University of Southern Denmark

Marble Finding Robot

Christian Eberhardt
chebe17@student.sdu.dk

Jakob Grøftehauge
jakra17@student.sdu.dk

Handover Date: 08-12-2019

Contents

1 Introduction

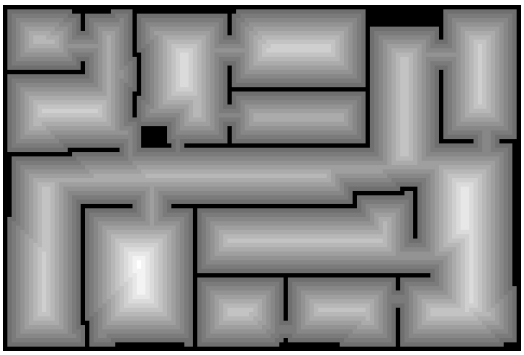
In this section various methods will described which has been used to produce material that is needed for the various higher level algorithms described later in this report. Source and test code can be found at: https://github.com/chrillemanden/rb-rca5.

1.1 Generating Waypoints

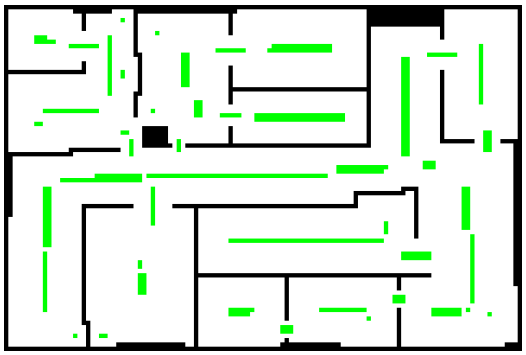
When using path finding algorithms or employing local obstacle avoidance in the simulated world it is necessary to have waypoints. In both cases the waypoints need to be generated so that there is always at least one path to every room where paths must consist of a sequence of waypoints that does not lead the robot through obstacles. The number of waypoints should be kept to a minimum. After all every obstacle-free pixel in the original map could be used as a waypoint, however this would result in many unnecessary computations for the path finding algorithm. Additionally it is desired to locate waypoints within entrances to rooms as these are characterised by narrow passages.

1.1.1 Brushfire Algorithm

To generate waypoints, first the brushfire algorithm is used to generate a map of distances to nearest obstacles for every pixel in the map. Distance is determined using eighth point connectivity. The results can be seen in figure 1a.



(a) Gradient map after using the brushfire algorithm with eighth point connectivity. The black pixels mark the obstacles while the graytones mark the distances to the nearest obstacle. A more white tone indicates a greater distance to the nearest obstacle.



(b) Map showing local maxima (green pixels) where distance to obstacle is greatest. This image is generated using non-maxima suppression on the gradient map with a kernel-size of dimension 3x3 pixels.

Figure 1: The two maps show preliminary steps towards finding waypoints.

1.1.2 Non-maxima Suppression and Point Reduction

Non-maxima suppression is used to find local maxima of the gradient map. This is done by running a 3x3 pixels kernel over every pixel in the map and only saving the pixel if none of the eight pixels around it has a higher pixel value. This yields the map shown in figure 1b.

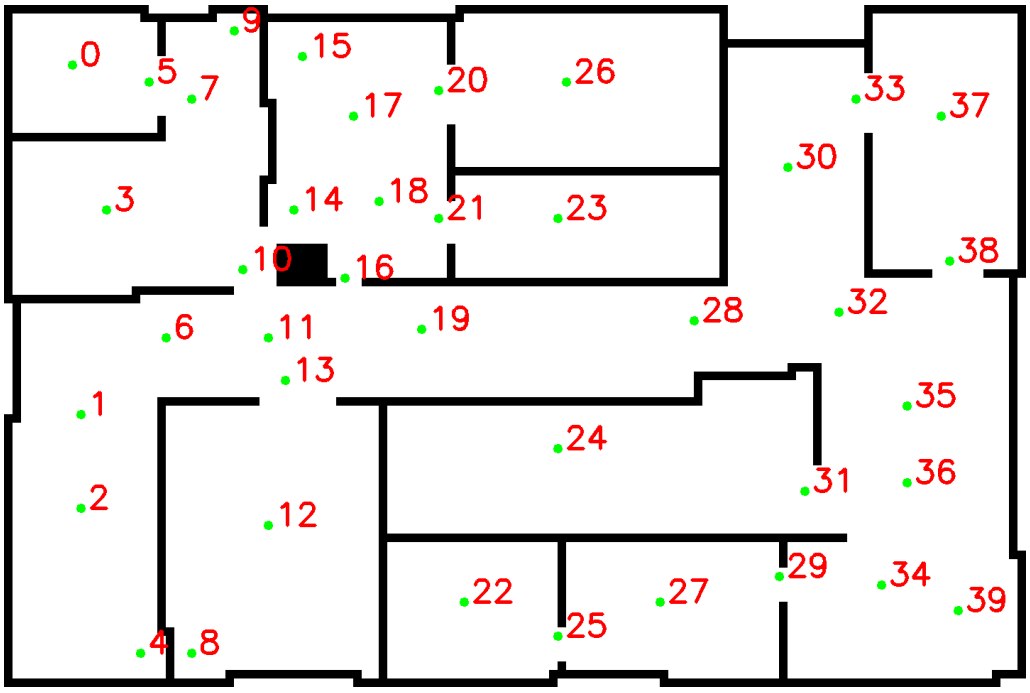


Figure 2: Generated waypoints in the map with indexes. Green dots are the waypoints and red text is numbering of the waypoints.

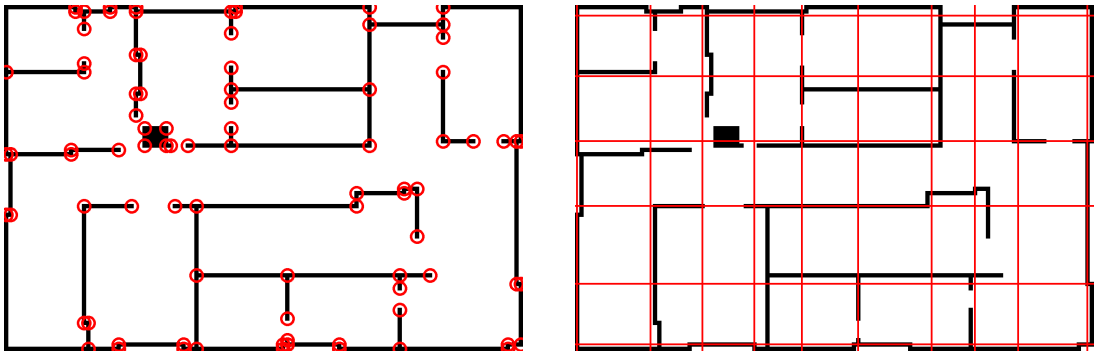
Hereafter an iterative algorithm is run that groups the local maxima into waypoints. For every local maxima neighbours within a distance are found and a mean pixel is found between these close local maxima. The algorithm stops when the waypoints cannot be reduces further. This yields the waypoints shown in figure 2.
As can be seen in the figure, waypoints end up being placed in every entrance to a room, although some are not completely centered in the entrance due to the rounding down of integers resulting in waypoints pulling left or up in the map, as the upper left corner in the map has index (0, 0). However as can also be seen there are some redundant waypoints, namely waypoints 4, 8, 9, 15 and 39, that result from local minima close to features in rooms, as most rooms in the map are not completely rectangular but have bends and kinks. One way of mitigating this could be to use a 5x5 pixels kernel when doing non-maxima suppression.

1.2 Gridification of the map

Another way to group pixels implemented in this project is done by dividing the map by a grid.

1.2.1 Black and white corner detection

To construct the grid, all the corners of obstacles are found. Corners are found by looking at each individual pixel and looking at the eight neighbors around that pixel. In its most simple form the algorithm used for finding corners just compares opposite pixels that are not diagonal to the pixel. If these pixels do not have the same value, it is assumed the pixel is a corner. The result of running this algorithm can be seen in figure 3a.



(a) In this map corners of obstacles are highlighted by red circles.

(b) In this map the grid is highlighted by the red lines.

Figure 3: Gridification of the map.

1.2.2 *k*-Means clustering

When the corners have been found, they are processed by the k-Means clustering algorithm, introduced in the CV course, to group the corner-points along the x-axis and the y-axis in the map separately. The found clusters are then used to draw the grid lines. The result of this can be seen in figure 3b. It has proven to be sufficient to use 10 bins in the width and 6 bins in the height of the map, yielding a 9x5 grid.

1.2.3 Bug algorithm

For implementing Q-learning it is necessary to know the available actions in each grid square. These have been found by using a Bug-like algorithm. From the midpoint in a grid square, the algorithm tries to find a valid way to the midpoint in one of the adjacent grid squares while being bounded by grid lines. The results of running the algorithm and the valid grid actions are visualised in figure 4.

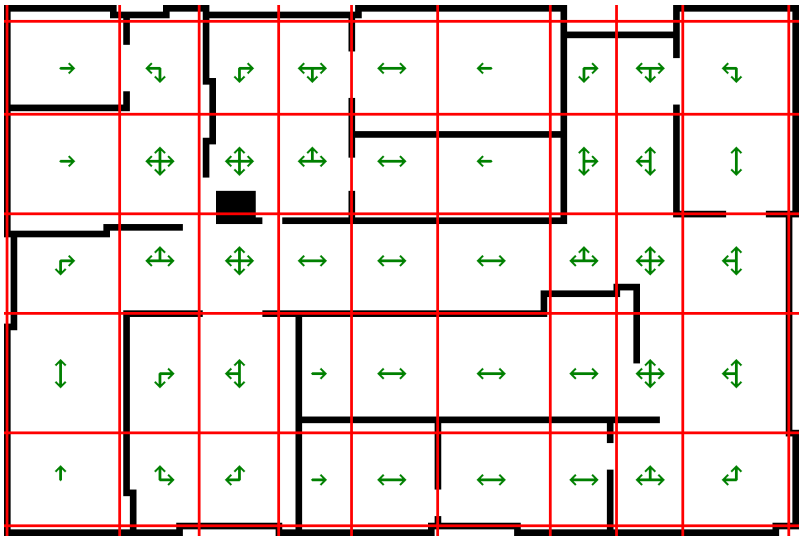


Figure 4: Map with grid and the possible actions in every grid square highlighted with green.

2 Fuzzy Logic Control

The concept of Fuzzy Control Logic uses the same approach as a human expert would use to control a given system. A fuzzy controller consist of 3 main components: fuzzification, rule-base and defuzzification. The first design process of a fuzzy controller is to have an human expert describe, in linguistic terms, how he would control the process. This also involves describing which parameters he uses when operating the system. This linguistic description of the control process, will later form basis for the rule-base. Next the method for converting between continuous input values and the linguistic terms has to be found. For this a set membership function is defined, one for each input variable. The membership function describes to what degree a certain value belong to a certain linguistic term. A way of mapping between the linguistic terms and the output variables also has to be found. For this another set of membership functions are found. A block diagram of the designed fuzzy controller can be seen figure 5. The controller is designed to only navigate between waypoints, higher level algorithms is responsible for finding the waypoints and the optimal path between the starting point and the end point as described in section 1.1 and section 6 respectively.

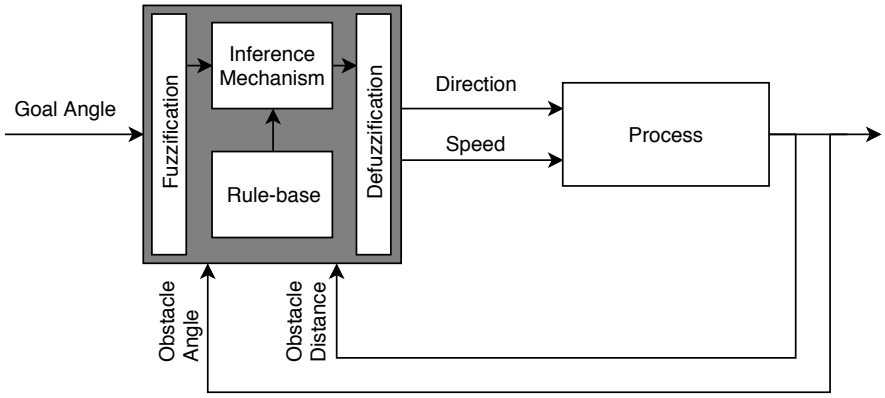


Figure 5: Block diagram of the designed Fuzzy controller.

2.1 Fuzzification and Defuzzification

As seen in figure 5 the reference input to the controller is the goal angle. The goal angle represents the angle the robot needs to turn in order to point directly towards the next waypoint. Since this angle changes as the robot moves, logic for continuously updating the value has been implemented, outside the controller. This logic is also responsible for setting a new waypoint when the robot is nearing the current waypoint. The goal angle is a crisp value, why a membership function for fuzzyfying has been defined. The membership function is defined as seen in figure 6b.
The feedback loop of the controller consists of the distance to the nearest obstacle, and the angle the nearest object seen from the moving direction of the robot. The membership functions used for these variables can be seen in figure 6c and figure 6a.
The controller outputs two values. One for controlling the speed of the robot, and an other one for controlling the angular velocity. The membership functions used for defuzzification can be seen in figure 7. Both variables is deffuzzified using the term "Centroid 100". The official FuzzyLite Language documentation [1] do not supply explanation of the defuzzification terms, why it not

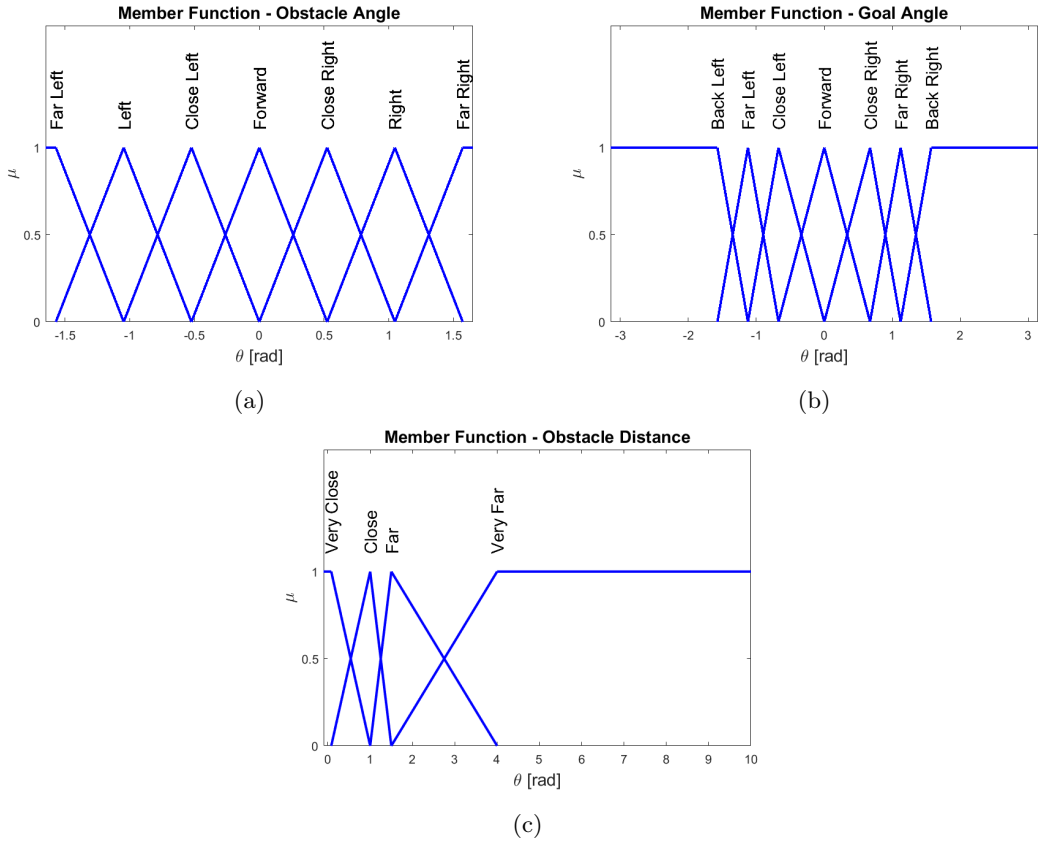


Figure 6: Membership functions for the fuzzification

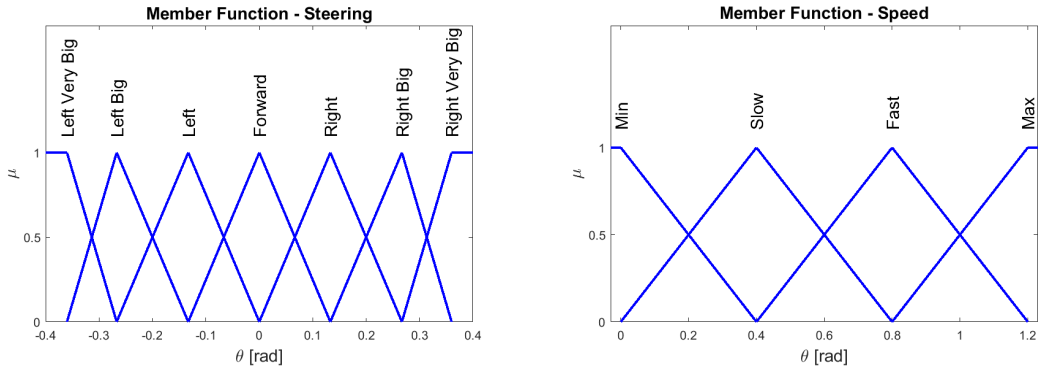
possible to determine which method it is implementing. Though it is expected to defuzzify using a method similar to the "Center of gravity" method. Testing different defuzzyfication methods has not been a scope of this project, why it is difficult to tell how the method influence the performance of the controller.

2.2 Rule-Base and Inference Mechanism

The Rule-Base and Inference Mechanism for this project is using the Mamdani/system fuzzy system. An example of the form of a rule in the rule-base is: ***if*** *obstacle distance is veryClose* ***and*** *obstacle angle is farLeft* ***and*** *goal angle is backRight* ***then*** *mSteer is rightVeryBig*. A rule for every combination of the linguistic term from the three membership functions for the three input variables, has been created. This was done to ensure, that every combination of input variable terms are handled, so the controller acts as intended.
The interference mechanism is setup to use the minimum function for conjunctions and implications and the maximum function for disjunctions. No further thought has been put into picking these function, why it will not be covered further in this project.

2.3 Tests with obstacle avoidance

To test the obstacle avoidance, the robot is set to follow a sequence of waypoints. The algorithm is then valuated according to whether it is successful in reaching the final waypoint in the sequence.



(a) (b)

Figure 7: Membership functions for the defuzzification

Subfigure	Sequence of waypoints
8a	13, 11, 10, 7, 5, 0, 5, 7, 3.
8b	19, 11, 6, 1, 2, 4, 2.
8c	19, 16, 14, 10, 11.
8d	28, 32, 30, 33, 37, 38, 35.
8e	13, 11, 10, 14, 17, 20, 26, 20, 17, 18, 21, 23.
8f	32, 35, 34, 29, 27, 25, 22, 25, 27.
9a	13, 11, 10, 3, 7, 5, 0.
9b	28, 30, 33, 37, 38, 32.

Table 1: The sequence of waypoints used for generating the trajectories seen in figure 8 and 9. The location of the waypoints in the map can be seen in figure 2.

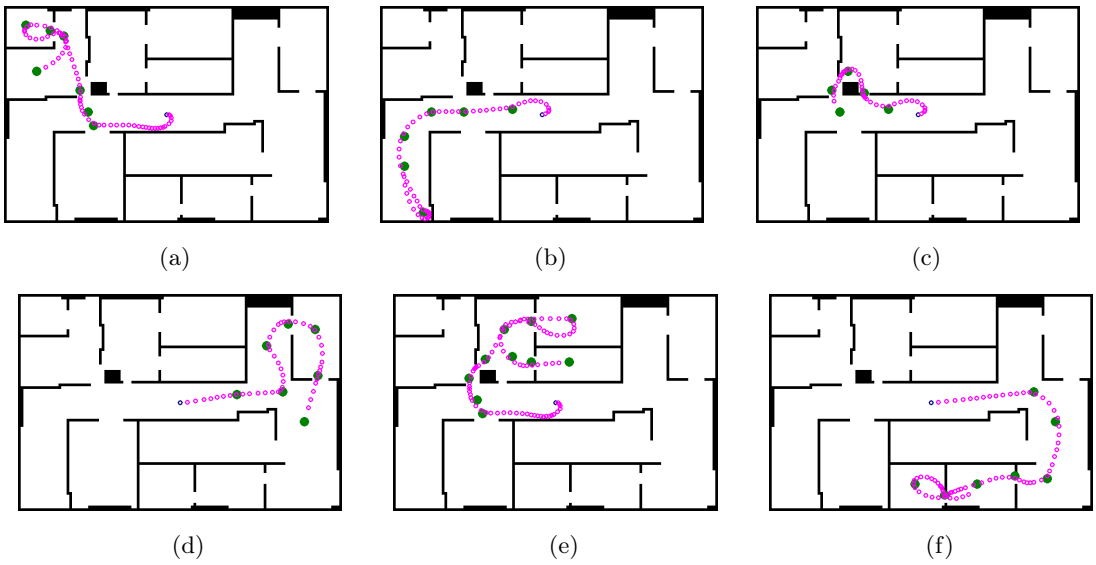
The sequences of waypoints used can be found in table 1. These sequences are visualised in figure 8 that show, that for at least these sequences of waypoints the robot is successful in avoiding obstacles and finding the final waypoint. As can also be seen from the different scenarios, the robot is able to navigate through narrow doorways as well as turn around when a waypoint is located in the opposite direction of the robots current heading.

2.4 Conclusion

Fuzzy logic control has been implemented. With this implementation it is possible to use the abstraction for the robots differential drive to control the robot. Additionally the fuzzy controller is used for local obstacle avoidance successfully avoiding obstacles between navigable waypoints.

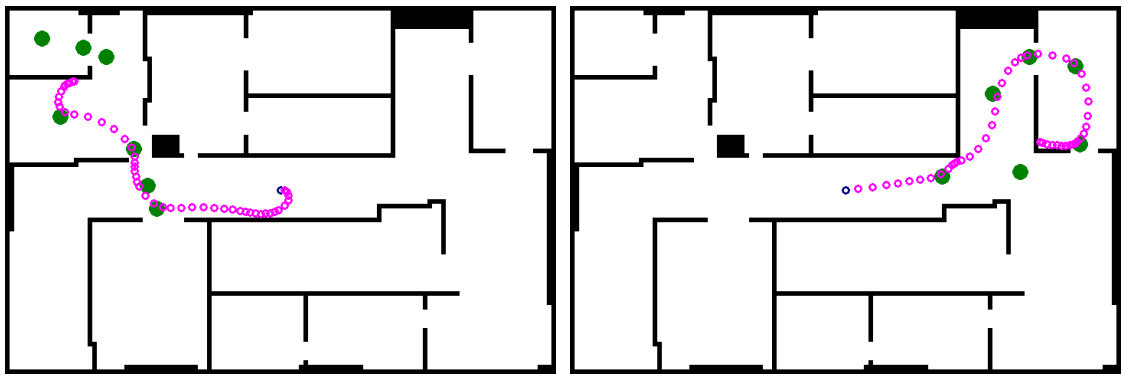
3 Computer Vision

It has been decided to only use the camera input for detection of marbles in the environment. "Determine the location of the marbles in the environment" according to the project report [2], will not be addressed in this report. The focus has instead been on detection if a marble is within the view of the robots camera and estimate its position in the camera feed.



(a) (b) (c) (d) (e) (f)

Figure 8: Different trajectories generated by using different sequences of waypoints described in table 1. The purple circles show the real position of the robot while the green circles mark the waypoints that the robot follow. In all these cases the robot was successful in following the sequence of waypoints avoiding obstacles along the way.



(a) (b)

Figure 9: Some trajectories generated by using the sequence of waypoints described in table 1. The purple circles show the real position of the robot while the green circles mark the waypoints that the robot follow. In these cases the robot was unsuccessful in following the sequence of waypoints resulting in the robot crashing into obstacles.

3.1 Pre-processing Algorithm

The raw camera input received from the gazebo environment appears overlaid with noise. The noise seems to resemble a Gaussian noise model. The camera input is also very dim, which makes it difficult for the human eye to easily distinct different contours from each other. In order to ease the detection of marbles it has been decided to develop a pre-processing algorithm for the raw camera input. The purpose of the algorithm is to reduce the implication of the noise in later algorithms and brighten the image to enhance the contours in the image.
The flow of the algorithm is visualized in the flow chart in figure 10. As seen on the figure the images is first brightened. This is done by converting the image into the HLS colour space. The channels are then split up. The luminance channels is eqaulised using the built-in OpenCV [3] function *equalizeHist()*. The channels are merged back together and afterwards the image is again converted back to the BGR color space. The result of this operation can been seen in figure 11b. It clear to see that the image appears significantly brighter, though it should also be noted that the amount of noise has increased considerable as well.
The image is filtered using Gaussian smoothing filter. A Gaussian smoothing filter works by calculating a weighted average for every pixel. The weighted average takes into account the neighbouring pixels, as well as there distance to the pixel the weighted average is calculated for. For this purpose a Gaussian kernels containing the weights is defined. The weights in the kernels is characterised by being distributed with a Gaussian distribution, where the distance to the center pixel defines the magnitude of the weight. The kernels is normalized, which means all weights sums to 1. In the CV pipeline the function, built-in to OpenCV, *GaussianBlur*, is used for applying the Gaussian smoothing filter to the image. A kernel size of 9x9. The kernels has a standard deviation of 2 in both the X and Y direction. For choosing the filter type, only the Gaussian filter and the median filter found in the OpenCV library was considered. The filters was tested on a sample from the camera feed. The Gaussian smoothing filter appeared to performed better than the competitor, why it was chosen.

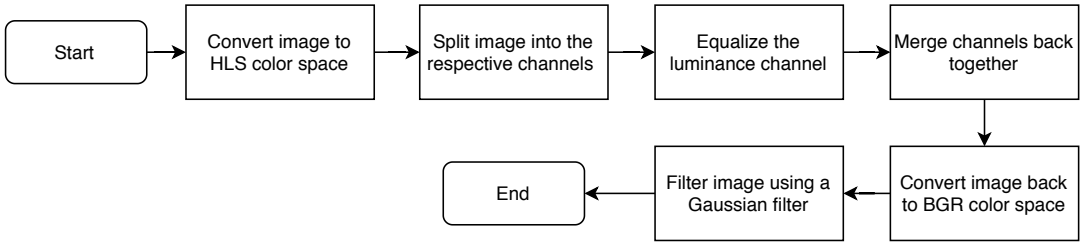
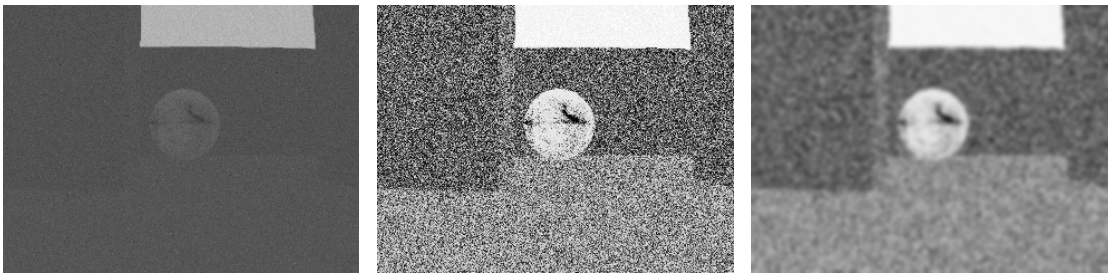


Figure 10: Flow chart of the pre-processing algorithm

3.2 Detection of Marbles

It was chosen to use two methods for detection of marbels. The use of two individual algorithm for detection marbels was chosen to increase the robustness of the final prediction. The first method works by detection contours in a binary image of the camera feed, and the other one uses an OpenCV function, *houghCircles()* to find round objects in camera feed.



(a) Raw camera input (b) Equalized camera input (c) Filtered output

Figure 11: Visualization of the effect of different steps in the pre-processing algorithm

3.2.1 Detection Using Contour Detection

The flow of the algorithm is visualized on the flowchart in figure 12. First step to extract contours from an BGR-image, is to convert it into a binary image. To do this the function *adaptiveThreshold()* has been utilized. The function considers the difference between the current pixel and a local average of block of pixels around the current pixel. If the current pixel value is greater than the local average minus an offset then the current pixel position in the binary image is set to 255, otherwise 0. The block size used is 101 times 101. The offset used is 1. Then the image is eroded 3 times and dilated 8 times in order to enhance the contours and connect contours which has falsely been separated. For the erosion operation a kernels size of 5x5 is used and for the dilation operation a 3x3 kernel is used. To determine the applied number of erosions and dilations and the associated kernel size, trial and error has been used to estimate the optimal number.
The images is now at the stage where it is possible to extract contours from the image. For this the OpenCV function *findContours()* is used. The function return a vector, which contains vectors containing points defining the different contours. Next each contour is tested to see if the contour could origin from a marble or it is of no relevance. First small contours with a area smaller than 50 pixels is rejected, for computing the area the OpenCV function *contourArea()* has been utilized. Afterwards a rotated bounding box (BB) is fitted to contour. The area of the BB is obtained and a difference between the area of the contour and the BB is calculated.
If the formula for calculating the area of a square and a circle is compared, it can be seen that the biggest circle that can be fitted within the perimeter of the square will occupy $\frac{\pi}{4}$ of the square's area. This information has been used to reject contours which do not comply with this characteristic. It is known that the found contours of marbles will not necessarily be perfect round, due to the adaptiveThreshold algorithm not being perfect as well as the erosion and dialation operations could modify the shape slightly. Therefor a margin of error of 10 percent has been allowed when comparing the two areas.
All the remaining contours, which has not been rejected, is then draw into a binary image. An example of the output of this sub-algorithm can be seen in figure 13b, where figure 13f, without the BB, is the initial image.

3.2.2 Detection Using Hough-Tranform

For this algorithm the OpenCV implementation of hough transform for circles is used. The algorithm works by converting the input BGR-image into a binary image using the canny edge detector

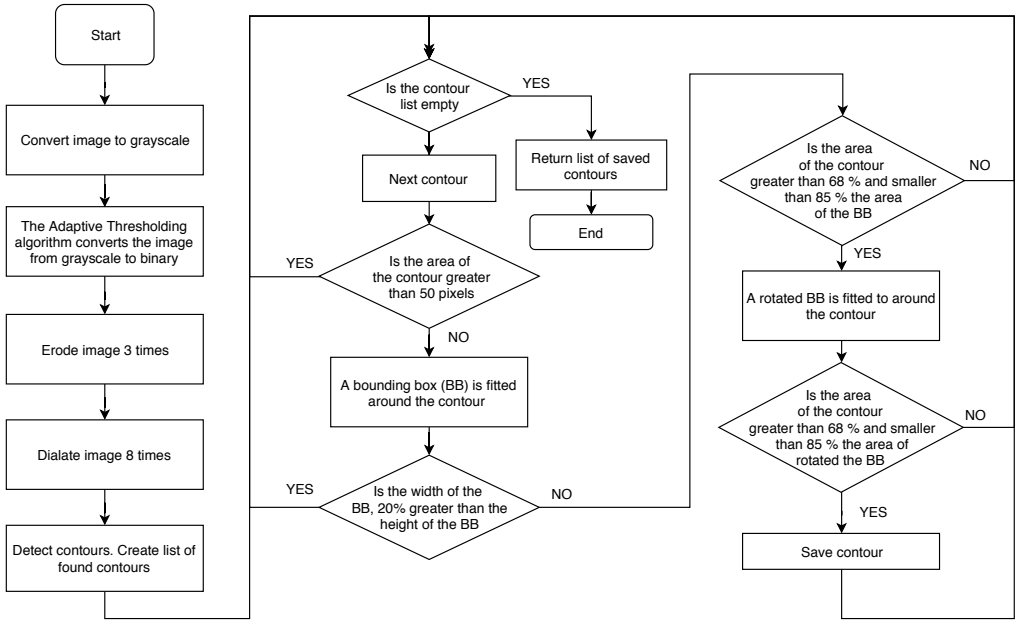


Figure 12: Flow chart of the detection algorithm using a binary image

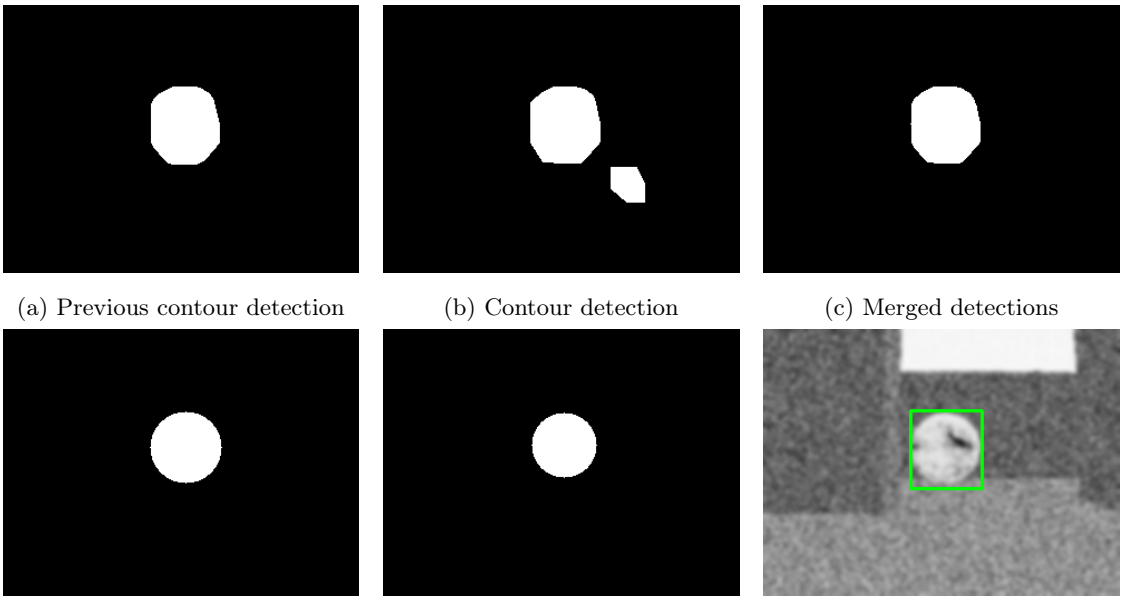
algorithm. When using the circle hough transform the threshold of the canny detector needs to be specified, in this case an upper threshold of 170 is used, the lower threshold is automatically set twice smaller. The threshold was found by experimenting with the *canny()* function to determine a suitable upper threshold where the boundary of the marble appeared well defined, and the image did not contain to much noise.

The algorithm then iterate through each edge pixel found by the *canny()* function. The algorithm then finds every possible circle whose perimeter intersects the pixel. A 3-dimensional matrix is then created. The matrix contains possible parameter combinations describing circles which could possible be found in the image. Every time a possible circles has been found, the circles is "up voted" in the matrix. When the algorithm has iterated through every edge pixel, circles above a certain threshold is accepted as a detection of a circles. In this case the accumulated threshold was picked to 24, meaning the perimeter of the circle has to intersect at least 24 edge pixels. Afterwards non-maxima suppression is used to eliminate circles who is almost identical.

An example of the output of this algorithm can be seen in figure 13e, where figure 13f, without the BB, is the initial image. The circle is filled due later steps in the algorithm as well as eliminating circles within other circles.

3.2.3 Combinations Detection Algorithms

In order to have only one prediction for each image, the individually predictions from the two methods has to be combined. It was decided to convert the information from the hough circle transform into a binary image with the contour of the found circles, in order to ease the comparison between the outputs of the two algorithms. For merging the contours in the two images the function *bitwise_and()* function was used. The function perform the AND operation on each individual pixel pair. Though this creates problems if one of the algorithms fails the detection, as no contour then can be detected. Therefore it was decided to implement a form of filter, where the previous



(a) Previous contour detection (b) Contour detection (c) Merged detections

(d) Previous Hough detection (e) Hough detection (f) Found Bounding box

Figure 13: Merging of marbles detection from the two algorithms

detections from each algorithm is utilised, as it is assumed that the position of a marble do not change significantly from frame to frame. The merging of the 4 input is done by performing the AND operation on all combination of image pairs, and then do the OR operation all the resulting images. An example of the input and output of the merging operation can be seen in figure 13. It is worth noting that a false positive is detected by the Contour Detection algorith, in figure 13b, though it is rejected due to only being present in one of the frames used to determine the final output.

3.3 Test of Marble Detection

The performance of computer vision algorithm is tested on a video sequence consisting of 150 frames. The video is captured in the small-world environment. The developed algorithm is then performed on each individual image, and then each prediction is manually inspected to count the total number of false negatives (FN), false positives (FP), true positives (TP) and true negatives (TN). The following results was obtained:

$$TP = 98$$
$$TN = 41$$
$$FP = 0$$
$$FN = 11$$

From the results the accuracy of the algorithm can be calculated:

$$Accuracy = \frac{TP + TN}{Total\ Samples} = \frac{41 + 88}{150} = 0.927 \quad (1)$$

as seen in equation 1 the accuracy of the algorithm is around 92 percent. When evaluating the bounding boxes the focus has been on if the marble is detected or not. The ground truth placement and size of the bounding box just covering the marble is not known due to lack of time and an annotation-tool. Therefore a bounding boxes is seen as true positive, if it covers most of the visible part of the marble. Though it has been regarded as a false positive if a bit of the marble is still visible in the camera feed, and no bounding box is present. The video with predicted bounding boxes can be found on the github page in the folder vision_dev.

3.4 Conclusion

A computer vision method to detect marbles has been been implemented. If more time had been available the methods should have been tested under different lightning conditions. Therefore it is also difficult to conclude on the performance of the algorithm, as the test is rather limited in exposing the robot and the marbles to different lightning conditions.

4 Q-learning

Initially it was decided to develop an environment, where an agent could be trained to find the optimal path to visit all the rooms. The environment is prepared by applying a grid to the occlusion map as described in section 1.2 and finding valid discrete actions from each grid cell. However to solve this particular problem, it is necessary to keep track of which cells are already visited in order to maintain the Markov property. One single cell can then have a lot of different states depending on which combination of cells has already been visited. Therefore, the number of states grow exponentially with the number of grid cells which makes it impractical to solve with the available computing power. By implementing a way of dynamically allocating states the computing power needed to solve the problem could have been reduced, however it was decided to downscale the environment, to illustrate the principles of Q-learning, and the influence of hyperparameters.

4.1 Test of Q-learning Algortihm

4.1.1 Setup of Test Environment

A overview of the test environment can be seen in figure 14, the occlusion map is divided into the grids as seen in figure 4. The agent will always start where the start sign is located. A key and a treasure chest is located in the environment. When the agent finds the treasure chest the current episode terminates. If the key has not been obtained prior to finding the treasure chest a reward of 5 points is given. If the key has been obtained the reward is 25 points. Furthermore finding the key triggers a reward of 3 points. The agent does not receive a reward or a punishment for every step it takes.
Data for the test is obtained by following the current optimal path through the state space. This episode is not used to update the Q-matrix, only for measuring how much reward a greedy-only policy is collecting with the present Q-matrix. For every 4 updating episode, a test episode is conducted. The score from each episode is calculated using the formula in 2. For every step taken the score is updated, only the reward obtained from latest action is used. The reward is multiplied by the discount factor to the power of the current step, in order for the score to increase as fever

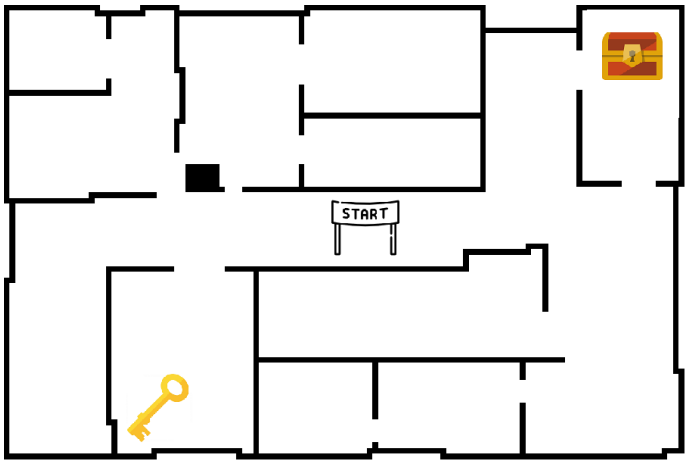


Figure 14: Placement of the key, treasure chest and start in the Q-learning test environment.

steps is required to find the key and the treasure chest. The same discount factor is used always for calculating the score for every test, regarding if the discount factor is different when the Q-matrix is updated. This is done to make it possible to compare the different test results.

$$score = score + r \cdot \gamma^t \,, \tag{2}$$

where r is the reward and γ^t is the discount factor to the t-th timestep.

4.1.2 Test Results

Figure 15 and figure 16 show how tweaking hyperparameters in this case the learning rate and discount, respectively affect convergence time. From the figures it can be deduced that a learning rate of around 0.7 and a discount factor of around 0.4 gives the fastest convergence time for this particular problem.

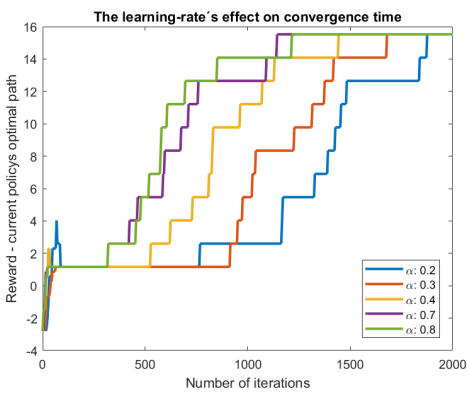


Figure 15: Learning-rate's effect on convergence time of the Q-matrix. Discount factor is 0.97. Epsilon is 0.8.

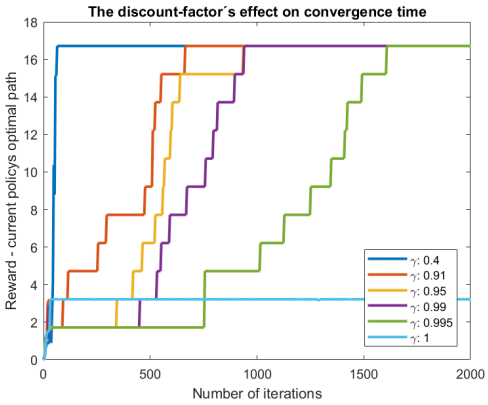


Figure 16: Discount-factor's effect on convergence time of the Q-matrix. Learning rate is 0.8. Epsilon is 0.8.

4.2 Conclusion

Q-learning has been implemented in this project to demonstrate how the robot can learn to find a path that gives it the biggest return. It is shown that the agent finds efficient navigation strategies for maximising return in the provided test environment.

5 Particle Filter

The idea of the particle filter is to have a number of particles each representing an estimate of the position and orientation of the robot in the environment. Additionally each particle has assigned a weight that describes the likelihood of that estimate being true. The position and orientation of the robot can then be estimated combining the estimates in various ways. In each iteration the weights of the particles are updated by looking at how well each individual particle fit the current sensor readings. Also at each iteration the particles are resampled with a higher probability of keeping the particles with high weights.

5.1 Algorithm

Figure 17 shows the steps in the algorithm. Since the project description states that a known initial pose can be assumed [2], the particles are all initialised in close proximity to the robot, however with a completely random orientation. In this implementation only 50 particles are initialised and used for pose estimation.
It has been chosen to only use 50 particles since the test showed decent results with this amount and because a known initial pose can be assumed. If the initial pose could not be assumed, it would probably have been necessary with a lot more particles (as in two-three orders of magnitude more) with an even distribution across the entire map.

5.2 Prediction of new position of particles

In this implementation when iterating through all the particles, estimates are updated in the following way:

1. The particle position is updated by translating the original position in the direction of the original orientation by applying the control speed with some added gaussian noise with a mean of zero and standard deviation of 2.0 although the noise is capped at 0.6.

2. The particle orientation is then updated by applying the control rotation to the original orientation with some added gaussian noise (with a mean of zero and standard deviation of 0.2 rad).

Calculating the error

For every particle the LIDAR observation is compared to a simulated LIDAR observation originating from that particle. In this comparison the error of the particle is the sum of the square of the difference in length between each of the real and simulated LIDAR rays. The weight of that particle is then the inverse of the error calculated used for resampling later in the algorithm. This process is repeated for all the particles to update the weights.

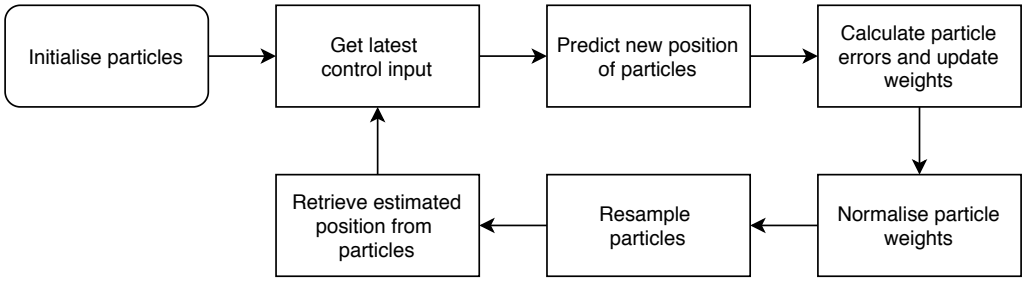


Figure 17: Flowchart showing the process of the particle filter.

Resampling

The process of resampling ensures that particles with bad estimates, characterised by a low weights are discarded and particles with good estimates, characterised by high weights are duplicated. In this implementation of the particle filter, this is done by normalising all weights and then multiplying by the amount of particles. By flooring the normalised weights, each floored normalised weight describes how many of that particular particle needs to be in the new pool of particles. However this process leaves fewer particles than to begin with. Therefore the new pool of particles is filled up with particles randomly sampled from the old pool of particles until there is the required amount of particles.

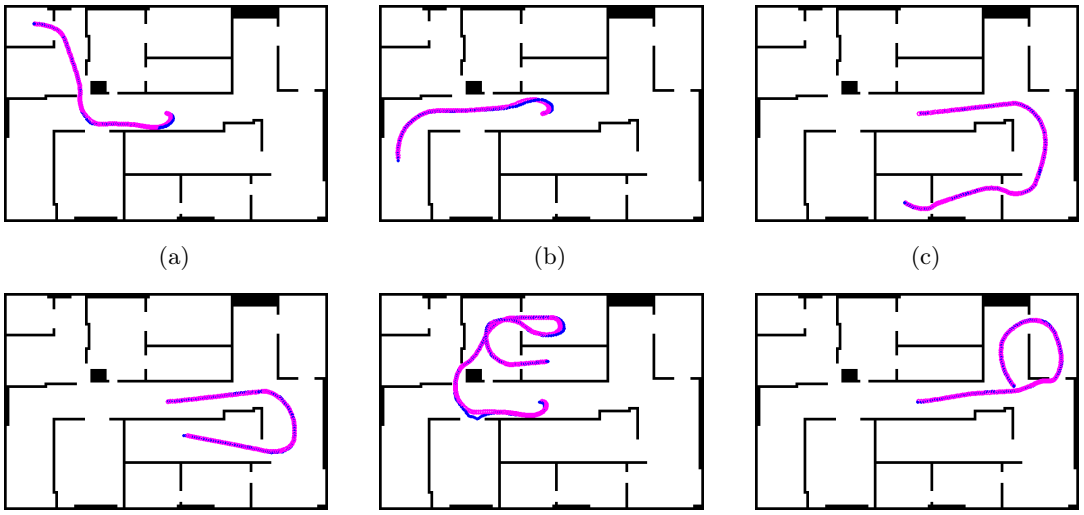
5.3 Tests

To test the particle filter, the robot is repeatedly starting from the same location and then navigating to the same goal. After each run, an algorithm decides whether the particle filter was successful in accurately estimating the position. The run is determined unsuccessful if the distance between estimated position and real position at any time step is higher than a set threshold. Figure 19 shows different runs where the particle filter was not successful in estimating the pose while figure 18 shows other runs where the particle filter was successful in estimating the pose. Results of this test showed that the particle filter was successful in estimating 62 of the 100 runs with this particular goal position.

Additionally figure 20 shows plots of the error in position estimation and orientation estimation for every time step for a sample trajectory, where the pose estimation was deemed successful. This was done qualitively by looking at figure 20a, where it can be seen, that the estimated position and real position seem to follow. However on closer look, the estimated position is often leading the real position, which is why there is often up to 1 m difference between real and estimated positions as shown in the plot in figure 20b. Figure 20c show that difference in estimated and real orientation is at most times not significant especially on long straights, however it becomes more noticeable when the robot is making a turn. Around the 800th timestep the error is very large and flips sign rather quickly. The error of distance and orientation could caused by a flaw in the way that orientation and position is logged and interpreted as well as a flaw in the way particles are updated. Due to time constraints no effort has been put into finding out why these significant errors exist, which is why the proposed reasons above is only conjecture.

Subfigure	Sequence of waypoints
18a	13, 11, 10, 7, 5, 0.
18b	19, 11, 6, 1, 2.
18c	32, 35, 34, 29, 27, 25, 22.
18d	32, 35, 36, 31, 24.
18e	13, 11, 10, 14, 17, 20, 26, 20, 17, 18, 21, 23.
18f	28, 32, 38, 37, 33, 30, 32.

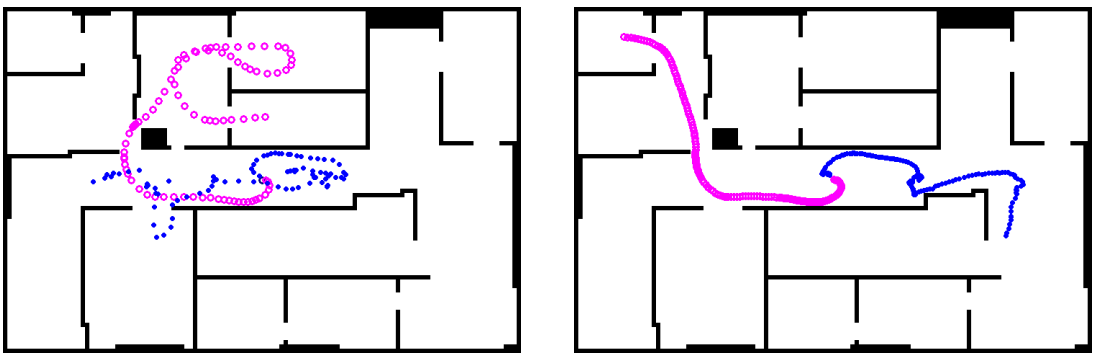
Table 2: The sequence of waypoints used for generating the trajectories seen in figure 18 a-f. The waypoints highlighted are numbered in figure 2.



(a) (b) (c)

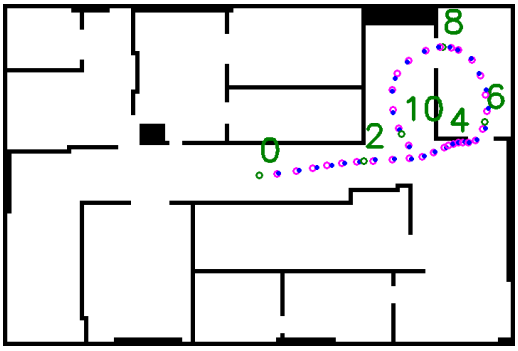
(d) (e) (f)

Figure 18: Different trajectories generated by using different sequences of waypoints described in table 2. Purple circles are real positions while blue dots are estimated positions by a particle filter described in section 5.

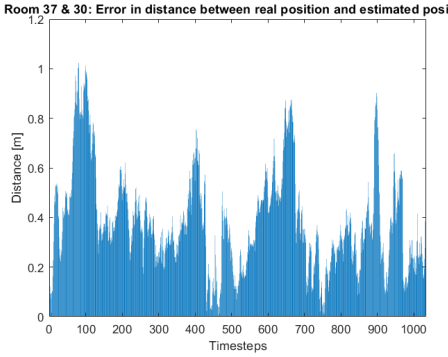


(a) (b)

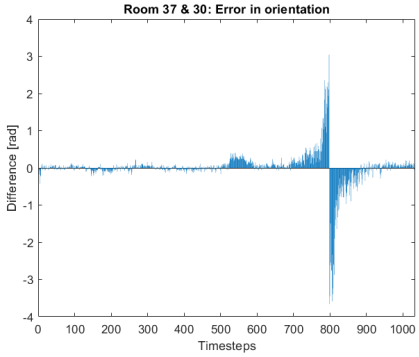
Figure 19: Two scenarios where the particle filter failed to estimate position. Estimated positions are marked with blue dots while real positions are marked with purple circles.



(a) Shows the trajectory of the robot for which the error was logged at every timestep. Numbered green text multiplied by 100 marks the timestep at that point in the trajectory.



(b) Distance between real position and estimated position at each time step.



(c) Difference in rotation between real orientation and estimated orientation at each time step.

Figure 20: Plots of the differences between real and estimated poses at every time step during a trajectory, in this instance the sequence of waypoints corresponding to figure 20a.

5.4 Improvements

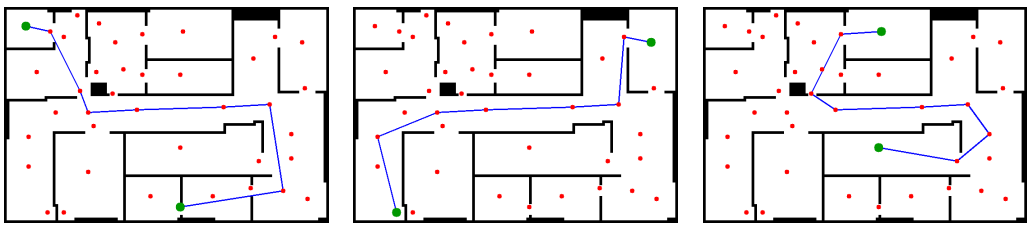
From figure 19 it can be seen that once the particle filter estimated pose diverges from that of the real pose, then the particle filter will never manage to recover. One way of preventing this problem is to improve the particle filter to a point where it never diverges. However should the particle filter begin diverging, a way to recover could be to reinitialise all particles and distribute them evenly across the entire map if the error between the current LIDAR estimation and the LIDAR observation of every particle is above a threshold for a set period of time. As can also be seen in the figure, some of the cases where the particle filter starts diverging is when the robot rotates in a corridor without significant features, i.e. corners. Without significant features such as in a corridor, a small change in rotation of particles yields very similar LIDAR observations, resulting in it being difficult to distinguish between particles whose estimations are good and particles whose estimations are poor.

5.5 Conclusion

A particle filter has been implemented that can be used to localise the robot in the environment. Scenarios are shown where the particle filter manages to correctly estimate the position as well as scenarios where the particle filter fails to correctly estimate the position. If more time had been available, these failures would have been investigated further.

6 Path Planning Using Dijkstra's algorithm

As stated in the project description, an efficient strategy of navigation around the map has to be developed. For achieving this, it was decided to use Dijkstra's algorithm, which is an algorithm that can find the shortest path between two vertices in a graph. Therefore the occlusion map needs to converted into a graph, for this some waypoints is needed. The waypoints used is the same as the ones introduced in section 1. The waypoints then needs to be connected, for this the OpenCV Class *LineIterator* is used. This makes it possible to draw a line between two points and then iterate through each pixel on the line. A line is then drawn between every point in the map. If a wall pixel encountered along the line, then the connection is rejected, if not the it is accepted. The result is stored in an adjacency list. A visualisation of the shortest path found by the algorithm can be seen in figure 21.



(a) (b) (c)

Figure 21: Visualization of the shortest path found by djikstra's algorithm. The green way-points represents the starting and end point.

7 References

[1] *FuzzyLite Library*. http://www.fuzzylite.com/.

[2] Jakob Wilm. *Project Description*. SDU, sep 2019.

[3] *OpenCV Documentation*. https://docs.opencv.org/2.4/index.html.

Implementation code can be found at: https://github.com/chrillemanden/rb-rca5