University of Southern Denmark

Fall 2020

Autonomous Fence Inspection Robot

Mathias Emil Slettemark-Nielsen
23/02-1995
masle16@student.sdu.dk

Mikkel Larsen
08/08-1996
milar16@student.sdu.dk

Johan Bomme Ousager
29/11-1996
joous16@student.sdu.dk

Muhammad Shaheer
25/12-1993
mshah19@student.sdu.dk

Supervisor: Ulrik Pagh Schultz & Jes Grydholdt Jepsen

Hand-in date: 04/01-2021

Abstract

In this report, a prototype of a fence inspection robot is developed, to explorer the possibility to automate fence inspection. This problem was given by the company Lorenz Technology and the robot platform used for development was the Frobit from SDU Biorobotics. The project focuses on two main parts of the robot, hole detection and fence following. In the hole detection, a semantic segmentation neural network and a neural network object detector were implemented. Results showed that the object detector was the best to locate holes and the object detector YoloV5 had a miss rate of 0.069 in real life.

In the fence following a RANSAC based algorithm was developed to detect the minimal distance to a fence from a lidar scan. This algorithm was then used by two different controllers, an input correlation controller and a proportional integral derivative controller, to keep a target distance to the fence. Evaluating these two algorithms in simulation showed that the input correlation controller had 64.8% smaller cumulative squared distance error than the proportional integral derivative controller. However, the proportional integral derivative controller was 33.8% faster doing one lap in a fence map compare to the input correlation controller.

Lastly, a simultaneous localization and mapping method was implemented and tested in simulation to make the robot capable of navigating more accurately and make it capable of avoiding obstacles in the environment.

For future work, the controllers need a more thorough test in real life as due to corona restrictions this was not possible. Furthermore, tests in real-life showed that the lidar scanner attached to the robot had too low of a resolution to see the fence, thus, another sensor must be used.

Contents

1 Introduction

In this project, the task of performing autonomous fence inspection with a mobile robot is explored. The mobile robot used for development is a frobit from SDU Biorobotics. This platforms allows for easy implementation of hardware and is therefore ideal for a pilot project like this.

The focus of this project is not to fine tune every method into perfection, but more simple to explore possibilities of creating a mobile robot capable of fence inspection with the hardware at hand. In that regards, a clean interface and platform independence is also an important aspect of this project. This will allow for easy integration of the software developed to any platform of interest.

The project has two phases, simulation and real world, thus the simulation setup and the real life platform will be elaborated. This will be the general structure of the project, since methods firstly are evaluated in simulation and then in real world. However, due to COVID-19 not all of the methods is tested in the real world.

For fence inspection two different deep learning based approaches are implemented namely, semantic segmentation and object detection. For each approach different methods are evaluated. To navigate the mobile robot three different approaches are implemented namely, input correlation learning, proportional integral derivative controller, and simultaneous localization and mapping.

The code for this project can be found here[1] and the workload of each contributor can be found in Appendix A.

2 Project Description

This project is in collaboration with the company Lorenz Technology and explores the solutions for creating an autonomous system for fence inspection. The primary fence inspection areas are airports, however, the solution should apply to other environments.

In the following section, the project requirements and design goals are elaborated, followed by how the workflow and system design will look. An estimated timetable for this project can be found in Appendix B.

2.1 Project Requirements

Lorenz Technology wants an autonomous drone/mobile robot capable of inspecting the fence of a particular perimeter. The requirements of Lorenz are based on the MoSCoW [1]

[1] `https://github.com/mikkellars/EiT_project`

method and are divided into must have, should have, could have, and will not have:

The solution must have:

- An autonomous system capable of navigating the environment on its own when at the fence.
- The system must be capable of performing even in minimum daylight.
- The system must be able to inspect a fence of the type chain-link with a thickness of a minimum of 9 Gauge.
- The system must be able to detect breaches in the fence of 30x30 cm, as a minimum.
- The system must report the breaches within 24 hours.
- The system must be capable of performing the inspection three times a day.
- The system must be able to inspect the fence in normal weather conditions, thus no heavy rain, etc.

The solution should have:

- The system should be able to recognize that a full inspection of the fence has been completed.
- The system should be able to detect obstacles in the path and stop.
- The system should be able to specify the location of the holes in the fence with a precision of $\pm$ 5 meters.
- The system should be able to specify its location.
- The user should be able to interface with the system through a terminal.

The solution could have:

- The system could be capable of performing at nighttime.
- The system could be capable to detect a hole under the fence.
- The system could be capable to return to base or another specified endpoint.
- The system could be capable of automatically driving to the fence.
- The system could be able to drive around a detected obstacle in the path.
- The system could have online breach detection and localization.
- The system could be able to inspect the fence 24 hours a day.

The solution will not have:

- The system will not be able to inspect different types of fences.
- The system will not be able to transfer real data wirelessly in real-time, thus only when the robot is at the base.

- The system will not have a graphical user interface.

This project has first and foremost focused on the must have and implemented nearly all the points mentioned there. The other points from the MoSCoW method can be implemented if Lorenz shows interest in the project and its potential.

2.2 Design Goals

To meet the above-stated requirements from Lorenz a set of design goals are made. The different goals are divided into must have, should have, could have, and will not have.

The system must have:

- The system must be able to keep a fixed distance to the fence at all times, except when avoiding obstacles. The distance to the fence depends on the camera used since it must be able to capture the whole fence.
- The system must be able to have a clean interface.
- The system must be able to run successfully in a gazebo simulated environment.
- The system must be able to be manually controlled.
- The system must have a killswitch.

The system should have:

- The system should have a clean interface by e.g. using ROS as the framework.
- The system should be constructed so it is easy to switch platforms, e.g. when applying a new sensor or a new robot platform.
- The system should be able to run successfully in a simple indoor testing environment.
- The system should be able to localize itself through GPS.

The system could have:

- The system could be capable to run successfully in a real-world environment.
- The system could be able to use SLAM/vision-based localization for moving indoors.

The system will not have:

- The system will not be able to run in every environment that needs fence inspection.

2.3 Workflow

The workflow of the system is visualized in Figure 1. The workflow shall be seen as the vision of how the robot solution should work in the end and in this report not all features have been implemented to make the robot fully follow the workflow.

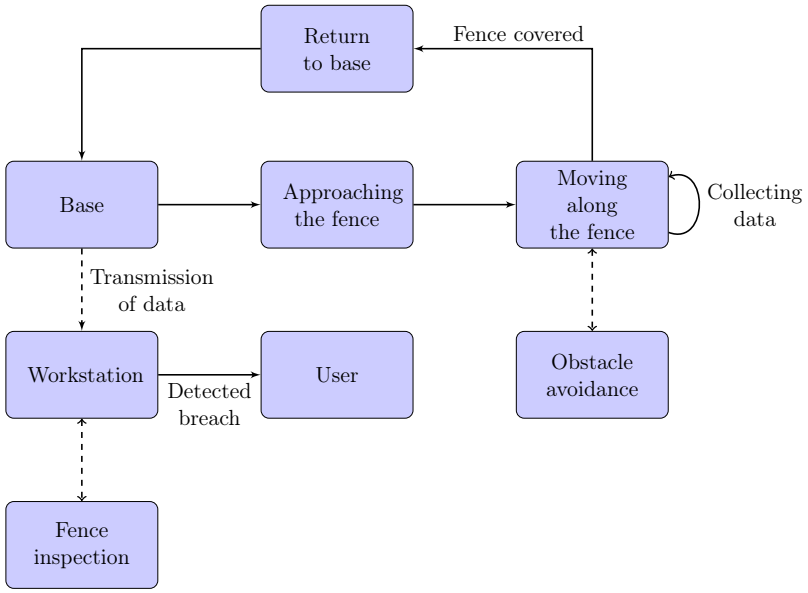


Figure 1: Visualization of the workflow.

In Figure 1, the mobile robot starts at the base and for that matter ends at the base. When the mobile robot starts, it first approaches the fence so that a good image of the fence can be obtained. Afterward, it moves along the fence with a fixed distance to the fence, while taking images and simultaneously logging the position with the attached GPS. If it is the first operation of the robot in that environment, it uses a fence following behavior while creating a map with a SLAM method, else it navigates using the map created from the SLAM method. Depending on the images captured and the performance of the fence inspection, the robot might have to stop and then take the image to eliminated blurring as much as possible. When the mobile robot has inspected all of the fences, it returns to its base. At the base, the information gathered is sent to the workstation for data processing. If a breach is detected, the system informs the user via the user interface, and the user can then inspect the location.

2.4 System design

The system design is shown in Figure 2 as a top-down illustration with arrows that illustrate the flow of information in the system.

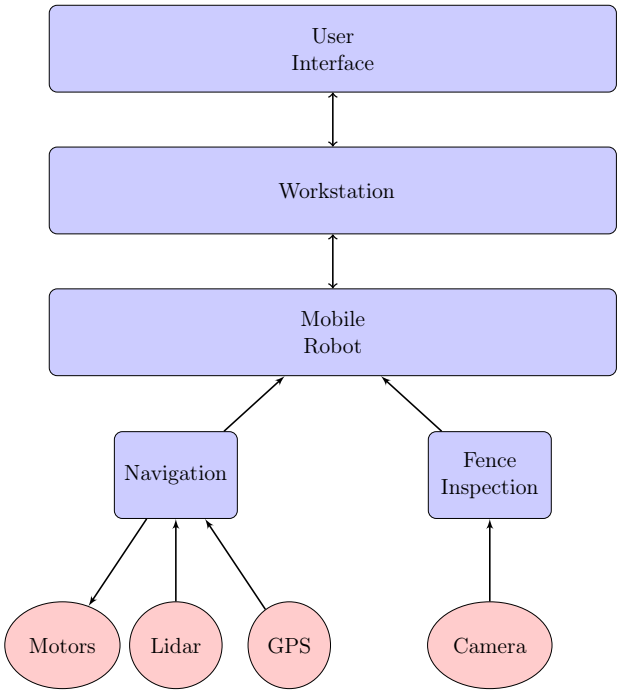


Figure 2: System design.

In Figure 2 the top layer is the user interface that communicated with the workstation system, the workstation system then communicates with the mobile robot system. The mobile robot receives information from the fence inspection software regarding the hole detection, and the navigation software, which e.g. sends information about the localization of the robot, constructs lines of the data points from the lidar and, keeping a fixed distance to the fence. Thus, the fence inspection software has contact with the camera and, the navigation software uses the motors, lidar, and GPS.

2.5 Acceptance Criteria

To test the developed product different tests are performed. Thus, the testing phase is divided into robot control (navigation and controlling the robot), and fence inspection (computer vision system for inspecting the fence).

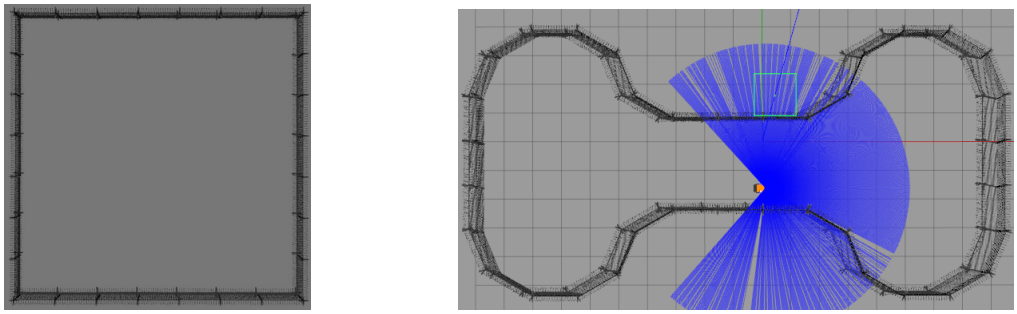
Fence inspection: To test the vision system's ability to detect breaches in the fence, images of the fence need to be collected. The different vision algorithms' final performances are evaluated on a test set. The test set contains images that have a distribution representing the real use case to get the best real-life estimation of the performance of the algorithms. Thus, the images need to contain different daylight brightness and holes no smaller than 30x30 cm. The algorithms' performance is evaluated as binary classifiers where for each image there is either a hole or not. The score used to evaluate the perfor-

mance is accuracy and false negative rate, miss rate, because it is more important than the algorithms do not miss any holes more than predicting false positives. The miss rate is calculated as

$$miss\ rate = \frac{FN}{P} = \frac{FN}{FN + TP} \tag{1}$$

where P and N are positives and negatives in the data set respectively and TP, TN and FN are true positives, true negatives, and false negatives predicted by the algorithm on the data respectively. The vision algorithm should have as low a miss rate as possible, thus, the method is deemed acceptable if it has a miss rate below 0.1. The reason for not having a miss rate of 0 is because it is deemed acceptable for the robot to miss a hole in one frame if the hole is detected in the next frame. Thus, as long as the robot sees the hole at some point in the sequence of images, the system is still able to notify the user.

Navigation: To show that the robot can navigate consistently in different environments, three environments in gazebo will be created. Firstly, two simple environments will be tested. These are shown in Figure 3. The map in Figure 3b is only used for an input correlation controller to find the optimal weights, whereas the map in Figure 3a is used to choose between the input correlation controller and a proportional integral derivative controller.

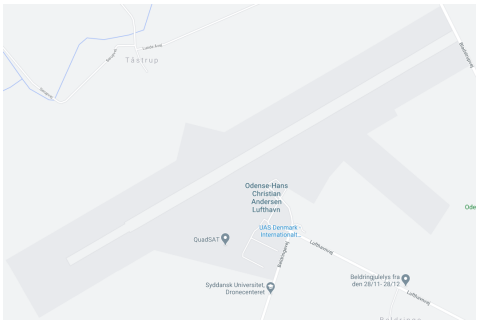


(a) Square fence structure containing 29 $[m]$ of fence.

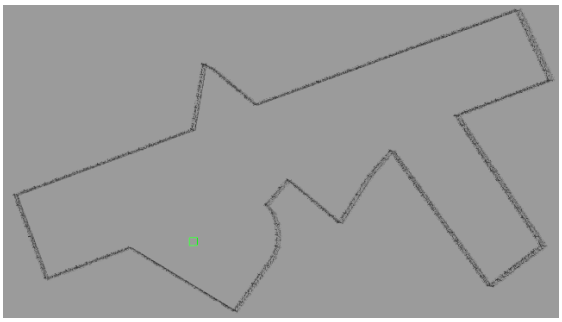
(b) Dog bone fence structure containing 54 $[m]$ of fence.

Figure 3: Gazebo environments for testing robot navigation.

Lastly, an environment that resembles HCA airport will be created and the robot will be tested in it. The environment is shown in Figure 4 besides a top view of the HCA airport from google maps. This map is used to accept the controllers.



(a) Top view of HCA airport from google maps.



(b) Top view of Gazebo world. Contain 240 $[m]$ of fence.

Figure 4: Comparison of HCA airport top view against the created Gazebo world.

For the robot to succeed in the different environments, it must hold a fixed distance to the fence with a small variation while driving along with it. Thus, the less deviation from the fixed distance to the fence the better. Furthermore, must the robot cover all the fence without getting stuck in a loop or deviating from the track. Two metrics are used to evaluate the distance controllers, a lap time and a cumulative squared distance which is calculated as

$$\text{cumulative square distance} = \sum (\text{target}_{\text{distance}} - \text{current}_{\text{distance}})^2 \tag{2}$$

The cumulative squared distance is chosen as a metric as it states how good the Frobit sticks to the target distance to the fence.

Since the robot is meant to drive in the real world and not in a simulation, the robot will be tested in real life. In this test, it will be observed if the robot can function in the real world when driving along a fence.

3 Simulation

The simulation platform used for testing the working and behavior of robot and algorithms was Gazebo. Gazebo offers the ability to accurately and efficiently simulate robots in complex environments. The two main components required to test the working of a robot and other algorithms implemented on it are:

- Gazebo World
- Robot Model (URDF format)

3.1 The Gazebo World

To create a realistic representation of the actual world in which the robot will operate, a Computer Aided Design, CAD, model of the environment is needed. To create a Gazebo world for this environment, potentially any chain-link fence CAD could be used because the aim of the project, as mentioned in previous sections, was to detect the breaches in the fence. Therefore, a CAD model of a simple fence was used which and can be seen in Figure 5.

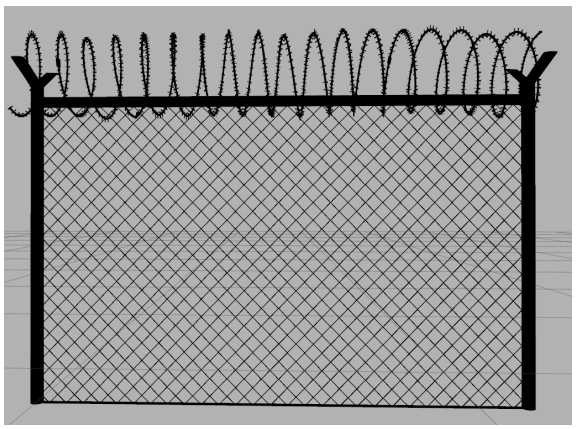
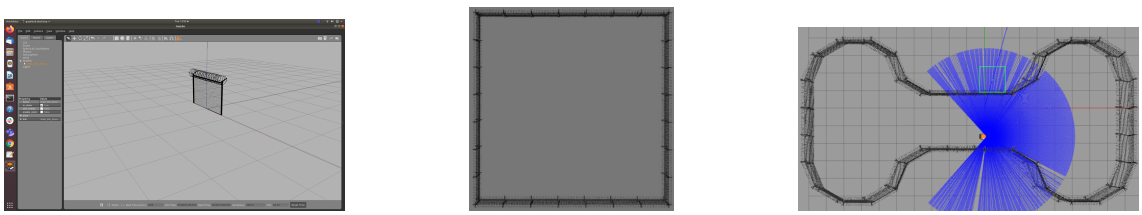


Figure 5: Fence CAD model used for Gazebo Environment [2]

If the CAD model of the environment is available, then creating a Gazebo world was quite straight forward. These are the main steps, this is also visualised in Figure 6, taken for creating and saving the Gazebo world:

- Created a model.sdf file
- Opened the model.sdf file using the command *gazebo model.sdf* (This launched the fence model in empty gazebo environment)
- Placed multiple fence model next to each other in order to create a near-realistic representation of the actual environment.
- Saved the fence.world file.



(a) Single Fence Imported (b) Squared Fence structure (c) Dog bone fence structure

Figure 6: Importing CAD model in Gazebo to create environment

After creating an environment in which the robot will be tested, the next step was to create a robot model that be simulated in this environment, and on which the further testing of algorithms would take place.

3.2 Robot Model

The robot model for gazebo is created in the Unified Robotic Description Format, URDF. The URDF is an XML file format used in ROS to describe all elements of a robot. There are several steps to get a URDF robot properly working in Gazebo.

- Creating a .xacro file which describes the visual, inertial and collision properties of the physical elements of the robot, e.g chassis, wheels etc.
- Creating a .gazebo file to simulate the sensors, e.g laser, imu, gps, etc., and the drive controller, e.g differential drive controller.

3.2.1 Creating a Xacro File

The main components of the robot model were: Chassis, drive wheels, front caster wheel, Hokuyo lidar, and camera. To ensure the proper working of the robot, the inertial and collision properties of all these parts should be accurately calculated. Moreover, the transformations between all the links and joints should be properly defined.

Visual Properties: Visual properties is, as the name suggests, a visual representation of the part/link. Therefore, the mesh file, .STL or .DAE, of the specific part/link was used to visualize it.

Inertial Properties: Inertial properties of a part/link include mass of the object and moment of inertia of the object. Moment of inertia was a bit tricky to calculate depending on the varied geometries and physical properties of every part/link. Therefore, these values are believed to be not the most accurate in this project because they were calculated by Meshlab software, and not calculated by hand.

Collision Properties: Collision properties basically define the footprint of each part/link. They define how much area is covered by a specific part/link, and how far away each part/link should be placed in order to avoid collisions among bodies and with the environment.

Transformations: The most important element of a robot model in ROS and Gazebo is the transformations between different links and joint of the entire robot. They describe how each element of the robot behaves relative to each other when the robot is moving. The most important link is the base/chassis link. All of the other links are, mainly, joined with the chassis frame. Figure 7 describes the transformation tree of the frobit robot model

used in this project.

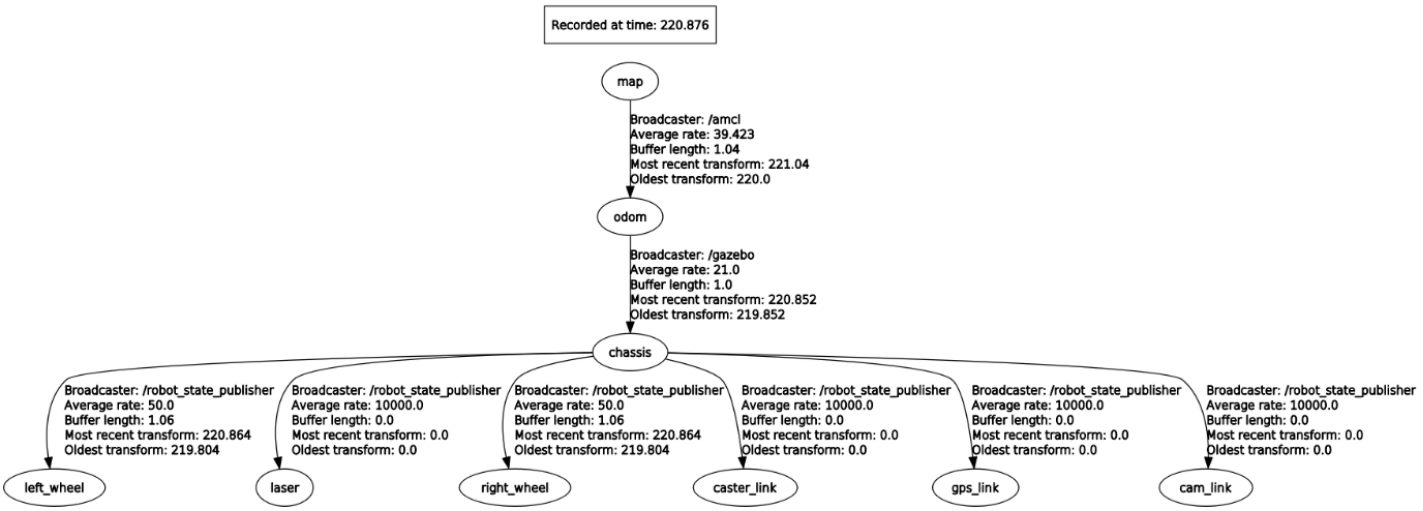


Figure 7: Transformations between various links and joints of the robot and environment

3.2.2 Creating a .gazebo file

The .gazebo file contains all the <gazebo> elements of the robot model. The <gazebo> element is an extension to the URDF used for specifying additional properties needed for simulation purposes in Gazebo. They are basically responsible for simulating the behavior of sensors and actuators. Various plugins are defined to simulate the sensors and actuators. Some of the plugins used in this project are :

- Differential drive plugin: libgazebo_ros_diff_drive.so, this plugin controls the actuators (rear drive wheels) of the robot according to a differential drive. Base frame, left and right wheel frame and wheel separation are defined, and the topic on which twist messages will be received is defined.
- Lidar controller plugin: this plugin simulates the lidar sensor, in this case th Hokuyo Lidar. It also has multiple parameters including update rate, sample size, field of view etc. the sensor readings are published on \laser \scan topic.
- Camera controller plugin: this simulates the camera module. Similar to other plugins, this plugin also has multiple parameters which include and are not limited to field of view, image size, and camera parameters etc. The images from the camera are published on \image_raw topic.
- gazebo_ros_control plugin: this plugin controls the behavior of actuator joints, left and right wheel joints, of the robot.

3.3 Robot Control

Once all the physical properties and sensor and actuator plugins of the robot are defined, the robot model is ready to be launched in the gazebo environment made in the first step. Once the robot model is spawned in the environment, this is explained in next step, the next step is to properly control the movement of robot in the environment. *teleop_twist_keyboard* is an open source package for controlling the movement of robot. The package publishes twist messages on \cmd_vel topic, defined in the differential drive controller plugin in .gazebo file, and the robot can be controlled via keyboard just like a normal differential drive car.

3.4 Launch File

Once everything is modelled, it is launched together in a single environment using a *launch file*. Launch file allows to launch multiple nodes from multiple packages using a single *roslaunch* command. The launch file created for this project, launches the following nodes and capabilities of simulated robot:

- **gaezbo_ros:** It launches the gazebo world in the gazebo environment (created in first step).
- **mybot_spawn:** It launches the URDF model of the robot in the gazebo environment. it takes robot_description parameter (.xacro file) as input parameter.
- **robot_state_publisher:** It uses the URDF specified by the parameter robot_description and the joint positions from the topic joint_states to calculate the forward kinematics of the robot and publish the results via tf.
- **rviz:** This is a visualization software used to visualize the robot, its transformations, the map, the planned path, the sensor readings etc.
- **map_server:** This node launches the pre-built map of the environment that will be used for navigation (explained in the navigation section).
- **amcl:** this node helps the robot to localize itself in the map provided by map_server.
- **move_base:** This node helps defining the desired goal location in the map, plans the local and global paths and execute the robot movement.

4 Platform

The project is developed on the Frobit platform, which is a simple and cheap platform designed for development, shown in Figure 8. A camera and lidar are attached to the platform and connected to the Raspberry Pi.

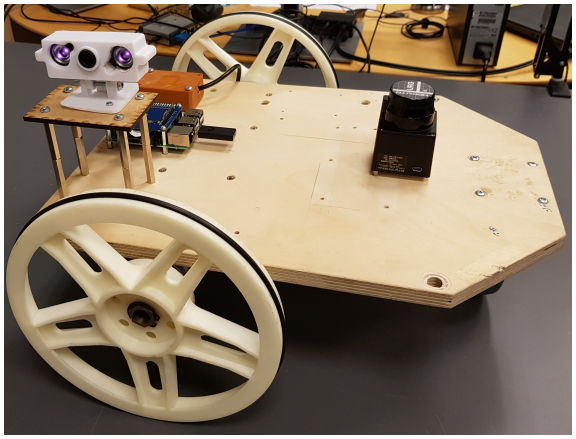


Figure 8: The frobit platform with camera, GPS, and lidar attached.

The camera on Figure 8 is a ZeroCam NightVision [3], furthermore, a Raspberry Pi Camera Board v2.1 [4] was also used. The lidar is a Hokuyo URG-04LX-UG01 [5].

Since the used components are fairly cheap an important aspect of this project is also to see if the task is doable with these components.

4.1 Platform Independence

Since the project is developed on the development platform Frobit, it is important to have a platform-independent system for easy migration to a final platform. Therefore, is the software docker [6] used. Docker can package an application and its dependencies in a virtual container that can run on any Linux, Windows, or macOS computer. This enables the application to run on a variety of platforms, e.g. a CAPRA robot [7].

To further ensure platform independence, the flexible framework Robot Operating System [8], ROS, is used. ROS is a collection of tools, libraries, and conventions that simplifies the task of creating robot behavior across robotic platforms. This allows for easy integration of new components.

4.1.1 Drivers for Components

As shown on Figure 8 different components are attached to the Frobit and these need to have a driver, i.e. software for the system to communicate with the hardware.

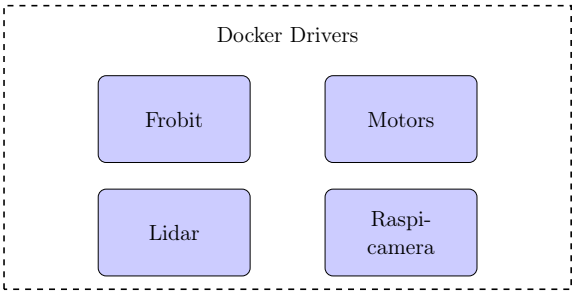


Figure 9: Illustration of docker driver containers.

From Figure 9 the drivers used are illustrated. The drivers Frobit and Motors were developed by SDU Biorobotics and will not be elaborated further here. For the Hokuyo lidar, the hokuyo node [9] is used and implemented in its own container. The angle of the lidar is set to $\pm$ 2.2689 radian and clustering is set to 1, i.e. no clustering of points, in the Hokuyo launch file. The driver for the raspberry camera [10] also have its own container as illustrated in Figure 9. The camera was publishing images to a node with a resolution of 410×308 pixels and a framerate of 30 Frames Per Second, FPS.

4.1.2 Fence Inspection Application

The fence inspection application is implemented as one container with various ROS nodes, which is illustrated in Figure 10. Figure 10 also illustrates the communication between the core and both application and drivers.

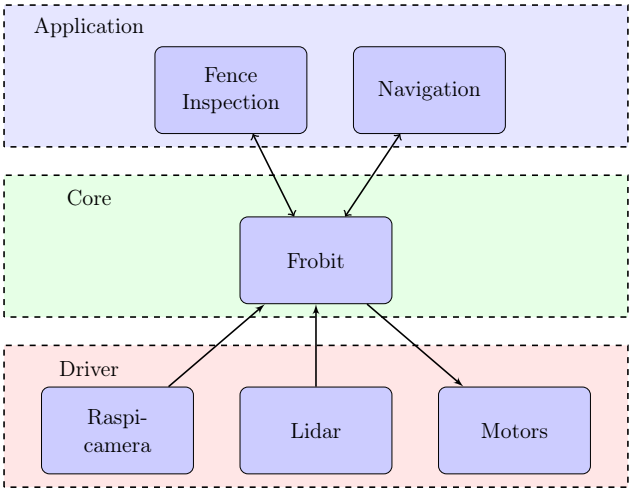


Figure 10: Overview of the structure of robot.

The applications are gathered in one docker as these are necessary for the robot to be capable of fence inspection. This makes it easy to switch the whole application layer with another docker if the robot should be capable of solving a different task. The reason the driver is not implemented under the same docker is that it should be easy to switch out the

sensors. When the drivers have their own docker, a new docker with e.g. another camera driver is easily replaceable with the raspi-camera driver without interacting with the other drivers.

5 Inspection of Fence

To detect holes in a fence with a camera, a vision algorithm must be implemented. Today most traditional vision methods are replaced by neural networks which achieve much higher accuracy and are currently State-Of-The-Art, SOTA. This is mainly due to neural networks learning through supervision rather than traditional methods which are the programmer's optimal way to solve the problem. Thus, traditional methods tend to be good on the data from which they are programmed from but when evaluated on real data, which e.g. have different light conditions, neural networks outperform the traditional methods.

Visual inspection of fences can be done in many ways, for example by looking at the periodic structure of the fence, using a deep neural network for object detection where the object would be a hole or using a deep neural network for semantic segmentation of the fence. Since the first approach was done in [11], the two other approaches will be investigated here.

5.1 Fence Segmentation

The first idea is to segment the fence in the image and thereby extracting the fence structure from the image. This fence structure can then be used to analyze the condition of the fence. This task in computer vision is referred to as semantic segmentation, i.e. predicting a label for each pixel in the image. The general structure of semantic segmentation consists of an encoder followed by a decoder. The encoder is, usually, a pre-trained feature extractor, e.g. ResNet [12]. The decoder semantically projects the features, from the encoder, onto the pixel space to get a dense classification.

The dataset used to train the semantic segmentation network is the Fence Segmentation Dataset [13] and examples can be seen in Figure 11. This dataset consists of 645 images and is split into 381 training images, 164 validation images, and 100 testing images. When training various image augmentations are used to artificially create more data by the use of [14], e.g. channel shuffle, horizontal flip, vertical flip, and blurring.

Figure 11: Examples of semantic segmentation data. The first row is the image and the second row is the corresponding ground truth, where black pixels indicates background and white pixels indicates fence.

Two different semantic segmentation methods are considered here, U-Net [15] with a ResNet34 feature extractor and DeepLabV3 [16] with a ResNet50 feature extractor. The results for both methods are shown in Table 1.

Method	**Epochs**	**Size [MB]**	**Loss**	**Accuracy [%]**
U-Net	100	73.4	0.34 (+/-0.0068)	97.7 (+/-0.0071)
DeepLabV3	100	168.4	0.091 (+/-0.0405)	96.84 (+/-0.0083)

Table 1: Training results for U-Net and DeepLabV3 on [13].

From Table 1, it can be seen that U-Net performs slightly better than DeepLabV3 on the test set in terms of accuracy. Examples of the prediction from both U-Net and DeepLabV3 can be seen in Figure 12.

Figure 12: Examples of images, first row, with corresponding predictions from U-Net, second row, and DeepLabV3, third row. In the predictions images, second and third row, the prediction and input image is mixed and purple pixels indicates background whereas yellow pixels indicates fence.

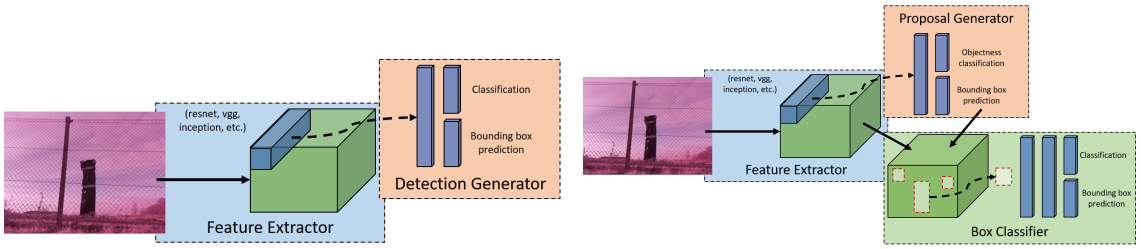
From Table 1 and Figure 12, it is shown that is it possible to segment the fence in an image that can be used for further analysis of the fence and its condition. However, this approach did not show promising results on images provided by Lorenz. This is most likely because the fence is further away than on the dataset [13]. Therefore, was this approach not investigated any further.

5.2 Hole Detection

The second idea, for detecting holes in fences, is to see it as an object detection problem. Object detection is the task of finding a set of detected objects in a single RGB image, and for each object predict a category label and bounding box.

An object detector, usually, consists of two parts: a backbone for feature extraction of the input image and a head for predicting classes and bounding boxes of objects. An example of a backbone on GPU platform is ResNet [12] and for CPU platform MobileNet [17]. The

head is split into two categories one-stage object detector and two-stage object detector, an illustration of this is shown in Figure 13.



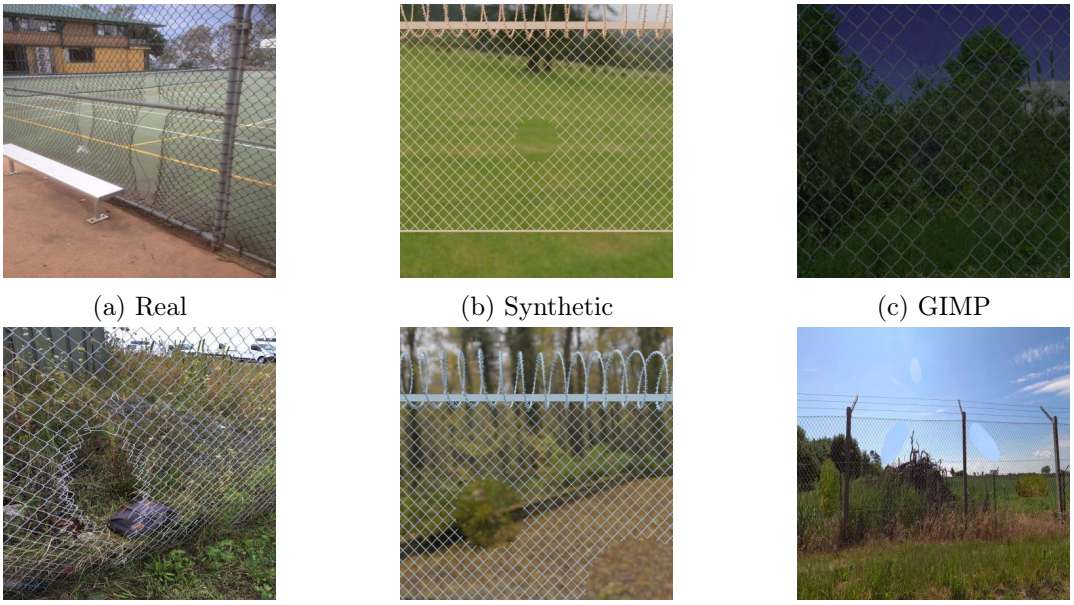
(a) One-stage object detector (b) Two-stage object detector

Figure 13: Illustration of common object detector structures

Since the problem only consists of one class, i.e. a hole, a one-stage detector would be sufficient, however, two-stage detectors normally perform better. Therefore, will both two-stage detector and one-stage detector be taking into account. From [18], one of the best performing object detection method is Faster Region-based Convolutional Neural Network [19], Faster R-CNN, thus, it will be investigated in this report. Furthermore, will the SOTA method You only look once version 5 [20], Yolov5, also be investigated. Lastly, since it would be preferable to run the neural network locally on the robot, the Mobile Detector [21], MobileDet, will also be investigated. More precise, the SSD Lite MobileDet since it is incorporated with an edge Tensor Processing Unit, TPU, [22] which is preferable when running on a small single-board like the Raspberry Pi. By utilizing a TPU is it possible to run the heavy computations of a neural network up to 85 % faster [23].

5.2.1 Training of Object Detectors

Since the object detectors are trained by supervision, a dataset with bounding boxes are needed. However, collecting enough images of fences with holes is problematic, thus, synthetic images are also created. To generate the synthetic images of the fence the software [24] is used with a CAD model of a fence. Furthermore, images of fences with holes are created in GIMP [25] by images without holes. The dataset consists of 418 images and is split into 325 training images and 93 validation images. Examples of the images used for training the object detectors are shown in Figure 14.



(a) Real (b) Synthetic (c) GIMP

(d) Real (e) Synthetic (f) GIMP

Figure 14: Examples of data used to train object detectors.

The performance of the three object detectors on the dataset is shown in Table 2. In the table, it can be seen the number of training cycles through the dataset, epochs, the best mean average precision with an intersection over union over 0.5, mAP@0.5, the prediction time, and the size of the model. mAP@0.5 is elaborated in [18].

Method	**Epochs**	**mAP@0.5**	**Prediction time [ms]**	**Size [MB]**
Faster RCNN	200	0.95	17.03 (GPU - GTX 1050 ti)	165.80
YoloV5	300	0.99	8.10 (GPU - GTX 1050 ti)	14.80
SSD Lite MobileDet	2461	0.87	8.00 (Edge TPU - USB Accelerator)	3.40

Table 2: Results of the three methods when trained on the dataset. Epochs refers to training cycles through the dataset. The term mAP@0.5 is elaborated in [18]

From Table 2, it can be seen that the two-stage detector Faster RCNN is more than twice as slow as Yolov5 and SSD Lite MobileDet, however, this is expected. Furthermore, the size of Faster RCNN is much bigger than the other two without a significant boost in mAP@0.5, thus, Faster RCNN is discarded. Since the SSD Lite MobileDet scores a good mAP@0.5 with the smallest model in this experiment and has a low prediction time without a GPU, thus it is not discarded and is preferable to run locally on the robot. Note when SSD Lite MobileDet is run on a CPU, Intel® Core™ i7-7700HQ CPU @ 2.80GHz, the average prediction time is 118 milliseconds. Thus, the edge TPU gives a significant performance boost in terms of inference time. Lastly, the Yolov5 scores the highest mAP@0.5 of all the

methods with a low prediction time, thus, it is not discarded and is preferable to run on the workstation. Therefore, is SSD Lite MobileDet and Yolov5 evaluated further.

5.2.2 Evaluation of Object Detectors

To evaluate the object detectors on unseen data, data is collected both from simulation and the real world. The simulation data is collected by following the fence while taking images of the fence, and the real-world data is collected by manually driving the robot along a fence with plastic bags attached.

Simulation: Firstly, the robot was tested in simulation on images with and without holes to estimate the performance of the object detectors. Examples of the images in simulation can be seen in Figure 15.

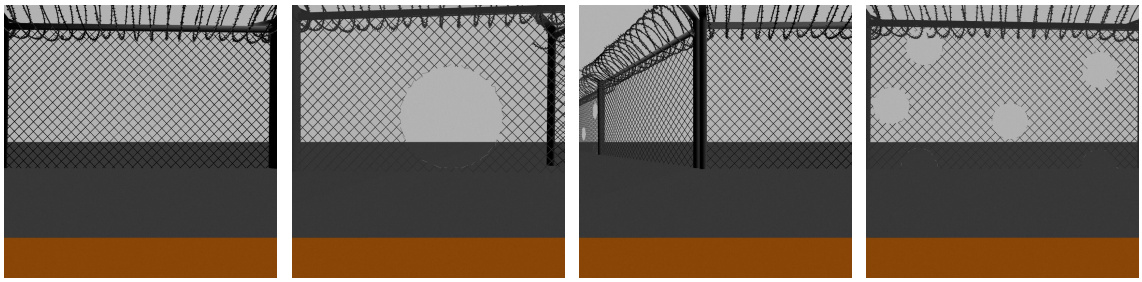


Figure 15: Examples of images from simulation.

The simulation images consist of 600 images with and without holes, in a total of 481 holes. The result from the test of Yolov5 and SDD Lite MobileDet can be seen in Table 3.

Method	**TP**	**FP**	**FN**	**mAP@0.5**	**Miss rate**
YoloV5	435	0	46	0.90	0.096
SSD Lite MobileDet	283	41	198	0.57	0.41

Table 3: Yolov5 and SDD Lite MobileDet results on simulation images with True Positives, TP, False Positives, FP, False Negatives, FN, Mean Average Precision with an intersection over union over 0.5, mAP@0.5, and miss rate.

From Table 3, Yolov5 outperforms SSD Lite MobileDet, however, this was expected when considering the model size difference. When considering the results in Table 3, it must be stated that the images are coming in a sequence as it would from the robot. Thus, if the object detector misses a hole in the first frame and then detects it in the next frame, it is still considered a success for this project. However, Table 3 does not estimate this, and to show that every hole is seen once, a video is created for both Yolov5 and SSD

Lite MobileDet. The video with Yolov5 can be found here[2]. The video with SSD Lite MobileDet can be found here[3].

Examples of the detections on the simulation images can be seen in Figure 16, where the first row is Yolov5 and the second row is SDD Lite MobileDet.

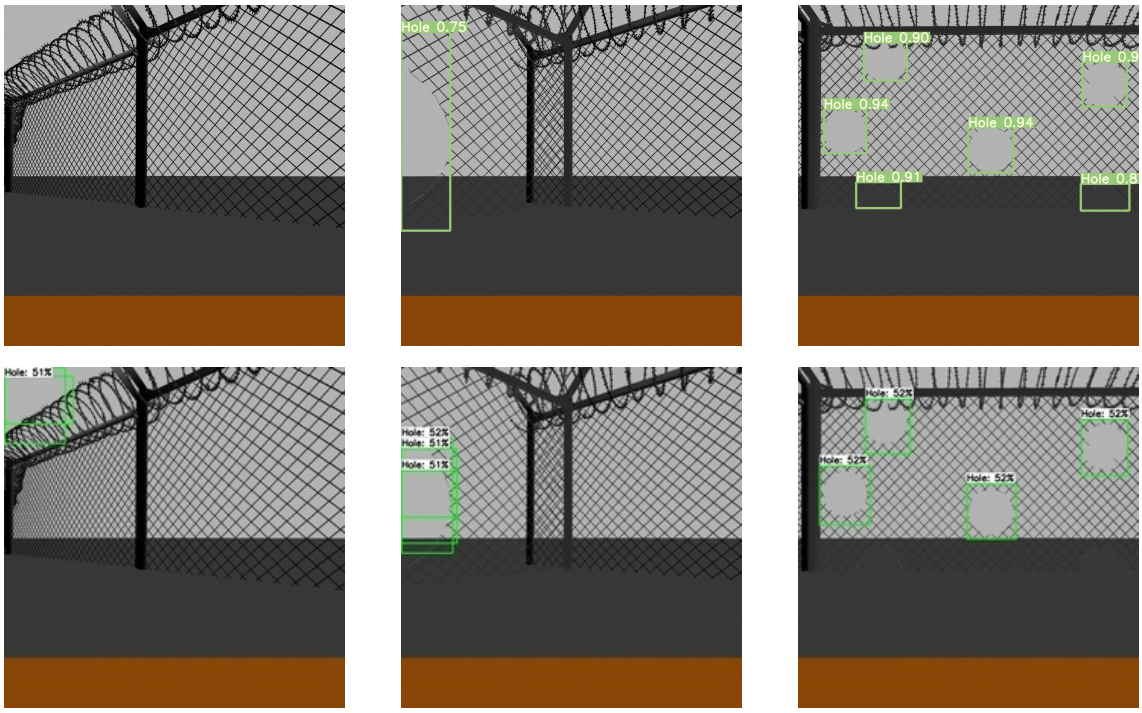


Figure 16: Examples of detection from Yolov5, first row, and SSD Lite MobileDet, second row.

From Figure 16, it can be seen that SDD Lite MobileDet detects false positives where Yolov5 does not. Furthermore, Yolov5 is more confident in its predictions, e.g. in the middle image Yolov5 only predicts one bounding box with a confidence of 75 % and SSD Lite MobileDet predicts three bounding boxes with the highest confidence of 52 %. Lastly, in the last image, SDD Lite MobileDet does not detect the holes at the bottom, however, Yolov5 does.

Real world: Lastly, the robot was exposed to the real world. However, not with real holes, because of security reasons, but with plastic bags attached to the fence to indicate anomalies. Examples of the real-world images can be seen in Figure 17.

[2] `https://github.com/mikkellars/EiT_project/blob/main/vision/assets/yolov5_sim.gif`
[3] `https://github.com/mikkellars/EiT_project/blob/main/vision/assets/ssd_mobiledet_sim.gif`

Figure 17: Examples of images from real world.

The sequence of real-world images consists of 213 images with and without holes and in total 261 holes. The result from the test of Yolov5 and SDD Lite MobileDet can be seen in Table 4.

Method	**TP**	**FP**	**FN**	**mAP@0.5**	**Miss rate**
YoloV5	243	3	18	0.93	0.069
SSD Lite MobileDet	99	55	162	0.38	0.62

Table 4: YoloV5 and SDD Lite MobileDet results on real world images with True Positives, TP, False Positives, FP, False Negatives, FN, Mean Average Precision with an intersection over union over 0.5, mAP@0.5, and miss rate.

From Table 4, Yolov5 outperforms SSD Lite MobileDet as with simulation images. However, both methods miss some holes in the frames. Therefore, are videos of both methods made to show that holes are detected at some point in the sequence of frames. The Yolov5 video can be seen here[4] and the SSD Lite MobileDet here[5].

Examples of the detections on the real images can be seen in Figure 18, where the first row is Yolov5 and the second row is SSD Lite MobileDet.

[4] `https://github.com/mikkellars/EiT_project/blob/main/vision/assets/yolov5_real_world.gif`

[5] `https://github.com/mikkellars/EiT_project/blob/main/vision/assets/ssd_mobiledet_real_world.gif`

Figure 18: Examples of detection from Yolov5, first row, and SSD Lite MobileDet, second row.

From Figure 18, it can be seen that Yolov5 is more confident in the prediction than SSD Lite MobileDet. Furthermore, does SSD Lite MobileDet miss the first hole in the middle image, however, it did detect it in the first image. Thus, the hole would still have been found by the robot. Figure 18, also shows that Yolov5 is more confident in its prediction than SSD Lite MobileDet.

In this section, a method for finding holes in fences was elaborated, both a method for running locally on the robot and one to run at the workstation was developed. This approach showed promising results both in simulation and the real world. However, due to constraints, the methods were not tested with real holes but plastic bags. The plastic bags did still indicate anomalies, thus, the approach will most likely work on real holes. The results showed also indicate that the camera used is satisfying and the mobile robot does not need to stop when capturing images. Furthermore, methods for segmenting the fence in the image were also developed but due to bad results on unseen data this approach was not investigated further.

6 Navigation

The other central function of the software is to enable the robot to travel autonomously. While the robot is traveling, three main goals are to be kept in mind.

- That the robot can follow the course of the fence, at a fixed distance.
- That the robot can localize itself, and thereby any holes it finds.
- That the robot can plan a collision free path towards a desired location

- That the robot can move along a predetermined route.

The software was split into two parts, to handle the fence following and mapping, respectively.

6.1 Fence Following

To follow the fence, a method was needed to find the distance to the fence, and a control mechanism for correcting the distance.

6.1.1 Localization of Fence

The distance to the fence was based on the distances measured by the lidar. To improve the robustness of this measure, a line fitting method was used. This method is elaborated and evaluated in this section and an illustration of the line fitting method flow is shown in Figure 19.

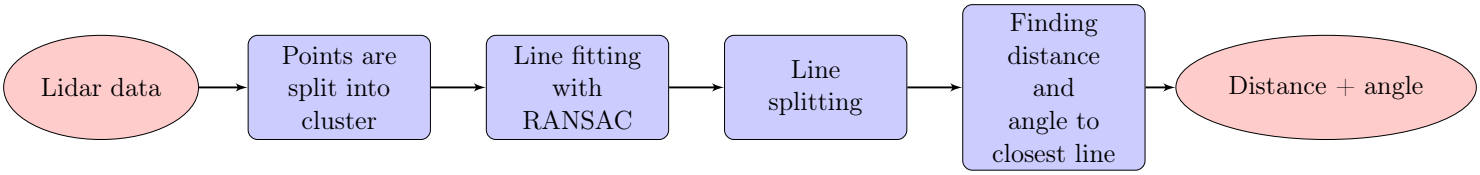
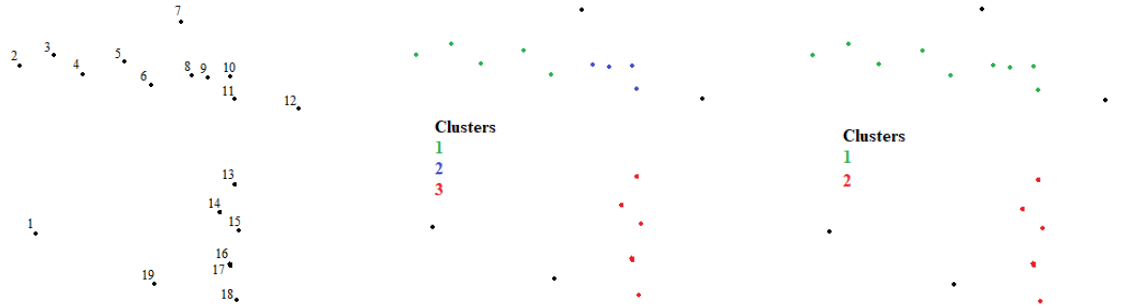


Figure 19: Illustration of the line fitting method flow.

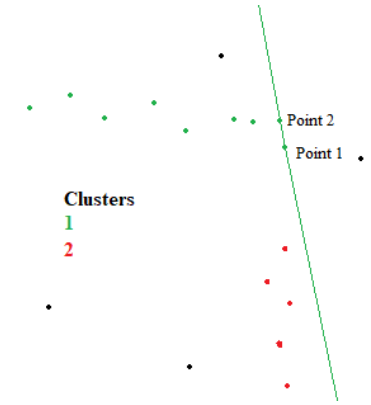
Firstly, the data points transformed from polar coordinates to cartesian coordinates, and any points outside of the lidars range were deleted. Next, the points were split into clusters. This was done so that lines would not be formed between discontinuous points. A naive implementation was used, which simply checks the distance between consecutive points. If that distance is greater than some threshold, the previous points are split into their own cluster. This method takes advantage of the natural structure of the data. As each consecutive data point is the depth measurement at a slightly greater angle than the previous data point, a small distance is expected between the points if they are hitting the same object or surface. This method is also very light on computational cost. However, it is vulnerable to outliers in the data. For this reason, a merging of clusters is also done. If the last point in any cluster is within some distance of the first point of other clusters, then the two clusters are merged. See Figure 20 for an illustration of the point clustering method.



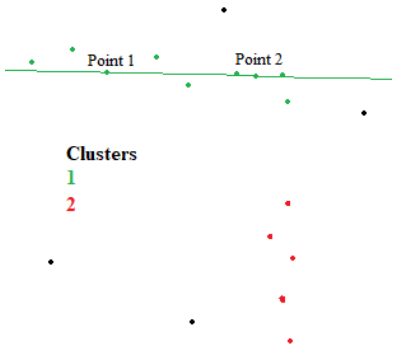
(a) Step 1: Point generated (b) Step 1: Clusters made (c) Step 3: Clusters merged

Figure 20: A constructed example of how the clustering method works.

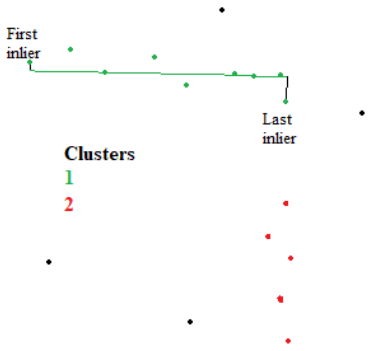
Lines are then fit to the data using RANdom SAmple Consensus, RANSAC. RANSAC works by selecting two random points from a sample and fitting a line to them. The rest of the points are then polled, to see if they are inliers. Inliers are defined as points that are within some distance of the line. If enough points are inliers, the line is accepted. Otherwise, the process is repeated, until it has failed to find an acceptable line in some set amount of iterations. Each found line is recorded as both a set of inlier and a start and endpoint. The start and endpoints are generated by projecting the first and last inlier onto the line. In this usage, RANSAC is run on each cluster, possibly multiple times. Whenever an acceptable line is found, the inlier points are removed from the cluster and saved separately. RANSAC is then run again on the remaining points. This is done until RANSAC fails to find a line, or there are too few unassigned points in a cluster to get the required amount of inliers. An illustration of the process can be seen in Figure 21.



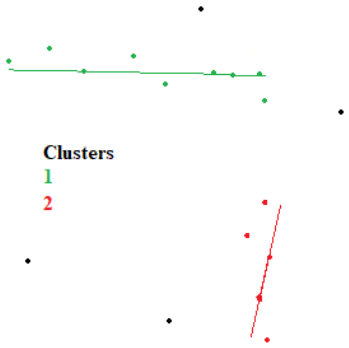
(a) Step 1a: An unacceptable line has been found



(b) Step 1b: An acceptable line has been found



(c) Step 2: The line start and end point have been found.



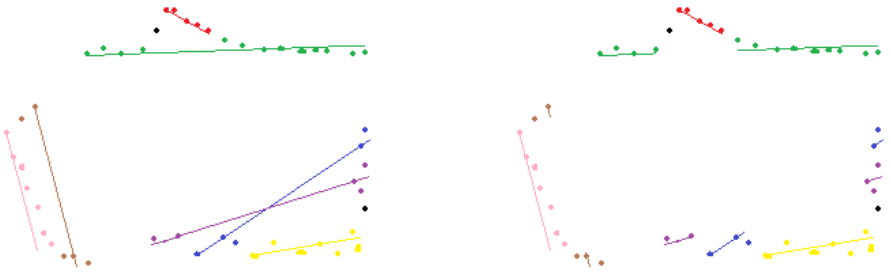
(d) Step 3: The method has been repeated until all clusters are exhausted.

Figure 21: A constructed example of how RANSAC works on the clusters. Step 2 and 3 only occur after step 1b, not 1a.

It was observed that lines would occasionally be fit which only had inliers at their extreme ends, thus, creating arguably false lines. To alleviate this, a splitting step was introduced. The naive clustering method was applied to the inlier sets, and the resulting inlier clusters were treated as separate lines. An illustration can be seen in Figure 22. It can be seen in Figure 22 that short and unintuitive lines can be created. It is expected that these will not be an issue, however, as they still manage to represent the fact that a wall exists at their placement, with greater certainty than if raw points were used.

The possibility of instead rerunning the clustering method on the raw data each time a line had been fit was also attempted. However, it was found to cause too many points to be discarded. Alternatively adding continuity as a requirement for a line to be accepted was also explored. This, unfortunately, resulted in many lines being rejected, and thus a need to run more iterations of RANSAC's step 1. This idea was thus discarded to avoid

the increase in computational costs.

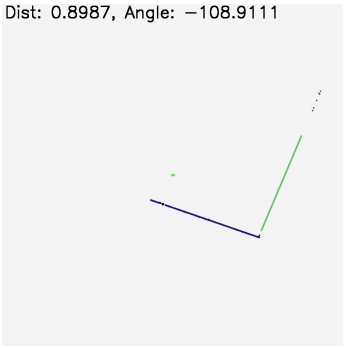


(a) Fit lines before splitting (b) Fit lines after splitting

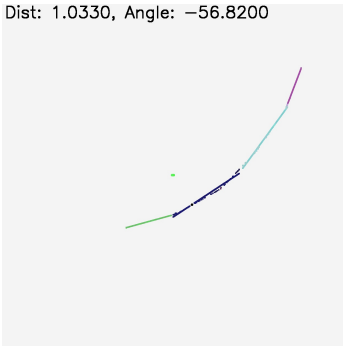
Figure 22: A constructed example of how the line splitting improves creates more representative lines. The colors indicate which points belong to which lines.

To find the distance to the fence, the two methods are proposed. The first method is simply to transform the points that have been fit to lines back into polar form, and then choosing the closest one. The other method is to find the angle and distance to the closest wall using the description of the lines. This option is used because it takes greater advantage of the robustness provided by the RANSAC algorithm.

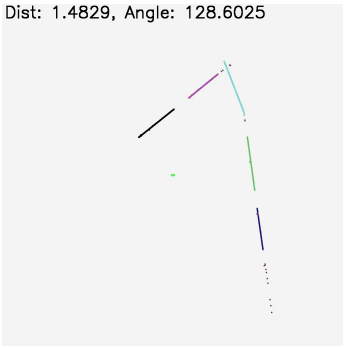
Results: To test whether the fence localization algorithm was functional, it was first tested in a simulated environment. The robot was manually driven along with a complex map, while the fence localization algorithm was run. The visualizations generated by the algorithm were then inspected and manually sanity checked. It was seen that the fence localization algorithm behaved as expected. Fence segments were occasionally split into multiple parts. This happened particularly when fence segments were far from the scanner, meaning that the point density was low. It is however not considered to have any negative effect on the performance of the fence localization method. See Figure 23, for some examples.



(a) The fence localization visualization when driving into a right angled corner.



(b) The fence localization visualization when driving along a curved fence.

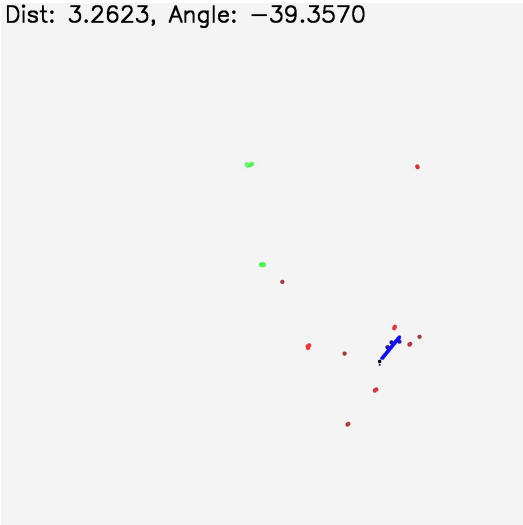


(c) The fence localization visualization when driving into a sharp corner.

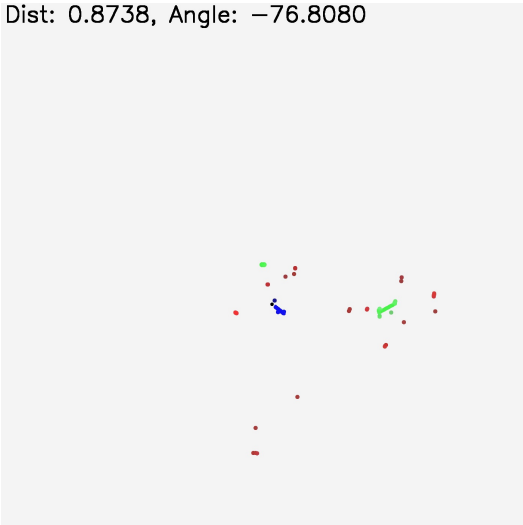
Figure 23: Example visualizations created by the fence localization method in different situations. The green marker represents the robots position. The lines represent the found lines. The point represent lidar scan points. The color indicates separate lines.

When doing preliminary testing on a real chain link fence, however, the measured fence distance was seen to be very unstable and erratic. In order to determine if the problem could arise from the lidar data simply not being good enough at representing a fence, an informal test was conducted. The Frobit with the mounted lidar was held up at roughly 180 cm from the ground, so that unevenness in the terrain would not affect the lidar data. The fence localization algorithm was then run. Visualizations generated from the lidar data and fence localization algorithm can be seen in Figure 24.

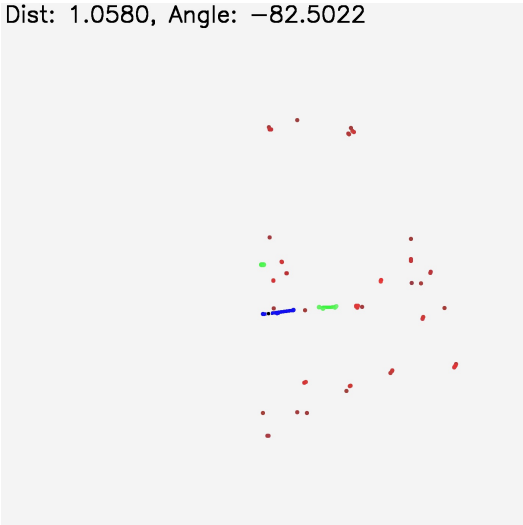
(a) A constructed example explaining the visualization and showing the desired or expected outcome.



(b) An actual lidar scan taken at roughly 2 meters distance to the fence.



(c) An actual lidar scan taken at roughly 1 meters distance to the fence.



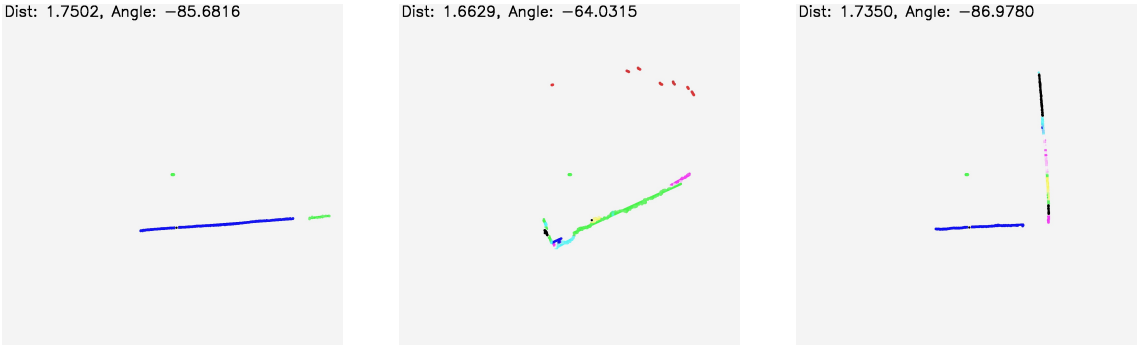
(d) An actual lidar scan taken at roughly 1 meters distance to the fence.

Figure 24: The two maps used for measuring the performance of the robot

Each colored dot represents a point, and is colored by the line it is fit to, or colored red if no line is fit to it. The robot is represented by the green dot in the center. It can be seen that the actual output does not match the desired output and that the data is not of a good enough quality, as the structure of the fence is not apparent in the pattern of where the points are found. From this, it is concluded that the lidar is simply not of high enough accuracy or resolution to enable detection of the chain-link fence. As the fence has a very sparse structure, many of the scan lines from the lidar are able to pass through the fence, thus not detecting it. With a high density of scan lines, it should be possible to counteract

this.

The method was then to be tested against a real life solid wall instead. However, it was not possible to conduct a thorough test, due to Corona pandemic restrictions. Some preliminary data was recorded and analyzed though. Some examples are shown in Figure 25.



(a) The fence localization visualization when driving along a straight wall.

(b) The fence localization visualization when driving down a hill and along a straight wall.

(c) The fence localization visualization when driving into a corner.

Figure 25: Example visualizations created by the fence localization method in different situations. The green marker represents the robots position. The lines represent the found lines. The point represent lidar scan points. The color indicates separate lines, with red points being points that were not fit to a line.

It can be seen that the method has the expected output in these situations. Similarly to the simulation, there was a tendency to generate multiple lines along the same straight wall, which is unintended but doesn't negatively impact the ability of the method to find the line point closest to the robot. The good performance in this test corroborates the idea that the failure recorded when the method was used on a fence, was caused by a too low lidar resolution.

To keep a fixed distance to the lines fitted from the fence distance algorithm, a control loop is needed. Two main methods for distance control were implemented, a traditional control method in form of a proportional–integral–derivative, PID, controller and an unsupervised artificial intelligence method in form of Input Correlation, ICO, learning.

6.1.2 Input Correlation Controller

To dynamically adapt to the potentially different shapes of the fences, the Input Correlation, ICO, learning rule can be used as a distance controller. In this project, the type of ICO controller is designed as a single layer feedforward network with unsupervised learning. This ICO controller type has two inputs, a predictive signal, and a reflective signal. The

purpose of the controller is to learn to give an output to a system so the reflective signal will never occur based on the predictive signal. The structure of a basic ICO controller is illustrated in Figure 26.

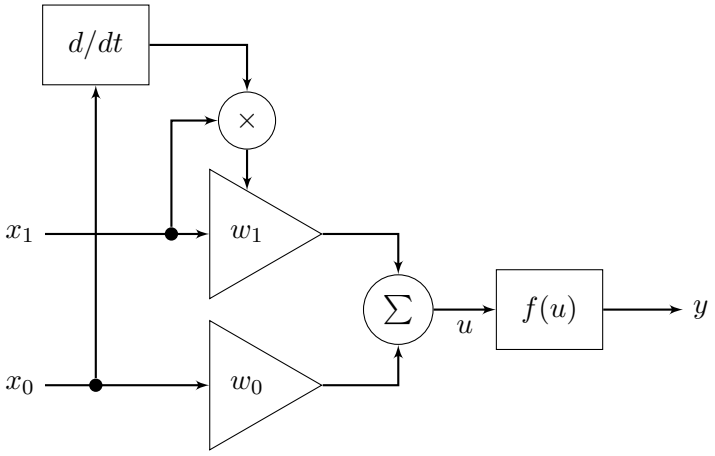


Figure 26: Mathematical structure of an single layer feedforward network ICO. x_1 and x_0 are the predictive and reflective signal, respectively, and their corresponding weights are w_1 and w_0. u indicates the output before the activation function, $f()$, and y is the final output.

The mathematical equation of the output of the neuron and the weight updates, illustrated in Figure 26, are shown in Equation 4 and Equation 5.

$$u = w_0 \cdot x_0 + w_1 \cdot x_1 \tag{3}$$

$$y = f(u) \tag{4}$$

$$\frac{d}{dt} \cdot w_1 = \mu \cdot x_1 \cdot \frac{d}{dt} \cdot x_0 \tag{5}$$

μ in Equation 5 is the learning rate. From Equation 5, it can be seen that the weights of the predictive signal are only updated when a change in reflective signal is present.

Input Correlation Distance: A single ICO controller is implemented to keep a fixed distance to the fence. It has the same structure as seen in Figure 26 with the predictive and reflective signals defined in Equation 6 and an illustration of predictive signal and

reflective signal can be seen in Figure 27

$$\text{predictive} = \text{error} = \text{target}_{\text{distance}} - \text{current}_{\text{distance}}$$

$$\text{reflective} = \begin{cases} 1 & \text{if error} > \text{learning band} \\ -1 & \text{if error} < \text{learning band} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

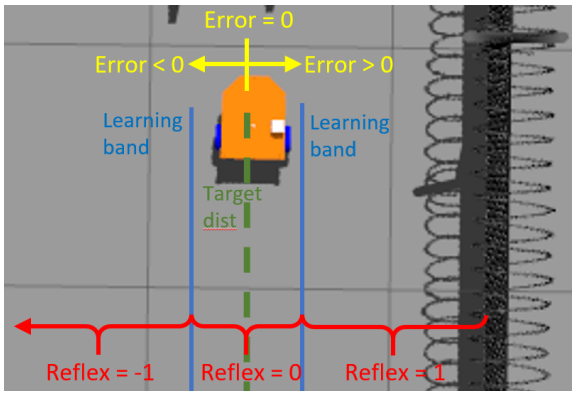


Figure 27: Illustration of predictive signal and reflective signal for the one ICO controller.

The controller uses the ICO to give a motor offset on a fixed velocity to either steer left or right based on the error is positive or negative. This is done by using *tanh* as the activation function, which characteristics can be seen in Figure 28.

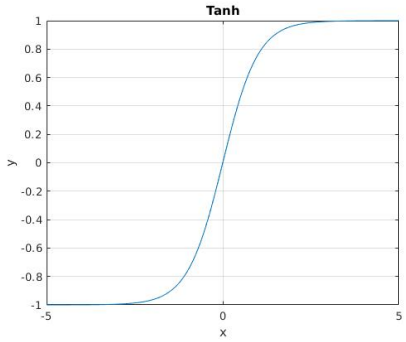


Figure 28: tanh output y as result of input x

Because the output value of *tanh* is 0 at input 0, a fixed velocity speed needs to be set at both wheels to drive with a forward velocity when the error is 0. An example of the input, weight, and output response of the ICO controller running on the Frobit in the simulation environment is illustrated in Figure 29.

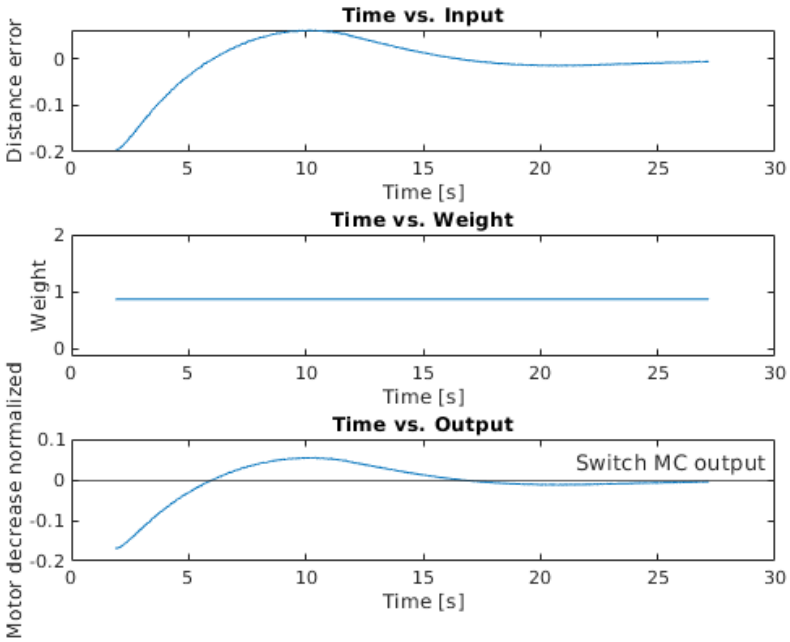


Figure 29: Example of the input, weight, and output of the ICO controller.

When the output of the ICO controller is negative, then the negated value of the output is added to the left motor to make the Frobit drive to the right. If the output is positive then the output value is added to the right motor to make the Frobit drive to the left. Both the constant speed and the output of the ICO is a pulse-width modulation, PWM, value in the range of [0 1.0]. The PWM output is then converted to a forward velocity in the x-axis direction of the Frobit and an angular velocity around the z-axis of the Frobit using Equation 7.

$$\begin{aligned} \text{forward}_{\text{velocity}} &= \frac{\text{pwm}_{\text{left}} + \text{pwm}_{\text{right}}}{2} \\ \text{angular}_{\text{velocity}} &= \frac{\text{pwm}_{\text{right}} - \text{pwm}_{\text{left}}}{\text{wheel}_{\text{dist}}} \end{aligned} \tag{7}$$

$\text{wheel}_{\text{dist}}$ in Equation 7 is set to 0.14 $[m]$ in simulation and 40 $[cm]$ in the Frobit docker in real life.

Optimal weights: From the weight update rule, shown in Equation 5, it can be seen with the chosen predictive and reflective signal that the weight will always increase. This becomes an issue when a 90° corners occur because these will always result in the robot exceeding the learning band. Therefore, for finding the optimal weights a weight optimization map is created which only contains soft corners. The map is shown in Figure 3b.

Two tests were conducted to find the optimal weights with the respective constant speed

on both wheels of the ICO controller. The first test was conducted to find the optimal weights by initializing the Frobit in the position seen in Figure 3b and then running ICO controller with weight update until the weights have converges. The weights were randomly initialized with a value between $[0.1, 0.2]$, thus, making the ICO capable of learning without the need for de-learning. The second test was conducted to evaluate the performance of the optimal weight and their respective constant speed in terms of the squared total error of the target distance and time. This was done by drive one lap around the dog bone map. In both tests were the PWM signals, before converting to forward- and angular velocities, scaled with 0.3 and limited to a maximum value of 0.5 to counteract oscillation. The target distance was set to 1.0 meter as this meant the robot could see all the fences from the camera attached in the simulation. Furthermore, the learning band was set to $\pm$ 0.2 meter of the error to give enough space so the Frobit would not get a reflective signal all the time, but also small enough so it would not learn to oscillate inside the band. The results from the first test and second test can be seen in Table 5 and Table 6 respectively.

Velocity	0.15	0.25	0.35	0.45
Completes One Lap	True	True	True	False
Weight	0.82	1.36	1.83	÷

Table 5: Results of the test with constant velocity and optimal weights for the ICO controller.

From Table 5, it can been seen that the weight increase as the constant velocity increases. This is due to the Frobit being faster when approaching a corner, thus, the output of the ICO must be higher to avoid getting into the reflective region.

Velocity [m/s]	0.15	0.25	0.35	0.45
Total squared error [m]	17.68	12.83	61.28	÷
Time [s]	519.05	309.98	242.84	÷
Unstable	False	False	True	True

Table 6: Evaluation of the optimal weight and constant velocity from Table 5. Unstable indicates that the robot does not settle at the target but keeps oscillating around it.

Table 6 shows that the constant velocity of 0.25 with the weight 1.36 has the best trade-off regarding the squared total error, the time, and does not become unstable. Thus, this constant velocity and weight will be used throughout the rest of the test in the report. Note from Table 5 that a constant velocity of 0.15 $[m/s]$ gets a higher total squared error

than a constant velocity of 0.25 $[m/s]$. This occurs because the total error is calculated as a sum of the deviation from the target distance to the fence. Thus, when driving slower the longer the robot will be deviating from the target and thereby get a higher total squared error. The input, weight, output, and reflective signal of the run on the dog bone map for the constant velocity of 0.25 can be seen in Figure 30.

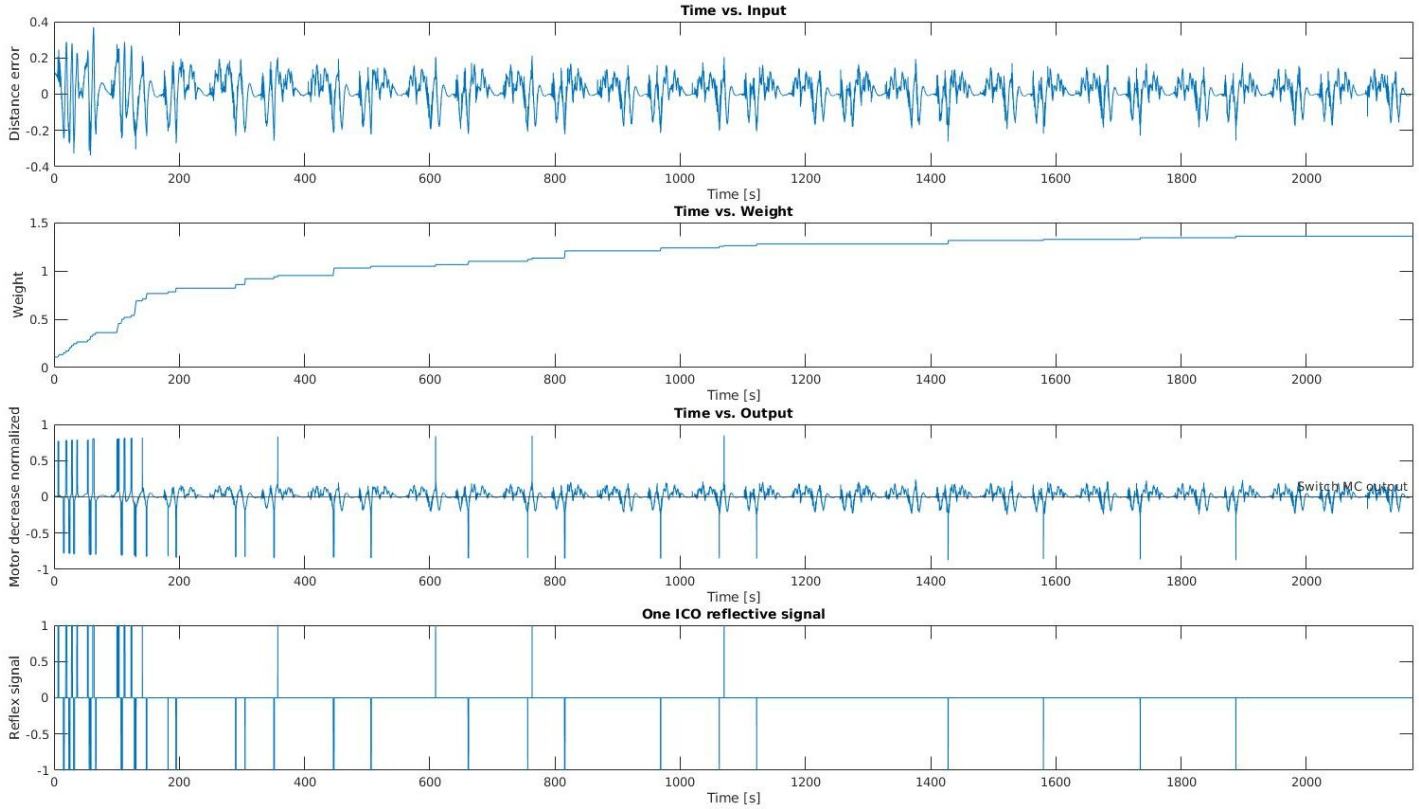


Figure 30: Logs during the run of the constant velocity of 0.25

Figure 30 shows that $\lim_{t\to+\infty}$ reflex signal $\to 0$. Furthermore, as the more the weights converges 1.36 the less spikes in output. This illustrates the ICO has learned to smoothly adjust the motor offset so the error stays inside the learning bands.

Higher velocity at straights: To reduce the time it takes to drive a lap around the weight map, a higher constant velocity is set at straight sections. The constant velocity is then

$$\text{Constant velocity} = \begin{cases} 0.5 & \text{if} -0.1 < \text{ICO}_{\text{output}} < 0.1 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

This gives a square total error of 16.90 meters and a lap time of 194.87 seconds, which is a significantly decrease in lap time with little increase in error. Thus, this velocity will be used along with the found optimal constant velocity and weight for the rest of the report.

A gif running the optimal weight and the higher velocity at straights can be seen here[6]

Acceptance test: To validate whether the controller would be applicable for use around the airport, the Frobit was run in simulation on the map shown in Figure 4b. The ICO controller ran a successful lap and the results can be seen in Table 7 and in Figure 31.

Cumulative squared distance	Lap time	Estimated average velocity
127.9 $[m]$	1139.7 $[s]$	0.2194 $[m/s]$

Table 7: Results of the ICO controller run on the airport map

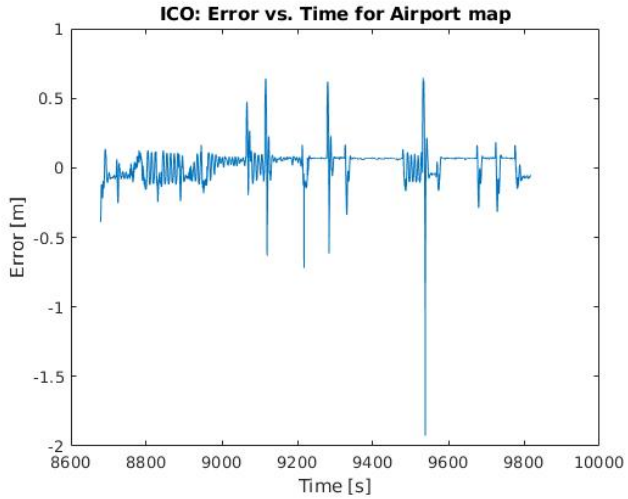


Figure 31: The error from a target distance of 1 $[m]$ to the fence using the ICO controller on the airport map

From the results it can be concluded that the ICO controller can do one lap on the down-sized airport map, this can be seen here[7]. Furthermore is can be seen from the error plot that the error mostly oscillate within 0.5 $[m]$, which is deemed reasonable.

6.1.3 Proportional Integral Derivative Controller

One method of controlling the distance to the fence is with the use of a PID controller. This controller is a good candidate, as it is fairly simple and very well understood, as it has been used for over a hundred years. In this case, the PID will control the desired angular velocity around the z-axis of the robot, and use the desired fence as its set point. The forward velocity of the robot will be set to constant. A block diagram of a PID controller is shown in Figure 32.

[6] https://github.com/mikkellars/EiT_project/blob/main/assets/dog_bone_ico_optim.gif
[7] https://github.com/mikkellars/EiT_project/blob/main/assets/airport_sim.gif

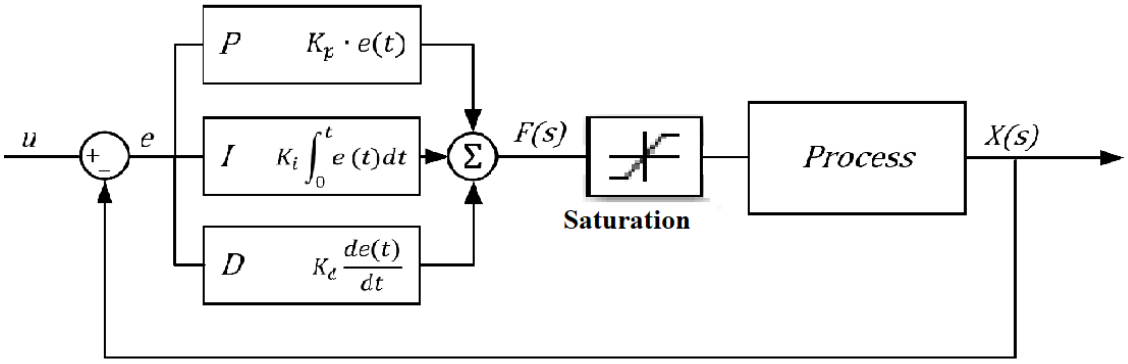


Figure 32: A block diagram of a PID controller with saturation.

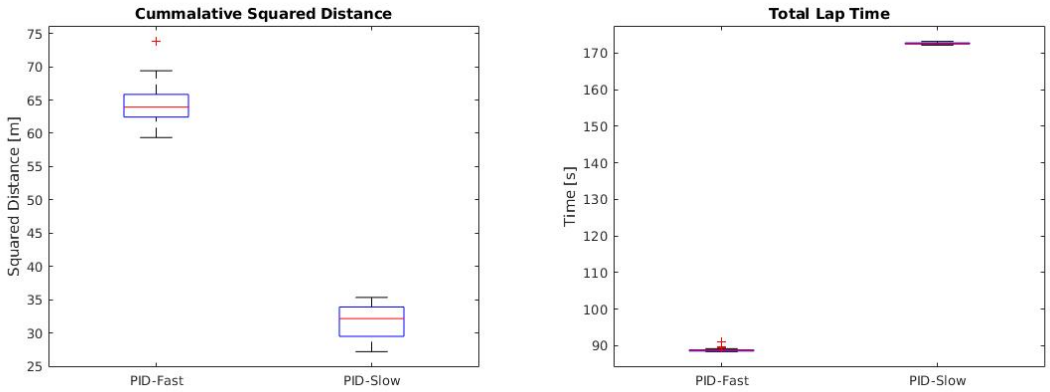
Commonly, a PID controller will be used to directly control a motor PWM or other actuator. That is not the case here for a few reasons. Firstly, this PID was first developed and tuned to work on the simulated Frobit platform, which does not support a PWM signal. Secondly, the Frobit and many other platforms already contain well-tuned motor controllers. These motor controllers allow the user to the only interface with the abstraction of cartesian and rotational velocity. This should mean that a good velocity controller should be able to be ported between different robots with different physical characteristics. It does however introduce some source of instability, that the PID controller is using a positional set point, but a rotational velocity output. In particular, the robot can begin spinning in circles if the output is too great. To limit this issue, a saturation value was set as the limit to the output. This saturation value was tuned alongside the terms of the controller.

Tuning: Typically, a PID would be tuned by analysis of the physical system, and method such as the root locus. However, the controller is not being tuned to directly control the physical system, but rather to control what is essentially the set point of an underlying velocity controller. For this reason, the PID is instead tuned by hand.

It is observed that the proportional, P, term needs to be fairly large, in order for the Frobit to turn fast enough in corners, to avoid collisions or loss of sight of the wall. However, a very large P term makes the system unstable, as the controller overshoots and oscillates. In order to counteract this, a fairly large derivative, D, term must be used as well. With these two terms however, the controller is still very slow to eliminate small errors. Therefore, is the integral, I, term introduced to eliminate such errors. However, the I term can also make the controller fail when faced with corners which go in the opposite direction of the

error which has accumulated within the I term. In such situations, the I term counteracts the P term, making the robot turn too slowly, and either hit or lose sight of the wall. A very small I term is therefore used. See Table 10.

The output saturation value was set to the greatest possible value which allowed the robot to turn fast while also remaining stable. The forward velocity of the robot was also tuned. Again, the goal was to maximize its value without introducing instability. It was seen that a large forward velocity would make the robot turn much slower than it was ordered to. It is suspected that this is due to the limitations of the underlying system. As the underlying system might down-prioritize turning velocity in favor of trying to reach the high forward velocity set point given to it. A forward velocity of 0.2 $[m/s]$ was found to be the highest stable value. In order to test the performance of the controller at this forward velocity compared to a lower one, a test was conducted. The controller was run 30 rounds with two speed settings (0.1`m/s and` 0.2`m/s`) inside a square fence. See Figure 3a. A logger was set up to record the content and timing of each message from the fence localization algorithm. The total time and cumulative error of each round was then recorded as a single observation, so that a sample of size 30 was recorded for each forward velocity setting. An F-test was first conducted to test for equal variance among the two samples. The test did not show a statistically significant difference ($p = 0.1625$) between the two samples cumulative squared distance. However, regarding the lap time it showed statistical significance ($p < .0$). This meant a two-sample student t-test with equal variance was done for the cumulative squared distance and one with unequal variance was done for the lap time. This test showed statistical significance ($p < .0$) for both the cumulative squared distance and the lap time.



(a) Box plot of the total squared error.

(b) Box plot of the total lap time.

Figure 33: Box plot of total squared error and total lap time for PID fast and PID slow.

Method	Mean	Std	Confidence interval 95% mean
PID-Fast	64.4049 $[m]$	3.0034 $[m]$	[63.2835 $[m]$, 65.5264 $[m]$]
PID-Slow	31.9371 $[m]$	2.30858 $[m]$	[31.0751 $[m]$, 32.7992 $[m]$]

Table 8: Mean, standard deviation and confidence interval for cumulative squared distance.

Method	Mean	Std	Confidence interval 95% mean
PID-Fast	88.8059 $[s]$	0.508 $[s]$	[88.6162 $[s]$, 88.9956 $[s]$]
PID-Slow	172.628 $[s]$	0.219 $[s]$	[172.547 $[s]$, 172.71 $[s]$]

Table 9: Mean, standard deviation and confidence interval for lap time.

It can be seen that both the speed and the error is significantly higher. This is deemed as a worthwhile trade-off. A more in-depth analysis of the error might be worthwhile at a later time. See Table 10 for an overview of the final values selected for each term.

Term	P	I	D	Sat.	Vel.
Simulation	1.2	0.02	1.5	1.2	0.2
Real life	0.015	0.001	0.01	0.04	0.5

Table 10: Overview of final values selected for each term. Sat. is the saturation value and Vel. is the constant target forward velocity.

It can be seen from Table 10 that two different sets of values were used in simulation and real-life operation. There are multiple reasons for this. Firstly, the real robot was much more sensitive than the simulated robot and appeared to turn much faster than its given velocity set point. This is very unfortunate, as it removes the usefulness of the velocity controller as an abstraction between the fence distance controller and the physical system. Secondly, the forward velocity used in the simulation did not function on rough terrains such as a gravel road or grassy field. The robot would simply get stuck on these obstacles, while the integral terms of the fence distance controller and the underlying system would build up massively. A higher forward velocity set point was therefore needed.

The PID is also running once on the large simulated map shown in Figure 4b, with the time and error logged. This is done to verify that the controller can handle a more difficult map with curved fence segments and sharp corners. The controller managed to navigate this map successfully. The time and cumulative error can be seen in Table 11 and Figure 34.

Cumulative squared distance	Lap time	Estimated average velocity
1542.1 $[m]$	644.0 $[s]$	0.3727 $[m/s]$

Table 11: Results of the PID controller run on the airport map

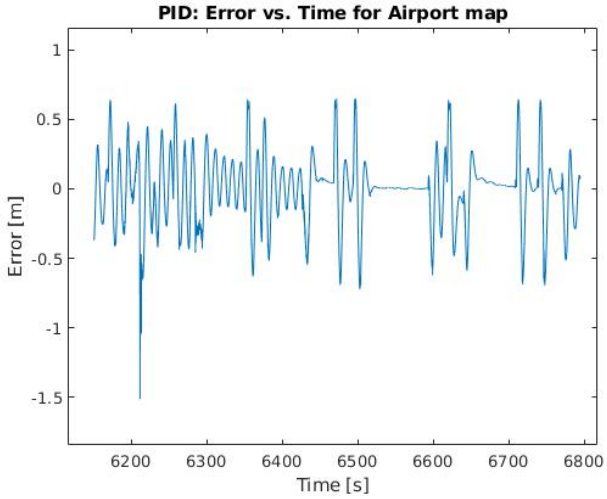


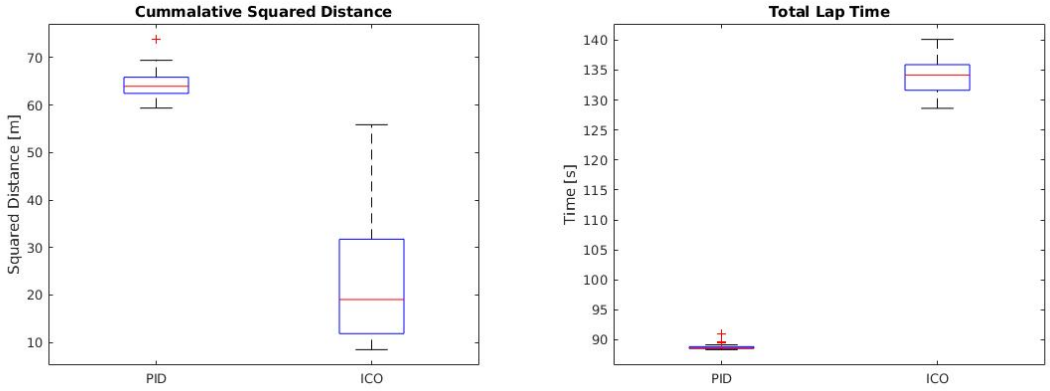
Figure 34: The error from a target distance of $1[m]$ to the fence using the PID-fast controller on the airport map

6.1.4 Comparison of Controllers

The performance of the PID and the ICO controller is evaluated in simulation, due to the restriction of the corona. The performance of the controllers is measured in time per lap and the squared cumulative error from the desired target distance to the fence. The controllers are evaluated on a simple square fence map which is illustrated in Figure 3a. A gif showing the ICO controller running on the square map can be seen here[8]

This map is chosen because if the controllers can handle the 90° turn, they are also able to handle less sharp turns. The two controllers are evaluated 30 times each from the same starting position on the map. A boxplot of the total squared error and the total lap time for each of the 30 runs for the controllers can be seen in Figure 35.

[8] `https://github.com/mikkellars/EiT_project/blob/main/assets/sqare_map_ico.gif`



(a) Box plot of the total squared error.

(b) Box plot of the total lap time.

Figure 35: Box plot of total squared error and total lap time for ICO- and PID-runs.

The box plots indicate the PID controller is the worst performer in terms of the squared distance, but the fastest in terms of the lap time. However, to confirm statistical significance a two-sample t-test is conducted on the samples from the PID- and ICO- controller. This is done by firstly confirm the samples from both the PID and the ICO controllers are normally distributed. By looking at the normal quantile plots in Appendix C both the samples from PID- and ICO-controller are confirmed to be normally distributed. Next, an F test is conducted to test whether the two independent samples have the same variance, against the alternative that they have different variances. The F-test indicated a statistical significance both regarding the cumulative squared distance ($p < .001$) and the total lap time ($p < .001$), meaning the samples does not have the same variance. Thus, a two-sample t-test with unequal variance is conducted. This test also showed a statistical significance both regarding the cumulative squared distance ($p < .001$) and the total lap time ($p < .001$), meaning the means of the two samples are not the same. The mean, standard deviation and confidence interval is shown in Table 12 and Table 13.

Method	**Mean**	**Std**	**Confidence interval 95% mean**
PID	64.4049 $[m]$	3.0034 $[m]$	[63.2835 $[m]$, 65.5264 $[m]$]
ICO	22.6467 $[m]$	13.2713 $[m]$	[17.6911 $[m]$, 27.6023 $[m]$]

Table 12: Mean, standard deviation and confidence interval for cumulative squared distance.

Method	Mean	Std	Confidence interval 95% mean
PID	88.8059 [s]	0.508 [s]	[88.6162 [s], 88.9956 [s]]
ICO	134.159 [s]	3.06983 [s]	[133.013 [s], 135.305 [s]]

Table 13: Mean, standard deviation and confidence interval for lap time.

From the results can it be concluded that the PID controller has a faster lap time at the expense of having a higher cumulative squared distance. However, it can be seen from the Table 12 and Table 13, that the ICO controller has a higher standard deviation, thus the PID controller is more consistent. Based on the results from the simulation the ICO would be chosen as the implemented controller, however, a real-life test should also be made before making the decision.

6.2 Simultaneous Localization and Mapping

Another approach to navigate around a fence is to construct a map of the unknown environment, to do this Simultaneous Localization And Mapping, SLAM, is used. However, this requires that the robot navigates around in the environment to construct the map. To do so the manual controller could be used or the robot could start by doing fence following and then, when a map is constructed, the map could be used for path planning along the fence. Having a map of the environment would gain the system more consistency and is preferable. The implemented SLAM is split into localization and mapping. Lastly, the path planning with the constructed map is elaborated.

6.2.1 Mapping

In order to autonomously navigate in an environment, a robot needs a map of the environment. That map then in turn will be used by the robot to localize itself and plan a collision free path from its current location to a desired location. There are various approaches for mapping an environment using various sensor combinations. Mapping can be done in 2D or 3D, it can be done using lidar or cameras. For this project, the 2D Mapping approach using a lidar was chosen. The reason for choosing 2D mapping instead of 3D is that this project is based on a ground robot which will always drive on the road. Therefore, there will not be any movement of robot in z-axis, only a negligible jerky motion because of uneven surface might be there. 3D mapping approach will make more sense if the platform being used for this project was an unmanned aerial vehicle.

ROS already has multiple implementations of mapping algorithms for example, GMapping, Cartographer, Hector SLAM, etc. For this project, the GMapping algorithm was chosen as a mapping algorithm because it is quite simple to set up, actively maintained and properly

documented. GMapping subscribes to \tf and \laser \scan topic and publishes the created map on \map topic which is then saved by running map_server node.

The most common way to test the performance of algorithms which constantly subscribe to various topics to get continuous readings, is to record rosbags. The rosbag contains all the reading from the desired topics needed by the algorithm. The robot in this case was manually moved along the fence in the environment via *teleop_twist_keyboard*.the recorded rosbag was then used to tune the Gmapping algorithm to create a 2D occupancy grid map of the environment.

Mapping: The resulting maps created for different environments are shown in Figure 36.

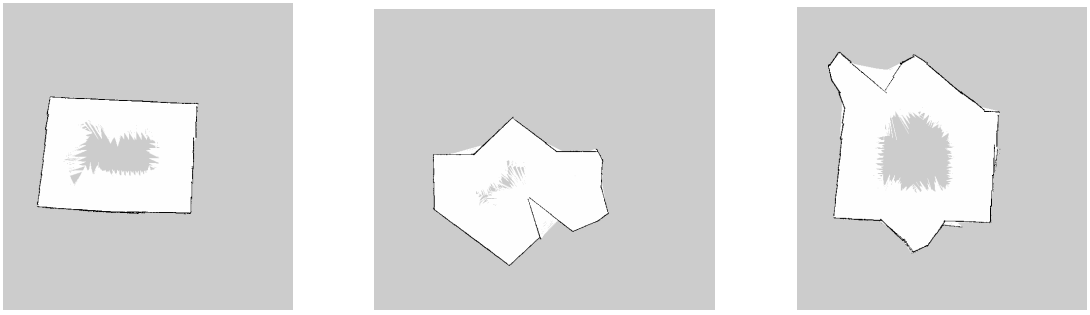
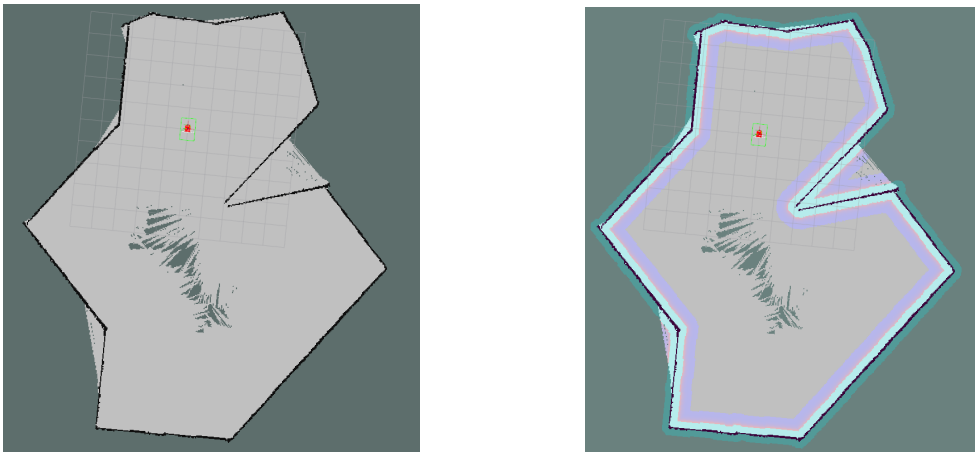


Figure 36: 2D Maps of the environments.

Global Cost Map: is used for global planning, meaning creating long-term plans over the entire environment. This defines the clearances around the boundaries and static obstacles already present in the environment. The robot's foot print should be clear of those boundaries defined by global cost map for the navigation stack to work properly and to avoid collisions with the walls. The following figures, in Figure 37, shows the environment map with and without global cost maps:



(a) Without Global CostMap. (b) With global cost map.

Figure 37: Comparison of maps with and without Global cost map.

Local Cost Map: is used for local planning and obstacle avoidance. It defines the clearance around both the robot foot print, and any obstacle in front of it. Local cost maps are shown in the figures below:

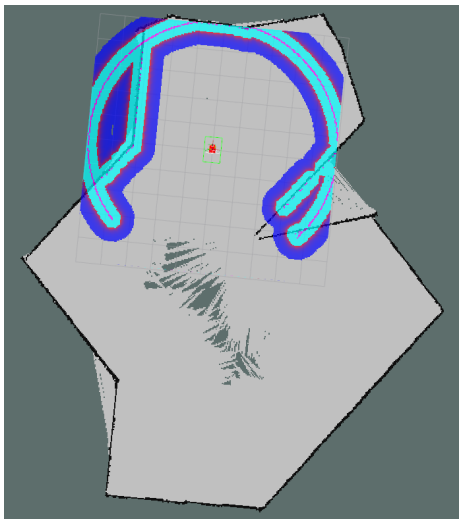
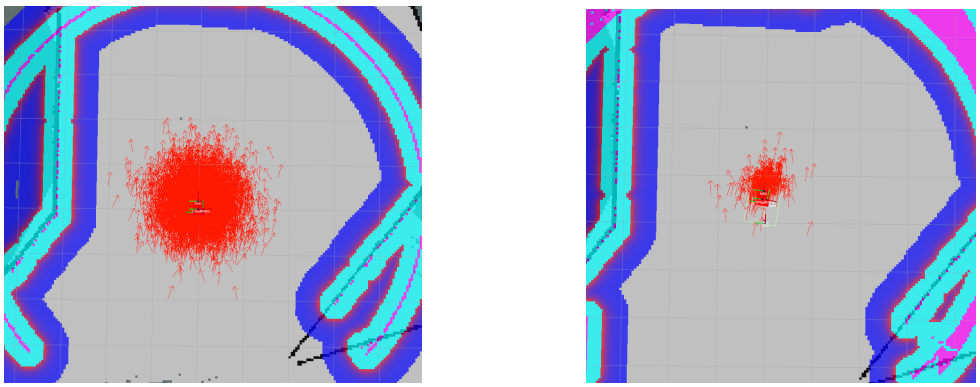


Figure 38: Local Cost Map.

6.2.2 Localization

Once the map of the environment is ready, it can be used for autonomous navigation by the robot. In order to plan its path from a start position to desired position, the robot needs to know where it is in the environment i.e. localize itself in the map. ROS has an open source implementation of Monte Carlo Localization. It is called Adaptive Monte Carlo Localization, AMCL. The AMCL package maintains a probability distribution over the set of all possible robot poses, and updates this distribution using data from odometry

and laser scan. The package also requires a predefined map of the environment against which to compare observed sensor values. The filter is "adaptive" because it dynamically adjusts the number of particles in the filter: when the robot's pose is highly uncertain, the number of particles is increased; when the robot's pose is well determined, the number of particles is decreased. This enables the robot to make a trade-off between processing speed and localization accuracy. Figure 39 show this behaviour.

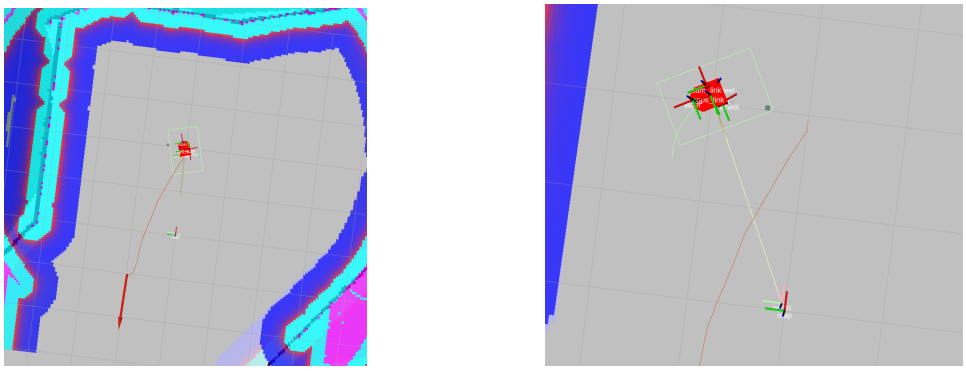


(a) Pose with uncertainty. (b) Less uncertain pose after movement.

Figure 39: Comparison of uncertainty in poses of robot calculated before and after some movement in the map.

6.2.3 Path Planning

Once the robot knows it starting location, the next step is to plan and execute a path autonomously to a desired location. There are mainly two kind of path planners i.e global and local. A global planner, plans its path from a starting point to the end location depending upon the information available at the time of planning. Whereas, the local planner takes into consideration any dynamic obstacle or static obstacle which comes in its way while it is executing the global path. ROS Navigation Stack has multiple implementations of path planners. In this project A Star is used as a global planner whereas Dynamic Window Approach, DWA, local planner is implemented.



(a) Global Path (Red). (b) Local Path (Green).

Figure 40: Visualization of global and local path generated by ROS navigation stack. Goal position is shown by purple arrow. Global path is shown in Red line and local plan is shown in green line

6.3 Navigating in Real Life

Since the robot must be able to drive in a real environment, this is investigated in this section. For the test, the PID navigation method is used at different location around SDU. Examples of the robot following a fence is shown in Figure 41, the full video can be found here[9].

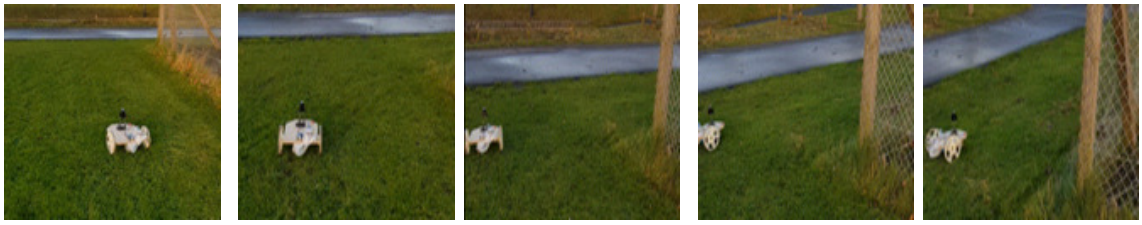


Figure 41: Sequence of images showing the robot driving along a fence.

Figure 41 shows the robot following a fence and driving around a corner. Thus, the robot is capable of driving on grass. However, some issues occurred when driving along a fence because the lidar not being able to estimate the fence correctly. In Figure 41, the density of the fence was big enough because of the grass growing close to the fence.

A preliminary test was also done in to test the navigation in real life along a solid wall. This was done to test the navigation in an environment where the fence localization algorithm would behave more consistently. The chosen building was the Mærsk Mc-Kinney Møller Institute, MMMI, building at SDU. It was chosen because of its angular design and mostly featureless facade, with no glass windows reaching the floor or outcroppings from the building. A satellite image with the desired route marked can be seen in Figure 42.

[9] `https://github.com/mikkellars/EiT_project/blob/main/assets/real_life_fence_follow.gif`

Figure 42: A satellite image of the route along the MMMI building.

The distance error over time is shown in Figure 43. It can be seen that a large error is introduced when crossing the small hill and at each turn. The error then oscillates for two or three periods before settling.

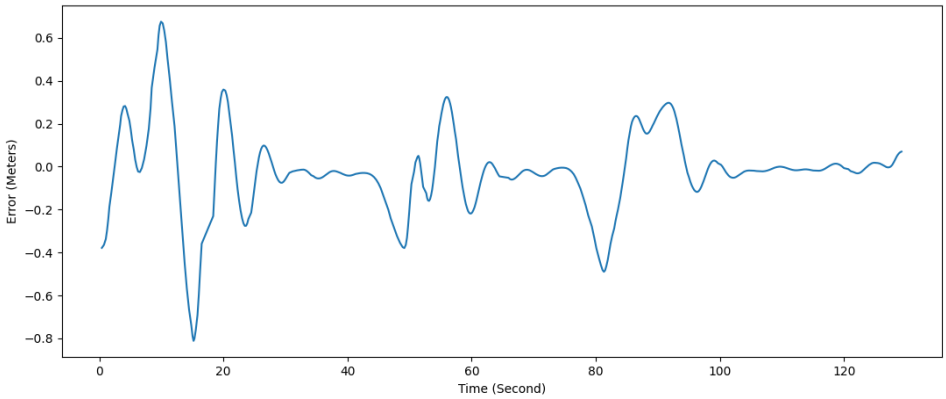


Figure 43: The error over time when driving along the MMI building.

Videos of the robot driving at Figure 42 can be found here[10] and here[11], where the first video shows a corner sequence and the second shows a straight sequence.

Unfortunately, a thorough test was not possible due to restrictions passed to combat the spread of the novel corona virus.

7 Budgeting

As this project was done in collaboration with the company Lorenz Technology the financial viability is important. An analysis was done in the expert in teams report and the analysis has been included in Appendix D.

[10] `https://github.com/mikkellars/EiT_project/blob/main/assets/real_life_corner.gif`
[11] `https://github.com/mikkellars/EiT_project/blob/main/assets/real_life_straight.gif`

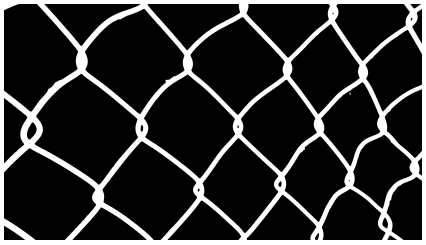
8 Discussion

Fence Segmentation for Fence Structure Analysis

In subsection 5.1, an approach to segment fence in an image was developed. However, this did not show good results on real images only on the dataset images. Therefore, if this approach should be developed further a dataset with pixel label precision must be created and because of time constraints, this was not done in this project even though the approach showed promising results shown in Figure 12.

Fence segmentation could be a valid approach when creating a system for PIT Hegn since they repair fences that have a damaged structure. Thus, by looking at the segmented fence a method to predict if the fence has a damaged structure could be developed, an example of this is illustrated in Figure 44b.

(a) Input image of fence.



(b) Segmentation of fence.

Figure 44: Example of input image of fence with corresponding segmentation that shows the structure of the fence.

In Figure 44, the input image Figure 44a shows a fence with uneven structure and the segmentation Figure 44b shows the structure of the fence. This could then be used to estimate if the fence needs to be replaced, because of the structure being damaged, by looking at Figure 44b and comparing it to a fence structure with no damage. Then, depending on the difference, the system would predict if the fence needs replacement.

Limitation of Lidar Scans

Because of the chosen lidar having a 240° scanning range a fence corner that has an angle of $< 90°$ could become a problem. This problem is illustrated in Figure 45. This confirms along with the resolution problem of the lidar that another sensor should be used.

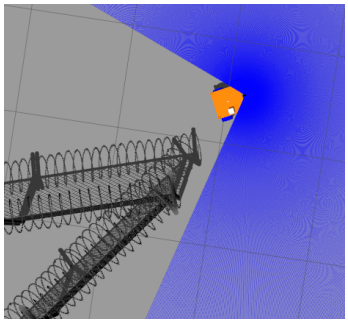
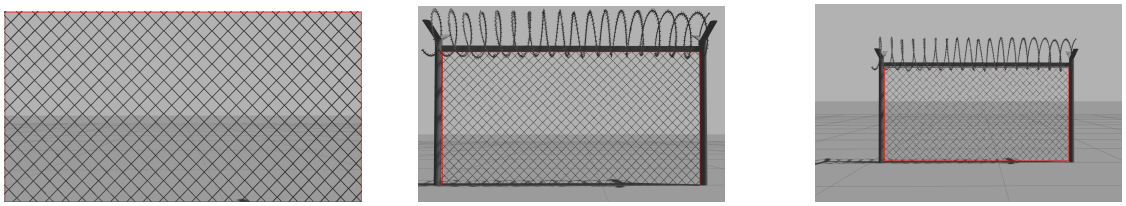


Figure 45: Illustration of the lidar not being able to detect any fence because of a too small field of view.

In subsubsection 6.1.1, a problem occurred when driving in real life with the lidar. The problem being that the lidar could not register the fence, thus, the fence could not be located and the robot would drive into it. A higher resolution lidar would be needed to detect the fence.

Keeping a Fixed Distance with Camera

An alternative to using the lidar for fence localization could be to use the camera instead, in Figure 17 examples of images captured with the camera can be seen. By using an object detector like the ones in subsection 5.2, but instead of finding holes, it would find the fence. The aspect ratio of the detected bounding box, as illustrated in Figure 46, could be used to estimating if the robot is too close or too far from the fence.



(a) Too close to fence. (b) A good distance to fence. (c) Too far from fence.

Figure 46: Illustration of how object detection, i.e. the red box, of fence could be used to estimate the distance to the fence.

Figure 46 illustrates how the detected bounding box of the fence could be used to indicate if the robot is driving too close to the fence or too far. Figure 46a illustrates the robot being to close to the fence, Figure 46b illustrates the robot having a good distance to the fence, and Figure 46c illustrates the robot being to far from the fence.

Two Input Correlation Controllers

A two ICO controller was implemented as a competing controller for keeping a fixed distance to the fence. This controller used two ICO's, one for each wheel, which focused on

controlling the wheel the ICO was assigned to. Having an ICO assigned to each wheel and using the activation function *sigmoid*, made it possible for the ICO's control the velocity at all times, which compared to the one ICO case needed a constant velocity. However, while results from the simulation environment showed the two ICO controllers learned a forward velocity, the control input when steering towards the target distance made the Frobit oscillate. With more testing time it should be possible to tune the controllers to become stable.

GPS Signal

Due to time constraints, a GPS driver and a GPS ROS node were not fully developed. The reasoning behind having a GPS module running on the Frobit, was to geotag all the images with holes. Thus, it would be easy afterward to upload to a web-page, e.g. pic2map.com, for showing the location of the holes.

Choice of Controller Type

In subsubsection 6.1.4 the PID and ICO controller were compared against each other in the simulation environment to chose one of them to be in a potential final product. The test showed that the cumulative squared distance was larger using the PID controller compared to the ICO controller. However, running the controllers it was seen the ICO controller was way more aggressive in its turn than the PID which path was more smooth. Thus, by running the controllers on a square map, the PID controller got a higher cumulative square distance in the corners, because it controlled the Frobit more smoothly.

In the future, the path taken by the ICO and PID controller should be plotted and the evaluation process should take these plots into account. As the PID relies on the underlying velocity controller to behave similarly on different robot platforms, it would also be useful to investigate to what degree that is the case. It could be seen that the Frobit velocity controller would cause the robot to rotate at much greater velocities than expected. The PID would need to be re-tuned for every new platform which had that unfortunate behavior. In such a case, it would be more efficient to use the ICO controller, as it could be adapted to different controllers with much less effort.

9 Conclusion

A mobile robot system was developed and implemented in ROS and docker. The drivers for each hardware module are in their own container and can easily be changed if needed. Furthermore, the fence inspection application was also implemented in its own docker, thus, allowing for an easy change of application if needed.

Visual inspection methods were developed both for running locally on the robot and for remotely on a workstation. The methods were tested in daylight and detected anomalies on the fence, in the form of a plastic bag, with a low miss rate. Furthermore, did each method detect the holes once in the sequence of images. The methods were able to detect holes with a minimum size of 30×30 centimeter with a prediction time for one image of maximum 8.10 milliseconds. The evaluation of the methods showed that the camera, Raspberry Pi Camera Board v2.1, is deemed acceptable for detecting anomalies in fences.

Two approaches for following a fence were developed and implemented, ICO and PID, both were able to follow a fence using a lidar in each environment they were tested. This made the robot able to keep an approximately fixed distance to the fence while driving along with it. A statistical test showed the PID controller was 33.8% faster than the ICO controller to do one lap around the square map in simulation. However, the ICO controller had a 64.8% smaller cumulative squared distance error than the PID controller, which indicated the ICO controller was better at holding the target distance. The experiments in the real world showed that the lidar, Hokuyo URG-04LX-UG01, could not register the fence, and therefore not locate it. Thus, is the lidar not deemed acceptable for locating chain-link fences.

One approach for autonomous navigation using SLAM was implemented in simulation. The robot was able to construct the map of the desired environment and localize itself in it. This enabled the robot to plan its collision free path from its starting position to any desired location, and execute it autonomously. Because of time constraints, this approach was only tested in simulation, and not in real life.

For future work should the fence following be put together with the construction of the map. In doing so it should be estimated if the robot can report breaches within 24 hours and perform inspection three times a day. Furthermore, must a killswitch be implemented for safety reasons. However, due to time limitations and corona restrictions, this is due for another time.

References

[1] Wikipedia. *MoSCoW method*. Sept. 24, 2020. URL: https://en.wikipedia.org/wiki/MoSCoW_method.

[2] awaisamjad66a. *Steel Fence 3D Model*. Dec. 31, 2020. URL: https://free3d.com/3d-model/steel-fence-3841.html.

[3] RaspberryPi.dk. *ZeroCam NightVision*. Dec. 27, 2020. URL: https://raspberrypi.dk/produkt/zerocam-nightvision/.

[4] RaspberryPi.dk. *Raspberry Pi Camera Board v2.1 (8MP)*. Dec. 27, 2020. URL: https://raspberrypi.dk/produkt/raspberry-pi-camera-board-v2-1-8mp/.

[5] HOKUYO. *URG-04LX-UG01*. Dec. 29, 2020. URL: https://www.hokuyo-aut.jp/search/single.php?serial=166.

[6] Inc. Docker. *docker*. Dec. 27, 2020. URL: https://www.docker.com/.

[7] CAPRA ROBOTICS. *CAPRA ROBOTICS, Our Product*. Dec. 27, 2020. URL: https://capra.ooo/.

[8] Open Robotics. *ROS*. Dec. 27, 2020. URL: https://www.ros.org/.

[9] ros-drivers. *hokuyo node*. Dec. 27, 2020. URL: https://github.com/ros-drivers/hokuyo_node.

[10] Ubiquity Robotics. *Raspicam Node*. Dec. 29, 2020. URL: https://github.com/UbiquityRobotics/raspicam_node.

[11] Henrik Skov Midtiby Henrik Egemose Schmidt. *Drone Inspection of Fences - Periodic Structure Detection*. Dec. 11, 2020. URL: https://www.syddanskuniversitet.eu/-/media/files/om_sdu/centre/c_uas/research+projects/drone+inspection+of+fences.pdf.

[12] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[13] chen-du. *De-fencing*. Dec. 28, 2020. URL: https://github.com/chen-du/De-fencing.

[14] Alexander Buslaev et al. "Albumentations: Fast and Flexible Image Augmentations". In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125.

[15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[16] Liang-Chieh Chen et al. "Rethinking atrous convolution for semantic image segmentation". In: *arXiv preprint arXiv:1706.05587* (2017).

[17] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[18] University of Michigan Justin Johnson. *Lecture 15: Object Detection*. Dec. 20, 2020. URL: https://web.eecs.umich.edu/~justincj/slides/eecs498/FA2020/598_FA2020_lecture15.pdf.

[19] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[20] Ultralytics LLC. *YOLOv5*. Dec. 11, 2020. URL: https://github.com/ultralytics/yolov5.

[21] Yunyang Xiong et al. “MobileDets: Searching for Object Detection Architectures for Mobile Accelerators”. In: *arXiv preprint arXiv:2004.14525* (2020).

[22] Coral. *USB Accelerator datasheet*. Dec. 28, 2020. URL: https://coral.ai/docs/accelerator/datasheet/.

[23] Alasdair Allan. *The Big Benchmarking Roundup*. Dec. 29, 2020. URL: https://www.hackster.io/news/the-big-benchmarking-roundup-a561fbfe8719.

[24] Rasmus Laurvig Haugaard. *synth-ml*. Version 0.0.1. Jan. 30, 2020. URL: https://gitlab.com/sdurobotics/vision/synth-ml.

[25] *GIMP*. Dec. 20, 2020. URL: https://www.gimp.org/.

Appendices

A Workload for Each Contributor

In this section the workload of each contributor is stated. Firstly, the workload of implementing the different methods are shown in Table 14 and lastly, the workload of written the different sections are shown in Table 15.

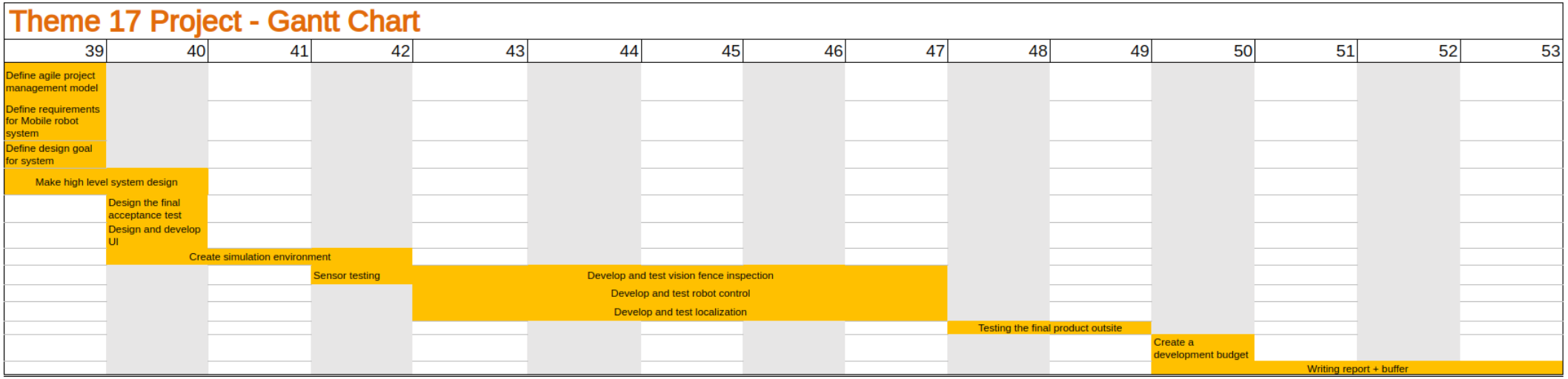
Contributor	Johan	Mathias	Mikkel	Shaheer
Simulation Setup			×	×
Platform Setup		×	×	
Fence Segmentation		×		
Faster RCNN & Yolov5		×		
SSD Lite MobileDet			×	
Localization of Fence	×			
ICO			×	
PID	×			
SLAM				×

Table 14: The workload for each contributor in regards of the project.

Contributor	Johan	Mathias	Mikkel	Shaheer
Abstract			×	
Introduction	×	×	×	×
Project Description	×	×	×	×
Simulation				×
Platform		×		
Inspection of Fence		×		
Localization of Fence	×			
Input Correlation Controller			×	
Proportional Integral Derivative Controller	×			
Comparison of Controllers			×	
Simultaneous Localization and Mapping				×
Navigating in Real Life	×	×		
Budgeting	×			
Discussion	×	×	×	×
Conclusion	×	×	×	×

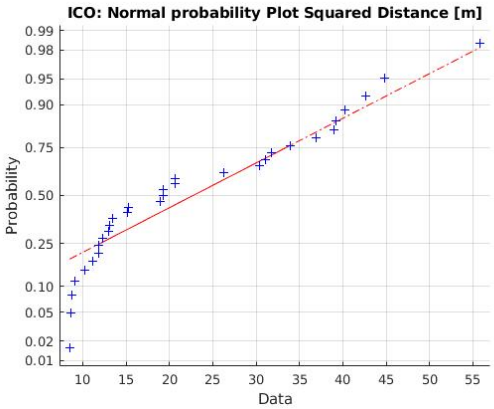
Table 15: The workload for each contributor in regards of the report.

B Timetable

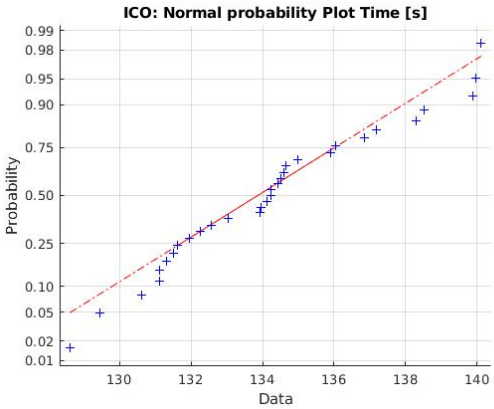

Theme 17 Project - Gantt Chart
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
Define agile project management model
Define requirements for Mobile robot system
Define design goal for system
Make high level system design
Design the final acceptance test
Design and develop UI
Create simulation environment
Sensor testing
Develop and test vision fence inspection
Develop and test robot control
Develop and test localization
Testing the final product outsite
Create a development budget
Writing report + buffer

C Controllers evaluation results

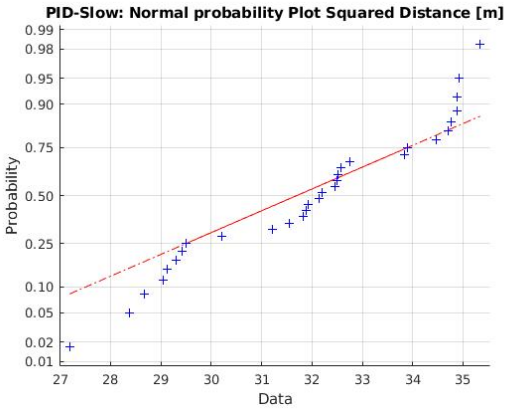
Normal quantile plots from the 30 runs on the square map seen in Figure 3a.



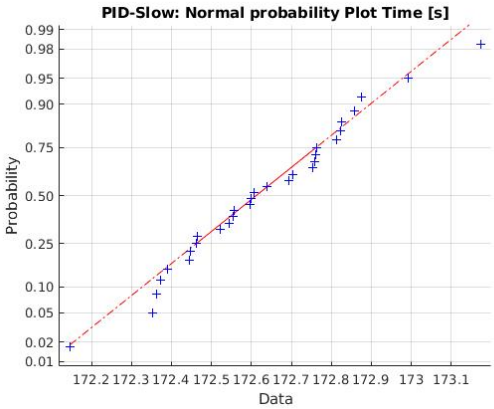
(a) Normal probability plot squared distance of ICO.



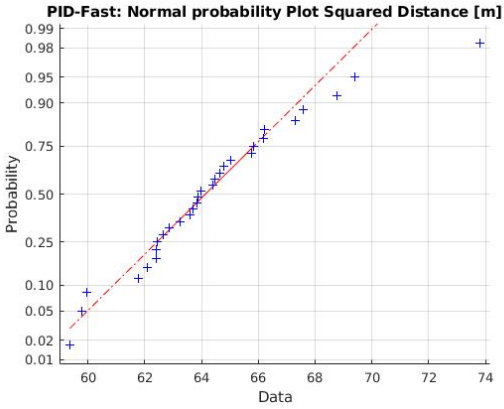
(b) Normal probability plot lap time of ICO.



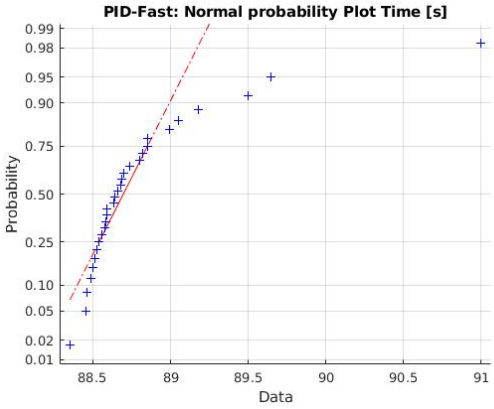
(c) Normal probability plot squared distance of PID-Slow.



(d) Normal probability plot lap time of PID-Slow.



(e) Normal probability plot squared distance of PID.



(f) Normal probability plot lap time of PID.

Figure 47: Normal quantile plot of the 30 PID- and ICO- runs

D Budgeting

A estimation of the price of the project was done in the expert in teams report to give the intended company an understand whether the project was viable or not. This estimation can be shown below and is directly taken from the expert in teams report.

Development Budget: The two major cost from the perspective of the developer is expected to be software development and hardware prices. The project would require significant testing and certification. Therefore, a development time of one year by a team of four developers is proposed. This based on our own experience. In order to pay the wages of these developers, as well as contractors internal or external to Lorenz, a development cost of 200.000 DKK per month is proposed or a total of 2.400.000 DKK. We also expect to pay an external contractor for the development of the unified hardware solution. Based on our communication with Christiansen and their experiences with developing the Lorenz AI-link, we estimate a cost of 1.000.000 DKK.

In order to calculate how many units must be sold to recoup the development cost, an estimate of the production cost is done. The hardware cost is estimated at 35.000 DKK, see Table 16. Added to this is the price Micro Technic or another third party hardware developer would charge for a unified solution. A maximal price of 15.000 DKK per unit is estimated. The total per module comes to 50.000 DKK per unit.

Break-even Point: If the solution is sold at a price of 100.000 DKK, that would lead to a profit of 50.000 DKK per unit and a total of 68 units would need to be sold to recoup the cost. Added to this would be the cost of software maintenance. At the least, one engineer working one day a week would be needed to make fixes and changes as different situations occur. The cash flow during the first 8 years of the solutions life are shown in table 48. The total loss/gain over time is shown in Figure 49.

Component	**Cost [DKK]**
Hokuyo URG-04LX-UG01 (lidar)	6.050
Raspberry Pi Camera Board v2.1 (camera)	300
AdaFruit Ultimate GPS (GPS)	300
Raspberry Pi 4 (CPU)	900
USB Accelerator (TPU)	530
Covering box	300

Table 16: Price of hardware components.

Operating budget									
Time (yearly)		1	2	3	4	5	6	7	8
	Clarification								
Software development	avg. DKK 45000/monthly salary	-DKK 3,160,000.00							
Product costs	DKK 50,000.00 / unit		-DKK 200,000.00	-DKK 400,000.00	-DKK 800,000.00	-DKK 1,200,000.00	-DKK 1,600,000.00	-DKK 2,000,000.00	-DKK 2,600,000.00
Maintainence costs	avg. DKK 280.00 / hourly salary	-DKK 116,480.00	-DKK 116,480.00	-DKK 116,480.00	-DKK 116,480.00	-DKK 116,480.00	-DKK 116,480.00	-DKK 116,480.00	-DKK 116,480.00
Sales	DKK 100,000.00 / unit		DKK 400,000.00	DKK 800,000.00	DKK 1,600,000.00	DKK 2,400,000.00	DKK 3,200,000.00	DKK 4,000,000.00	DKK 5,200,000.00
Net Cashflow	---	-DKK 3,276,480.00	DKK 83,520.00	DKK 283,520.00	DKK 683,520.00	DKK 1,083,520.00	DKK 1,483,520.00	DKK 1,883,520.00	DKK 2,483,520.00
Cumulative Cashflow	---	-DKK 3,276,480.00	-DKK 3,192,960.00	-DKK 2,909,440.00	-DKK 2,225,920.00	-DKK 1,142,400.00	DKK 341,120.00	DKK 2,224,640.00	DKK 4,708,160.00

Figure 48: The operating budget for the solutions first 8 years, including development.

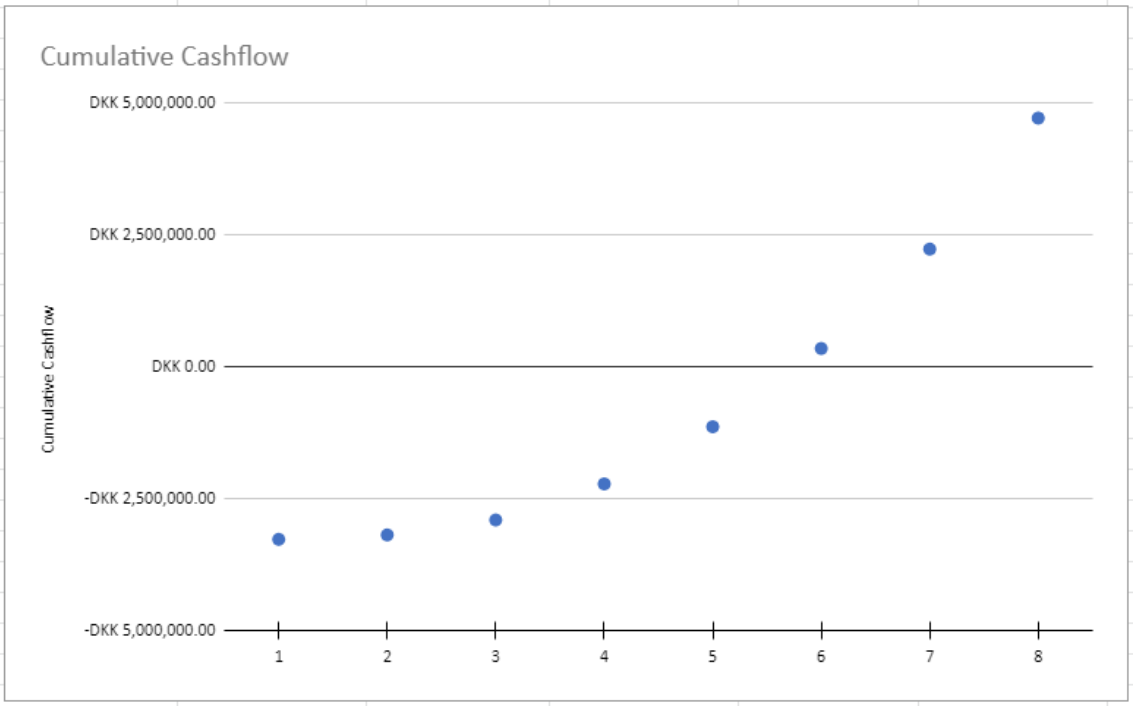


Figure 49: The cumulative cash flow for the solutions first 8 years, shown as a graph.