EE3L11: Bachelor Graduation Project

Occupancy grid mapping of ultrasound and LIDAR data for robotics

by

G.H.P. Dohmen & T. G. Vrijenhoek

to obtain the degree of Bachelor of Science
at the Delft University of Technology,

Student numbers:	5212782 & 5450780	
Project duration:	March 23, 2025 – June 27, 2025	
Thesis committee:	Dr. G. Joseph,	TU Delft, supervisor
	Prof. dr. ir. L. Abelmann,	TU Delft
	Dr. M. Alavi,	TU Delft

TU Delft

Abstract

This thesis presents the development of a sensor fusion framework that integrates ultrasonic sensors with a rotating Light Detection and Ranging (LiDAR) system to generate an occupancy grid map. The objective is to improve spatial awareness for autonomous navigation by employing an adaptive LiDAR approach, wherein ultrasonic sensors are used to identify regions of interest for focused scanning.

The design and implementation of a test system are described, along with the development of an occupancy grid map capable of representing data from both LiDAR and ultrasonic sensors. To enhance the accuracy and reliability of the environmental representation, the occupancy grid map incorporates an inverse sensor model in combination with Bayesian statistical methods.

Preface

This thesis is the result of over 2 months of work in the field of sensing and mapping. Together with 2 other group members we developed an innovative system for improving a robot's perception of the environment around itself by adapating the spin rate of the lidar and combining several data sources. We would like to express our gratitude to our supervisors Geethu Joseph, Nitin Meyers and Peiyuan Zhai for their guidance and support during this project. Furthermore, we would like to thank stichting Neobots for loaning several components necessary for our prototype. Lastly, we would like to thank our colleagues Evert-Jan Beiboer and Jesse van der Kooij for their collaboration in this project.

G.H.P. Dohmen & T. G. Vrijenhoek
Delft, June 2025

Contents

1 Introduction

In recent years, automation and autonomous functionality have become increasingly prevalent across various domains [1], [2], [3]. A prominent example is the modern automobile. Whilst fully autonomous vehicles are still under development, many consumer cars already incorporate semi-autonomous features such as lane-keeping assistance and emergency braking systems. These functions require continuous awareness of the environment, including the positions of obstacles and navigable free space. To achieve this, vehicles rely on an array of sensors [4], [5], [6]. One commonly used sensor type for detecting distances to nearby objects is the LiDAR sensor [6], [7], [8].

In robotic and automotive navigation, LiDAR sensors are often mounted on mechanical rotating platforms. As the platform rotates, the LiDAR emits laser pulses in specific directions. When these pulses encounter an object, they are reflected and detected by the LiDAR's receiver. By analyzing the time of flight of the reflected signals, the system calculates the distance to the object.

Conventional spinning LiDAR systems operate at a constant rotational speed, resulting in uniformly distributed measurements. While this ensures consistent spatial coverage and range, it can be inefficient. Specifically, the system allocates equal measurement density and power to both empty space and regions containing objects, leading to unnecessary use of the LiDAR in uninformative areas.

To address this inefficiency, research has explored adaptive spinning LiDAR systems. These systems dynamically adjust their rotational speed and power distribution across a single revolution, allowing them to concentrate measurement effort in regions of interest and reduce resource usage in open areas. As a result, such systems can achieve higher data quality. The difference in these two types of usage of LiDAR can be seen in figure 1.1.

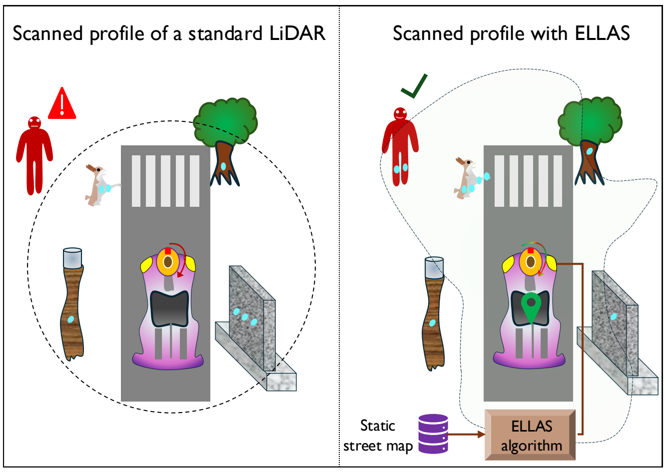


Figure 1.1: Figure from [9] comparing a conventional LiDAR system (left) with an adaptive LiDAR system (right).

A key challenge for adaptive LiDAR systems is determining where to allocate measurement resources. This requires prior information about the environment to identify regions of interest. Several methods for obtaining such information have been proposed, such as:

- Utilizing co-located cameras to estimate depth and detect object boundaries [10].
- Aggregating data over multiple rotations and using the previous rotation's measurements to guide the next [11], [12].
- Employing static topological maps to pre-allocate resources to known object locations, as demonstrated in [9], which served as an inspiration for this research.

In contrast to these approaches, this work explores the use of ultrasonic sensors to guide the adaptive LiDAR system. While ultrasonic sensors offer lower resolution compared to camera-based depth estimation or LiDAR-based feedback, they require significantly less computational power and provide useful information about free space due to the cone-shaped coverage of their emitted signals. This makes them particularly suitable for efficiently identifying traversable regions in the environment.

The goal of this project is to develop a system that integrates both ultrasonic sensors and a spinning LiDAR to collect environmental data. A software program is designed to control the ultrasonic sensors, and the resulting data is used to guide the behavior of an adaptive spinning LiDAR system, which will be implemented by the other subgroup we have worked with. An occupancy grid map is constructed to represent the environment, providing a basis for potential path planning. Data from both sensor types are fused into the map using an inverse sensor model and Bayesian statistics.

The remainder of this thesis is structured as follows: Chapter 2 outlines the system requirements, separated into hardware and software components. Chapter 3 describes the hardware design, including the rationale behind key design choices. Chapter 4 discusses the software implementation and sensor data processing. Chapter 5 presents experimental results and analyses, including parameter tuning and evaluation of measurements with both sensor types. Finally, Chapter 6 concludes the thesis and discusses possible future improvements.

2
Program of Requirements

As the project encompasses both software and hardware components, separate sets of requirements have been defined for each domain to ensure clarity and structure.

In addition to the core requirements, a number of desirable, yet non-essential, trade-off requirements have been formulated. These features are considered beneficial but are not critical to the core functionality of the system.

2.1. Hardware Requirements

The final hardware system must satisfy the following essential specifications:

- The localization system must provide a 360-degree field of view.
- The sensor must achieve a pointing accuracy of less than 1.5 degrees.
- The localization system must support a minimum operational range of 3 meters.

The system should ideally also meet the following trade-off specifications:

- The sensor array should complete data acquisition in under 5 seconds per sensor type.
- The sensor array should be compatible with the existing ELLAS robot frame.

2.2. Software Requirements

The software component must fulfill the following fundamental requirements:

- Sensor data acquired via the Arduino system must be transmitted to the server and represented in a 2D point cloud format.
- The software must generate an occupancy grid map based on the combined ultrasound and LIDAR datasets.
- The occupancy grid map must support a resolution finer than 5 centimeters per cell.

Additionally, the following trade-off requirements are considered desirable:

- A graphical user interface (GUI) should be available to visualize the system data.
- Sensor data should be processed within a maximum time frame of 2 seconds.

2.3. Project Objectives

The project objectives have been divided according to the two main components:

Hardware: Develop a functional prototype of an adaptive LIDAR system that integrates both ultrasound and LIDAR sensors within a single unit. The prototype should fully comply with the defined hardware requirements.

Software: Design and implement a software system capable of collecting data from the prototype's sensors and fusing this data into a coherent occupancy grid map.

In the next chapter, the design choices made for the hardware prototype are described.

3
Hardware Design

To facilitate testing of both the software developed by this subgroup (sensor fusion onto an occupancy grid map) and the other subgroup (adaptive spin rate of the LIDAR), a dedicated prototype was developed. An overview of the hardware design is presented in Figure 3.2.

3.1. Objective of the Hardware Design

The objective of the hardware design is to develop a functional prototype of an adaptive LIDAR system that integrates both ultrasonic and LIDAR measurements into a single package. This prototype must satisfy all the specifications outlined in the Program of Requirements.

3.2. The Starting Point: ELLAS

This project builds upon the foundation laid by the ELLAS system [9], a prototype that was available to the team at the beginning of the project. A photograph of the ELLAS prototype is shown in Figure 3.1. However, a decision was made to design a new hardware solution for the following reasons:

- **Incompatibility with project needs:** The ELLAS system was based on a single 1D LIDAR sensor and lacked the necessary space to integrate ultrasonic sensors without significant structural modifications. Additionally, the CAD models of the original system were not available, necessitating a full reverse-engineering effort. A complete redesign was deemed more efficient.
- **Limited availability:** Due to scheduling conflicts, the ELLAS system was unavailable for the final two weeks of the project. This limitation would have placed unnecessary pressure on the team and hindered final testing and tuning.

The original ELLAS system was mounted on a remote-controlled car using four standoffs, each 6 cm in height, to allow space for mounting electronics beneath the sensor and motor platform.

3.3. Our Design Approach

The following design constraints guided the development of the new prototype:

- **Compliance with the Program of Requirements**
- **Material availability:** Given the limited project duration (8 weeks) and a desire to minimize costs, off-the-shelf components and readily available materials were prioritized. This consideration significantly influenced the selection of sensors and the choice of manufacturing methods.
- **Modularity:** As this prototype serves as a test platform, modularity was essential. Components such as sensors should be easily swappable.

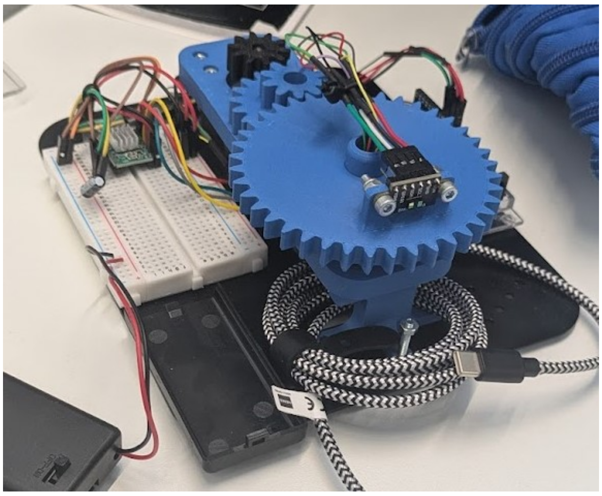


Figure 3.1: ELLAS prototype. The design lacks space for easy integration of additional sensors.

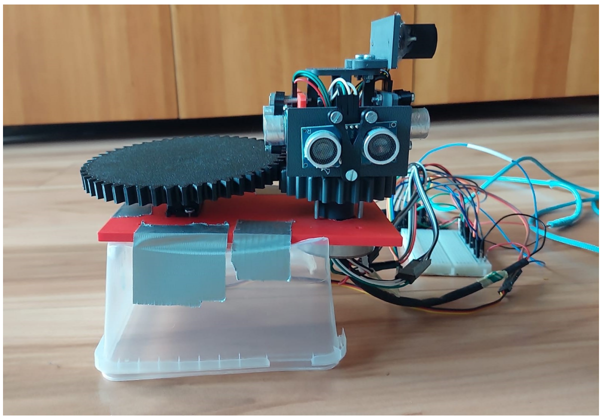


Figure 3.2: Overview of the test setup design

3.3.1. Sensor Selection

The first step in the design process involved selecting suitable sensors.

LIDAR

Two LIDAR options were considered: the laser rangefinder from the original ELLAS system and a 1D laser rangefinder owned by a team member. Both options met the requirements of a sufficient data acquisition speed and a minimum range of over 3 meters. The YDLIDAR SDM15 was ultimately selected.

Ultrasonic Sensor

For ultrasonic sensing, two primary categories were considered: separate transmitter-receiver modules or fully integrated sensor units. Due to availability and budget constraints, a set of hobby-grade sensors was considered:

- **HC-SR04**: A generic ultrasonic sensor commonly found in hobby kits. Readily available at the Tellegen Hall and among team members.
- **HC-SR04T**: A waterproof version of the HC-SR04.
- **SRF05**: An improved version of the HC-SR04 with a more optimized pin layout.

Each sensor had a maximum range of 4 meters, exceeding the minimum requirement of 3 meters, as well as a maximum data acquisition rate of 20 Hz and an angular resolution of 15°. The specifications are summarized in Table 3.1.

	SR-04	SR-04T	SRF-05
Range	4 m	4 m	4 m
Data acquisition rate	20 Hz	20 Hz	20 Hz
Angular resolution	15°	15°	15°
Availability	Readily available	To be ordered	To be ordered

Table 3.1: Comparison of ultrasonic sensor specifications

All candidates met the system's minimum requirements, and the HC-SR04 was selected due to its immediate availability.

3.3.2. Ultrasound Timing Analysis

Each HC-SR04 sensor covers a 15° cone. To cover a full 360°, a total of 24 sensors (or scanning positions) are required. Each sector measurement consists of up to five averaged measurements. With a measurement and travel time of 0.1 s each, a single sector requires $5 \cdot 0.1 + 0.1 = 0.6$ s. Thus, a full scan using a single sensor would take $24 \cdot 0.6 = 14.4$ s.

To achieve a sub-5-second acquisition time, four sensors were used in parallel, reducing the scan duration to $\frac{14.4}{4} = 3.6$ s.

3.3.3. Final Design

The hardware design is composed of three main parts: the sensor mount, the motion system, and the base plate. All components were designed using 3D CAD software and fabricated using 3D printing.

Sensors were connected to an Arduino UNO R3 via its GPIO pins. Since the Arduino's 5V output is limited to 200 mA, an external power bank was used to power the sensors and the servo motor (with a stall current of 1 A).

Technical drawings of the final prototype are included in Appendix A.

Motion System

The motion system employs a Hitec HS-485HB servo motor. A servo was selected over alternatives such as stepper motors due to its simplicity, team familiarity, and ease of integration.

Despite the general limitations of hobby-grade servos, such as low torque and accuracy, the HS-485HB offers an angular accuracy of 0.3° and a gear backlash of up to 0.5°, resulting in a total pointing error of less than 0.8°.

The servo's native range is ±90°, but to achieve ±180°, a 2:1 gear ratio was implemented. As a result, each degree of servo rotation produces 2° of rotation on the sensor mount.

Sensor Mount

The sensor mount accommodates:

- **Four ultrasonic sensors**, to meet the acquisition time requirement while averaging three measurements per sector.
- **One LIDAR sensor**.

To enhance modularity, the central mounting hub (which doubles as the gear) does not contain direct mounts for the sensors. Instead, sensor-specific mounting plates were attached to the gear using two M3 bolts each.

Due to spatial constraints, two ultrasonic sensors were offset by an extra 30 mm from the center of rotation. These offsets were accounted for in the Arduino's data processing logic.

The sensor mount is shown in Figure 3.3.

Base Plate

The base plate integrates the sensor mount and motion system. The bearing for the sensor mount is installed on the underside of the plate.

A key design constraint was compatibility with the ELLAS system's hole pattern, enabling the new prototype to be mounted on the same remote-controlled platform. A rendering is shown in Figure 3.4.

The next chapter details the software that is used with this hardware prototype.

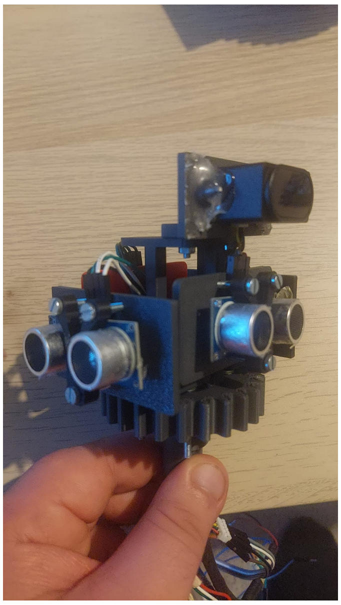


Figure 3.3: Sensor mount with ultrasonic and LIDAR sensors

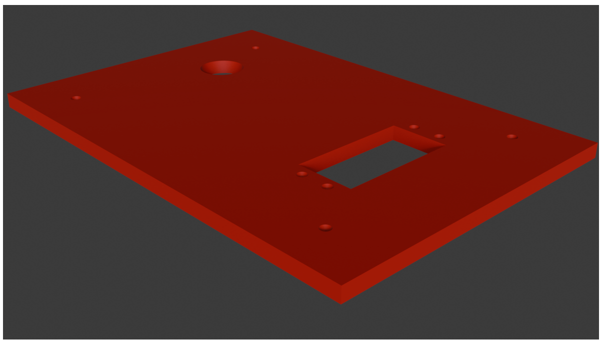


Figure 3.4: Rendering of the base plate

4

Software Design

This chapter describes the design choices made in the software component of the project. Section 4.1 outlines the overall structure and communication protocol. Section 4.2 explains the concept of occupancy grid maps and their implementation. Section 4.3 introduces the inverse sensor model and its application. Section 4.4 discusses the use of Bayesian statistics for updating probabilities. Lastly, Section 4.5 describes optimizations that improved processing speed.

4.1. General Project Setup

The system is divided across two devices: an Arduino, which is part of the hardware setup described in chapter 3, and a computer running a Python program (referred to as the server). The Arduino handles sensor data collection, while the server performs computation-heavy tasks such as grid mapping and sensor modeling.

4.1.1. Why Use Two Devices?

Integrating all functionality on the Arduino was not feasible due to its limited RAM and processing power [13]. Using a more powerful embedded system was also impractical due to accessibility and flexibility limitations. An efficient and simple solution was to use two devices.

4.1.2. Communication Between Devices

Communication occurs via a USB serial link, which constrains movement during testing. Data is exchanged in CSV format, with messages initiated by the server and responded to by the Arduino. Each message begins with a character identifier indicating the message type, as shown in Table 4.1.

Letter	Sender	Description
u	Arduino	Ultrasound data at 15°intervals
l	Arduino	LIDAR data at 2°intervals
a	Arduino	Angles of adaptive LIDAR measurements
b	Arduino	Values of adaptive LIDAR measurements
l	Server	Request basic LIDAR measurement
a	Server	Request adaptive LIDAR (ultrasound complement)
c	Server	Request adaptive LIDAR (automotive system)
u	Server	Request basic ultrasound measurement

Table 4.1: Overview of communication commands

4.1.3. GUI

A graphical user interface (GUI) was developed using the Python NiceGUI library. This web-based GUI allows users to test subsystems, initialize communication, and visualize the occupancy grid map. An example is shown in Figure 4.1.

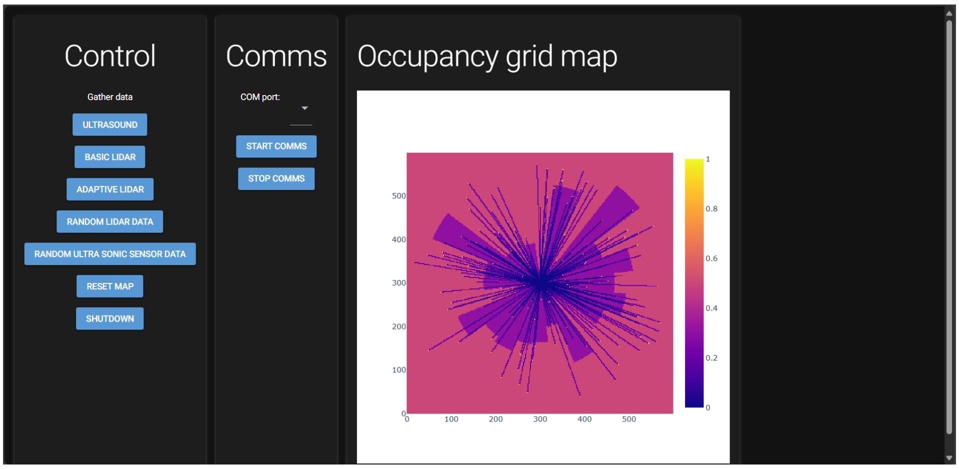


Figure 4.1: GUI with randomly generated LIDAR and ultrasonic measurements

4.2. Occupancy grid maps

An occupancy grid map is, in its simplest form, a 2D grid of the environment around the robot. Occupancy grid maps are widely used in several applications, such as in path planning [14] and object avoidance [15]

An $n \times m$ grid is declared in world space. Formally, each cell C has a discrete random state variable $s(C)$ with two possible states: A cell is either occupied (OCC) or empty (EMP). Since both states are exclusive, for each cell the following relation holds: $P[s(C) = OCC] + P[s(C) = EMP] = 1$ [16].
For our implementation, the value of each cell is defined as $P[s(C) = OCC]$ (shortened to P_{OCC}), the probability that the cell is occupied. Since in the initial situation nothing is known, for each cell C in our occupancy grid map, the following initial condition is used: $P_{OCC} = 0.5$. This means we are 50% sure that the cell is occupied (and thus 50% sure that it is empty). An example of an occupancy grid map can be found in figure 4.2.

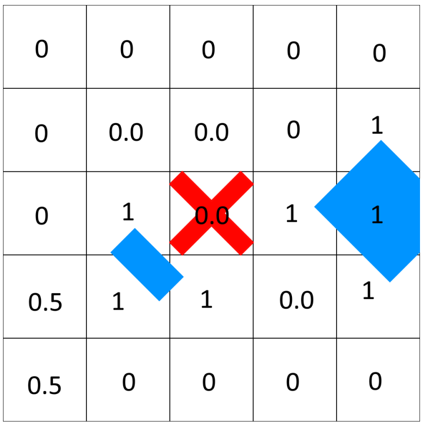


Figure 4.2: An example of an occupancy grid map, making use of infinitesimally small rays cast from the center of the grid (the red cross). A value of 0 indicates a cell that was detected as empty by the ray, and a value of 1 indicates an occupied cell.

4.2.1. Implementation

The occupancy grid map is implemented in code as a 2D numpy array, with the value of each cell being a float. Seeing as our requirement specified a measurement range of 3 meters, with a center placement of the robot, a grid of 6 by 6 meters was generated. The array size depends on the constant gridResolution from constants.py in appendix B.2. For example, if this constant is 50, then there are 50 cells per meter, and the array would be 300 by 300 cells. The measurements are scaled with this resolution to keep the results still in the right cell in the array. Before the grid map is plotted, it is upscaled depending on the grid resolution to ensure a map size of 6 x 6 meters.

4.3. Inverse sensor model

The inverse sensor model estimates cell occupancies based on sensor readings. The inverse part of the inverse sensor model reverses the forward model, which, given the state of the environment, predicts what the sensor readings will be. Thus, the inverse sensor model derives the environment based on the sensor data [17], [18], [19].

For a certain distance measurement z at angle θ of a sensor, the inverse sensor model returns the probability of occupancy $P_{OCC,new}$ for each cell covered by the measurement. For a LiDAR, this will be all cells on a line along θ. For an ultrasonic sensor, this will be all cells in a cone centered around θ, with an angular width equal to the Field of View of the sensor. The selected cells are categorized into three distinct groups by the inverse sensor model.

- **Occupied:** The cell is close to the measurement, so $P_{OCC,new}$ will be high.
- **Free:** The cell is before the measurement, so $P_{OCC,new}$ will be low.
- **Unknown:** The cell is after the measurement, so $P_{OCC,new}$ will be 0.5.

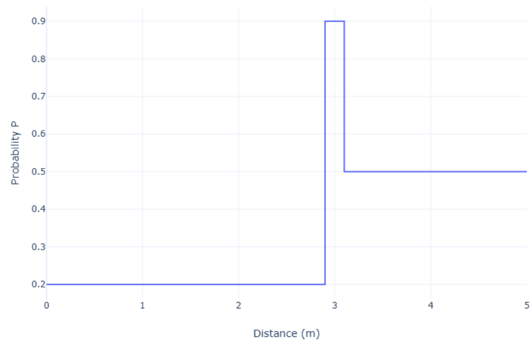


Figure 4.3: Example of ideal inverse sensor model with object detection at 3 meters

The categorization is dependent on the distance d of the cell from the origin and the standard deviation σ_{sensor} of the sensor. If $d < z - \sigma_{sensor}$, the cell is passed through by the sensor and thus can be assumed to be free. In the case that $z - \sigma_{sensor} < d \leq z + \sigma_{sensor}$, the cell is at the measured value and thus is assumed to be occupied. Lastly, in the case that $d > z + \sigma_{sensor}$, the cell is beyond the measurement, thus there is no information about this cell. Therefore, the cell is assumed to be unknown.

4.3.1. Implementation

The model is applied in three steps:

1. Identify all cells traversed by the sensor signal.
2. Classify them into the three defined categories: occupied, free, or unknown.
3. Assign probabilities based on sensor type and confidence.

LiDAR

To select the cells on which the inverse sensor model needs to be used, a ray is cast along the direction of the LiDAR sensor. This is done in the gridmap.py code in appendix B.6, the function can also be seen in listing 4.1. This function adds the coordinates of cells along an imaginary line from the center to the edge of the grid map on a given angle from the front of the localization system.

```python
def raycast(self, _angle:float):
    """Casts a ray over the gridmap on a specific angle.

    Args:
        _angle (float): angle in degrees

    Returns:
        tuple[]: array of coords of the cells crossed by the ray
    """
    #assuming center starting pos, let's get the start and end points of our line
    _lineStartPos = [int(self.sizeX/2),int(self.sizeY/2)]
    _lineEndPos = self.calcLineEndPos(_angle)

    #calculate delta's and set our initial x and y coords
    _dx = int(_lineEndPos[0]-_lineStartPos[0])
    _dy = int(_lineEndPos[1]-_lineStartPos[1])
    _x = int(_lineStartPos[0])
    _y = int(_lineStartPos[1])
    #amount of cells to visit (EG: line crossings)
    _n = np.sum([1,np.abs(_dx),np.abs(_dy)])
    #what to increment by. Either positive or negative, dependent on the delta;.
    _xInc = 0
    _yInc = 0
    if(_dx > 0):
        _xInc = 1
    elif (_dx < 0):
        _xInc = -1
    if(_dy > 0):
        _yInc = 1
    elif (_dy < 0):
        _yInc = -1

    _error = np.abs(_dx) - np.abs(_dy)
    _output = []

    while (_n>0):
        if(_x >= self.sizeX):
            #reaching out of bounds, so we should stop.
            break
        if(_y >= self.sizeY):
            #reaching out of bounds, so we should stop.
            break
        if ((_x == _lineStartPos[0]) and (_y == _lineStartPos[1])):
            #equals starting pos, so skip this.
            pass
        else:
            _output.append([_x,_y])

        if (_error > 0):
            _x += _xInc
            _error -= abs(_dy)
        else:
            _y += _yInc
            _error += abs(_dx)
        _n -= 1
    return _output
```

Listing 4.1: Python function to select all cells hit by a raycast

After the raycast, the inverse sensor model is used to determine a new probability for the selected cells of the occupancy grid map, as seen in listing 4.2 from sensorModel.py in appendix B.7. In this function, the difference between the distance from the cell to the origin and the measured distance by the LiDAR is taken. If this difference is smaller than or equal to the standard deviation of the LiDAR, this cell is likely occupied, and a high probability is returned. If the previous argument is not true and the distance from the cell to the origin is smaller than the measured distance, this cell is probably free, and a low probability is returned. In the other cases, an unknown is returned.

```python
def probabilityBasedOnMeasurement(self, _measurement,_distanceFromOrigin):
    delta = abs(_distanceFromOrigin - _measurement)
```

```python
    if delta <= self.stDevlidar:
        return 0.90  # likely obstacle
    elif _distanceFromOrigin < _measurement:
        return 0.20  # likely free
    else:
        return 0.50  # unknown
```

Listing 4.2: Python implementing the inverse sensor model for the LiDAR

Because of the small standard deviation, as seen in appendix B.2, and the maximum relative error of 4%, the likelihood that there is an object at the measured distance is high. To reflect this in the inverse sensor model, the $P_{OCC,new}$ of the cells classified as occupied is set to 0.9. Likewise, the $P_{OCC,new}$ for free cells should be quite low and is therefore 0.2.

Ultrasonic sensor

In case we do a raycast for the ultrasonic sensor, you would get all the cells in a cone with the angle of the angular resolution of the sensor from the center until the edge of the grid map. However, for each section, with the size of the angular resolution of the sensor, measurements are made. Therefore, every cell would be selected. Thus, instead of first selecting cells, the inverse model is done for every cell in the occupancy grid map. This can be seen in listing 4.3, which comes from sensorModel.py in appendix B.7. This function works the same as in listing 4.2, but with other return values.

```python
def probabilityBasedOnMeasurementultra(self, _measurement,_distanceFromOrigin):
    delta = abs(_distanceFromOrigin - _measurement)
    if delta < self.stDevultra:
        return 0.50  # likely obstacle
    elif _distanceFromOrigin < _measurement:
        return 0.30  # likely free
    else:
        return 0.50  # unknown
```

Listing 4.3: Python implementing the inverse sensor model for the ultrasonic sensor

Due to the higher standard deviation for the ultrasonic sensor, as seen in appendix B.2, and no way of knowing where on the width of the cone a detection happened, the cells within the range of the measured distance also get 0.5 returned by this inverse sensor model.

4.4. Bayesian statistics

The probability returned by the inverse sensor model can not be directly used in the occupancy grid map. This is because the sensors used are not noise-free, and the inverse sensor model does not take into account the existing probabilities as is done in [18], [20]. For this, Bayes' theorem is used:

$$P(m_{xy}|z_1, ..., z_t) \tag{4.1}$$

For which $z_1, ..., z_t$ denotes all the sensors' measurements from time 1 until time t, and m_{xy} denotes the probability of the cell being occupied. For the next part, the functions as seen in [16] are used. For computational efficiency, the log-odds representation is used:

$$l_{xy}^t = log\frac{P(m_{xy}|z_1, ..., z_t)}{1 - P(m_{xy}|z_1, ..., z_t)} \tag{4.2}$$

From the log-odds representation l_{xy}^T in function 4.2 we can get the probability from function 4.1 with:

$$P(m_{xy}|z_1, ..., z_t) = 1 - (1 + e^{l_{xy}^t})^{-1} \tag{4.3}$$

Using Bayes' rule on the last z_t in equation 4.1 we get:

$$P(m_{xy}|z_1, ..., z_t) = \frac{P(z_t|z_1, ..., z_{t-1}, m_{xy})P(m_{xy}|z_1, ..., z_{t-1})}{P(z_t|z_1, ..., z_{t-1})} \tag{4.4}$$

In the static world assumption, given the knowledge of the cell m_{xy}, the past sensor readings are independent for any point in time:

$$P(z_t|z_1,...,z_{t-1},m_{xy}) = P(z_t|m_{xy}) \tag{4.5}$$

Using equation 4.5 to simplify equation 4.4 and then using Bayes' rule on $P(z_t|m_{xy})$ gives us function 4.6 and 4.7.

$$P(m_{xy}|z_1,...,z_t) = \frac{P(z_t|m_{xy})P(m_{xy}|z_1,...,z_{t-1})}{P(z_t|z_1,...,z_{t-1})} \tag{4.6}$$

$$P(m_{xy}|z_1,...,z_t) = \frac{P(m_{xy}|z_t)P(z_t)P(m_{xy}|z_1,...,z_{t-1})}{P(m_{xy})P(z_t|z_1,...,z_{t-1})} \tag{4.7}$$

The same process can be done for the probability of a cell being free instead of being occupied. If we take $\bar{m}_{xy}$ as the probability of a cell being empty, we get:

$$P(\bar{m}_{xy}|z_1,...,z_t) = \frac{P(\bar{m}_{xy}|z_t)P(z_t)P(\bar{m}_{xy}|z_1,...,z_{t-1})}{P(\bar{m}_{xy})P(z_t|z_1,...,z_{t-1})} \tag{4.8}$$

dividing equation 4.7 by 4.8:

$$\frac{P(m_{xy}|z_1,...,z_t)}{P(\bar{m}_{xy}|z_1,...,z_t)} = \frac{P(m_{xy}|z_t)P(\bar{m}_{xy})P(m_{xy}|z_1,...,z_{t-1})}{P(\bar{m}_{xy}|z_t)P(m_{xy})P(\bar{m}_{xy}|z_1,...,z_{t-1})} \tag{4.9}$$

Rewriting equation 4.9 with $P(\bar{m}_{xy}) = 1 - P(m_{xy})$ and the same for any conditioning variable given, gives:

$$\frac{P(m_{xy}|z_1,...,z_t)}{P(1-P(m_{xy}|z_1,...,z_t)} = \frac{P(m_{xy}|z_t)}{1-P(m_{xy}|z_t)}\frac{1-P(m_{xy})}{P(m_{xy})}\frac{P(m_{xy}|z_1,...,z_{t-1})}{1-P(m_{xy}|z_1,...,z_{t-1})} \tag{4.10}$$

From this, we can write the desired log-odds equation as:

$$log\frac{P(m_{xy}|z_1,...,z_t)}{P(1-P(m_{xy}|z_1,...,z_t)} = log\frac{P(m_{xy}|z_t)}{1-P(m_{xy}|z_t)} + log\frac{1-P(m_{xy})}{P(m_{xy})} + log\frac{P(m_{xy}|z_1,...,z_{t-1})}{1-P(m_{xy}|z_1,...,z_{t-1})} \tag{4.11}$$

Finally we can substitute equation 4.2 into equation 4.11 to get:

$$l_{xy}^t = log\frac{P(m_{xy}|z_t)}{1-P(m_{xy}|z_t)} + log\frac{1-P(m_{xy})}{P(m_{xy})} + l_{xy}^{t-1} \tag{4.12}$$

Function 4.12 tells us that we can get l_{xy}^t, from the log-odds of the new measurement, adding the log-odds of the initial freeness of the cell and the log-odds previous value of the cell.

4.4.1. Implementation

In listing 4.4 below, a snippet of sensorModel.py from appendix B.7 is shown. In this code, the theory of section 4.4 is implemented in Python. First, the inverse sensor model probability, in the code _modelProbability, and the current probability of the cell in the grid map are truncated. This makes sure that the current probability is within the range of the natural logarithm and that no division by 0 occurs. After this, the log-odds of the current probability is calculated, and the new value gets updated by adding the log-odds of the model probability. The log-odds of the initial freeness as seen in function 4.12 is not added, seeing as the world is initially completely unknown, which gives a log-odds of 0 and therefore has no influence. Lastly, the new log-odds of the cell gets reverted into a probability, which can be set as the cell's value in the grid map.

```
eps = 1e-6
_modelProbability = max(eps, min(1 - eps, _modelProbability))
_currProbability = max(eps, min(1 - eps, _currProbability))


#converting old probability to log-odds
_logProb = np.log(_currProbability / (1 - _currProbability))


```

```
#calculate new probability
_logProb = _logProb + np.log(_modelProbability / (1 - _modelProbability))

#convert back to normal probability
_newProb = round(1 - 1/(1+math.exp(_logProb)),6)
```

Listing 4.4: Python code snippet where Bayesian statistics get used on the cell probabilities

4.5. Computational speed improvements

Each individual calculation that needs to be done for a new probability is, in itself, not computationally intensive. However, calculating all values for an entire grid of $n * n$ cells quickly adds up to a high execution time. To meet the requirement of a sub-2-second calculation time, several improvements had to be made. First of all, for big lists and arrays, only numpy arrays are used because of the higher operation speed compared to traditional Python lists.

Because of the low number of discrete outcomes of the inverse sensor model, there will also be a low number of discrete probabilities that get calculated. In listing 4.5, a dictionary is used to check if the combination of current and model probability has been calculated before, and if true, the associated value is returned. Otherwise, the calculations are finished, and the combination is added to the dictionary.

```
_dict_key = (_modelProbability,_currProbability)
if (_dict_key in _inverse_dict):
    return _inverse_dict[_dict_key]
```

Listing 4.5: Python code snippet where a dictionary is used to skip the otherwise needed calculations

As mentioned before, if a probability of 0.5 is used for log-odds, it will return a 0. So, for the cells that the inverse sensor model returns that lie in the region that gets classified as unknown, the functions will return the current probability of the cell. If the cell gets checked before the function calls for if it lies in the region beyond the measurement, the functions can be skipped and the value of the cell stays the same. Lastly, a major improvement in execution time is accomplished by calculating the distance from the cell to the origin, as it gets used in listing 4.2 and 4.3, for every cell beforehand. By creating a numpy array with the distance from every cell to the origin as the value of the cells at the start of the program, a faster execution time can be accomplished with a trade-off for a higher startup time.

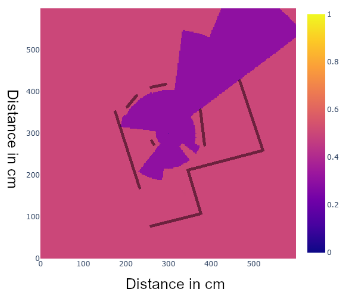
5 Results

This chapter presents the results of several tests conducted to evaluate the system's performance. All figures shown here are also included in Appendix D, where they are displayed at a larger size for improved readability.

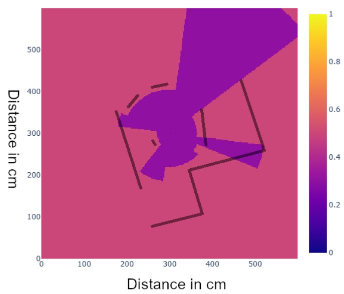
The first test assesses the ultrasonic sensor's capability to detect objects in a controlled environment. The goal was to determine whether the ultrasonic system can generate meaningful regions of interest for the adaptive LiDAR model. Subsequently to each ultrasonic scan, a scan using a non-adaptive LiDAR was performed. This scan was used to generate reference lines (in dark red/black) in the maps that illustrate the ultrasonic sensor data.

In the first set of results (Figure 5.1), the ultrasonic sensors collected one measurement per angular sector. In the second set (Figure 5.2), three measurements were collected per sector, and the median was used as the representative value. In the third set (Figure 5.3), five measurements were taken per sector, with the median again used.

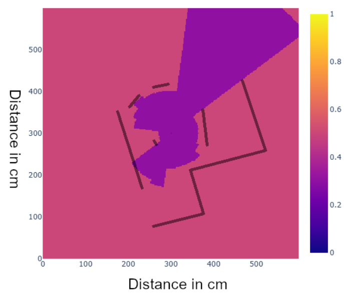
Across all figures, it is evident that the detected object position is slightly in front of the actual object location. This phenomenon is likely due to the sensor's detection region being wider than the object itself, as discussed in Section 4.3, because of the higher standard deviation of the sensor. Additionally, the ultrasonic sensor consistently struggles to accurately detect objects in the lower corner of the environment. This is likely the result of suboptimal object placement relative to the sensor, causing the ultrasonic pulse to reflect off other parts and get into that corner. Furthermore, the gap between the two top objects is not detected in any figure, likely because the objects are too close together to be distinguished by the ultrasonic sensor at the given distance.



(a)



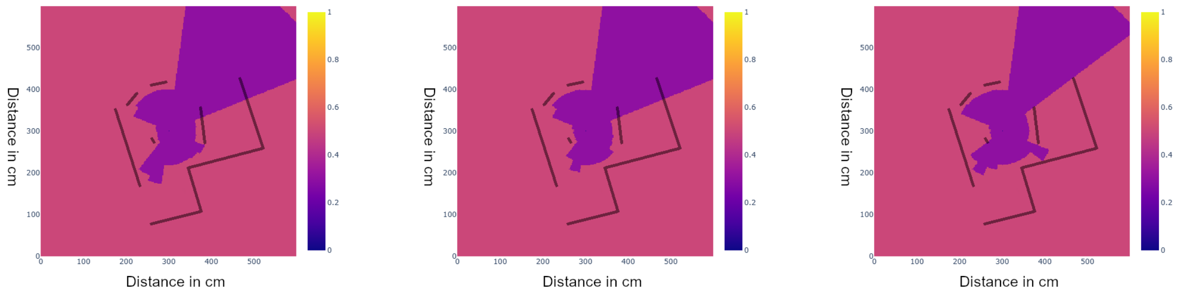
(b)



(c)

Figure 5.1: Ultrasonic sensor test with one measurement per angular sector, including a schematic of the object placement

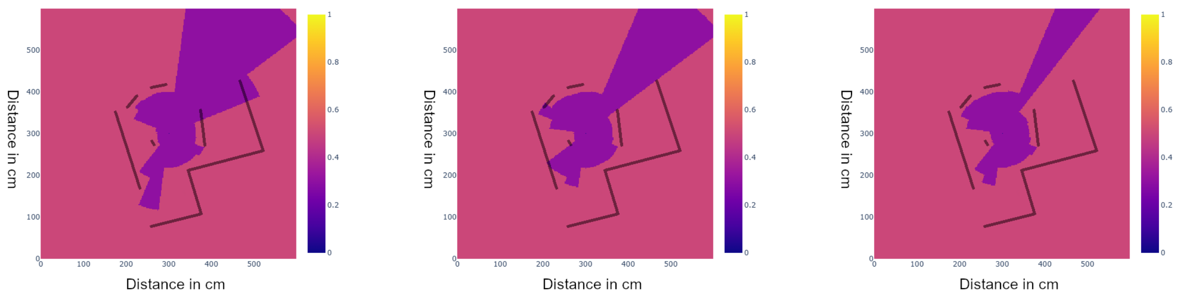
In Figure 5.1, most ultrasonic measurements do not intersect with objects, except for a single outlier in Figure 5.1b. There is noticeable variation across the three subfigures, indicating sensitivity to noise when only one measurement per sector is used. The sensor can also mistakenly detect nearby objects slightly outside its measurement cone if they are closer than objects inside the cone. This effect becomes more pronounced when fewer measurements are taken.



(a) (b) (c)

Figure 5.2: Ultrasonic sensor test using the median of three measurements per angular sector

The results in Figure 5.2 are comparable to those in Figure 5.1, but show improved consistency and uniformity in object detection across the three trials.

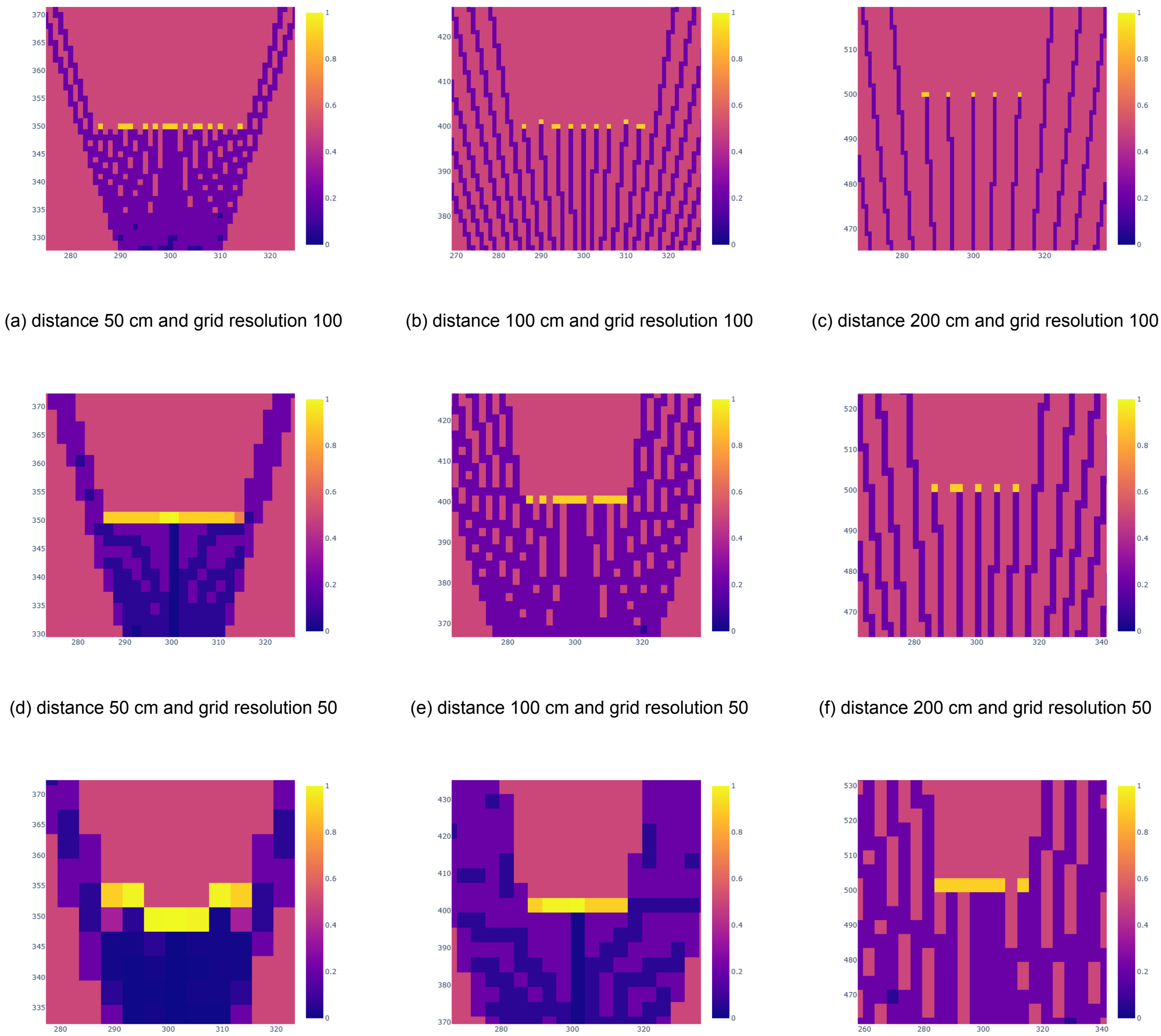


(a) (b) (c)

Figure 5.3: Ultrasonic sensor test using the median of five measurements per angular sector

Figure 5.3 shows slightly more variation between the three tests compared to Figure 5.2. This suggests that there may be an optimal number of measurements beyond which the accuracy does not improve, and might even degrade. Taking too few measurements increases the likelihood of detecting irrelevant objects outside the measuring cone. Taking too many may dilute the accuracy of the closest measurement. Based on this evaluation, using the median of three measurements per sector was selected as the final configuration for the ultrasonic system.

The second test focused on determining the optimal resolution for the occupancy grid map. An object with a width of 30 cm was placed at distances of 50 cm, 100 cm, and 200 cm from the sensor. Scans were performed using a standard LiDAR at a 2° interval over a 40° field of view. The grid map was tested at three resolutions: 100, 50, and 25 cells per meter (corresponding to cell sizes of 1x1 cm, 2x2 cm, and 4x4 cm, respectively). The results are presented in Figure 5.4.

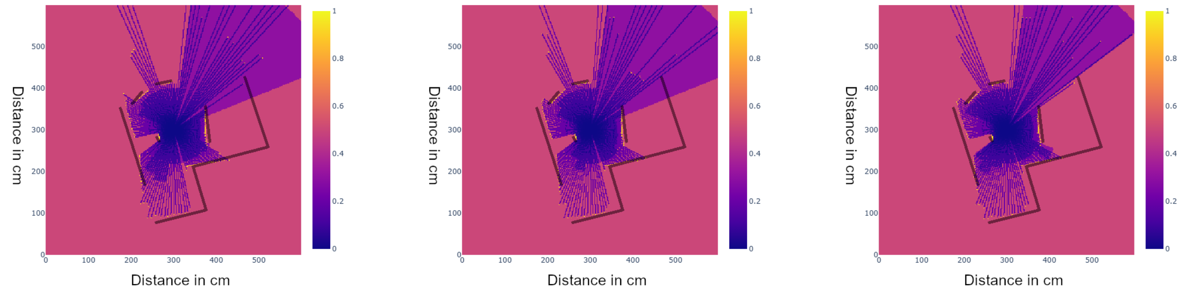


(a) distance 50 cm and grid resolution 100
(b) distance 100 cm and grid resolution 100
(c) distance 200 cm and grid resolution 100
(d) distance 50 cm and grid resolution 50
(e) distance 100 cm and grid resolution 50
(f) distance 200 cm and grid resolution 50
(g) distance 50 cm and grid resolution 25
(h) distance 100 cm and grid resolution 25
(i) distance 200 cm and grid resolution 25

Figure 5.4: Results of testing different grid map resolutions for an object with a width of 30 cm at different distances

From the 100 cells/m resolution results, small gaps begin to appear between measurement points, which increase with distance. At 50 cm, some gaps of approximately 1 cm are visible; at 100 cm, 2 cm gaps; and at 200 cm, 5 cm gaps. Reducing the resolution to 50 cells/m eliminates or reduces these gaps. At 25 cells/m, most gaps disappear entirely, but distortion becomes apparent. For example, Figure 5.4g shows an otherwise straight object appearing slightly curved due to the coarse resolution.

Although a 5 cm gap at 200 cm appears in Figure 5.4c, such a small gap is functionally irrelevant for typical applications, especially since the robot or vehicle is physically larger. Therefore, a lower resolution does not hurt the functionality of the system. However, too low a resolution leads to the aforementioned warping effect. Considering all factors, our final measurements utilized a resolution of 50 cells per meter.



(a) (b) (c)

Figure 5.5: Occupancy grid map updates using both ultrasonic sensor and LiDAR data

Finally, a test was conducted to evaluate the combined use of both ultrasonic and LiDAR data in updating the occupancy grid map. After each ultrasonic scan from the first test, a standard LiDAR scan was performed. The results are shown in Figure 5.5, using the configuration as the ultrasonic sensors tests in Figure 5.2. As expected, the LiDAR confirmed object detections already identified by the ultrasonic sensor and additionally captured areas that the ultrasonic system failed to detect.

Examining the occupancy grid map, it is evident that regions traversed by multiple sensor signals tend to have values that converge toward zero, indicating high confidence in free space. This is particularly noticeable around the center, where the density of intersecting sensor rays is highest. In areas where only a few rays pass through, the cell values decrease but remain above zero. Lastly, where LiDAR detects object hits, the corresponding cell values increase, representing probable obstacles.

6 Conclusion and Discussion

In this thesis, a framework was developed that enables the use of an adaptive LiDAR system, which adjusts its behavior based on data from ultrasonic sensors. Additionally, an occupancy grid map was implemented to process and visualize sensor data from multiple sources. The resulting environmental map serves as a foundation for autonomous navigation, either for automotive control or routing in environments where GPS is unavailable.

Looking back at the program of requirements, all requirements set at the beginning of the project were met. Furthermore, the trade-off requirements were also met by the final product.

6.1. Further Work & Improvements

Several opportunities exist for further development and enhancement of the system:

- **Sensor Hardware:** One of the main limitations in measurement accuracy stems from the sensors used. For this proof of concept, inexpensive and readily available components were utilized. Replacing these with higher-quality sensors and using a more precise servo motor could significantly improve both data quality and angular resolution of the rotating LiDAR system.
- **Computation Speed:** The number of usable data points per LiDAR rotation is currently constrained by the speed of occupancy grid map generation. While performance improvements were discussed in Section 4.5, further gains might be achieved through GPU-based grid mapping methods such as those described in [21], or by adopting raycasting-free techniques like in [22].
- **Dynamic Environments:** The current implementation assumes a static environment. For deployment in real-world scenarios, the occupancy grid map must account for dynamic changes. This can be achieved using aging or decay techniques, as explored in [20], which allow for gradual fading of outdated occupancy information.
- **Device Movement:** At present, the system does not account for the movement of the sensing device. To enable this, the position from which measurements are taken must be adjustable, allowing the device to move within the mapped region. Another approach to handle movement is to update the grid map dynamically: adding new unknown cells in the direction of movement, shifting existing data accordingly, and removing outdated information that moves outside the map boundary.

Bibliography

[1] Q. Zou, Q. Sun, L. Chen, B. Nie, and Q. Li, “A comparative analysis of lidar slam-based indoor navigation for autonomous vehicles,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 6907–6921, 2022.

[2] Y. Li and J. Ibanez-Guzman, “Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems,” *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, 2020.

[3] J. Panchal and Z. Wang, “Design of next generation automotive systems: Challenges and research opportunities,” *Journal of Computing and Information Science in Engineering*, vol. 23, pp. 1–9, 07 2023.

[4] A. Carullo and M. Parvis, “An ultrasonic sensor for distance measurement in automotive applications,” *IEEE Sensors Journal*, vol. 1, no. 2, pp. 143–, 2001.

[5] S. Campbell, N. O’Mahony, L. Krpalcova, D. Riordan, J. Walsh, A. Murphy, and C. Ryan, “Sensor technology in autonomous vehicles : A review,” in *2018 29th Irish Signals and Systems Conference (ISSC)*, pp. 1–4, 2018.

[6] A. Pandharipande, C.-H. Cheng, J. Dauwels, S. Z. Gurbuz, J. Ibanez-Guzman, G. Li, A. Piazzoni, P. Wang, and A. Santra, “Sensing and machine learning for automotive perception: A review,” *IEEE Sensors Journal*, vol. 23, no. 11, pp. 11097–11115, 2023.

[7] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang, “Fast-lio2: Fast direct lidar-inertial odometry,” 07 2021.

[8] J. Schulte-Tigges, M. Förster, G. Nikolovski, M. Reke, A. Ferrein, D. Kaszner, D. Matheis, and T. Walter, “Benchmarking of various lidar sensors for use in self-driving vehicles in real-world environments,” *Sensors*, vol. 22, no. 19, 2022.

[9] T. Rhemrev, E. De Jong, G. Van Triest, R. Kalkman, J. Pronk, A. Pandharipande, and N. Jonathan Myers, “Ellas: Enhancing lidar perception with location-aware scanning profile adaptation,” *IEEE Sensors Journal*, vol. 25, no. 5, pp. 8766–8775, 2025.

[10] F. Pittaluga, Z. Tasneem, J. Folden, B. Tilmon, A. Chakrabarti, and S. J. Koppal, “ Towards a MEMS-based Adaptive LIDAR ,” in *2020 International Conference on 3D Vision (3DV)*, (Los Alamitos, CA, USA), pp. 1216–1226, IEEE Computer Society, Nov. 2020.

[11] E. Gofer, S. Praisler, and G. Gilboa, “Adaptive lidar sampling and depth completion using ensemble variance,” *IEEE Transactions on Image Processing*, vol. 30, pp. 8900–8912, 01 2021.

[12] M. A. A. Belmekki, R. Tobin, G. S. Buller, S. McLaughlin, and A. Halimi, “Fast task-based adaptive sampling for 3d single-photon multispectral lidar data,” *IEEE Transactions on Computational Imaging*, vol. 8, pp. 174–187, 2022.

[13] H. K. Kondaveeti, N. K. Kumaravelu, S. D. Vanambathina, S. E. Mathe, and S. Vappangi, “A systematic literature review on prototyping with arduino: Applications, challenges, advantages, and limitations,” *Computer Science Review*, vol. 40, p. 100364, May 2021.

[14] T. Nakahara, Y. Hara, and S. Nakamura, “Localizability based path planning on occupancy grid maps,” *Advanced Robotics*, vol. 39, no. 3, p. 127–143, 2024.

[15] S. Thrun, W. Burgard, and D. Fox, “A probabilistic approach to concurrent mapping and localization for mobile robots,” *Autonomous Robots*, vol. 5, no. 3-4, p. 253–271, 1998.

[16] S. Thrun, “Learning occupancy grid maps with forward sensor models,” *Autonom. Rob.*, vol. 15, 11 2003.

[17] E. G. Alonso, “Lidar inverse sensor modelling for occupancy grid mapping in the context of autonomous vehicles,” 06 2024.

[18] E. Kaufman, T. Lee, Z. Ai, and I. S. Moskowitz, “Bayesian occupancy grid mapping via an exact inverse sensor model,” in *2016 American Control Conference (ACC)*, pp. 5709–5715, 2016.

[19] S. E. Hadji, T. H. Hing, M. S. M. Ali, M. A. Khattak, and S. Kazi, “2d occupancy grid mapping with inverse range sensor model,” in *2015 10th Asian Control Conference (ASCC)*, pp. 1–6, 2015.

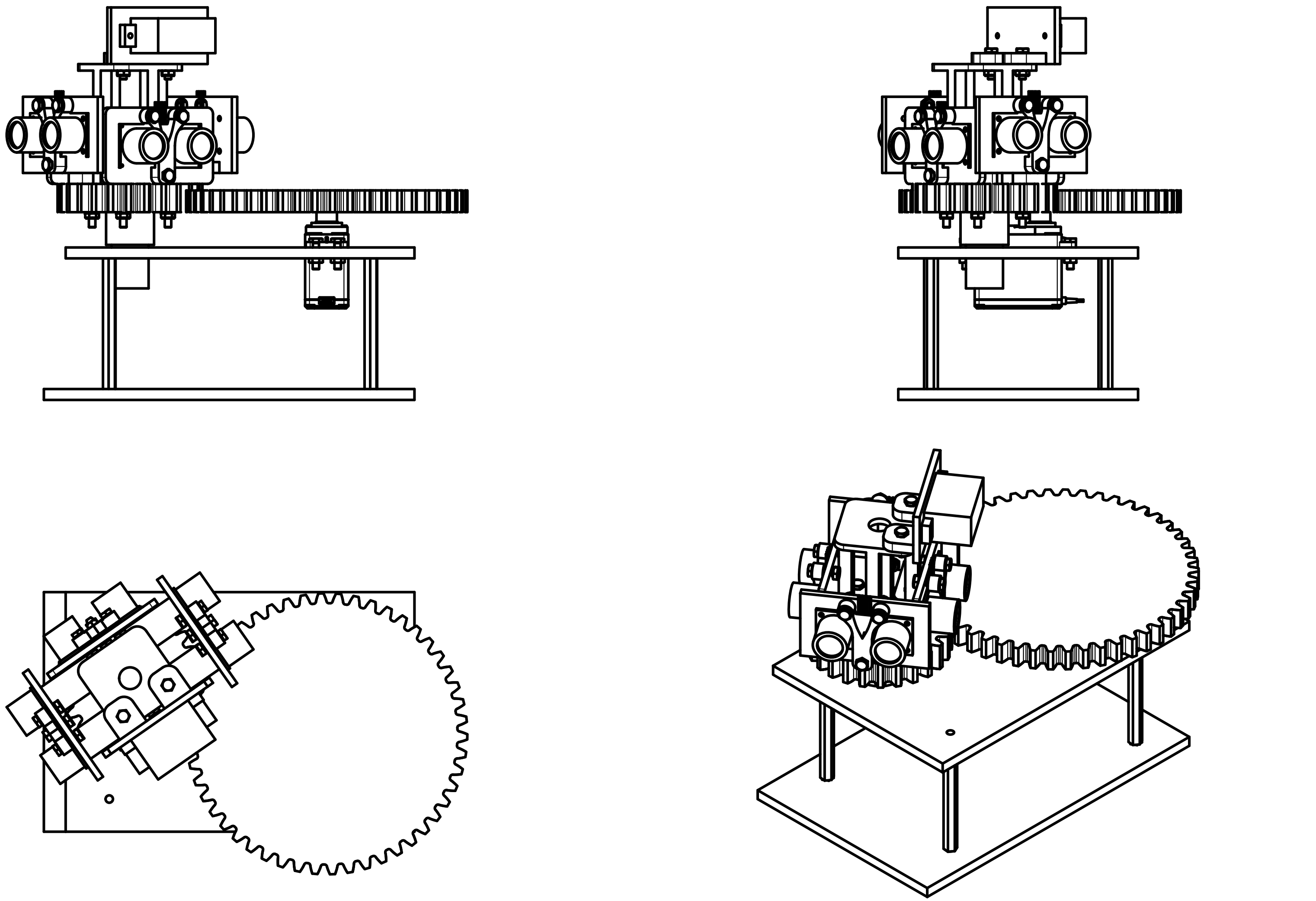
[20] G. Ferri, A. Tesei, P. Stinco, and K. D. LePage, “A bayesian occupancy grid mapping method for the control of passive sonar robotics surveillance networks,” in *OCEANS 2019 - Marseille*, pp. 1–9, 2019.

[21] K. Stepanas, J. Williams, E. Hernández, F. Ruetz, and T. Hines, “Ohm: Gpu based occupancy map generation,” *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 11078–11085, 2022.

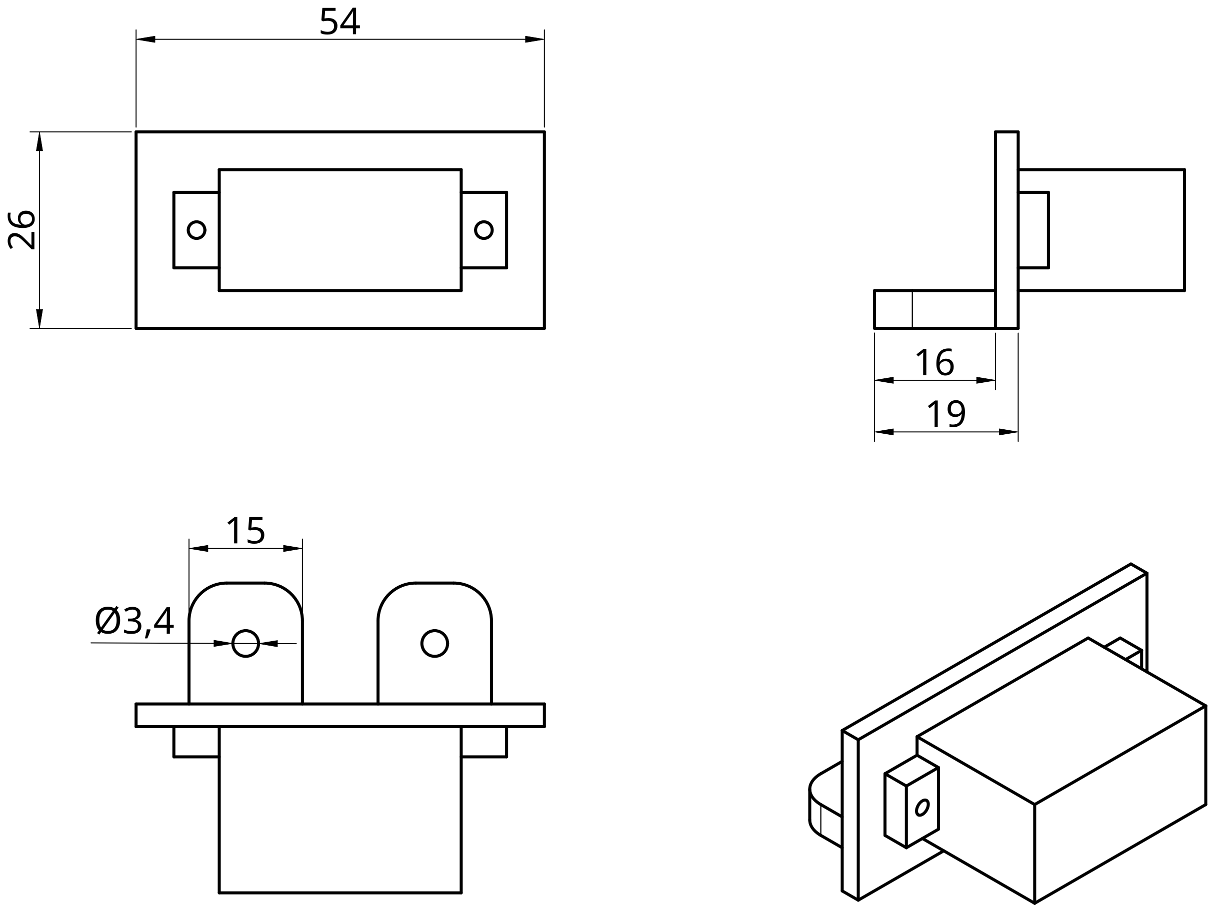
[22] Y. Cai, F. Kong, Y. Ren, F. Zhu, J. Lin, and F. Zhang, “Occupancy grid mapping without ray-casting for high-resolution lidar sensors,” *IEEE Transactions on Robotics*, vol. 40, pp. 172–192, 2024.

A

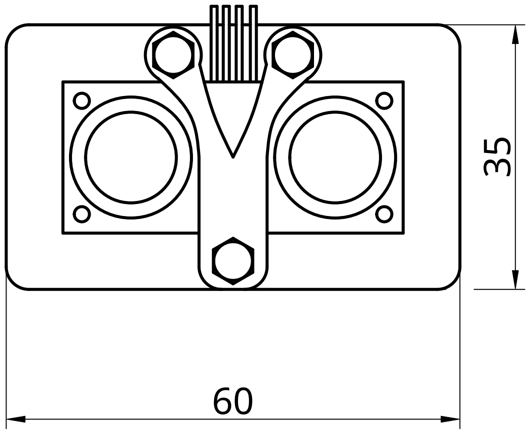
Technical drawings

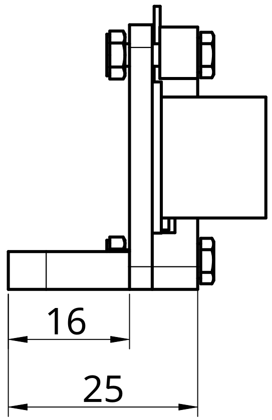


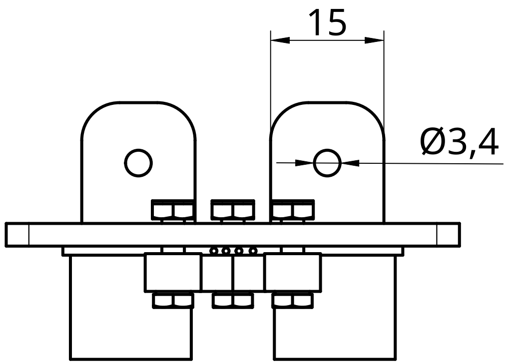
UNLESS OTHERWISE SPECIFIED, DIMENSIONS ARE IN MILLIMETERS ANGULAR = ± ° SURFACE FINISH		NAME	SIGNATURE	DATE	
	DRAWN	G. DOHMEN		2025-06-03	
	CHECKED				TITLE BAP: Adaptive LIDAR test setup
	APPROVED				
DO NOT SCALE DRAWING					
BREAK ALL SHARP EDGES AND REMOVE BURRS					SIZE A4 · DWG NO. 1 · REV. 1
FIRST ANGLE PROJECTION	MATERIAL		FINISH		SCALE 1:3 · WEIGHT · SHEET 1 of 6

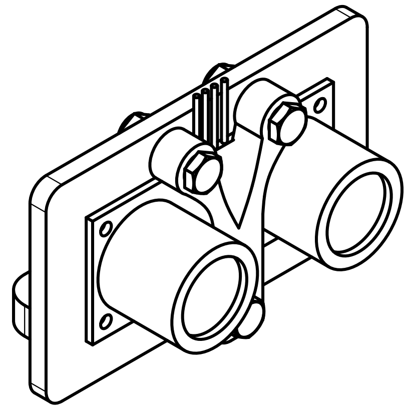


DWG NO.	SIZE	SCALE	SHEET	REV.
LIDAR mount	A4	1:1	2 of 6	



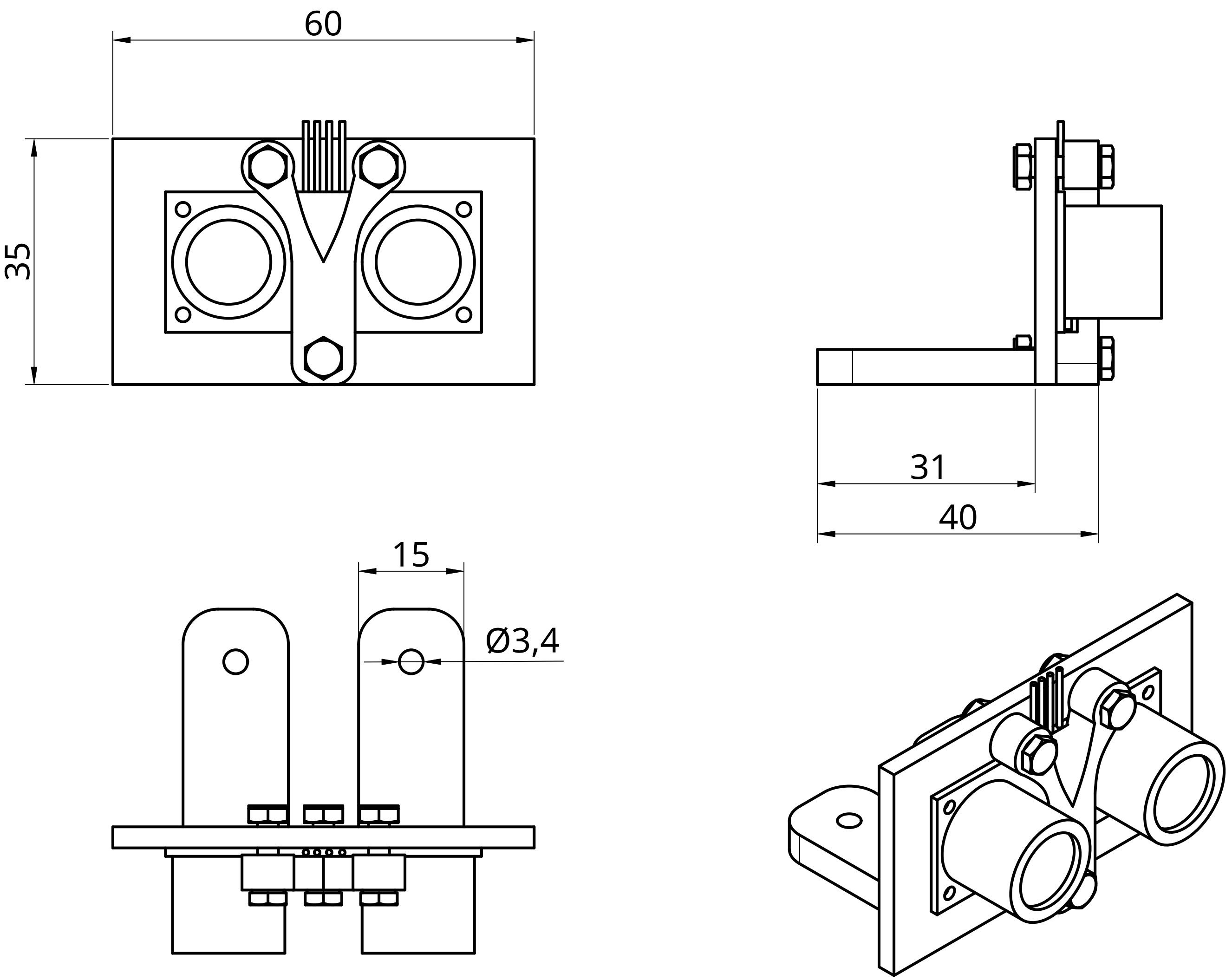






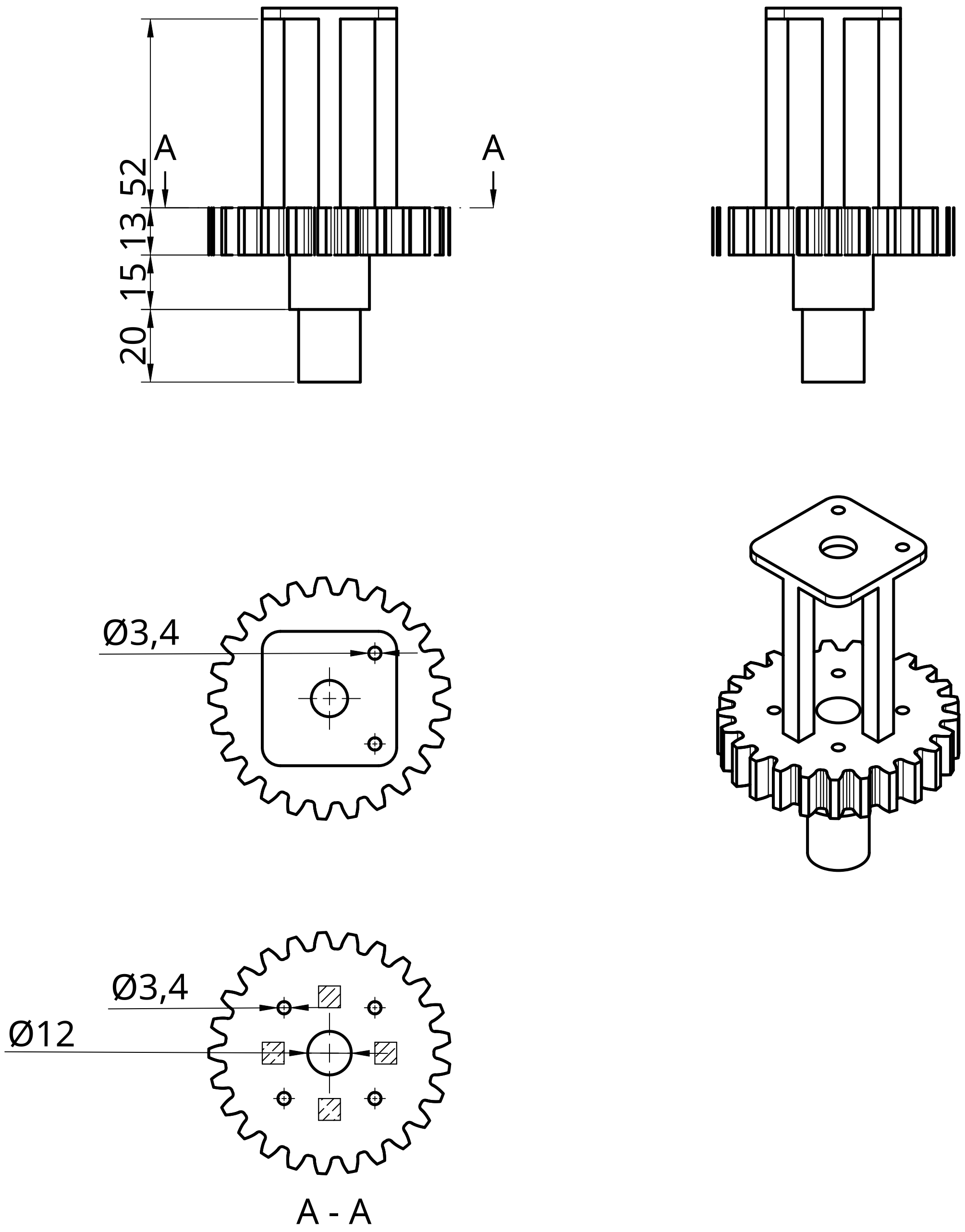
Item	Quantity	Part number	Description
1	1	SR-04	
2	1	Bodemplaat ultraosund	
3	1	Beugel ultrasound boven	
4	3	M3-12	M3 x 12mm hex bolt
5	3	M3-Nut	Hex nut style 1 grade A & B M3x0.5 Stainless Steel

DWG NO.	SIZE	SCALE	SHEET	REV.
Default Ultrasound mount	A4	1:1	3 of 6	

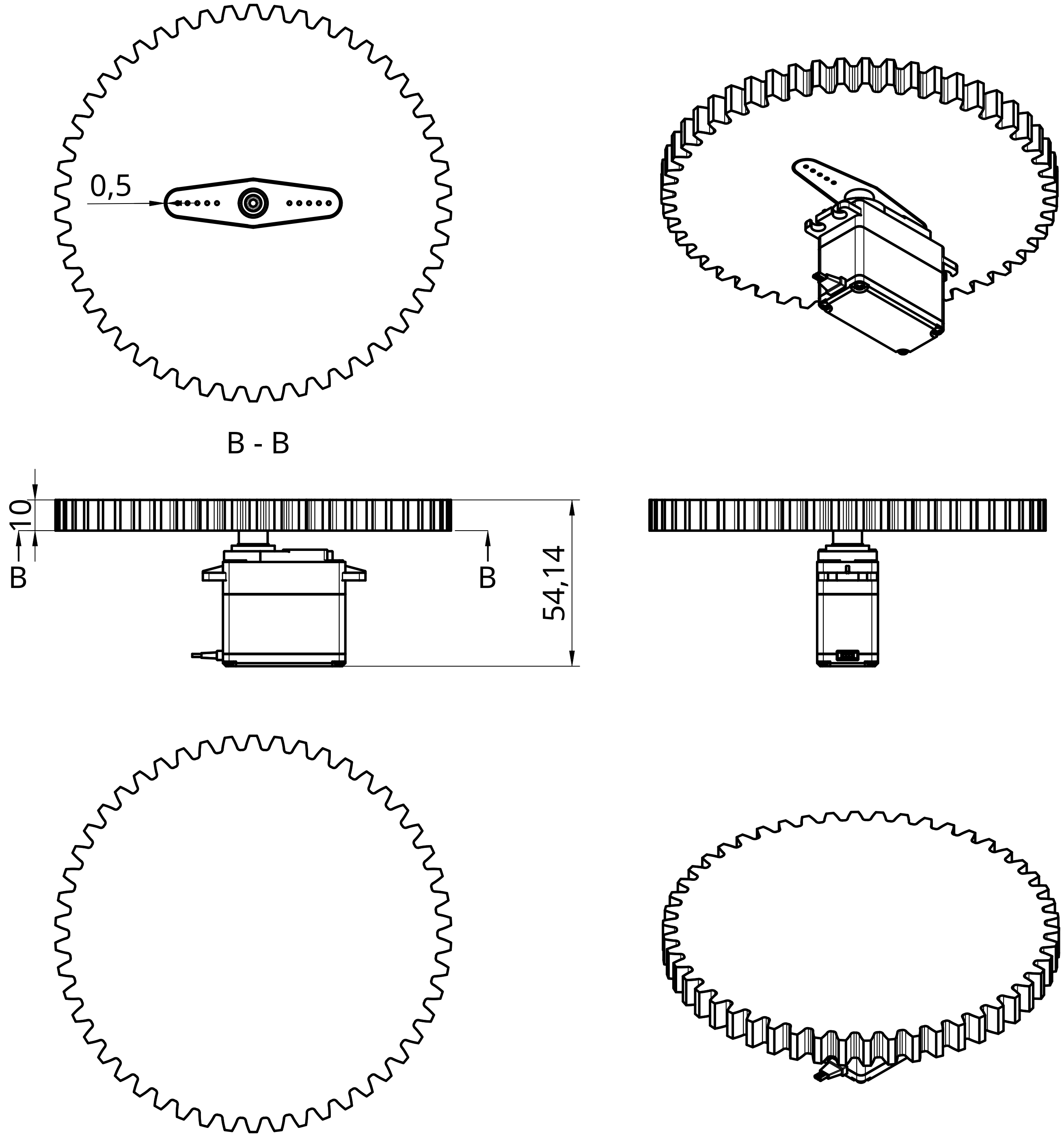


Item	Quantity	Part number	Description
1	3	M3-Nut	Hex nut style 1 grade A & B M3x0.5 Stainless Steel
2	3	M3-12	M3 x 12mm hex bolt
3	1	Beugel ultrasound boven	
4	1	Bodemplaat ultrasound verlengd	
5	1	SR-04	

DWG NO.	SIZE	SCALE	SHEET	REV.
Long Ultrasound mount	A4	1:1	4 of 6	



DWG NO.		SIZE	SCALE	SHEET	REV.
	Sensor gear	A4	1:2	5 of 6	



DWG NO.	SIZE	SCALE	SHEET	REV.
Servo gear	A4	1:2	6 of 6	

B

Python code

B.1. requirements.txt

All Python packages used in the project

```
aiofiles==24.1.0
aiohappyeyeballs==2.6.1
aiohttp==3.12.12
aiosignal==1.3.2
annotated-types==0.7.0
anyio==4.9.0
attrs==25.3.0
bidict==0.23.1
certifi==2025.4.26
charset-normalizer==3.4.2
click==8.2.1
colorama==0.4.6
contourpy==1.3.2
cycler==0.12.1
docutils==0.21.2
fastapi==0.115.12
fonttools==4.58.2
frozenlist==1.7.0
h11==0.16.0
httpcore==1.0.9
httptools==0.6.4
httpx==0.28.1
idna==3.10
ifaddr==0.2.0
iniconfig==2.1.0
itsdangerous==2.2.0
Jinja2==3.1.6
kiwisolver==1.4.8
markdown2==2.5.3
MarkupSafe==3.0.2
matplotlib==3.10.3
multidict==6.4.4
narwhals==1.42.0
nicegui==2.19.0
numpy==2.3.0
orjson==3.10.18
packaging==25.0
pandas==2.3.0
pillow==11.2.1
plotly==6.1.2
pluggy==1.6.0
propcache==0.3.2
pscript==0.7.7
pydantic==2.11.5
pydantic_core==2.33.2
Pygments==2.19.1
pyparsing==3.2.3
```

```
pyserial==3.5
pytest==8.4.0
python-dateutil==2.9.0.post0
python-dotenv==1.1.0
python-engineio==4.12.2
python-multipart==0.0.20
python-socketio==5.13.0
pytz==2025.2
PyYAML==6.0.2
requests==2.32.4
simple-websocket==1.1.0
six==1.17.0
sniffio==1.3.1
starlette==0.46.2
typing-inspection==0.4.1
typing_extensions==4.14.0
tzdata==2025.2
urllib3==2.4.0
uvicorn==0.34.3
vbuild==0.8.2
wait-for2==0.3.2
watchfiles==1.0.5
websockets==15.0.1
wsproto==1.2.0
yarl==1.20.1
```

B.2. constants.py

```
class constants:

    #comms
    arduinoIp = "http://192.168.4.1"
    """IP address of the Arduino on the robot
    """


    #offsets
    shortUltrasoundOffset = 2.47
    """Offset of the short ultrasound from center of rotation of the robot. Measured in cm
    """
    longUltrasoundOffset = 4.02
    """Offset of the long ultrasound from center of rotation of the robot. Measured in cm
    """
    lidarOffset = 4.55
    """Offset of the lidar from center of rotation of the robot. Measured in cm
    """

    #sensor data
    ultrasoundAngularResolution = 15
    """Angular resolution of the ultrasound sensor. Measured in degrees.
    """
    lidarAngularResolution = 0.1
    """Angular resolution of the LIDAR. Measured in degrees.
    """
    ultrasoundStDev = 2
    """Standard deviation of the ultrasound sensor. Measured in cm.
    """
    lidarStDev = 0.5
    """Standard deviation of the LIDAR. Measured in cm.
    """

    #GUI
    windowWidth = 1200
    """Width of the TKinter window in pixels
    """
    windowHeight = 800
    """Height of the TKinter window in pixels
    """
    programName = "BAP"
```

```
    """Name of the program. Used as program title and in the top bar
    """

    #Grid
    gridSizeX = 6
    """Size of the grid in meters, x direction
    """
    gridSizeY = 6
    """Size of the grid in meters, y direction
    """
    gridResolution = 50
    """Number of coords per meter
    """
```

B.3. main.py

```
from nicegui import ui
from program import program
import cProfile
import pstats

def main():
    #starts the profiler to see how fast the code functions run
    profiler = cProfile.Profile()
    profiler.enable()

    p = program()

    #stops the profiler
    profiler.disable()
    stats = pstats.Stats(profiler)
    stats.strip_dirs()
    #sorts the information based on the total time of the function and makes it so only the 30 slowest get
    stats.sort_stats('tottime')
    stats.print_stats(30)

if __name__ in {"__main__", "__mp_main__"}:
    #runs the main
    main()
```

B.4. program.py

```
from nicegui import ui,app
from comms import comms
from server import server
import cProfile
import pstats



class program:
    instance = None
    """ contains a reference to the only instance of this class. Init function makes sure only one program
    """
    server = None
    comms = None
    plot = None
    isCommsRunning = False


    def gui(self):
        with ui.row().classes('items-stretch'):
            with ui.card().classes('items-center text-center'):
                #making the left sound of the GUI with the buttons and which functions it calls
                ui.markdown("#Control")
                ui.label("Gather data")
                ui.button("Ultrasound", on_click=self.ultrasoundDataCollection)
                ui.button("Basic LIDAR", on_click=self.lidarBasicDataCollection)
                ui.button("Comparative LIDAR", on_click=self.lidarCompareDataCollection)
                ui.button("Adaptive LIDAR UC", on_click=self.adaptiveLidarDataCollectionUC)
```

```
                ui.button("Adaptive LIDAR AU", on_click=self.adaptiveLidarDataCollectionAU)
                ui.button("Random LIDAR data", on_click=self.randomLidarData)
                ui.button("Random ultra sonic sensor data", on_click=self.randomUltraData)
                ui.button("Reset map", on_click=self.mapreset)
                ui.button("Shutdown", on_click=self.shutdown)
            with ui.card().classes('items-center text-center'):
                #creates middle part with status information
                ui.markdown("#Comms")
                with ui.row():
                    ui.label("COM port: ")
                    ui.select(self.comms.listPorts()).bind_label(self.comms,'comPORT')
                ui.button("Start comms", on_click=self.startComms)
                ui.button("Stop Comms", on_click=self.stopComms)
            with ui.card().classes("Items-center text-center"):
                #creates right part with the grid heat map
                ui.markdown("#Occupancy grid map").classes("center")
                self.plot = ui.plotly(self.server.gridMap.drawPlotly()).style("width: 600px; height: 600px;")
        ui.timer(0.1,self.update)


    def ultrasoundDataCollection(self):
        self.comms.write("u")
        print("gather ultrasound data, sent command to arduino")

    def update(self):
        if self.isCommsRunning:
            self.comms.update()

    def lidarBasicDataCollection(self):
        self.comms.write("l")
        print("gather lidar data basic")

    def lidarCompareDataCollection(self):
        self.comms.write("o")
        print("gather comparative lidar data")

    def adaptiveLidarDataCollectionUC(self):
        self.comms.write("a")
        print("gather adaptive lidar data for Ultrasound Complement system")

    def adaptiveLidarDataCollectionAU(self):
        self.comms.write("c")
        print("gather adaptive lidar data for Automotive system")
    def randomLidarData(self):
        #runs the random lidar test data, times it and shows the 30 slowest functions
        profiler3 = cProfile.Profile()
        profiler3.enable()
        print("Adding random lidar data to map")
        self.server.testLidarArray()
        self.updateMap()
        profiler3.disable()
        stats = pstats.Stats(profiler3)
        stats.strip_dirs()
        stats.sort_stats('tottime')
        stats.print_stats(30)

    def randomUltraData(self):
        #runs the random ultrasone test data, times it and shows the 30 slowest functions
        profiler2 = cProfile.Profile()
        profiler2.enable()

        print("Adding random ultra data to map")
        self.server.testUltra()
        self.updateMap()

        profiler2.disable()
        stats = pstats.Stats(profiler2)
        stats.strip_dirs()
        stats.sort_stats('tottime')
        stats.print_stats(30)

```

```
    def mapreset(self):
        #resets the map
        print("Reseting the map")
        self.server.resetgrid()
        self.updateMap()

    def stopComms(self):
        print("stopping comms")
        self.comms.stopComms()
        self.isCommsRunning = False

    def startComms(self):
        print("Starting comms")
        self.comms.startComms()
        self.isCommsRunning = True

    def shutdown(self):
        print("shutting down")
        self.comms.stopComms()
        app.shutdown()

    def updateMap(self):
        print("updating map")
        self.plot.figure = self.server.gridMap.drawPlotly()
        self.plot.update()

    def __init__(self):
        if program.instance is not None:
            return
        print("Starting server")
        program.instance = self
        self.server = server()
        self.comms = comms()
        #ser = serial.Serial('COM5', 9600)  # Adjust port and baud rate as needed
        # while True:
        #     _ultrasone_data = ser.readline().decode('utf-8').strip()
        #     print("here")
        #     try:
        #         values = list(map(float, _ultrasone_data.split(',')))
        #         print(values)
        #         break
        #     except ValueError:
        #         print("Corrupted line:", _ultrasone_data)
        self.gui()
        #self.startCommms()
        print(self.comms.listPorts())
        ui.run(title="bap",dark=None)
        pass
```

B.5. server.py

```
import random
from matplotlib.pylab import rand
from gridMap import gridMap
from constants import constants
from sensorModel import sensorModel
import numpy as np

class server:
    status = None
    gridMap = None
    lidarSensorModel = None

    def handleLidarMeasurements(self,_angles, _data):
        """_summary_

        Args:
            _angles (int[]): angle for each measurement in degrees.
            _data (int[]): distance for each measurement in cm
        """
```

```
        _inverse_dict = {}
        print(len(_data))
        i=0
        for _packet in _angles:
            #first raycast
            _rayResult = self.gridMap.raycast(_packet)
            #then do sensor model.
            self.SensorsModel.doModel(_data[i], _rayResult, _inverse_dict)
            i+=1

    def handleUltrasoundMeasurements(self, _angleArray,_dataArray):
        _inverse_dict_2 = {}
        self.SensorsModel.doModelultra(_dataArray, _inverse_dict_2)

    def testUltra(self):
        _inverse_dict_2 = {}
        #create random ultrasone distance data and starts the inverse model
        _distances = np.random.uniform(20, 300, 24)
        self.SensorsModel.doModelultra(_distances, _inverse_dict_2)

    def testLidarArray(self):
        """Tests the sensor model using random data
        """
        #create random lidar distance data and starts the inverse model
        _testangle = np.random.uniform(0, 360, 180)
        _testData = np.random.uniform(20, 300, 180)
        self.handleLidarMeasurements(_testangle, _testData)

    def testRaycastAllQuadrants(self):
        """Tests the raytracer by casting a ray in all 8 octants
        """
        _q = 0
        while _q < 8:
            _angle = random.randrange(44) + _q*45
            _distance = random.randrange(300)
            squares = self.gridMap.raycast(_angle, _distance)
            for square in squares:
                self.gridMap.set(square[0],square[1],0)
            _q += 1

    def resetgrid(self):
        #resets the grid
        _sizeX = constants.gridSizeX*constants.gridResolution
        _sizeY = constants.gridSizeY*constants.gridResolution
        self.gridMap._grid = np.full((_sizeX,_sizeY),0.5)
        #set center of field to 0, since our robot is there. It's impossible for something else to be there ;)
        self.gridMap._grid[int(_sizeX/2),int(_sizeY/2)] = 0

    def __init__(self):
        self.gridMap = gridMap(constants.gridSizeX*constants.gridResolution,constants.gridSizeY*constants.gridRe
        self.gridMap.resolution = constants.gridResolution
        self.SensorsModel = sensorModel(constants.lidarStDev, constants.ultrasoundStDev)
        pass
```

B.6. gridmap.py

```
import math
from matplotlib.figure import Figure
import numpy as np
import plotly.graph_objects as go


class gridMap:
    """represents data on a 2D grid. Includes functions to transfer grid into matplotlib fig
    """

    sizeX = 0
    """size of the grid in the X (horizontal) direction in cells.
    """

```

```
    sizeY = 0
    """size of the grid in the Y (vertical) direction in cells.
    """

    _grid = None
    """Should not be accessed from outside of class, should use helper functions. Holds all data.
    """

    resolution = None
    """Resolution with which the size in m was converted to grid cells. Could be None
    """


    def distance(_point1, _point2):
        """Calculates the distance between two points

        Args:
            _point1 (tuple): tuple with [0] being the x and [1] being the y coordinate of the point
            _point2 (tuple): tuple with [0] being the x and [1] being the y coordinate of the point

        Returns:
            float: euclidian distance between the two points
        """
        return math.sqrt((abs(_point2[0] - _point1[0]))**2 + (abs(_point2[1] - _point1[1]))**2)


    def distanceFromCenter(self, _point):
        """Calculates distance from the center

        Args:
            _point (tuple): tuple with [0] being the x and [1] being the y coordinate of the point

        Returns:
            float: euclidian distance from the center
        """
        return gridMap.distance([self.sizeX/2,self.sizeY/2],_point)


    def calcLineEndPos(self, _angle:float):
        """Calculates the end position for a line, starting in the center of the gridmap

        Args:
            _angle (float): angle in degrees. 0 degrees is forward, CCW from there

        Returns:
            Tuple: (X,Y) of the end position
        """
        #get size and center point of the grid
        _size = self.sizeX
        _cx, _cy = self.sizeX / 2, self.sizeY / 2

        #angle in radians for our system
        _rad = math.radians(_angle + 90)

        #get the vector of the direction
        _dx = math.cos(_rad)
        _dy = math.sin(_rad)

        #normalize to grid edge
        if abs(_dx) > abs(_dy):
            scale = (_size / 2) / abs(_dx)
        else:
            scale = (_size / 2) / abs(_dy)

        end_x = _cx + _dx * scale
        end_y = _cy + _dy * scale

        return (int(end_x), int(end_y))


    def raycast(self, _angle:float):
```

```
        """Casts a ray over the gridmap on a specific angle.

        Args:
            _angle (float): angle in degrees

        Returns:
            tuple[]: array of coords of the cells crossed by the ray
        """
        #assuming center starting pos, let's get the start and end points of our line
        _lineStartPos = [int(self.sizeX/2),int(self.sizeY/2)]
        _lineEndPos = self.calcLineEndPos(_angle)

        #calculate delta's and set our initial x and y coords
        _dx = int(_lineEndPos[0]-_lineStartPos[0])
        _dy = int(_lineEndPos[1]-_lineStartPos[1])
        _x = int(_lineStartPos[0])
        _y = int(_lineStartPos[1])
        #amount of cells to visit (EG: line crossings)
        _n = np.sum([1,np.abs(_dx),np.abs(_dy)])
        #what to increment by. Either positive or negative, dependent on the delta;.
        _xInc = 0
        _yInc = 0
        if(_dx > 0):
            _xInc = 1
        elif (_dx < 0):
            _xInc = -1
        if(_dy > 0):
            _yInc = 1
        elif (_dy < 0):
            _yInc = -1

        _error = np.abs(_dx) - np.abs(_dy)
        _output = []

        while (_n>0):
            if(_x >= self.sizeX):
                #reaching out of bounds, so we should stop.
                break
            if(_y >= self.sizeY):
                #reaching out of bounds, so we should stop.
                break
            if ((_x == _lineStartPos[0]) and (_y == _lineStartPos[1])):
                #equals starting pos, so skip this.
                pass
            else:
                _output.append([_x,_y])

            if (_error > 0):
                _x += _xInc
                _error -= abs(_dy)
            else:
                _y += _yInc
                _error += abs(_dx)
            _n -= 1
        return _output


    def get(self,_posX:int,_posY:int):
        """gets a datapoint from the grid based upon a grid index

        Args:
            _posX (int): grid index to read from
            _posY (int): grid index to read from

        Returns:
            var: Value of grid at this point

        Raises:
            IndexError: X position is out of bounds
            IndexError: Y position is out of bounds
        """
```

```
        if(_posX > self.sizeX):
            raise IndexError("X position is not in this grid!")
        if(_posY > self.sizeY):
            raise IndexError("Y position is not in this grid!")

        return self._grid[_posY,_posX]


    def set(self,_posX:int,_posY:int,_value):
        """sets a datapoint of the grid to _value based upon a grid index

        Args:
            _posX (int): grid index to read from
            _posY (int): grid index to read from
            _value (_type_): value to set

        Raises:
            IndexError: X position is out of bounds
            IndexError: Y position is out of bounds
        """
        if(_posX > self.sizeX):
            raise IndexError("X position is not in this grid!")
        if(_posY > self.sizeY):
            raise IndexError("Y position is not in this grid!")

        self._grid[_posY,_posX] = _value
        return

    def print(self):
        """Prints the grid to the console
        """
        print(self._grid)
        return


    def draw(self, _sizeX:int, _sizeY:int, _dpi:float):
        """draws a heatmap of the occupancy grid map using matplotlib

        Args:
            _sizeX (int): x dimension of the figure in pixels
            _sizeY (int): y dimension of the figure in pixels
            _dpi (float): pixels per inch (resolution of image)

        Returns:
            Figure: figure of the grid
        """
        _fig:Figure = Figure(figsize=(_sizeX/_dpi,_sizeY/_dpi),dpi=_dpi)

        _ax = _fig.subplots(1,1)
        _pos = _ax.imshow(self._grid)
        _fig.colorbar(_pos)
        _ax.invert_yaxis()

        #adds a red cross to the center of the screen to indicate the robot position
        _ax.plot((self.sizeX/2),(self.sizeY/2),"r+")
        return _fig


    def drawPlotly(self):
        """Draws a heatmap of the occupancy grid map using Plotly

        Returns:
            Plotly.GraphObjects.Figure: heatmap figure
        """
        for x in range(600):
            for y in range(600):
                self._grid2[y][x] = self._grid[int(y/2)][int(x/2)]
        _fig =  go.Figure(go.Heatmap(z=self._grid2, zmin = 0.0, zmax = 1.0))
        #_fig.update_layout(xaxis_scaleanchor="y")
        return _fig

```

```python

    def __init__(self, _sizeX:int, _sizeY:int):
        """Constructs a grid based upon the number of cells in X and Y direction.

        Args:
            _sizeX (int): number of cells in X direction
            _sizeY (int): number of cells in Y direction
        """
        self.sizeX = _sizeX
        self.sizeY = _sizeY
        self._grid = np.full((self.sizeX,self.sizeY),0.5)
        self._grid2 = np.full((600,600),0.0)
        #set center of field to 0, since our robot is there. It’s impossible for something else to be there ;)
        self._grid[int(_sizeX/2),int(_sizeY/2)] = 0

        self._grid_angle = np.zeros((int(_sizeX),int(_sizeY)))
        self._grid_distance = np.zeros((int(_sizeX),int(_sizeY)))

        for x in range(self.sizeX):
            for y in range(self.sizeY):
                #filling the angle array with all angles compared to the center
                #also filling the distance array with all distance from that point to the center
                dx = x - int(_sizeX/2)
                dy = y - int(_sizeY/2)
                self._grid_angle[x, y] = math.degrees(math.atan2(int(_sizeX/2) - x, y - int(_sizeY/2))) % 360
                self._grid_distance[x, y] = math.hypot(dx, dy)

        #convert the angle array into values of the index for which ultrasone measurement it belongs
        self._grid_angle_index = np.mod(np.floor_divide(np.add(self._grid_angle, 7.5), 15), 24).astype(int)
        pass
```

B.7. sensorModel.py

```python
import math
import numpy as np
from constants import constants
import program

class sensorModel:
    stDevlidar = 0
    stDevultra = 0

    def probabilityBasedOnMeasurement(self, _measurement,_distanceFromOrigin):
        delta = abs(_distanceFromOrigin - _measurement)
        if delta <= self.stDevlidar:
            return 0.90  # likely obstacle
        elif _distanceFromOrigin < _measurement:
            return 0.20  # likely free
        else:
            return 0.50  # unknown

    def calcNewProbability(self, _measurement, _point, _inverse_dict, _distance):
        _currProbability = program.program.instance.server.gridMap._grid[_point[1],_point[0]]

        _modelProbability = self.probabilityBasedOnMeasurement(_measurement, _distance)

        #using log-odds representation, as described in Thrun Forward Sensor Models
        _dict_key = (_modelProbability,_currProbability)
        if (_dict_key in _inverse_dict):
            return _inverse_dict[_dict_key]
        else:
            eps = 1e-6
            _modelProbability = max(eps, min(1 - eps, _modelProbability))
            _currProbability = max(eps, min(1 - eps, _currProbability))


            _logProb = np.log(_currProbability / (1 - _currProbability))


            #calculate new probability
```

```
            _logProb = _logProb + np.log(_modelProbability / (1 - _modelProbability))

            #convert back to normal probability
            _newProb = round(1 - 1/(1+math.exp(_logProb)),6)
            _inverse_dict.update({_dict_key : _newProb})
            return _newProb

    def doModel (self, _measurement, _gridSquares, _inverse_dict):
        """Calculates the gridmap based upon a measurement and the respective gridsquare

        Args:
            _measurement (float): measurement in cm.
            _gridSquares (tupe): gridsquares hit by the ray
        """
        _res_measurement = _measurement/100*constants.gridResolution
        for _square in _gridSquares:
            _distance = program.program.instance.server.gridMap._grid_distance[_square[0],_square[1]]
            if (abs(_distance - _res_measurement) >= self.stDevlidar) and (_distance >= _res_measurement):
                #checks of the if the point is behind the measurement and if so, continue so the functions
                continue
            #calculate the new probability and setting this point in the grid to that
            _newProb = self.calcNewProbability(_res_measurement, _square, _inverse_dict, _distance)
            program.program.instance.server.gridMap._grid[_square[1],_square[0]] =_newProb


    def probabilityBasedOnMeasurementultra(self, _measurement,_distanceFromOrigin):
        delta = abs(_distanceFromOrigin - _measurement)
        if delta < self.stDevultra:
            return 0.50  # likely obstacle
        elif _distanceFromOrigin < _measurement:
            return 0.30  # likely free
        else:
            return 0.50  # unknown

    def calcNewProbabilityultra(self, _x, _y, _inverse_dict, _measurement, _distance):
        _currProbability = program.program.instance.server.gridMap._grid[_y,_x]

        _modelProbability = self.probabilityBasedOnMeasurementultra(_measurement, _distance)

        #using log-odds representation, as described in Thrun Forward Sensor Models
        _dict_key = (_modelProbability,_currProbability)
        if (_dict_key in _inverse_dict):
            return _inverse_dict[_dict_key]
        else:
            eps = 1e-6
            _modelProbability = max(eps, min(1 - eps, _modelProbability))
            _currProbability = max(eps, min(1 - eps, _currProbability))


            #converting old probability to log-odds
            _logProb = np.log(_currProbability / (1 - _currProbability))


            #calculate new probability
            _logProb = _logProb + np.log(_modelProbability / (1 - _modelProbability))

            #convert back to normal probability
            _newProb = round(1 - 1/(1+math.exp(_logProb)),6)
            _inverse_dict.update({_dict_key : _newProb})
            return _newProb

    def doModelultra (self, _measurements, _inverse_dict):
        """Calculates the gridmap based upon a measurement and the respective gridsquare

        Args:
            _measurement (float): measurement in cm.
            _gridSquares (tupe): gridsquares hit by the ray
        """
        _res_distnaces = _measurements/100*constants.gridResolution
        print(_res_distnaces)
        for _x in np.arange(constants.gridSizeX*constants.gridResolution):
```

```
            for _y in np.arange(constants.gridSizeY*constants.gridResolution):
                _angle_index = program.program.instance.server.gridMap._grid_angle_index[_x,_y]
                _measurement = _res_distnaces[_angle_index]
                _distance = program.program.instance.server.gridMap._grid_distance[_x,_y]
                if (abs(_distance - _measurement) >= self.stDevultra) and (_distance >= _measurement):
                    #checks of the if the point is behind the measurement and if so, continue so the functions d
                    continue
                #calculate the new probability and setting this point in the grid to that
                _newProb = self.calcNewProbabilityultra(_x,_y, _inverse_dict, _measurement, _distance)
                program.program.instance.server.gridMap._grid[_y,_x] =_newProb

    def __init__(self, _stDevlidar, _stDevultra):
        self.stDevlidar = _stDevlidar
        self.stDevultra = _stDevultra
        pass
```

B.8. comms.py

```
import pandas
import serial
import serial.tools
import serial.tools.list_ports
import program
import io
import sys
import numpy as np

class comms:

    comPORT = ”COM5”
    baudrate = 9600
    serialConnection = None
    adaptiveAnglesLIDAR = None


    def __init__(self):
        pass

    def write(self, _string):
        if(self.serialConnection == None):
            raise Exception(”No serial connection has started”)
        self.serialConnection.write(_string.encode(”utf-8”))

    def update(self):
        ”””Should be called every x ms, checks if there is new data to read.
        ”””
        if(self.serialConnection == None):
            raise Exception(”No serial connection has started”)
        if(self.serialConnection.in_waiting > 0):
            #there is something to read, so let’s attempt that
            bytes = self.serialConnection.readline()
            self.handleRead(bytes)

    def processBasicLIDAR (self, _data):
        ”””Processes a default LIDAR measurement. Assumes 180 measurements spread equally around 360 degrees
            Assumes that the data is rounded values in cm

        Args:
            _data (dataFrame): pandas dataframe
        ”””
        print(”Received basic LIDAR data! Processing now”)
        _angleArray = np.arange(0, 360, 2) #expects 180 datapoints, spaced every 2 degrees.
        _dataFrameArray = np.array(_data.iloc[0])
        #remove first element, since that indicates the type of data
        _dataFrameArray = _dataFrameArray[1:]
        program.program.instance.server.handleLidarMeasurements(_angleArray,_dataFrameArray)

    def processAdaptiveLIDAR (self, _angles, _data):
        ”””Processes an adaptive LIDAR measurement.”””
        print(”Received advanced LIDAR data! Processing now”)
```

```
        _angleArray = np.array(_angles.iloc[0])
        _angleArray = _angleArray[1:]
        _dataFrameArray = np.array(_data.iloc[0])
        _dataFrameArray = _dataFrameArray[1:]
        program.program.instance.server.handleLidarMeasurements(_angleArray,_dataFrameArray)

    def processUltrasound (self, _data):
        """Processes a default ultrasound measurement. Assumes 24 measurements spread equally around 360 d
            Assumes that the data is rounded values in cm

        Args:
            _data (dataFrame): pandas dataframe
        """
        print("Received Ultrasound data! Processing now")
        _angleArray = range(360, 15)
        _dataFrameArray = np.array(_data.iloc[0])
        #remove first element, since that indicates the type of data
        _dataFrameArray = _dataFrameArray[1:]
        program.program.instance.server.handleUltrasoundMeasurements(_angleArray,_dataFrameArray)

    def handleRead(self,_buffer):
        """Processes a line from the serial port into it's subsequent processing functions

        Args:
            _buffer (byte[]): line read from serial port
        """
        print("here")
        buffer_str = _buffer.decode('utf-8')
        buffer_io = io.StringIO(buffer_str)
        print(sys.getsizeof(buffer_io))
        dataFrame = pandas.read_csv(buffer_io, header=None)
        if dataFrame.iloc[0, 0] == "u":
            #we have an ultrasound data packet
            self.processUltrasound(dataFrame)
        elif dataFrame.iloc[0, 0] == "p":
            #arduino status packet
            print(dataFrame.columns)
        elif dataFrame.iloc[0, 0] == "l":
            #we have a basic LIDAR data packet
            self.processBasicLIDAR(dataFrame)
        elif dataFrame.iloc[0, 0] == "a":
            #save lidar angles for later
            dataFrame.to_csv("Adaptive lidar angles", index = False)
            self.adaptiveAnglesLIDAR = dataFrame
        elif dataFrame.iloc[0, 0] == "b":
            dataFrame.to_csv("Adaptive lidar data", index = False)
            self.processAdaptiveLIDAR(self.adaptiveAnglesLIDAR, dataFrame)
        elif dataFrame.iloc[0, 0] == "k":
            old_frame = np.array(dataFrame.iloc[0, 1:])
            new_frame = dataFrame
            for i in range(24):
                new_frame.iloc[0, i + 1] = old_frame[(i+18)%24]
            print(new_frame.to_string)
            new_frame.to_csv("Adaptive ultrasound data", index = False)
            self.processUltrasound(new_frame)

        program.program.instance.updateMap()

    def startComms(self):
        """Sets up a communication link and listening threadf
        """
        self.serialConnection = serial.Serial(port=self.comPORT,baudrate=self.baudrate)

    def stopComms(self):
        """Stops the current serial connection
        """
        self.serialConnection.close()

    def listPorts(self):
        """Sends all comports as a list of strings

```

```
        Returns:
            _type_: _description_
        """
        stringPorts = []
        for _port in serial.tools.list_ports.comports():
            stringPorts.append(_port.name)
        return stringPorts
```

C

C++ code

Below the C++ code can be found that was use to control the ultrasonic sensors and the LiDAR. In this code the function for controlling the ultrasonic sensors and the non-adaptive LiDAR where created by our subgroup while the code for the adaptive LiDAR was written by the other subgroup.

```
#include <string.h>
#include <Servo.h>
#include <NewPing.h>
#include ”SDM15.h”
#include <Array.h>

#define SONAR_NUM 4
#define MAX_DISTANCE 400

Servo myservo;
float pos = 0;  // Variable to store the servo position
float distances1[3];
float distances2[3];  //
float distances3[3];
float distances4[3];
int lidardistances[180];
int final_distance1;
int final_distance2;  //
int final_distance3;
int final_distance4;
int datapoints[24];
float time1;
float time2;  //
float time3;
float time4;
String input = ””;


NewPing sonar[SONAR_NUM] = {
  NewPing(9, 8, MAX_DISTANCE),
  NewPing(7, 6, MAX_DISTANCE),  // also used for the adaptive boy
  NewPing(5, 4, MAX_DISTANCE),
  NewPing(3, 2, MAX_DISTANCE)
};


SDM15 sdm15(Serial1);
const int MAX_ELEMENTS = 240;             // Maximum Array elements for the lidar data
Array<float, MAX_ELEMENTS> lidar_data;    // Variable Array for storing the lidar measurements
Array<float, MAX_ELEMENTS> lidar_angles;  // Variable Array for storing the angles that belong to the lida
float sector_pointer = -3.75;             // Variable to store the position of the start of a sector
float distance_array[3];                  // Array to contain the three ultrasound measurements to be aver
float ultrasound_data[24] = {};           // Array to cointain all final ultrasound measurements

int sub_points;           // Variable to contain the amount of lidar sub points that are needed in that se
int ultrasound_flag = 0;  // Boolean flag for intrasector ultrasound measurement
```

```

void lidarcheck();
void Ultradata();
void Lidardata();
void AdaptiveLidardata();
void ComplementLidardata();


void setup() {
  Serial.begin(9600);
  myservo.attach(13);
  Serial1.begin(460800);
  lidarcheck();
}

void loop() {
  input = ””;
  pos = 0;
  myservo.write(pos);  // tell servo to go to position in variable ’pos’
  if (Serial.available() > 0) {
    input = Serial.readString();
  }
  if (input == ”u”) {
    Ultradata();
  }
  if (input == ”l”) {
    Lidardata();
  }
  if (input == ”a”) {
    AdaptiveLidardata();
  }
  if (input == ”c”) {
    ComplementLidardata();
  }
}

int degree_to_ms(float degree) {
  return int(degree * (1870 / 180) + 550);  // heeft een gekke offset vandaar de getallen
}



void Ultradata() {
  // myservo.write(0);
  for (int j = 0; j <= 5; j++) {
    myservo.writeMicroseconds(((j * 7.5) / 180) * 2000 + 500);
    delay(100);
    time1 = sonar[0].ping_median(3);
    delay(100);
    time2 = sonar[1].ping_median(3);
    delay(100);
    time3 = sonar[2].ping_median(3);
    delay(100);
    time4 = sonar[3].ping_median(3);
    //Serial.println(distances1[i]);
    final_distance1 = time1 / 58.31;
    final_distance2 = time2 / 58.31;
    final_distance3 = time3 / 58.31;
    final_distance4 = time4 / 58.31;

    if (final_distance1 == 0) {
      datapoints[j] = 403;
    } else {
      datapoints[j] = final_distance1 + 3;
    }

    if (final_distance2 == 0) {
      datapoints[j + 6] = 404;
    } else {
      datapoints[j + 6] = final_distance2 + 4;
    }
```

```

    if (final_distance3 == 0) {
      datapoints[j + 12] = 403;
    } else {
      datapoints[j + 12] = final_distance3 + 3;
    }

    if (final_distance4 == 0) {
      datapoints[j + 18] = 404;
    } else {
      datapoints[j + 18] = final_distance4 + 4;
    }
  }
  myservo.write(0);
  Serial.print("u,");
  for (int i = 0; i < 24; i++) {
    Serial.print(datapoints[i]);
    Serial.print(i < 23 ? "," : "\n");
    datapoints[i] = 0;
  }
}


void lidarcheck() {
  VersionInfo info = sdm15.ObtainVersionInfo();

  if (info.checksum_error) {
    // String message = "";
    Serial.println("checksum error");
    // for (int i = 0; i < 25; i++)
    //   message += String(info.recv[i], HEX);

    // Serial.println(message);
  }

  Serial.print("model: ");
  Serial.println(info.model);
  Serial.print("hardware_version: ");
  Serial.println(info.hardware_version);
  Serial.print("firmware_version_major: ");
  Serial.println(info.firmware_version_major);
  Serial.print("firmware_version_minor: ");
  Serial.println(info.firmware_version_minor);
  Serial.print("serial_number: ");
  Serial.println(info.serial_number);

  // get self check test
  TestResult test = sdm15.SelfCheckTest();

  if (test.checksum_error) {
    Serial.println("test checksum error");
  }

  if (test.self_check_result) {
    Serial.println("self check success");
  } else {
    Serial.println("self check failed");
    Serial.print("error code: ");
    Serial.println(test.self_check_error_code);
    return;
  }
}




int scan(float pos) {
  int result;
  sdm15.StartScan();
  ScanData data = sdm15.GetScanData();
  if (data.checksum_error) {
```

```
    //   // Serial.println("checksum error");
    // lidardistances[pos] = 0;
    result = 0;
  } else {
    //   //Serial.println(data.distance);
    result = int(data.distance / 10) + 5;
  };
  // lidar_data.push_back(result);
  // lidar_angles.push_back(pos);
  sdm15.StopScan();
  return result;
}

void Lidardata() {
  // myservo.writeMicroseconds(500);
  // delay(100);
  for (int pos = 0; pos < 180; pos += 1) {  // goes from 0 degrees to 360 degrees
    //Serial.println("here");
    // myservo.writeMicroseconds((pos/180)*2000 + 500);
    //Serial.println(pos);
    myservo.write(pos);  // tell servo to go to position in variable 'pos'
    delay(20);
    //Serial.println(pos);
    // scan(pos);  // voert 1 scan uit
    // int(data.distance / 10) + 5;
    lidardistances[pos] = scan(pos);
    //delay(10);
  }
  myservo.writeMicroseconds(500);
  Serial.print("l,");
  for (int i = 0; i < 180; i++) {
    Serial.print(lidardistances[i]);
    Serial.print(i < 179 ? "," : "\n");
    lidardistances[i] = 0;
  }
}

int calculate_sector_interest(int sector) {
  if (sector < 6) {
    return 6;
  } else {
    float edge_array[sector - 1];
    for (int i = 0; i < sector - 1; i += 1) {
      edge_array[i] = abs(ultrasound_data[i] - ultrasound_data[i + 1]);
    }
    float sum = 0;
    for (int i = 0; i < sector - 1; i += 1) {
      sum += edge_array[i];
    }
    for (int i = 0; i < sector - 1; i += 1) {
      edge_array[i] = edge_array[i] / sum;  //Normalize array
    }
    if (sector == 6) {
      return round((edge_array[sector - 6] + edge_array[sector - 5]) / 2 * 6 * (sector + 1));
    } else {
      return round((edge_array[sector - 7] + edge_array[sector - 6] + edge_array[sector - 5]) / 3 * 6 * (sector
    }
  }
}

int calculate_sector_interest_2(int sector){
  if(sector < 6){
    return 6;
  }
  float inverse_array[sector];
  for(int i = 0; i < sector; i += 1){
    inverse_array[i] = 1/ultrasound_data[i];
  }
  float sum = 0;
  for(int i = 0; i < sector; i += 1){
    sum += inverse_array[i];
```

```
  }
  for(int i = 0; i < sector; i += 1){
    inverse_array[i] = inverse_array[i]/sum;
  }
  return round(inverse_array[sector - 6]*6*sector);
}


float average(float *array) {
  float sum = 0L;
  int num_of_measer = 0;
  for (int i = 0; i < 3; i++) {
    if (array[i] > 5) {
      num_of_measer += 1;
      sum += array[i];
    }
  }
  return ((float)sum) / num_of_measer;
}


void AdaptiveUltrasound(int sector) {
  for (int i = 0; i < 3; i += 1) {

    time2 = sonar[1].ping_median(3);
    final_distance2 = time2 / 58.31;
    if (final_distance2 == 0) {
      distance_array[i] = 404;
    } else {
      distance_array[i] = final_distance2 + 4;
    }
  }
  ultrasound_data[sector] = average(distance_array);
}

void AdaptiveLidardata() {
  // pos = 0;
  sector_pointer = -3.75;
  for (int sector = 0; sector < 24; sector += 1) {
    if (sector == 0) {
      myservo.writeMicroseconds(degree_to_ms(pos));
      delay(10);
      AdaptiveUltrasound(sector);
      ultrasound_flag = 1;
      sub_points = calculate_sector_interest(sector);
      for (int i = 0; i < sub_points; i += 1) {
        if (sector_pointer + 7.5 / sub_points * i < 0) {
          pos += 7.5 / sub_points;
        } else {
          // scan(pos);
          lidar_angles.push_back(pos);
          lidar_data.push_back(scan(pos));
          pos += 7.5 / sub_points;
          myservo.writeMicroseconds(degree_to_ms(pos));
          delay(5);
        }
      }
    }

    else {
      sub_points = calculate_sector_interest(sector);
      if (sub_points == 0) {
        sub_points = 1;
      }
      for (int i = 0; i < sub_points; i += 1) {
        if ((7.5 / sub_points * i > 3.75) && (ultrasound_flag == 0)) {
          myservo.write(sector_pointer + 3.75);
          delay(10);
          AdaptiveUltrasound(sector);
          ultrasound_flag = 1;
          pos += (7.5 / sub_points);
```

```
          myservo.writeMicroseconds(degree_to_ms(pos));
          delay(5);
        } else {

          scan(pos);
          lidar_angles.push_back(pos);
          lidar_data.push_back(scan(pos));
          pos += (7.5 / sub_points);
          myservo.writeMicroseconds(degree_to_ms(pos));
          delay(5);
        }
      }
    }
    sector_pointer += 7.5;
    pos = sector_pointer;
    ultrasound_flag = 0;
  }

  Serial.print("a,");
  int size = lidar_angles.size();
  for (int i = 0; i < size; i++) {
    Serial.print(lidar_angles[0]*2);
    Serial.print(i < (size-1) ? "," : "\n");
    lidar_angles.remove(0);
    // lidar_angles[i] = 0;
  }
  Array<float, MAX_ELEMENTS> lidar_angles;  // Variable Array for storing the angles that belong to the lidar me
  Serial.print("b,");
  size = lidar_data.size();
  for (int i = 0; i < size; i++) {
    Serial.print(lidar_data[0]);
    Serial.print(i < (size-1) ? "," : "\n");
    lidar_data.remove(0);
  }
  Array<float, MAX_ELEMENTS> lidar_data;  // Variable Array for storing the lidar measurements
  Serial.print("k,");
  for (int i = 0; i < 24; i++) {
    Serial.print(ultrasound_data[i]);
    Serial.print(i < 23 ? "," : "\n");
    ultrasound_data[i] = 0;
  }
}

void ComplementLidardata() {
  sector_pointer = -3.75;
  for (int sector = 0; sector < 24; sector += 1) {
    if (sector == 0) {
      myservo.writeMicroseconds(degree_to_ms(pos));
      delay(10);
      AdaptiveUltrasound(sector);
      ultrasound_flag = 1;
      sub_points = calculate_sector_interest(sector);
      for (int i = 0; i < sub_points; i += 1) {
        if (sector_pointer + 7.5 / sub_points * i < 0) {
          pos += 7.5 / sub_points;
        } else {
          // scan(pos);
          lidar_angles.push_back(pos);
          lidar_data.push_back(scan(pos));
          pos += 7.5 / sub_points;
          myservo.writeMicroseconds(degree_to_ms(pos));
          delay(5);
        }
      }
    }

    else {
      sub_points = calculate_sector_interest_2(sector);
      if (sub_points == 0) {
        sub_points = 1;
      }
```

```
      for (int i = 0; i < sub_points; i += 1) {
        if ((7.5 / sub_points * i > 3.75) && (ultrasound_flag == 0)) {
          myservo.write(sector_pointer + 3.75);
          delay(10);
          AdaptiveUltrasound(sector);
          ultrasound_flag = 1;
          pos += (7.5 / sub_points);
          myservo.writeMicroseconds(degree_to_ms(pos));
          delay(5);
        } else {

          scan(pos);
          lidar_angles.push_back(pos);
          lidar_data.push_back(scan(pos));
          pos += (7.5 / sub_points);
          myservo.writeMicroseconds(degree_to_ms(pos));
          delay(5);
        }
      }
    }
    sector_pointer += 7.5;
    pos = sector_pointer;
    ultrasound_flag = 0;
  }

  Serial.print("a,");
  int size = lidar_angles.size();
  for (int i = 0; i < size; i++) {
    Serial.print(lidar_angles[0]*2);
    Serial.print(i < (size-1) ? "," : "\n");
    lidar_angles.remove(0);
    // lidar_angles[i] = 0;
  }
  Array<float, MAX_ELEMENTS> lidar_angles;  // Variable Array for storing the angles that belong to the li
  Serial.print("b,");
  size = lidar_data.size();
  for (int i = 0; i < size; i++) {
    Serial.print(lidar_data[0]);
    Serial.print(i < (size-1) ? "," : "\n");
    lidar_data.remove(0);
  }
  Array<float, MAX_ELEMENTS> lidar_data;  // Variable Array for storing the lidar measurements
  Serial.print("k,");
  for (int i = 0; i < 24; i++) {
    Serial.print(ultrasound_data[i]);
    Serial.print(i < 23 ? "," : "\n");
    ultrasound_data[i] = 0;
  }
}
```

D

Test result figures

In this appendix, all the figures of chapter 5 are shown again. Here they are shown in a bigger size for better readability.

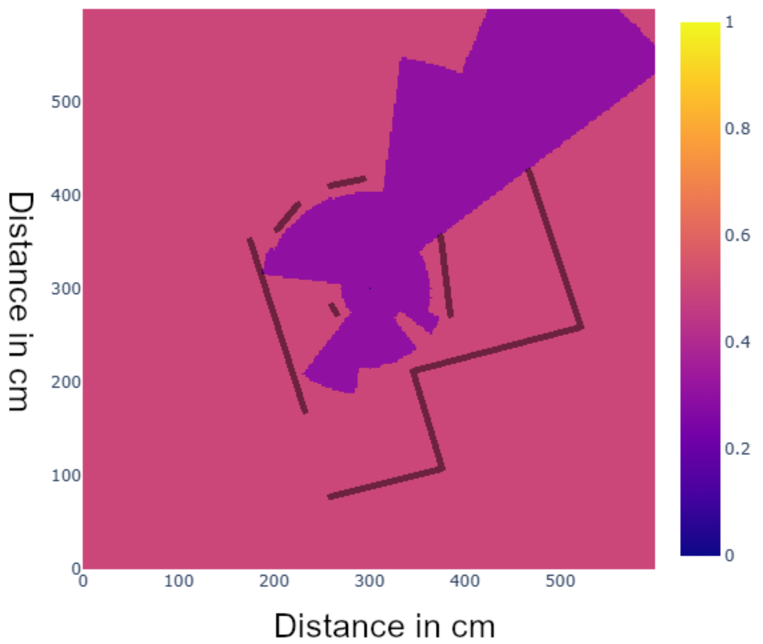


Figure D.1: One ultrasonic measurement figure 5.1a

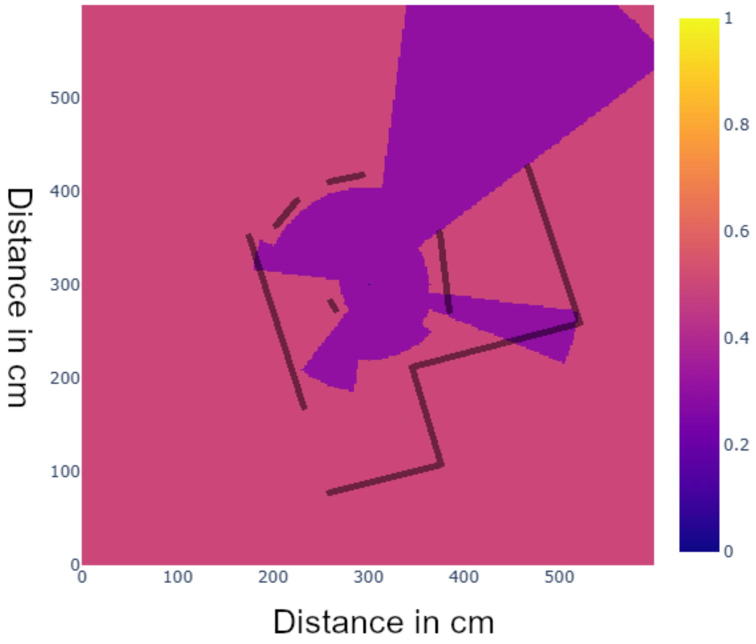


Figure D.2: One ultrasonic measurement figure 5.1b

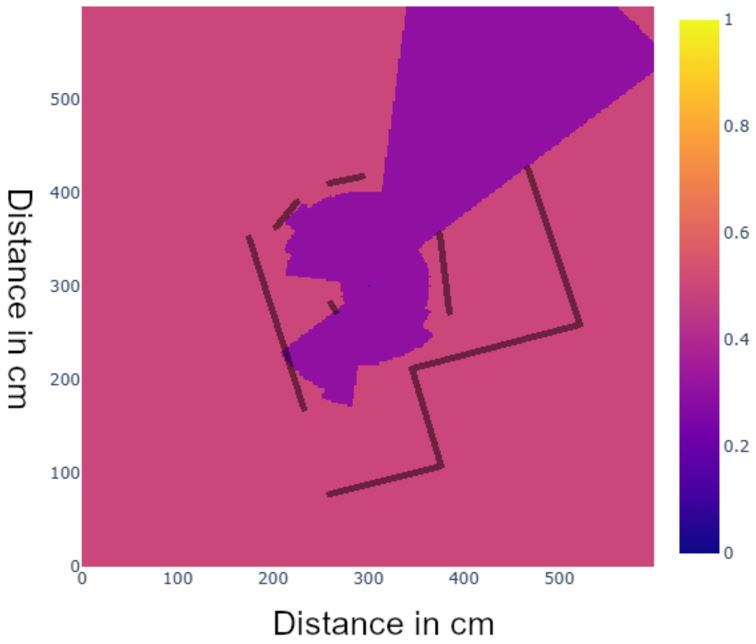


Figure D.3: One ultrasonic measurement figure 5.1c

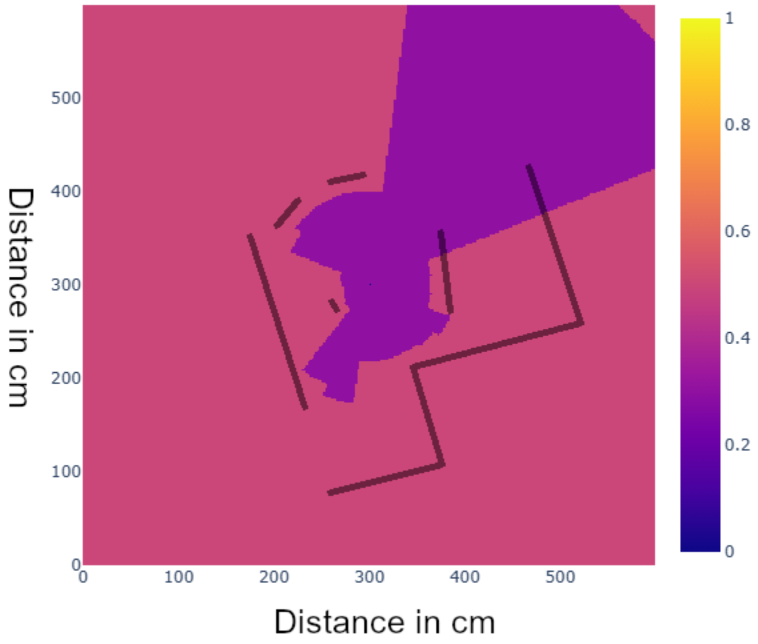


Figure D.4: Average three ultrasonic measurements figure 5.2a

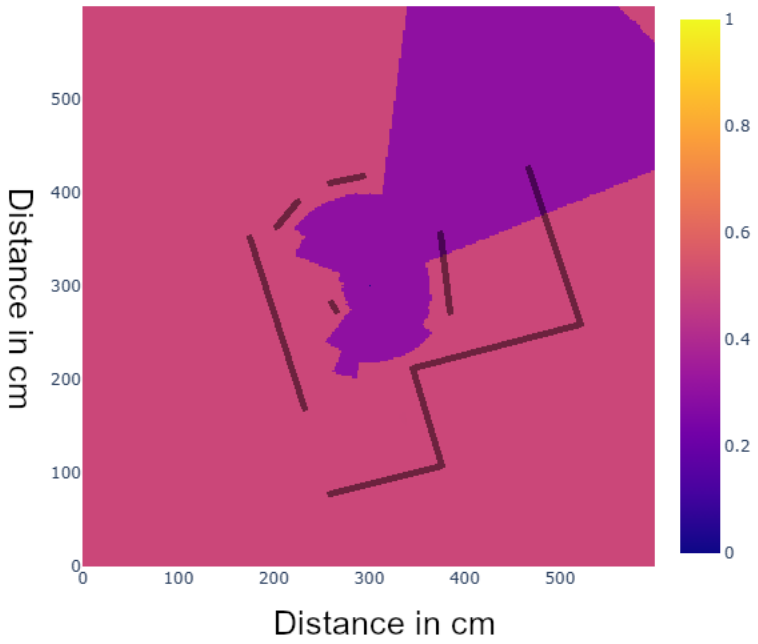


Figure D.5: Average three ultrasonic measurements figure 5.2b

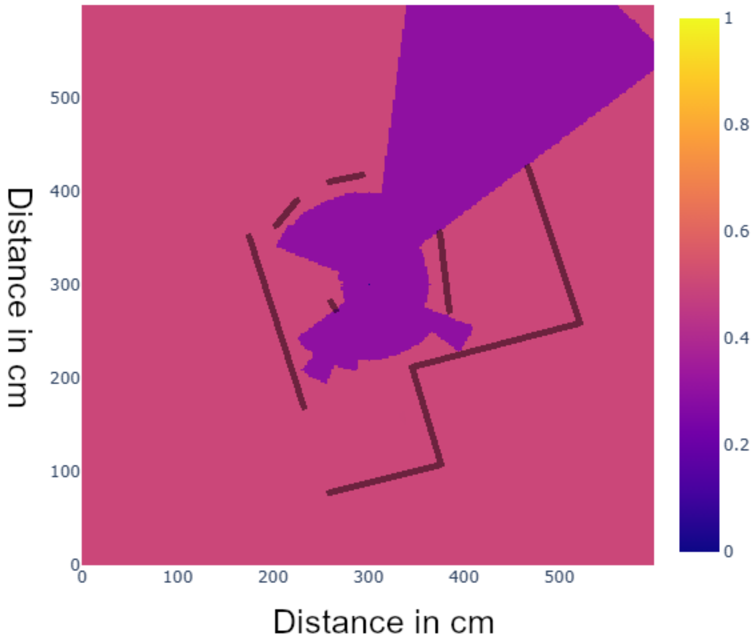


Figure D.6: Average three ultrasonic measurements figure 5.2c

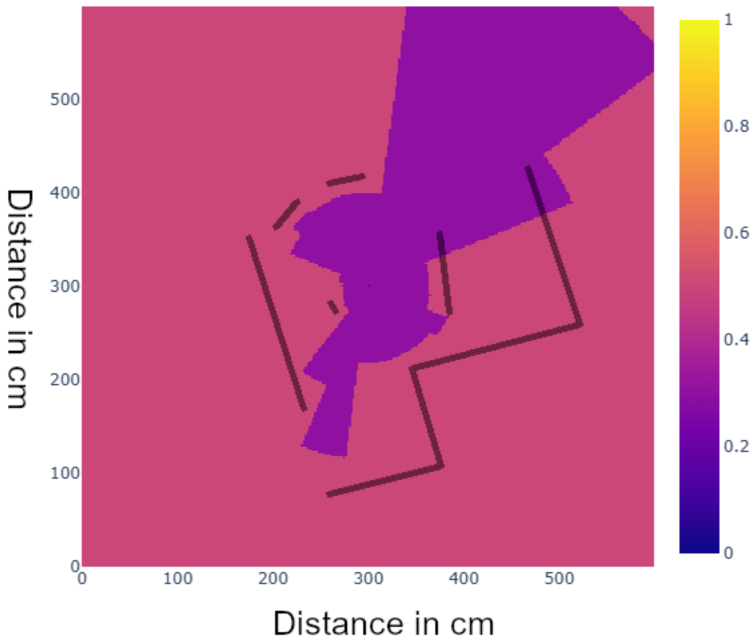


Figure D.7: Average five ultrasonic measurements figure 5.3a

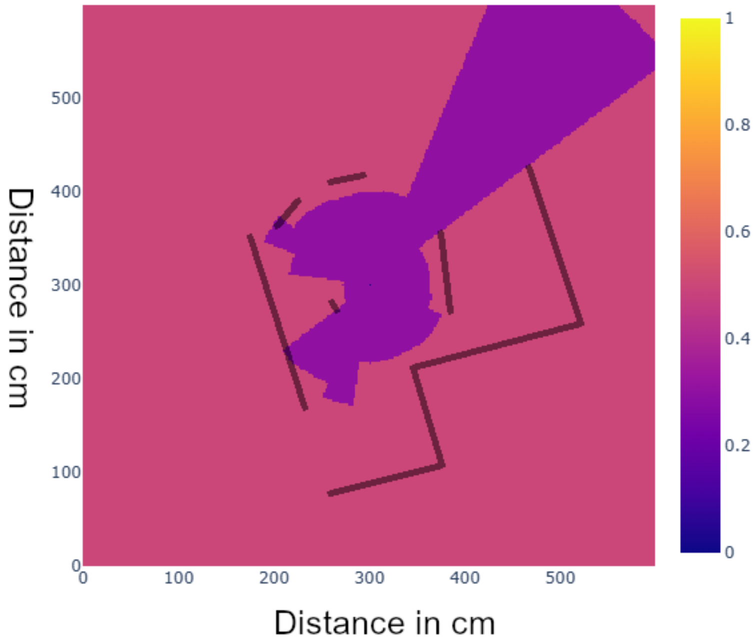


Figure D.8: Average five ultrasonic measurements figure 5.3b

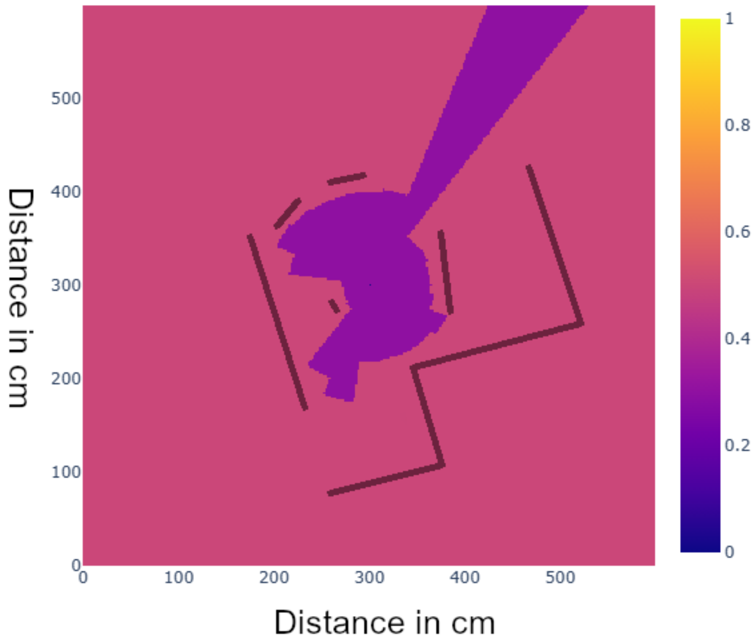


Figure D.9: Average five ultrasonic measurements figure 5.3c

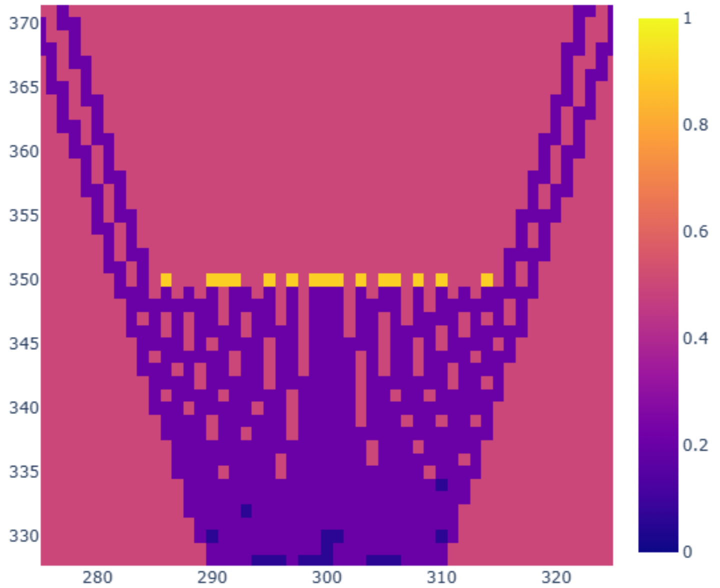


Figure D.10: Distance 50 cm and resolution 100 figure 5.4a

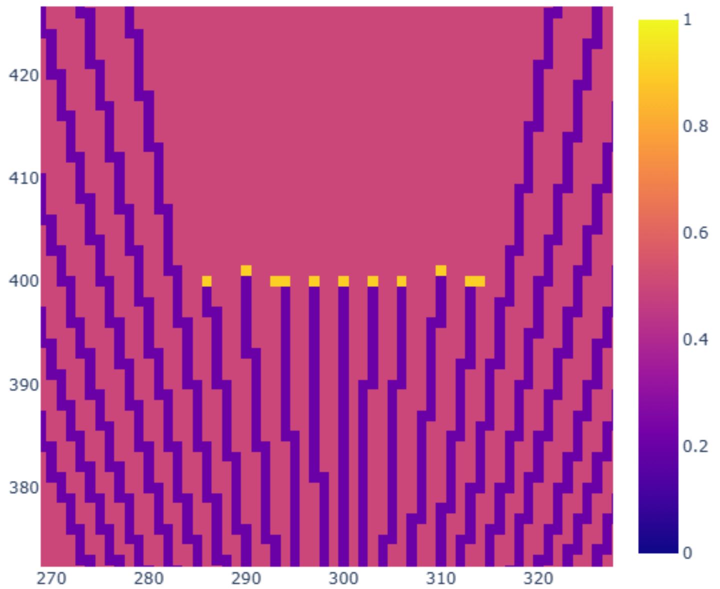


Figure D.11: Distance 100 cm and resolution 100 figure 5.4b

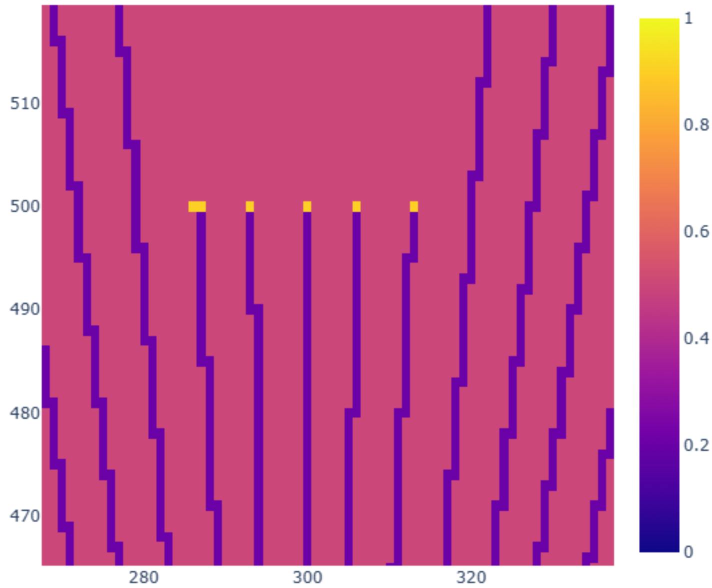


Figure D.12: Distance 200 cm and resolution 100 figure 5.4c

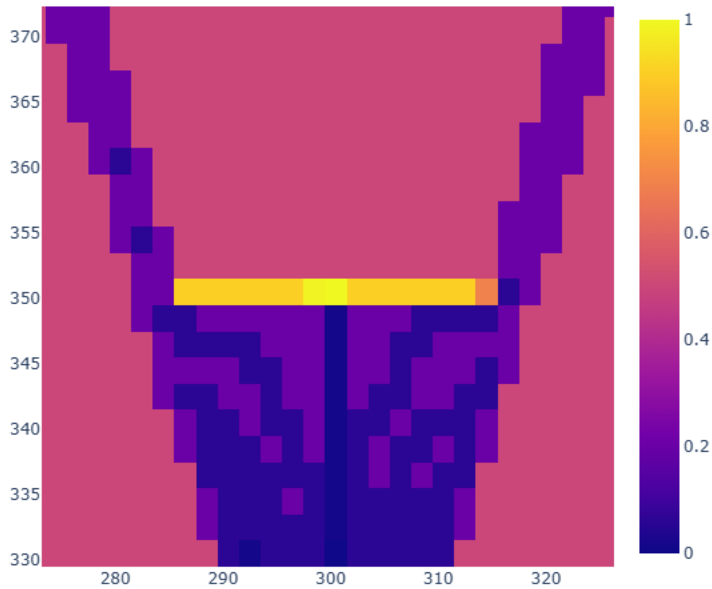


Figure D.13: Distance 50 cm and resolution 50 figure 5.4d

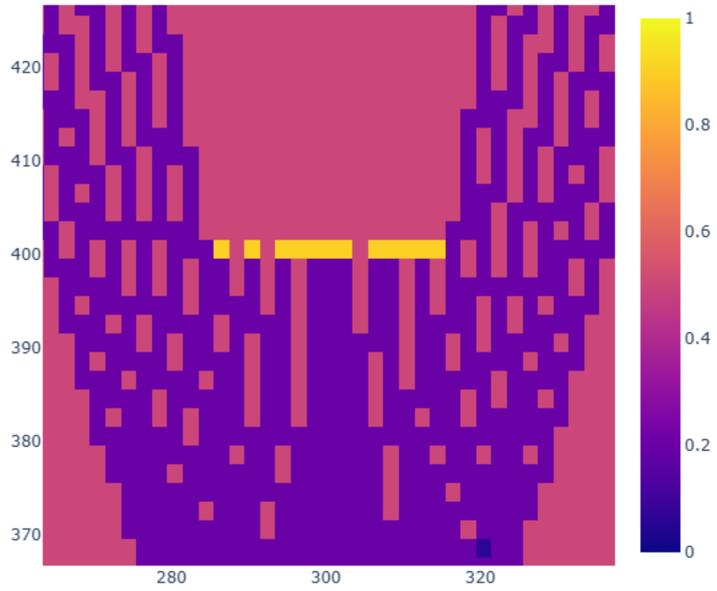


Figure D.14: Distance 100 cm and resolution 50 figure 5.4e

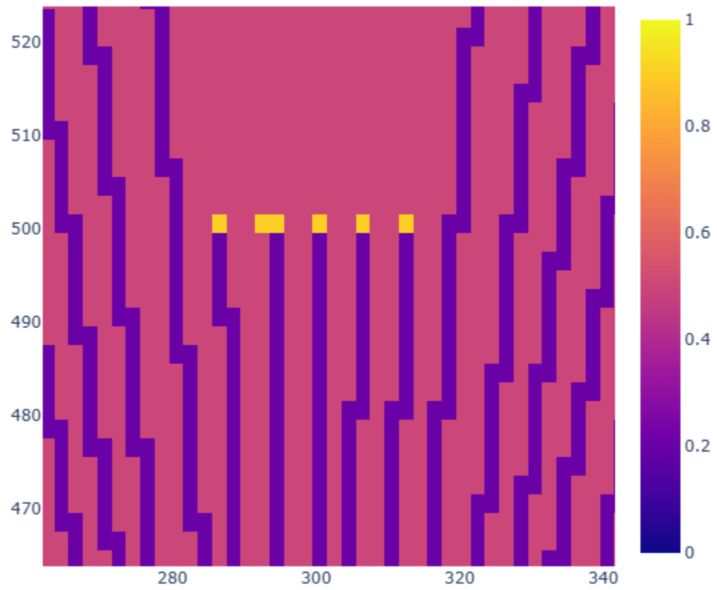


Figure D.15: Distance 200 cm and resolution 50 figure 5.4f

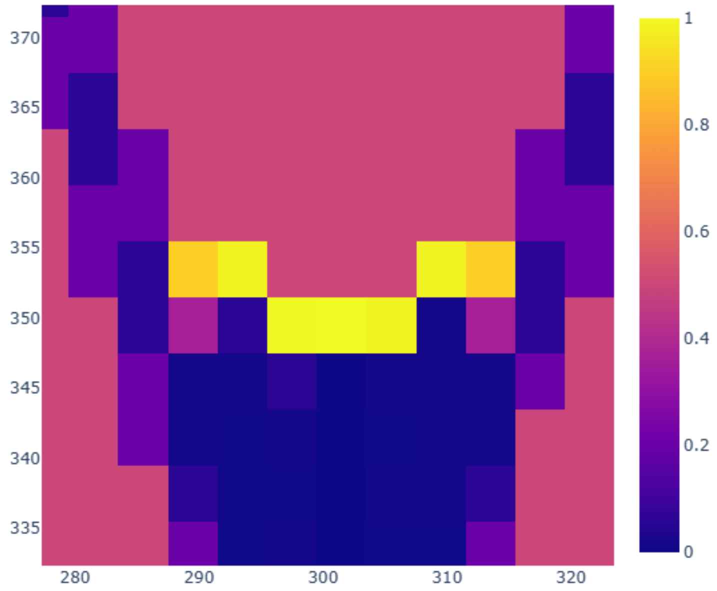


Figure D.16: Distance 50 cm and resolution 25 figure 5.4g

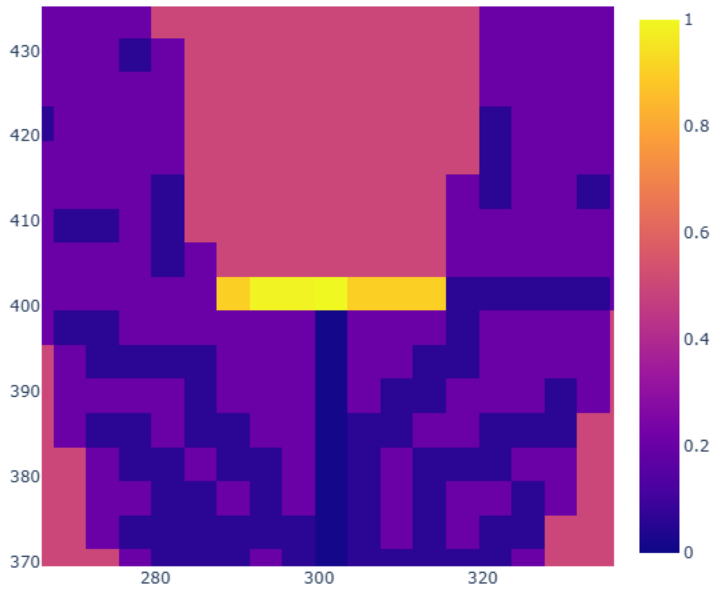


Figure D.17: Distance 100 cm and resolution 25 figure 5.4h

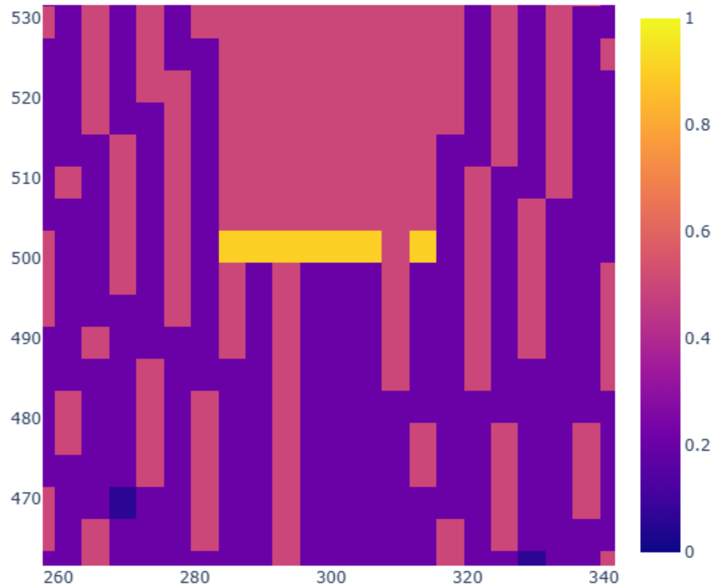


Figure D.18: Distance 200 cm and resolution 25 figure 5.4i

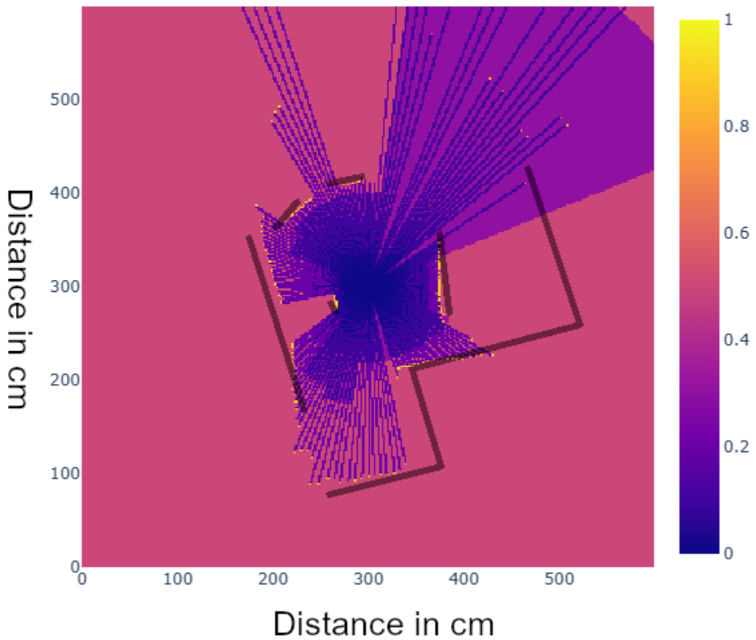


Figure D.19: Figure 5.5a

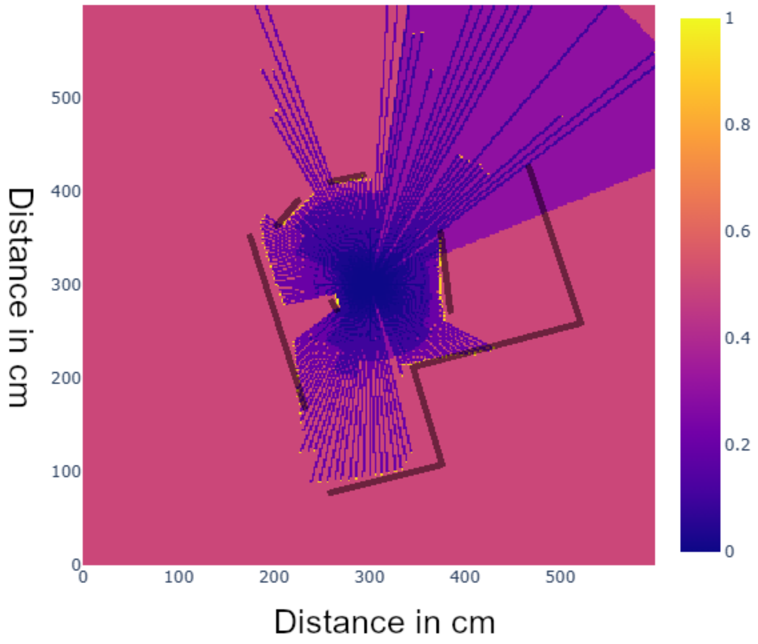


Figure D.20: Figure 5.5b

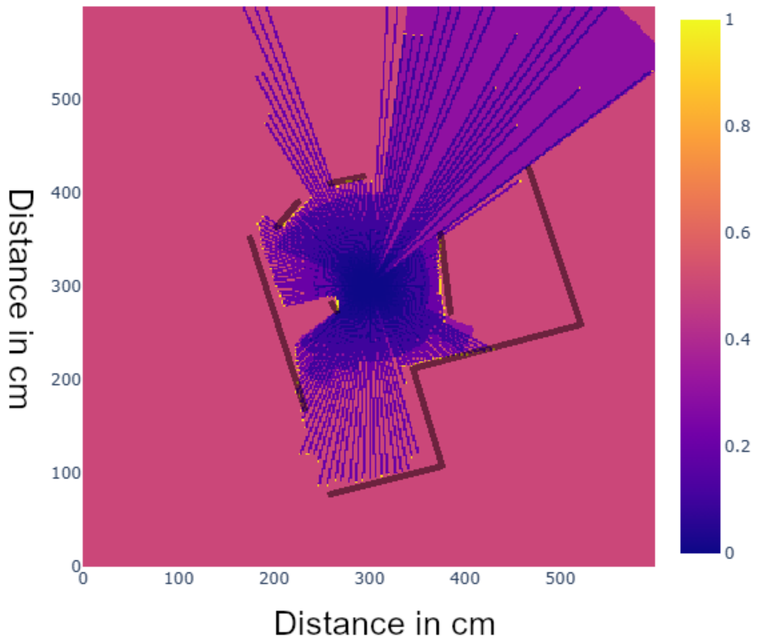


Figure D.21: Figure 5.5c