the set of points will always be the first and last point of the input buffer. The IEPF based clustering method proceeds to find the middle point between the endpoints in the input buffer. Further on, the IEPF algorithm finds the orthogonal projection of the middle point on the line between the endpoints. If the euclidean distance between the projected point and the middle point is less than a given threshold distance T, then, all points of the input buffer belong to the same line segment. Otherwise, the IEPF algorithm splits the input buffer in two separate buffers, the first buffer holding all points up until and including the split point, and the second buffer holding all points after and including the split point as seen in lines 15-18 in algorithm 3. The IEPF algorithm is then recursively performed on both of the new buffers. This is why the IEPF algorithm is commonly referred to as a *split and merge* approach to finding clusters of points forming line segments. The recursion along one branch haults when either the distance between the orthogonal projection of the middle point is less than T, or the length of the input buffer is ≤ 2. The IEPF algorithm then does the same for the other branch (second buffer).

Figure 14 illustrates the principal of the IEPF algorithm and how it manages to cluster the points of the buffer `point_buffer` into two line segments. Green points belong to one IEPF cluster, whilst blue points belong to the other cluster. In the case of figure 14 only one split was necessary in order to satisfy the conditions for terminating the IEPF recursion since the distance from point the split point M to its projection on the line between the endpoints P was found to be less than T.

9.3.5 Line fitting

The third step of the line extraction scheme computes estimates for the line parameters of each cluster of points comprising individual line segments found by the IEPF algorithm in the last step. The line fitting procedure is described by [44], and is here repeated in the following paragraphs.

The standard (Cartesian) model of a straight line $y = ax + b$ (a representing the slope of the line and b the y-axis intercept), is problematic to use when dealing with vertical lines. Therefore, the *Hesse normal form* stated in (3) is a more suitable choice of line model to avoid dealing with infinite slopes. Figure 15 illustrates r as the line with the shortest distance between the origin of the Cartesian coordinate system to the line we are trying to estimate the parameters of, and θ as the angle between the x-axis and r.

$$r = x\cos(\theta) + y\sin(\theta) \tag{3}$$

In order to find the Hesse normal form parameters of a line (r, θ) that fit the measured points of the environment a weighted mean squared error (MSE) line fitting procedure is employed. The

Algorithm 2 DBSCAN

```
procedure DBSCAN(pointBuffer, ϵ, numPoints, distFunc)        ▷ Density-based Spatial Clustering for Applications with Noise
    n ← 0                                                    ▷ Initialize number of clusters
    l ← length of pointBuffer
    i ← 0                                                    ▷ pointBuffer index
    while i < l do
        if point == UNDEFINED then
            neighbors = getNeighbors(pointBuffer[i], pointBuffer, ϵ, distFunc)
            if length of neighbors < minPts then
                point ← NOISE
            else
                n ← n + 1                                    ▷ Found a valid cluster
                pointBuffer[i] ← n                           ▷ Label point with the cluster it belongs to
                expandCluster(n, neighbors, pointBuffer, distFunc, ϵ, minPts)
            end if
        end if
        i ← i + 1
    end while
    return pointBuffer                                       ▷ Points labeled by which cluster they belong to or noise
end procedure
```

```
procedure getNeigbors(point, pointBuffer, ϵ, distFunc)
    i ← 0
    l ← length of pointbuffer
    neighbors ← []                                           ▷ Initialize list for points that are neighbors to point
    while i < l do
        dist ← distFunc(point, pointBuffer[i])               ▷ For this implementation the distance measure used is the euclidean distance
        if dist ≤ ϵ then
            Append point to neighbors
        end if
        i ← i + 1
    end while
    return neighbors
end procedure
```

```
procedure expandCluster(numClusters, neighbors, pointBuffer, distFunc, ϵ, minPts)
    i ← 0
    l ← length of neighbors
    while i < l do
        if neighbors[i] == UNDEFINED then
            expandedNeighbors ← getNeighbors(neighbors[i], pointBuffer, ϵ, distFunc)
            if length of neighbors < minPts then
                Append expandedNeighbors to neighbors
            end if
        end if
        if neighbors[i] == UNDEFINED or neighbors[i] == NOISE then
            neighbors[i] ← numClusters                       ▷ Label point with cluster number
        end if
        i ← i + 1
    end while
end procedure
```

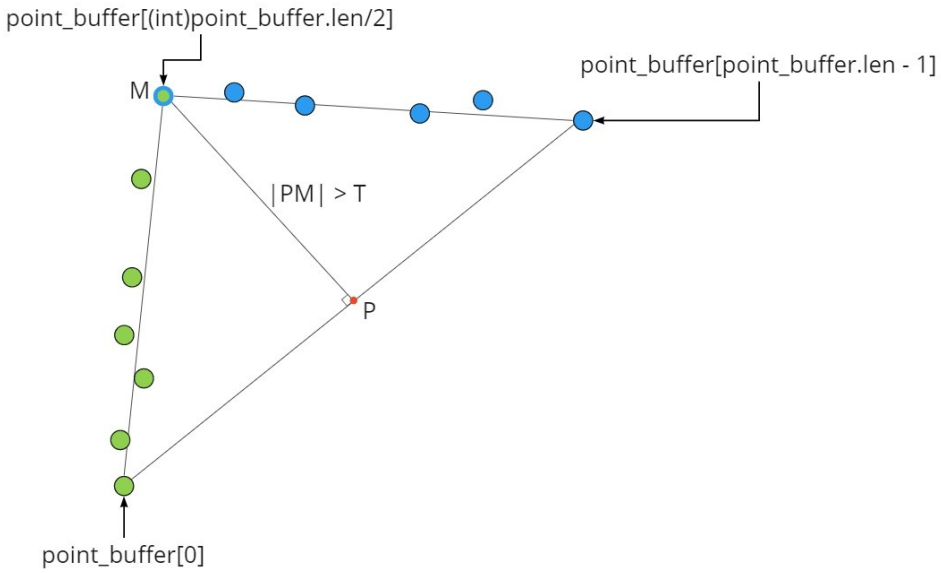


Figure 14: Illustration of IEPF based clustering

Algorithm 3 IEPF

procedure IEPF(`inputPoints`, `outputPoints`, T) ▷ `outputPoints` is initialized as an empty list
    `n` ← length of inputPoints
    if `n` == 2 **then**
        `outputPoints` ← `inputPoints`
    else
        `A` ← `inputPoints[0]` ▷ First endpoint
        `middleIndex` ← `floor(n / 2)`
        `B` ← `inputPoints[middleIndex]` ▷ Middle point
        `C` ← `inputPoints[n-1]` ▷ Last endpoint
        `AC` ← line segment from `A` to `C`
        `projectedPoint` ← `getProjectedPointOnLine(AC, B)`
        if |`AC`| $< T$ **then**
            Append `inputPoints` to `outputPoints`
        else ▷ Perform split
            `leftPoints` ← all points in `inputPoints` up to index `n`
            `IEPF(leftPoints, outputPoints, T)`
            `rightPoints` ← all points in `inputPoints` from `n`
            `IEPF(rightPoints, outputPoints, T)`
        end if
    end if
end procedure ▷ `outputPoints` contains lists of points that belong to the same cluster

perpendicular distance from a measured point (x_i, y_i) to the fitted line is given by (4)

$$\rho_i = x_i \cos(\theta) + y_i \sin(\theta) - r \tag{4}$$

The estimated line parameters (r, θ) that minimize the perpendicular distance from of all measured points of a cluster may be found by minimizing the weighted mean squared error (MSE) defined

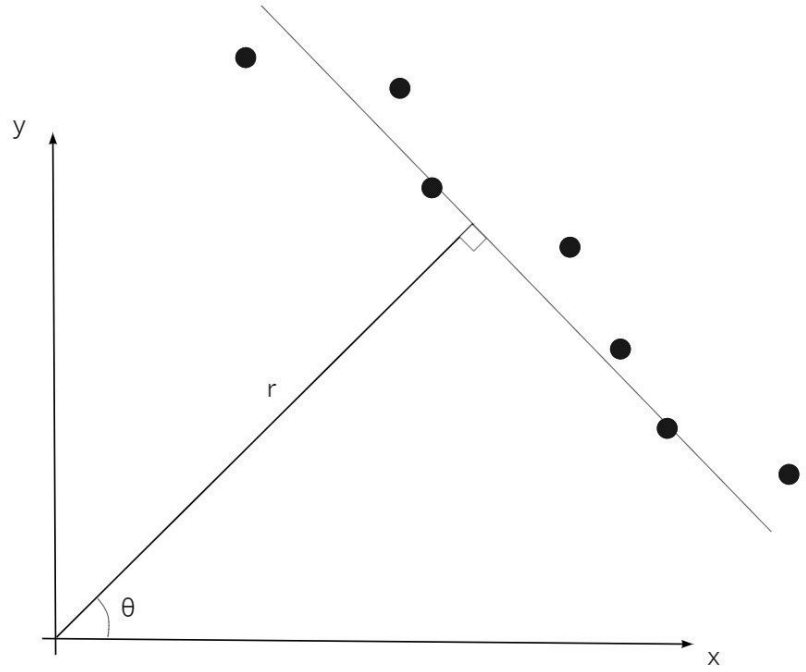


Figure 15: Illustration of parameters for a line on Hesse normal form

by (5). The scaling values s_i are used in order to account for the reliability of a point i. Note that for the current implementation for the robot the scaling values are all set to 1, thus this is not a weighted line fitting scheme at the time being. However, we will later discuss how one may want to employ the scaling factors for making the line extraction algorithm more accurate.

$$\text{MSE}(r, \theta) = \sum_{i=1}^{N} (s_i \rho_i)^2 \tag{5}$$

An analytical expression for the optimum values of the line parameters for a given set of points may be found by computing the partial derivatives of r and θ, and solving the normal equations. This gives the estimators for r and θ given in (6) and (7).

$$\hat{r} = \frac{\cos(\hat{\theta})}{N} \sum_{i=1}^{N} x_i + \frac{\sin(\hat{\theta})}{N} \sum_{j=1}^{N} y_j \tag{6}$$

$$\hat{\theta} = \frac{1}{2} \text{atan2}(-2\sigma_{xy}, \sigma_y^2 - \sigma_x^2) \tag{7}$$

The variances σ_x^2 and σ_y^2 and covariance σ_{xy} used in (7) are given in (8)-(10), where the weights w_i are computed from the scaling factors s_i using (11) and then normalized.

$$\sigma_x^2 = \frac{1}{N} \sum_{i=1}^{N} w_i (x_i - \frac{1}{N} \sum_{j=1}^{N} x_j)^2 \tag{8}$$

$$\sigma_y^2 = \frac{1}{N} \sum_{i=1}^{N} w_i (y_i - \frac{1}{N} \sum_{j=1}^{N} y_j)^2 \tag{9}$$

$$\sigma_{xy} = \frac{1}{N} \sum_{i=1}^{N} w_i (x_i - \frac{1}{N} \sum_{i=j}^{N} x_j)(y_i - \frac{1}{N} \sum_{j=1}^{N} y_j) \tag{10}$$

$$w_i = \frac{s_i^2}{\frac{1}{N}\sum_{i=1}^{N} s_i^2} \tag{11}$$

After computing the MSE fit parameters from (6) and (7), the orthogonal projections of the endpoints on the fitted line are used as the endpoints of the line segment. Projecting the endpoints onto the fitted line parametrized by r and θ is illustrated in figure 16. The red point shows point P's projection onto the fitted line which may be computed by (12)-(14), and depending on the sign of $r_p - r$, one may either subtract or add dx and dy to the coordinates of P to find its projection.

$$r_p = x\cos\theta + y\sin(\theta) \tag{12}$$

$$dx = |r_p - r|\cos\theta \tag{13}$$

$$dy = |r_p - r|\sin\theta \tag{14}$$

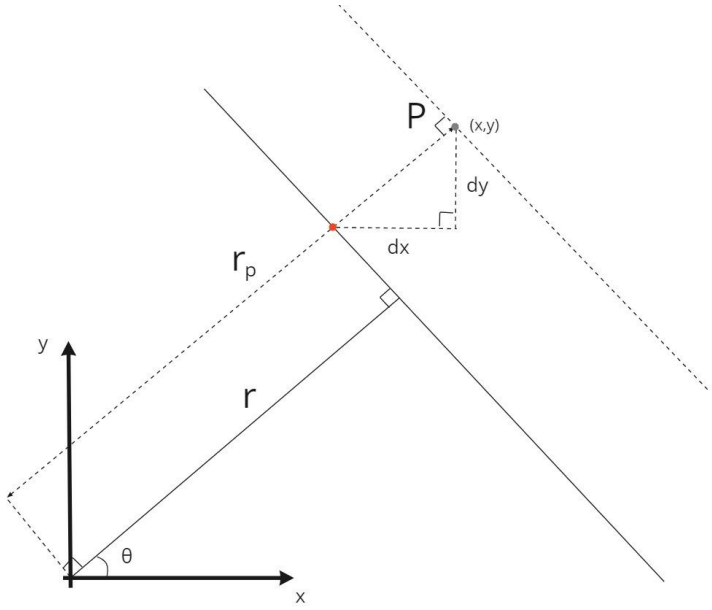


Figure 16: Orthogonally project endpoints on to fitted line parametrized by r and θ

9.3.6 Line merging

At this point in the line segment detection scheme, initial line segments are finally found from the noisy IR sensors. However, the line segments are only derived from points from the same point buffer. Line segments derived from different point buffers may be fragments of the same line segment. Therefore, line segments derived from different point buffers that should be represented by one line segment are merged together in the fifth step of the line extraction procedure.

After having found initial line segments derived from points from each individual point buffer, line segments are gathered into a common line buffer. The line merging procedure will perform data association on neighbouring line segments, and if two line segments are deemed *mergeable*, the line segments are merged. Whenever a merge occurs the same procedure is carried out on the merged line segment for the next stored line in the line buffer. Thus, also in this step we exploit that we know how line segments are placed in relation to each other due to the bearing ordering of the points forming the line segments. The recursive merging of line segments terminates when none of the neighbouring line segments of the buffer meet the conditions for merging, or there is only one line segment in the buffer.

In order for two lines segments to be merged they must satisfy the conditions given in (15)-(17). Here r_i, θ_i denotes the normal form parameters of the line passing through the endpoints (P_i, Q_i) of line segment l_i. In (15) the function $\texttt{ssa}(\theta_1, \theta_2)$ given in listing 7 returns the smallest signed angle between θ_1 and θ_2. It is necessary to find the smallest signed angle between the normal form angle parameters in order to avoid not being able to detect that a line segment with for instance $\theta_1 \approx 0$ and another line segment with $\theta \approx 2\pi$ have almost the same orientation relative to each other. Note that since the C-language modulo operator, `%`, returns the same sign as the dividend a custom modulo function must be used instead [15, p. 388]. The function $\texttt{dist\_from\_point\_to\_line\_segment}(p_i, l_j)$ given in listing 8 returns the shortest distance between an endpoint of a line segment to a point on the line of the other line segment. α, β and ζ are constant threshold parameters.

$$|\texttt{ssa}(\theta_1, \theta_2)| < \alpha \tag{15}$$

$$|r_1 - r2| < \beta \tag{16}$$

$$\texttt{dist\_from\_point\_to\_line\_segment}(p_1, l_2) < \zeta \tag{17}$$

```
float mod(float a, float b) {
    return a - floorf(a / b)*b;
}

float ssa(float rad) {
  return mod(rad + M_PI, 2*M_PI) - M_PI;
}
```

Listing 7: Smallest signed angle

```

typedef struct point {
    // Cartesian coordinates
    float x;
    float y;
} point_t;

typedef struct line {
    // Endpoints of line segment
    point_t P;
    point_t Q
}line_t;

// Returns shortest distance from a point to a line passing through two points
float distance_from_point_to_line(point_t point, line_t line) {
    return fabs((line.Q.x - line.P.x)*(line.P.y - point.y) - (line.P.x - point.x)*(
    line.Q.y - line.P.y))/get_length(line);
}

// Returns shortest distance from a point to a line segment
float distance_from_point_to_line_segment(point_t point, line_t line) {
    float t = -((line.P.x - point.x)*(line.Q.x - line.P.x) + (line.P.y - point.y)*(
    line.Q.y - line.P.y)) / (pow(get_length(line), 2));
    if (t >= 0 && t <= 1) {
        // The point is perpendicular to the line segment
        return distance_from_point_to_line(point, line);
    }
    float d1 = get_length((line_t){.P = point, .Q = line.P});
    float d2 = get_length((line_t){.P = point, .Q = line.Q});
    if (d1 < d2) {
        return d1;
    }
    return d2;
}
```

Listing 8: Distance from point to line segment

During line extraction merging the points forming a particular line segment are still readily available. The points are not removed until after the line extraction process (over one scan) has finished. Therefore, whenever two lines have been found to satisfy the criteria for merging, the merge procedure of the line extraction process will join the buffers holding the points of the two line segments. This is followed by computing the MSE line fitting parameters again for the union of points of the two mergeable line segments. Note that for the implementation for the nRF52840 robot given in listing 9 it was necessary to rewrite the recursive (calling the function inside the function itself) merge procedure to do the computations iteratively in order to prevent the function from consuming too much stack memory.

```
void merge_linebuffer(line_segment_buffer_t* lb, float alpha, float beta, float
    zeta) {
    int i = 0;
    while (i != lb->len-1 && lb->len > 1) {
        for (i=0; i<lb->len-1; i++) {
            line_segment_t line = lb->buffer[i];
            line_segment_t nextLine = lb->buffer[i+1];
            if (is_mergeable(line, nextLine, alpha, beta, zeta)) {
                // Line and nextLine satisfy eq. (15) - (17)

                line_segment_t joint_line_segment;
                // Perform MSE line fitting on points from line and nextLine
                join_line_segments(&joint_line_segment, line, nextLine);

                // Move lines of line buffer up one index (overwriting line at
    index i)
                lb->buffer[i] = joint_line_segment;
                for (int j=i+1; j<lb->len-1; j++) {
                    lb->buffer[j] = lb->bufffer[j+1];
                }
                lb->len -= 1; // decrease length of line buffer

            }

        }
        if (i == lb->len-1 && lb->len > 1) {
            // Check if last line in buffer is mergeable with first line (wrap
    around)
            line_segment_t line = lb->buffer[0];
            line_segment_t line = lb->buffer[lb->len-1];
            if (is_mergeable(line, nextLine, alpha, beta, zeta)) {
                join_line_segments(&joint_line_segment, line, nextLine);
                 // Perform MSE line fitting on points from line and nextLine
                join_line_segments(&joint_line_segment, line, nextLine);

            }
            lb->buffer[0] = joint_line_segment;

        }


    }

}
```

Listing 9: Merge lines of line segment buffer

9.3.7 Line uncertainty estimation

In order to use detected line segments in the EKF framwork for sensor fusion with odometry and inertial measurements of the robot, an error model for the detected lines has to be established. Specifically, we are interested in the finding estimates of the errors of r and θ, and how these correlate. I.e. the covariance matrix of the line parameters given in (18) must be found.

$$R_{r\theta} = \begin{bmatrix} \sigma_r^2 & \sigma_{r\theta} \\ \sigma_{r\theta} & \sigma_\theta^2 \end{bmatrix} \tag{18}$$

The simplified closed-form error model of a straight line proposed by Sommer (2018)[44] was chosen to be used for line uncertainty estimation in this thesis. Sommer claims that using the estimated covariance matrix for the line parameters r and θ given in (19) gives a sufficiently accurate estimate of (18) for most applications. Furthermore, computing the estimated covariance matrix is fast, and does not suffer from numerical instability such as the analytically derived exact solution.

$$R_{r\theta} \approx \sigma_\rho^2 \begin{bmatrix} \frac{12x_{off}^2}{L^2 N} + \frac{1}{N} & \frac{-12x_{off}}{L^2 N} \\ \frac{-12x_{off}}{L^2 N} & \frac{12}{L^2 N} \end{bmatrix} \tag{19}$$

Following the derivation from [44], the elements of the estimated covariance matrix (19) are parametrized by the variance of the distance between a sampled point and the fitted line σ_ρ^2, the number of sampled points in the line segment cluster N, the length of the fitted line segment L and the geometric offset x_{off} illustrated in figure 17. x_{off} and σ_ρ^2 are computed using (20) and (21).

$$x_{off} = \frac{r}{2}(\tan(\theta - \theta_N) + \tan(\theta - \theta_1)) \tag{20}$$

$$\sigma_\rho^2 = \sum_{i=1}^{N} s_i \rho_i(r, \theta)^2 \tag{21}$$

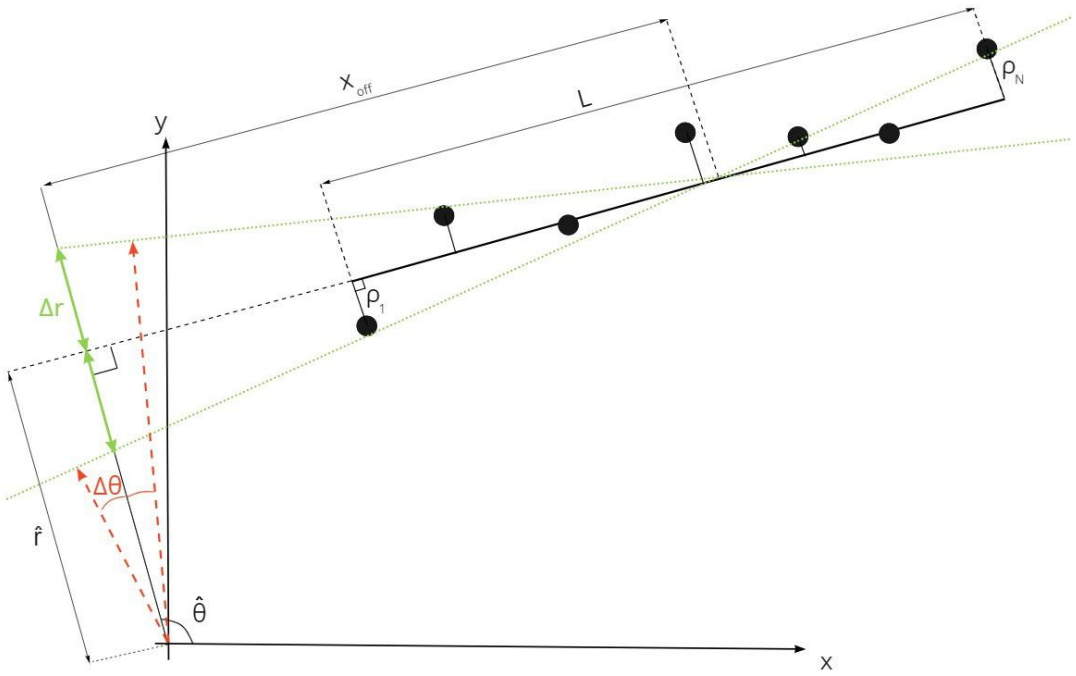


Figure 17: Illustration of parameters of the estimated closed-form error model. Adapted from [44]

9.4 EKF-SLAM

The following sections will derive the filter equations used for the line segment based EKF-SLAM implementation, in addition to presenting how the SLAM approach handles updates, initializing new line segments, predictions and map management. The whole filter algorithm is outlined in pseudo code in algorithm 4. Most derivations of the filter equations follow the *recipe* provided by Brekke (2020) [9], Fossen (2021) [15] and Solà (2007) [38] applied to models specific for the characteristics of the robot in-use.

9.4.1 Motion model

In order to derive the filter equations used for the EKF-SLAM implementation we have to obtain a suitable kinematics model for the differential drive robot. Two-wheeled robots such as the nRF52840-based robot are capable of moving forwards by rotating both wheels at the same speed, and turn by rotating one of the wheels at a higher speed than the other. However, they are not able to suddenly move laterally. This is known as a non-holonomic system, and the kinematics describing these types of robots' movement may be captured by the *unicycle model* [10][2]. The unicycle model was also used by [19] when designing the EKF for the nRF52832-robot.

For the unicycle model the pose vector of the robot given in (22) contains the x- and y-coordinates of the robot, and θ is the heading of the robot, with respect to a two-dimensional global reference frame. The origin of the global reference frame is set to the initial pose of the robot.

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{22}$$

Using the unicycle model the pose of the robot evolves over time by the non-linear differential equation given in (23), where v is the linear velocity and ω is the rotational velocity of the robot. The process noise (24) is modeled as an additive zero-mean Gaussian term $\boldsymbol{w}$, with a diagonal covariance matrix $\boldsymbol{Q}$.

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \omega \end{bmatrix} + \boldsymbol{w} \tag{23}$$

$$\boldsymbol{w} \sim \boldsymbol{\mathcal{N}}(\boldsymbol{0}, \boldsymbol{Q}) \tag{24}$$

The linear velocity v and rotational velocity ω are considered as control inputs to the system, thus we may define the control vector given in (25). Following the lines of [19], v may be computed from counting the number of encoder ticks between each cycle of the filter. w can be extracted from the gyroscope. It was decided not to use the accelerometer of the IMU due to its high noise level as explained in [19]. The variance elements of $\boldsymbol{Q}$ must be set appropriately in order to capture the uncertainty from the encoder and gyroscope measurements.

$$\boldsymbol{u} = \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{25}$$

Following Fossen (2021)[15] one may discretize (23) using forward Euler integration and setting the Gaussian white noise term to zero. This results in the discrete-time predictor for the k-th time-step given in (26), where h is the sample period of the estimator.

$$\boldsymbol{x}[k+1] = \boldsymbol{f}_{\boldsymbol{x}}(\boldsymbol{x}[k], \boldsymbol{u}[k]) = \begin{bmatrix} x[k] + hv[k]\cos\theta[k] \\ y[k] + hv[k]\sin\theta[k] \\ \theta[k] + h\omega[k] \end{bmatrix} \tag{26}$$

Algorithm 4 EKF-SLAM

procedure EKF-SLAM
  $\nu \leftarrow \mathbf{0}_{3\times 1}$ ▷ Initialize state vector
  $\mathbf{P} \leftarrow \mathbf{0}_{3\times 3}$ ▷ Initialize state covariance matrix
  $\mathbf{R} \leftarrow$ `[]` ▷ Initialize stacked measurement covariance matrix
  `endpoints` $\leftarrow$ `[]` ▷ Initialize list for endpoints
  $n \leftarrow 0$ ▷ Number of landmarks

  while True **do**
    `tmp` $\leftarrow$ `[]` ▷ Temporary list for storing indexes of updated landmarks
    while new line segment available $\mathbf{y}$, $\mathbf{R}_{r\phi}$, $\mathbf{e}$ **do** ▷ $\mathbf{y} = [r, \phi]^T$, $\mathbf{R}_{r\phi} = [[\sigma_r^2, \sigma_{r\phi}], [\sigma_{\phi r}, \sigma_\phi^2]]$ $\mathbf{e} = [[x_s, y_s]^T, [x_e, y_e]^T]^T$

      `merged` $\leftarrow$ `False`
      `i` $\leftarrow 0$

      while `i` $< n$ **do** ▷ Iterate through map
        `idx` $\leftarrow 3 +$ `i` $* 2$ ▷ Landmark index in ν
        if `isMergeable`($\mathbf{y}$, $\mathbf{e}$, [ν`[idx]`, ν`[idx+1]`], `endpoints[i]`) **then**
          $\nu, \mathbf{P} \leftarrow$ `update`$(\nu, \mathbf{P}, \mathbf{R},$ `idx`$, \mathbf{y}, \mathbf{R}_{r\phi}, \mathbf{e})$ ▷ Perform update sequence
          Append `i` to `tmp` ▷ Keep track of updated line segments

          `p1, p2, p3, p4` $\leftarrow$ Compute projections of points in `endpoints[i]` and $\mathbf{e}$ onto updated line

          `p_s, p_e` $\leftarrow$ Find the pair of points among (`p1, p2, p3, p4`) that form the longest line segment

          `endpoints[i]` $\leftarrow$ `[p_s, p_e]` ▷ Update endpoints corresponding to updated line
          `merged` $\leftarrow$ `True`
        end if
        `i` $\leftarrow$ `i` $+ 1$
      end while

      if not `merged` **then** ▷ New line segment was not mergeable with any stored line segments
        add_new_landmark(ν, $\mathbf{P}$, $\mathbf{R}$, $\mathbf{y}$, $\mathbf{R}_{r\phi}$) ▷ Append $\mathbf{y}$ to ν, and extend $\mathbf{P}$
        $n = n + 1$
      end if
    end while

    `mapManagement`(`tmp`, ν, $\mathbf{P}$, $\mathbf{R}$, `endpoints`) ▷ Perform update sequence again for all mergeable line segments that were updated during this iteration, and remove redundant features.
    `tmp` $\leftarrow$ `[]` ▷ Clear `tmp`

    $u \leftarrow$ `[linear velocity, rotational velocity]`
    `dt` $\leftarrow$ delta time
    $\nu, \mathbf{P} \leftarrow$ `predict`$(\nu,$ `P,` $u,$ `dt,` $n)$ ▷ Predict state for next time step

  end while
end procedure

9.4.2 Map

The map of the robot's environment is comprised of n line features parametrized using the Hesse normal form. (27) represents line i extracted from the points of the environment collected by the IR sensors. r_i is the perpendicular distance from the origin of the global reference frame to line i, and ϕ_i is the angle from the x-axis of the global reference frame to the line spanning from the origin to line i.

$$\boldsymbol{l_i} = \begin{bmatrix} r_i \\ \phi_i \end{bmatrix} \tag{27}$$

Thus, one may represent the entire map of the environment with (28). Note that in this EKF-SLAM solution only the line parameters r_i and ϕ_i are used in the estimation problem. In order to form the line segments we obviously also need the endpoints of the line. Although the endpoints of the line segment are not part of the state estimation they must be stored and used for data association and line segment merging.

$$\boldsymbol{m} = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_n \end{bmatrix} \tag{28}$$

It is reasonable to assume that line-features are both stationary and permanent. Thus, there is no need to model the kinematics of the line features forming the map, and the map prediction may be simply written as given in (29).

$$\boldsymbol{m}[k+1] = \boldsymbol{m}[k] \tag{29}$$

9.4.3 Measurement model for a line feature

In order to exploit correlations between observed line features and the robot's pose we need find a relationship between the line features and the pose. This is where the measurement model and its inverse come in to play.

We may define the measurement model for a line i at time-step k as given in (30), similarly as used in Garulli et al. (2005)[16]. The measurement prediction $\boldsymbol{h}_i(\boldsymbol{x}[k], \boldsymbol{l}_i)$ is the state-to-measurement mapping, i.e. it transforms line features in the state vector to observations of line features from the point of view of a robot with the given input pose as illustrated in figure 18. Similarly, the inverse measurement prediction for a line i given in (31) yields the measurement-to-state mapping.

$$\boldsymbol{y}[k] = \boldsymbol{h}_i(\boldsymbol{x}[k], \boldsymbol{l}_i) + \boldsymbol{\epsilon}_i[k] = \begin{bmatrix} r_i^B + \epsilon_{r,i}[k] \\ \phi_i^B + \epsilon_{\phi,i}[k] \end{bmatrix} = \begin{bmatrix} r_i^G - (x[k]\cos\phi_i^G + y[k]\sin\phi_i^G) + \epsilon_{r,i}[k] \\ \phi_i^G - \theta[k] + \epsilon_{\phi,i}[k] \end{bmatrix} \tag{30}$$

$$\boldsymbol{g}(\boldsymbol{x}[k], \boldsymbol{l}_i) = \begin{bmatrix} r_i^G \\ \phi_i^G \end{bmatrix} = \begin{bmatrix} r_i^B + x\cos(\phi_i^B + \theta) + y\sin(\phi_i^B + \theta) \\ \phi_i^B + \theta \end{bmatrix} \tag{31}$$

Stacking each measurement model of a line into the same vector gives the full measurement model in (32).

$$\boldsymbol{y}[k] = \boldsymbol{h}(\boldsymbol{x}[k], \boldsymbol{m}) + \boldsymbol{\epsilon}[k] = \begin{bmatrix} \boldsymbol{h}_1(\boldsymbol{x}[k], \boldsymbol{l}_1) \\ \vdots \\ \boldsymbol{h}_n(\boldsymbol{x}[k], \boldsymbol{l}_n) \end{bmatrix} + \boldsymbol{\epsilon}[k] \tag{32}$$

We may assume that each measurement of a line are independent of one another and is Gaussian distributed according to $\boldsymbol{\mathcal{N}(0, R^i_{r\phi})}$. The closed-form line uncertainty estimation technique from

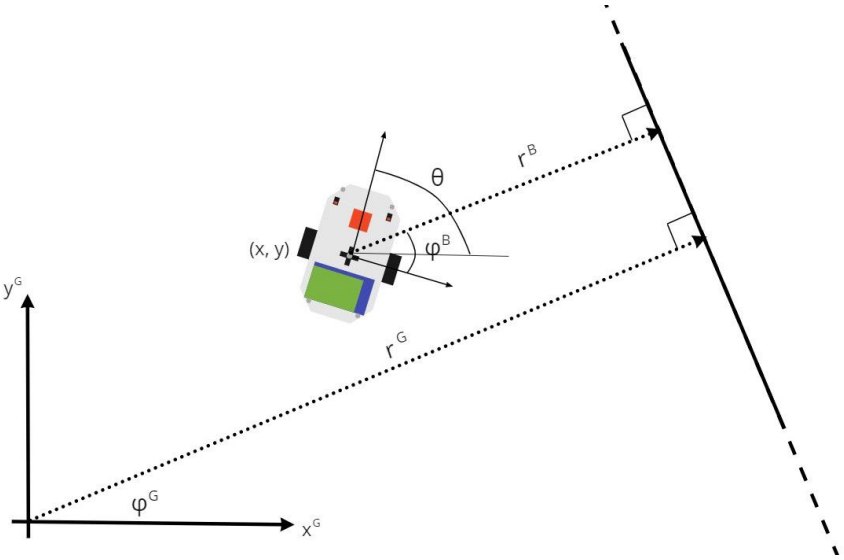


Figure 18: Illustration of parameters for line feature measurement model

section 9.3.7 has been used to find an estimate of the covariance matrix $\boldsymbol{R}^{\boldsymbol{i}}_{\boldsymbol{r\phi}}$ for a line i. Therefore, the covariance matrix of the full (stacked) measurement noise vector $\boldsymbol{\epsilon}[k]$ may be expressed as given in (33).

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}^{\boldsymbol{1}}_{\boldsymbol{r\phi}} & & \\ & \ddots & \\ & & \boldsymbol{R}^{\boldsymbol{n}}_{\boldsymbol{r\phi}} \end{bmatrix} \tag{33}$$

9.4.4 Prediction

In SLAM we wish to estimate both the pose of the robot and the map of the robot's surroundings simultaneously, thus, we define the joint pose-map vector given in (34).

$$\boldsymbol{\nu} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{m} \end{bmatrix} \in \mathbb{R}^{3+2n} \tag{34}$$

Combining the derivations from section 9.4.1 and 9.4.3 one may formulate the joint pose-map discrete prediction given in (36). For clarity we drop indexing with time-step k.

$$\dot{\boldsymbol{\nu}} = \boldsymbol{f}(\boldsymbol{\nu}, \boldsymbol{u}) + \boldsymbol{N}\boldsymbol{w} \tag{35}$$

$$\boldsymbol{\nu} \leftarrow \boldsymbol{f}(\boldsymbol{\nu}, \boldsymbol{u}) = \begin{bmatrix} \boldsymbol{f}_{\boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{u}) \\ \boldsymbol{m} \end{bmatrix} \tag{36}$$

$$\boldsymbol{N} = \begin{bmatrix} \boldsymbol{I}_3 \\ \boldsymbol{0}_{2n\times 3} \end{bmatrix} \tag{37}$$

The state (joint pose-map) covariance matrix is given in (38).

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{P}_{\boldsymbol{xx}} & \boldsymbol{P}_{\boldsymbol{xl}_1} & \dots & \boldsymbol{P}_{\boldsymbol{xl}_n} \\ \boldsymbol{P}_{\boldsymbol{l}_1\boldsymbol{x}} & \boldsymbol{P}_{\boldsymbol{l}_1\boldsymbol{l}_1} & \dots & \boldsymbol{P}_{\boldsymbol{l}_1\boldsymbol{l}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{P}_{\boldsymbol{l}_n\boldsymbol{x}} & \boldsymbol{P}_{\boldsymbol{l}_n\boldsymbol{l}_1} & \dots & \boldsymbol{P}_{\boldsymbol{l}_n\boldsymbol{l}_n} \end{bmatrix} \tag{38}$$

$$\boldsymbol{P_{xx}} = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{y\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_\theta^2 \end{bmatrix} \tag{39}$$

$$\boldsymbol{P_{l_i x}} = \boldsymbol{P_{x l_i}}^T = \begin{bmatrix} \sigma_{r_i x} & \sigma_{r_i y} & \sigma_{r_i \theta} \\ \sigma_{\phi_i x} & \sigma_{\phi_i y} & \sigma_{\phi_i \theta} \end{bmatrix} \tag{40}$$

$$\boldsymbol{P_{l_i l_j}} = \boldsymbol{P_{l_j l_i}}^T = \begin{bmatrix} \sigma_{r_i r_j} & \sigma_{r_i \phi_j} \\ \sigma_{\phi_i r_j} & \sigma_{\phi_i \phi_j} \end{bmatrix} \tag{41}$$

The goal of EKF-SLAM is to maintain estimates of the state vector $\boldsymbol{\nu}$ and the state covariance $\boldsymbol{P}$ in every iteration of the filter. All variants of the Kalman filter have in common that estimates are computed with a prediction step in every iteration. Whenever a measurement is available an update step allows for the filter to correct its estimates. The EKF state prediction is already given in (36). Following Brekke (2020) [9] and Solà (2007) [38], in order to tackle the nonlinearities in the process and measurement model in order to apply the Kalman filter framework, the EKF solution involves finding the Jacobians of the process and measurement models. Thus, computing the Jacobian of the part of the process model including the robot pose with respect to the pose yields (42).

$$\boldsymbol{F_x} = \frac{\partial \boldsymbol{f_x}(\boldsymbol{x}, \boldsymbol{u})}{\partial \boldsymbol{x}} = \begin{bmatrix} 1 & 0 & -hv\sin(\theta) \\ 0 & 1 & hv\cos(\theta) \\ 0 & 0 & 1 \end{bmatrix} \tag{42}$$

To find the Jacobian of $\boldsymbol{h}(\boldsymbol{\nu})$, we may break down the problem by finding the Jacobian of the measurement prediction function of a single line feature $\boldsymbol{h_i}(\boldsymbol{x}, \boldsymbol{l_i})$ with respect to the pose and line feature vector separately.

$$\boldsymbol{H_x^i} = \frac{\partial \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{l_i})}{\partial \boldsymbol{x}} = \begin{bmatrix} -\cos\phi_i & -\sin\phi_i & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{43}$$

$$\boldsymbol{H_l^i} = \frac{\partial \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{l_i})}{\partial \boldsymbol{l^i}} = \begin{bmatrix} 1 & x\sin\phi - y\cos\phi \\ 0 & 1 \end{bmatrix} \tag{44}$$

Followingly, the joint pose-map Jacobians can be defined as given in (45)-(46) [9][38].

$$\boldsymbol{F} = \begin{bmatrix} \boldsymbol{F_x} & \boldsymbol{0} \\ \boldsymbol{0} & I \end{bmatrix} \tag{45}$$

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{H_x}^1 & \boldsymbol{H_l}^1 & & \\ \vdots & & \ddots & \\ \boldsymbol{H_x}^n & & & \boldsymbol{H_l}^n \end{bmatrix} \tag{46}$$

Thus, the state covariance prediction may be formulated according to (47).

$$\boldsymbol{P} \leftarrow \boldsymbol{F}\boldsymbol{P}\boldsymbol{F}^T + \boldsymbol{N}\boldsymbol{Q}\boldsymbol{N}^T \tag{47}$$

```
1
2 def f(nu, u, dt):
3     x = nu[0,0] + dt*u[0]*np.cos(nu[2,0])
4     y = nu[1,0] + dt*u[0]*np.sin(nu[2,0])
5     theta = nu[2,0] + dt*u[1]
6     state =  np.array([
7         [x],
```

```
        [y],
        [theta],
    ])
    if (np.shape(nu)[0] > 3):
        # State vector contains line features
        return np.block([[state],
                         [nu[3:,:]]])
    else:
        # State does currently not contain any line features
        return state

def Fx(nu, u, dt):
    matrix = np.eye(3,3)
    matrix[0,2] = -dt*u[0]*np.sin(nu[2, 0])
    matrix[1,2] = dt*u[0]*np.cos(nu[2, 0])
    return matrix


def F(nu, u, dt):
    n = int((np.shape(nu)[0] - 3) / 2)
    matrix = np.block([[Fx(nu, u, dt), np.zeros((3,n*2))],
                       [np.zeros((n*2,3)), np.eye(n*2,n*2)]])
    return matrix

def N(n):
    return np.block([[np.eye(3,3)],
                     [np.zeros((n*2,3))]])

def predict(nu, P, u, dt, num_landmarks):
    # Predict state
    nu_ = f(nu, u, dt)
     # Predict covariance
    P_ = F(nu, u, dt) @ P @ F(nu, u, dt).T + N(num_landmarks) @ Q @ N(num_landmarks
    ).T
    return nu_, P_
```

Listing 10: EKF-SLAM prediction (python)

9.4.5 Initializing new line segments

As the robot explores its surrounding environment whenever the robot observes a new line segment - one that was not mergeable with any of the exisiting line features of the state vector, it should be added to the map. I.e. the state vector increases its size by two when we add the newly observed line parameterized by r and ϕ. Since the state vector stores line features with respect to the global frame, and the mapping task outputs the extracted lines with respect to the same coordinate system, the *measurement* may be added directly to the state vector. Seperately from the state vector ν, the endpoints corresponding to the line parametrized by r and ϕ are appended to a list. One may find the normal form parameters of endpoint `i` (zero-indexed) in the the state vector at index position `3+2*i`.

Additionally, the state covariance matrix must be appended with the newly extracted line feature's covariance and its cross-variance with the existing line features constituting the map. According to [38], the new line feature's covariance and cross-covariance may be computed from the Jacobian of the inverse measurement model. For our case, the Jacobian of (31) with respect to the robot pose and the measurement may be computed as given in (48)-(49). Note that the measurement $\boldsymbol{y} = \begin{bmatrix} r^G & \phi^G \end{bmatrix}^T$ first has to be converted to the body-frame $\{B\}$ of the robot.

$$\boldsymbol{G_x} = \frac{\partial \boldsymbol{g}(\boldsymbol{\nu}, \boldsymbol{y})}{\partial \boldsymbol{x}} = \begin{bmatrix} \cos(\phi^B + \theta) & \sin(\phi^B + \theta) & -x\sin(\phi^B + \theta) + y\cos(\phi^R + \theta) \\ 0 & 0 & 1 \end{bmatrix} \tag{48}$$

$$\boldsymbol{G_y} = \frac{\partial \boldsymbol{g}(\boldsymbol{\nu}, \boldsymbol{y})}{\partial \boldsymbol{y}} = \begin{bmatrix} 1 & -x\sin(\phi^B + \theta) + y\cos(\phi^B + \theta) \\ 0 & 1 \end{bmatrix} \tag{49}$$

Followingly, the covariance and cross-variance of the new line feature l_{n+1} may be computed using (50)-(51)

$$\boldsymbol{P}_{\boldsymbol{l}_{n+1}\boldsymbol{l}_{n+1}} = \boldsymbol{G}_{\boldsymbol{x}}\boldsymbol{P}_{\boldsymbol{x}}\boldsymbol{x}\boldsymbol{G}_{\boldsymbol{x}}{}^{T} + \boldsymbol{G}_{\boldsymbol{y}}\boldsymbol{R}_{\boldsymbol{n+1}}\boldsymbol{G}_{\boldsymbol{y}}{}^{T} \tag{50}$$

$$\boldsymbol{P}_{\boldsymbol{l}_{n+1}\boldsymbol{x}} = \boldsymbol{G}_{\boldsymbol{x}}\begin{bmatrix}\boldsymbol{P}_{\boldsymbol{x}\boldsymbol{x}} & \boldsymbol{P}_{\boldsymbol{x}\boldsymbol{m}} & \boldsymbol{P}_{\boldsymbol{x}\boldsymbol{l}_1} & \dots & \boldsymbol{P}_{\boldsymbol{x}\boldsymbol{l}_n}\end{bmatrix} \tag{51}$$

```python

def add_new_landmark(nu, P, R, y_meas, R_meas):
    # Map landmark to state using inverse measurement model
    l = inv_h(nu, y_meas)
    nu_ = add_new_landmark_to_state(nu, y_meas) # Append new landmark to state
    vector

    # Initial landmark covariance
    P_ll = Gx(nu_, l) @ P[0:3, 0:3] @ Gx(nu_, l).T + Gy(nu_, l) @ R_meas @ Gy(nu_,
    l).T
    # Initial landmark cross-covariance
    P_lx = Gx(nu_, l) @ P[0:3, :]
    # Append initial landmark covariance matrix to full-state covariance matrix
    P_ = np.block([[P, P_lx.T],
                   [P_lx, P_ll]])

    if np.shape(nu)[0]-3 > 1:
        R = add_new_landmark_covariance(R, R_meas)
        return nu_, P_, R
    else:
        # This line is the first one observed
        return nu_, P_, R_meas
```

Listing 11: EKF-SLAM Initializing new line segments (python)

9.4.6 Update

In order for an update to take place for correcting the state estimates, the robot must observe a line feature which already has been added to the state vector. The robot should be able to recognize partial observations of line segments due to the short measurement range of the IR sensors. I.e. the data association approach should both be able to infer that a new observation of a line feature is an extension of a previous observation, thus, updating the line segment with endpoints further apart and correcting the line features parameters r and ϕ, and the robot's pose $\boldsymbol{x}$ in such a way that the new observation makes sense from the perspective of the pose. Additionally, the robot might observe a line segment which is embedded inside a larger segment which should result in using the same previously observed line segment's endpoints for the updated line segment's endpoints whilst also correcting the state vector.

For recognizing previously observed lines, the new observation is checked against all previous observations stored in the current state vector to test if the new observation meets the condition's for merging line segments given in (15)-(17). Note that when using the same merging conditions for data association in the filter as for when merging line segments during feature extraction does not imply that the same thresholds should be used. During feature-extraction the robot collects range-bearing measurements over several poses from a much smaller area than the potential size of the whole map. Therefore, the feature-extraction thresholds for merging may be set more relaxed. However, during an update step of the EKF the entire map is being searched through, and falsly claiming that two line segments should be merged would most likely lead to the filter diverging. This would not only make the map of the environment bad, but also following pose estimates would be useless.

Whenever a new observation of a line $\boldsymbol{y}$ is found to be mergeable with a line feature i already stored in the state vector, the EKF update given in (53)-(54) from [38] is computed. Also here, $\boldsymbol{y}$ has to be transformed to the robot's frame of reference in order for the computations to be carried out properly.

$$\boldsymbol{z} = \boldsymbol{y} - \boldsymbol{h}_i(\boldsymbol{x}, \boldsymbol{l_i}) \tag{52}$$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{K}\boldsymbol{z} \tag{53}$$

$$\boldsymbol{P} \leftarrow \boldsymbol{P} - \boldsymbol{K}\boldsymbol{Z}\boldsymbol{K}^T \tag{54}$$

The Kalman gain $\boldsymbol{K}$ is computed according to (56).

$$\boldsymbol{Z} = \begin{bmatrix} \boldsymbol{H_x}^i & \boldsymbol{H_l^i} \end{bmatrix} \begin{bmatrix} \boldsymbol{P_{xx}} & \boldsymbol{P_{xl_i}} \\ \boldsymbol{P_{l_ix}} & \boldsymbol{P_{l_il_i}} \end{bmatrix} \begin{bmatrix} \boldsymbol{H_x^i}^T \\ \boldsymbol{H_{l_i}}^T \end{bmatrix} + \boldsymbol{R_i} \tag{55}$$

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{P_{xx}} & \boldsymbol{P_{xl_i}} \\ \boldsymbol{P_{mx}} & \boldsymbol{P_{ml_i}} \end{bmatrix} \begin{bmatrix} \boldsymbol{H_x^i}^T \\ \boldsymbol{H_{l_i}}^T \end{bmatrix} \boldsymbol{Z^{-1}} \tag{56}$$

Through the Kalman gain, the single re-observation of a landmark will correct the entire state including previous observations of other line features. Followingly, the separately stored endpoints must be updated. The endpoints corresponding to a updated line are updated by orthogonally projecting the endpoints onto the line. Special care has to be taken for the line feature that was found to be mergeable starting the update sequence. For this particular line one must find the endpoints which are furthest apart among the endpoints of the two merged lines. The endpoints that are furthest apart are then projected onto the updated line.

```
''' Called when the robot observes a line feature already part of the map.

    nu: State vector with dimensions (3+2*n, 1), for n landmarks already stored.

    P: State covariance matrix with dimensions (3+2*n, 3+2*n).

    R: Full measurement covariance matrix with dimensions (2*n, 2*n).

    landmark_index: Index of state vector for line feature to be updated

    y_meas: Measurement with dimensions (2, 1)

    R_meas: Measurement covariance matrix with dimensions (2,2)

    endpoints: List of endpoints, where endpoints[i] gives the endpoints of line
    feature i in state vector (nu[3+2*i:3+2*i+2])

'''
def update(nu, P, R, landmark_index, y_meas, R_meas, endpoints):
    i = landmark_index

    R_ = update_measurement_noise_for_observation(R, R_meas, i)

    P_xx = P[0:3,0:3]
    P_xli = P[0:3,3+i*2:3+i*2+2]

    P_lli = P[3+i*2:3+i*2+2, 3+i*2:3+i*2+2]



    # computing the innovation is sparse: only take care of the robot state, the
    concerned landmark and the robot-landmark i covariances
    Z = np.block([[Hx(nu,i), Hl(nu, i)]]) @ np.block([[P_xx, P_xli],
                                                      [P_xli.T, P_lli]]) @ np.block
    ([[Hx(nu, i).T],

     [Hl(nu, i).T]]) + R_[i*2:i*2+2, i*2:i*2+2]

    P_mr = P[3:,0:3]
    P_mli = P[3:, 3+i*2:3+i*2+2]

    # Compute Kalman gain
```

```
    K = np.block([[P_xx, P_xli],
                  [P_mr, P_mli]]) @ np.block([[Hx(nu, i).T],
                                              [Hl(nu, i).T]]) @ inv(Z)

    # ssa() computes smallest signed angle for the angle term
    z = ssa(convert_measurement_to_robot_frame(nu, y_meas), h_i(nu, i))

    nu_ = nu + K @ z    # Update state
    P_ = P - K @ Z @ K.T # Update covariance

    for j in range(len(endpoints)):
        if (nu_[3+j*2] < 0):
            nu_[3+j*2] = np.abs(nu_[3+j*2])
            nu_[3+j*2+1] += np.pi

    # Update endpoints
    endpoints_ = []
    for j in range(len(endpoints)):
        proj_point_p = get_projected_point_on_line(nu_[3+j*2], nu_[3+j*2+1],
    endpoints[j][0])
        proj_point_q = get_projected_point_on_line(nu_[3+j*2], nu_[3+j*2+1],
    endpoints[j][1])
        endpoints_.append([proj_point_p, proj_point_q])

    return nu_, P_, R_, endpoints_
```

Listing 12: EKF-SLAM Update (python)

9.4.7 Map management and loop-closing

As explained in section 9.4.6, whenever a new line segment has been extracted, this line segment is checked if it is mergeable with all previously observed line features. However, from the line extraction process, multiple new line segments may be available at a given time-step. Followingly, an update step is carried out one-by-one for each mergeable line segment. After every update step from the current set of new line features, the updated line feature is kept track of until all the new line features have been processed by the filter (either being added directly to the state, or updating a re-observed line feature). For all the updated line features after one line extraction scan, these are again checked if they are mergeable with line features that were updated from the same scan. If the updated line segments are mergable with other merged (updated) line segments, they are merged through an additional update sequence. This may be seen as a *map management* step, as the goal of checking the newly updated line features with line features that were updated is to remove redundant line features. Intuitively this makes sense to do for a line-feature based SLAM approach, as after an initial update is performed on a newly extracted line segment and a line already stored in the state, the resulting updated line feature may now be mergeable with other line features it previously would not consider to be the same line feature. This is required in order to generate a map with the least amount of features necessary whilst still representing the surrounding environment with enough accuracy.

The requirement for an additional update step is most visible from the perspective of the generated map after a full loop-closure, i.e. when the robot is able to recognize that it has returned to a previously observed part of the map which also encloses the robot's surroundings. In this case the robot will have observed a line segment which *stitches together* two previously observed line features - *placing the last piece of the jigsaw-puzzle*. Since updates are performed one-by-one, two update steps are required in order to represent the three observations as one feature.

10 Testing and verification

10.1 Robot-server MQTT-SN communication

One may verify that the robot and server are able to communicate by examining the logs from the Mosquitto broker. Figure 19 shows a screenshot of the Raspberry pi terminal displaying how messages are sendt from the C++-server to the robot and vice versa.

Figure 19: Raspberry Pi terminal showing MQTT broker log

10.2 *Step-by-step* line extraction

For initial testing of the line extraction process the robot was placed inside a *garage* as illustrated in figure 20 in order to compare the extracted line segments from the method used in [5] and from the method developed during work on this thesis. Similarly to [5], verifying the algorithm was done by collecting data from the robot at each step of the extraction scheme from section 9.3. However, instead of collecting data over a wired connection as done in [5], the developed MQTT-SN task was put into use by making the mapping task publish various data whilst extracting line segments. On the receiving side a subscribing MQTT client, implemented in python with the Paho MQTT client library, was run on the computer at the workstation.

10.2.1 Data collection and preprocessing

Figure 21 shows the point cloud collected from each of the IR sensors after a 90° turn of the sensor tower. The color of a point illustrates which IR sensor the point was sampled from. The most important part of this step is removing any points that are outside the valid measuring range of the IR sensors. Not seen in the figure is that a number of the points would erroneously be placed on top of the robot (marked with a green square).

10.2.2 DBSCAN

In figure 22 the output clusters after DBSCAN has been performed once on each of the IR sensor point buffers is plotted. During this test the parameter ϵ defining the neighborhood radius of a point was set to 20 mm, and `min_Pts` which sets the minimum number of points in a cluster was set

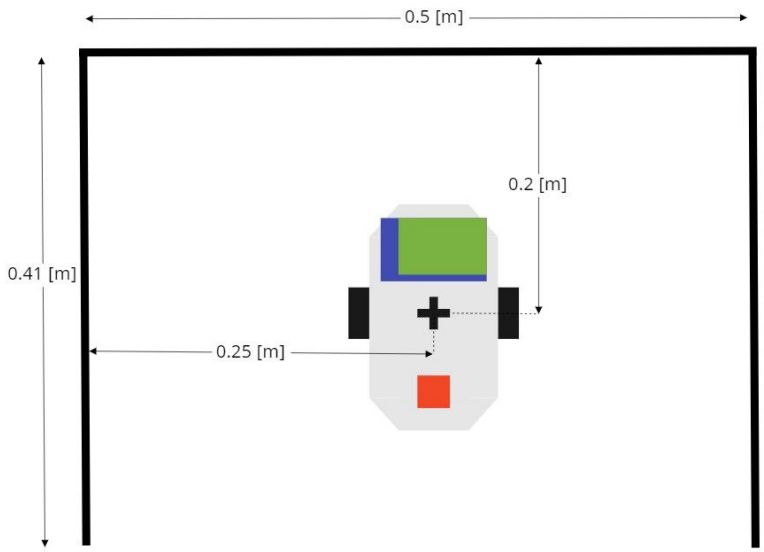


Figure 20: Test setup for line extraction verification

to 5. Larger dots with the same colour are part of the same cluster. Smaller gray dots in the figure were labeled as noise by DBSCAN. From inspecting figure 22 it is evident that one could fine-tune especially ϵ smaller to remove the smaller deviations from the apparent line's mean. However, DBSCAN is still capable of removing the most noisy point measurements.

10.2.3 IEPF

The third step of the line extraction process is plotted in figure 23. Here the IEPF algorithm has been performed on each of the clusters produced by DBSCAN in order to find clusters of points which form straight line segments. For this test parameter T defining the maximum length between the line spanning from the middle indexed point of a cluster to the projection point on the line between the endpoints was set to 10 mm. Of particular interest in the figure are the corners of the garage where the IEPF algorithm has successfully clustered both the DBSCAN clusters (blue cluster in upper corner and yellow cluster in lower corner in figure 22) into two clusters each. From the figure one could state that the IEPF algorithm has over-segmented its input clusters from DBSCAN, i.e. T should have been set lower to allow for the more points to be considered as part of the same cluster. This is especially clear for the lower most wall of the garage where ideally the IEPF algorithm should not have further clustered the points from DBSCAN. However, the IEPF cluster results from the upper and inner wall are satisfactory given the DBSCAN cluster inputs since no further clustering has taken place. Note that all points within every cluster throughout the feature extraction process has maintained its sorted order by bearing, thus, there is no need to search for endpoints of the input clusters for the IEPF algorithm.

10.2.4 MSE line fitting

In figure 24 the IEPF clustered points are fitted onto a line which minimizes the distance among all points in the cluster to the fitted line. In order for the MSE fitted line to represent the environment accurately outliers must be kept at a minimum since each point's contribution is weighted equally. For the case of figure 24, DBSCAN has performed sufficiently well in removing outliers, therefore using the MSE approach works reasonably well for finding initial line segments.

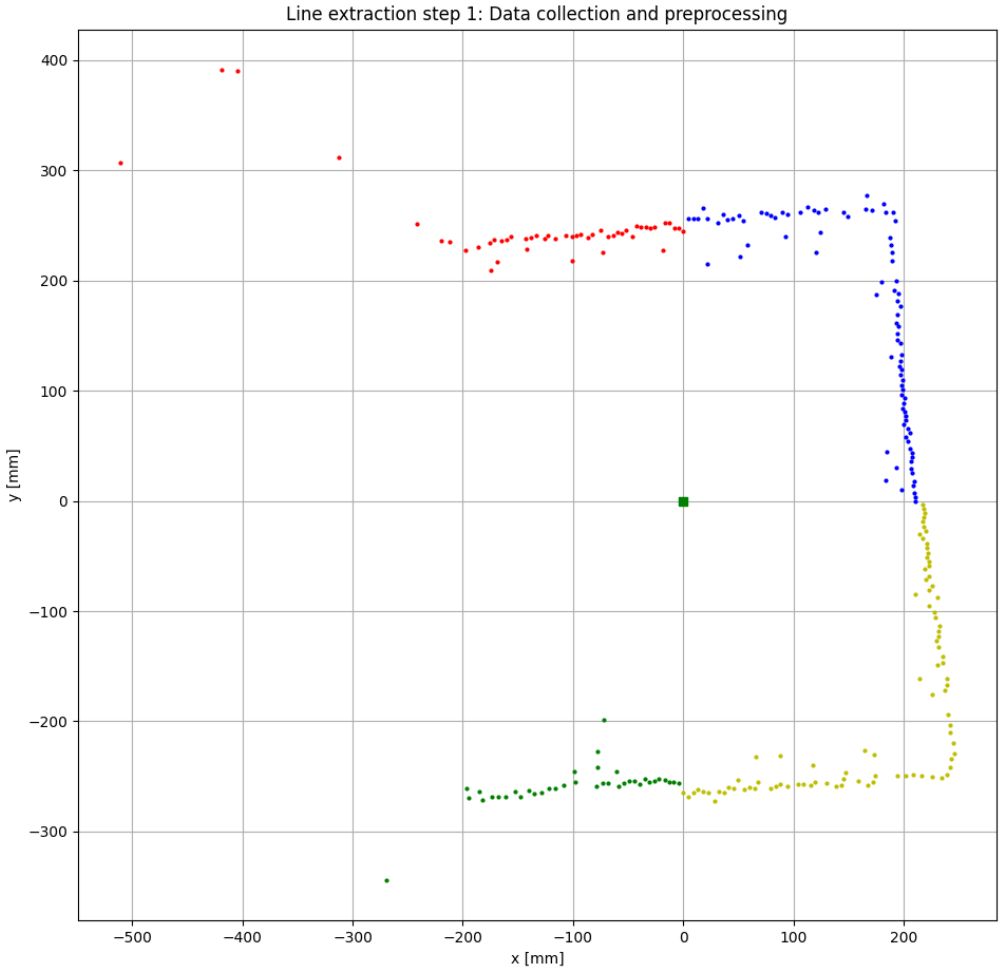


Figure 21: Step 1 of line extraction: Data collection and preprocessing

10.2.5 Line segment merging

Using the MSE fitted line segments directly from figure 24 as measurements for EKF-SLAM would not not be ideal - a line segment merging step within the extraction process is needed to represent the environment with the least amount of features as possible. Figure 25 shows one step in the recursive merging process of the feature extraction scheme. More specifically, the figure shows two dotted line segments, and the resulting merged line. The rightmost (orange) dotted line segment has previously already been merged with the first two line segments from the left of the upper corner of the garage seen in 24. Since the MSE fitted line's points are temporarily stored during line extraction the points of the two mergeable line segments may be used for MSE fitting again, and the resulting fitted line is the blue line in figure 25. The red points in the figure depict which points are used to form the merged line segment.

10.2.6 Extracted line segments

Figure 26 shows the final extracted line segments. Given the points sampled from the IR sensors, the line segments are able to make out the contours of the garage, albeit, they appear slightly slanted. The main reason behind the errors in the line segments' orientations may be due to the lack of points in the corners of the garage. Unsurprisingly, points that are sampled at a right angle from the IR sensor to the point of contact appear more dense, and less dense the higher the angle of attack becomes. I.e. the reflection angle of the IR beam plays a major role in the discrepancies

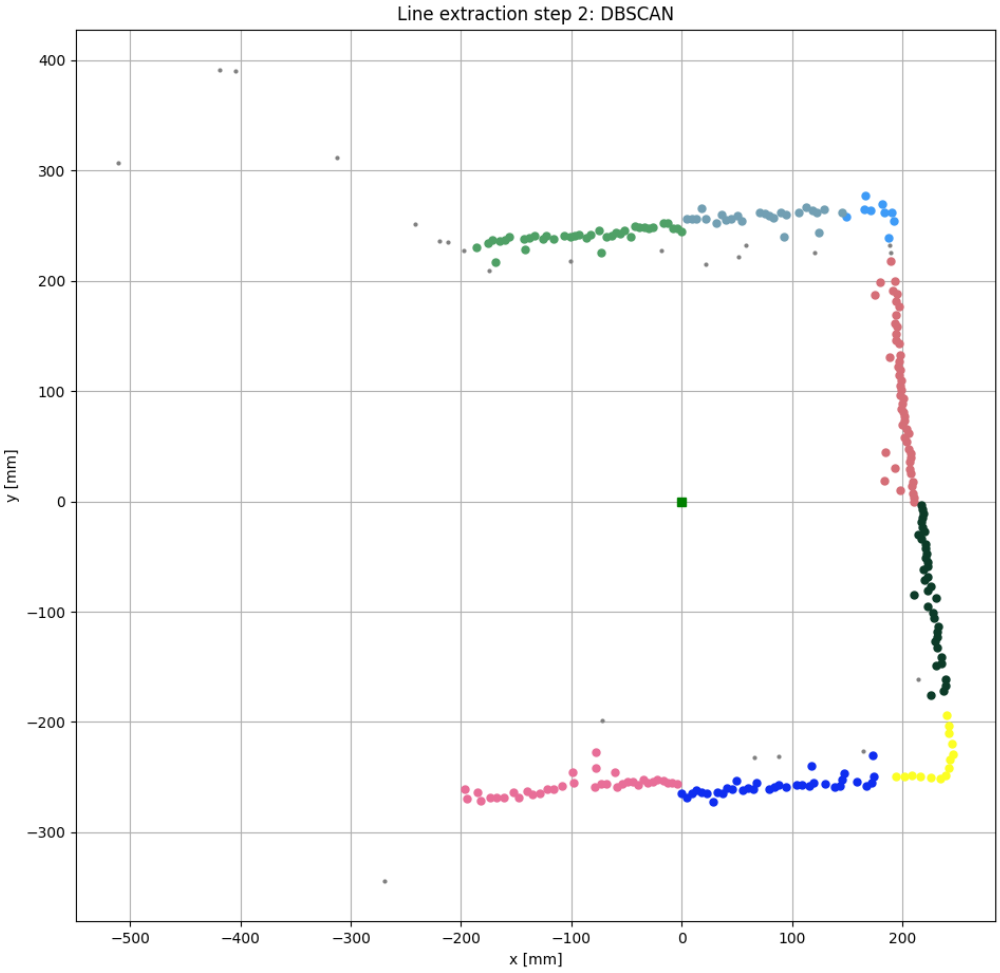


Figure 22: DBSCAN clustering per IR sensor point measurement

between the extracted line segments and the actual line features of the environment.

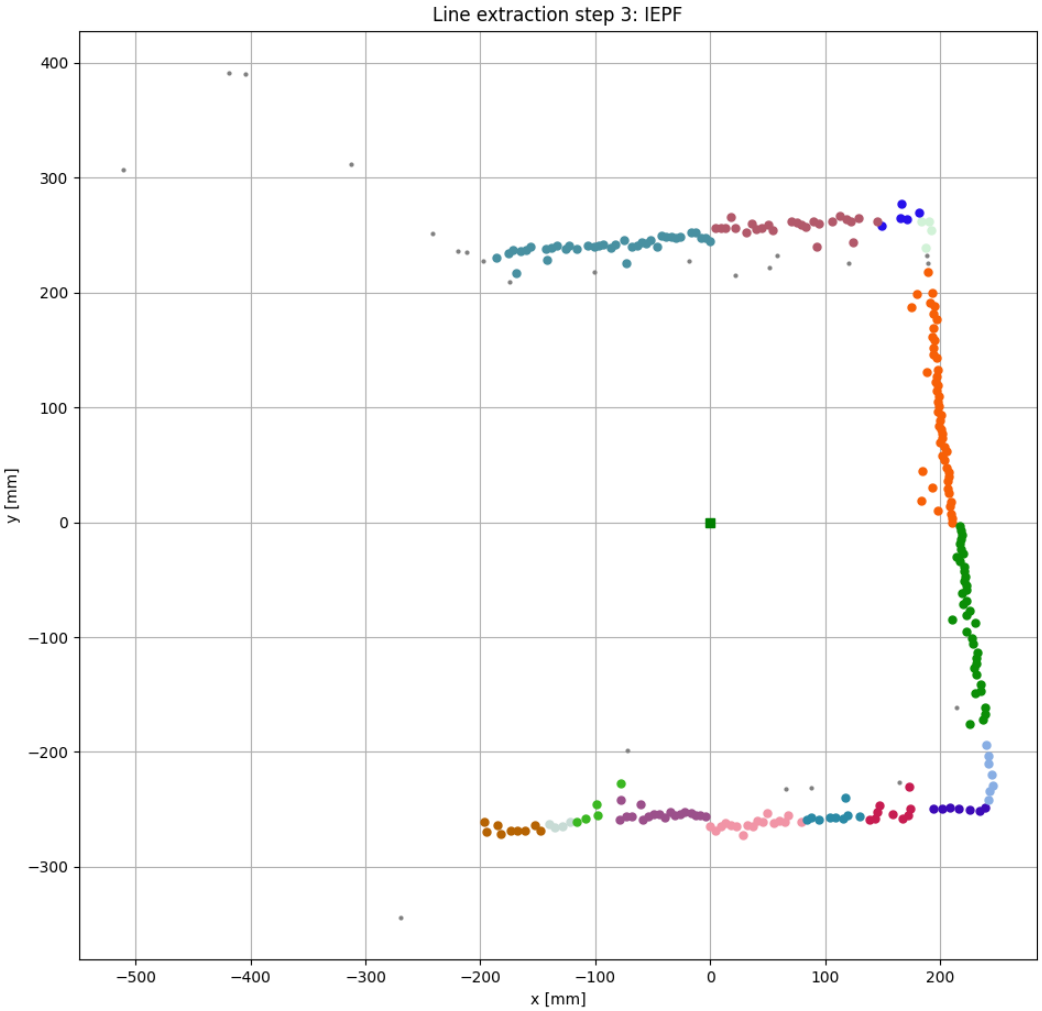


Figure 23: IEPF-based clustering per DBSCAN cluster

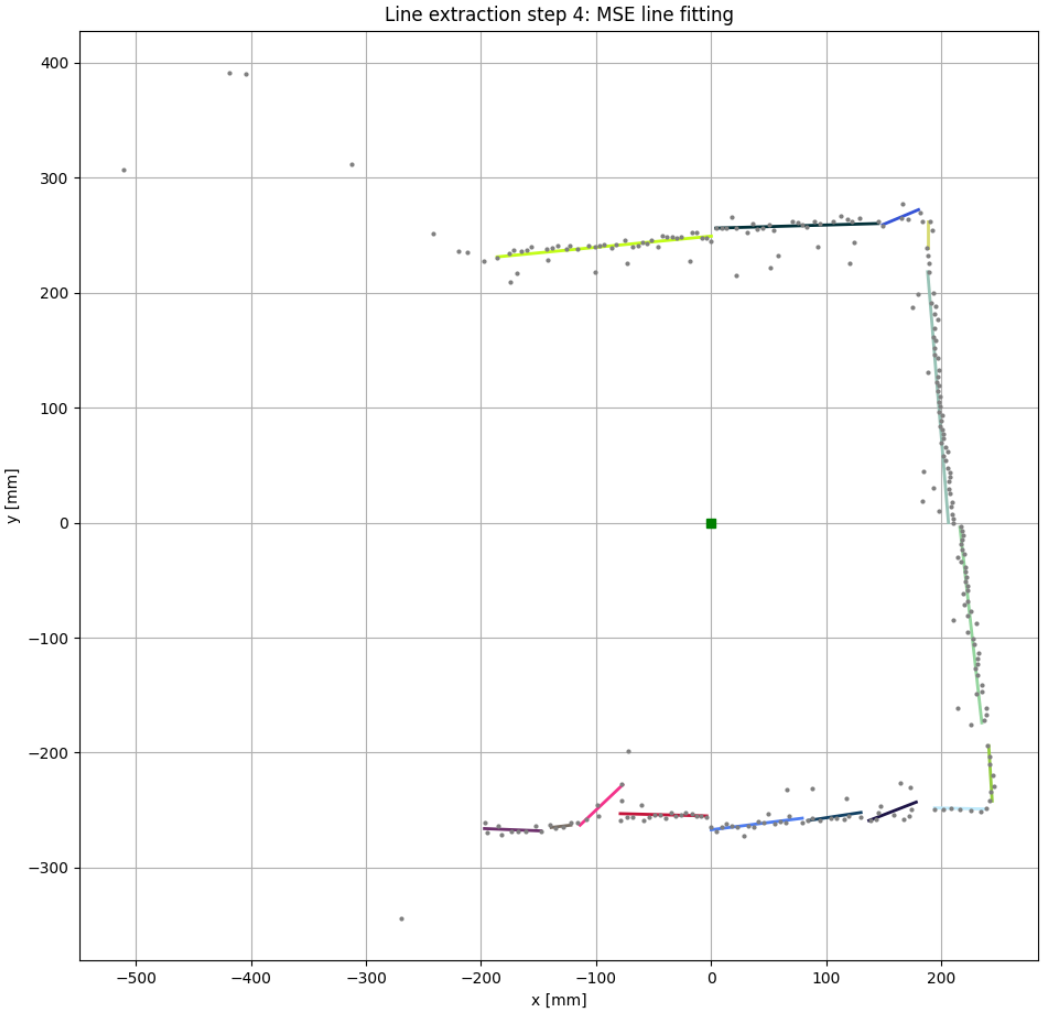


Figure 24: MSE line fitting performed on each IEPF cluster

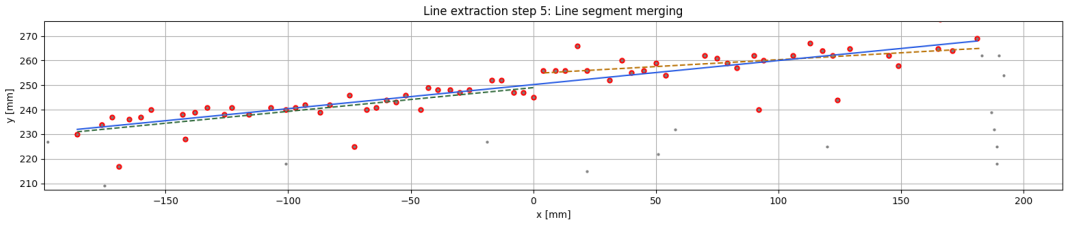


Figure 25: Line segment merging

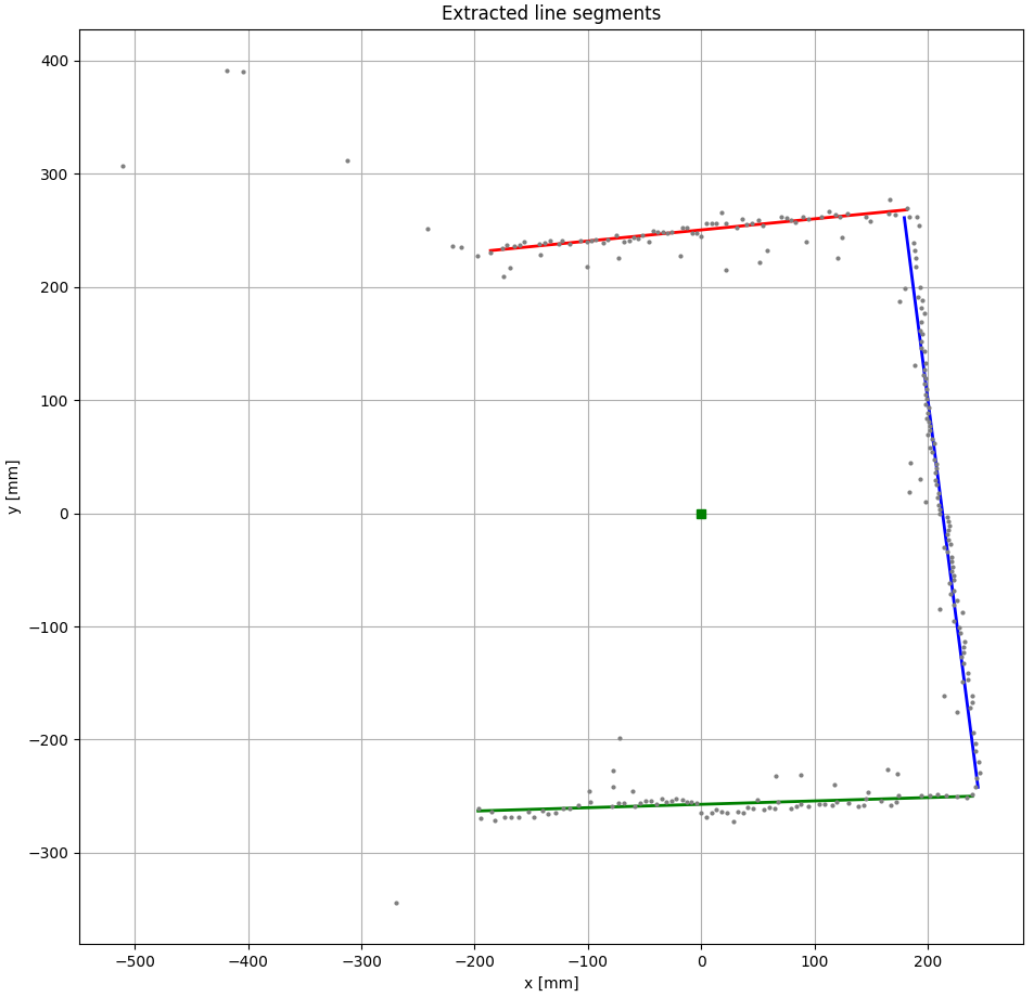
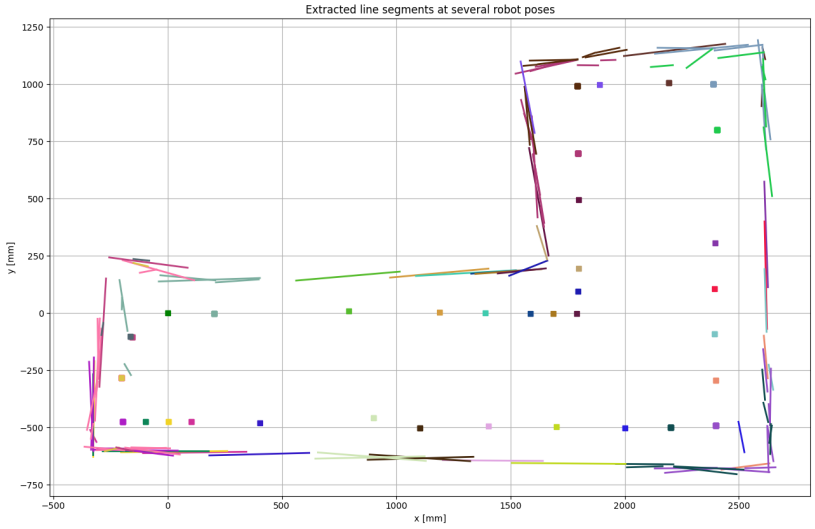


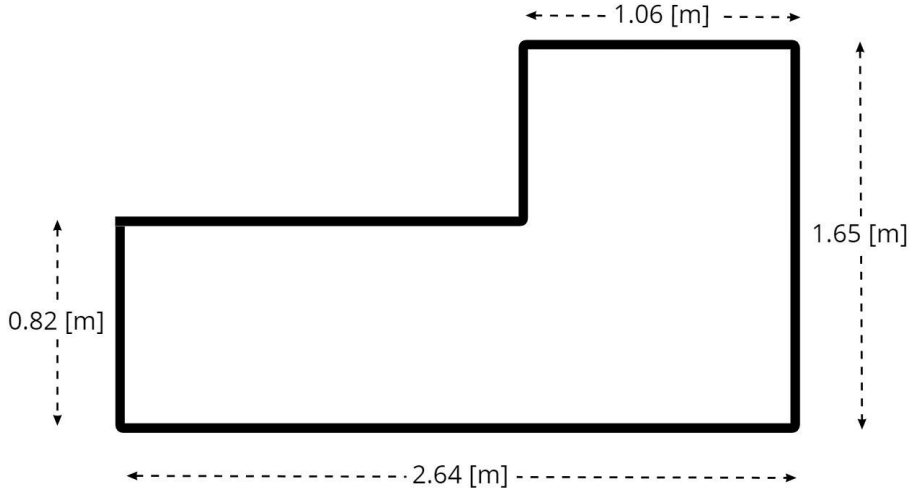
Figure 26: Extracted line segment features

10.3 Line segments extracted at several robot poses

In figure 27a the extracted line segments are plotted whilst the robot was commanded around the maze illustrated in figure 27b. The robot was stopped at every colored dot of figure 27a, line segments with the same color as the position marker illustrate which line segments were extracted at the given pose. Most notable from figure 27a is that the robot struggles to extract line segments close to corners of the maze. Moreover, the length of the extracted line segments are generally smaller close to the corners. The results are plotted by making the robot publish extracted line segments to a subscribing MQTT client in python for plotting, thus, also serves to verify the MQTT-SN task is working as intended. The python script used for collecting data from the robot is given for reference in appendix A.



(a) Lines extracted from several robot poses



(b) Maze

10.4 Line feature EKF-SLAM on dataset from robot

Due to lack of time an EKF-SLAM implementation in C for the nRF52840 robot was not finished. However, a python implementation of the EKF-SLAM approach described in section 9.4 was made. A dataset from the robot was created by publishing over MQTT-SN extracted line segments, pose estimates and control inputs from the estimator task running a standard EKF. Using the dataset

the EKF-SLAM approach was tested.

Initially it was thought that using the linear velocity computed from encoder ticks, and gyro readings for rotational velocity would be enough to drive the EKF-SLAM predictions with similar accuracy as the existing EKF by [19] running on the robot. However, this was not the case as may be observed in figure 31. [19] additionally also used the encoder ticks for computing the rotational velocity of the robot, and these measurements proved to be more reliable than any of the measurements from the IMU. As the created dataset only contained control inputs for linear velocity computed from encoder ticks and gyro measurements, the EKF state estimates themselves were used to compute the control inputs, $\boldsymbol{u}$, which drive the EKF-SLAM predictions. In effect, this should be similar to fusing measurements of rotational velocity from the encoders into the filter. However, instead of explicitly adding a new element in the control vector, the rotational velocity component ω may be seen as a combination of both encoder derived and gyroscope measurements. Of course, this would not be the case for when EKF-SLAM is implemented for running on the robot. For the robot implementation of the EKF-SLAM approach described here the control vector should be expanded for additional measurements, thus, most notably the $\boldsymbol{Q}$ matrix also would have to be expanded to account for this change. Alternatively, one may compute rotational velocity from the encoder ticks alone - disregarding using any inertial measurements due to the high noise level of the IMU. In this case, the EKF-SLAM approach could be implemented for the nRF52840-robot exactly as described here.

Figure 28a shows the robot's track and extracted features while exploring the maze seen in figure 28c. Since all the thresholds for merging were set to zero no updates take place, and the robot relies solely on its predictions for pose estimation. The green dotted line is considered the ground truth position of the robot measured by the OptiTrack motion system. OptiTrack uses the reflective balls mounted on the robot to track the robot's position with high enough accuracy to be considered the ground truth for our purposes. The blue dotted line is the estimates from the EKF computed onboard the robot. The black dotted line represents the EKF-SLAM estimates. Unsurprisingly, the EKF pose estimates are more or less the same as the EKF-SLAM predictions alone since the predictions are based on the control input computed from the EKF. From figure 28a it is interesting to see how the pose has drifted away from the ground truth. Followingly, the generated map is also displaced according to the discrepancies between the filter's pose estimates and ground truth.

In figure 28b the conditions for merging have been set appropriately in order to allow for EKF-SLAM updates to take place. Table 6 summarizes the parameters used for the EKF-SLAM results. Note that using the line uncertainty estimation technique from section 9.3.7 was not used as the computed variances and covariances of the line parameters r and ϕ would ruin the filter. Instead, the measurements covariance matrix of a line $\boldsymbol{R^i}$ was set to be a constant. Tuning $\boldsymbol{Q}$ and $\boldsymbol{R^i}$ is quite demanding as none of the elements of matrices are directly related to the characteristics of the sensors of the robot. Tuning was therefore done experimentally.

From figure 29e one can see how the robot correctly updates its pose estimates as it approaches its starting position. I.e. a loop closure has taken place as the robot is able to recognize that the leftmost wall of the maze is the same wall it observed close to its starting position. Followingly, the robot is able to merge the last extracted line features with the first extracted line features. Since the robot re-observes the same line features it is able to detect the pose drift, and updates its pose accordingly. Note that the entire map is also updated when the robot corrects its pose. From figure 28b this is especially evident as the walls further away from the robot's origin have been corrected in an unwanted manner.

The robot's pose is not only corrected after a full loop-closure. For instance, as can be seen in figure 29a the robot re-observes the same line feature forming the upper wall of the maze at multiple time-steps. When an update occurs one can distinctly see that the robot's trajectory is corrected closer to the ground truth as the robot is using the wall as a point of reference. The same can also be seen in figures 29b-29c where the longest line segment to the right is used to correct the robot's trajectory.

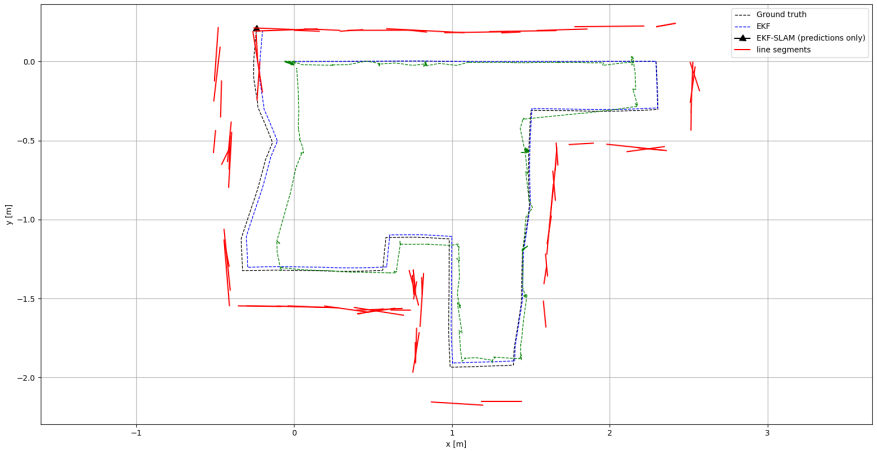
Figures 28a-28b also show how the robot struggles to extract line segments from the corners of the maze whilst the robot is moving. With the approach developed in this thesis the robot should preferably stop and capture full scans of the environment as was done in figure 27a for areas of the

Parameter	Value	Comment
α	0.7854	Defines similarity threshold for ϕ-parameter of normal form line.
β	4.5	Defines similarity threshold for r-parameter of normal form line.
ζ	0.3	Max distance from point to line segment.
$\boldsymbol{Q}$	$\begin{bmatrix} 0.0001 & 0 & 0 \\ 0 & 0.0001 & 0 \\ 0 & 0 & 0.001 \end{bmatrix}$	Process noise covariance.
$\boldsymbol{R^i}$	$\begin{bmatrix} 0.3 & 0 \\ 0 & 0.4 \end{bmatrix}$	Used instead of line uncertainty estimation technique from section 9.3.7 due to unreasonably large values.

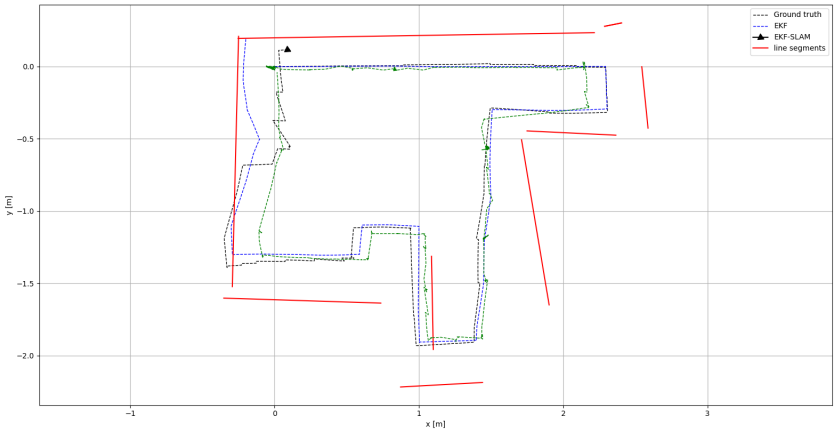
Table 6: Parameters for EKF-SLAM

map where the IR sensor's of the sensor tower are not able to gather enough valid range-bearing measurements.

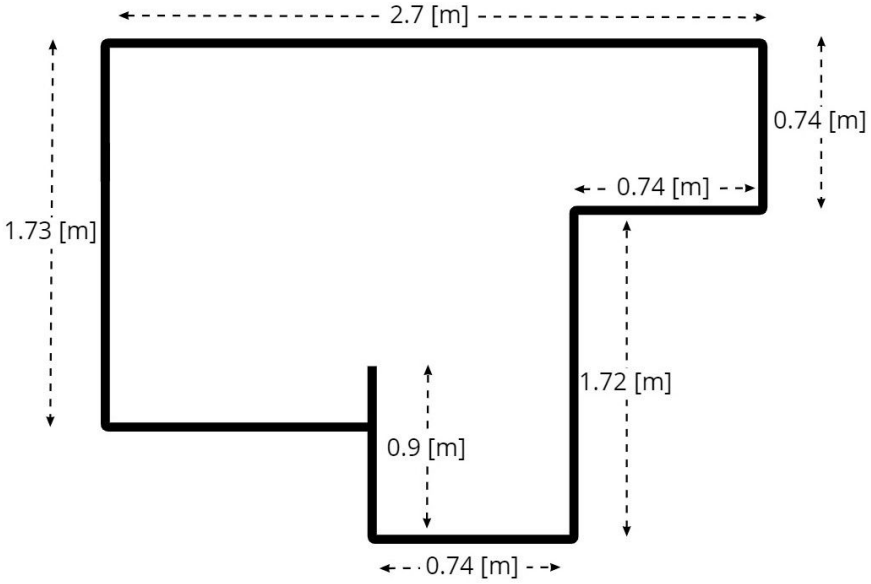
In figure 30 the covariance ellipses of the robot's pose have been plotted at several time-steps whilst the robot explores the maze. For figure 30a the conditions for merging have again been set too strict, thus only EKF predictions are performed for pose estimation. As one would expect, since the robot is dead-reckoning, the robot's uncertainty in its position and heading increase all the way around the maze. On the other hand, in figure 30b the robot's pose covariance is limited by updates caused by re-observing line segment features. One may say that the observed line segments *anchor* the relative odometry measurements to the fixed (absolute) reference the line segments provide. From figure 30b one may also observe that the covariance ellipses at the bottom of the maze are relatively large compared to the other ellipses. This may partially explain why the the loop-closure updates the line segments at the bottom more dramatically than the other line segments of the map.



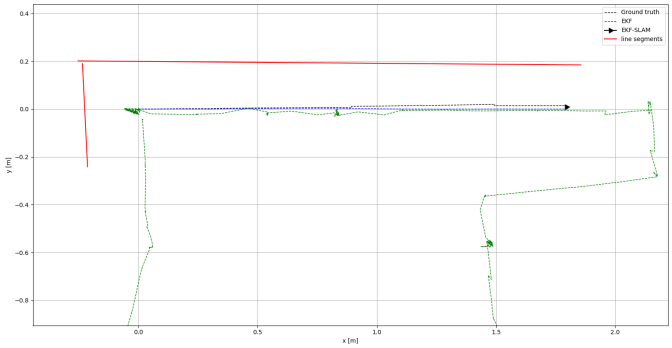
(a) EKF-SLAM predictions only (no updates), EKF, and ground truth



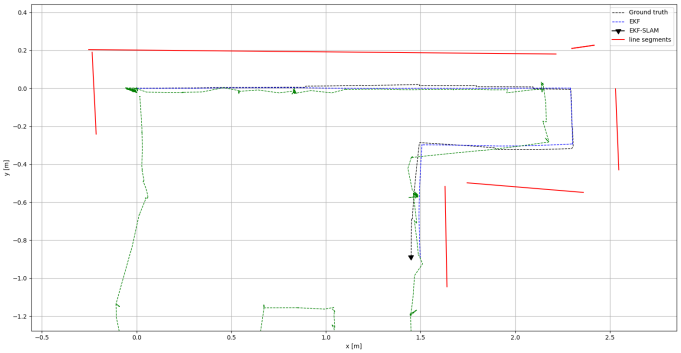
(b) EKF-SLAM, EKF, and ground truth



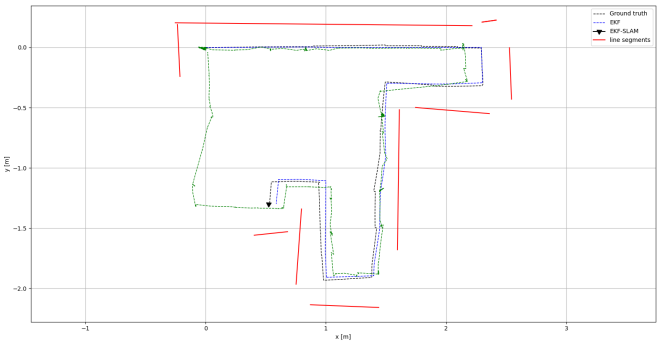
(c) Maze



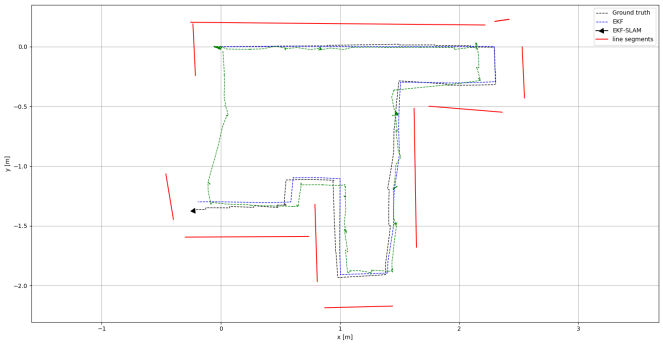
(a) 15 line segments extracted. 2 line features.



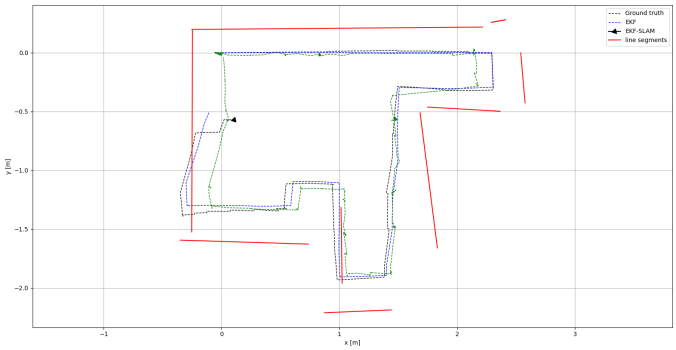
(b) 30 line segments extracted. 6 line features.



(c) 45 line segments extracted. 9 line features.

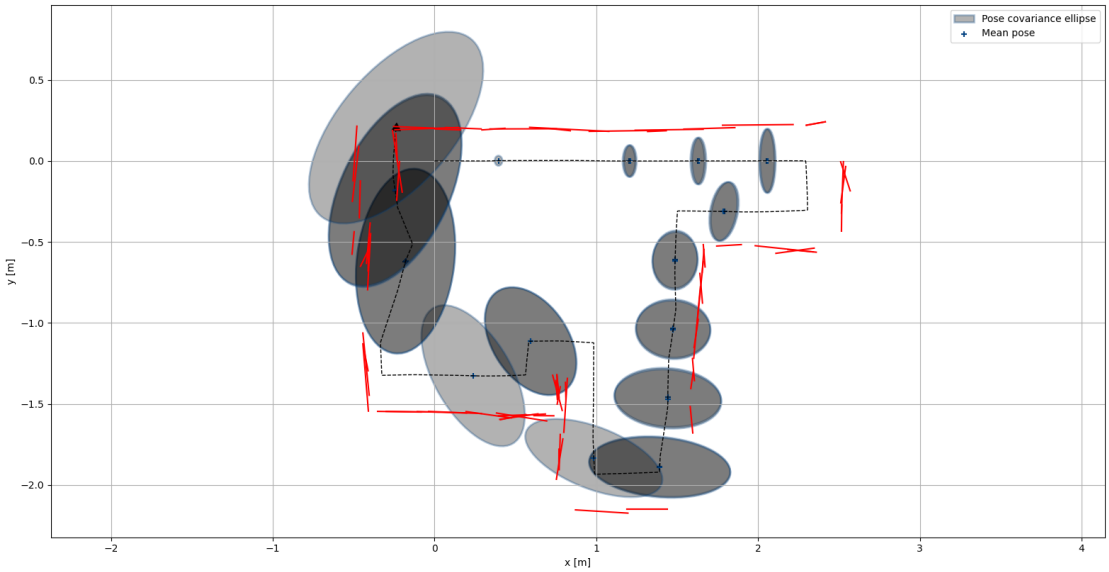


(d) 60 line segments extracted. 10 line features.

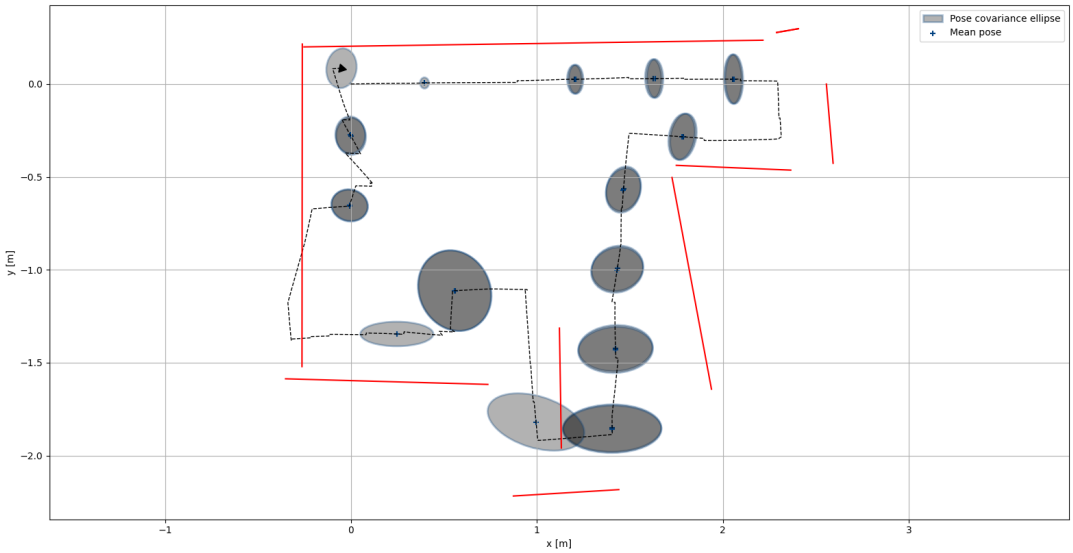


(e) 63 line segments extracted. 10 line features. Full loop-closure.

Figure 29: Robot gradually maps out its surrounding environment whilst estimating its current pose



(a) Pose covariance ellipses with only predictions.



(b) Pose covariance ellipses with line feature updates.

Figure 30: Covariance ellipses of the robot's pose with and without updates.

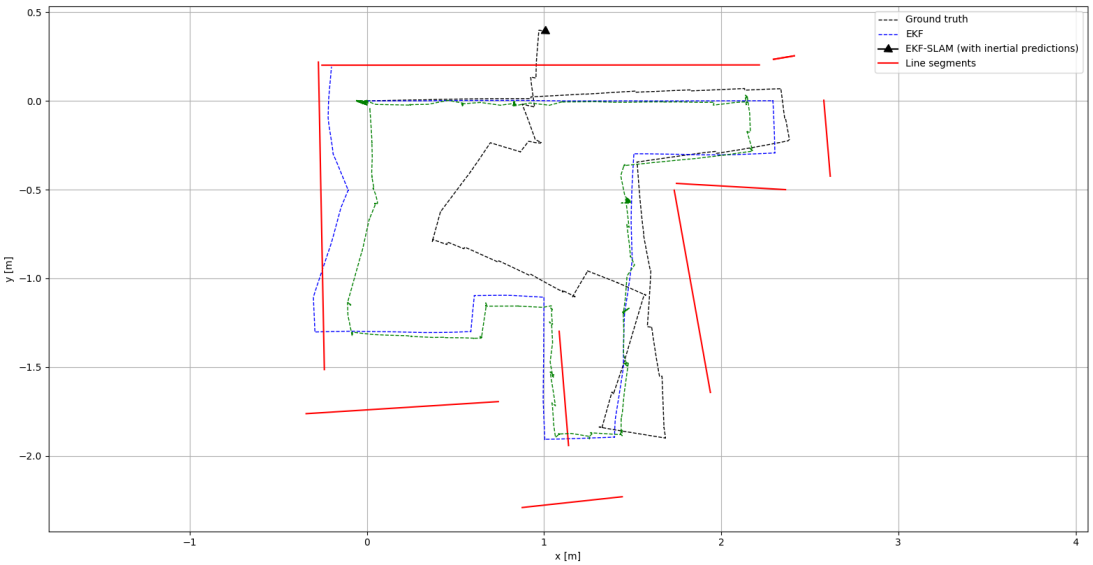


Figure 31: Using gyroscope measurements directly to drive EKF-SLAM predictions does not work.

11 Discussion

11.1 Details on MQTT-SN task implementation

The MQTT-SN task implementation from section 9.1 provides a reliable communication link without the need for Grindvik's legacy layer capable of publishing outgoing messages from other FreeRTOS tasks, distributing received messages to designated tasks, handles reconnections with the broker and retransmission of packets. In the following paragraphs some of the thoughts behind the design choices of the task will be elaborated on.

The reason for dedicating an own task for handling MQTT-SN communication is both a necessity in terms of the protocol being connection-oriented, ensures that access to internal data structures of the MQTT-SN client is not subject to race conditions, and keeps the robot application modular. Periodic keep-alive messages have to be scheduled in order to keep the connection to the broker alive. Furthermore, handling disconnections and re-connections to the broker has to be dealt with in order to provide a fault-tolerant communication module for the robot. It is therefore a natural design choice to run the MQTT-SN client in its own task which also makes it easy for future developers to add the MQTT-SN task to other Thread-supporting, FreeRTOS-robot applications.

The MQTT-SN task functions as a *post box* for other FreeRTOS tasks which is beneficial in order to comply with the timing requirements of the robot application. This ensures that tasks with harder real-time constraints such as the controller- or estimator task are minimally delayed when publishing messages. It is only when these time critical tasks are in either a blocked or suspended state that the lower priority MQTT-SN task may read the contents of the shared inter-task queue and publish messages to the network. Consider a design where instead the methods and data structures of the bear-metal MQTT-SN client are protected with mutual exclusions. This may either lead to a task having to wait for access to the client (which may take too long time), or the task may end up not being able to access the client at all causing possibly valuable information for other nodes on the network to be lost.

The choice of using FreeRTOS queue objects with entries in the queue inserted by copy was made to ensure that the entries would not be lost in the case where the MQTT-SN task is not able to read from the queue before the publishing task overwrites the original message. This does have several drawbacks. Firstly, FreeRTOS queue objects are only capable of inserting entries with the same data structure in one queue. Therefore, multiple queues must be used for multiple MQTT-SN payload formats. This is not scalable when introducing more MQTT-SN payloads. Additionally, queuing large data structures by copy requires more RAM than if the data structures were queued by reference. An alternative solution could be to queue pointers to dynamically allocated memory, and make the MQTT-SN task deal with deallocating the memory when the message has been published. In this way, only one queue holding pointers would have to be used, thus creating a more generic and scalable interface for other FreeRTOS tasks. This does however come at the risk of invalid reads since the publishing tasks are able to modify the contents the pointers are referencing.

11.2 Properties of MQTT(-SN)

There are both advantages and disadvantages of using MQTT(-SN) as the communication protocol between devices of the SLAM-project.

The most prominent advantage of using MQTT(-SN) is the protocol's characteristic publish/subscribe architecture and data-centricity. The CoAP implementations by Murad (2021) [30] will likely not be as scalable as using MQTT(-SN) due to the CoAP protocol's request/response pattern. If one should increase the number of robots communicating over CoAP in the SLAM-project, at a certain point managing connections to other robots at the application layer will not be feasible. With the argument of scalability in mind, using BLE broadcasts would be more beneficial than CoAP. However, BLE does not have the favourable semantics of exposing a node's resources by topics such as MQTT(-SN).

The need for an additional device, the Raspberry Pi running the MQTT-SN gateway, in the system architecture is one of the disadvantages of using MQTT-SN over both CoAP and BLE. Murad discovered a way to join the C++-server directly to the Thread network using a nRF52 USB-dongle inserted into the workstation running the server. This entails essentially implementing a serial-to-CoAP/Thread gateway. Similarly, when using BLE all one needs to connect the robots to the server is a dongle inserted into the workstation running the server.

Using MQTT-SN over Thread may also be viewed as wasting valuable network bandwidth. Since all communication has to be relayed through the broker, one does not take advantage of mesh-networking properties of Thread. E.g. an MQTT-SN client may publish messages that are forwarded through another MQTT-SN client to the broker only for the broker to distribute the same message back to the client who forwarded the message.

11.3 Details on line segment extraction implementation

With hindsight the line segment extraction implementation should not have focused on buffering points from each of the IR sensors into separate point buffers and performing the DBSCAN and IEPF clustering computations on each buffer. The speed performance gain of not having to sort the points in order to find the endpoints of line segments is small compared to especially the runtime of DBSCAN. An alternative implementation would be to store points in the same point buffer when receiving new range-bearing measurements from the sensor tower task. From a logical point of view, the series of steps of the line extraction algorithm would not have to change. DBSCAN takes care of finding clusters of points that form individual features of the environment, not necessarily only straight line segments as well as removing *noisy* points. The clusters produced by DBSCAN may then be fed into the IEPF clustering scheme. Followingly, the IEPF clusters may be fitted to a straight line segment. This approach would have several benefits over the current implementation. Firstly, there would be no need for synchronizing the contents of the FreeRTOS queue which the sensor tower task places the range-bearing measurements into. This does not imply that operations on the queue itself do not require usage of synchronization primitives, e.g. placing and removing elements from the queue between the tasks. However, such access synchronization is provided *out-the-box* by the FreeRTOS queue management API. The current implementation adds a synchronization layer on top of the FreeRTOS API functions for controlling the contents of the queue since it was a requirement that range-bearing elements were sorted by bearing before triggering the line extraction process. Secondly, given that the IEPF parameter T is set appropriately, there would be no need for the last merging step of the line extraction algorithm. The merging step was necessary for the current implementation in order to join similar line segments whose points were derived from different IR sensors. Lastly, since the described modification of the algorithm does not have a requirement in regards to when feature extraction should take place, triggering the extraction process could be done after a given number of range-bearing measurements have been sampled. Thus, no samples would go to waste and the accuracy of the extracted segments may be improved by the higher number of samples.

One could of course make the claim that doing the computations on each of the point buffers separately could be useful in the case where one or several of the robot's IR sensors are replaced by another model with different characteristics than the ones that are currently in use. Thus, one could start tuning the parameters of the line extraction algorithm per IR sensor - a neat feature, however over-complicates the line extraction process.

Further improvements on the line extraction algorithm could involve a weighted line fitting approach as described in section 9.3.5. MSE line fitting without weighting is vulnerable to outliers and may be the source of the extracted line segments appearing slightly skewed in comparison to the contours of the surrounding environment. Since the IR beam's angle of attack on the reflection surface plays an important part in the robot's perceived surroundings one could for instance opt for a weighting scheme which weights points further away from the endpoints of a line segment higher in combination with the measured range. This might help alleviate the problems of finding line segments in corners.

11.4 Comparison of implemented line extraction algorithms

Comparing the line extraction algorithm from [5] to the algorithm developed during the course of this thesis one may conclude that the new implementation is better at extracting line segments from the noisy IR sensor range-bearing measurements. It must be pointed out that in [5] one of the IR sensors were found to be faulty whilst running the tests. Therefore, to be fair, only line segments extracted from the IR sensors in [5] with a similar noise level from the tests in section 10.2 are considered. Additionally, the fact that the extraction method in [5] fails whilst the robot is moving will be disregarded.

The line extraction method used in [5] essentially boils down to a greedy search for collinear points. Although this makes the extraction process faster than the method developed here, the resulting line segments would be sub-optimal as measurements for EKF-SLAM. Consider figure 18a from section 10 in [5], the figure illustrates extracted lines after the sensor tower has rotated 90°. Comparing this to figure 26 it is evident that the new line extraction method is able to capture the contours of the garage more accurately and with less features. The main reason for the improvement in accuracy is likely due to DBSCAN detecting the most noisy measurements and removing these from the following steps of the extraction process.

11.5 Line parametrization

During design of the data association scheme used in both the line extraction process and for the EKF-SLAM it was also experimented with only using the endpoints of two line segments for determining if the segments in consideration should be merged given in (57) in addition to (17). In (57) m_1 and m_2 are the slopes of the two line segments, respectively. (57) tests if the angle between the two line segments under consideration to be merged is smaller than a threshold α [rad]. Special consideration has to be taken in situations when $m_1 m_2 \approx -1$, meaning the lines spanning through their respective endpoints are perpendicular to one another. In the case of perpendicular lines, the test may return immediately that no merge should take place. Additionally, representing slopes parallel (or almost) to the y-axis of the global frame must be done carefully. This would require extra thresholds for when to determine a slope should be represented as infinite. Considering that the C-language will not have any problem with multiplying two `floats` represented by their maximum value (for most hardware this would simply result in the floating point number *wrapping around*) one would have to deal with such edge cases in software. This is the main reason why the normal form parameters of the line were used.

$$\arctan\left(\frac{|m_1 - m_2|}{1 + m_1 m_2}\right) < \alpha \tag{57}$$

At the cost of increased robustness of using the normal form parameters of a line is that setting the threshold parameters for the data association becomes less intuitive. Additionally one also must take into consideration that the parameters r and ϕ have a sensitive relationship dependent on where along the infinitely long line the endpoints of the corresponding line segments are located.

11.6 The uncertainty of a line parametrized by r and ϕ

Any line in the Cartesian plane may be represented using the normal form parameters r and ϕ, thus also any line segment may easily be represented using the normal form parameters given two points representing the line segment's endpoints. However, computing an error model describing a given line segment's uncertainty may become demanding. This is the reason why the simplistic closed-form solution from section 9.3.7 was tried out for the described EKF-SLAM approach. In order for the measurement covariance matrix for a line segment i, $\boldsymbol{R_i}$ (computed according to (19)) to represent the true uncertainty of the line segment with enough accuracy more range-bearing measurements would have to be used in the extraction process than what is currently used. Thus, the improvements on the line extraction scheme described in section 11.3 may also increase

the accuracy of the computed measurement covariance matrix. This would allow for more data points to be collected before triggering the extraction process. Additionally one would expect the length of the extracted line segments to increase with the described improvements on the extraction algorithm. Thus, the problems regarding the blow-up of the measurement covariance matrix might be solved, and one might get away with using the simple computation of the covariance matrix.

Alternatively [44] also outlines how one could compute an estimate of the measurement covariance matrix by propagating the covariance matrix of individual range bearing measurements. At the cost of being more computationally demanding, it may be preferable as one could use information from the datasheet of the servo rotating the IR sensor tower, and the datasheet of the IR sensors to find reasonable values for the variance in bearing, and range. Anyway, the results from section 10.4 may indicate that using a constant measurement covariance matrix is not sufficient for generating an accurate map. The main concern here is the sensitivity of the normal form parameters. Consider figure 17 from section 9.3.7, one can see how small changes in r may influence ϕ. This is what the x_{offset} term is trying to account for in the computation of $\boldsymbol{R_i}$. Thus without the cross-covariance terms in the measurement covariance matrix, it is believed that the EKF is not able to properly propagate the contributions of a measurement which results in the the map being updated slightly wrong when a loop-closure takes place. However, the higher pose covariance where the line segments are updated in an unwanted manner compared to the rest of the map should also be taken into consideration of why these line segments were updated accordingly.

11.7 EKF-SLAM for the nRF52840 robot

The choice of EKF-SLAM for tackling pose estimation and mapping may be summarized by three points. First of all other SLAM approaches described in literature with a low-end sensor suite where only sparse measurements of the environment are available generally seem to solve the SLAM problem with the classical filtering approaches [7][53][12][52][45][28][24][31][1]. Of course many of these approaches precede the era of when graph-based approaches gained traction, however, *state-of-the-art* graph SLAM implementations are normally used when cameras are available (not designed for the sparse sensing capabilities of the nRF52840 robot). Secondly, EKF-SLAM is a natural development step towards making the nRF52840 robot autonomous considering it builds upon the work done by [19] in design of the EKF. Therefore, since for instance general matrix operations are already implemented in C for the nRF52840 robot one could reuse this for the EKF-SLAM implementation. Thirdly, the environments the robots of the SLAM-project are to explore are assumed to be small. It is generally known that EKF-SLAM does not scale well when many landmarks are present. In terms of memory usage the state covariance matrix $\boldsymbol{P}$ requires allocating $(3 + 2n)^2$ floating point values (4 bytes) for n line features. Calling the FreeRTOS API function `xPortGetFreeHeapSize()` whilst running the current robot application returns that approximately $8kB$ of heap memory remains unused. Therefore one may presume the robot will struggle to map an area with double the amount of features that was tested. Also with the increase in number of landmarks the matrix operations may become computationally demanding. This also underlines the importance of using a feature-based SLAM method for the robots, unlike the dense SLAM approach used by the C++-server. Storing raw point-measurements would most likely not be feasible in terms of available memory, and additionally does not deal with the high noise level of the low-cost IR sensors. Using a feature extractor for detecting line segments enables the robot to more efficiently represent its surroundings whilst also provides a way to filter away the most noisy point-measurements. Moreover, the computational capacity of the nRF52840 SoC should be able to handle the extra step of extracting line segments from the range-bearing measurements considering Mane et al. [28] also use a line-feature EKF-SLAM approach for a robot built upon the less powerful 8-bit AVR ATmega2560.

The data association scheme used for the developed EKF-SLAM approach will likely not work for more complex environments than what has been tested. Considering that most of the walls of the maze are long, and all of the walls are placed orthogonal in relation to each other makes the task of identifying line segment features easy since the data association method was tuned for that particular environment. For more complex surroundings it is more probable that false loop-closures will be made, and setting the thresholds for the data association conditions will become hard. A

more robust form of data association could involve a probabilistic approach.

12 Further work

A natural progression for the SLAM-project following the contributions of this thesis is to finish the embedded C implementation for the developed EKF-SLAM approach. This would entail replacing the estimator task currently running an EKF on the nRF52840-based robot with an EKF-SLAM task resembling the developed python implementation of the SLAM solution. Figure 32 illustrates the suggested architecture of the robot from a functional point of view.

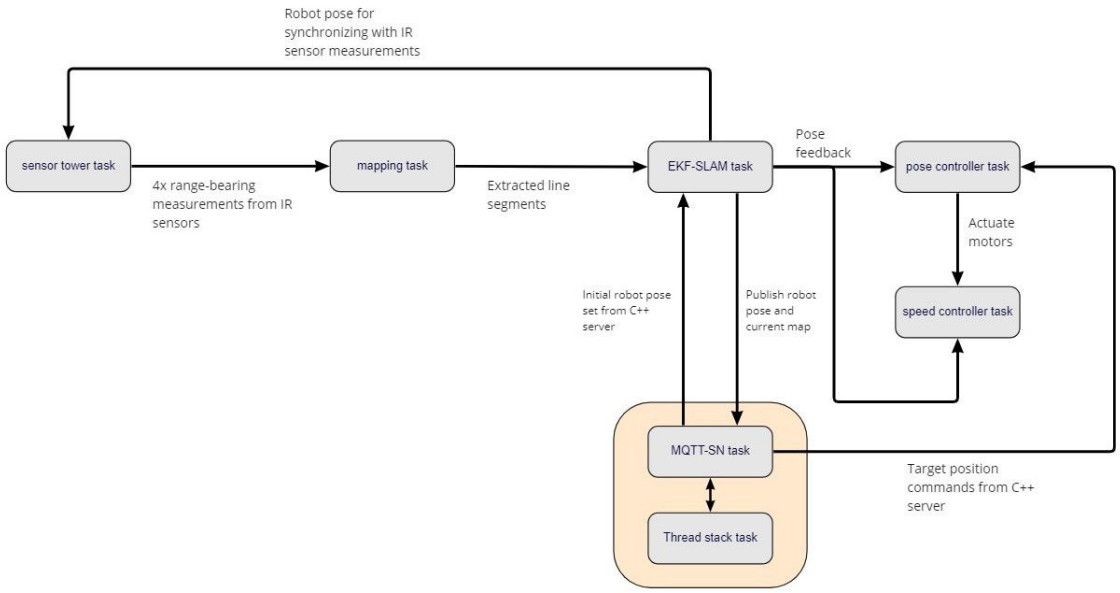


Figure 32: Suggested robot application functional architecture.

Furthermore, it would be interesting to extend the EKF-SLAM solution to incorporate features extracted from other robots as well. Theoretically the EKF framework allows for simply appending new line features to the state vector whenever they are available [39]. As long as all robots are exploring the same map it would not matter which robot detected any line segment, or where any of the robots would be located at the time of extraction. Using the multi-threaded MQTT-SN client, robots may subscribe to a topic reserved for publishing lines, and whenever the EKF-SLAM task of a robot receives a line segment from another robot it would either initialize it as a new feature or proceed with an update step. Although the main part of the EKF-SLAM algorithm would not have to change, one would have to consider how one should initialize the pose of each robot if one were to extend this to a multi-robot SLAM problem. With the current solution, the robot's initial pose is set from the C++ server (by default set to $\mathbf{0}$). With multiple robots, one would have to design a procedure for determining a common frame of reference used by all robots.

When a functional SLAM system has been implemented for one of the robots of the SLAM-project it would be reasonable to start designing path planning and guidance algorithms for the robots in order for the robots themselves to make decisions on what part of the incrementally built map should be explored. For the current system, these decisions are made by the server.

The C++ server application should be further extended with settings in order to turn off the server-side RBPF-SLAM for robots that do the mapping by themselves. Instead of computing SLAM estimates for these robots, the server application should render the most recent update of the map computed by the robots. Furthermore, the server application should be extended to serve as a real-time control center for all robots. I.e. the server application could function as a program for logging and plotting historical data sent from the robots. This is similar to how the plots seen in section 10 were generated. Ideally it would in terms of practicality be beneficial to make the server application run on the Raspberry Pi instead of an additional machine (windows workstation/laptop) as was used during the course of this project. This would make the the system

as a whole more portable and less time would be used for setting up the required devices and software. Alternatively, conducting research on how one may run the MQTT-SN gateway software on the windows machine running the C++-server would remove the need for the Raspberry Pi. The MQTT-SN gateway which is bundled together with Nordic Semiconductor's debian-based Raspberry Pi image currently only has working versions for Linux distributions.

Bibliography

[1] Fabritzio Abrate, Basilio Bona and Indri Marina. 'Experimental EKF-based SLAM for mini-rovers with IR sensors only'. In: *Proceedings of the 3rd European Conference on Mobile Robots* (2007).

[2] L Aguiar et al. 'Kalman Filtering for Differential Drive Robots Tracking'. In: *Aeronautics Institute of Technology* (2017).

[3] Mary B. Alatise and Gerhard P. Hancke. 'A review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods'. In: *IEEE Access* (2020).

[4] Robert Alexander et al. 'Thread Specification'. In: (2017).

[5] Thomas Andersen. 'Feature Extraction Implementation for nRF52840-based Robot'. In: *Project Thesis, NTNU, Trondheim* (2022).

[6] Richard Barry. 'Mastering the FreeRTOS Real Time Kernel'. In: (2016).

[7] K. R. Beevers and W. H. Huang. 'SLAM with sparse sensing'. In: *Proceedings 2006 IEEE International Conference on Robotics and Automation* (2006).

[8] G. Benet, F. Blanes and D. Navarro. 'Line-Based Incremental Map Building Using Infrared Sensor Ring'. In: *IEEE International Conference on Emerging Technologies and Factory Automation* (2008).

[9] Edmund Brekke. *Fundamentals of Sensor Fusion: Target tracking, navigation and SLAM.* 3rd ed. 2020.

[10] B. Siciliano C. Canudas de Wit and G. Bastin. *Theory of Robot Control.* 1st ed. Springer, 1996.

[11] C. Cadena et al. 'Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age'. In: *IEEE Transactions on Robotics* 32.6 (2016), pp. 1309–1332.

[12] Young-Ho Choi, Tae-Kyeong Lee and Se-Young Oh. 'A line feature based SLAM with low grade range sensors using geometric constraints and active exploration for mobile robot'. In: *Autonomous robots* 24 (1) (2007), pp. 13–27.

[13] Eclipse. *Eclipse Paho MQTT-SN C/C++ client for Embedded Platforms.* 2022. URL: https://github.com/eclipse/paho.mqtt-sn.embedded-c (visited on 8th Mar. 2022).

[14] Martin Ester et al. 'A density-based algorithm for discovering clusters in large spatial databases with noise'. In: *Proceedings of the 2nd ACM International Conference on Knowledge Discovery and Data Mining (KDD)* (1996), pp. 226–231.

[15] Thor I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control.* 2nd ed. John Wiley & Sons, 2021.

[16] Andrea Garulli et al. 'Mobile robot SLAM for line-based environment representation'. In: *Proceedings of the 44th IEEE Conference on Decision and Control* (2005).

[17] Antoni Grau et al. 'Robots in Industry: The Past, Present, and Future of a Growing Collaboration with Humans'. In: *IEEE Industrial Electronics Magazine* 15 (2021).

[18] Torstein Grindvik. 'Creating the foundations of a graphical slam application in modern c++'. In: *Master's Thesis, NTNU, Trondheim* (2019).

[19] Knut Høie Håland. 'Improving Navigation in the nRF52 Robot'. In: *NTNU Master's Thesis, NTNU, Trondheim* (2021).

[20] Rolando Herrero. *Fundamentals of IoT Communication Technologies.* Springer, 2020.

[21] InvenSense. *ICM-20948 datasheet.* URL: https://datasheet.octopart.com/ICM-20948-InvenSense-datasheet-115290367.pdf (visited on 17th Dec. 2021).

[22] E. H. Jølsgard. 'Embedded nRF52 robot'. In: *Master's Thesis, NTNU, Trondheim* (2018).

[23] Johan Korsnes. 'Development of a Real-Time Embedded Control System for SLAM Robots'. In: *Master's Thesis, NTNU, Trondheim* (2018).

[24] Tae-Kyeong Lee, Seongsoo Lee and Se-Young Oh. 'A Hierarchical RBPF SLAM for Mobile Robot Coverage in Indoor Environments'. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems* (2011).

[25] Endre Leithe. 'Embedded nRF52 robot'. In: *Master's Thesis, NTNU, Trondheim* (2019).

[26] Kristian Lien. 'Embedded Utvikling på en fjernstyrt kartleggingsrobot'. In: *Master's Thesis, NTNU, Trondheim* (2017).

[27] Magnus Flaten Lindefjeld. 'Obstacle Detection and Avoidance in a 3D Environment'. In: *Master's Thesis, NTNU, Trondheim* (2021).

[28] Akshay A. Mane et al. 'Robotics Based Simultaneous Localization And Mapping of an Unknown Environment using Kalman Filtering'. In: *5th Nirma University International Conference on Engineering (NUiCONE)* (2015).

[29] Michael Skibeli Mullins. 'Implementation of Simultaneous Localisation and Mapping in Robotic System using the improved Rao-Blackwellized Particle Filter'. In: *Master's Thesis, NTNU, Trondheim* (2020).

[30] Sigurd Murad. 'Serial and Thread/CoAP message implementation for the Slam robot'. In: *Master's Thesis, NTNU, Trondheim* (2021).

[31] Viet Nguyen et al. 'Orthogonal SLAM: a Step toward Lightweight Indoor Autonomous Navigation'. In: *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2006).

[32] Nordic Semiconductor. *FreeRTOS CoAP Server Example*. 2020. URL: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsdk_tz_v4.1.0%2Fthread_client_server_example.html&cp=8_3_2_10_4_2_3&anchor=thread_client_server_example_coap-freertos-server (visited on 22nd Feb. 2022).

[33] Nordic Semiconductor. 'nRF52840 Development Kit (PCA10056 v1.0.0): User Guide v1.3'. In: ().

[34] Nordic Semiconductor. *SDK library reference - Application Timer*. 2018. URL: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Fgroup__app__timer.html (visited on 8th Mar. 2022).

[35] Nordic Semiconductor. *Thread Border Router*. 2020. URL: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsdk_tz_v4.1.0%2Fthread_mqttsn_example.html (visited on 27th May 2022).

[36] Nordic Semiconductor. *Thread MQTT-SN Example*. 2020. URL: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fstruct_sdk%2Fstruct%2Fsdk_thread_zigbee_latest.html (visited on 22nd Feb. 2022).

[37] Oasis. 'MQTT Version 5.0 - OASIS Standard'. In: (2019).

[38] Joan Solà Ortega. 'Towards Visual Localization, Mapping and Moving Objects Tracking by a Mobile Robot: a Geometric and Probabilistic Approach'. In: *Networked Digital Library of Theses and Dissertations* (2007).

[39] Sajad Saeedi et al. 'Multiple-Robot Simultaneous Localization and Mapping: A Review'. In: *Journal of field robotics* 33 (1) (2016), pp. 3–46.

[40] Erich Schubert et al. 'DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN'. In: *ACM transactions on database systems* 42 (3) (2017), pp. 1–21.

[41] Nordic Semiconductor. *Getting started with Thread and Zigbee: Hardware support and requirements*. URL: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.threadsdk.v0.8.0%2Fgroup__udp.html (visited on 1st June 2021).

[42] Sharp. *GP2YA021YK datasheet*. URL: https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0a21yk_e.pdf (visited on 17th Dec. 2021).

[43] Silicon Laboratories. *Thread Fundamentals*. URL: https://www.silabs.com/documents/public/user-guides/ug103-11-fundamentals-thread.pdf (visited on 28th May 2021).

[44] Volker Sommer. 'A Closed-Form Error Model of Straight Lines for Improved Data Association and Sensor Fusing'. In: *Sensors (MDPI)* (2018).

[45] Z. Song, K. L. Chen Y.Q. Moore and L. Ma. 'Applications of the sparse Hough transform for laser data line fitting and segmentation'. In: *International Journal of Robotics and Automation* 21 (3) (2006).

[46] Andy Stanford-Clark and Hong Linh Truong. 'MQTT For Sensor Networks (MQTT-SN) Protocol Specification - Version 1.2'. In: (2013).

[47] Arild Stenset. 'nRF52 robot with OpenThread'. In: *Master's Thesis, NTNU, Trondheim* (2019).

[48] Handson Technology. *L298N Dual H-Bridge Motor Driver*. URL: http://www.handsontec.com/dataspecs/L298N%20Motor%20Driver.pdf (visited on 8th Nov. 2021).

[49] Thread Group, Inc. *Co-Processor Designs*. 2022. URL: https://openthread.io/platforms/co-processor (visited on 27th May 2022).

[50] Thread Group, Inc. *Openthread API reference - UDP*. 2022. URL: https://openthread.io/reference/group/api-udp (visited on 8th Mar. 2022).

[51] Thread Group, Inc. *OpenThread Guides: Node Roles and Types*. 2022. URL: https://openthread.io/guides/thread-primer/node-roles-and-types (visited on 3rd May 2022).

[52] T.N Yap and C. R. Shelton. 'SLAM in Large Indoor Environments with Low-Cost, Noisy, and Sparse Sonars'. In: *IEEE International Conference on Robotics and Automation* (2009), pp. 1395–1401.

[53] N. M. Yatim and N. Buniyamin. 'Development of Rao-Blackwellized Particle Filter (RBPF) SLAM Algorithm Using Low Proximity Infrared Sensors'. In: *9th International Conference on Robotic, Vision, Signal Processing and Power Applications* 398 (2016), pp. 395–405.

Appendix

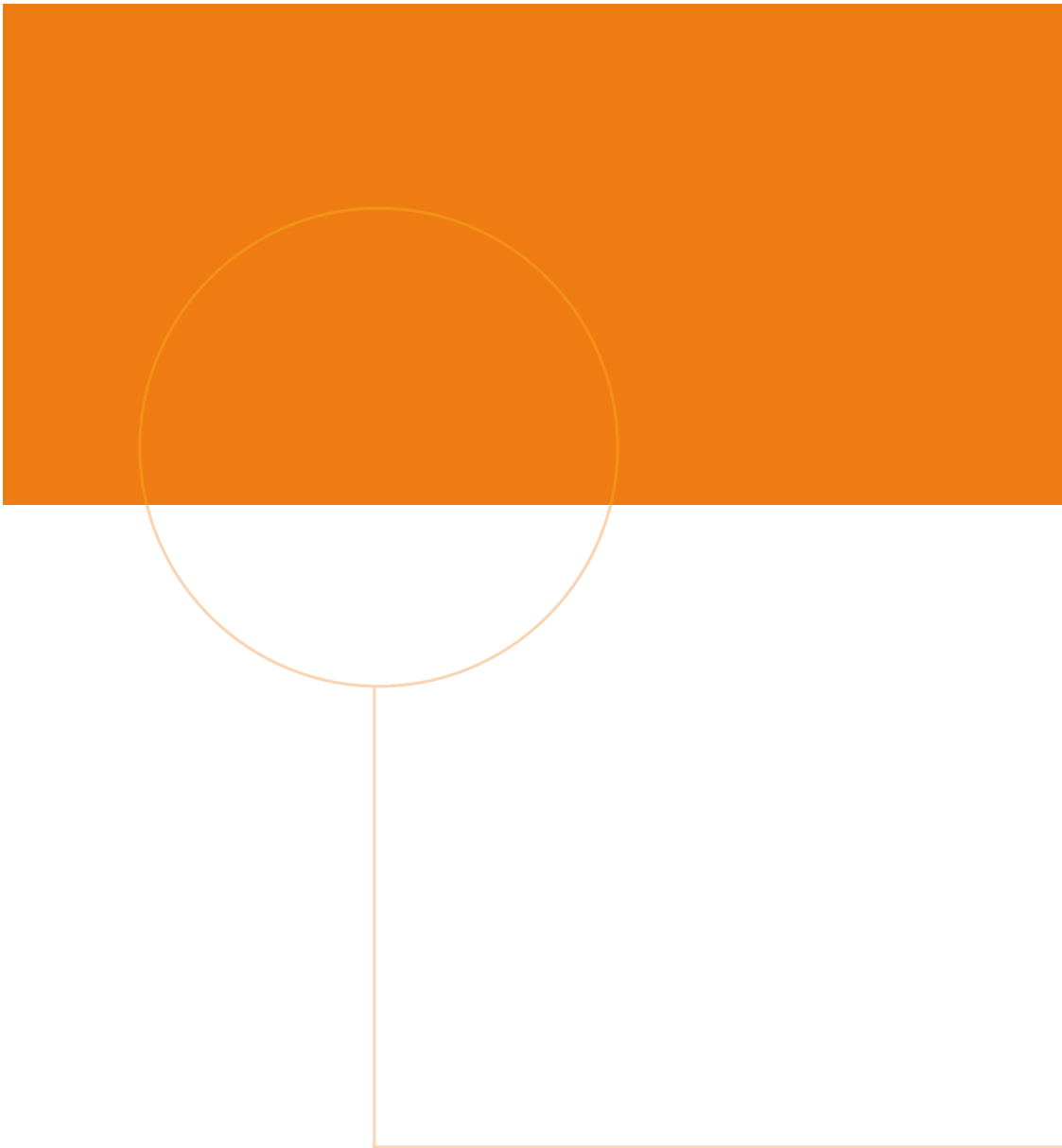
A Python MQTT subscriber script for logging data from robot

```
'''

Script for receiving data from robot. Subscribes to several MQTT topics, and saves the received payloads as JSON-formatted .txt files

'''
import paho.mqtt.client as mqtt
import struct
import json
import matplotlib.pyplot as plt
import signal

ip_addr = "10.53.49.161" # Find broker address on Raspberry Pi with 'ifconfig -a'
port = 1883
keepalive = 60

class SignalHandler():

    def __init__(self):
        self.state = False
        signal.signal(signal.SIGINT, self.change_state)

    def change_state(self, signum, frame):
        signal.signal(signal.SIGINT, signal.SIG_DFL)
        self.state = True

    def exit(self):
        return self.state


flag = SignalHandler()


line_file = open("line_log.txt", "w")
point_file = open("point_log.txt", "w")
dbscan_file = open("dbscan_log.txt", "w")
iepf_file = open("iepf_log.txt", "w")
common_point_file = open("common_point_log.txt", "w")
mse_file = open("mse_log.txt", "w")
mse_point_file = open("mse_point_log.txt", "w")
merge_file = open("merge_log.txt", "w")
join_file = open("join_log.txt", "w")
debug_file = open("debug_log.txt", "w")
estimator_file = open("estimator_log.txt", "w")

def on_connect(client, userdata, flags, rc):
    print("Connected with result code", rc)
    client.subscribe("v2/robot/NRF_5/controller", qos=0)
    client.subscribe("v2/robot/NRF_5/coordinate", qos=0)
    client.subscribe("v2/robot/NRF_5/DBSCAN", qos=0)
    client.subscribe("v2/robot/NRF_5/IEPF", qos=0)
    client.subscribe("v2/robot/NRF_5/line", qos=0)
    client.subscribe("v2/robot/NRF_5/adv", qos=0)
    client.subscribe("v2/robot/NRF_5/point", qos=0)
    client.subscribe("v2/robot/NRF_5/MSE", qos=0)
    client.subscribe("v2/robot/NRF_5/mse_point", qos=0)
    client.subscribe("v2/robot/NRF_5/merge", qos=0)
    client.subscribe("v2/robot/NRF_5/join", qos=0)
    client.subscribe("v2/robot/NRF_5/debug", qos=0)
    client.subscribe("v2/robot/NRF_5/estimator", qos=0)


def on_message(client, userdata, msg):
    if (msg.topic == 'v2/robot/NRF_5/DBSCAN'):
        (cluster_id, x, y) = struct.unpack('<bhh', msg.payload)
```

```
        entry = json.dumps({"id": cluster_id, "x": x, "y": y}) + '\n'
        dbscan_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/point'):
        (cluster_id, x, y) = struct.unpack('<bhh', msg.payload)
        entry = json.dumps({"id": cluster_id, "x": x, "y": y}) + '\n'
        point_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/IEPF'):
        (cluster_id, x, y) = struct.unpack('<bhh', msg.payload)
        entry = json.dumps({"id": cluster_id, "x": x, "y": y}) + '\n'
        iepf_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/join'):
        (cluster_id, x, y) = struct.unpack('<bhh', msg.payload)
        entry = json.dumps({"id": cluster_id, "x": x, "y": y}) + '\n'
        join_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/debug'):
        (cluster_id, x, y) = struct.unpack('<bhh', msg.payload)
        entry = json.dumps({"id": cluster_id, "x": x, "y": y}) + '\n'
        debug_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/mse_point'):
        (cluster_id, x, y) = struct.unpack('<bhh', msg.payload)
        entry = json.dumps({"id": cluster_id, "x": x, "y": y}) + '\n'
        mse_point_file.write(entry)


    elif (msg.topic == 'v2/robot/NRF_5/line'):
        (time, start_x, start_y, end_x, end_y, sigma_r2, sigma_theta2, sigma_rtheta
    ) = struct.unpack('<fhhhhfff', msg.payload)
        entry = json.dumps({"time": time, "start": {"x": start_x, "y": start_y}, "
    end": {"x": end_x, "y": end_y}, "sigma_r2": sigma_r2, "sigma_theta2":
    sigma_theta2, "sigma_rtheta": sigma_rtheta}) + '\n'
        line_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/MSE'):
        (id, start_x, start_y, end_x, end_y, sigma_r2, sigma_theta2, sigma_rtheta)
    = struct.unpack('<Bhhhhfff', msg.payload)
        entry = json.dumps({"start": {"x": start_x, "y": start_y}, "end": {"x":
    end_x, "y": end_y}, "sigma_r2": sigma_r2, "sigma_theta2": sigma_theta2, "
    sigma_rtheta": sigma_rtheta}) + '\n'
        mse_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/merge'):
        (id, start_x, start_y, end_x, end_y, sigma_r2, sigma_theta2, sigma_rtheta)
    = struct.unpack('<Bhhhhfff', msg.payload)
        entry = json.dumps({"id": id, "start": {"x": start_x, "y": start_y}, "end":
     {"x": end_x, "y": end_y}, "sigma_r2": sigma_r2, "sigma_theta2": sigma_theta2,
    "sigma_rtheta": sigma_rtheta}) + '\n'
        merge_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/adv'):
        (id, x, y, theta, ir1x, ir1y, ir2x, ir2y, ir3x, ir3y, ir4x, ir4y, valid) =
    struct.unpack('<B11hB', msg.payload)
        entry = json.dumps({"x": x, "y": y, "theta": theta, "ir1": {"x": ir1x, "y":
     ir1y}, "ir2": {"x": ir2x, "y": ir2y}, "ir3": {"x": ir3x, "y": ir3y}, "ir4": {"
    x": ir4x, "y": ir4y}}) + '\n'
        point_file.write(entry)

    elif (msg.topic == 'v2/robot/NRF_5/estimator'):
        (time, x, y, theta, enc, gyro) = struct.unpack('<6f', msg.payload)
        entry = json.dumps({"time": time, "x": x, "y": y, "theta": theta, "enc":
    enc, "gyro": gyro}) + '\n'
        estimator_file.write(entry)



client = mqtt.Client(client_id="robot-subscriber")
client.connect(ip_addr, port, keepalive)
client.on_connect = on_connect
client.on_message = on_message
```

```
125
126 client.loop_start()
127
128 while(True):
129     if flag.exit():
130         print("Closing files")
131         point_file.close()
132         line_file.close()
133         dbscan_file.close()
134         iepf_file.close()
135         common_point_file.close()
136         mse_file.close()
137         mse_point_file.close()
138         merge_file.close()
139         join_file.close()
140         debug_file.close()
141         estimator_file.close()
142         exit()
```

Listing 13: Python MQTT subscriber script for logging data from robot



NTNU

Kunnskap for en bedre verden