University of Southern Denmark

Introduction to Robotics, Computer Vision and Artificial Intelligence, (e19) - RCA5

Civ. Robot Technology 5th semester

Deadline for handing-in: December 8th 2019

Marble Finding Robot

Student:
Andreas Kjeldsgaard Jensen
ankj17@student.sdu.dk
08/03-1997
Eks.nr.: 113284997

Student:
Troels Zink Kristensen
tkris17@student.sdu.dk
30/04-1997
Eks.nr.: 455427

Lector:
Anders Lyhne Christensen
andc@mmmi.sdu.dk

Lector:
Jakob Wilm
jaw@mmmi.sdu.dk

Lector:
Thomas Fridolin Iversen
thfi@mmmi.sdu.dk

Table of Contents

1 Introduction **1**

2 Fuzzy Control **2**
2.1 Requirements . . . 2
2.2 Linguistic Variables and Values . . . 2
2.3 Rules . . . 4
2.4 Test of Design . . . 5
2.5 Conclusion . . . 6

3 Reinforcement Learning **7**
3.1 Requirements . . . 7
3.2 Q-Learning . . . 7
3.3 Test of Design . . . 9
3.4 Conclusion . . . 11

4 Image Analysis **12**
4.1 Requirements . . . 12
4.2 Design and Method . . . 12
4.3 Test of Design . . . 16
4.4 Conclusion . . . 17

5 Path Planning **19**
5.1 Requirements . . . 19
5.2 Wavefront Planner . . . 19
5.3 The Brushfire Algorithm and the GVD . . . 20
5.4 Test of Design . . . 22
5.5 Conclusion . . . 23

6 Localisation of the Robot **24**
6.1 Requirements . . . 24
6.2 Particle Filter . . . 24
6.3 Test of Design . . . 26
6.4 Conclusion . . . 28

1 Introduction

This project is a combination of three major fields: Robotics, Computer Vision and Artificial Intelligence (AI). The goal of the project is to navigate a 3D mobile robot in an environment simulated in Gazebo collecting marbles while avoiding obstacles. The field of Robotics is used to implement a path planning algorithm, and an algorithm to localise the robot in the environment. The field of Computer Vision is used to detect marbles and determine their global positions. Lastly, the field of AI consists of a fuzzy controller used to avoid obstacles while driving towards a goal. Furthermore, Q-Learning must be implemented to learn efficient navigation strategies to collect as many marbles as possible within a certain time frame.

Figure 1 shows the environments: *small world* and *big world*, used for debugging and testing.

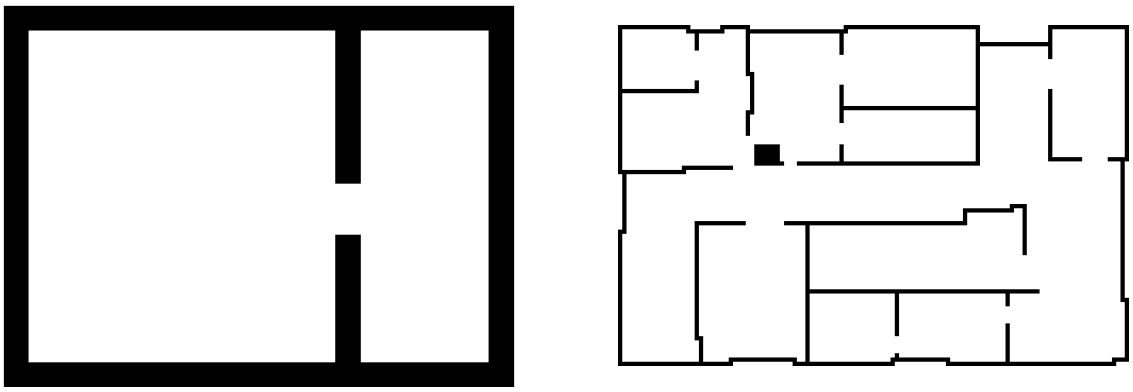

Figure 1: The floor plan of the small world and the big world, respectively.

Figure 2 shows the 3D robot shown as a 2D robot with three degrees of freedoms; that is, translation along the x- and y-axis, and orientation along yaw. The robot is equipped with a camera that has a FOV (Field Of View) of 60°. The pink circular arc shows the range of the lidar-sensors with a radius of 10cm.

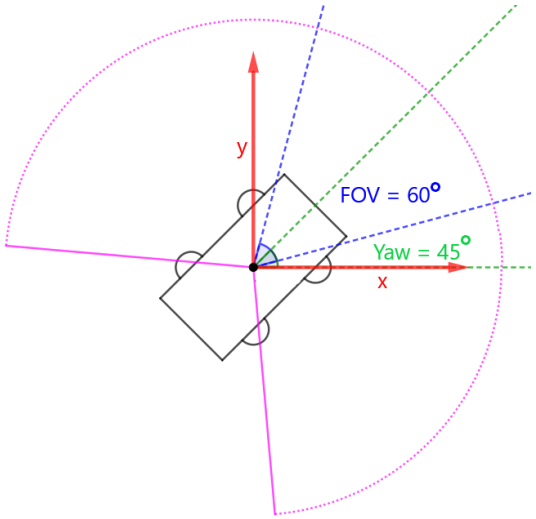

Figure 2: The 3D robot used as a 2D robot with dimensions, FOV, and lidar-sensors

2 Fuzzy Control

Fuzzy control is a method of low-level control, which is able to specify how a robot should act in specific scenarios based upon a set of fuzzy rules.
Figure 3 shows how the fuzzy controller is used within a closed-loop system.

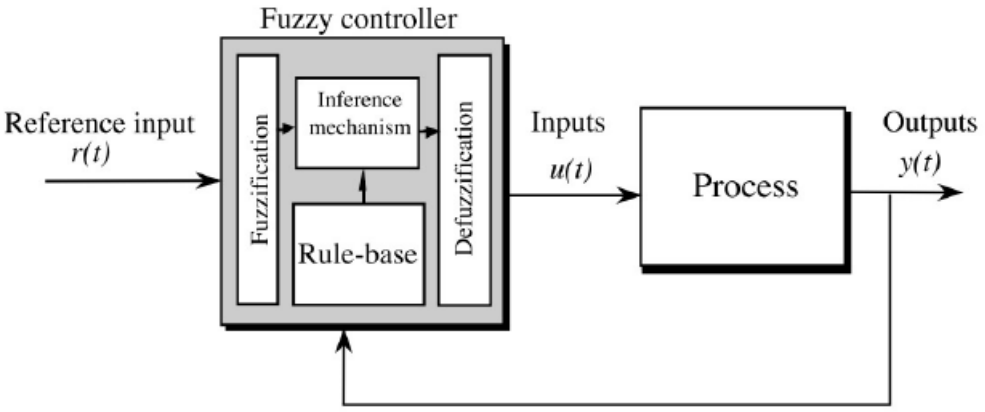

Figure 3: A closed-loop system with a fuzzy controller to fuzzify the crips values and defuzzify the linguistic values.[1]

2.1 Requirements

The robot must to be able to safely navigate around in the environment, without colliding with any obstacles. For test purposes, the robot must be able to move towards a predefined goal; therefore it needs the ability to move from one room to another through a passage. The robot should be designed with the purpose of low-level control, and will therefore not be solving any complex tasks, such as visiting more than two rooms. During the development and tests, all marbles are considered as obstacles.

2.2 Linguistic Variables and Values

Four linguistic input variables and two output variables have been created to solve the requirements. The range of linguistic values, the aggregation and the defuzzifier are chosen according to standard usage. These parameters are shown in table 1 and 2:

[1] Passino, Kevin M. and Stephen Yurkovich: *Fuzzy Control*. Page 11. 1. edition. Addison-Wesley, 1998.

DirectionToObstacle - Range: -1.00 to 1.00					
Variable	Left	DiagonalLeft	Forward	DiagonalRight	Right
Type	Ramp	Triangle	Triangle	Triangle	Ramp
Value	-0.50 -1.00	-0.6 -0.35 -0.10	-0.20 0.00 0.20	0.10 0.35 0.60	0.50 1.00
DistanceToObstacle - Range: -1.00 to 1.00					
Variable	Small	Medium		Large	
Type	Ramp	Triangle		Ramp	
Value	-0.80 -1.00	-0.90 -0.80 -0.70		-0.80 1.00	
CornerType - Range: -1.00 to 1.00					
Variable	Left		Right		
Type	Ramp		Ramp		
Value	0.25 -1.00		-0.25 1.00		
DirectionToGoal - Range: -1.00 to 1.00					
Variable	Left	DiagonalLeft	Forward	DiagonalRight	Right
Type	Ramp	Triangle	Triangle	Triangle	Ramp
Value	-0.40 -1.00	-0.5 -0.25 0.00	-0.10 0.00 0.10	0.00 0.25 0.50	0.40 1.00

Table 1: Linguistic variables and values for input variables.

Steer - Range: -1.00 to 1.00					
Variable	Left	DiagonalLeft	Forward	DiagonalRight	Right
Type	Triangle	Triangle	Triangle	Triangle	Triangle
Value	-1.00 -0.67 -0.33	-0.67 -0.33 0.00	-0.33 0.00 0.67	0.00 0.33 0.67	0.33 0.67 1.00
Speed - Range: 0.00 to 1.00					
Variable	Slow	Medium		Fast	
Type	Triangle	Triangle		Triangle	
Value	0.00 0.05 0.10	0.00 0.40 0.80		0.60 0.80 1.00	

Table 2: Linguistic variables and values for output variables.

Figure 4 represents the membership functions for the linguistic variables: *DirectionToObstacle* and *Steer*. All of the linguistic variables and values were initially chosen based upon prior knowledge and experience. The linguistic variables and values were changed according to the results of several experiments, with the purpose of improving the fuzzy controller. It became apparent that the value distribution of *DirectionToObstacle* should not be uniform, as the data from the lidar-sensors has a specific interval of $[-130°; 130°]$, which would affect the *Steer* variable, thereby causing the robot to change direction even though the obstacle was behind the robot.

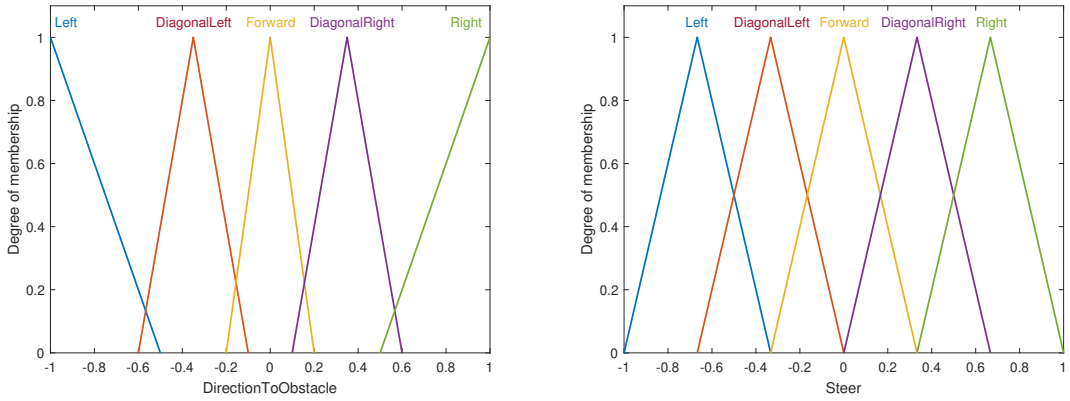

Figure 4: Membership functions for DirectionToObstacle and Steer.

2.3 Rules

A rule-base has been implemented based on the Mamdani rule-block. Several rules have been designed to achieve the requirements. Although a large number of inputs and outputs are used, the number of rules is limited to 26, which covers all possible scenarios. They are categorised in three blocks according to the distance to the nearest obstacle; as a small, medium, and large distance. Only the most unique and important rules will be described.

Whenever the distance to the nearest obstacle is either medium or large, the robot should move towards the goal. During tests it became apparent that the speed of the robot, when closing in on the obstacle, could be too high for the robot to react. Due to this issue the speed is controlled by the distance to the nearest obstacle.

- **rule:** if DistanceToObstacle is Large and DirectionToGoal is Left then Steer is Left and Speed is Fast
- **rule:** if DistanceToObstacle is Medium and DirectionToGoal is Left then Steer is Left and Speed is Medium

When the distance to the nearest obstacle is small, the robot will steer according to both the direction to the obstacle and the goal. The issue is moving towards a goal while avoiding the obstacles.

- **rule:** if DistanceToObstacle is Small and DirectionToObstacle is Left and DirectionToGoal is Left then Steer is Forward and Speed is Fast
- **rule:** if DistanceToObstacle is Small and DirectionToObstacle is Left and DirectionToGoal is Right then Steer is Right and Speed is Fast

When the robot drives towards a corner, it needs to know which way to steer. Therefore a method is implemented, which determines which type of corner it is, `left` or `right`. The method is used as an input to the fuzzy controller, `CornerType`.

- rule: if DistanceToObstacle is Small and DirectionToObstacle is Forward and CornerType is Left then Steer is Right and Speed is Slow
- rule: if DistanceToObstacle is Small and DirectionToObstacle is Forward and CornerType is Right then Steer is Left and Speed is Slow

These rules will make sure that the robot moves towards a goal, without colliding with any obstacles.

2.4 Test of Design

The fuzzy controller is tested by setting different end goals and saving its path in the environment. The fuzzy controller is tested in two different environments: *small world* and *big world*. Figure 5 shows the test results from the *small world*. The black dot is the starting position, and the coloured paths show three different paths.

Although the robot avoids obstacles in every path, it does not always reach the end goal. In the green path the robot starts by moving towards the goal in a straight line. When it gets close to the wall, it will determine which `CornerType` it is. In this scenario the robot detects the pathway as a `RIGHT CornerType`, and will therefore turn left, which is clearly not the optimal path. Because the distance to the wall is small, the robot will follow

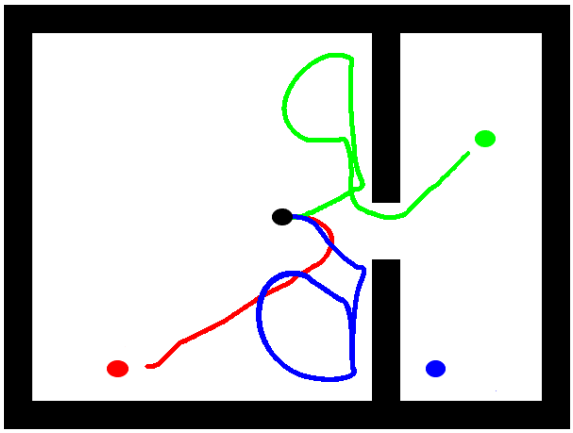

Figure 5: Test with different scenarios on the small world.

the wall until a different obstacle is closer than the current nearest obstacle. The blue path is an example, where the robot is not able to reach its end goal. Instead it will go into an endless loop, driving in a circle. The only difference between the green and the blue is the placement of the end goal compared to the pathway.

Applying the fuzzy controller to the *big world* (see figure 6), demonstrates that the relative clearance between the robot and the obstacles is much smaller compared to the *small world*. To increase safety, the clearance could have been increased. The red path is executed efficiently, as the robot does not encounter any problems. The blue path, however, encounters a problem. It moves the shortest way towards the goal, and will encounter a infinite loop going up and down, when a corner is detected, as the robot prioritises avoiding obstacles in proportion to reaching the goal. The green path has no problems, as it detects new obstacles at a convenient time. When it detects a new obstacle, the shortest path to the goal is still the correct way to go. In proportion to the cyan path, the goal is to the left of the first corner detected. Therefore, it turns around. The fuzzy controller as a single entity is not intelligent, as it only knows how to avoid obstacles and move towards a goal. It has no predefined knowledge about the environment, as it lacks artificial intelligence from Reinforcement Learning or another type of path planning.

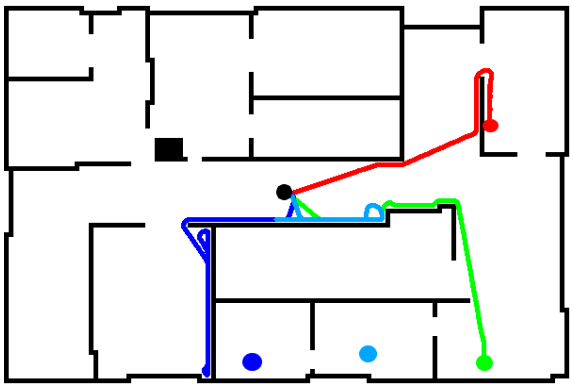

Figure 6: Test with different scenarios on the big world.

2.5 Conclusion

Based on the experiments, the robot is able to safely navigate around the two environments, *small world* and *big world*, without colliding with any obstacles. However the robot is not always able to move to an end goal, as it depends on the obstacles surrounding the path between the two. Nevertheless, if the fuzzy controller was to be implemented in conjunction with a path planning algorithm, this would not be an issue, as the path planner could define a safe path. Thereby the fuzzy controller would only overrule the path planner, if the robot drives too close to an obstacle.

3 Reinforcement Learning

In order for the robot to learn effective navigation strategies, Reinforcement Learning is used to maximise the amount of marbles collected in a specific time frame. The solution is based on a simple environment, and is focused on high-level control.

3.1 Requirements

The robot must be able to learn effective navigation strategies in a predefined environment, in order to collect as many marbles as possible in a specific time frame. To simplify the problem, the robot is only able to move in four directions: *up*, *down*, *left* and *right*. Additionally the environment is split into cells, meaning that the robot can move instantaneously between cells. Therefore the time frame can be measured in a number of steps. The implementation has to follow the Markov property.

3.2 Q-Learning

To achieve the aforementioned requirements the off-policy temporal-difference algorithm, *Q-learning*, is applied. The algorithm is implemented according to the following pseudocode segment:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 7: Q-learning algorithm.[2]

Environment

The environment used to develop and test the Q-learning implementation is grid-based, where each grid corresponds to a certain position: (x,y). Figure 8 is the implemented environment.

[2]Sutton, Richard S. and Andrew G. Barto: *Reinforcement Learning: An Introduction*. Page 131. 2. edition. MIT Press, 2018.

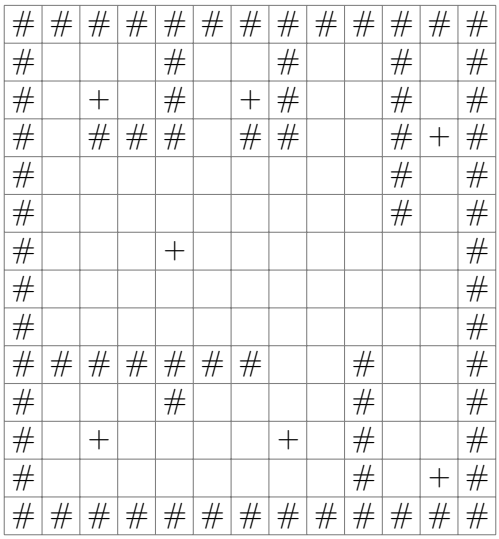

Figure 8: The test environment for Q-Learning.

Q-table

The learned action values are stored in several Q-tables. Each Q-table consists of n columns containing all possible states and m rows containing the action values for each possible step.

State (x , y)	UP	DOWN	LEFT	RIGHT
(0 , 0)	value	value	value	value
(0 , 1)	value	value	value	value
...	value	value	value	value
(m , n)	value	value	value	value

Table 3: Q-table structure

Definition of States

The states are implemented as a `struct`, which contains the following information: the Q-values for the next possible actions; the position of the current state; and whether the current state is outside the environment. To make sure the states have the Markov property, it is necessary for the robot to know which states it has been to. This is implemented by keeping track of each position, the robot has visited. For each new state the robot visits, a new Q-table is created. The path of the robot does not affect the creation of the Q-table, but only the states the robot has been to. When the robot is at a certain position, it finds the Q-table corresponding to the history of positions visited. If the Q-table does not exist, a new one is created.

3.3 Test of Design

In order to test the performance of the implemented Q-Learning algorithm, several experiments are performed. It was examined how many steps are required for the robot to collect three marbles. Additionally, it was tested separately how a change in the discount factor, the learning rate, and epsilon (ϵ) affects the performance. For this purpose a set of standard values are chosen to be compared. The standard values are as follows:

- Discount factor = 0.9
- Learning rate = 0.5
- $\epsilon = 0.1$

The first 10 episodes are excluded from the graphs, as the algorithm has yet to learn any useful paths, and the amount of steps are considered random. The blue graph in figure 9 shows the standard values, and the red graph shows a change in ϵ to 0.3. The experiment shows that the increased value of ϵ, hence increasing the rate of exploration, decreased the amount of spikes. This could be due to the relative low amount of episodes.

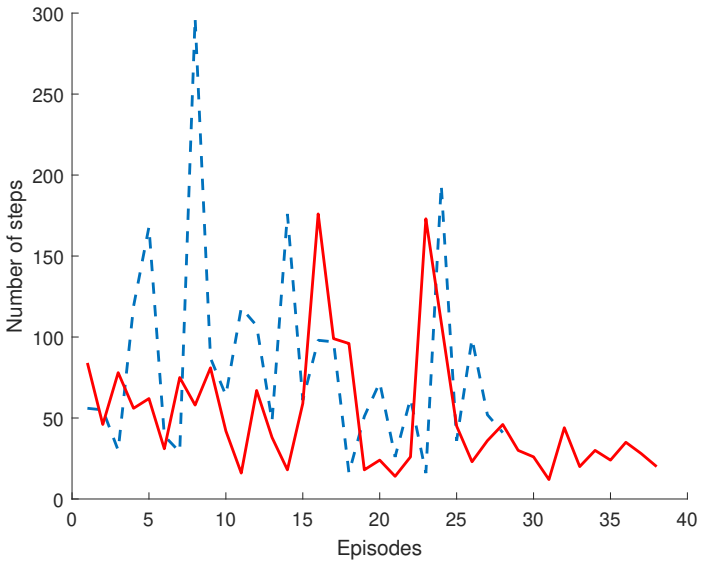

Figure 9: Standard values vs. change in ϵ to 0.3

The blue graph in figure 10 shows the standard values, and the red graph shows a change in the discount factor to 0.1. The test indicates that a lower discount factor affects the speed at which the algorithm initially learns; additionally, it reduces the amount of spikes.

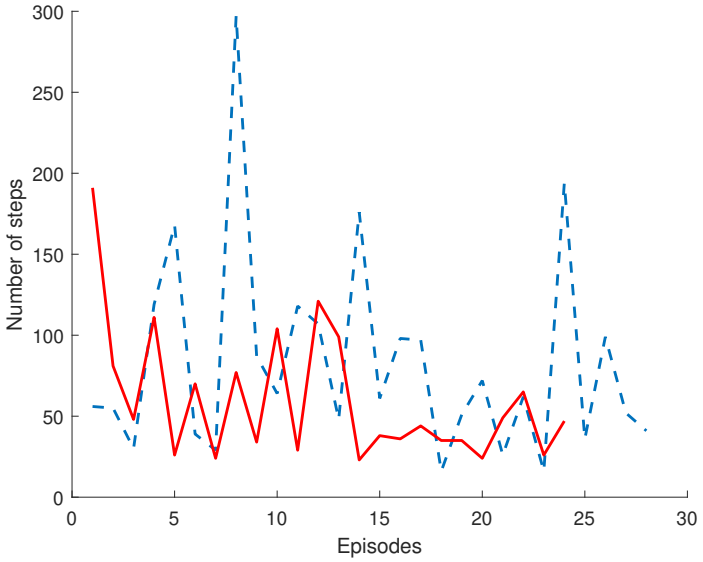

Figure 10: Standard values vs. change in discount factor to 0.1

The blue graph in figure 11 shows the standard values and the red graph shows a change in the learning rate to 0.75. The change in the learning rate changes the rate of which the agent accepts new information and forgets the old. As the amount of episodes is rather low in the tests, the increased learning rate improves the performance rapidly.

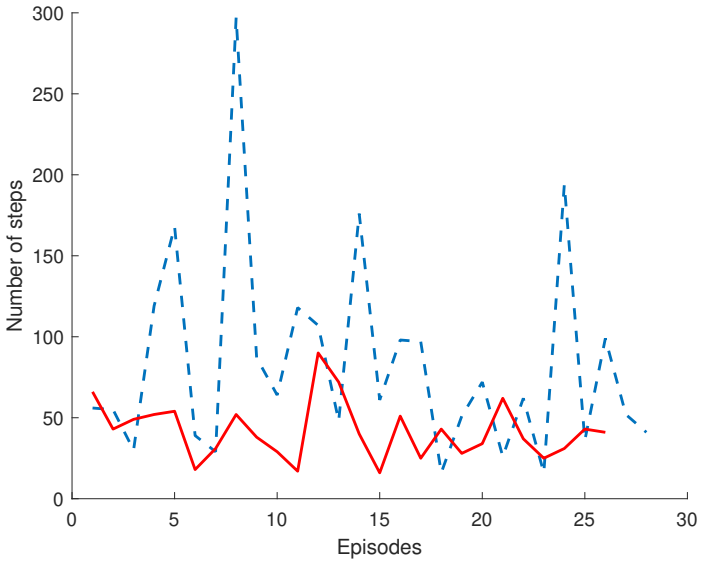

Figure 11: Standard values vs. change in learning rate to 0.75

To further develop the performance of the project a few changes could have been made. Firstly, instead of creating a new Q-table for every new state, a new table could have been created for each room in the environment. This change would increase the computational time and the overall speed of the implementation. Furthermore, it would have been

interesting to test additional changes to the standard values, and possibly examine the effect of changing the rewards.

3.4 Conclusion

The expectation of this part of the project, was to create an environment with pre-positioned marbles inside it, and examine during tests how many steps is required to collect three marbles. The implemented method is able to train and test the model with standard values for the discount factor, the learning rate and epsilon. Furthermore, these values were changed one at a time (keeping the other two standard values) to see which differences would appear. A lower discount factor and a higher learning rate made the most significant improvements to the model.
The overall result of the implemented model was not great. A large number of steps to collect only three marbles was the result of the implemented model, and could not be decreased to a relative small amount.

4 Image Analysis

In order to detect and locate the marbles, image analysis using computer vision must be implemented. The robot is equipped with a camera, which can be used in collaboration with various computer vision methods. The aim is to develop a method which can locate marbles and return their global positions in the given environment.

4.1 Requirements

In addition to detecting marbles and returning their positions, the robot should be able to detect several marbles at the same time. A marble should only be detected when a marble exists. The marbles have to be detected within a precision, that does not result in false negatives or false positives.

4.2 Design and Method

To achieve the aforementioned requirements, the following methods are used: a method to blur and denoise the camera image; edge detection to detect the boundaries of the image; a method to distinguish the marbles from the surrounding environment; and a function to determine the global position of the marbles. These methods will be discussed below.

Preprocessing

Figure 12 shows a single image of the camera in the center of *big world* without any preprocessing. There is a lot of static noise, assumed to be Gaussian noise.

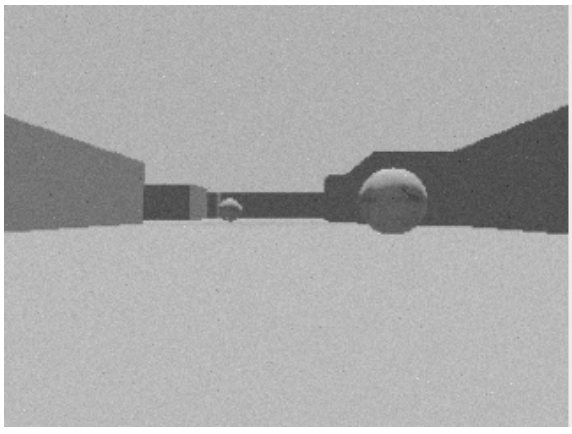

Figure 12: Original image from the camera.

One way to address this problem is the use of `GaussianBlur` to blur the image. The method determines an average of the weights surrounding a pixel based on a Gaussian distribution.

An alternative method is `fastNLMeansDenoising`, which replaces the colour of a pixel with the average colour of similar pixels. The similar pixels do not have to be neighbouring pixels, which makes the method slow compared to `GaussianBlur`.[3] The values for the methods were initially set to the standard recommended values from opencv, and thereafter changed according to performance. Figure 13 shows the result of `GaussianBlur` and `fastNLMeansDenoising`, respectively.

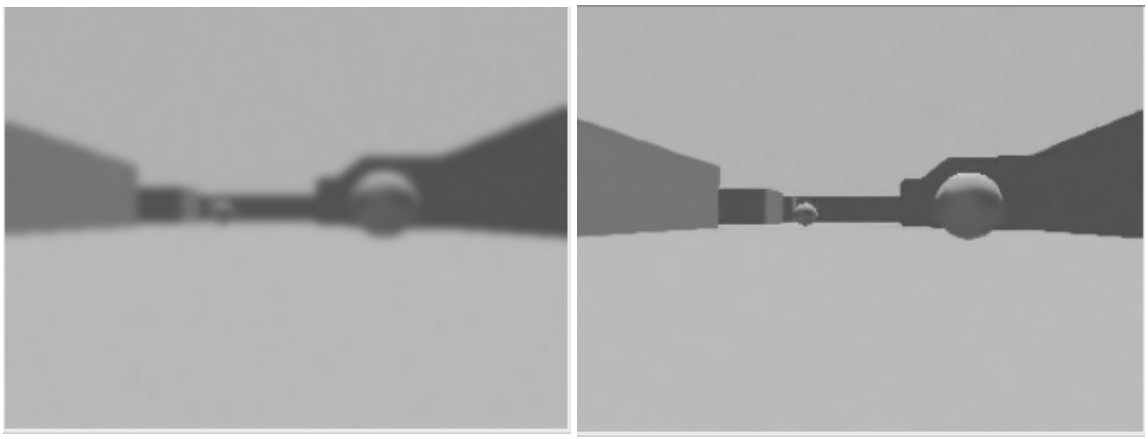

Figure 13: Original image with `GaussianBlur` (left) and `fastNlMeansDenoising` (right).

The `GaussianBlur`-method is simple to implement and to understand. However, it did not remove the noise properly compared to the level of smoothing; a larger kernel did not improve the performance significantly. Thus, the `fastNLMeansDenoising`-method was chosen to improve performance with a slightly larger computation time.

Edge Detection

After testing several different edge detection methods, Sobel derivatives turned out superior. An edge in an image can be defined as a sudden change in the pixel intensity. By calculating the derivative of the pixel intensity, edges can be detected. Figure 14 shows the result of the implemented edge detection. The edges are shown as white colour, and the rest as black colour.

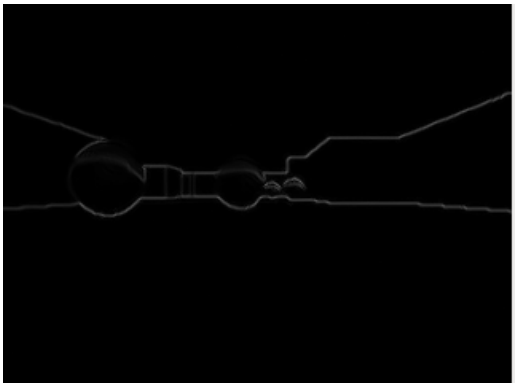

Figure 14: Edge detection with gradients and Sobel derivatives.

[3]Non-Local Means Denoising. Published by IPOL. Web address: http://www.ipol.im/pub/art/-2011/bcm_nlm/ - Visited: 2019.12.05.

Hough Transform

Along with the edge detection, it is necessary to distinguish between the edges belonging to the marbles and the remaining environment. To achieve this, the Hough transform can be used. The function used is the opencv-function: `HoughCircles`, to detect and outline a marble. The method is shown below:

cv::HoughCircles(image, circles, CV_HOUGH_GRADIENT, 1, 3000, canny_value, 30, 1, 200**)**

The fifth parameter, with the value 3000, determines the minimum distance between two circles. With this particular value, the function will only detect one circle at a time. This was chosen to avoid false detection of marbles, when a marble is visible. The `canny_value` is passed to the `opencv` Canny edge detector as a threshold. The threshold is calculated with the opencv-function: `threshold`.The seventh parameter, with a value of 30, indicates the 'minimum number of votes'; therefore, a high value corresponds to a high likelihood of a correct detection. If the value is too high, it can result in false negatives, and if it is too low, it can result in false positives. The two last parameters, 1 and 200, determines the minimum and maximum radius for a marble to be detected, respectively. These values are changed according to the requirements. The values make a decent result for the detection of marbles; see figure 15. However, the circle deviates a small amount in size in proportion to how close the robot is.

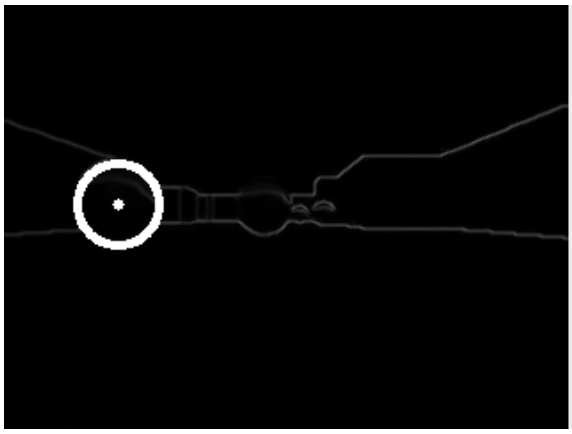

Figure 15: Hough transformation with circle detection.

Location of the Marble

The location of a marble can be determined when `HoughCircles` detects a marble. The function returns the diameter of a marble in pixels and the relative position in the camera's FOV (Field Of View). Figure 16 shows an example of determining a marble's location. The distance from the robot to the marble on the horizontal axis (n) is given by: $\frac{2 \cdot r \cdot focalLength}{Width}$ and the focal length is given by: $\frac{Width \cdot m}{2 \cdot r}$. The opposite line, o, is given by:

$\frac{centerX - cameraWidth/2}{cameraWidth} \cdot$ FOV, where *centerX* is the x-coordinate of a marble's centre in pixels (from the camera's perspective), and the *cameraWidth* is the width of the camera's FOV in pixels. With n and the opposite line, o, the true distance to the marble, m, can be calculated using trigonometry. Furthermore, the robot's current yaw, Yaw, is given by the angle between the horizontal axis and its own orientation. The radius, r, is always equal to 0.5cm, as m is actually equal to the distance to the marble's circumference plus the radius, r, of the marble. The Width (in pixels) of the marble depends on how close the robot is to the marble, as this is the relative width in proportion to the camera view.

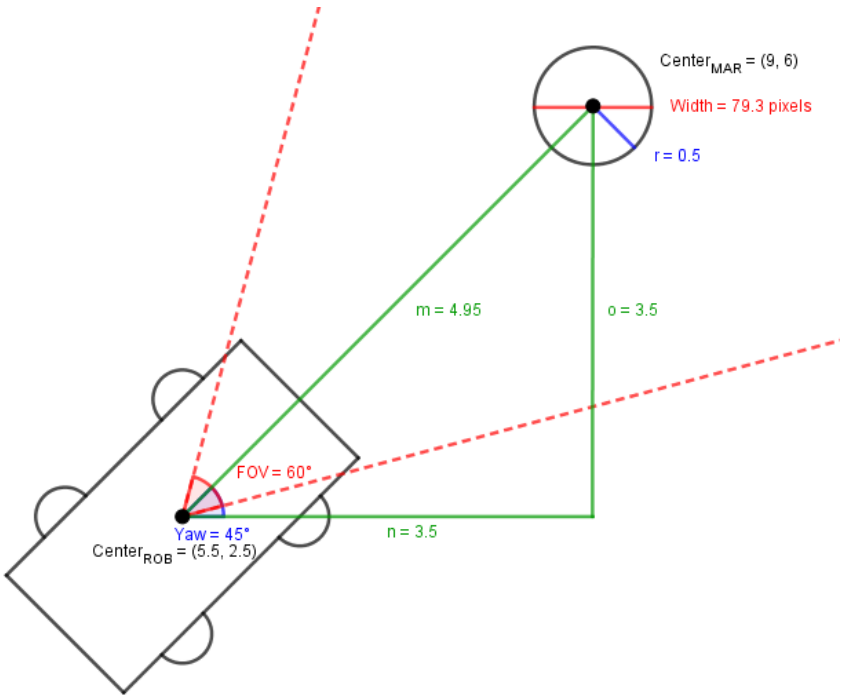

Figure 16: Determining the global position of the marble; an example. Values are in [cm] except Width, which is in [pixels].

The equations to determine the global position of the marble are given as follows:

$$x_{\text{MAR}} = x_{\text{ROB}} + m \cdot \cos(\text{Yaw}), \qquad y_{\text{MAR}} = x_{\text{ROB}} + m \cdot \sin(\text{Yaw})$$

$$m = \sqrt{n^2 + o^2}$$

However, this only implies to the situations in which the Yaw are:

$$0.25\pi \leq Yaw < 0.75\pi$$

There are three more situations, where the sign of the trigonometry part has to be the opposite, and where cosine and sine have to be the opposite of each other.

This method will only work, when the angle between the robot's orientation and the marble is zero; in other words, the robot has to face the marble directly.

4.3 Test of Design

Five different scenarios have been chosen to examine the methods' efficiency, shown in figure 17 and 18.

Scenario **1**, **2** and **3** are three different locations of the robot. The difference between the three scenarios is the distance from the robot to the marble. The orientation of the robot is shown by a black arrow.
The location of the robot and the true position of the marble for every scenario can be seen in the right side of figure 17. The estimated positions of the marbles, and the deviations from the true distance, are given in table 4.

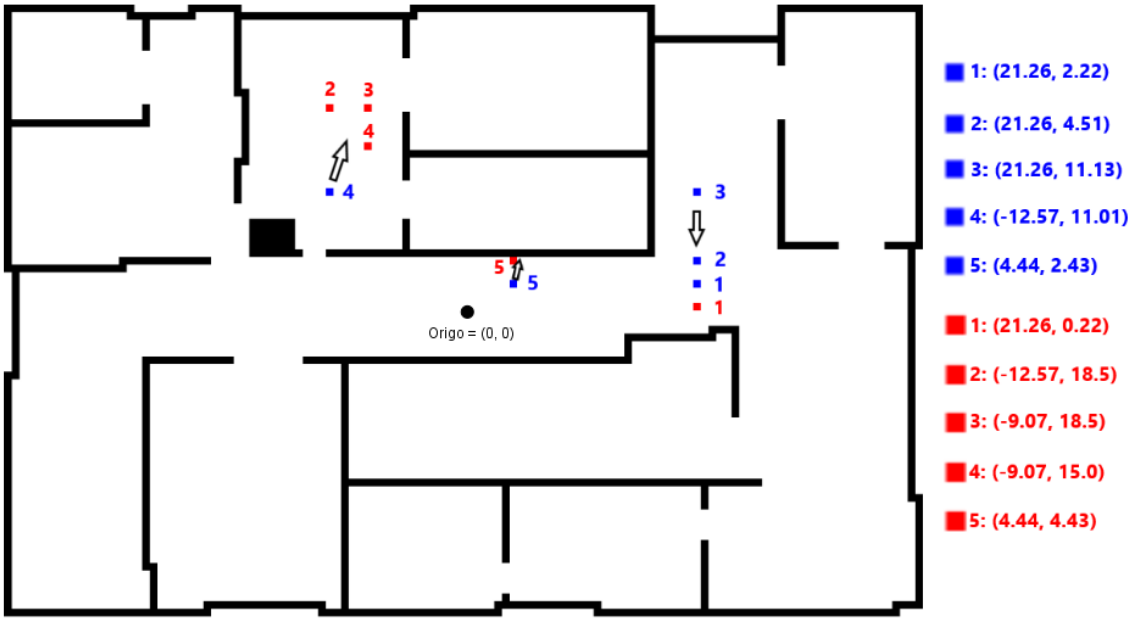

Figure 17: Test with the image analysis. Coordinates are in [cm].

Scenario **4** is a situation in which there are three marbles within the view of the camera. The implemented design does now allow more than one marble to be detected at the same time. Thus, the robot will switch between the three marbles and calculate the positions accordingly. In the test, the detected marble was number **2**.

The last scenario, **5**, demonstrated a problem that occurred, when the robot was facing a marble with an angle between the robot's orientation and the marble itself. Normally, the x-position should fit almost perfectly, as the distance to the marble is calculated along the y-axis in the different scenarios described in figure 17. The deviation of the y-value

will, however, change according to this distance, as the implemented method will result in errors according to the precision of the circle detection. There would have been errors on the x-axis if the orientation of the robot was from *left* to *right* or vice versa, and not from *up* to *down* or vice versa (as used in the tests).

	Estimated location (x, y)	**Deviation (x)**	**Deviation (y)**
Scenario 1	(21.28, 0.27)	0.02	0.05
Scenario 2	(21.28, 0.38)	0.02	0.16
Scenario 3	(21.28, 1.06)	0.02	0.84
Scenario 4	(-12.64, 18.78)	0.07	0.28
Scenario 5	(4.15, 4.35)	0.29	0.08

Table 4: The estimated locations and the matching deviations.

Figure 18 shows all five different scenarios as seen from the camera view:

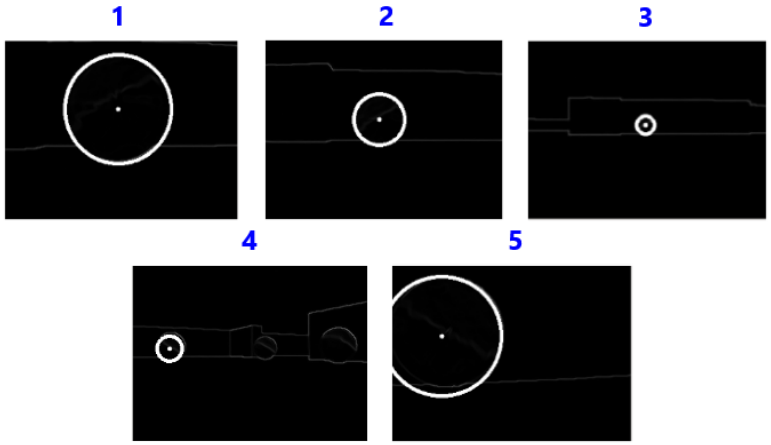

Figure 18: The marbles from the camera's perspective.

4.4 Conclusion

The expectation was to detect exact positions of the marbles, regardless of the robot's pose. The implemented method can determine an estimated position of a marble, if there is only one marble within the camera view. Most importantly, the orientation of the robot has to be exact towards the marble. Nonetheless, it does not matter what the yaw of the robot is, meaning that the marble can be detected anywhere in the environment.
The maximum distance from the robot to the marble is dependent on the `HoughCircles`'s ability to detect and outline the marble. Nevertheless, as the robot goes farther away from

the marble (within the range of which it can be detected), the error will become larger, as seen in scenario **3** in table 4.

If the robot turns its orientation away from the marble (the marble is still inside the camera view), the estimated position will be less accurate as well, as this parameter has not been taken care of.

This means that the robot can always detect a single marble anywhere in the environment. However, its ability to locate a marble requires that the orientation of the robot is directly towards the marble.

5 Path Planning

As the locations of the marbles are unknown, it is necessary to make sure the robot visits all rooms in the environment. Therefore, a path planning algorithm must be implemented to determine an efficient strategy to visit all rooms.

5.1 Requirements

The path planning algorithm must make sure the robot visits every room in the environment. A method which enables the robot to travel from point A to B, must be implemented. In addition, another method must be implemented with the purpose of setting goals for the first method to follow, thereby making sure the robot visits every possible room.

5.2 Wavefront Planner

In order for the robot to travel from point A to B, the Wavefront planner is implemented. The chosen environment is divided into a predefined number of cells. The cell of the start position is marked by a '0', and the end position is marked by a '2'. Every cell that contains an obstacle obtains a value of '1'. Using 4-adjacency, the cells to the left, right, above and under the end position, will obtain a value equal to '3'. 8-adjacency would include the four corner-cells as well. The cells containing the value of '3' have to be checked the same way. Up to four values (if 4-adjacency is chosen) around this cell will get the value: '4'. This continues until all cells have obtained a value. Afterwards, the path will be given by going from the start position towards the cell of lowest value, until the end position has been found. An example is given in figure 19 with 8-adjacency:

0	12	12	11	10	9	9	9	9	9
11	11	11	1	10	9	8	8	8	8
11	10	10	1	10	9	8	7	7	7
11	10	9	1	9	1	1	1	6	6
11	10	9	8	8	8	8	7	6	5
11	10	9	8	7	7	7	1	1	4
11	10	9	8	7	6	6	1	1	3
11	10	9	8	7	6	5	4	3	2

Figure 19: An example of the Wavefront Planner.[4]

The implemented Wavefront planner in this project has the 4-adjacency property, as this was most simple one to implement. This method has been implemented in a larger environment in the table below (see table 5), which is a representation of the *small world* with two rooms. The robot starts in '0' and drives along the red path until it reaches the end position marked by '2'. There is few paths to go from start to end, as it could also have

[4]Robotics, *Potential Fields* - 4th lecture, by Thomas Fridolin Iversen.

driven down directly after entering the small room. The *small world* has been mapped with larger walls to avoid possible errors of the robot driving into the walls.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	1	1	1	1	30	29	28	27	26	25	26	27	1	1	1
1	1	1	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	1	1	1	1	29	28	27	26	25	24	25	26	1	1	1
1	1	1	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	1	1	1	1	28	27	26	25	24	23	24	25	1	1	1
1	1	1	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	1	1	1	1	27	26	25	24	23	22	23	24	1	1	1
1	1	1	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	1	1	1	1	26	25	24	23	22	21	22	23	1	1	1
1	1	1	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	1	1	1	1	25	24	23	22	21	20	21	22	1	1	1
1	1	1	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	1	1	1	1	24	23	22	21	20	19	20	21	1	1	1
1	1	1	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	1	1	1	1	23	22	21	20	19	18	19	20	1	1	1
1	1	1	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	1	1	1	1	22	21	20	19	18	17	18	19	1	1	1
1	1	1	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	1	1	1	1	21	20	19	18	17	16	17	18	1	1	1
1	1	1	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	1	1	1	1	20	19	18	17	16	15	16	17	1	1	1
1	1	1	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	1	1	1	1	19	18	17	16	15	14	15	16	1	1	1
1	1	1	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	0	26	25	24	23	22	21	20	19	18	17	16	15	14	13	14	15	1	1	1
1	1	1	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	13	14	1	1	1
1	1	1	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	1	1	1	1	16	15	14	13	12	11	12	13	1	1	1
1	1	1	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	1	1	1	1	15	14	13	12	11	10	11	12	1	1	1
1	1	1	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	1	1	1	1	14	13	12	11	10	9	10	11	1	1	1
1	1	1	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	1	1	1	1	13	12	11	10	9	8	9	10	1	1	1
1	1	1	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	1	1	1	1	12	11	10	9	8	7	8	9	1	1	1
1	1	1	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	1	1	1	1	11	10	9	8	7	6	7	8	1	1	1
1	1	1	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	1	1	1	1	10	9	8	7	6	5	6	7	1	1	1
1	1	1	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	1	1	1	1	9	8	7	6	5	4	5	6	1	1	1
1	1	1	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	1	1	1	1	8	7	6	5	4	3	4	5	1	1	1
1	1	1	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	1	1	1	1	7	6	5	4	3	2	3	4	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5: An implementation of the Wavefront planner in the small world.

5.3 The Brushfire Algorithm and the GVD

A solution to visit all rooms is the Brushfire algorithm combined with the GVD (General Voronoi Diagram). The Brushfire algorithm is a method to map an entire environment with values; the same way as the Wavefront planner. The environment is divided into cells. Cells with obstacles contain a value of '1'. All other cells is determined from an obstacle and outwards, meaning cells next to obstacles obtain '2' and cell next to these obtain '3', and so on. An example is given in figure 20:

4	3	2	2	2	3	4	4	4	4
4	3	2	1	2	3	3	3	3	3
4	3	2	1	2	2	2	2	2	3
4	3	2	1	2	1	1	1	2	3
4	3	2	2	2	2	2	2	2	2
4	3	3	3	3	3	2	1	1	2
4	4	4	4	4	3	2	1	1	2
5	5	5	5	4	3	2	2	2	2

Figure 20: An example of the Brushfire algorithm.[5]

The General Voronoi Diagram (GVD) is a way to visualize possible paths after the implementation of the Brushfire algorithm. The GVD are marked in all cells that have equal

[5]Robotics, *Potential Fields* - 4th lecture, by Thomas Fridolin Iversen.

distance to walls/obstacles. Figure 21 shows an example of a GVD:

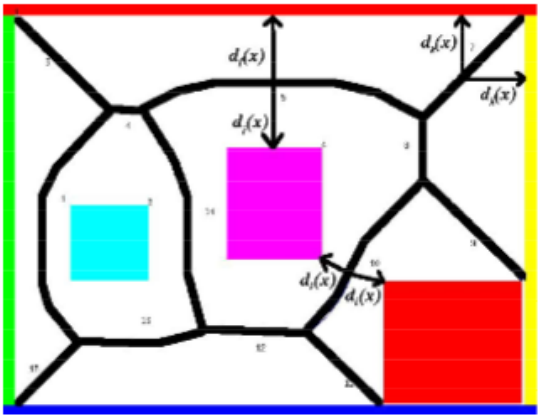

Figure 21: An example of the GVD.[6]

A combination of those two algorithms results in an efficient path planning. Figure 22 shows the result of the combination in the *big world*. The red and blue squares combined are the GVD. The blue cell in the middle of each sequence of blue cells is the point that will be used to go from room to room. The rest of the GVD is not relevant, and are not used for this implementation.

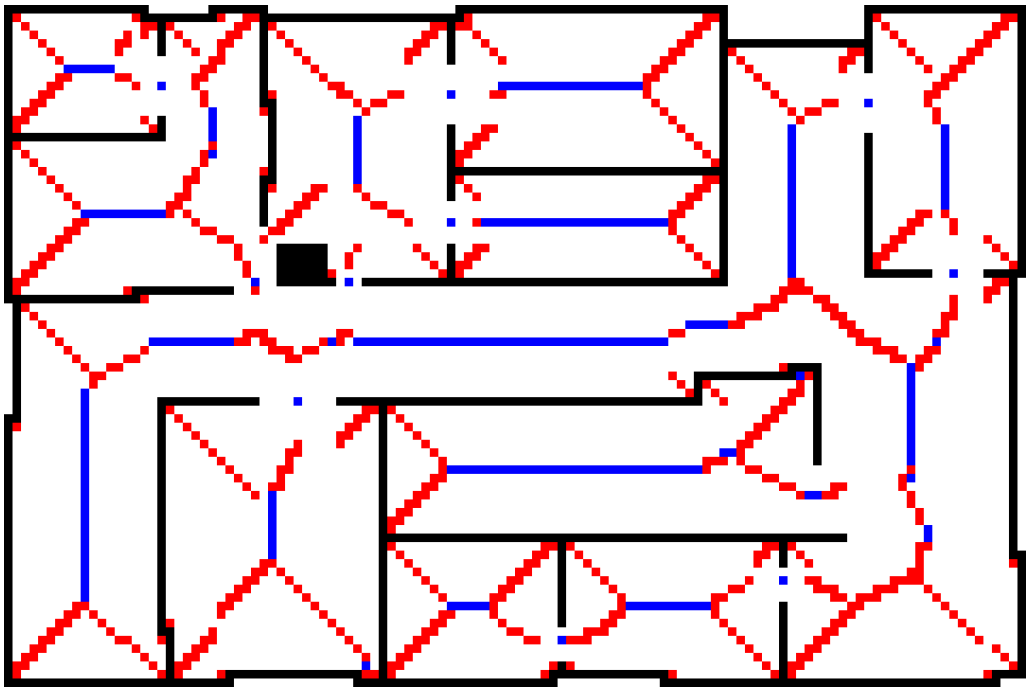

Figure 22: The Brushfire algorithm combined with the GVD.

[6]Robotics, *Roadmaps* - 4th lecture, by Thomas Fridolin Iversen.

5.4 Test of Design

The Wavefront planner is tested by setting a start and end position in the *small world*. Figure 23 shows the situation from section 5.2 with the same start and end position. It is tested with the fuzzy controller implemented to move between cells, described in section 2, which is the reason why the path is curvy. There is a few paths the robot could go, as seen in table 5. Nonetheless, the robot chooses this path, as a result of the fuzzy controller.

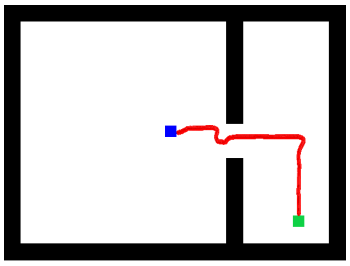

Figure 23: Test of the Wavefront planner using the fuzzy controller for navigation.

The object of this part of the project is to make sure that the robot visits all rooms in the *big world*. Therefore, the Wavefront planner should be tested together with the Brushfire algorithm and the GVD.

The left image of figure 24 shows the upper-left corner of the big world only with the important part of the GVD (the blue squares), refer to section 5.3. The grid in the other two images is bigger compared to the blue cells, just to show an example that can be seen. The object is to use the middle blue cell in each sequence of blue cells. Therefore, the Wavefront planner must be implemented for a start and end position, and make the path between these two, as seen in the middle image. The Wavefront planner must be recomputed whenever a new goal is set, as seen in the image to the right, which makes this solution suboptimal.

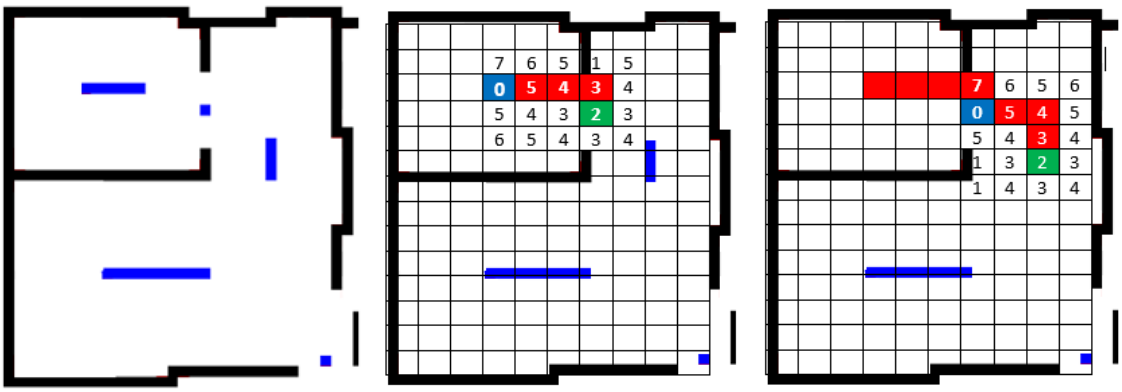

Figure 24: A simple implementation of the Wavefront planner combined with the Brushfire algorithm and the GVD.

5.5 Conclusion

The expectation of this part of the project was to determine an efficient strategy to visit all rooms in the *big world*. Some sort of path planning should be implemented to achieve the goal.

The path planning can determine the Brushfire and the GVD for an arbitrary environment with a predefined number of cells. The number of cells could be increased to obtain higher precision of the positions; however, that would increase the computational time. The GVD is not complete at certain locations in the environment (see figure 22), which could be caused by a small amount of cells. This is not a problem in the implemented solution, as the important part (the blue cells) are complete.

The combination of the Brushfire algorithm and the Wavefront planner is not the most efficient way to solve this problem. It will visit all the rooms, but the computation time will be unnecessarily large, as the Wavefront planner has to be recomputed for each end position. A better solution could have been to implement the A-star algorithm.

6 Localisation of the Robot

In the previous section, it was assumed that the robot's position was already known in advance, as Gazebo has a built-in function that determines this position. Assuming the built-in function did not exist, it would be necessary to implement a method to localise the robot in a given environment.

6.1 Requirements

The robot has to be localised in the environment, *big world*, with the assumption that the initial pose is known.
No matter where the robot is located in the environment, the implemented algorithm has to be able to localise it. The position and the orientation of the robot should not intervene with the efficiency of the algorithm. From the curriculum of Robotics, two major algorithms was presented; the Kalman Filter and the Particle Filter.

6.2 Particle Filter

The Particle Filter seemed most simple to implement, and therefore, this was chosen.
The Particle Filter is a method used to determine the robot's global position within a known environment. The algorithm uses 'particles' to locate the robot at an exact moment. Originally, the particles have to be normally distributed around the environment, but due to the fact that a known initial pose can be assumed, all particles should have their initial position at the robot's initial position.

Every particle is constructed as a `struct`, which contains the `id`, `position`, `yaw`, `weight` and the `lidar-data`. The orientation of a particle will be uniformly distributed from $-\pi$ to π, as the initial orientation of the robot is unknown. The `weight` of a particle is a measurement of the probability that the pose of the particle is equal to the robot's pose. The `lidar-data` is the distance from a specific particle to an obstacle for each lidar-sensor. As the lidar-sensors only cover a certain area in proportion to the orientation, that is from $-130°$ to $130°$, there will be an empty area of $100°$ behind the robot/particle. This is shown below in figure 25. The `weight` of the a particle is calculated by comparing the `lidar-data` of the particle with the `lidar-data` from Gazebo. However, a particle can be at the exact same position as the robot, but its orientation can be different than the robot's orientation.

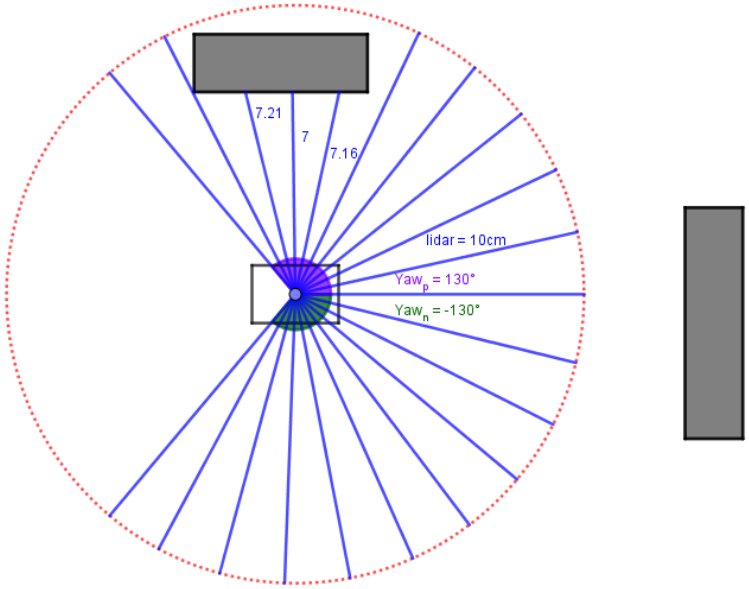

Figure 25: An example of how the lidar-sensors are represented in an environment.

This figure represents how the lidar-sensors interact with an environment. The maximum distance a lidar-sensor can reach is equal to 10cm. In this example, every lidar-sensor will return a distance of 10cm except three that are equal to 7.21, 7, and 7.16cm.
All lidar-data is saved to the `struct` of every particle according to their position and orientation.

The implemented algorithm is based on *Algorithm 17* shown in figure 26. The weight is updated according to the following general normal distribution:[8]

$$P(y) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot \exp\left(\frac{-(y-d)^2}{2 \cdot \sigma^2}\right)$$

Where y is the lidar-data calculated for the actual robot, and d is the true distance to an obstacle for the particles. Afterwards, the weight is normalised from 0 to 1.

Algorithm 17 Probabilistic localization using a particle filter

Input: Sequence of measurements $y(1:k)$ and movements $u(0:k-1)$ and set $\mathcal{M}$ of N samples (x_j, ω_j) corresponding to the initial belief $P(x)$

Output: A posterior $P(x(k) \mid u(0:k-1), y(1:k))$ about the configuration of the robot at time step k represented by $\mathcal{M}$.

1: **for** $i \leftarrow 1$ to k **do**
2: **for** $j \leftarrow 1$ to N **do**
3: compute a new state x by sampling according to $P(x \mid u(i-1), x_j)$.
4: $x_j \leftarrow x$
5: **end for**
6: $\eta \leftarrow 0$
7: **for** $j \leftarrow 1$ to N **do**
8: $w_j = P(y(i) \mid x_j)$
9: $\eta = \eta + w_j$
10: **end for**
11: **for** $j \leftarrow 1$ to N **do**
12: $w_j = \eta^{-1} \cdot w_j$
13: **end for**
14: $\mathcal{M} = resample(\mathcal{M})$
15: **end for**

Figure 26: Algorithm 17: Particle Filter.[7]

[7] Horset, H. and more.: *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Page 317. 1. edition. MIT Press, 2005.

[8] Normal Distribution. Published by Wolfram. Web address: http://mathworld.wolfram.com/-NormalDistribution.html - Visited: 2019.12.05.

After each measurement of the localisation of the robot, all particles have to be resampled. The implemented algorithm for resampling is based on *Algorithm 18* shown in figure 27. The new particle is pushed back onto a new vector of particles. This vector is returned and used for the next measurement in *Algorithm 17*.

Algorithm 18 The procedure *resample*($\mathcal{M}$)
Input: Set $\mathcal{M}$ of N samples
Output: Set $\mathcal{M}'$ of N samples obtained by importance resampling from $\mathcal{M}$

1: $\mathcal{M}' \leftarrow \emptyset$
2: $\Delta \leftarrow rand((0; N^{-1}])$
3: $c \leftarrow \omega_0$
4: $i \leftarrow 0$
5: **for** $j \leftarrow 0$ to $N-1$ **do**
6: $u \leftarrow \Delta + j \cdot N^{-1}$
7: **while** $u > c$ **do**
8: $i \leftarrow i+1$
9: $c \leftarrow c + \omega_i$
10: **end while**
11: $\mathcal{M}' \leftarrow \mathcal{M}' \cup \{(x_i, N^{-1})\}$
12: **end for**

Figure 27: Algorithm 18: Resampling of particles.[9]

6.3 Test of Design

The Particle Filter can be tested in different ways. It has been decided to test a specific path for the robot to go from and to. The actual path will be compared to the estimated path; including a comparison of the difference in x- and y-values. This test is executed with 100 particles, 21 (of maximum 200) lidar-sensors, and a measurement of the position every 0.1cm. The robot moves in a total of 50 cells equal to 50cm, which equals to 500 measurements. Therefore, the particles are resampled 500 times.

Increasing the amount of resamples will increase the precision of the localisation process as well as the computation time. All the lidar-data for every possible point in the environment has not been calculated using a look-up table, but done during the computation. Therefore, the amount of particles has to be low and the number of sensors to be low as well; less lidar-sensors allow more particles. It is assumed that 21 sensors is enough to detect the environment's shape due to few detailed objects. All of these parameters could have been increased, but would slow down the computation time quite much.
Figure 28 shows the test of the robot driving from (0, 0) to (30, -25). The actual path is represented by the red line and the estimated path by the blue line.

[9] Horset, H. and more.: *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Page 318. 1. edition. MIT Press, 2005.

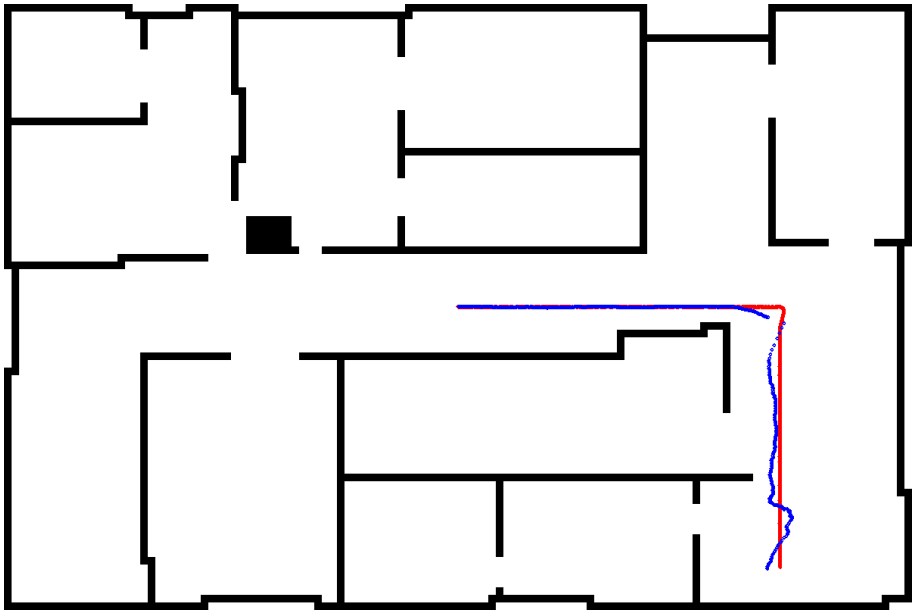

Figure 28: The actual path vs. the predicted path of the robot in the environment.

The estimated path follows the actual path almost perfectly during the first half. Actually, the particles moves in two directions at first (for a very short distance). That is to the left and to the right, but as the robot moves further to the right, its sensors will obtain different values as a result of a different set of lidar-data that corresponds to where the robot is. However, when the robot drives down, it enters a wide corridor that is harder do determine for the robot. Therefore, the estimated path down the corridor results in miscalculations.

Figure 29 shows how accurate the implemented algorithm is. The graph shows that the estimated location follows the actual position with a difference in x equal to ca. 0.5 pixels, and the difference in y equal to ca. 0.05 pixels. The conversion from cm to pixels is: $pixel = 1.41735 \cdot \text{cm}$. When the robot reaches the turning point, the error increases rapidly, especially on the y-axis. Up to 4 pixels in error are obtained.
The average error is:

$$x_{error} \approx 0.58\text{pixels} \qquad y_{error} \approx 1.08\text{pixels}$$

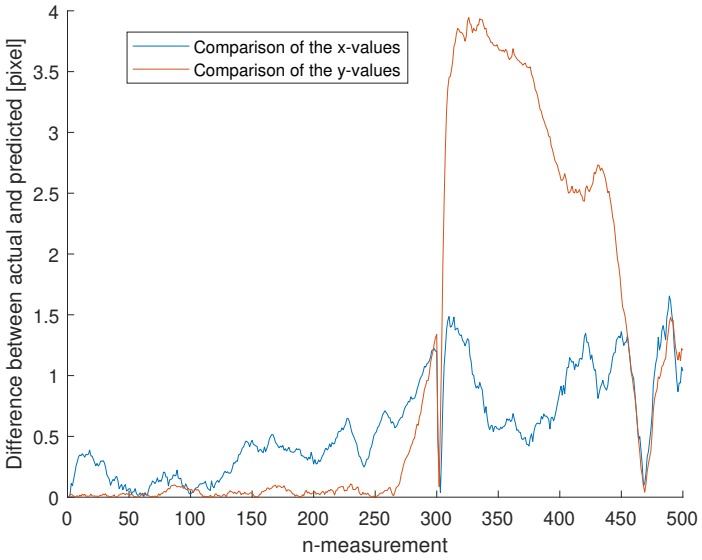

Figure 29: Comparison of difference between x- and y-values for the actual and estimated position.

6.4 Conclusion

The robot was expected to localise itself with an initial known pose. The performance of the implemented algorithm depends on where the robot is. If it is located in a room, which is unique compared to other rooms, it will most likely find itself quickly, as seen in the long hall in figure 28. As it drives down the corridor, the particles will disperse, but the average of the particle's positions will be near the robot.

The algorithm would probably have been more efficient, if a look-up table was created to speed up the computation time, as the all the data from the lidar-sensors would have been known beforehand. Using a look-up table would have allowed an increase of the amount of particles, lidar-sensors, and measurements.

Experiments with more particles were conducted, however a measurement was only executed every 1cm. The more often a measurement is carried out, the more precise the estimation of the position will be; this was clear after the change from every 1cm to every 0.1cm. However, the number of measurements should have been even larger, thus updating as fast as the program can handle, if it could be executed at an acceptable computation time.