NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK4555 - Specialization Course

Feature Extraction Implementation for nRF52840-based Robot

Author:
Thomas Andersen

Autumn 2021

Table of Contents

List of Figures

List of Tables

1 Summary and conclusion

The nRF52840-based robot is envisioned as a part of a distributed system of other autonomous robots communicating with a central server. The server will receive pose and range-bearing measurements from the robot, and uses a particle filter approach for mapping the environment of the robots. The server is also responsible for issuing commands to the robots, e.g. setting target positions for each robot. It is desirable to make the robots operate more autonomously as this would remove the system's single point of failure - the server. This would entail solving the SLAM problem locally on each robot. This would remove the *master-slave* relationship of the server and robots. The server could then simply function as a GUI for human operators visualizing the robot's environment as well as for giving more complex commands to the robots.

The main contribution to the robot-project presented in this report is the implementation of a line extraction algorithm for an nRF52840-based robot ported with a real-time operating system. Two-dimensional line features of the environment are extracted from four sparse infrared (IR) sensors mounted on a rotating tower. The line extraction algorithm continues partly the work done by [7]. The algorithm is based on the proposed line extraction method by [15], and utilizes the *multiscan* approach of buffering range-bearing measurements in order to increase the denseness of the measurements following the approach of [3]. Extracting line segments from the environment is the first step in the direction of making the robots fully autonomous (without server interaction).

Due to connectivity issues with the server, testing the line extraction algorithm was restricted to stationary mapping a simple "garage" environment. Line segments were successfully extracted from the environment, and used to update a map. Updated line segments of the map are then available for sending to the server, whenever a connection is established.

2 Introduction

Since 2004 the *robot-project* (also known as the *SLAM-project*) has been a project arranged by the Department of Engineering Cybernetics for students to conduct research and development in order to write their specialization and master theses. Embedded systems, sensor fusion and Internet of things (IoT) are the main technical areas in focus as the goal of the project is to map an unknown environment by exploring the area using multiple autonomous robots connected to a server. The idea is to use the robots to collect information about their surroundings and send the information to the server, in order for the server to make a map.

The main focus of this specialization project will be to implement more of the server-side functionality in the autonomous robot itself. More specifically, making the robot able to map out its own surroundings in order to send aggregated sensor data to the server. In the long-run one can envision a distributed system of robots running SLAM locally, sending the robot's local map of the environment to the server for fusion into the global map.

3 Problem statement

Implementation of a feature extraction algorithm on the robot using the Nordic Semiconductor nRF52840 Developement Kit (DK), and sending the aggregated measurements to the Java server is the main goal of this specialization project. In order to accomplish this task, the following work is to be done:

- Testing and resolving any problems of previously developed software and/or hardware to run on the nRF52840 microcontroller.
- Testing and resolving any problems in connecting the robot to the Java-server.
- Calibration of the robot's sensors and tuning of controllers.
  - IR sensors
  - Wheel encoders
  - Inertial Measurement Unit (IMU)
  - Speed controller
  - Pose controller
- Initial tests of the robot's maneuvering and mapping capabilities, referred to as the square- and mapping-test.
- Research different feature-based SLAM algorithms, and identify a feature extraction algorithm suitable for implementing on the nRF52840 microcontroller.
- Implement a line-extraction algorithm.
- Testing and verification of new functionality on the robot.

The structure of the this report follows the outlined tasks to be done.

4 Overview of hardware and software from starting point

This section will give a brief overview of the main hardware and software components of the robot from the starting point of project development.

4.1 Hardware

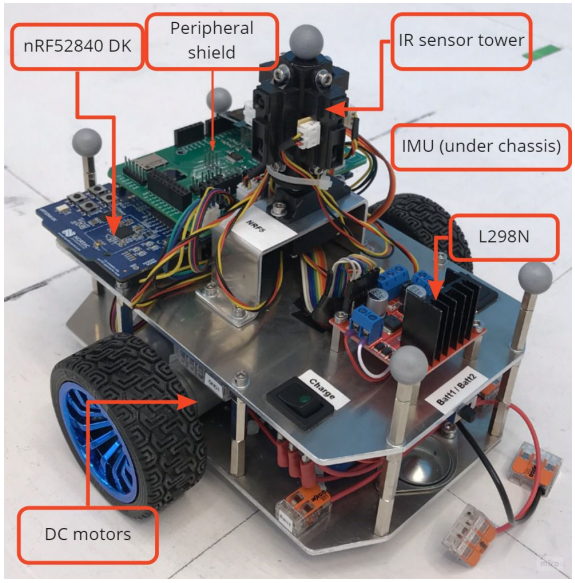


Figure 1: Overview of hardware

4.1.1 nRF52840 Developement Kit

The nRF52840 DK from Nordic Semiconductors is the core module of the robot. The developement kit itself is built around the nRF52840 system-on-chip (SoC) with a 32-bit ARM Cortex-M4 central processing unit (CPU). The developement kit enables fast prototyping of applications designed to use the nRF52840 SoC as it includes a NFC antenna, easily accessible GPIO connectors and headers, buttons, LEDs, external memory and an on-board debugger. [18]

4.1.2 Peripheral shield

The peripheral shield placed on top of the nRF52840 DK was developed by Jølsgård to easily connect the other peripheral units listed below. Refer to [13] for more details.

4.1.3 L298N Motor Driver

The L298N motor driver is used to control wheel speed and direction of the DC motors. Using the motor driver one can power the the DC motors using PWM signals and digital control signals directly from the nRF52840. [21]

4.1.4 Machifit 25GA370 DC motors

Each of the robot's wheels are powered by a separate 12V DC motor. The DC motors are stated to be able to run at 110 rpm. These DC motors also have built-in quadrature encoders used for measuring shaft displacement. [1]

4.1.5 ICM-20948 Inertial Measurement Unit

The ICM-20948 IMU from TDK InvenSens is placed below the metal chassis of the robot. The IMU consists of a MEMS-based 3-axis accelerometer, gyroscope and magnetometer. [12]

4.1.6 IR sensor tower

The robot is equipped with four 2YA21 Sharp IR sensors mounted radially together on top of a servo enabling the tower to rotate. The datasheet for the IR sensors state that they can effectively measure ranges between 10 and 80 centimeters. [19]

4.2 Software

4.2.1 FreeRTOS

The ARM Cortex-M4 CPU running on the nRF52840 has been ported with the open-source Real-Time-Operating-System (RTOS) named FreeRTOS. FreeRTOS is specifically designed to run on embedded devices with real-time requirements. The operating system enables multi-threaded programming functionality through an API which can manage threads of execution (called tasks in FreeRTOS), semaphores, mutexes, queues for inter-task communication, timers, etc. [2]

4.2.2 Software base

The current software application developed over the years by several students consists of a total of six FreeRTOS tasks, these are:

- A communication task responsible for handling communication with the server.
- An estimator task for state estimation, implemented using an Extended Kalman Filter.
- A pose controller task.
- A speed controller task,
- A sensor tower task for controlling the servo rotating the tower and reading IR measurements.
- A display task for controlling the OLED screen.

Along with the running tasks, there have also been implemented several software drivers for interfacing peripherals, among these are drivers for the:

- Motors
- IR-sensors
- OLED-screen
- I2C communication
- Accelerometer
- Gyroscope
- Encoder

5 Calibration of robot

Since the robot was recently purchased and assembled, calibration of the robot before performing initial tests of the system was required. In order to ease the process of calibrating sensors and tuning controllers a python application was made to read and plot values over serial communication with the nRF52.

5.1 Sensor calibration

5.1.1 Encoder calibration

In order to measure encoder ticks per wheel rotation, each wheel of the robot was rotated forwards and backwards one rotation. As seen from Figure 2, one rotation of the robot's wheels accounts for ~ 855 ticks. When performing a line-test, commanding the robot via the server to move in a straight line forward 3.5 m, the robot would consistently move passed its target position by ~ 0.3 m, whilst the robot's estimated position was 3.48 m. To account for this error, a scaling factor was heuristically introduced:

$$SF = \frac{\text{distance\_estimate}}{\text{actual\_distance}} \tag{1}$$

Scaling the observed encoder ticks per rotation during manually turning the wheels of the robot with (1), gives 783 encoder ticks per rotation. Consequently, updating the variable `ENCODER_TICKS_PER_ROT` defined in `robot_config.h` to 783 reduced the robot would not reach its target by 0.07 m. By repeating the process above, we obtain a new value for encoder ticks per rotation of 794. With encoder ticks per rotation set to 794 the distance error is less than 0.01 m, which is good enough for our purposes.

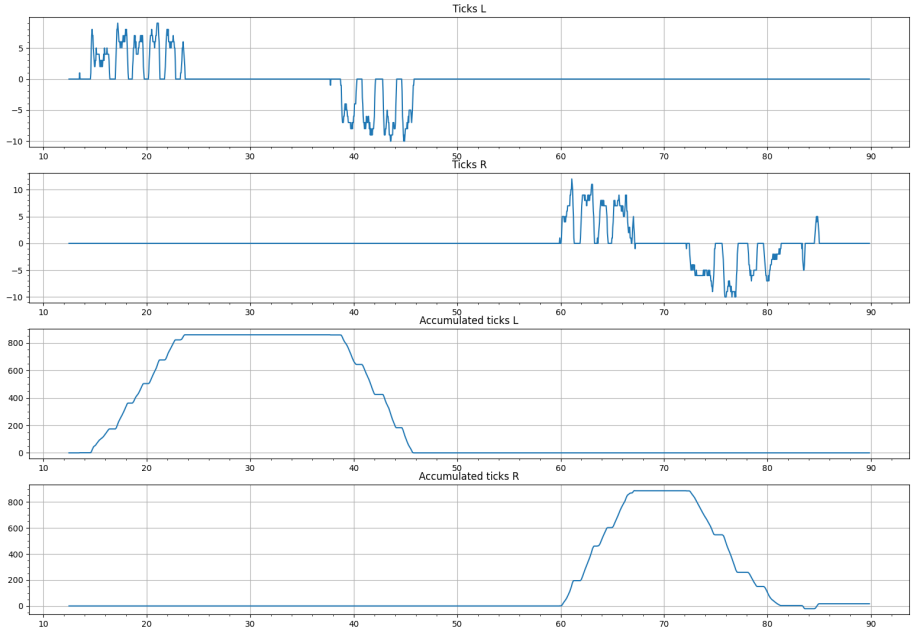


Figure 2: Results after manually turning each wheel one rotation forward and backward. The horizontal axis unit is seconds. The upper two plots show encoder ticks between each sample for the left and right wheel of the robot. The lower two plots show the accumulated value of encoder ticks for the left and right wheels.

5.1.2 IR calibration

Calibration of the IR sensors was carried out by Lindefjeld [14].

5.1.3 IMU calibration

The IMU is automatically calibrated by software before the robot starts to move, therefore, it was not deemed necessary to calibrate the IMU.

5.2 Controller tuning

5.2.1 Tuning of speed controller

From testing the software driver for the motor, it was evident that setting an equal duty cycle of the PWM signal to each of the motors would not result in the wheels rotating at the same speed. This can be seen in Figure 3, where the left wheel seemingly rotates faster. In practice, this could also be seen by the fact that the robot would always slightly turn right. This issue probably stems from manufacturing inaccuracies of the DC motors and/or motor driver card. Followingly, this advocates the usage of a speed controller for each wheel making up for any inaccuracies in hardware and varying friction between the wheels and terrain. Thus, tuning of the speed controller was conducted.

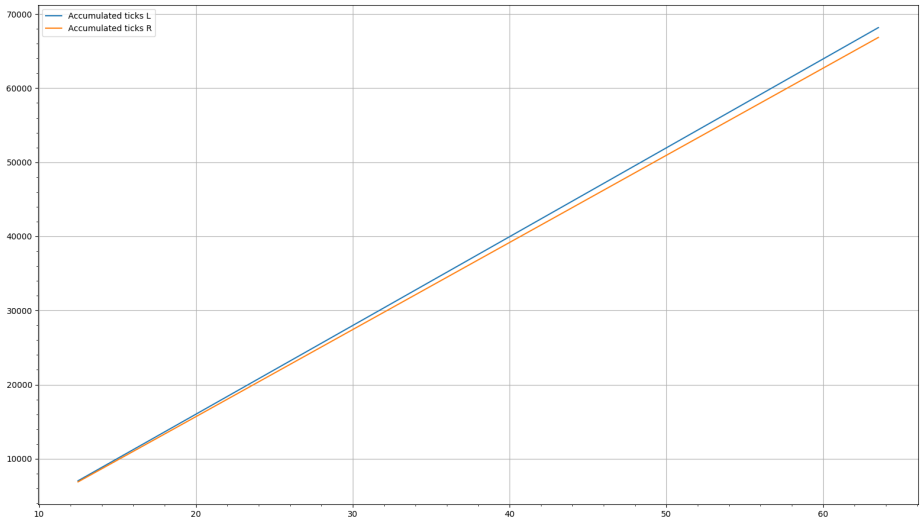


Figure 3: Using the motor driver directly would result in the left wheel rotating faster than the right wheel. The horizontal axis unit is seconds, the vertical axis unit is accumulated encoder ticks.

The input to the speed controller is a reference speed given in m/s, since the motor driver uses a percentage of PWM duty-cycle as input, it is necessary to convert m/s to a percentage of the top-speed of the robot. By filming the robot moving across a given distance, the maximum speed of the robot was found to be $\sim 0.32 m/s$. Consequently, the variable `MOTOR_MAX_SPEED_M_PER _S` was declared in *robot_config.h* and set appropriately.

The speed controller derives the speed of the each wheel (to be used as feedback) by differentiation of the length each wheel has rotated respectively. Thus, it was necessary to measure the the circumference of the wheels, and update the appropriate variable in `robot_config.h`. `WHEEL_CIRCUMFERENCE_MM` was therefore set to 210.

Differentiating the length of rotation of each wheel to obtain the estimated speed of the robot results in noisy speed estimates. Using the estimated speed of the robot as feedback directly would not give satisfactory results. More specifically, it would cause the robot to move jittery

which would likely strain the DC motors over time. Therefore, the signal is digitally low-passed filtered before being used as feedback. In order to ease tuning of the low-pass filter the variable, `LOW_PASS_SPEED_WEIGHT` was declared in `robot_config.h`. This variable weights the last estimated speed with the new estimated speed measurement. Increasing its value will smooth out the signal at the cost of increasing delay. In Figure 4 one can see the raw unfiltered speed estimate compared to the smooth low-passed filtered speed estimate. In this case the weight variable was set to 0.8.

The PID controller for speed implemented by previous students was tuned experimentally in order to obtain a satisfactory response. The proportional gain was set to $K_p = 800$, integral gain was set to $K_i = 1000$, and differential gain was set to $K_d = 0$. The resulting response can be seen in Figure 4.

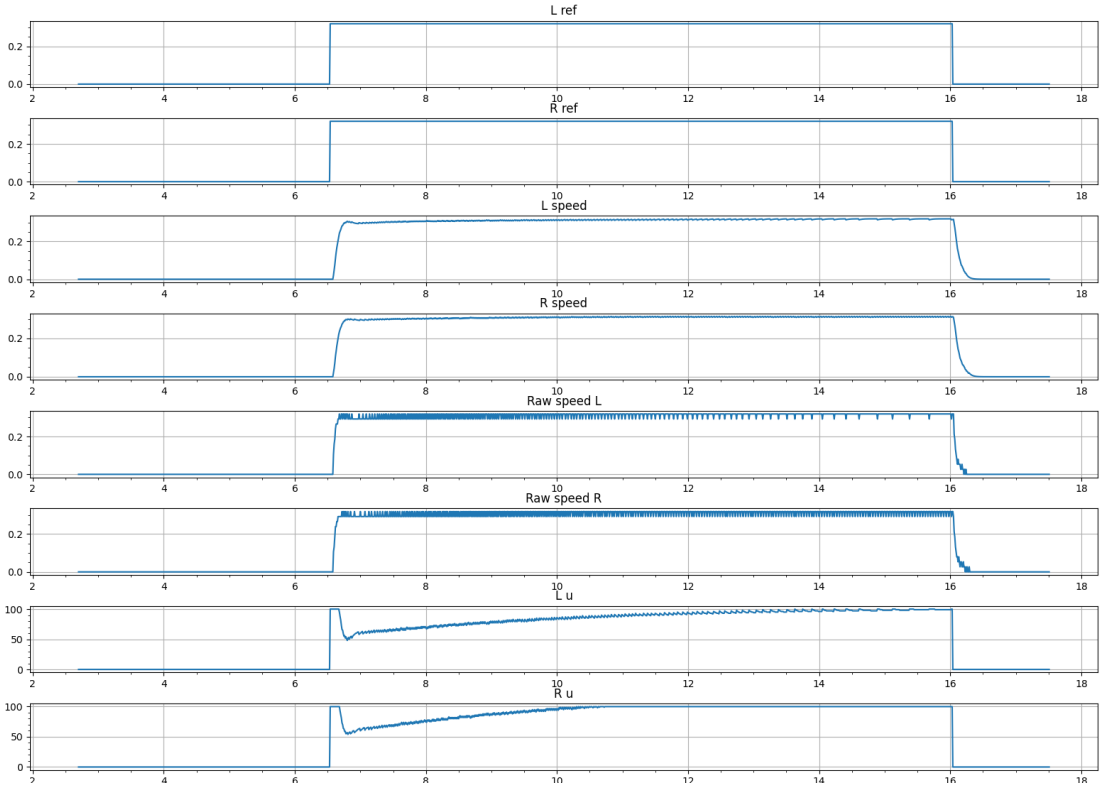


Figure 4: Speed controller response. In pairs from top to bottom are plots of the reference speed $[m/s]$, estimated speed $[m/s]$, unfiltered (raw) speed $[m/s]$, and control output [%duty cycle]. The horizontal axis unit is time $[s]$.

5.2.2 Tuning of pose controller

When examining the software implementation of the pose controller, it was noticed that the control loop frequency was not user-defined. Thus, the pose controller would run at the frequency that the FreeRTOS scheduler would allocate for the task. Since the speed controller is used in the pose controller for successive loop closure, it is important that the pose controller runs slower than the speed controller, in order to ensure stability. Therefore a delay was introduced in the pose controller loop, larger than the delay in the speed controller loop.

Tuning the controller for pose (position and heading) was difficult since there was no way to analyse the response of the robot without connecting it to the stationary PC using a micro usb cable. However, the resulting response when setting a reference position of (x, y) = (1, 0) $[m]$ is given in Figure 5. The robot starts of in position (x, y) = (0, 0), with a heading of $\theta = 0$ $[rad]$, as wanted, the robot will immediately throttle up to max-speed, and de-accelerate upon approaching its reference position.

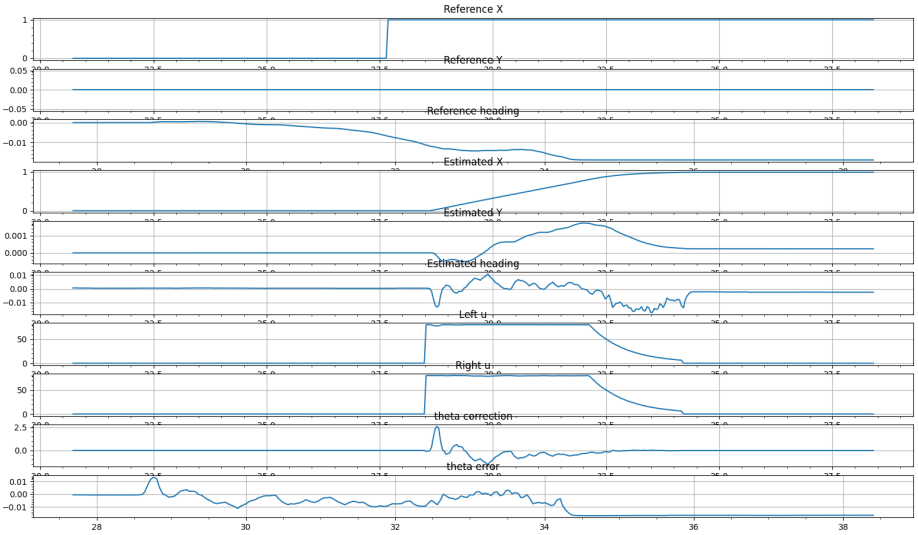


Figure 5: Pose controller response.

6 Initial tests

After calibrating the robot, the so-called *square-test* and test of the combined robot and server mapping abilities were conducted. The square-test is conducted in order to test the robots maneuvering capabilities, by making the server send consecutive target positions to the robot, to make the robot maneuver in a square shape. During the mapping tests of the system, the robot was controlled entirely by the server. The robots sends IR scans to the server enabling the server to compute a map of the robot's surroundings using SLAM.

6.1 Square-test

In 6 the robot was maneuvered clockwise through targets making up a 1x1 m square. From the figure it can be seen that the robot's heading is inaccurately off its intended path consistently by a couple of degrees. Since, there is no positional or heading feedback from the server, the robot wrongfully believes it is on its intended path. By rotating the real path of the robot to adjust for its initial heading error, one can see that the robot indeed is more or less following what is illustrated as the robot's intended path.

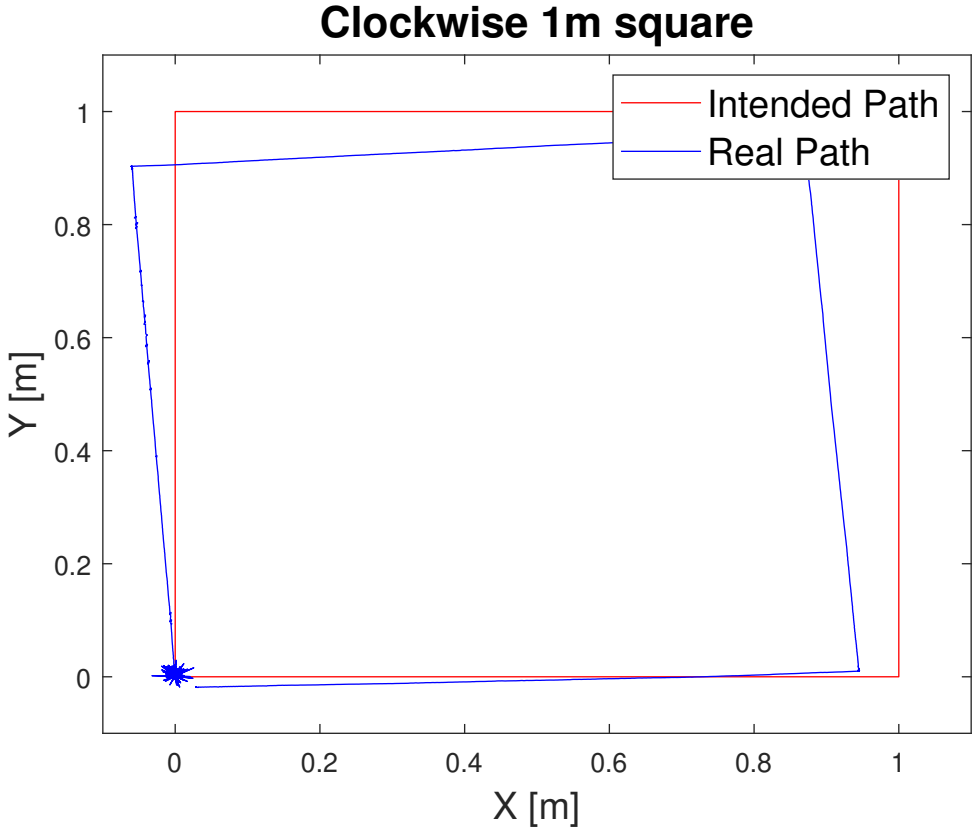


Figure 6: Clockwise movement of robot.

In 7 the robot was tasked to move counter-clockwise. From this figure the estimation errors of the robot become more visible. The robot consistently does not reach its target distance it should cover with an error of $\sim 1-2$ $[cm]$ (when not accounting for heading error). Including the overshoot of every right turn the robot performs, the distance error to the target accumulates.

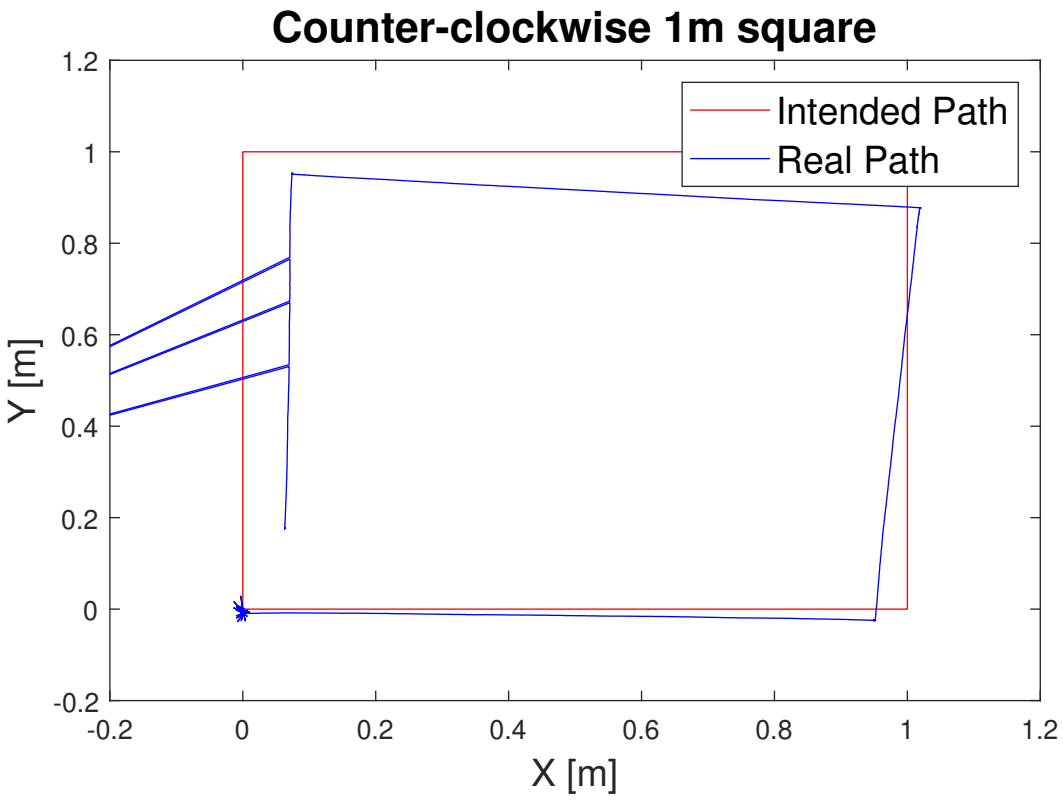


Figure 7: Counter clockwise movement of robot.

6.2 Mapping

From figure 8 and 9 the results from the maze-mapping test can be observed. The robot was unsuccessful in mapping out the whole maze due to the robot abruptly disconnecting from the server. This is a known error using the robot together with the Java-server, it will simply disconnect after a certain period of time.

Other than connectivity issues, there seems to be a lot of noise in one or more of the IR sensors. This is especially well highlighted in figure 9 where the robot is not able to move through the maze due to it falsely detecting objects in its path.

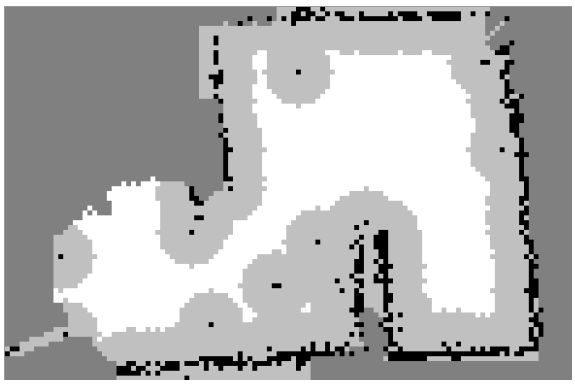


Figure 8: Mapping of maze

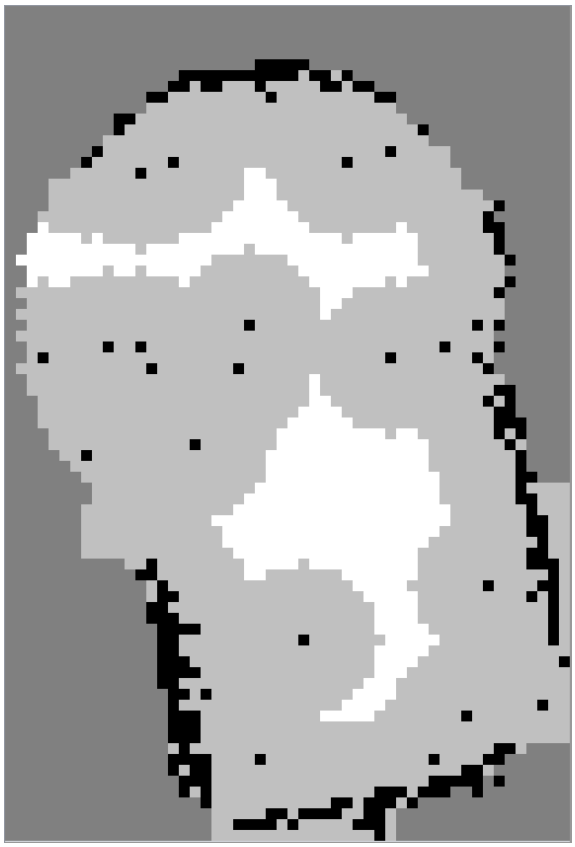


Figure 9: Mapping of maze

It was later discovered that a loose connection to IR sensor 2 was may have been the root cause of the noisy generation of the map in figure 8 and figure 9. After properly connecting the IR sensor wire, another mapping test was conducted on the circular track. Results from mapping of the circular track can be seen in figure 10. However, noisy range-bearing measurements still proved to be an issue.

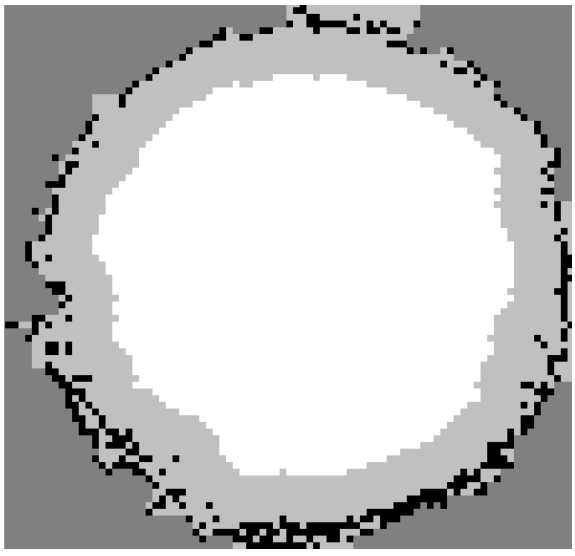


Figure 10: Mapping of maze

7 Theory

7.1 Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is a collection of method's which intend to estimate the relative position and orientation of a robot to its local surroundings, while at the same time generating a global map of the environment to place the robot in. In order for the robot to be able to sense its surroundings the use of exteroceptive sensors is necessary, typically IR-sensors, ultrasonic sensors, LIDARs, or cameras. Furthermore, odometry and inertial sensors are typically combined to estimate position and orientation (also known as *pose*) of the robot. A key idea behind SLAM is that the pose of the robot and the map correlate, meaning that for example bearing-distance measurments from IR-sensors have the ability to correct errors in odometry measurements from the wheel encoders of the robot (and vice versa). Without a map of the environment, or any other point of absolute reference, the robot will be left dead-reackoning making its pose estimate drift.

SLAM methods are commonly in use in autonomous systems where there does not exist an *a priori* map of the robot's surroundings and GNSS is not available due to the operating environment of the robot (e.g. indoors, mountainous areas, underwater). Even under circumstances where GNSS is commonly available, it is common to still use SLAM due to GNSS's main vulnerability, namely, the need for at least 3 satellites to function at all. Furthermore, the generation of a global map of the environment the robot ventures is advantageous as it may be used for path planning, obstacle avoidance and as a visual tool for human operators.

All SLAM architectures can be generalized to consisting of a front-end and back-end. The front-end component of the SLAM system consists of a feature-extraction module and a data association module. The front-end is responsible for pre-processing the gathered sensor data in order for the back-end to perform *maximum a posteriori* (MAP) estimation of the pose and map.

It is normal to categorize SLAM methods into two different categories depending on the type of map representation used, these are; feature-based SLAM and dense SLAM. In visual SLAM methods, i.e. where a camera is used as an exteroceptive sensor instead of IR measurements, feature-based and dense SLAM methods are commonly referred to as indirect and direct methods, respectively.

Using dense SLAM methods, the map of the environment (2D or 3D) is commonly represented using occupancy grids. For a 2D map, the occupancy grid could be a 2D-array of probabilities representing the existence of something occupying space in a specific location of the environment mapped to the grid. In its most simplistic form the grid could consist of entirely boolean values. The main advantage of using a dense SLAM method are that they can provide a lot more information than a feature-based SLAM method (for instance there is no need to categorize features, everything in the generated dense map would ideally be captured). The main restrictions of using a dense SLAM method is the large resource usage, especially memory consumption, for generating and updating large maps. Embedded devices are typically limited in CPU usage and memory, making dense SLAM in many cases infeasible.

In feature-based SLAM methods, the map of the environment is built by recognizing certain geometric features. Features can for instance be trees, corners, walls, or any other obstacle in the environment which can be recognized in order to relate the pose of the robot to the map. In feature-based SLAM applications the need for feature-extraction methods is needed when the positions of landmarks themselves are not readily available as measurements (a beacon broadcasting its position could serve as a direct measurement of the landmark). Feature-based SLAM methods are generally more suited for embedded systems than dense SLAM methods, due to that feature-based methods generally require less memory. However, since a feature-extraction scheme is needed for feature-based SLAM methods, one has to take into account that an extra pre-processing step has to be taken. Thus, depending on the scheme used, this may require more CPU usage than working directly with the IR measurements (dense). [5][6]

Using IR sensors as the exteroceptive sensor for SLAM applications, as is used in this project, is

not the most conventional area of use for IR sensors. Typically, IR sensors are used for obstacle detection and avoidance. This is mainly due to the IR sensor's limited range (< 1 m), and their dependence on the reflectance properties of objects in the scene. Due to their fast response times, and low-cost, the IR sensors are mostly used as binary proximity sensors. [3][4][14].

7.2 Feature extraction

The feature extraction step is the first pre-processing step in feature-based SLAM methods, and can be done in various ways. For indoor mobile robots navigating in environments where straight lines and edges make up the main features of the environment (structured environment), lines are well suited as features.

The IR sensor tower used for bearing-range measurements of the environment only has the ability to sample measurements from four radially-placed IR sensors at a time. Although, the sensor tower rotates, it will take approximately 10 seconds to cover a 360° scan of the environment. This sparsity of baring-range measurements has to be taken into account for feature-extraction in our case, thus, an incremental approach for gathering measurements has to be used.

7.2.1 Non-probabilistic incremental line extraction

Masehian et al. (2016) [15] proposes a line extraction algorithm for cooperating heterogeneous mobile robots for mapping of unknown environments, similar to the setup of robots used in this project. The line extraction procedure works by storing measurements of objects in a Cartesian global frame in a point buffer PB. Point measurements are transformed from the sensor frame $\{S_i\}$ to the body frame $\{B\}$ of the robot using (2), where $i = [1, ...4]$, d_i is the measured IR range to a point in the environment given in $\{S_i\}$, β_i is the rotation of $\{S_i\}$ with respect to $\{B\}$, Δx_i is the x-axis translation from $\{S_i\}$ to $\{B\}$, and Δy_i is the y-axis translation from $\{S_i\}$ to $\{B\}$ for IR sensor i.

$$\mathbf{P^B} = \begin{bmatrix} x^B \\ y^B \end{bmatrix} = \begin{bmatrix} d_i \cos(\beta_i) + \Delta x_i \\ d_i \sin(\beta_i) + \Delta y_i \end{bmatrix} \tag{2}$$

In order, to transform a measurement in $\{B\}$ to the global cartesian frame $\{G\}$ we use (3), where (x, y) is the position and θ is heading, with respect to $\{G\}$. The coordinate origin of $\{G\}$ is the initial position of the robot, with x-axis pointing out the front of the robot, and y-axis completing the right-handed frame of reference.

$$\mathbf{P^G} = \begin{bmatrix} x^B \cos(\theta) - y^B \sin(\theta) + x \\ x^B \sin(\theta) + y^B \cos(\theta) + y \end{bmatrix} \tag{3}$$

The points added to the point buffer are to be used for creating new line-segments. The creation of line-segments is carried out by connecting the first two points in the point buffer (as these measured points are expected to be close to each other), followed by successive checks for if the next point in the point buffer is collinear with the last two points. The three Cartesian coordinates (x_1, y_1), (x_2, y_2) and (x_3, y_3) are considered collinear if they satisfy (4), where ϵ is a threshold which is set based upon how close the three points are to forming a straight line-segment between them.

$$|(y_1 - y_2)(x_1 - x_3) - (y_1 - y_3)(x_1 - x_2)| \leq \epsilon \tag{4}$$

When three points pass the collinearity check, the algorithm will proceed by checking if the next point in the point buffer is collinear with the two proceeding points in the buffer. In this fashion, the line segment successively appendes new collinear points, stretching out the line segment, until a new point does not satisfy the collinearity threshold. When a new point is not collinear with the proceeding two points, the line segment generated up until the last point is stored in a separate

line buffer *LB*. A new line segment is formed starting from the last point in the point buffer which did not satisfy (4). For every collinearity check of a point, the point will be removed from the point buffer. The procedure of iteratively checking for collinearity among the points in the point buffer and generation of line segments is carried on until the point buffer is empty.

After lines have been extracted and stored in the line buffer, the lines will be merged with similar lines and stored in a *line repository*. In the first iteration, the line repository will be empty, thus, all lines are added directly in to the line repository. In the next iteration of of adding/merging lines to the line repository, each line in the line buffer will be merged with an existing line in the line repository if the lines are deemed *mergeable*. If the lines are not meargable, the line in the line buffer will simply be added to the line repository.

Two lines can be considered mergeable if (5) is satisfied, where δ is a threshold for the distance between each endpoints of the line segments $(\mathbf{p_1}, \mathbf{q_1})$, $(\mathbf{p_2}, \mathbf{q_2})$, and μ is a threshold for the difference between the slopes m_1, m_2, of each line. $d(\mathbf{p}, \mathbf{q})$ denotes the Euclidean distance between the points $\mathbf{p}$ and $\mathbf{q}$.

$$(|m_1 - m_2| \leq \mu) \wedge (d(\mathbf{p_1}, \mathbf{p_2}) \leq \delta) \wedge (d(\mathbf{p_1}, \mathbf{q_2}) \leq \delta) \wedge (d(\mathbf{q_1}, \mathbf{p_2}) \leq \delta) \wedge (d(\mathbf{q_1}, \mathbf{q_2}) \leq \delta) \tag{5}$$

[15] states that the condition for merging lines in (5) is used in simulations with no sensing or positioning errors. I.e. the thresholds δ and μ are used only for floating point rounding errors. Under non-ideal circumstances with noisy measurements and position estimates, [15] proposes to use *fuzzy membership functions* for determining the mergeability of line segments.

For a set X consisting of elements x, a *fuzzy set* A in X is defined by a membership function $f_A(x)$ which maps every element in X to a real number between 0 and 1 [22]. The value of $f_A(x)$ expresses the degree of membership of x in A. Similarly to probability distributions, they represent a *degree of truth*, however, the value of a fuzzy membership function does not represent a probability. Thus, mergeability of two line segments can be determined based on the condition given in (6) using the fuzzy membership functions in figure 11.

$$min(\mu_1, \mu_2, \mu_3, \mu_4) \geq \gamma \tag{6}$$

In equation (6), μ_1 is the value of a fuzzy membership function which takes as input the difference in angle of the slopes of two line segments. The value of μ_2 is a function of the maximum euclidean distance of each line-segment's midpoint to a point on the other line-segment. The value of μ_3 is a function of the distance between the midpoints of the two line-segments, and μ_4 is the value of a function which takes as input the minimum distance from the endpoint of a line-segment to a point on the on the other line-segment. The parameters $a, ..., i$ of the fuzzy membership functions in figure 11 are used to define the criteria for similar line segments.

7.2.2 Probabilistic incremental line-extraction

A probabilistic (stochastic) approach to line extraction has the ability to provide an accurate method to fit a line segment to a set of points from exteroceptive sensor measurements embedded in noise by formulating noise models for the sensors. Using these approaches one is able to both fit a line to the uncertain points from the environment according to its uncertainty, and provide information on the uncertainty of a given line segment by propagating the uncertainty.

A general measurement model for range-bearing measurements is outlined by Pfister et al. (2003) [16]. They propose to model sensor noise with additive zero-mean Gaussian random variables as follows:

$$\mathbf{u} = (\mathcal{D} + \epsilon_d) \begin{bmatrix} cos(\theta + \epsilon_\theta) \\ sin(\theta + \epsilon_\theta) \end{bmatrix} \tag{7a}$$

$$\epsilon_d \sim \mathcal{N}(0, \sigma_d^2) \tag{7b}$$

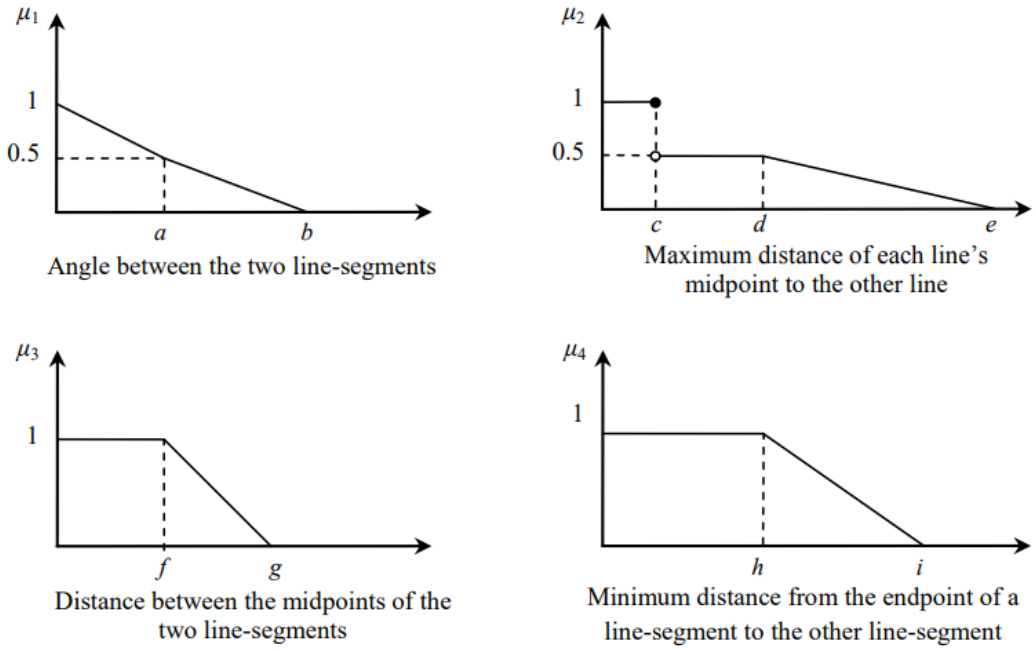


Figure 11: Fuzzy membership functions for merging two line segments [15, p. 211]

$$\epsilon_\theta \sim \mathcal{N}(0, \sigma_\theta^2) \tag{7c}$$

Here u is the measured coordinate of a point in the environment given in the robot's body frame, $\mathcal{D}$ is the true range to the point, and θ is the true bearing of the coordinate in relation to the robot. ϵ_d is the additive noise term to the distance component of the measurement with variance σ_d^2, and ϵ_θ is the additive noise term to the bearing component with variance σ_θ^2. For IR range measurements specifically, one could also include reflection coefficients into the sensor model using the photometry inverse square law such as proposed by Benet et al. (2008)[4]. Once measurement prediction models are established one may formulate the weighted line fitting problem using a maximum likelihood formulation. Generally, this boils down to solving a nonlinear least-squares problem which can be solved using optimization techniques such as Gauss-Newton or Levenberg-Marquardt. This is the main component of the line-extraction method proposed by Beevers et al. (2006)[3], who similarly to the robots used in this project only have sparse IR range-bearing measurements available. For implementation of such an approach on the nRF52840-based robots it would be beneficial to include the *GNU Scientific Library* (GSL) which provides readily available non-linear solvers in C [11].

8 Implementation of feature extraction algorithm on robot

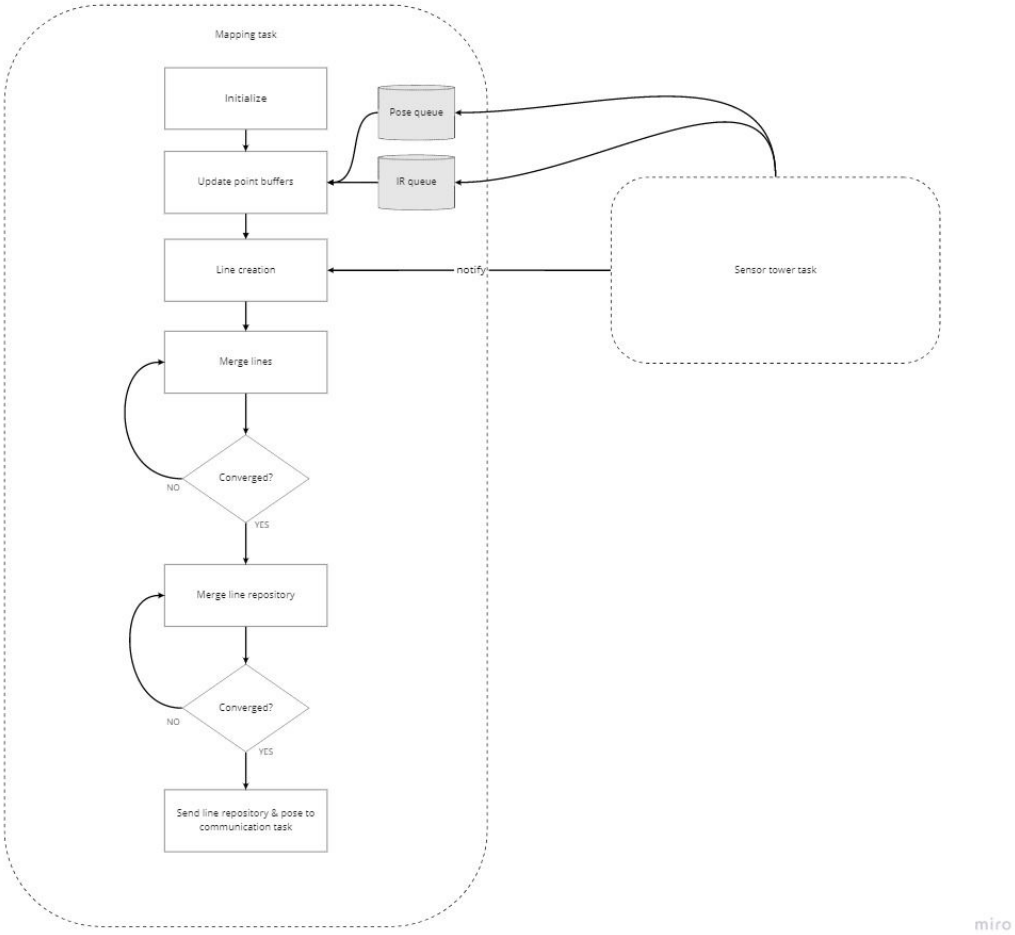


Figure 12: Overview of line-extraction implementation

The line-extraction algorithm is based on the implementation proposed by [15] outlined in Section 7.2.1, and continues the work of [7], who implemented the line-extraction algorithm on an NXT-lego robot as part of his work on the SLAM-project for his master thesis. Segger Embedded Studio was used for software development [17], and the initial project setup was configured by Lindefjeld [14].

The overall software architecture of the implemented feature-extraction scheme is visualized in figure 12.

The line-extraction procedure was implemented in a new FreeRTOS task, referred to as the *mapping task*. The algorithm consists of the four main steps:

- Whenever a new pose estimate and IR measurement is available a point buffer per IR sensor will be populated with the global cartesian coordinates derived from the bearing-range measurement to an obstacle.
- Line creation is initiated whenever a point buffer becomes full, the servo motor steering the rotation direction of the sensor tower changes rotation direction, or the robot's movement has changed (as indicated by the "notify-arrow" in figure 12). Successive checks for collinear coordinates is done separately over each point buffer. Collinear coordinates are populated as line segments into their corresponding line buffer.

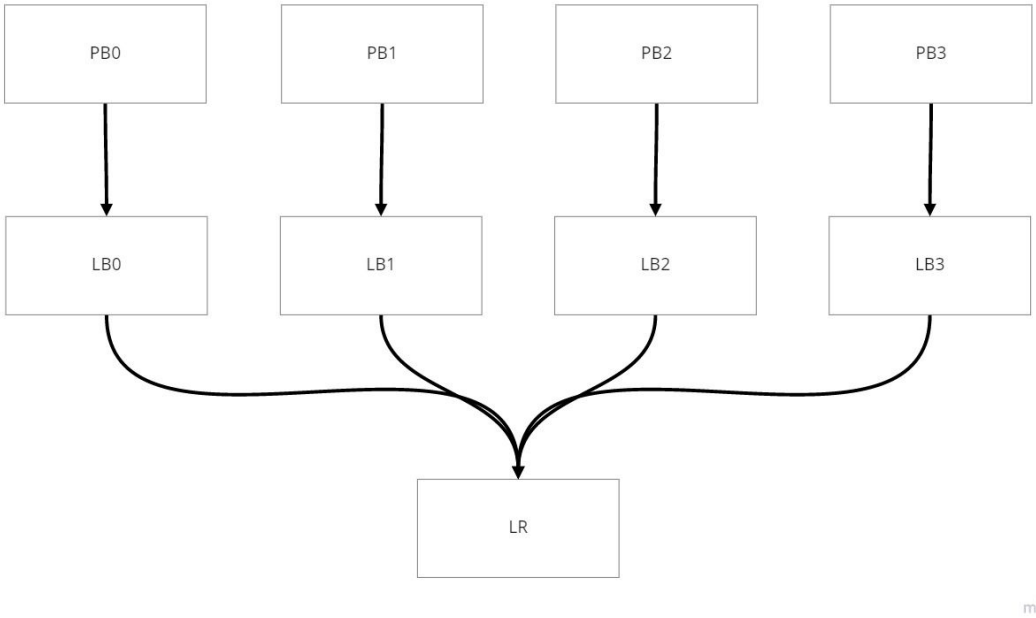


Figure 13: Flow of data between point buffers (PB), line buffers (LB) and the line repository (LR).

- For each line buffer, similar line segments are merged together. The merged lines from all line buffers are then appended to a common line repository.
- The line repository is merged with itself until no more lines are able to be merged. Ideally, the line repository should now hold a local map of the environment represented as line segments.

The flow of how range-bearing measurements are processed is illustrated in figure 13. The figure explains the interaction between point buffers, line buffers and the line repository.

8.1 Task initialization

The mapping task is created with the FreeRTOS API function `xTaskCreate()`, with the lowest task priority. In order to determine the required stack depth of the task, the `vApplicationStackOverflowHook()` utility-function was used for stack overflow detection along with `uxTaskGetStackHighWaterMark()` for measuring the largest consumption of stack at any time [8]. The required stack depth of the mapping task was found to be approximately 4kB.

On initialization of the mapping task, each point buffer is allocated a fixed size of 800 bytes on the heap, enabling a point buffer to maximally store 100 floating point coordinates of reflection points. Each line buffer is statically allocated 1600 bytes in order to store 100 line segments. The line repository is initially allocated to store 100 line segments as well, however, will be expanded if maximum size is reached.

8.2 Updating point buffers

It is necessary to use a point buffer per IR sensor due to the sparseness of range-bearing measurements from each *"scan"*. In the proposed implementation by [15] the robots were equipped with laser rangefinders which would provide the robots with much denser range-bearing measurements than with the configuration of the four IR sensors mounted on top of a rotating tower used in this case. Therefore, the robot must incrementally build a full scan of the environment as the tower turns. This in turn makes it necessary to buffer the Cartesian coordinates derived from

the range-bearing measurements in separate buffers depending on the source of the measurement since we only expect to be able to detect a line-segment from measurements at consecutive time increments from the same IR sensor. This will make line creation in the next step a lot easier.

The queues for pose estimates and IR range-bearing measurements are implemented using the FreeRTOS API for queue management [9]. Since the pose of the robot is only relevant for line extraction if there exists a range-bearing measurement, the pose and IR queues are populated using the `xQueueSend()` API function from the sensor tower task. If a queue becomes full or the range-bearing measurement does not fall within its valid measurement range (threshold is set in `robot_config.h`), the pose and range-bearing measurements are discarded. The pose estimates of the robot are derived from an extended Kalman filter implemented in the estimator task. Each element of the IR sensor queue consists of four range-bearing measurements, corresponding to one *"scan"* of the environment using the four IR sensors mounted on top of the rotating sensor tower.

Using the queues as described above ensures that the pose and IR queue are always of same length and that the estimated pose with the corresponding range-bearing measurement line up in the queues. Furthermore, this removes the hard real-time constraints of the mapping task from the rest of the robot application, i.e. the mapping task is not dependent on receiving relating data from different tasks at different rates, and can process the data at its own frequency.

The mapping thread will hault execution until both the pose and IR sensor queue are populated. Data is extracted from the queues using the API function `xQueueReceive()` in a FIFO manner before appending the Cartesian coordinates of a measured point in the environment to its corresponding point buffer.

8.3 Line creation

A line segment $\mathbf{L_i}$, is parametrized by a pair of global Cartesian coordinates, $\mathbf{p_i} = \begin{bmatrix} x_p & y_p \end{bmatrix}^T$ and $\mathbf{q_i} = \begin{bmatrix} x_q & y_q \end{bmatrix}^T$, representing the start and endpoint of the line segment. Each line segment is stored as a data type of x bytes.

In order for the line merging procedure to work satisfactory it is important that line creation is initiated at the right moment. Line creation will only take place when the pose and IR sensor measurements queues have been populated with measurements, and one of the following events have taken place:

1. *The servo motor rotating the four IR sensors has reached its maximum rotation angle of* $90°$*, and therefore changes rotation direction.* At this point we would expect each point buffer to be filled with 90 range-bearing measurements each. The sensor tower only rotates with respect to the body frame of the robot whilst the robot is stationary. Therefore, if there exists an object blocking the path of the IR beams, under ideal conditions we would expect that the straight lines of that object facing the direction of the single IR sensor would be measured as a set of collinear Cartesian coordinates in the world frame. In this way we may iterate over all points in a point buffer corresponding to the respective IR sensor, whilst appending new endpoints to the line segment if the point passes the collinearity check. If a point under consideration of being the new endpoint of a line segment fails the test for collinearity with the current endpoint of the line segment, then Masehian et al. proposes that the point is used as the start point of a new line segment. In this way a $90°$ sweep with one IR sensor one could expect that all sets of collinear points in the point buffer would correspond to line segments capturing the geometry of the object (wall) in the environment with a minimal number of points.

2. *The robot's movement state has changed.* The robot will at any point in time be in one of the following states: `STOP`, `FORWARD`, `BACKWARD`, `CLOCKWISE`, `COUNTER_CLOCKWISE`, indicating whether the robot is currently not moving, moving forwards or backwards, or conducting either a clockwise or counter clockwise pinpoint turn. The states discretely characterize the current movement of the robot, and are used in the sensor tower task to decide when to rotate the tower and to notify the mapping task that the robot has changed state. It is beneficial

to start line segment creation when the robot's movement has changed since we expect that Cartesian points derived from a particular IR sensor's measurements do not form a collinear line segment when the robot transitions from a translating to a rotating state. In transitions from `FORWARD` to `BACKWARD` (and *vice versa*) and `CLOCKWISE` to `COUNTER_CLOCKWISE` (and *vice versa*), the robot will have already scanned over the same area in the environment. Thus, following the same argumentation as in point 1) we would ideally have enough information about the environment to capture its geometry in form of line segments.

3. *A point buffer has reached its maximum size.* A common case where a (or more) point buffers may reach its maximum size is when travelling along a long hallway. At some point in time line segment creation should begin, i.e. starting extraction of points from the point buffers looking for collinear points. Without this event for triggering line segment creation, it would take a long time for line segments to be generated for a robot travelling down the long hallway, and one would have to allocate memory for the point buffers dynamically.

Notifications for starting line segment creation are implemented using the FreeRTOS inter-communication and synchronization API functions `xTaskNotify()` and `ulTaskNotifyTake()` . Task notifications are used for sending events directly from a task to another, instead of indirectly via intermediary data types such as queues, and is a more RAM effective data type than counting semaphores [10]. For each iteration of the main loop in the mapping task, the mapping task will check for notifications from the sensor tower triggered by the events above. The mapping task will not hault execution to wait for a notification in order to continue populating the point buffers. In the case where a point buffer has reached its maximum size, the mapping task will notify itself.

The extraction of line segments is done by iterating over all points in one point buffer at a time. For each point with index `i` in the point buffer, the point will be checked if it is collinear with the point at index `i+1` and `i+2`. If the distance between the collinear points also is smaller than a threshold `D`, then the line creation step has found a valid line segment from the range-bearing measurements. Furthermore, when a valid line segment has been found, the line creation step will continue the search for collinear points that are close enough together, with the intention of finding the longest line segment possible. I.e., the procedure will check if the next point in the point buffer at index `i+2+j` ($j >= 1$) is collinear with points `i` and `i+1` until either the next point is not found to be collinear with the first two points or the distance between the points has become too large. Here we have exploited that each point buffer's points have been stored in order with respect to the timestamp of the range-bearing measurement. Thus, one will expect the points to line-up in the global coordinate frame if there is an object in the environment with distinct line features (e.g. a wall). Futhermore, following the argumentation of [4], we may assume that if a set of collinear points are found to be close enough together, then these points probably correspond to the same reflection surface. Extracted line segments derived from a specific point buffer are then appended to the IR sensor's corresponding line buffer. Pseudo code for the line creation step is outlined in algorithm 1.

An important modification to the proposed initial line creation procedure by Masehian et al. is to disregard their assumption regarding creation of a new line segment after a collinearity check with a new point in a point buffer has failed. They have assumed that since the collinearity check with the next point in the buffer failed this should imply that a new line segment should be created starting from the point that did not pass the collinearity check. And, therefore, there is no need to check for collinear points among the points that already have been iterated over. This assumption would have been valid under ideal conditions with zero noise. However, the noisy range-bearing measurements, makes it necessary to search for the longest chain of collinear points for each two consecutive points in the point buffer. This has to be done in order for line-extraction to work satisfactory, because, we have no way of determining whether the initial three collinear points are trustworthy starting points for line segment creation. The line segment creation step therefore relies on the following merging steps to average out any line segments that were created with points that in a probabilistic framework would be considered as outliers. Although this modification of the line creation step will increase this step's worst-case run-time from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$ (where n is the number of points in a point buffer), it is still better than the naive approach of finding all collinear points in each point buffer which would take $\mathcal{O}(n^3)$. The $\mathcal{O}(n^2)$ run-time is enabled by the sequential buffering of points in the point buffer.

Algorithm 1 Initial line segment creation

```
1: procedure createLines(pointBuffer, lineBuffer)
2:     remove line segments from lineBuffer
3:     i ← 0
4:     l ← length of pointBuffer
5:     if l < 3 then
6:         remove line segments from pointBuffer                ▷ Not enough points
7:         return
8:     end if
9:     while i < l − 2 do
10:        A ← pointBuffer[i]
11:        B ← pointBuffer[i+1]
12:        C ← pointBuffer[i+2]
13:        j ← 0
14:        while dist(A, B) < D & dist(B,C) < D & collinear(A,B,C) do
15:            j++
16:            C ← pointBuffer[i+2+j]
17:        end while
18:        if j > 0 then
19:            newLine ← line from point A to point C
20:            insert newLine into lineBuffer
21:        end if
22:        i ← i + 1
23:    end while
24:    remove points from pointBuffer
25: end procedure
```

8.4 Line merging

At this step in the line-extraction algorithm, the line buffers have been populated with initial line segments created in the previous step. These line segments will generally be small and partially overlapping. Therefore, in order to improve accuracy and obtain a minimal representation of the environment to reduce memory footprint we have to perform a step where we merge lines together. I.e. this process will "stitch together" the line segments contained in the line buffers, and append the resulting lines to a line repository. The line repository is the data structure which holds the map of the environment.

There exists various approaches to how one can merge two line segments. Since [15] does not explicitly state how line merging should be conducted, the implemented line merging procedure is based on the proposed method by [20]. The line merging scheme works as follows: Let $\mathbf{L_1} = \begin{bmatrix}\mathbf{p_1 q_1}\end{bmatrix}^T$ and $\mathbf{L_2} = \begin{bmatrix}\mathbf{p_2 q_2}\end{bmatrix}^T$ be two line segments which we want to merge. We then define the origin of a frame $\{M\}$ as the centroid of the four endpoints weighted by each line segment's length using (8)

$$\begin{bmatrix} x^M \\ y^M \end{bmatrix} = \begin{bmatrix} \frac{l_1(x_{p1}+x_{q1})+l_2(x_{p2}+x_{q2})}{2(l_1+l_2)} \\ \frac{l_1(y_{p1}+y_{q1})+l_2(y_{p2}+y_{q2})}{2(l_1+l_2)} \end{bmatrix} \tag{8}$$

where $l_i = \sqrt{(x_{pi} - x_{qi})^2 + (y_{pi} - y_{qi})^2}$ is the length of $\mathbf{L_i}$. We then must find the orientation of each line segment with respect to the x-axis of the global frame $\{G\}$ using (9)

$$\theta_i = \text{atan}(\frac{y_{qi} - y_{pi}}{x_{qi} - x_{pi}}) \tag{9}$$

We can then define the orientation of the merged line as the line-segment-length-weighted-sum of orientations of the two line segments using (10)

$$\theta_M = \frac{l_1\theta_1 + l_2\theta_2}{l_1 + l_2} \tag{10}$$

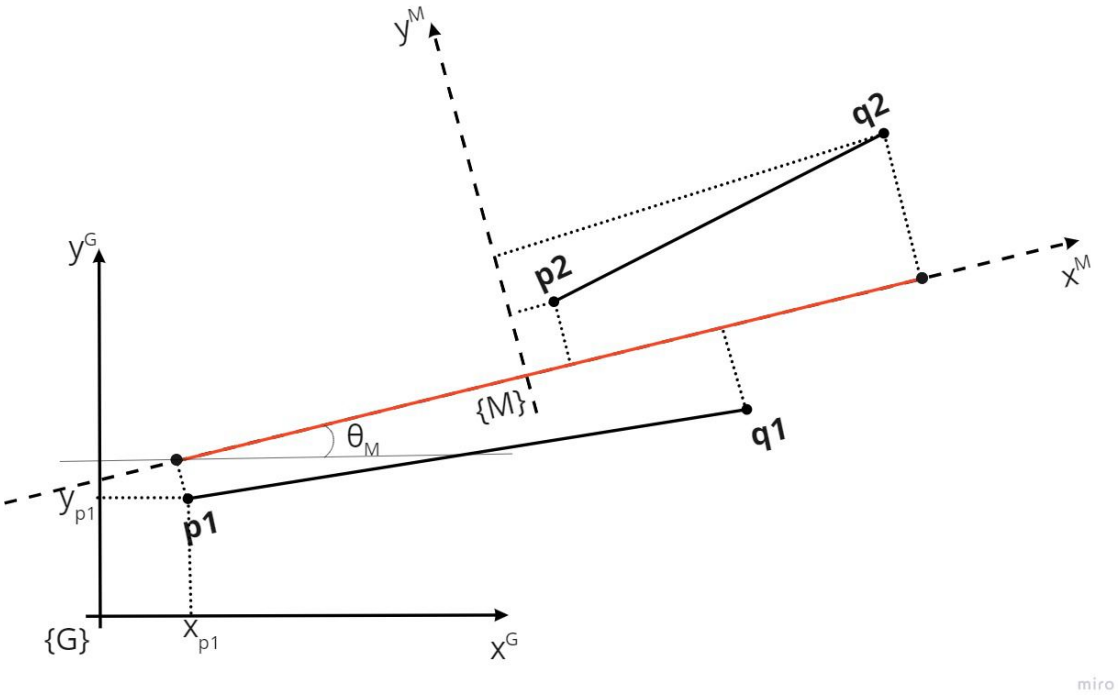


Figure 14: Merging two line segments.

The frame $\{M\}$ has its x-axis parallel to the direction of θ_M. We can then define the coordinate endpoints $\delta_\mathbf{i}$ of each line segment in the frame $\{M\}$ by applying the transformation in (11)

$$\begin{bmatrix} x_{\delta i}^M \\ y_{\delta i}^M \end{bmatrix} = \begin{bmatrix} (y_{\delta i} - y^M)\sin(\theta_M) + (x_{\delta i} - x^M)\cos(\theta_M) \\ (y_{\delta i} - y^M)\cos(\theta_M) - (x_{\delta i} - x^M)\sin(\theta_M) \end{bmatrix} \tag{11}$$

The two orthogonal projections over the x-axis of the $\{M\}$ frame of the endpoints δ_i which are furthest apart are then defined to be the endpoints of the merged line. We can then transform the endpoints back into the global reference frame. Pseudo code for the described merging scheme is given in algorithm 2.

Note that the length of each line segment is used to weight the influence that line segment has on the resulting merged line. Given our assumption from the initial line creation step - many collinear points close to each other should be considered as a valid line segment - weighting the merged line by the length of each input line segment will implicitly make the merged line be influenced the most by the line segment that passes through the most collinear points. In this way, the merging process has the ability to "average out" the influence of noisy range-bearing measurements.

In figure 14 the merging process is illustrated. The red line segment is the result of a merge between the line segments $\mathbf{L_1}$ and $\mathbf{L_2}$.

The line merging algorithm will generate a new line from two lines that satisfy the conditions for merging as outlined in Section 7.2.1. The mapping task will iterate through all lines in each of the four line buffers and merge lines in the same line buffer that are eligible for merging. The implemented algorithm for merging lines in the same line buffer is done with a *greedy recursive* approach, and is outlined with pseudo code in algorithm 3. This is a recursive process since line segments that have already been merged will be checked if they are mergeable with other line segments, and are merged if they are found to be mergeable with the next line stored in the line buffer. We call it a greedy approach since the algorithm will not look for lines that might be better to merge other than the first mergeable line it finds. After iterating through all four line buffers, and it is not possible to find any more merges, the line buffers should contain longer line segments that are more representative of the surrounding environment of the robot.

Algorithm 2 Merge two line segments

procedure MERGE(A, B) ▷ A, B are the lines to be merged
  $l1 \leftarrow |A|$
  $l2 \leftarrow |B|$
  P1, Q1 ← coordinates of endpoints for line segment A
  P2, Q2 ← coordinates of endpoints for line segment B
  xm $\leftarrow (l1 * (\texttt{P1.x} + \texttt{Q1.x}) + l2 * (\texttt{P2.x} + \texttt{Q2.x}))/(2 * (l1 + l2))$ ▷ Origin of {M} (merge) frame is (xm, ym)
  ym $\leftarrow (l1 * (\texttt{P1.y} + \texttt{Q1.y}) + l2 * (\texttt{P2.y} + \texttt{Q2.y}))/(2 * (l1 + l2))$
  $\theta_A \leftarrow atan((\texttt{Q1.y} - \texttt{P1.y})/(\texttt{Q1.x} - \texttt{P1.x}))$
  $\theta_B \leftarrow atan((\texttt{Q2.y} - \texttt{P2.y})/(\texttt{Q2.x} - \texttt{P2.x}))$
  if $|\theta_A - \theta_B| < \pi/2$ **then** ▷ Orientation of merged line
    $\theta_M \leftarrow (l1 * \theta_A + l2 * \theta_B)/(l1 + l2)$
  else
    $\theta_M \leftarrow (l1 * \theta_A + l2 * (\theta_B \pi * \theta_B/|\theta_B|))/(l1 + l2)$
  end if
  P1 ← transform(P1, (xm, ym), θ_M) ▷ Determine coordinates of endpoints in {M}
  Q1 ← transform(Q1, (xm, ym), θ_M)
  P2 ← transform(P2, (xm, ym), θ_M)
  Q2 ← transform(Q2, (xm, ym), θ_M)
  list ← [(P1.x, 0), (Q1.x, 0), (P2.x, 0), (Q2.x, 0)] ▷ List of coordinates of the orthogonal projections onto the x-axis of the {M} frame
  i ← 0
  while i < 4 **do** ▷ Find largest distance between endpoints
    j ← i +1
    while j < 4 **do**
      d = distanceBetween(list[i], list[j)
      if d > largestDist **then**
        largestDist ← d
        start, end ← list[i], list[j]
      end if
      j ← j +1
    end while
    i ← i +1
  end while
  start ← invTransform(start, (xm, ym), θ_M) ▷ Transform back to {G} frame
  end ← invTransform(end, (xm, ym), θ_M)
  return line from start to end
end procedure

procedure TRANSFORM(point, origin, θ) ▷ Translate point to origin, then rotate by θ
  x ← (point.y - origin.y)*sin(θ) + (point.x - origin.x)*cos(θ)
  y ← (point.y - origin.y)*cos(θ) - (point.x - origin.x)*sin(θ)
  return (x, y)
end procedure

procedure INVTRANSFORM(point, delta, θ) ▷ Rotate point by θ, then translate by delta
  x ← point.x*cos(θ) - point.y*sin(θ) + delta.x
  y ← point.x*sin(θ) + point.y*cos(θ) + delta.y
  return (x, y)
end procedure

Algorithm 3 Recursive algorithm for merging line segments in a line buffer

```
1: procedure MERGELINES(lineBuffer)                    ▷ lineBuffer list of line segments
2:     i ← 0
3:     l ← length of lineBuffer
4:     while i < l − 1 do                              ▷ Greedy search for mergeable lines
5:         line ← lineBuffer[i]
6:         nextLine ← lineBuffer[i+1]
7:         if mergeable(line, nextLine) then
8:             mergedLine ← merge(line, nextLine)      ▷ Tavares et al. (1995)
9:             remove line and nextLine from lineBuffer
10:            insert mergedLine into lineBuffer
11:            return mergeLines(lineBuffer)
12:        end if
13:        i ← i + 1
14:    end while
15:    return lineBuffer                               ▷ Contains merged lines
16: end procedure
```

8.5 Line repository merging

After having merged together all lines in the same line buffer in the previous step it is time to combine all lines from each line buffer into a common line repository, serving as the map of the environment.

The mapping task will use the recursive merging approach outlined above. However, instead of merging lines in the same line buffer, the line segments in a line buffer will be merged with line segments already stored in the line repository. This is done by appending all line segments in each line buffer to the common line repository, before applying algorithm 3 to the line repository. In this way, the line repository will continuously be updated as the robot explores the environment. One may optionally, randomly shuffle the line segments in the repository, and then apply the procedure for merging until the procedure has been applied a set amount of times without reducing the number of segments in the repository. This would lower the risk of over-segmentation, however, during testing the reshuffling of line segments did not have a large effect on the resulting map.

If a connection is established with the server, one can send all updated line segments in the line repository to the server after the recursive merging algorithm is not able to merge any more line segments. On the server-side, received updated line-segments from a robot may be merged using the same last steps of updating a line repository as is used on the robot. In this way, continous updates of the line repository will result in time result in a more accurate representation of the environment in which the robots are exploring.

9 Communication problems with server

A lot of time during working with the project was related to trying to get a reliable connection established to the Java server in order to display the extracted line segments from the robot whilst moving. However, this was not successfully achieved. This section will provide a discussion on possible answers to the reason why the communication with the server is not working as intended, and solutions that were implemented to try to fix the problem.

As commented in Section 6, during initial testing of the system, the robot would abruptly disconnect the connection with the Java server given enough time. The most common error seen in the debugging terminal of the robot application after a program failure was: `<error> app: ERROR 7 [NRF_ERROR_INVALID_PARAM] at C:\nRF5SDK1500a53641a\nRF5_SDK_15.0.0 _a53641a\examples\project\nRF52840dk_source_code\blecommunication\bluetooth.c:553 PC at: 0x00030B6F <error> app: End of error report`.

During implementation of the line extraction algorithm, points in the environment from the IR sensors and extracted lines were tried to be sent to the server for visualization. However, in this case the connection to the server was even more short-lived. It was therefore initially thought that a lack of synchronization was the cause of program failure. The robot communicates with the server over Bluetooth to an nRF52840 Bluetooth USB-dongle. Two different communication protocols have been implemented on top of Nordic Semiconductor's SDK for bluetooth communication by previous students. These two protocols are referred to as *simple protocol* and *Automatic Repeat Request* (ARQ). Simple protocol, is only used for defining the format of messages to the Java server, whilst ARQ also tries to provide a reliable connection with message delivery guarantees. ARQ was implemented with multi-threaded usage in mind, as synchronization primitives such as semaphores are used in order to ensure thread-safe usage when multiple tasks are sending messages to the server concurrently. Simple protocol on the other hand seems to lack synchronization entirely.

Using the current ARQ implementation over bluetooth however, does not work sufficiently well as the connection between the robot and server is lost after a while. From observed debug terminal outputs of the robot application and the server, it is evident that the server will send *"keepalive"* messages to the robot, and the robot will try to send an acknowledgement to the keepalive message. However, the server is not able to receive the acknowledgement, thus, eventually the connection will timeout. An effort to sniff the bluetooth packets from the robot was carried out by installing Nordic Semiconductor's bluetooth extension to the network protocol analyzer *Wireshark* (v. 3.6.0), however, the extension did not work. It is also possible that the current Java server does not comply with the timing requirements of the ARQ protocol, thus one could simply relax these requirements by increasing the period between the sending of keepalive messages. However, this is not a sustainable solution to the problem.

An effort to make simple protocol thread-safe was carried out unsuccessfully. Firstly, simple protocol messages were wrapped to enforce mutual exclusion on the internals of the bluetooth driver. This was done using the FreeRTOS inter-task communication mutexes, using `xSemaphoreCreateMutex()`, `xSemaphoreTake()`, and `xSemaphoreGive()`. After wrapping the already existing simple protocol functions with mutual exclusions, the communication problem was still apparent. An alternative solution which was not attempted is to try to make tasks publish their messages to an outgoing message queue/buffer. Instead of tasks sending messages to the server directly within the task, it would be the responsibility of the main communication task to send each message in the message queue to the server (in the current version of the robot application the sole responsibility of the main communication task is to listen for incoming messages from the server). In this way message priority could be handled by tasks either appending to the front or back of the message buffer.

10 Testing and verification

To test the implemented line-extraction algorithm outlined in Section 8 the robot was placed in an environment with three walls as depicted in figure 15. The problems related to server communication, explained in Section 9, made it difficult to examine the performance of the line-extraction scheme whilst the robot was moving. Line segments, measured points in the environment, and the robot's pose were sent over UART to the working station for plotting.

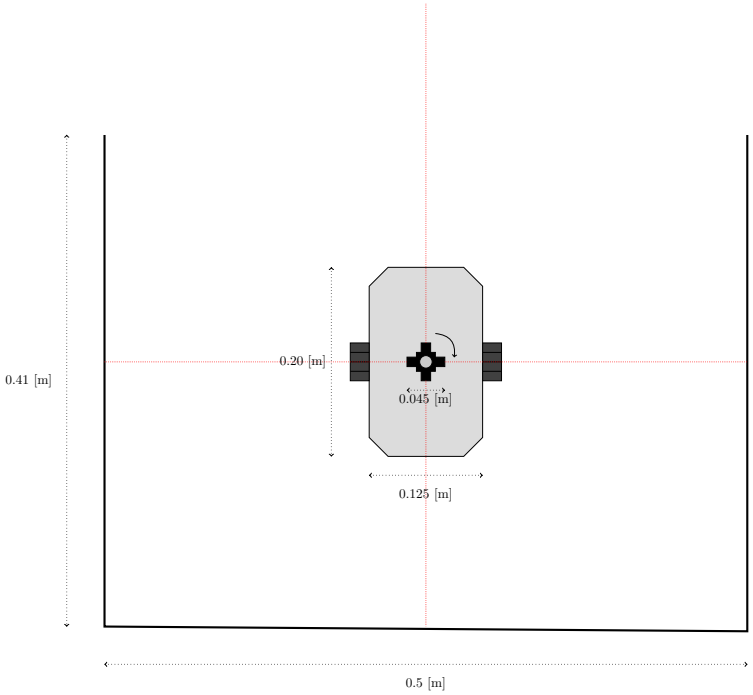


Figure 15: Environment for testing line-extraction algorithm

10.1 Line extraction *step-by-step*

To verify the implementation of the line-extraction algorithm, data was collected and visualized in each step of the process.

The first two steps of the algorithm consisted of updating point buffers with range-bearing measurements, followed by initial line-creation when either a point buffer became full, the IR sensor tower has changed direction, or the movement of the robot has changed. In figure 16, the line-segments of each line buffer are plotted along with the points in each point buffer. The resulting line segments in the line-buffer are a result of searching for collinear points. In order to reduce the influence of the noisy IR measurements (can especially be observed for IR sensor 2 in figure 16), collinear points must also be within a certain distance set by the `DISTANCE_BETWEEN` parameter. As can be seen in figure 16, most of the line segments in each of the line buffers are overlapping.

The third step consisted of merging lines in each individual line-buffer, followed by placing the resulting lines in a common line repository. Figure 17 shows the lines in each line buffer after the individual line buffer merge step.

In the fourth step, line segments in each of the line buffers are merged into the common line repository. This step is responsible for maintaining the current map of the robot's surroundings. Figure 18 illustrates the contents of the line repository incrementally over three "sweeps" of the IR sensor tower. In this way the reader can observe how the map has been updated over three steps.

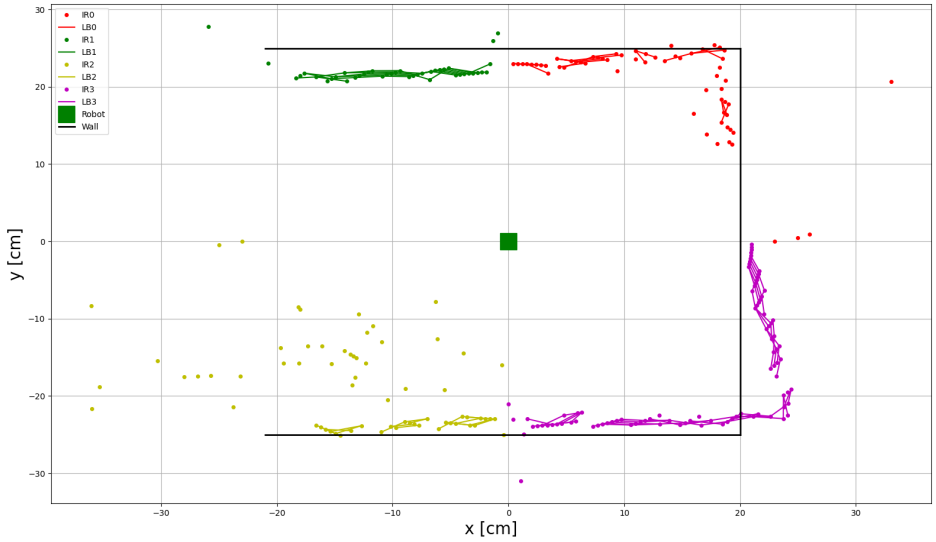


Figure 16: Line segments before merging in each line buffer after sensor tower has rotated 90°.

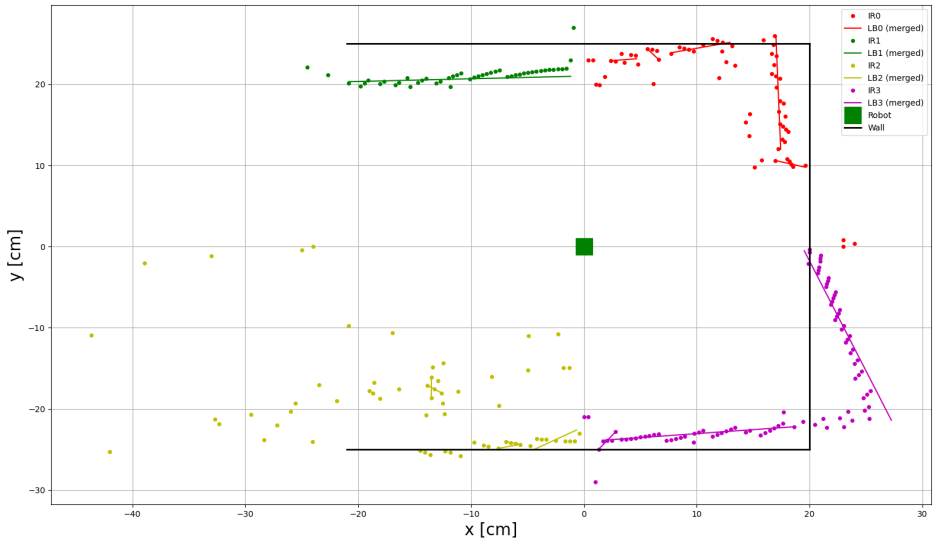
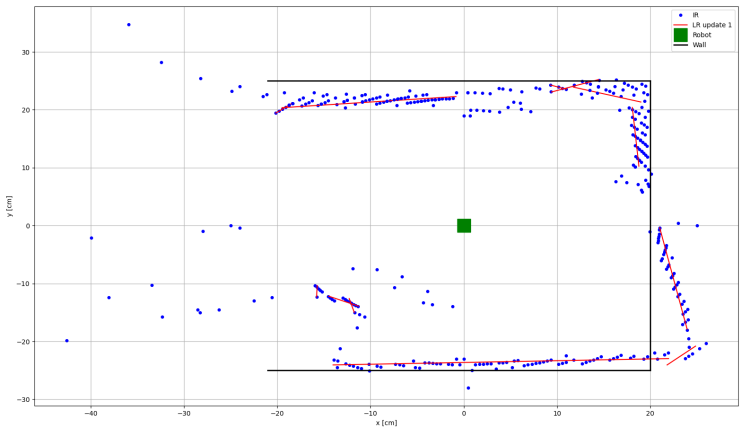


Figure 17: Merged line segments in each line buffer after sensor tower has rotated 90°.

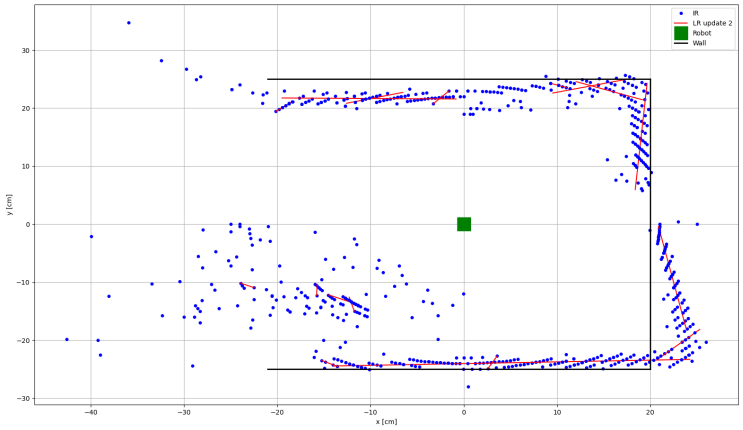
10.2 Tuning parameters for line-extraction

An experimental approach to tuning the parameters necessary for line extraction was conducted, and table 1 gives suggested values. The sections below discuss how these parameters were obtained. Figure 18 shows the extracted line segments after each line repository update with the parameters suggested parameters from table 1.

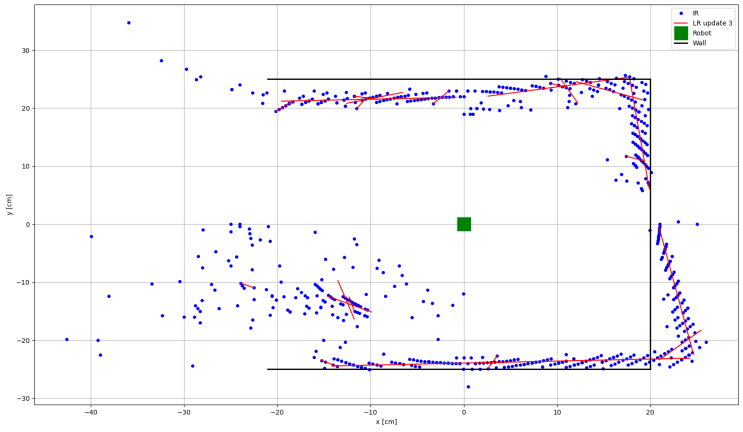
Note that optimal selection of parameters for line extraction is dependent on the environment itself, instead of only parameters inherent to the exteroceptive sensors, e.g. standard deviations of measurements. In other words, more detailed environments with many walls of similar angles, would require parameters tuned for that certain environment in order to map the environment



(a) Line repository update 1



(b) Line repository update 2



(c) Line repository update 3

Figure 18: Updating the line repository

Table 1: Suggested parameters for line extraction

Parameter	Value	Description
ϵ	5	Collinearity tolerance [cm].
D	2	Max. distance between collinear points [cm].
μ	0.5	Angle threshold between mergeable segments [rad].
δ	5	Max. distance between endpoints of mergeable segments [cm].

with sufficient accuracy. Environments with less details, such as the one used for testing the line extraction algorithm, will represent the map with a lot more line segments than necessary if tuned to work in a detailed environment, thus wasting memory resources.

10.2.1 Line creation thresholds

The parameters which are used in the initial line segment creation step are a collinear tolerance and a maximum distance between points that satisfy the condition for collinarity.

The collinear tolerance parameter, ϵ, sets the boundaries for which sets of three points are considered to be collinear. I.e. consider setting $\epsilon = 1$. This will make the initial line creation step consider three points, A, B, C to lie on the same line if the difference between the slope of the line AB and AC is ± 1. Figure 19 shows how ϵ influences line extraction. Firstly, a high collinear tolerance will extract too many lines from the range-bearing measurements, and makes line merging more expensive. This is especially illustrated in figure 20, where the lines in each line buffer are plotted before merging any line segments. Secondly, a low collinear tolerance will extract too few lines from the range-bearing measurements due to the stricter condition placed on extracting lines embedded in noisy range-bearing measurements.

Due to noisy range-bearing measurements a threshold distance between the collinear points was also needed. This provides a simple way of filtering outliers from the range-bearing measurements. From observations it was apparent that outliers might form a collinear line, however, the distance between the points would generally be larger than the distance between collinear points that accurately represent the robot's surroundings.

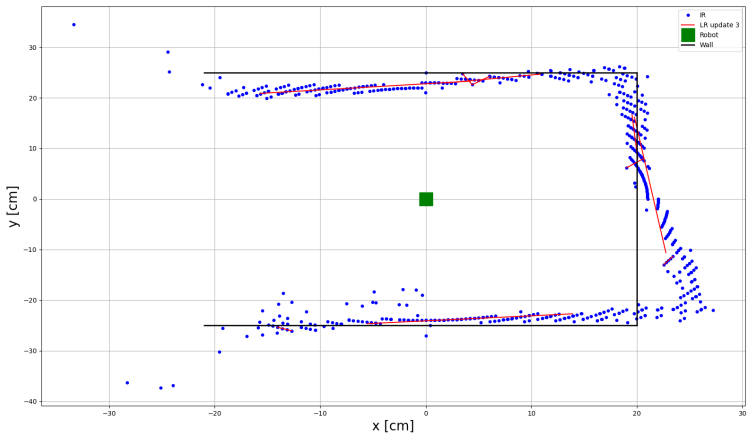
The parameter D, denoting the maximum distance between collinear points, should be set small enough in order for collinear range-bearing measurements corresponding to entirely different line features in the environment to not form a line segment. If D is set too large, line extraction will not be satisfactory as seen in figure 23b.

10.2.2 Merging thresholds

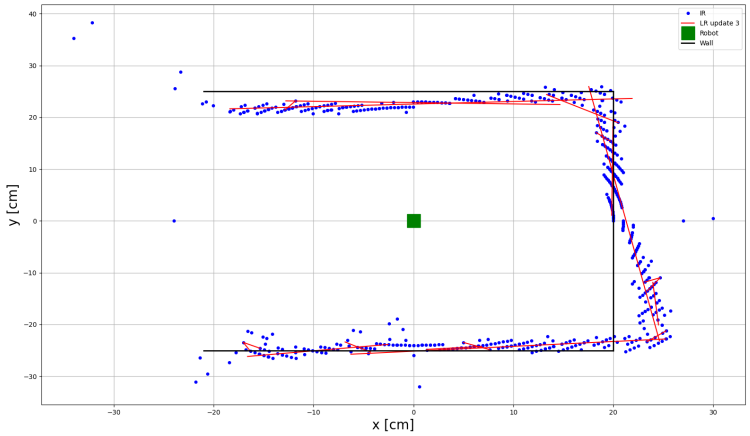
In Section 7.2.1 two approaches to how one can can determine if two lines should be merged was outlined. In practice, using *fuzzy membership functions* in order to determine the "mergeability" of two lines is not worth the effort of finding good choices for the constant parameters of the membership functions. If one where to use this approach which Masehian et al. (2016)[15] proposed for determining mergeable line segments for non-ideal measurements, one would have to set a total of nine parameters appropriately. If one instead, uses the approach which was proposed to be used under ideal conditions (zero noise), one is still able to obtain satisfactory results with only two (arguably more) intuitive parameters. Therefore, the similarity function for two line segments using fuzzy set theory was not used.

The mergeability of two lines is determined by the smallest angle between the line segments, and the distance between the endpoints of the line segments.

The first threshold μ is the maximum angle (in radians) between two line segments. Figure 22 shows how different values of μ affect the line extraction results. Setting μ too low will result in a line repository filled with many small line segments. By setting μ too large one will risk not being able to map the environment with enough detail since walls next to each other with similar angles



(a) Line repository after three updates with $\epsilon = 0.5$

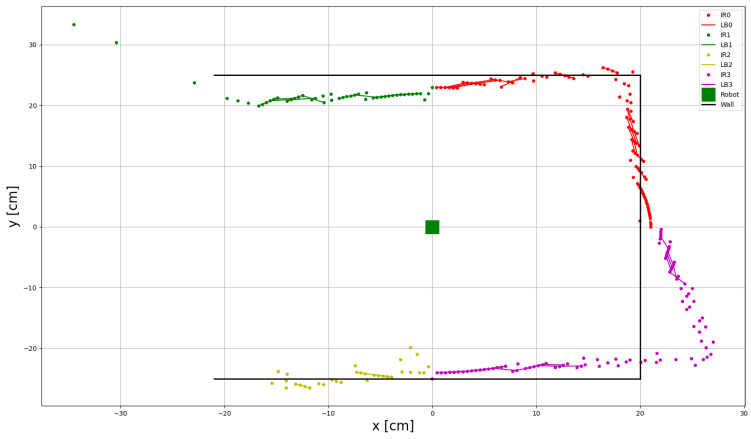


(b) Line repository after three updates with $\epsilon = 100$

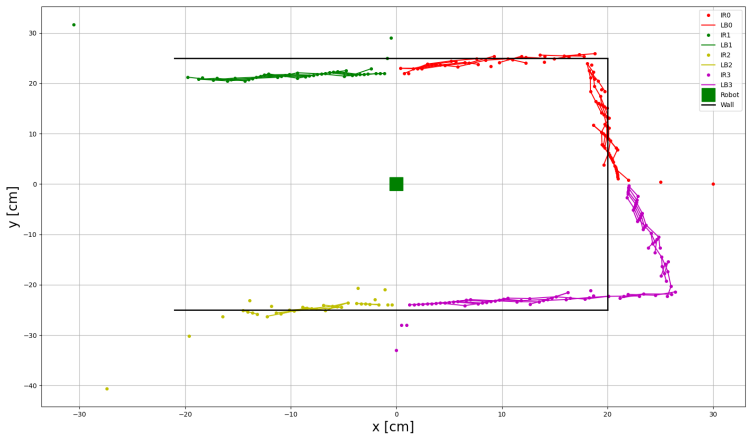
Figure 19: Line repository after three update steps with different collinear tolerances.

would be merged together to one line segment.

The second threshold δ sets the maximum distance between endpoints of mergeable line segments. Similarly, to the threshold μ, the choice of δ sets and upper bound on how detailed the resulting map of the environment can be made using the implemented line extraction procedure. Figure 23 shows the contents of the line repository after three update steps with varying values of δ.

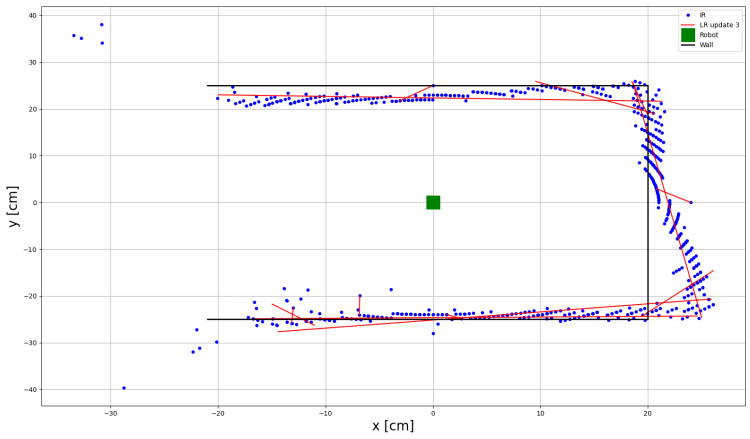


(a) Line buffers with $\epsilon = 0.5$ and $D = 2$

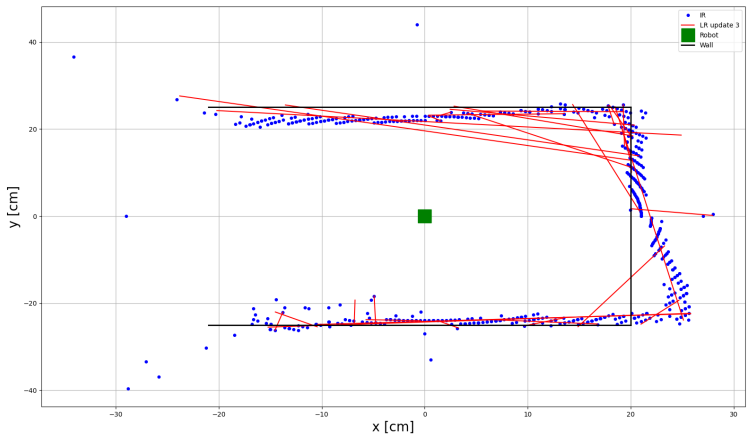


(b) Line buffers with $\epsilon = 100$ and $D = 2$

Figure 20: Contents of line buffers with different collinear tolerances

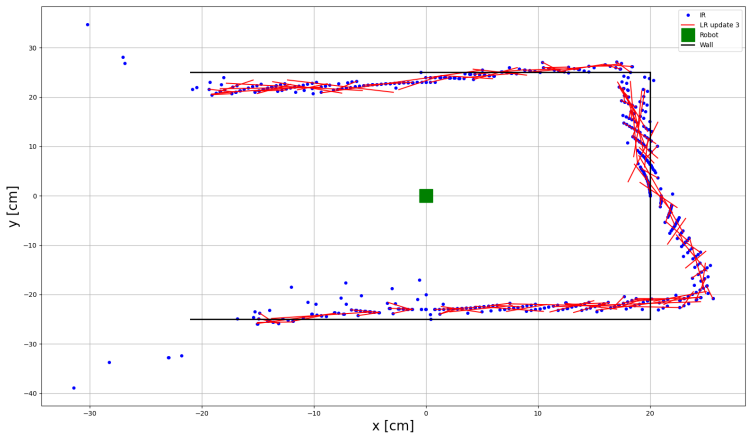


(a) Line repository after three updates with $\epsilon = 5$ and $D = 10$

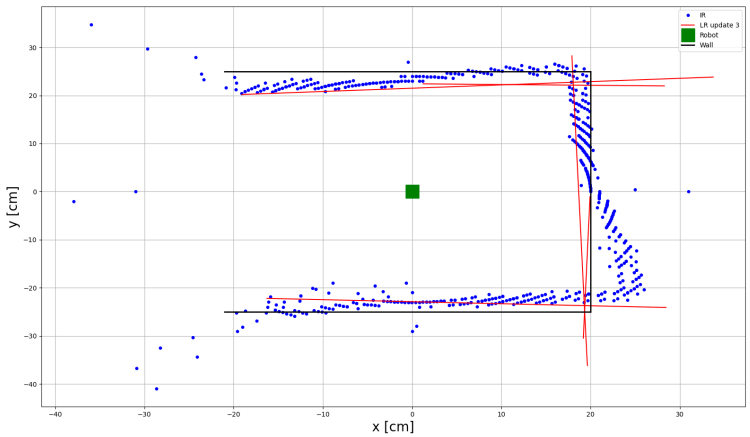


(b) Line repository after three updates with $\epsilon = 5$ and $D = 20$

Figure 21: Line repository with different maximum distance thresholds between collinear points.

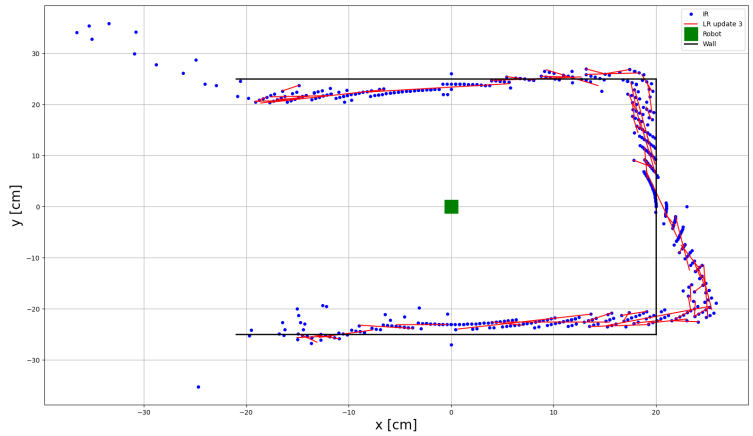


(a) Line repository after three updates with $\mu = 0.1$ and $\delta = 5$

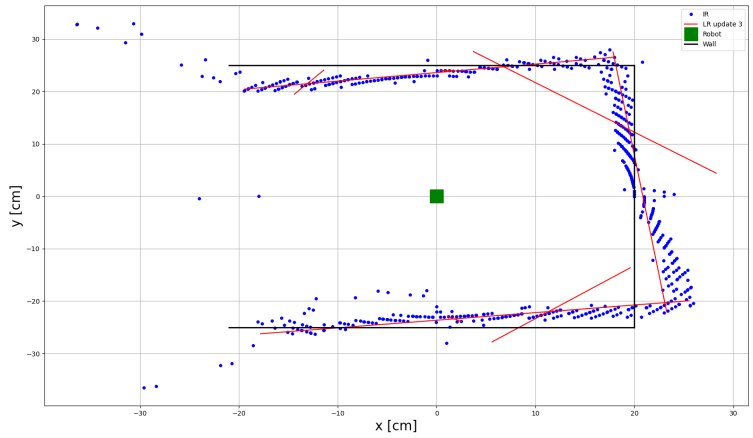


(b) Line repository after three updates with $\mu = 1.5$ and $\delta = 5$

Figure 22: Line repository with different maximum angle difference thresholds for merging.



(a) Line repository after three updates with $\mu = 0.5$ and $\delta = 1$



(b) Line repository after three updates with $\mu = 0.5$ and $\delta = 20$

Figure 23: Line repository with different maximum distance between endpoints thresholds for merging.

11 Future work

Future work on the robot-project should first and foremost be dedicated to fixing server communication issues. This might entail the adoption of the new C++ server, over the older Java server, since C++ is more familiar to most of the cybernetics students.

Expanding the server-side logic for more easily plotting custom data acquired from the robot during run-time would be a benefit for future development of the robot application. Currently, reading data from robot during run-time is restricted to the simplistic method of sending data over a USB connection to the developer's working station.

Building upon the feature-extraction implementation developed during the course of this project for integration into a local SLAM application on the robot itself, is also a natural way of further development of the robot-project. This will enable fully autonomous operation of the robots, thus increasing robustness of the system has a whole. Furthermore, correlating range-bearing measurements over time, would enable the detection of loop-closures, thus minimizing drift in pose estimates of the extended Kalman filter. This could entail research and development of a fitting robust data association algorithm (e.g. RANSAC), and adding extracted line segments as states to the existing extended Kalman filter.

The current particle filter-based SLAM application running on the Java server should be taken better advantage of than displaying a GUI of the robot's location and surrounding environment. This could entail reporting back the updated pose estimate of the robot back to the robot itself. The pose estimate from the server could be fused into the current pose estimate the robot has computed, possibly enabling further decrease in pose drift.

The magnetometer embedded in the IMU on the robot is currently not in use. It is expected that fusion of the heading measurements from the magnetometer with the odometry and intertial measurements already in use would further increase accuracy of pose estimation.

References

[1] Banggood. *Machifit 25GA370 DC 12V Micro Gear Reduction Encoder Motor with Mounting Bracket and Wheel - 110RPM*. URL: https://www.banggood.com/Machifit-25GA370-DC-12V-Micro-Gear-Reduction-Encoder-Motor-with-Mounting-Bracket-and-Wheel-p-1532242.html?utm_source=googleshopping&utm_medium=cpc_organic&gmcCountry=NO&utm_content=minha&utm_campaign=minha-no-en-pc&currency=NOK&cur_warehouse=CN&createTmp=1&ID=6157423&utm_source=googleshopping&utm_medium=cpc_union&utm_content=sandra&utm_campaign=sandra-ssc-no-all-0302&ad_id=337428064977&gclid=Cj0KCQjwnoqLBhD4ARIsAL5JedJP5LpKAdGG2ce0sGZKJwxHG02JDMGT-2iXCdv6jYSNI-es1Uh9RYQaAkKnEALwwcB (visited on 10th Oct. 2021).

[2] Richard Barry. *Mastering the FreeRTOS Real Time Kernel - A hands on tutorial guide*. URL: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf (visited on 16th Nov. 2021).

[3] K.R Beevers and W. H. Huang. 'SLAM with Sparse Sensing'. In: *IEEE International Conference on Robotics and Automation* (2006).

[4] 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).

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

[6] 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.

[7] G. H. Eikeland. 'Title missing'. In: *Master thesis, NTNU* (2018).

[8] *FreeRTOS FAQ - Memory Usage, Boot Times Context Switch Times*. URL: https://www.freertos.org/FAQMem.html#StackSize (visited on 19th Dec. 2021).

[9] *FreeRTOS Queue Management API reference*. URL: https://www.freertos.org/a00018.html.

[10] *FreeRTOS Task Notifications - Inter-task communication and synchronization*. URL: https://www.freertos.org/RTOS-task-notifications.html.

[11] *GSL - GNU Scientific Library*. URL: https://www.gnu.org/software/gsl/ (visited on 19th Dec. 2021).

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

[13] E. H. Jølsgard. 'Embedded nRF52 robot'. In: *Master thesis, NTNU* (2018).

[14] M. F. Lindefjeld. 'Obstacle Detection and Avoidance in a 3D environment'. In: *Master thesis, NTNU* (2021).

[15] E. Masehian, M. Jannati and T. Hekmatfar. 'Cooperative mapping of unknown environments by multiple heterogeneous mobile robots with limited sensing'. In: *Robotics and Autonomous Systems* 8 (2016).

[16] S.T Pfister, S.I. Roumeliotis and J.Q Burdick. 'Weighted Line Fitting Algorithms for Mobile Robot Map Building and Efficient Data Representation'. In: *IEEE International Conference on Robotics and Automation* (2003).

[17] *Segger Embedded Studio*. URL: https://www.segger.com/products/development-tools/embedded-studio/ (visited on 19th Dec. 2021).

[18] Nordic Semiconductor. *nRF52840 DK*. URL: https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk (visited on 17th Dec. 2021).

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

[20] João Manuel R. S. Tavares and A. Jorge Padilha. *A New Approach for Merging Edge Line Segments*. 1995. URL: https://www.researchgate.net/publication/37649765_A_New_Approach_for_Merging_Edge_Line_Segments.

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

[22] L. A. Zadeh. *Fuzzy Sets: Information and Control*. 1965. URL: https://www-liphy.univ-grenoble-alpes.fr/pagesperso/bahram/biblio/Zadeh_FuzzySetTheory_1965.pdf.