UNIVERSITY OF SOUTHERN DENMARK

COURSE PROJECT

ROBOTICS 5TH SEMESTER - FALL 2018

Marble finding robot

Alexander Tubæk Rasmussen
alras16@student.sdu.dk

Kenni Nilsson
kenil16@student.sdu.dk

Marcus Enghoff Hesselberg Grove
grov16@student.sdu.dk

1 Abstract

This project is about motion planning and controlling a robot in a closed environment. The idea is to implement a number of algorithms to be used in motion control to navigate the robot.

In order to do so two different planning algorithms were implemented, one sensor based and a model based planner. The model based planner had a higher success rate in navigating around the environment compared to the sensor based planner. However, the sensor based planner had a shorter average path when navigating successfully. The robot must be able to collect marbles and optimise its search strategy through the use of reinforcement learning. The protect the robot from colliding with obstacles, a minimum distance of 0.2 meters must be kept. This was accomplished by making a fuzzy controller, which was used to control the robot. Through the use of Q-learning the robot was able to optimise its search strategy.

In order to detect marbles and obstacles a number of algorithms were designed and implemented utilising both LIDAR and camera data. In general the algorithm based on LIDAR performed well, but the marbles detecting algorithm had a tendency to detect marbles in corners yielding the line detecting method to merge perpendicular lines. The marble detecting algorithm based on the camera generally performed well, but produced falsely results when marbles overlapping or cut off at either top or bottom.

Contents

2 Introduction

Mobile robots are an increasingly larger part of our world. This increases the demand for algorithms that enables the robot to navigate around in its environment, and effectively search the environment. This sets focus on creating planning algorithms and using reinforcement learning to optimise different problems.

In this report a mobile robot platform will be given the task of navigating an environment and collect marbles. While optimising its strategy for each run. The robot will be fitted with a camera, and a LIDAR scanner.
To accomplish this algorithms must be written to enable the robot to navigate the environment. To do this marble detecting and obstacle detecting algorithms must be written, so the robot would be able to localise the marbles and circumnavigate the obstacles.
To ensure that the robot will learn from its previous trials, and optimisation algorithm based on reinforcement learning must be written.

2.1 Problem statement

The overall problem was to design and implement algorithms, that would enable the robot platform to navigate an environment, collect marbles, circumnavigate obstacles as well as optimise its strategy for each trial.
This should be done utilising the knowledge of reinforcement learning, computer vision and robot motion theory.

For this project a two-wheeled robot platform was given the task of collecting blue marbles in a simulated environment. The following problems needs to be solved in order to fulfil the overall objective.

1. How can marbles and obstacles be detected from LIDAR and camera data?
2. How can a planning algorithm be designed and implemented?
3. How can Fuzzy-logic be applied to control the robot?
4. How can reinforcement learning be applied to path optimisation?

To solve the problems stated above, the following tools will be utilised: QT Creator, Gazebo, Fuzzy Lite, OpenCV and Matlab.

2.2 Specification of requirements

From the outline of the project as well as the problem statement, the following requirements to the robot have been formulated:

- The robot must be able to detect blue colours in an image so that blue marbles will be detected and collected by the robot.
- To insure a safe working space for the robot to operate, a minimum distance of 0.2 meters to obstacles must be respected.
- To be able to collect as many marbles as possible, the robot must be capable of finding all areas in a closed environment.
- It must be able to optimise its current path and find an optimal path with minimal numbers of unrewarded moves.
- The robot must be able to distinguish obstacles from marbles as well as localising the position of the marbles.

2.3 Readers guide

In this report the following references will be used:

- Figures and tables will be referred to by numbers in parentheses (1) or (1.a)
- Sections will be referred to by numbers 1 or 1.1.
- Equations will be referred to by numbers in parentheses (1) or (1.1).
- Citations will be referred to by author and year as well as page numbers in parentheses (Weiss, 2014, pp. 417-419).
- Sections in appendix will be referred to by the capital letter followed by a number in parentheses (A.1).

The following will be used to describe software elements in this report:

- Classes, data types and methods will be shown in `verbatim`.
- Linguistic variables will be shown in *italic*.
- Code commands in the report will be shown in **bold style**.

The following will be used to show algorithms in the report:

Algorithm: Find

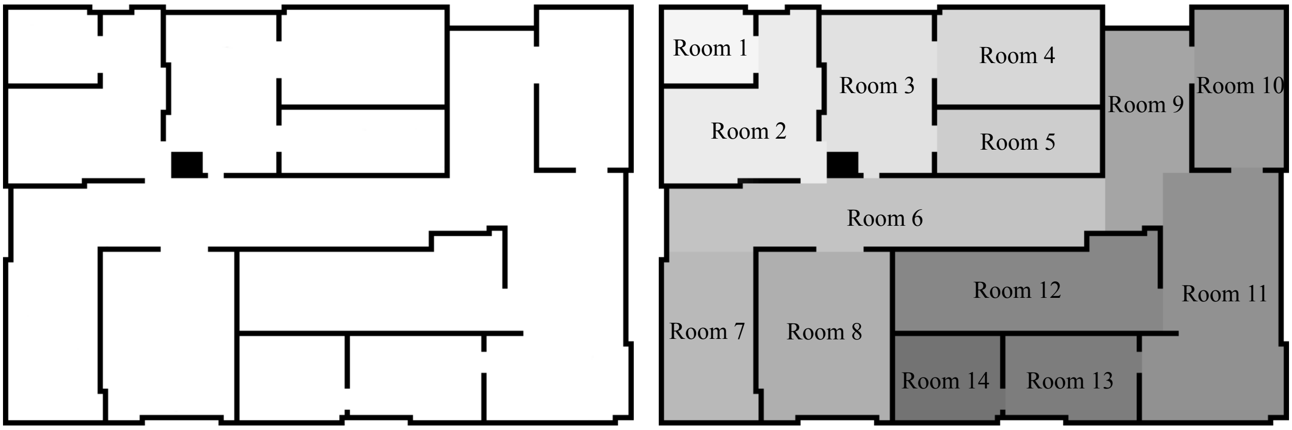
Input: A vertex
Output: The set containing the vertex

if vector(vertex) < 0
return vertex
else
return find(vector(vertex))

3 Design

3.1 Environment

The two-wheeled robot should navigate around the environment called bigworld shown in figure (1a). To do so, it have been chosen to divide the environment into rooms. The natural definition of *'rooms'* have been taken into account, resulting in a 14 room layout. This can be seen in figure (1b), where each room can be distinguished by different grey scale colours.



(a) Illustration of the world "bigworld" *(b) "bigworld" divided into 14 rooms*

Figure 1: Illustration of the world "bigworld" before and after it has been divided into 14 rooms

The division into rooms are useful in terms of knowing which rooms that have been searched and which that not yet have been searched for marbles.
It is also useful as an abstract state space for reinforcement learning such that it can be found which order the rooms should be visited.

3.2 LIDAR Sensor

The two-wheeled robot given for this project is among other equipped with a 2D LIDAR (Light Detecting and Ranging) sensor. A LIDAR scanner detects the distance to targets by emitting a laser pulse and analysing the time it takes for the beam to reflect and return to its source. The LIDAR sensor maps the environment and has a detecting range of 10 pixels in Gazebo and a 260 degrees field of view.

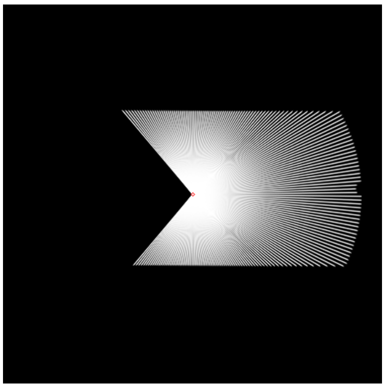
The script converts this range into mm by first scaling the pixel map to double size, such that the detecting range is 20 pixels. Afterwards, the script trace the pixel map to a eps-figure using 72 dpi as standard for the postscript. Then the script converts this range from inches to mm by using a conversion factor of 25.4 mm per inch. This means that this range can be converted into mm by using the following formula:

$$\frac{20\,pixels}{72\,dpi} \cdot 25.4\,\frac{mm}{inch} = 7.06\,mm$$

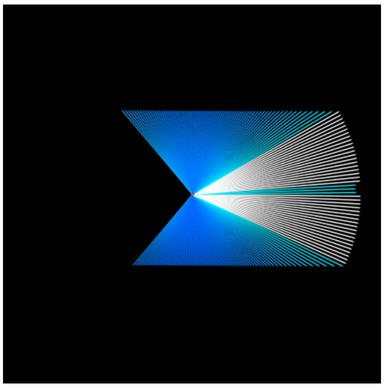
Now the script scales the world from mm to m, which means that the range of the LIDAR sensor therefore is 7.06 m. The LIDAR sensor maps the surrounding environment by collecting 200 datapoint, which must be processed in order to recognise objects such as walls and marbles. This means that circle and line detecting algorithms must be writing.

The datapoints from the LIDAR sensor is first visualised by drawing white lines from the robots location to each of the 200 datapoints using the `cv::line()` function as shown in figure (2a). Afterwards the datapoints are sorted by checking if the range is equal to sensor range in order to get the points that reflects from different objects.

The filtered datapoints are drawn as lines upon the unsorted data. These lines can be distinguished by different blue colours depending on the range between the two points ranging from blue to cyan. This blue coloured and white lines are shown in figure (2b).



(a) Illustration of the unfiltered data



(b) Illustration of the filtered data

Figure 2: Illustration of the unfiltered and filtered data

The two sections below, describes the design of a line and a marble detection algorithm.

3.2.1 Marble detection

The objective for the two-wheeled robot is to collect marbles effectively while avoiding obstacles such as walls. The Hough transform is a algorithm that maps from image space to the probability of the existence of features such as lines, circles or other general shapes. The algorithm is also capable of detecting partial object as in the case of the LIDAR scanner.
However the algorithm are not very robust to noise and can be very hard to calibrate in order to make the result from the algorithm useful. Furthermore the algorithm have a high computational cost. Because of this, it have been chosen to design a simpler method for finding marbles in the LIDAR data.

Due to the fact that only a part of the circles would be visible from the data, it was chosen to calculate the center and radius of the circles from the chord and arch height of the circles. This can be used to determine the location of the marbles relative to the robot. Given this information the planner would be able to drive towards the marbles and "collect" them.
It is assumed that the two detected outer points on the circle periphery defines the circle chord. It is also assumed that the orthogonal distance from the detected middle point on the circle periphery to the circle chord defines the circle arch height. This only applies for an uneven number of detected point.
The circle chord can be determined using the formula:

$$C = 2r \cdot \sin\left(\frac{\theta}{2}\right)$$

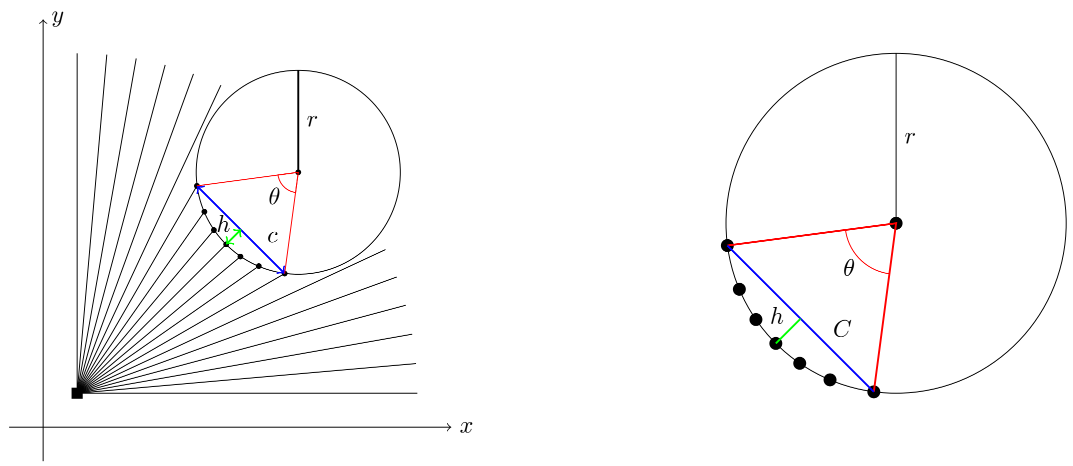
The circle camber can be determined using the formula:

$$h = r\left(1 - \cos\left(\frac{\theta}{2}\right)\right)$$

Solving this equation system, the circle radius was found to be:

$$r = \frac{C^2 + 4h^2}{8h}$$

The polar coordinates of the circle center can be found by adding the radius to the range of the detected middle point.



(a) Illustration of detected points (marked as ending point of the black lines), the circle chord and arch heigh can be found from the LIDAR data

(b) Illustration of the circle chord and arch heigh can be found from the LIDAR data and thereby can determine the circle's radius and the chord angle

Figure 3: Illustration of determination of the circle radius from LIDAR data

3.2.2 Line detection

It is usually important for a mobile robot to know its environment. There are several reasons for that. One is that robot must know the location of the obstacles (walls) relative to it to avoid driving into them. In the "bigworld" environment, the obstacles are walls which can be represented as lines from the filtered datapoints. These lines can be represented using a normal parametrisation in polar coordinates, given by the following formula (Munoz L. R. 2014, pp. 1–3)

$$l: \quad r = x \cdot \cos(\alpha) + y \cdot \sin(\alpha) \tag{3.1}$$

where r represents the distance from the origin to the closest point on the line and α is the angle between the x-axis and the plane normal. (Munoz L. R. 2014, pp. 1–3)

The Total Least Square method assumes that a line can be represented as in equation (3.1). This method also involves a determination of the orthogonal distance (the shortest distance) from a point p_i to a line l as shown in figure (4). The normal parametrisation of the line l_i is given by the following formula (Munoz L. R. 2014, pp. 1–3)

$$l_i: \quad r_i = x_i \cdot \cos(\alpha) + y_i \cdot \sin(\alpha) \tag{3.2}$$

The separation between those two lines (l and l_i) is given by the difference $d_i = r_i - r$, since both lines have the same α. This means that the orthogonal distance can be described using the following formula (Munoz L. R. 2014, pp. 1–3)

$$d_i = x_i \cdot \cos(\alpha) + y_i \cdot \sin(\alpha) - r \tag{3.3}$$

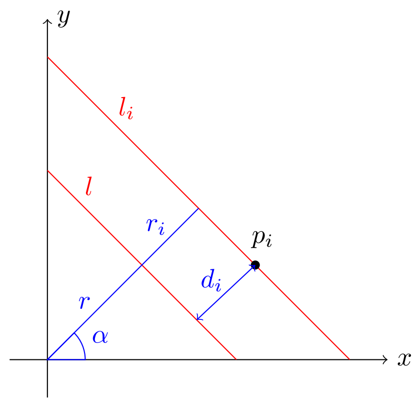


Figure 4: Orthogonal distance from point p_1 to line l

This only applies if we assume that there is no noise on the measurements. (Munoz L. R. 2014, pp. 1–3)

This method gives a solution to the problem of fitting a straight line to a dataset of points p with n measurements having errors. The problem of fitting a line can be determined using the following sum (Munoz L. R. 2014, pp. 1–3)

$$\chi^2\left(l, z_1, ..., z_n\right) = \sum_{i=1}^{n}\left[\frac{\left(x_k - X_k\right)^2}{u_{x,k}^2} + \frac{\left(y_k - Y_k\right)^2}{u_{y,k}^2}\right] \tag{3.4}$$

where (x_k, y_k) are the points coordinates with corresponding uncertainties $(u_{x,k}, u_{y,k})$ and (X_k, Y_k) denote its corresponding point of the straight line l. In the case of fitting the best line to the dataset, minimises the expression for χ^2 by setting $u_{x,k} = u_{y,k} = \sigma$ and $k = 1, ..., n$. This reduces the problem to the Total Least Square method and minimising is equal to minimising the orthogonal distance of the measurements to the fitting line. Therefore, in the case of fitting the best line minimises the expression above to the following (Munoz L. R. 2014, pp. 1–3)

$$\chi^2\left(l; Z\right) = \sum_{i=1}^{n}\frac{d_i^2}{\sigma^2} \tag{3.5}$$

$$= \sum_{i=1}^{n}\frac{\left(x_i\cos(\alpha) + y_i\sin(\alpha) - r\right)^2}{\sigma^2} \tag{3.6}$$

$$= \frac{1}{\sigma^2}\cdot\sum_{i=1}^{n}\left(x_i\cos(\alpha) + y_i\sin(\alpha) - r\right)^2 \tag{3.7}$$

A condition for minimising χ^2 is done by solving the non-linear equation system with respect to each of the two line parameters (r and α). (Munoz L. R. 2014, pp. 1–3)

$$\frac{\partial\chi^2}{\partial r} = 0 \qquad \frac{\partial\chi^2}{\partial\alpha} = 0 \tag{3.8}$$

The solution of this non-linear equation system is determined to the following (Munoz L. R. 2014, pp. 1–3)

$$r = \overline{x}\cos(\alpha) + \overline{y}\sin(\alpha) \tag{3.9}$$

$$\alpha = \frac{1}{2}\arctan 2\left(\frac{-2\sum_{i=1}^{n}\left[(x_i - \overline{x}) - (y_i - \overline{y})\right]}{\sum_{i=1}^{n}\left[\left(x_i - \overline{x}\right)^2 - \left(y_i - \overline{y}\right)^2\right]}\right) \tag{3.10}$$

where $\overline{x}$ and $\overline{y}$ are the means of x and y. (Munoz L. R. 2014, pp. 1–3)

There are some practical considerations to take by using equation (3.9) and (3.10). Equation (3.10) uses the four quadrant arc tangent by using the function atan2, which computes the quadrant in which the angle lies. This function will result in the same result in the case of $y = x = -2$ and $y = x = 2$, which means that an distinction would be lost using this single-argument arc tangent function. Another practical consideration is the case of $r < 0$ in equation (3.9), which is solved by multiplying the range r with -1 and adding π to the angle θ. (Munoz L. R. 2014, pp. 1–3)

As explained earlier, the two-wheeled robot should avoid obstacles (walls). To do so, the robot needs to know the location of the walls. It is done by processing the datapoints from the LIDAR sensor and determining the points which fits to a straight line using a line extraction algorithm. There are several different line extraction algorithms to choose from. The incremental line extraction algorithm is chosen, because it is simple to implement. The pseudo code for the incremental algorithm is shown below. (Nguyen V. 2005)

Algorithm: Incremental

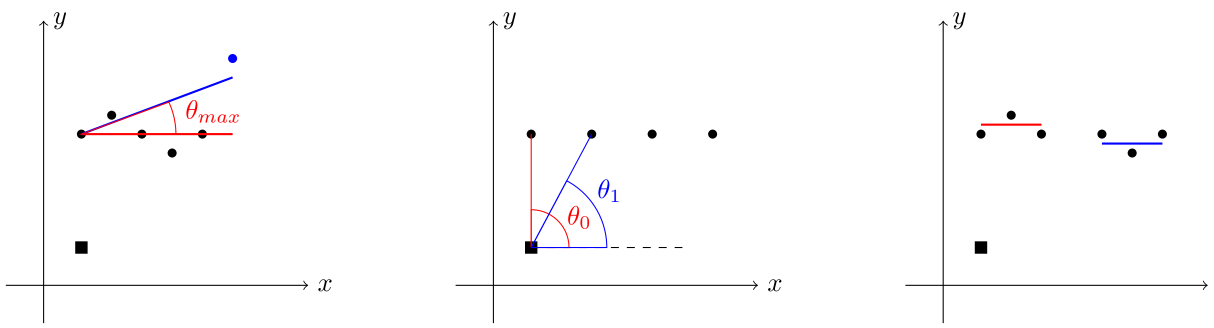
1. Start by the first 2 points, construct a line
2. Add the next point to the current line model
3. Recompute the line parameters
4. If it satisfies line condition (go to 2)
5. Otherwise, put back the last point, recompute the line parameters, return the line
6. Continue to the next 2 points, go to 2

This algorithm implements the Total Least Square method then computing the line parameters. Furthermore, the line model consists of points which all must comply some line conditions. In that case, all points must comply these conditions. The first condition is a threshold for the angle between the previous and current line model as shown in figure (5a). The threshold is defined to be the following

$$\theta_{max} = 0.0025$$

The second condition is the angle between two points relative to the robot location as shown in figure (5b). This angle should be greater than this difference, but not twice as great, since this condition should separate the points into two lines if one point is missing on the list as shown in figure (5c). This angle is therefore defined to be the following

$$\Delta\theta = (\theta_0 - \theta_1) \cdot 1.25$$



(a) Illustration of the angle between the current line model (blue line) and the previous line model (red line)

(b) Illustration of the angle from horisontal to datapoint relative to the robot

(c) Illustration of the separation of to lines due to a missing datapoint

Figure 5: Illustration of the angle between two datapoints and two line models

3.3 Camera marble detection

The camera fitted to the two-wheeled robot could be used to detect marbles at a greater distance than the LIDAR scanner, giving the robot the ability to search a room for marbles and then collect them using the LIDAR.
To detect marbles in the image from the camera, a marble detecting algorithm must be written.

The size of the detected marbles would given an indication of the distance to it and the distance to the center of the image in the x direction would indicate the direction the robot should move in order to collect the marble. Because of this it have been chosen to implement a complex data structure to contain the information on the center in the image and radius, as well as the difference to the center of the image in the x direction.

Because the environment have different nuances of grey, and the marbles being blue, it was chosen to make a colour separation of what could be marbles and the background. This would result in a binary image where the white parts would have the colour of the marbles while the rest of the image being black.

This image could contain multiple marbles that should be detected. One way of doing this could be to the use the Hough transform for circles on the image.
Even though the Hough transform are considered a very powerful algorithm , the drawback of using this algorithms cannot be overlooked. The Hough transform are very computationally heavy, and can be very difficult to tune correctly. The tuning parameters should also change depending on the size of the marbles. Furthermore there are a fine line between not detecting any circles to getting many falsely detected circles. These falsely detected circles should then be discarded, which would be an additional computational cost. (Dawson-Howe K. 2014, pp. 108–112)
Therefore this solution was considered infeasible and it was chosen to separate the different marbles using connectivity. By doing this non-overlaying marbles would be detected as different objects, and overlaying marbles would be detected as a single object.

Due to this the height and width of the objects can be calculated. By assuming the objects are marbles, the center of the marble could be defined as the center of the object, and the radius as half of then height of the objects. Because the marbles could be blocked by walls or parts of the marble being left out of the image, the radius would be calculated based on the height and not the width of the object.

This is a simple and crude way of detecting marbles, and it would be subject to the following errors. A marble being cut off at either the left or right side would have a shifted center. Marbles being cut off at either the top or bottom would have a smaller radius. Overlapping marbles would be detected as one with the radius of the larger one, and the center shifted towards the smaller one.

The problem of marbles being cut of at either the left or right side could be prevented, by shifting the center either further the left or further to the right.

3.4 Motion planning

Two motion planers are going to be used as motion planning for the robot. As the sensor based planer a tangent bug algorithm will be used. The reason for choosing this algorithm is the effectiveness of the tangent bug which only needs an initial position and target location. A number of algorithms will be taken into consideration of the model based planner for generating a road map. Both motion planners will use fuzzy logic for robot control. The fuzzy controller will be discussed in section 3.5 and 4.4.

3.4.1 Sensor based planner

The tangent bug algorithm is a sensor based planer which relies on inputs from a sensor to determine whether it should continue towards a specified location q_{goal} or to follow an obstacle O_i until a free path is available. The advantage of using this algorithm is the computational minimisation and the fact that it only needs a start and end point. The reason for using this algorithm is the benefits just explained and that it will always try to achieve a shortest path solution to the goal location. Pseudo code for the implementation for the tangent bug algorithm is shown below. (Choset H. 2005, pp. 23–37)

Algorithm: Tangent Bug Algorithm

Input: A robot with a range sensor
Output: A path to the q_{goal} or a conclusion no such path exist
while True **do**
repeat
Continuously move toward the point *n* ϵ $(T,\ O_i)$ which minimises $d(x,\ n) + d\,(n,\ q_{goal})$
until
- The goal is encountered **or**
- The direction that minimises $d(x,n)$ +d$(n,\ q_{goal})$ begins to increase d$(x,\ q_{goal})$ so the robot detects a local minimum d$(x,\ O_i)$ + d$(O_i,\ q_{goal})$ on the boundary *i*.

Now choose the boundary following direction which continuous in the same direction as the most recent motion-to-goal direction.
repeat
Continuously update d_{reach}, $d_{followed}$ and O_i.
Continuously moves toward n ϵ $\{O_i\}$ that is the chosen boundary direction.
until
- The goal is reached
- The robot completes a cycle around the obstacle in which case the goal cannot be reached.
- $d_{reach} < d_{followed}$

end while

It has been decided to make small changes to the algorithm and make it greedy. Instead of following the boundary $d_{reach} < d_{followed}$, the robot will follow that obstacle that minimises $d(x,n)$ + d$(n,\ q_{goal})$. The reason for doing this is to reduce the path from a giving point to a target location.

Because the point giving to the tangent bug is relative to the world frame in Gazebo a mapping between to configurations is needed. The robot is shifted π radians relative to the world frame which give the following transformation matrix. (Choset H. 2005, pp. 60–66)

$$T = \begin{bmatrix} cos(\theta - \frac{\pi}{2}) & -sin(\theta - \frac{\pi}{2}) & robotPos.x \\ sin(\theta - \frac{\pi}{2}) & cos(\theta - \frac{\pi}{2}) & robotPos.y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

$$p^{robot} = T^{-1} \cdot p^{world} \tag{3.12}$$

Now by calculating equation (3.12) one can get the desired orientation to the target location from p_{robot} by using atan2.

3.4.2 Model based planner

To be able to generate a complete road map a number of algorithms have been taken into consideration namely the brushfire algorithm, Kruskal's algorithm and an extended version of the Depth-First Search algorithm. In the beginning of the project other algorithms were taken into consideration like the trapezoidal decomposition. The problem about this algorithm is that it works well on polygons and other odd structures, but because the map used in the project mainly consist of rectangles oriented either along the x-axis or perpendicular to the x-axis this algorithm was rejected. Instead the brushfire algorithm was considered appropriate. (Choset H. 2005, pp. 162–168)
The reason for choosing this algorithm is that it gives a set of values to the pixels furthest away from the obstacles, which can be used to generate a path for the robot by picking out the pixels that leads to a complete road map for the robot to navigate through the entire map. To be able to connect the points on the map, Kruskal's algorithm was used. Using this algorithm, one can connect points on a map and avoid cycles between points, which makes path planning easier. An extended version of the Depth-First Search algorithm is used to generate a set of points which the robot has to follow to reach its goal. This algorithm was seen efficient to solve the problem of path generation and therefore used.

The brushfire algorithm uses a grid to approximate distance to obstacles. The idea is to give obstacles a starting value of 1 and free-space pixels a value of 0. Then continue until the $'fire'$ has consumed all free pixels thereby giving pixels furthest away from obstacles the highest value. It was decided to use a eight-point connectivity grid. (Choset H. 2005, pp. 86–89)

Algorithm: Brushfire algorithm

while True **do**
label++
  for all i height of image
    for all j width of image
    if adjacent pixel values to image(i, j) is label & image(i, j) is 0 set image(i, j) to label + 1
  until
    All pixels have an assigned value
end **while**

Now the goal is to find center points from the bushfire algorithm so that they can be used to establish connections between rooms. The idea is to use Kruskal's algorithm to connect points to one another. Kruskal's algorithm uses the disjoint set union/find algorithm which is an algorithm used to find relations between vertices. It starts by initialising a vector by the size of the number of vertices and sets them to -1. The `find(int V1)` method uses recursion to see if the vertices are joint or disjoint, meaning that they form a cycle if they are connected. If they are not connected the method `unionSets(int V1, int V2)` connects the two vertices, which would otherwise result in a cycle between a number of vertices. (Weiss M. A. 2014, pp. 356–419)

Algorithm: Kruskal's algorithm

for all edges
  integer V1 = find(edge.V1)
  integer V2 = find(edge.V2)
  if V1 $\neq$ V2
  push edge on vector
  unionSets(V1, V2)

Now the idea of generating connections between all vertices on the map is complete. The last thing which needs to be solved is to be able to generate a path between vertices from an initial position to a target location. Depth-First Search algorithm uses recursion to find a path between vertices. It uses a vector of vectors (vertices) to recursively generating a full path from a starting vertex to target vertex. (Weiss M. A. 2014, pp. 419–420)

Algorithm: DFS extended

Input: Start and target location
Output: Vector of vertices in order
push start on vector
mark start as visited

if start $==$ target
return vector

for all vertices adjacent to start
if adjacent $==$ target
push adjacent on vector
return vector

else if adjacent $\neq$ visited
DFS(adjacent, target)
pop last element from vector

Thus the user should be able to give a start and target location from the number of vertices from the list yielding a complete route for the robot to travel.

3.5 Search strategy

For the robot to be able to avoid obstacles as well as navigating through the map, Fuzzy Control will be used. For short the a fuzzy controller consist of a fuzzification interface that converts input into information that can be used by the inference mechanism. The inference mechanism evaluates the giving input in addition to the rule base, which is based on the expert's linguistic description. The output of the fuzzy controller will then be defuzzified to crisp values which will be used as input to control the plant (robot). The following linguistic terms will be used:

- The linguistic input variable called $ObstacleDirection = \{right, center, left\}$ with the named linguistic values. The universe of discourse is set to $U = [-1.6, 1.6]$. It defines the direction towards and obstacle and the choose of U is based on the angle range from the sensor. The membership function for this linguistic variable can be seen in figure (7a).
- The linguistic input variable called $obstacleFree = \{right, center, left\}$ with the named linguistic values. The universe of discourse is set to $U = [-1.6, 1.6]$. It defines the angle to which and obstacle is furthest away from the robot. This is used when the robot is driving towards a corner and has to avoid collision. The choose of U is based on the angle range from the sensor. The membership function for this linguistic variable can be seen in figure (10).
- The linguistic input variable called $ObstacleDistance = \{veryclose, close, far\}$ with the named linguistic values. The universe of discourse is set to $U = [0, 10]$. It defines the distance to an obstacle and the chose of U is based on the sensors maximum detection range. The membership function for this linguistic variable can be seen in figure (7b).
- The linguistic input variable called $MarbleDirection = \{right, center, left\}$ with the named linguistic values. The universe of discourse is set to $U = [-30, 30]$. It defines the direction from the robot's point of view as a changes

in pixel values in the picture from the camera placed on the robot. The universe of discourse was found as a suitable deviation from the center point. The membership function for this linguistic variable can be seen in figure (8a).

- The linguistic variable called $MarbleFound = \{no, yes\}$ with the named linguistic values. The universe of discourse is set to $U = [0, 50]$. It defines if an marble is detected where the input is a radius of the marble on the picture from the camera on the robot. The chose of U was found suitable. The membership function for this linguistic variable can be seen in figure (8b).
- The linguistic variable called $GoalDirection = \{right, straight, left\}$ with the named linguistic variables. The universe of discourse is set to $U = [-3.14, 3.14]$. It defines the direction in which a target location is located. The chose of U is based on a complete rotation from the robot's point of view. The membership function for this linguistic variable can be seen in figure (9a).
- The linguistic input variable $BoundaryDirection = \{right, straight, left\}$ with the named linguistic values. The universe of discourse is set to $U = [-3.14, 3.14]$. It defines boundary direction on an obstacle in which the robot has to follow if it is in an obstacle following behaviour. The chose of U is based on a complete rotation from the robot's point of view. The membership function for this linguistic variable can be seen in figure (9b).
- The linguistic output variable called $SteerDirection = \{sharpright, right, softright, straight, softleft, left, sharpleft\}$ with the named linguistic values. The universe of discourse is set to $U = [-1.57, 1.57]$. It defines the direction in which the robot has to navigate. The chose of U was found suitable. The membership function for this linguistic variable can be seen in figure (11).
- The linguistic output variable called $Velocity = \{backward, softbackward, softforward, forward\}$ with the named linguistic values. The universe of discourse is set to $U = [-1, 1]$. It defines the velocity giving to the robot and the chose of U was found suitable for the implementation of the controller. The membership function for this linguistic variable can be seen in figure (12).

In order for the inference mechanism to work, one has to define a rule base in which the linguistic variables and values are used. To be able to move the robot, find marbles and avoid obstacles the following rule base has been defined:

- **Rule 1:** ***if*** *ObstacleDistance* is *veryclose* and *ObstacleDirection* is *left* and *MarbleFound* is *no* ***then*** *SteerDirection* is *softright*
- **Rule 2:** ***if*** *ObstacleDistance* is *veryclose* and *ObstacleDirection* is *right* and *MarbleFound* is *no* ***then*** *SteerDirection* is *softleft*
- **Rule 3:** ***if*** *ObstacleDistance* is *veryclose* and *ObstacleDirection* is *center* and *ObstacleFree* is *left* and *MarbleFound* is *no* ***then*** *SteerDirection* is *softleft*
- **Rule 4:** ***if*** *ObstacleDistance* is *veryclose* and *ObstacleDirection* is *center* and *ObstacleFree* is *right* and *MarbleFound* is *no* ***then*** *SteerDirection* is *softright*
- **Rule 5:** ***if*** *ObstacleDistance* is *veryclose* and *MarbleFound* is *no* ***then*** *Velocity* is *forward*
- **Rule 6:** ***if*** *ObstacleDistance* is *close* and *MarbleFound* is *no* and *BoundaryDirection* is *left* and *GoalDirection* is *left* ***then*** *SteerDirection* is *softleft*
- **Rule 7:** ***if*** *ObstacleDistance* is *close* and *MarbleFound* is *no* and *BoundaryDirection* is *right* and *GoalDirection* is *right* ***then*** *SteerDirection* is *softright*
- **Rule 8:** ***if*** *ObstacleDistance* is *close* and *MarbleFound* is *no* and *BoundaryDirection* is *left* and *GoalDirection* is *right* ***then*** *SteerDirection* is *softleft*
- **Rule 9:** ***if*** *ObstacleDistance* is *close* and *MarbleFound* is *no* and *BoundaryDirection* is *right* and *GoalDirection* is *left* ***then*** *SteerDirection* is *softright*

- **Rule 10:** ***if*** *ObstacleDistance* is *close* and *MarbleFound* is *no* and *BoundaryDirection* is *straight* then *SteerDirection* is *straight*
- **Rule 11:** ***if*** *ObstacleDistance* is *close* and *MarbleFound* is *no* ***then*** *Velocity* is *forward*
- **Rule 12:** ***if*** *ObstacleDistance* is *far* and *MarbleFound* is *no* and *GoalDirection* is *right* ***then*** *SteerDirection* is *right*
- **Rule 13:** ***if*** *ObstacleDistance* is *far* and *MarbleFound* is *no* and *GoalDirection* is *left* ***then*** *SteerDirection* is *left*
- **Rule 14:** ***if*** *ObstacleDistance* is *far* and *MarbleFound* is *no* and *GoalDirection* is *straight* ***then*** *SteerDirection* is *straight*
- **Rule 15:** ***if*** *ObstacleDistance* is *far* and *MarbleFound* is *no* and *GoalDirection* is *straight* ***then*** *Speed* is *forward*
- **Rule 16:** ***if*** *MarbleFound* is *yes* and *MarbleDirection* is *left* ***then*** *SteerDirection* is *softleft*
- **Rule 17:** ***if*** *MarbleFound* is *yes* and *MarbleDirection* is *right* ***then*** *SteerDirection* is *softright*
- **Rule 18:** ***if*** *MarbleFound* is *yes* and *MarbleDirection* is *center* ***then*** *Velocity* is *forward*

The inputs to the fuzzy controller can be seen in figure (6). It takes the inputs and makes a fuzzyfication of the crisp values which will be used in the controller. The inference mechanism will evaluate the fuzzyfied inputs by the rule base of the fuzzy controller. The defuzzification interface will then convert the conclusions into crisp values as outputs of the fuzzy controller to be used as input to the plant.

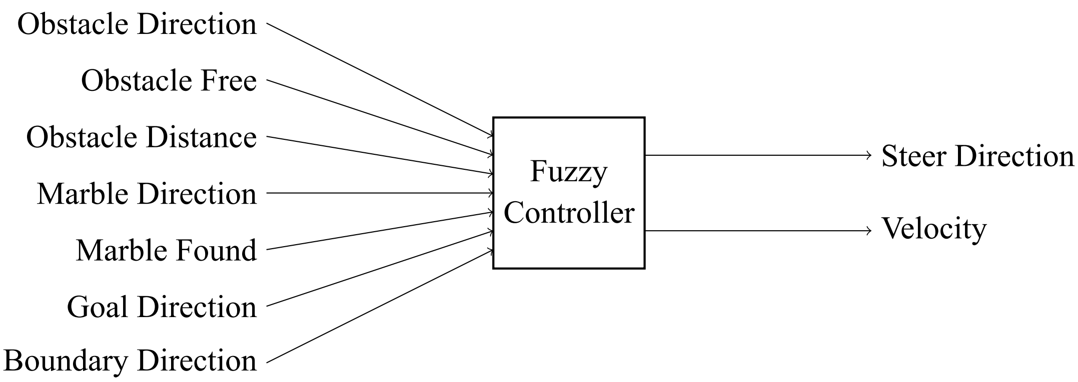
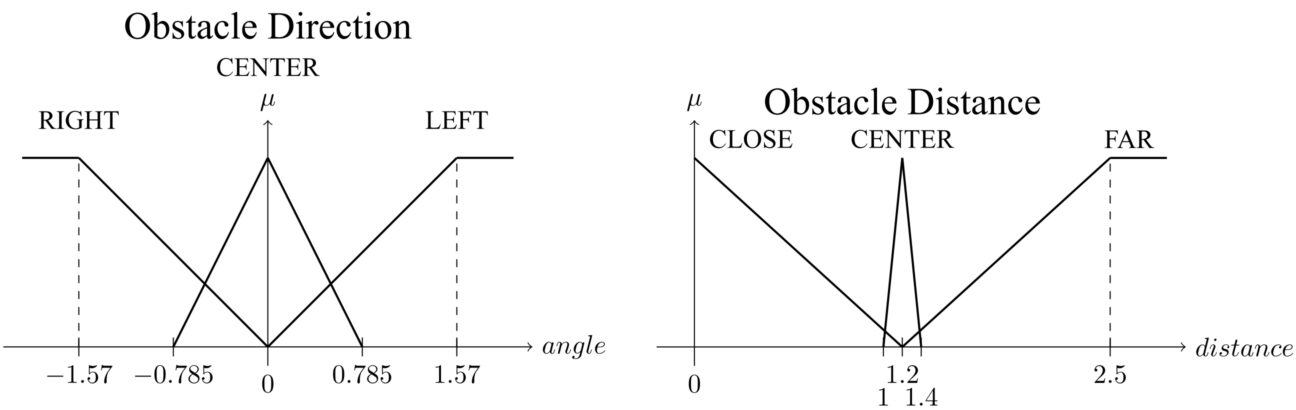


Figure 6: Illustration of inputs and outputs to the fuzzy controller.

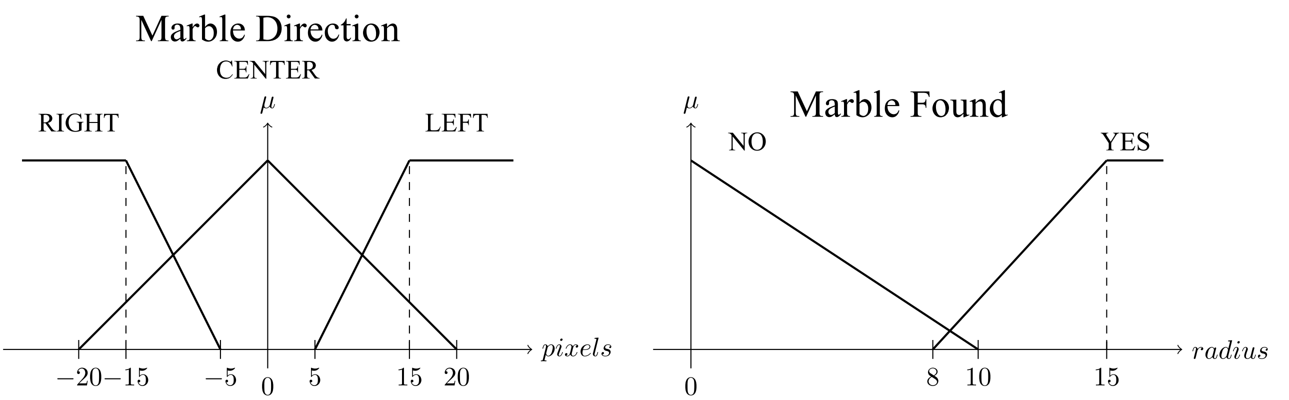
The membership functions used in the fuzzy controller can be seen in the figures below.



(a) Angles giving as inputs

(b) Distance giving as input

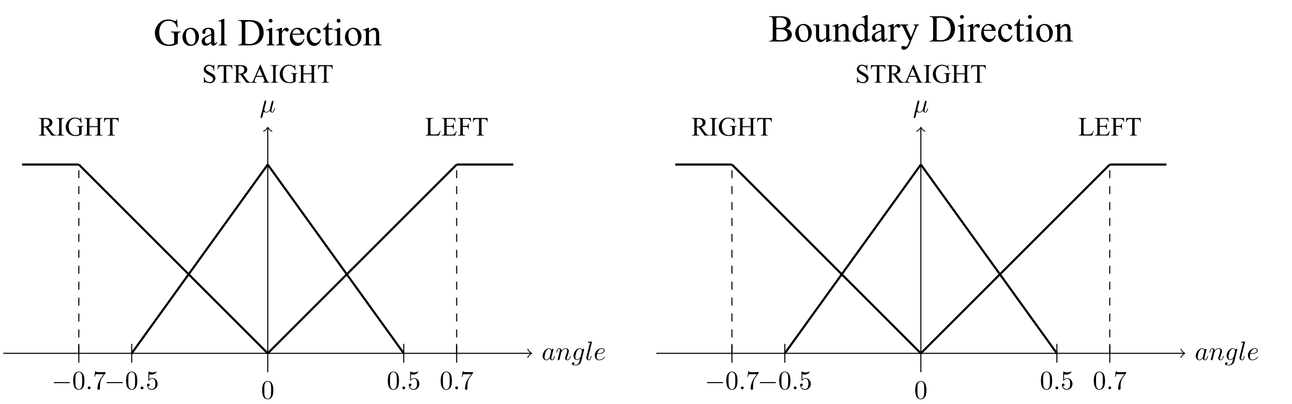
Figure 7: Illustration of the membership functions Obstacle Direction and Obstacle Distance with μ as the degree of truth



(a) Pixels giving as inputs

(b) Radius giving as input

Figure 8: Illustration of the membership functions Marble Direction and Marble Found with μ as the degree of truth



(a) Angle to goal giving as input

(b) Angle to boundary direction giving as input

Figure 9: Illustration of the membership functions Goal Direction and Boundary Direction with μ as the degree of truth

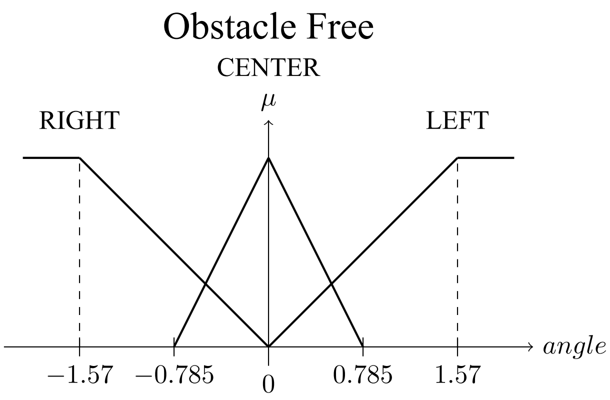


Figure 10: Membership function Obstacle Free with an angle to free space giving as input and μ as the degree of truth

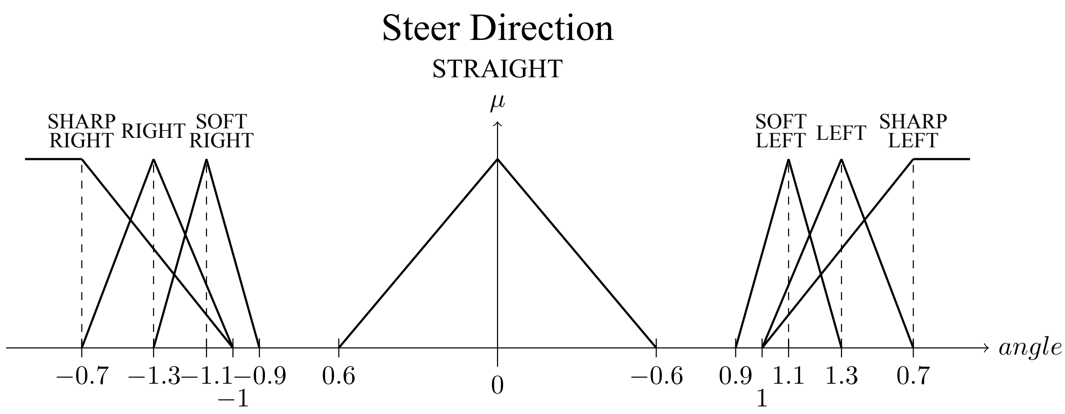


Figure 11: Membership function Steer Direction with an angle to the target as output and μ as the degree of truth

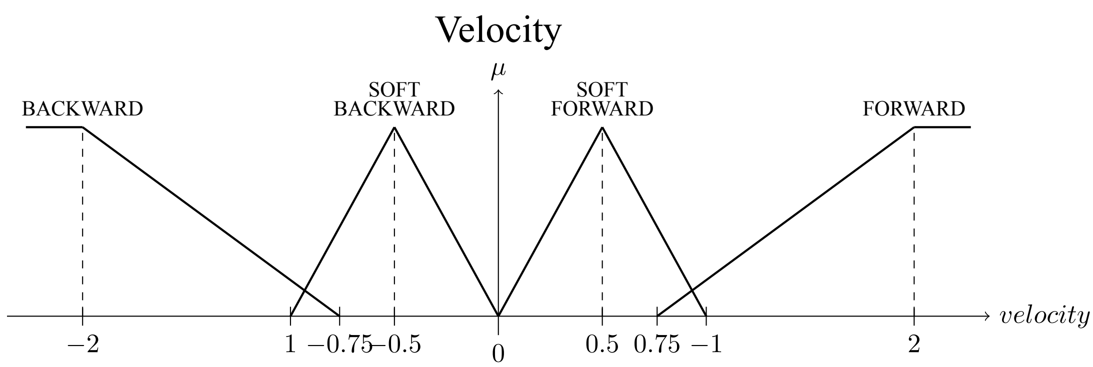


Figure 12: Membership function Velocity with a velocity to the target as output and μ as the degree of truth

3.6 Q-learning

In order to effectively search the environment and collect marbles, a good search strategy must be found. This can be done by utilising reinforcement learning. By using reinforcement learning, the robot can learn from its experience and obtain a good strategy for navigating the environment.

By using a Temporal-Difference learning strategy the optimal action-value function can be estimated by every move taken, unlike a Monte Carlo strategy where an episode terminates before any learning is obtained. In some cases with long episodes the Monte Carlo strategy is considered too slow.
Generally there are two categories of Temporal-Difference learning; on-policy and off-policy methods. One of the advantages of an off-policy over an on-policy method are that the action-value function can be estimated independent from the policy being used. The policy only influences which state-action pairs that are visited and updated.

Based on this, Q-learning is chosen. The Q-learning updates the action-value function (Q-values) by using the formula in equation (3.13).

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{3.13}$$

The update function for Q-learning consists of the old value for a given state-action combination plus a scaled difference between the old value, the immediate reward and the maximal value for the next state. The learning rate are denoted α and ranging from 0-1 preferable closer to 0, in order not to base the policy on this action only. γ denotes the discount factor and is also ranging from 0-1, preferable closer to 1, to ensure that future actions matter.

In the box below, the algorithm for Q-learning can be seen. (Sutton R. S. 2018, pp. 131–136)

Algorithm: Q-learning

Algorithm parameters: step size: $\alpha \in$ [0,1], small ϵ > 0
Initialise $Q(s, a)$, for all $s \in S^+$, $a \in A(s)$, arbitrarily except that $Q(terminal, \dot{}) = 0$
Loop for each episode:
  Initialise S
  Loop for each step of episode:
    Choose A from S using policy derived from $Q(e.g., \epsilon - greedy)$
    Take action A, observe R, $S^{'}$
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
    $S \leftarrow S^{'}$
  until S is terminal

3.6.1 Definition of states

In order to perform Q-learning a definition of states must be made. These states must have the Markov property meaning that the probability of a marble being in a room must not depend on whether other rooms have been visited or not.

In order to achieve this, it has been chosen to implement a vector of boolean values, one element for each room. This will be used to store which rooms have been visited.
By doing this, all possible combinations will be possible and the reward of entering a room will not depend on the other rooms, only the room itself.

It has also been chosen to implement the state with an integer storing the room number and a boolean for storing whether the state is terminal or not. The definition of the state can be seen below.

```
qState
    int roomNumber
    std::vector<bool> roomsVisited
    bool isTerminal
```

Due to the definition of the states, the number of states would grow rapidly with increasingly higher number of rooms. The nature of the state definition gives the possibility to divide the state space into smaller matrices, and by adding a bit of logic the mapping between the matrices could be obtained.
By defining a base state as the room number and sub-states as whether specific rooms have been visited or not, the state space could be reduced to a matrix with the size $(rooms + 1) \times (rooms + 1)$ and depth of $2^{room} - 1$. The last combination would not be relevant because all rooms would have been visited. All states in this matrix would be a terminal state, meaning it would not make sense to move from there and therefore not make sense to update the Q-value of those states. This principle can be seen in figure (13), where the numbers over the matrices describes the vector in the states.

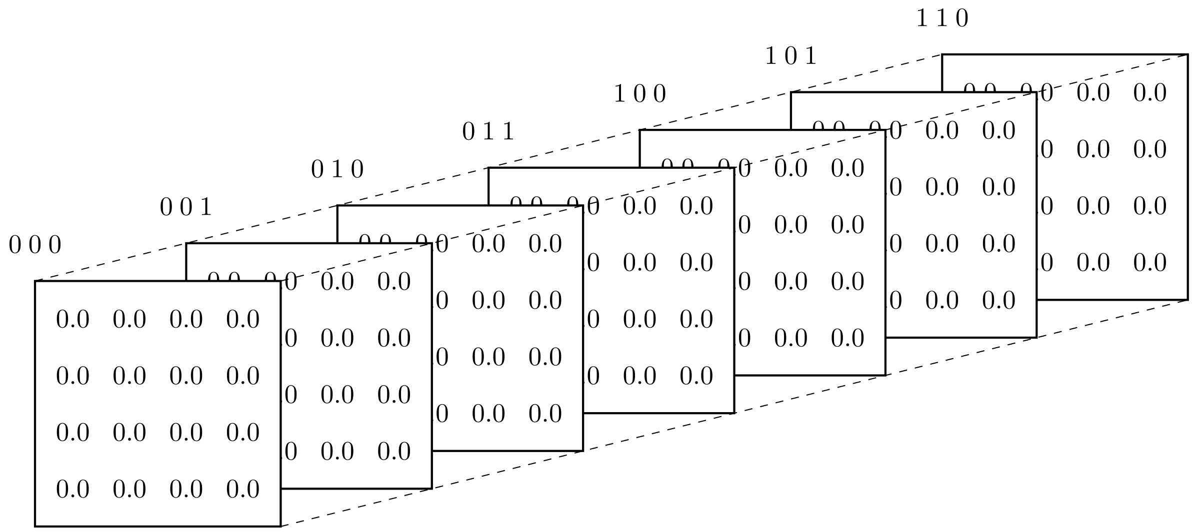


Figure 13: Illustration of how the matrices are layered

4 Implementation

4.1 LIDAR

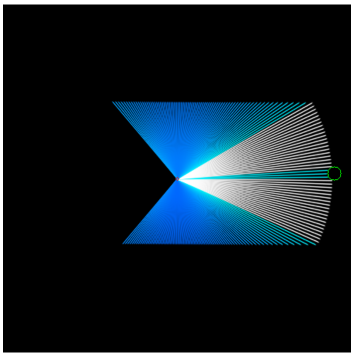
The implementation of the line and marble detection is done by designing a class called `lidar_sensor` containing the key methods: `find_marbles()`, `find_lines()` and `merge_lines()`. The first two key methods detect the marbles and lines in the image respectively. The third key method merges the lines on either side of a marble. The following two section describes the key methods in detail.

4.1.1 Marble detection

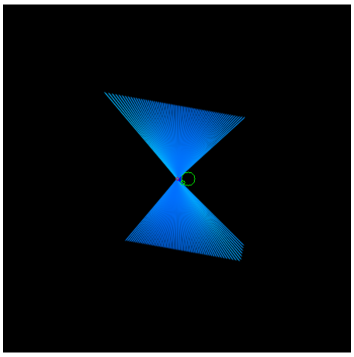
The marble detection method `find_marbles()` processes the datapoints by checking if a point satisfies either one of the marble conditions. If so, it breaks and calculates the parameters described in section 3.2.1.

One marble condition is a threshold for the range between two point and is defined to 0.2. This condition ensures that a marble can be detected from points from a partial circle periphery. Another marble condition is a range, that ensures that a marble can be detected in outer edges of the sensor range. This range is defined as the subtraction of marble condition one from the sensor range for the LIDAR sensor.

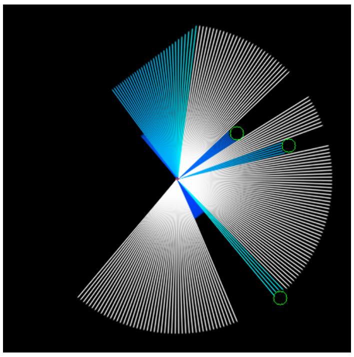
To test the performance and robustness of the algorithm a test with several repetitions were conducted. This test in described in appendix (A.1).



(a) Illustration of data for test 2



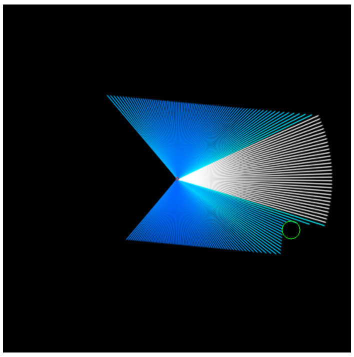
(b) Illustration of data for test 3



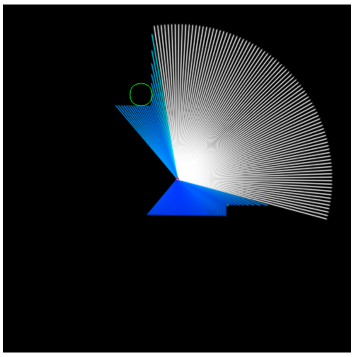
(c) Illustration of data for test 5

Figure 14: Illustration of data for marble detection tests

The marble detected in figure (50b) verifies that the marble condition holds, and that a marble can be detected in the outer edge of the detection area. The same applies for one of the marbles on figure (51a). Figure (51a) also shows that marbles can be detected anywhere in the sensor range. Sometimes the marble detection algorithm can detect an extra marble, where there is only one, as shown on figure (50c).



(a) Illustration of data for test 6



(b) Illustration of data for test 7

Figure 15: Illustration of data for marble detection tests

Figure (51b) and (51c) indicates that a marble can be detected in a corner, which might as well give problems. Due to the fact, that the robot will steer against marbles in the corners, the robot will hit an obstacle and tilt.

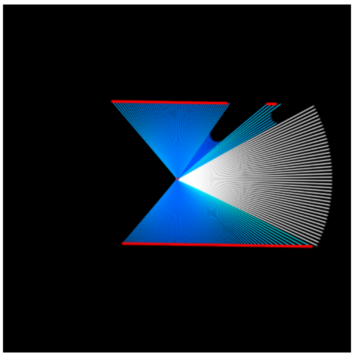
Due to this, the marble detection algorithm is not robust and do not have a good performance, since it only gives the expected outcome in some cases and in most cases detects marbles, where there is none.

4.1.2 Line detection

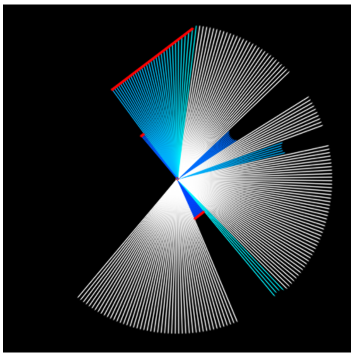
The line detection consists of the methods `find_lines()` and `merge_lines()`.
The method `find_lines()` processes the datapoints by using the incremental line extraction algorithm. This algorithm first computes a line model for 2 points by calculating the line parameter described in section 3.2.2. When it recomputes the line model every time an extra point is added. Before recomputing the line model, the line parameters are stored. This means that the algorithm constantly updates the line parameters for the current and previous line model.

Due to the fact, that the algorithm constantly updates the line parameters for the current and previous line model, it does not have to recompute the previous line model, when the current line model does not satisfy the line conditions.

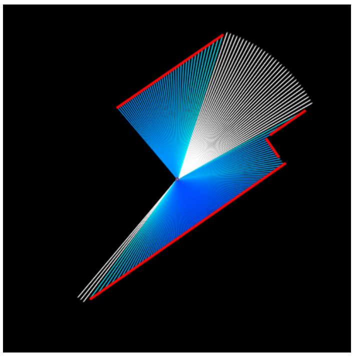
To test the performance and robustness of the algorithm a test with several repetitions were conducted. This test in described in appendix (A.2).



(a) Illustration of data for test 1



(b) Illustration of data for test 5



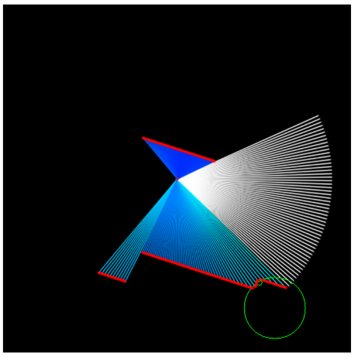
(c) Illustration of data for test 6

Figure 16: Illustration of data for marble detection tests

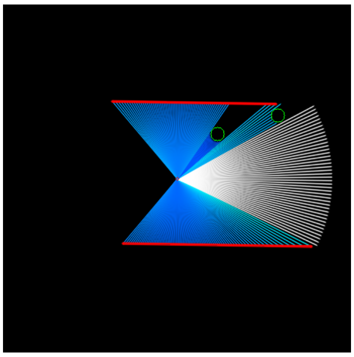
As mentioned in appendix (A.2), the method `find_lines` is robust and has a good performance, since it finds all obstacles such as walls, corners and doorframes. This is also shown in figure (16).

All the found lines in the area of the sensor range is stored for further processing by the line merging method `merge_lines()`. The method `merge_lines()` takes parameters from found lines, and check for two merging conditions The first condition is the angle between two found lines. This condition is called `threshold` and is defined to 0.1. The second condition is the area the marble provides shade. The algorithm does that by checking the difference in the angle from the end/start point of an found line to the closest line that strikes the marble. This condition is called `thresholdAlpha` and is defined to 0.5. The second condition prevents that two lines, that corresponds to doorframes, is merged. This means that the lines are not merged, if a marble is in front of a doorway to another room.

To test the performance and robustness of the algorithm a test with several repetitions were conducted. This test in described in appendix (A.2).



(a) Illustration of data for test 7



(b) Illustration of data for test 8

Figure 17: Illustration of data for marble detection tests

As mentioned in appendix (A.2), the line merging method `merge_lines()` is robust and has a good performance, since it merges the two lines, which is divided by a marble as shown in figure (17b).

The method depends on a marble detection method, which is not robust and do not have a good performance, since it detects corners as marbles as shown on figure (17a) and detects multiple marbles, when there is only one, as explained in section 4.1.1.

4.2 Camera marble detection

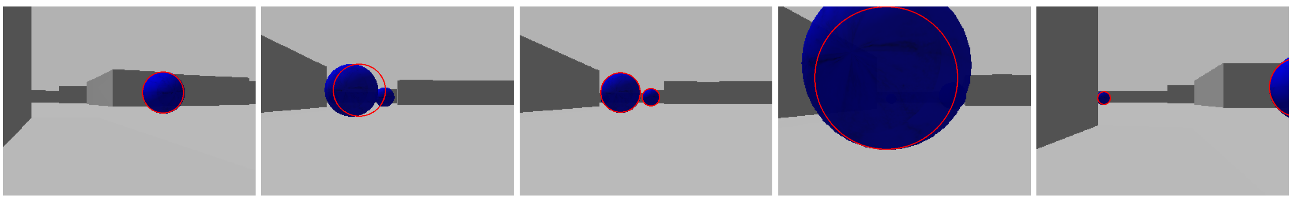
Marble detection on images from the camera was implemented with two main function `find_color()` and the method `find_marbles()`.
The first method `find_color()`, was implemented to separate the blue marbles from the grey background. This was done by converting the image from BGR to HLS with the OpenCV method `cv::cvtcolor(cv::Mat input, cv::Mat output, CV_BGR2HLS)`. Then the hue value of each pixel was compared to hue value of the marbles. If the colour matched the pixel would be white in the output image otherwise black. (Dawson-Howe K. 2014, pp. 18–20)
A test was conducted to find the hue value of the marbles, this test can be found in appendix (A.3). The hue value was to be 120. To find out whether this was sufficient a test was conducted to find the performance of this method. This test can be found in appendix (A.4). In the test is was found that no other criteria had to be considered in order to separate the marbles from the background. Based on the black and white images generated from this method, the method `find_marbles()` would detect the marbles.

The method `find_marbles()` was implemented by first finding the number of objects in the image. This was done by using the theory of connectivity, using the OpenCV method `cv::findContours(input_image, contours, hierachy, CV_RETR_TREE, CV_CHAIN_APPROX_NONE)`. This method returns the contour of objects found in the image, meaning that the method will return one object if two marbles where overlapping. (Dawson-Howe K. 2014, pp. 66–70)
Then max and min value of both the x and y direction was found. Based on these informations the height and width of the objects where found.
The center of the marbles was set in between min and max values on both axis, and the radius was found based on the height of the object.
To check whether the marbles was cut off at either the left or the right side, the height and the width of the objects was compared. If the width was smaller or equal to the height, the marbles was considered cut at either side. To find out which side the center of the marbles was checked. If the marbles where to the left of the center of the image, the marbles was considered cut off in the left side, and if the marbles where to the right of the center of the image, the marbles was considered to be cut off at the right side.
The center of the detected marbles would in these cases not be correctly placed, but shifted towards the middle of the image. To overcome this the difference was found by calculation the difference between the edge of the object, and the edge of the detected marble. The marbles center could then be shifted either left or right with value.

A final test was conducted to find the performance of the methods. The test was conducted on a total of five different scenarios, to show the performance of the algorithm. The test consisted of the following scenarios.

- One fully visible marble
- Two overlaying marbles
- Two non-overlaying marbles
- One marble cut off at the top
- Two marbles cut of on the sides. One on the left and one on the right

The result from the test can be seen in figure (18), where the algorithm performed perfect in three out of five cases. In the last two cases the detected marbles where either different in radius than the actual marble or had shifted center, while missing one marble.



(a) Marble detection on scenario 1 *(b) Marble detection on scenario 2* *(c) Marble detection on scenario 3* *(d) Marble detection on scenario 4* *(e) Marble detection on scenario 5*

Figure 18: Marble detection of scenario 1 to 5

These issues was however not considered critical, because the one on figure (18d) will only occur when a marble is very close. The direction the robot needs to follow in order to collect the marbles would not be influenced by this error, only how close marble is.
The second issue, seen on figure (18b), could be a potential for the robot, due to the fact that the center of the marble are shifted. But as the robot moves closer the two marbles would either drift further apart enabling the algorithm to detect both, or the second marble found drift in behind the first one.

4.3 Motion planning

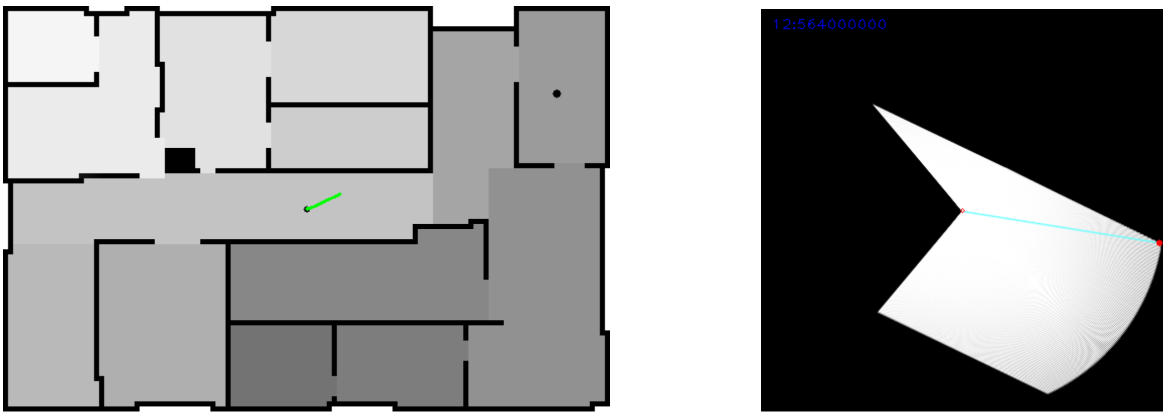
A number of methods have been used in the creation of the two planers. A short description of the methods used and the approach is giving in this section as well as tests analysis of the two planners in action.

4.3.1 Sensor based planner

The tangent bug algorithm have been implemented in a class called `motion_planning` containing the methods.

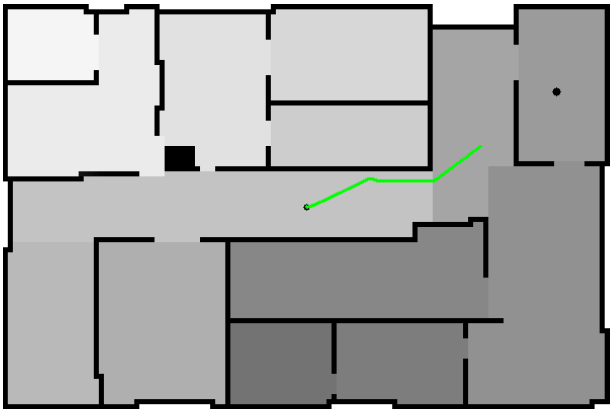
- `tanget_bug_algorithm(ct::current_position pos, std::vector<ct::room> Rooms)`
- `homogeneous_transformation(ct::current_position robot, cv::Point goal)`

This methods uses fuzzy logic to controlled the robot as explained in section 3.5. The `tanget_bug_algorithm(p,R)` takes the current position of the robot and a vector of room locations as inputs. Also this method gets the information about the shortest distance to obstacles from the robots LIDAR sensor which must be known for the robot to navigate along obstacles and avoid hitting them. The method `homogeneous_transformation(r,g)` takes the current position of the robot and a goal location as inputs. This method is critical for the implementation of the tangent bug algorithm because it returns an orientation relative to the robot to the goal location. This method is actually used twice in the `tanget_bug_algorithm(p,R)` because both the orientation to the goal location as well as the orientation towards the closest obstacle must be known to be fetch by the fuzzy controller so that the robot can be giving the proper motion to reach the target position.

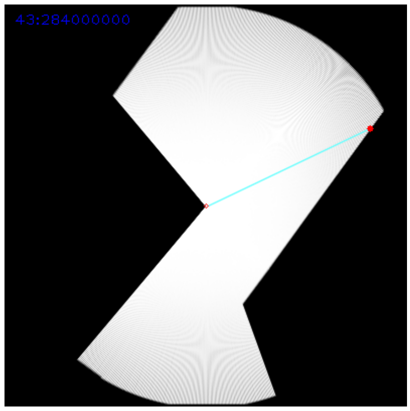


(a) The robots path from a initial position to a goal *(b) Closest boundary on an obstacle to goal*

Figure 19: Tangent bug test 1

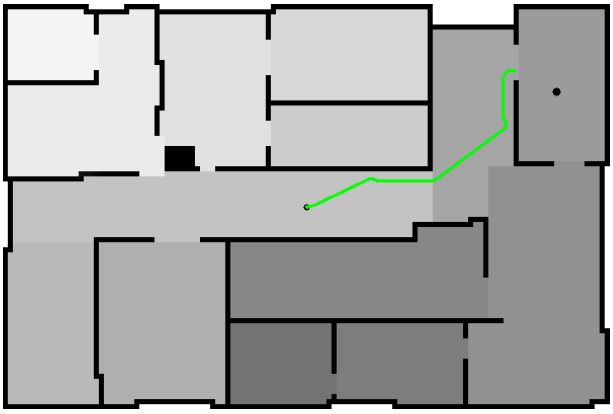


(a) The robots path from a initial position to a goal

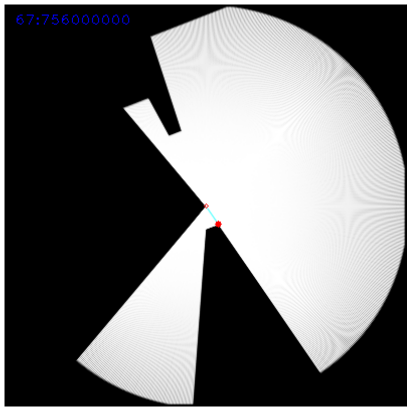


(b) Closest boundary on an obstacle to goal

Figure 20: Tangent bug test 2



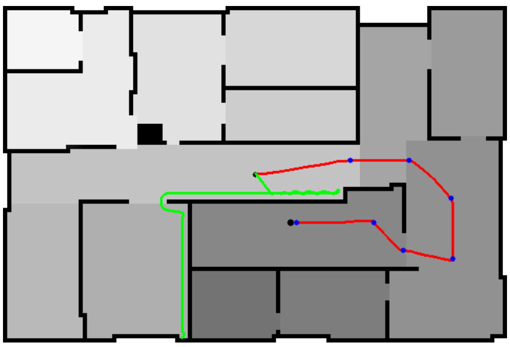
(a) The robots path from a initial position to a goal



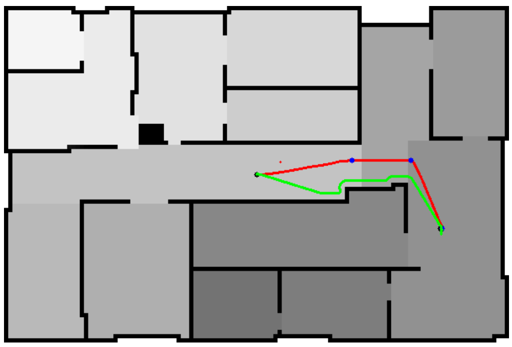
(b) Closest boundary on an obstacle to goal

Figure 21: Tangent bug test 3

As mentioned in the design phase of constructing the tangent bug algorithm the robot follows the obstacle which minimizes the distance from the robots position to the boundary on the obstacle and from that point on the boundary to the goal. Figure (19a) shows the path of the robot from an initial position to the target. Figure (19b) illustrates that point on the obstacle that minimises the complete path to the target if the robot is in an obstacle following behaviour. The boundary following behaviour only happens if an obstacle is blocking the way towards the goal. A number of test were conducted on the tangent bug algorithm and thereby the fuzzy control to test the implementation. The test can be seen in appendix (A.6).



(a) The robots travelling path from the origin to the target



(b) The robots travelling path from the origin to the target

The test showed that through a number of runs the average distance from the closest obstacles was 1.6 meters with a success rate of finding rooms at 64.3 %. From figure (22a) a scenario in which the tangent bug fails to reach a target location can be seen. The bug algorithm simply gets stuck in a corner, where the fuzzy controller also seems to have some limitations. The green line indicates the path of the sensor based planner and the red line the model based planner. On the other hand in figure (22b) it performs quite well and follows the obstacles as intended. It is also worth noting that when the tangent bug finds its target location, it outperforms the model based planner regarding distance travelled. The closes distance to an obstacle was found to be 0.4 meters which satisfies the specification requirements.

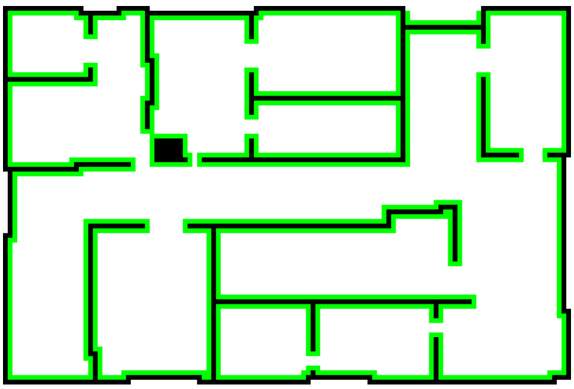
4.3.2 Model based planner

The method `brusfireAlgorithm(i)` takes and input as the number of iterations one wants to process. After doing so it is possible to get the pixels furthest away from an obstacle and thereby the center points of rooms and doors.

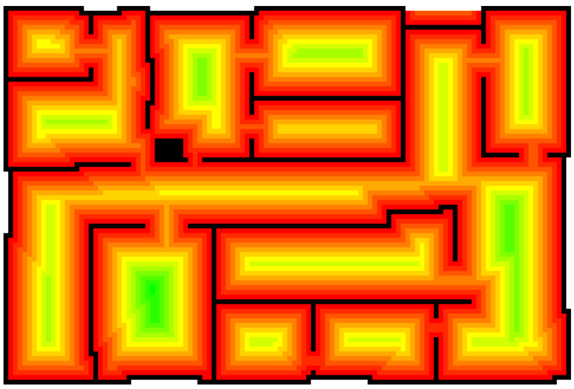
The implementation of the model based planner is implemented in the classes `motion_planning` and `brushfire` contain the key methods.

- `brushfireAlgorithm(int iterations)`
- `findCornerPoints()`
- `findCenterPoints()`
- `findLinePoints()`
- `ConnectPoints()`
- `findPathPoints(cv::Point curPos, cv::Point goal)`
- `find(int V)`
- `unionSets(int V1, int V2)`
- `DFS(cv::Point start, cv::Point goal)`

The classes uses fuzzy logic to control the robot. The method `brusfireAlgorithm(i)` takes and input as the number of iterations one wants to process. After doing so it is possible to get the pixels furthest away from an obstacle and thereby the center points of rooms and doors.



(a) Illustration of brushfire in use with 1 iteration

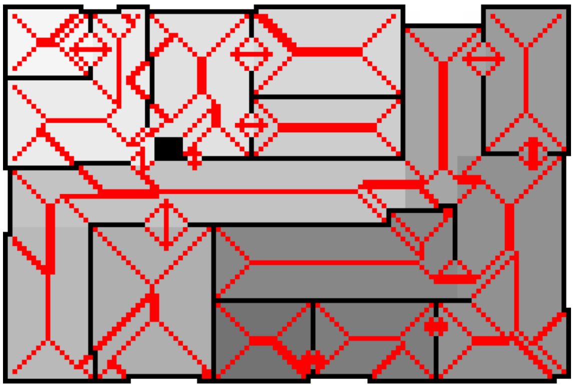


(b) Illustration of brushfire in use with 13 iteration

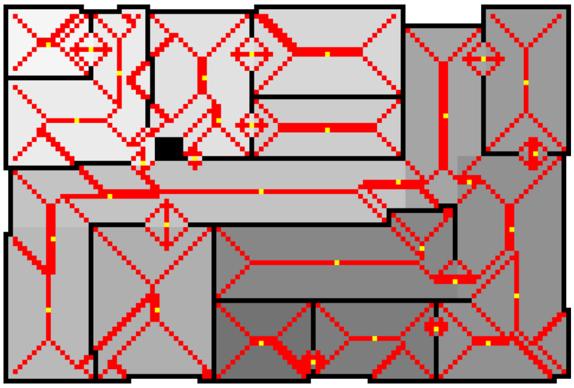
As seen in figure (23a) and figure (23b) the algorithm start by consuming all free pixels closest to the obstacle and iterate through the map. For visualisation the pixels furthest away from the obstacle, which have been assigned a value, are pictured green and the once closest to an obstacle are red. This gives a map with connections between rooms.

Now it is possible to find corner and center points via the newly created brushfire map. For this to be accomplish this two methods were used namely `findCornerPoints()` and `findCenterPoints()`. The idea is to use four-point connectivity by comparing $pixel(i, j)$ to its neighbours. For instance if $pixel(i-1, j)$ equals $pixel(i+1, j)$ and the center point between them, $pixel(i, j)$, has a different value, it is considered to be a point on a line. The same thing goes for the horizontal case. To find corner points, the $pixel(i-1,j)$ must equal to $pixel(i,j+1)$ and $pixel(i-1,j+1)$ must be different from $pixel(i,j)$. This is just one of the instances of find corner points but the principles is the same for the four cases.

After completing this operation on the brushfire map it is possible to see corners and center lines in rooms as well as doors. This i visualised in figure (24a).



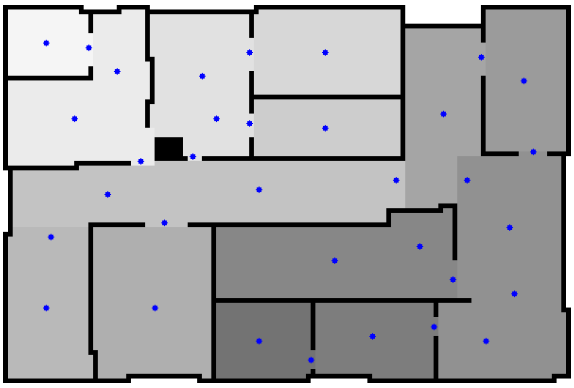
(a) Illustration of findCornerPoints and centerPoints in use



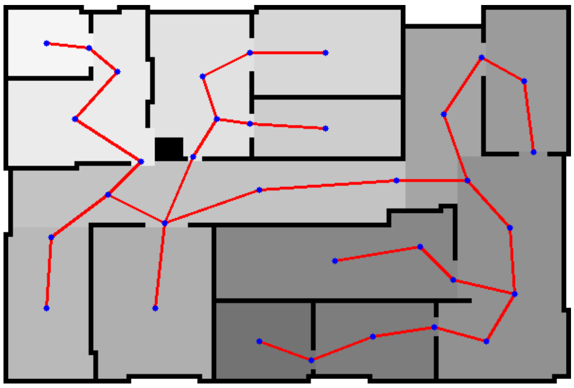
(b) Illustration of findLinePoints in use

It is quite interesting to see the amount of possible points that can be used to make connections between rooms after using the method just mentioned. Because the goal is to make full connections between all rooms, the vertical and horizontal lines in figure (24a) is going to be taken into consideration as it will satisfy the objective of finding a path between all points without hitting an obstacle. The method `findLinePoints()` was created to solve this task. The method iterates through a matrix containing all values on the map and pushes the found line points on a vector and then saves the center point on that detected line. It is worth noting that there are horizontal as well as vertical lines. This will lead to a set of lines points as shown in figure (24b).

Now in the creation of the road map every point is considered to be a vertex and a connection between two vertices is an edge. Now the interesting part is to find out which vertices that can be connected to one another without hitting an obstacle. Because all obstacles have been assigned the value of 1 it is possible to iterate through the map and see if $pixel(i,j) == 1$ is located somewhere on the edge between these two vertices. If that is not the case, the edge is considered valid and saved as an edge and thereby a possible connection between two vertices. This is done in the method `connectPoints()`. It is critical to point out that Kruskal's algorithm is used in this method as it is a big part of the procedure of connecting the vertices by the shortest edge first. Kruskal's algorithm was discussed in section 3.4.2.



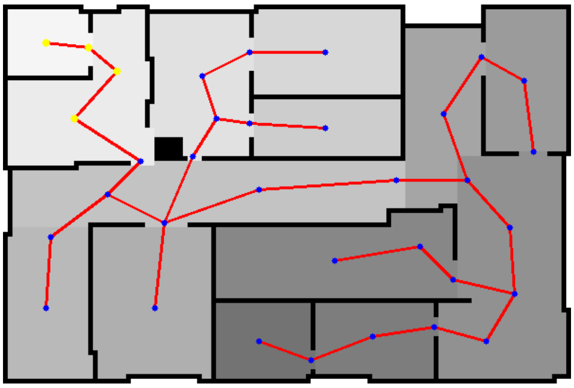
(a) Illustration of the vertices on the map



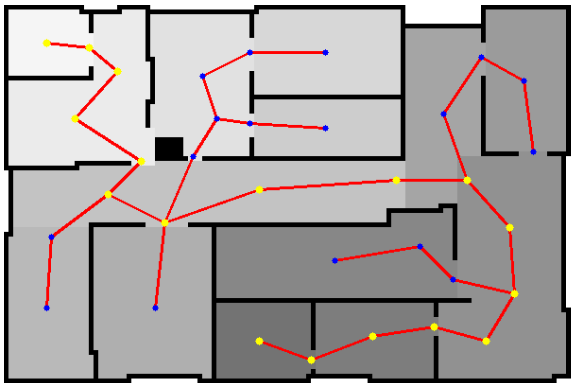
(b) Illustration of the connected map between vertices via edges

One can see all the vertices in figure (26a) and there connections to one another after using the method `connectPoints()` in figure (26b). Notice that the edges between vertices do not contain any cycles which is one of the big advantages of using Kruskal's algorithm and thereby making it easier to make a path between vertices.

Now the method `findPathPoints(c,p)` can be used to generate a map from an initial position to a target location giving the two points as parameters. `FindPathPoints(c,p)` uses the methods `find(int V)`, `unionSets(V1,V2)` and `DFS(s,g)` which are already discussed in section 3.4.2. The outcome of this method is a set of points starting from an initial position to a target location.



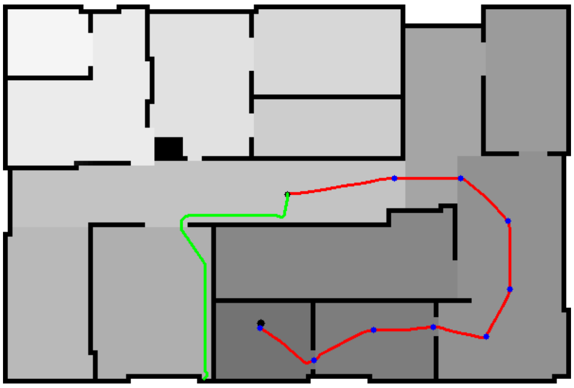
(a) Illustration of a path between 3 edges



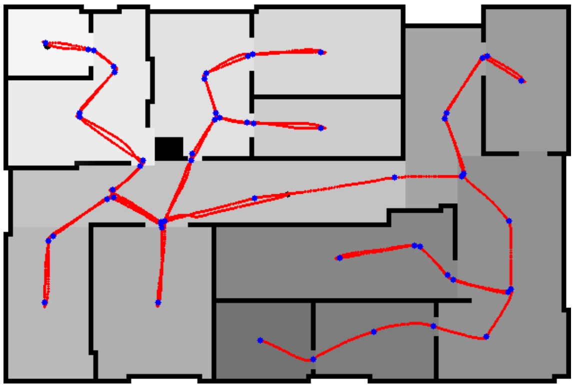
(b) Illustration of a path between 16 edges

Now the method `model_based_planner(p,p,i,i)` can be called using the robots current position, a vector of points which the robot have to follow, an index of the vector of points as well as an index to the wanted room number.

Several test were made to see how well the model based planner worked, this test can be seen in appendix (A.6). First of all 14 test was conducted to see if the robot was able to find all the rooms in the bigworld map starting from the origin of the map. The average success rate of this test was 100% meaning that it found a way to all 14 rooms. The test also showed that the average distance to obstacle was 2.345 meters. Another important factor was that it was able to run through the entire map starting from the origin and then to all rooms from room 1 to 14 which can be seen in figure (27b).

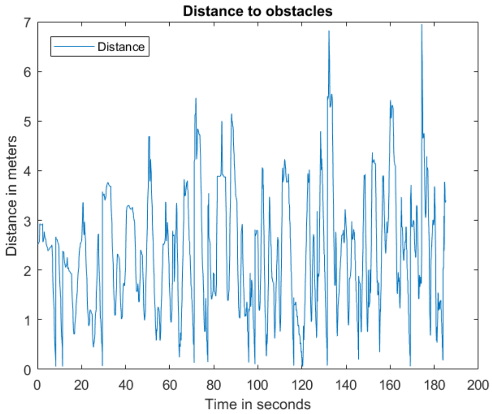


(a) Path from origin to room 14. Red line for model based.

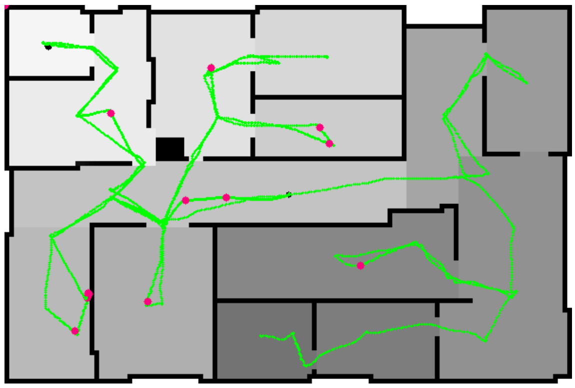


(b) All rooms visited with model based planer

Figure (27a) shows the working of the brushfire algorithm going from the origin to room 14 marked with a red path. Here the green path is the test of the sensor based planer. One can see that the model based planer follows the graph nicely and ends up in the target location.
Another test using the model based planner was performed to test the fuzzy control and see if the robot was able to collect marbles, this test can be seen in appendix (A.7). The was concluded that the fuzzy control works and that the robot was able to find marbles where on of the test resulted in a full search of all rooms with 10 marbles collected as seen in figure (27b). But unfortunately Gazebo crashed several times while collecting marbles which affected the results.



(a) Distance to the closest obstacles



(b) Illustration of marble collection with red dots as marbles.

In figure (28a) one can see when a marble is collected when the spike of a distance to an obstacle is very close to zero.

4.4 Search strategy

To be able to implement the fuzzy controller defined in the design phase fuzzylite has been used. Fuzzylite is a library for fuzzy control logic and it has all the necessary tools for the implementation. The rule block uses Mamdani-style inference which require us to find the two-dimensional shape by integrating across a continuously varying function. Here the minimum triangle norm (T-norm) is used and the center of gravity (COG) as defuzzification which was found suitable. A class called `fuzzybugcontroller` was made for the implementation with methods `getControlOutput(ct::marble marble_input, ct::robot_orientation angle)` and `buildController()`. The `buildController()` initialises the controller by loading the membership functions and rule base for the fuzzy controler to follow. The `fuzzybugcontroller` gets information about distances to obstacles through an object called `pc_laser_scanner` which is initialised in the constructor of the fuzzybugcontroller. The object `pc_laser_scanner` is of type `laserscanner` which is a class used to get data from the laser scanner of the robot. The method `getControlInput(m,a` have two parameters where information about marbles and the current position of the robot are giving as inputs. This information along with the knowledge of the distance to obstacles are then giving to the controller which analyses the data and returns the giving action for the robot to follow. Analyses of how well the fuzzy controller works is documented in section 4.3.1 and 4.3.2.

4.5 Q-learning

Q-learning was implemented by designing a set of key methods.

- `getNextState(qState s, float a)`
- `qUpdate(qState s, float` α`, float` γ`, float` ϵ`)`
- `visitRoom(qState s)`
- `getNextAction(qState s)`
- `findStateMatrixIndex(qState s)`
- `findQMatrixIndex(qState s)`
- `setDistancePunishment(qState s1, qState s2, float p)`
- `setReward(int roomNumb, float reward)`

All these methods was implemented in a class called `q_learning`. This gives the possibility of initialising all matrices and parameters in the constructor, only given the number of rooms as argument. This makes code scalable to any number of rooms.

It has been chosen to initialise all rewards to -100 except those on the main diagonal, whose initial value was set to 0. Calling the methods `setDistancePunishment(s1, s2, p)`, and `setReward(roomNumb, reward)` afterwards would update the total reward .
An example of how that would look like can be seen in table (1), where all distance punishments are based on those found in the test in appendix (A.8) multiplied by a factor of 1.2. The rewards for entering a room was found in the test in appendix (A.9), and divided by the max value and then multiplied with a factor of 20. The initial room was set to 3 in this example.

	Start	**Room 1**	**Room 2**	**Room 3**	**Room 4**	**Room 5**
Start	0	-100	-100	13.33971	-100	-100
Room 1	-100	0	8.681309	-100	-100	-100
Room 2	-100	0.405394	0	10.74771	-100	-100
Room 3	-100	-100	7.757309	0	7.837891	17.456
Room 4	-100	-100	-100	11.01171	0	-100
Room 5	-100	-100	-100	10.79571	-100	0

Table 1: Rewards for all state-action combinations given that no rooms have been visited and initial state is room 3

The vector `stateMatrix` contains the rewards, and it was chosen to keep this matrix dynamic, so only the combinations needed was added to the vector instead of all possible combinations, unlike the vector that contains the `qMatrix`. This was done because it makes no difference whether the state matrix only applies for one episode or all combinations, as long as the matrix is reset each time a new episode starts. Doing this saves memory.

The method `setDistancePunishment(qState s1, qState s2, float p)` inserts a distance punishment between two states, by indexing the matrix with the room numbers of the two states. The indices are then reversed to apply the punishment in both directions.

The method `visitRoom(qState s)` are one of the primary methods behind the mapping between the different layers in the Q-matrix. This method updates the vector in states by changing the value to true for the current room.
Then it is checked whether the state is terminal or not and updates this information in the state.
If the room does not appear in the list of visited rooms, it is then added and a new reward matrix will be generated based on the new states. The new state is then returned from the method.

The method `findStateMatrixIndex(qState s)` is used to find the index for the `stateMatrix` containing all the rewards. The method works by comparing the values in the vector from the state with the values corresponding to the order in the state matrix. When a match is found, the corresponding index will be returned.
Due to the fact that is was chosen to make the `stateMatrix` dynamic, a second version of this method was written to find the index for the `qMatrix`. This method was called `findQMatrixIndex(qState s)`.

The method `getNextState(qState s, float a)` returns the next state, given the current state and action. This simple method replaces the room number with action (new room) and returns the state.

To get good performance and good results a series of tests were conducted to find good values of α, γ and ϵ. All the following tests were made using the world 5-room world, which can be seen on figure (29).

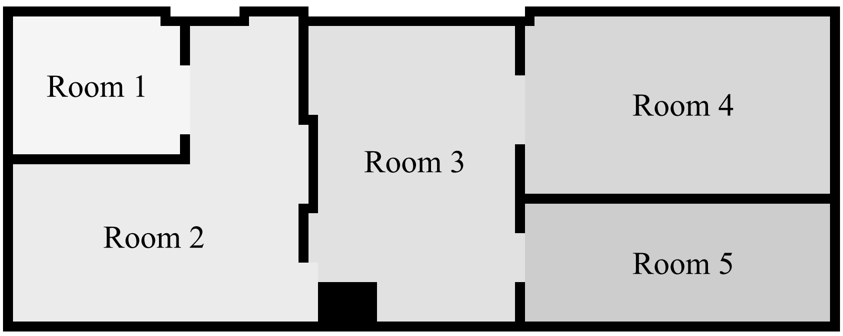


Figure 29: Illustration of the world 5-room world used for the test on Q-learning

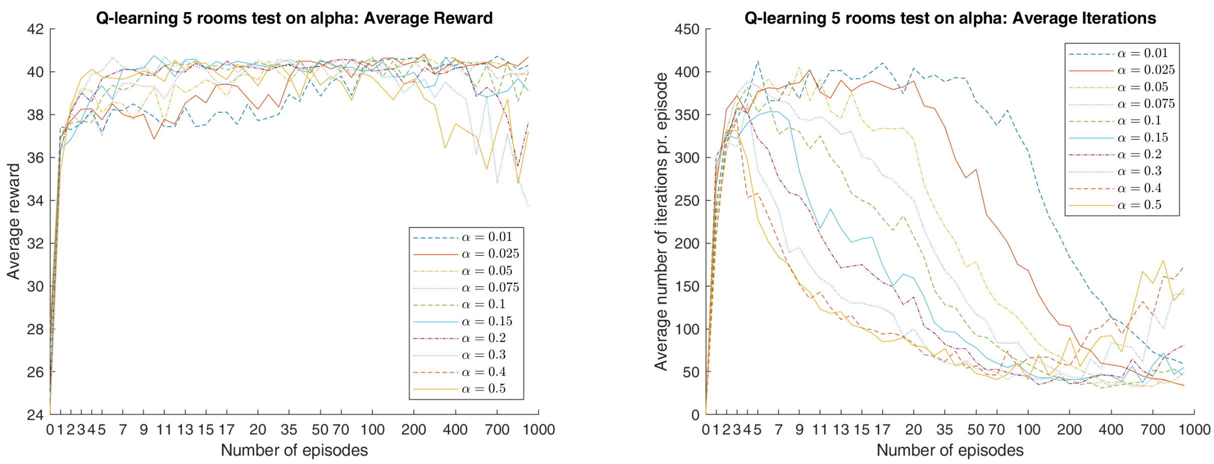
The first test conducted was to show the influence of ϵ and chose an appropriate value for future tests. This test can be seen in appendix (A.10).
In figure (30) the results from the test can be seen. It can be seen that there is a clear connection between the value of ϵ and the performance of the Q-learning algorithm. The higher the value of ϵ is the more random the agent acts and more lower is the average reward after a given number of episodes. The average number of iterations per episode falls drastically with increasingly larger value of ϵ. Based on the test, the best value of ϵ was chosen to be 0.05, due to the fact that this value gives a good average reward while taking fewer iterations, than the slightly better performing ($\epsilon = 0.01$).

(a) Plot of how the average reward develops as a function of the number of episodes for each value of ϵ

(b) Plot of how the average number of episodes develops as a function of the number of episodes for each value of ϵ

Figure 30: Plots of both the average reward and average number of iterations pr. episode for each value of ϵ

The next test conducted was to show the influence of the learning rate α. The test can be seen in appendix (A.11).
In figure (31) the result from the test can be seen. From the test results it can be seen that with a decreasing learning rate the average reward increases and the longer the number of iterations keeps high. This means that the lower the learning rate the longer it will take to learn, but what it have learnt would be more optimal than with higher values of α. Based on the test a value of α was chosen to be 0.025.

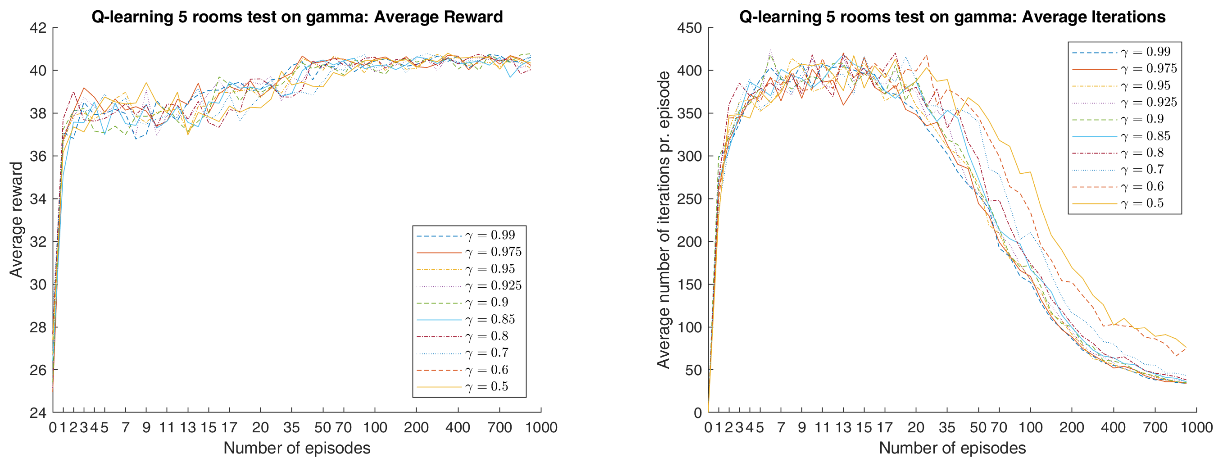


(a) Plot of how the average reward develops as a function of the number of episodes for each value of ϵ

(b) Plot of how the average number of episodes develops as a function of the number of episodes for each value of α

Figure 31: Plots of both the average reward and average number of iterations pr. episode for each value of α

The final parameter was tested in the following test. The purpose of this test was to show the influence of the discount factor γ. This test can be seen in appendix (A.12). In figure (32) the results from the test can be seen. From the results it can be seen that the higher the value of γ the fewer iterations are needed to perform an episode, and the less varying the average reward will be. Given the test a value of γ was chosen to be 0.99.



(a) Plot of how the average reward develops as a function of the number of episodes for each value of ϵ

(b) Plot of how the average number of episodes develops as a function of the number of episodes for each value of α

Figure 32: Plots of both the average reward and average number of iterations pr. episode for each value of γ

Based on those three test, a good set-up was found for the final test on getting a good path. It has also been chosen to conduct a test on finding an optimal path on the 5-room world. This test can be seen in appendix (A.13). In this test, the parameters used were found in the previous tests. A total of 5 subtest were made in order to find an optimal path for navigating the 5-room world. The agent was started in all rooms, and had to learn its way through the environment. The agent was given 2000 episodes to learn, before returning the learnt path. This was done ten times for each room to get an idea of the average case. Based on this test, the best average case was when the agent started in room 1, but the best path was found with start in both room 1 and 5. These paths can be seen in table (2).

Initial room	**Path**	**Reward**
1	0, 1, 2, 3, 5, 3, 4	44.4681
5	0, 5, 3, 4, 3, 2, 1	44.4681

Table 2: The two best paths through the 5-room world found using the `q_learning` class

This test would be scalable to larger environments, but test on higher number of rooms was not conducted due to time constrains.

5 Discussion

The reason the line detection algorithm was not used, was that the marble detection algorithm was not properly performing at times. This means that the merging part of the line detecting algorithm would merge lines that should not have been merged. Threshold on the angle between two lines could prevent this behaviour to some extent if this threshold was tuned. The main issue was the falsely detected marbles. This was due to the marble detection algorithms tendency to detect marbles in corners. This could have been prevented by detecting how good the points fitted the found circles and by setting a threshold on the radius of the circles.

Due to time constrains the motion planning was not coupled together with q-learning. If this had been the case, the robot would have been able to search for marbles in rooms in the most optimised way from reinforcement learning.

The marble detection on camera data gave falsely results when two marbles were overlapping or when a marble was cut off on either the top or bottom. To solve these issues it could be checked whether or not white pixels would appear in the top or bottom row in the binary image if the detected objects are wider than they are tall. If this is not the case it is a possibility that the detected object are two overlapping marbles. These issues were not addressed due to time constraints.

6 Conclusion

A line detecting algorithm was developed to detect obstacles and marbles from the LIDAR data. It can be concluded that the algorithm was able to distinguish marbles from obstacles and the algorithm performed well in detecting lines and corners, but often falsely detecting marbles in corners. This lead to the general limitations of the algorithm where a marble detected in a corner would result in two perpendicular lines being merged.

This algorithm performed good in general, but has some limitations when obstacles were blocked by a marble. This happened when marbles were detected in corners resulting in to perpendicular lines being merged.

The marble detecting algorithm based on camera data performed quite when one or several none cut off marbles, as well as marbles where cut off on either the left or right side. Marbles cut off on either top or bottom caused issues with a falsely detected radius. When marbles were overlapping the algorithm was only capable of detecting one of the marbles and placed the center slightly shifted towards the smaller overlapping marble.

Both sensor and model based algorithms were made as planning algorithms for the robot. The tests showed the sensor based planer had an average success rate of 64.3% of finding rooms compared to the model based on 100%. On average, the sensor based planner would have an 40.47% shorter path. The shortest distance to obstacles was 0.258 meters for the model based planner and 0.4 for the sensor based planner which satisfied the specification requirements.

For robot control a fuzzy controller was made. Tests showed that the fuzzy controller was able in controlling the robot around the environment as well as avoiding obstacles and finding marbles.

Q-learning was applied to optimise the search strategy for an environment with 5 rooms. The test showed that that a solution was found in all cases and that the planned path got better for each episode. It can be concluded that a path was found only consisting of 6 movements where only one room was visiting more than more time.

References

Choset H. Lynch K. M., Hutchinson S. Kantor G. Burgard W. Kavraki L. E. & Thrun S. (2005). *Principles of Robot Motion: Theory, Algorithms & Implementation*. (1th Edition). The MIT Press.

Dawson-Howe K. (2014). *A Practical Introduction to Computer Vision with OpenCV*. (1th Edition). Wiley.

Munoz L. R., Villanueva M. G. & Suarez C. G. (2014). *A tutorial on the total least squares method for fitting a straight line and a plane*. `https://www.researchgate.net/publication/272179120_A_tutorial_on_the_total_least_squares_method_for_fitting_a_straight_line_and_a_plane`. Accessed: 01-12-2018.

Nguyen V. Martinelli A., Tomatis N. & Siewart R. (2005). *A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics*. `https://ieeexplore.ieee.org/abstract/document/1545234`. Accessed: 01-12-2018.

Sutton R. S., Barto A. G. (2018). *Reinforcement learning: An introduction*. (2nd Edition). Westchester Publishing Services.

Weiss M. A. (2014). *Data Structures & Algorithm Analysis C++*. (4th Edition). Pearson.

Appendices

A Tests

In this section all tests conducted will be described.

A.1 Robustness and performance of LIDAR marble detection

The purpose of this test is establish the robustness and performance of the marble detection algorithm for the LIDAR sensor.

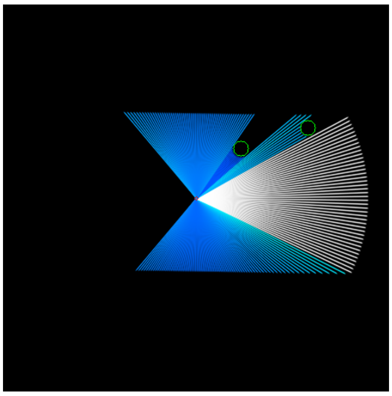
Description of test

The initial position of the robot for this test is the origin of the environment bigworld. Here the robot is steered around in the environment using the keypad. The idea of this test is to test if all marbles detected by the algorithm actually corresponds to marbles in the environment.

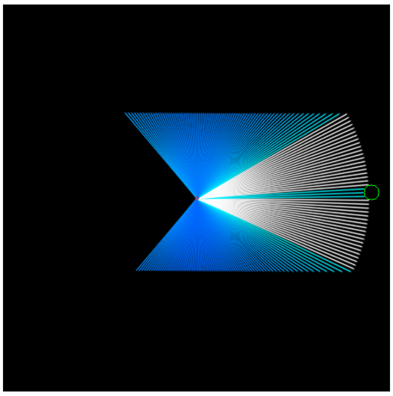
Test parameters

- World used	bigworld
- Number of spawn marbles	20
- Number of tests	9
- Range between two points	0.2
- Range to marbles in outer edges of sensor range	9.8

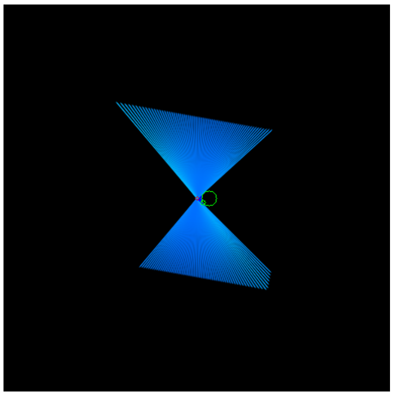
Data



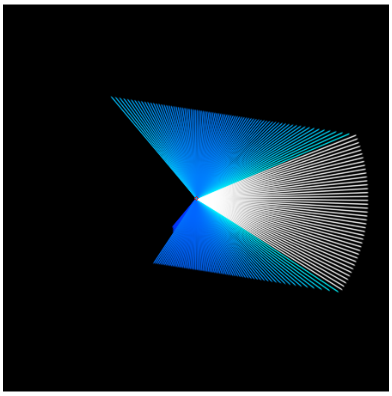
(a) Illustration of data for test 1



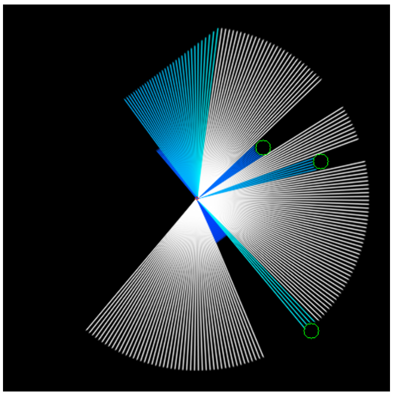
(b) Illustration of data for test 2



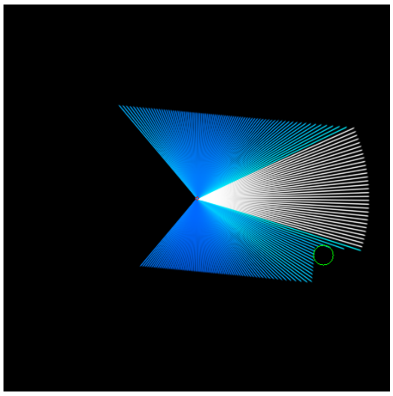
(c) Illustration of data for test 3



(d) Illustration of data for test 4

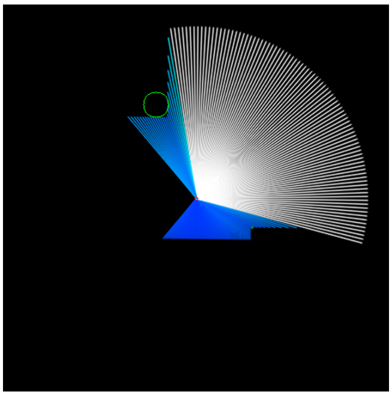


(e) Illustration of data for test 5

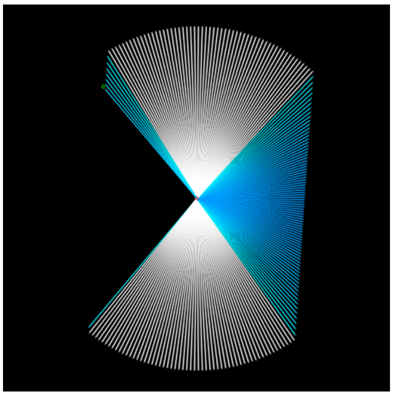


(f) Illustration of data for test 6

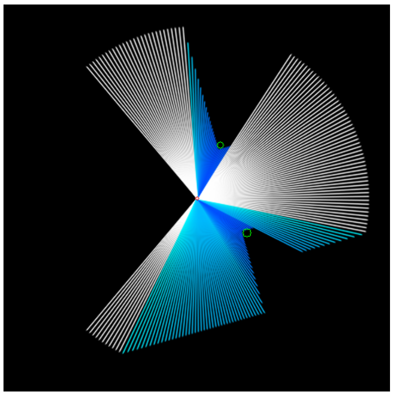
Figure 33: Illustration of data for marble detection test 1-6



(a) Illustration of data for test 7



(b) Illustration of data for test 8



(c) Illustration of data for test 9

Figure 34: Illustration of data for marble detection test 7-9

Conclusion

It can be concluded that the marble detection algorithm is not robust and does not have a good performance, since it sometimes detects more than one marble, marbles in corners and marbles of different size.

A.2 Robustness and performance of LIDAR line detection

The purpose of this test is establish the robustness and performance of the line detection algorithm for the LIDAR sensor.

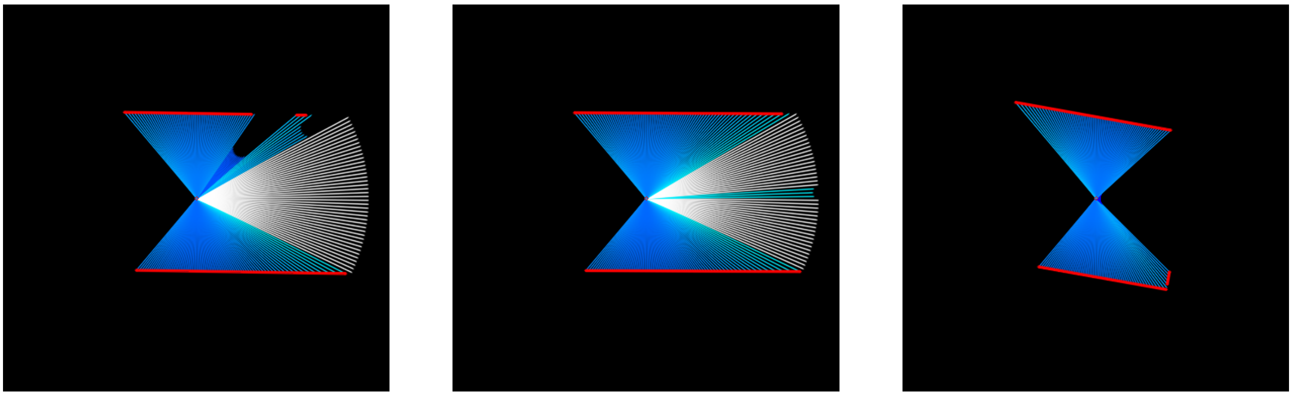
Description of test

The initial position of the robot for this test is the origin of the environment bigworld. Here the robot is steered around in the environment using the keypad. The idea of this test is to test if all marbles detected by the algorithm actually corresponds to marbles in the environment.

Test parameters

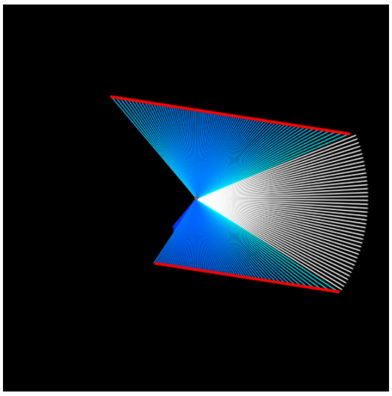
- World used	bigworld
- Number of spawn marbles	20
- Number of tests	9
- Angle between two found lines	0.1
- Angle from start/end point to closest line that strikes the marble	0.5
- Angle between previous and current linemodel θ_{max}	0.0025
- Angle between two points relative to the robot $\Delta\theta$	$(\theta_0 - \theta_1) \cdot 1.25$

Data

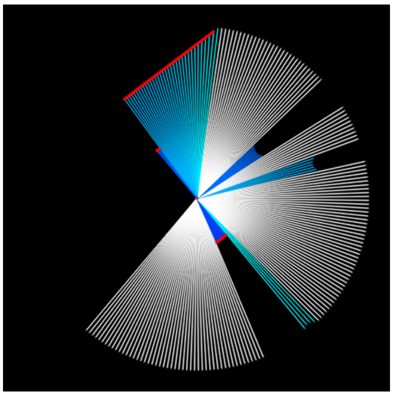


(a) Illustration of data for test 1 *(b) Illustration of data for test 2* *(c) Illustration of data for test 3*

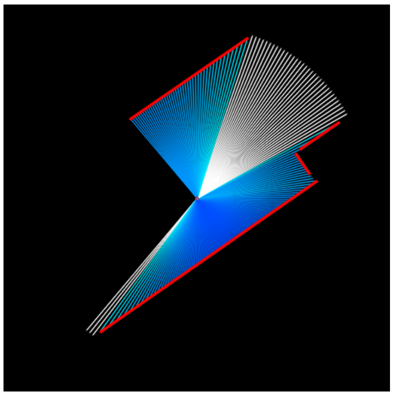
Figure 35: Illustration of data for line detection test 1-3



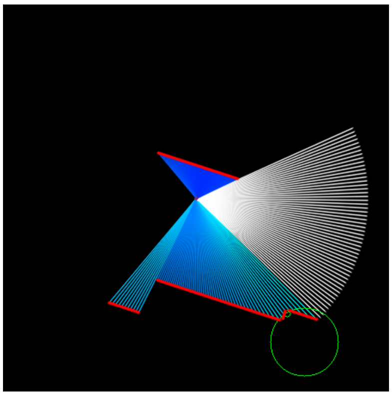
(a) Illustration of data for test 4



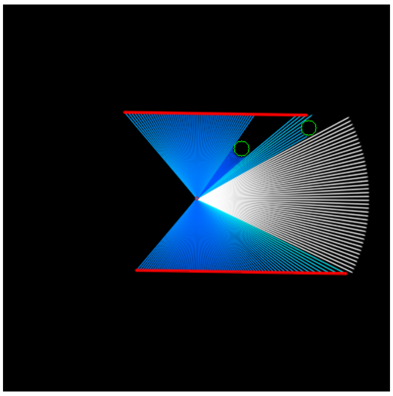
(b) Illustration of data for test 5



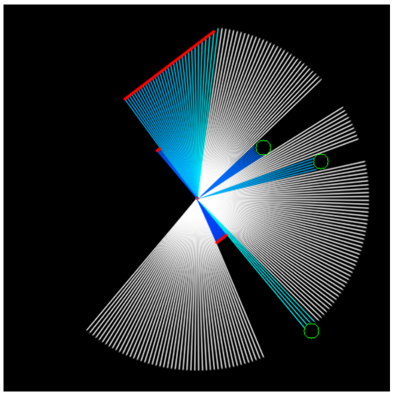
(c) Illustration of data for test 6



(d) Illustration of data for test 7



(e) Illustration of data for test 8



(f) Illustration of data for test 9

Figure 36: Illustration of data for line detection test 4-9

Conclusion

It can be concluded that line detecting part of this algorithm is robustness and has a good performance, since it detects all obstacles in all cases.

It can also be concluded that the line merging part of this algorithm is robust and has a good performance, but it relies on the marble detection algorithm, which is the opposite.

A.3 HLS Histogram

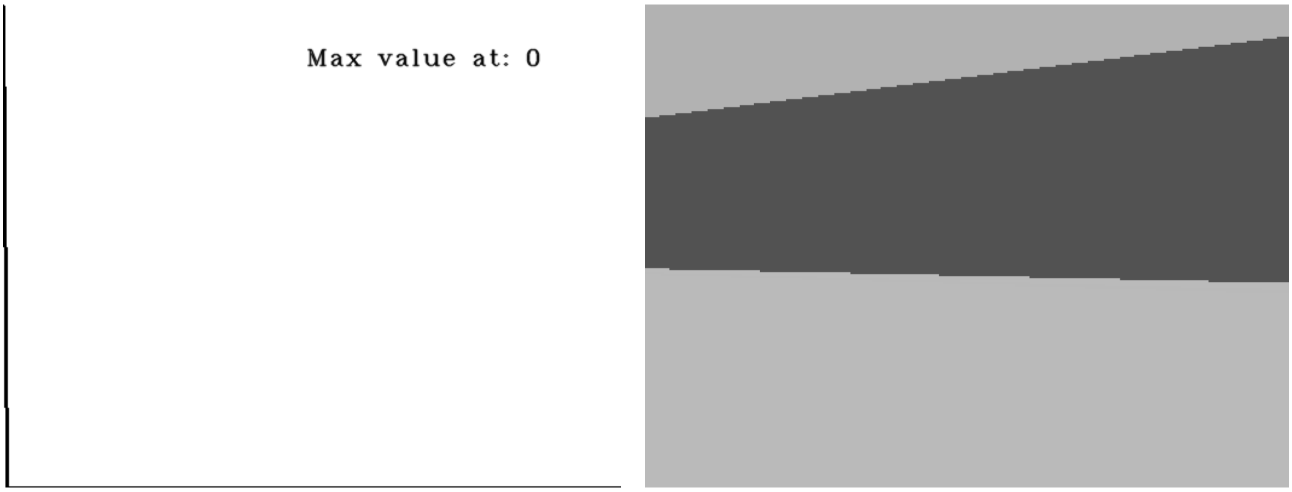
The purpose of this test was to find the hue value of the blue marbles, in order to convert the image to a binary image.

Description of test

This test was conducted using the method `hls_histogram()`, from the class `c_vision`. This method generates a histogram for the hue channel of the HLS colour-space, and marks local maxima.
The test was conducted using three different scenarios, one where no marbles would appear, one where a marble would fill a small part of the image, and one where a marble would will most of the image.
This was done to ensure that the right spike was found, by having knowledge of its peak compared to the background.

Data

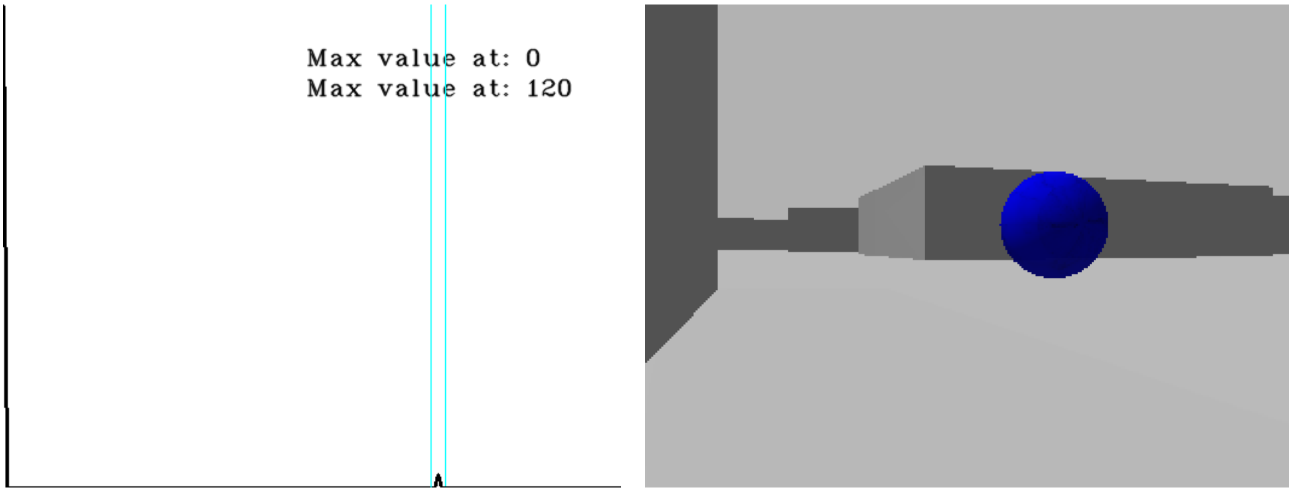
In figure (37a) the hue histogram for the first test can be seen. Only one spike was found on this image at the value of 0, meaning that the background will have its peak at a value of 0.



(a) Hue histogram for test 1 *(b) Image for test 1*

Figure 37: Input image and resulting histogram for the 1st test

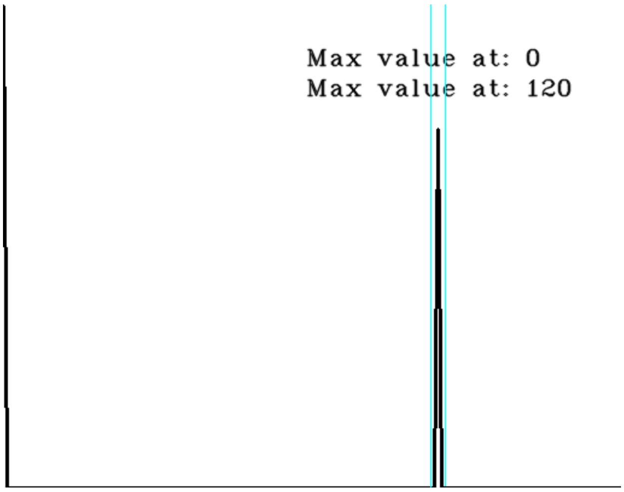
In figure (38a) the hue histogram for the second test can be seen. In this illustration a smaller spike was found with a mean value of 120.



(a) Hue histogram for test 2 *(b) Image for test 2*

Figure 38: Input image and resulting histogram for the 1nd test

In figure (39a) the hue histogram for the third test can be seen. In this histogram a larger spike was found at the value of 120.
The spike was found at the same value as in test 2, and both spikes correspond to the size of the marble found in the image. Therefore the marbles must have a hue value of 120.



(a) Hue histogram for test 3

(b) Image for test 3

Figure 39: Input image and resulting histogram for the 3rd test

Conclusion

It can be concluded that the marbles have a hue value around 120. This will be used as a threshold for separating the marbles from the background.

A.4 Colour separation

The purpose of this test was to find the performance of the colour separation method used to convert the camera image to a binary image. This was done to see whether additional separation parameters should be considered, or if the hue separation is sufficient to separate the marbles from the background.

Description of test

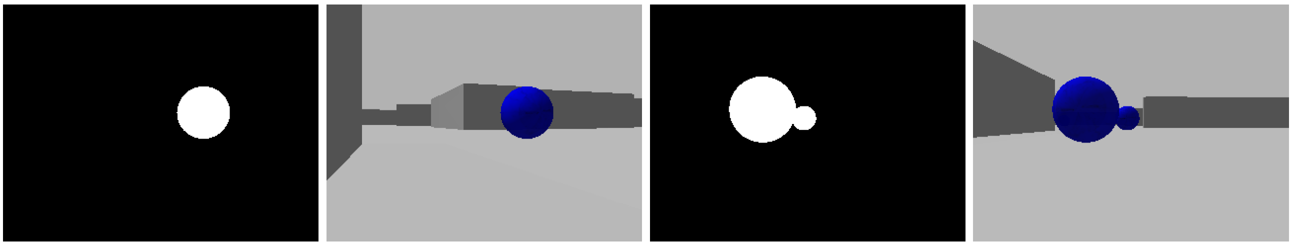
Based on the test in appendix (A.3), the marbles was found to have a hue value of 120. The threshold for the colour separating methods was therefore set at this angle $\pm$ 2, due to the deviation seen on the histograms.
A total of five tests was conducted to see the performance in different scenarios.

Test parameters

- Hue threshold 120 ± 2

Data

In the following illustrations the test results can be seen alongside the original image. It can be seen that the colour separation works well in all cases, without missing parts of the marbles or taking parts of the background into the final image.



(a) Marble separation for test 1 (b) Image for test 1 (c) Marble separation for test 2 (d) Image for test 2

Figure 40: Marbles separated from the background for test 1 and 2

(a) Marble separation for test 3 (b) Image for test 3 (c) Marble separation for test 4 (d) Image for test 4

Figure 41: Marbles separated from the background for test 3 and 4

(a) Marble separation for test 5 (b) Image for test 5

Figure 42: Marbles separated from the background for test 5

Conclusion

It can be concluded that the colour separation works with the base threshold found in the test in appendix (A.3). It can also be concluded that no further criteria must be made to ensure on the marbles are separated from the background.

A.5 Test of marble detection on camera data

The purpose of this test was to test the performance of the marble detecting method on different scenarios of camera data.

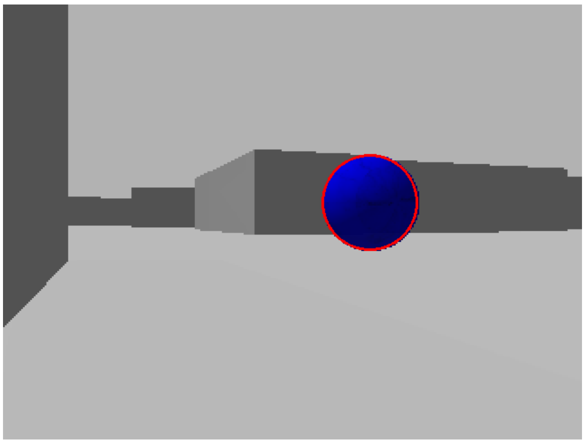
Description of test

In this test a total of 5 tests was conducted to see the performance of the method in different cases. The tests was conducted by first separation the marbles from the background using the method `find_color()`, and the run the marble detection method on the binary image.

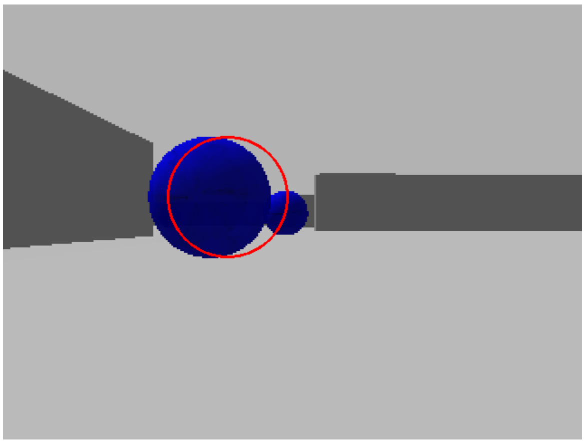
Data

In figure (43a) a single marble can be seen, without another marble being visible behind it or being blocked a wall. This marble was detected without any errors, meaning that the marble detecting method works very well in this case.

In figure (43) two marbles can be seen, one further into the distance and behind the first closer marble. In this case the methods only detects a single marble consistent with the closer marble, but the center slightly shifted to the right, due to the methods inability do distinguish the two marbles. In this case direction to the marble would be slightly off, which could lead to the robot missing the marble.



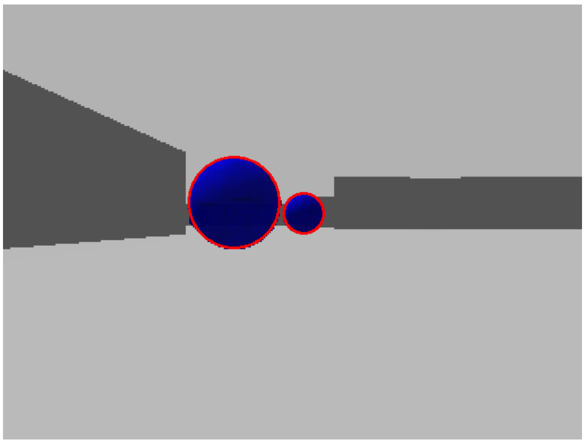
(a) Marble detection on scenario 1



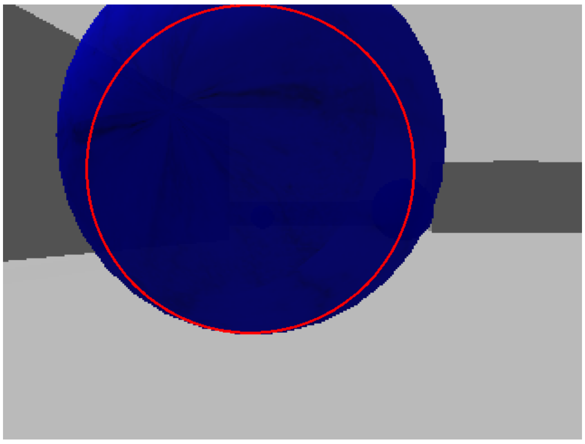
(b) Marble detection on scenario 2

Figure 43: Marble detection of scenario 1 and 2

In figure (44a) two marbles can be seen, on further away than the other. There are no overlap between the two marbles resulting in good detection of both marbles.
In figure (44b) a single very close marble can be seen. It can be seen that the detected marble has a smaller radius than the actual marble, due to the fact that a part of the marble are out of the image. This means that center of the marble are shifted slightly downwards.



(a) Marble detection on scenario 3



(b) Marble detection on scenario 4

Figure 44: Marble detection of scenario 3 and 4

In figure (45) two marbles can be seen, one slightly covered by a wall, and the other slightly in the image. Both marbles was detected correctly. The closet one with a slightly off radius and the one further away with a slightly off center.

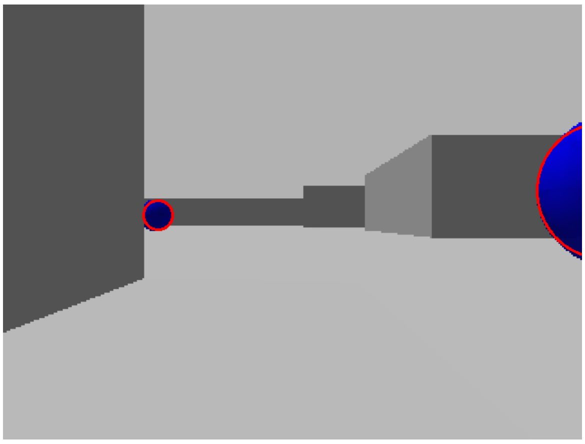


Figure 45: Marble detection on scenario 5

Conclusion

It can be concluded that the marble detecting method works great if the entire marble are visible in the image without another marble hiding behind it. If another marble was to be seen behind the first, it can be concluded that the center would be shifted towards the marble further away.
It can be concluded that marbles cut off on either the left or the right side would be detected correctly, but might have slightly off center and radius.
It can be concluded that a marble cut off on either the top or the bottom would result in incorrectly radius. The center would be placed correctly on the x-axis and off on the y-axis.

A.6 Motion planers in action

The purpose of this test is to find out how well the motion planners work in an actual environment with obstacles. The goal is to lead the robot from an initial position to a target location.

Description of test

The initial position of the robot will be in the origin of the environment of the Gazebo simulator. Here 14 test of both the tangent bug algorithm and model based planner was conducted for the 14 rooms of the bigworld map. The idea is to test the success rate of finding rooms, the robots distance to the closest obstacle on the path to the goal and the distance travelled along the way. The model based planer will be tested with a higher speed than the tangent bug algorithm because it will be further away from obstacle most of the time and thereby less likely of hitting obstacles.

Test parameters

- World used	bigworld	
- Velocity of tangent bug	$U = \{-1 \; to \; 1\}$	(Universe of discourse)
- Velocity of model planner	$U = \{-2 \; to \; 2\}$	(Universe of discourse)
- Number of tests	14	

Data

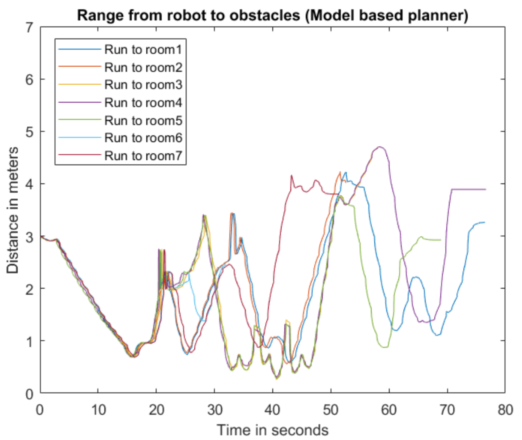
If one compares the model based planner in table (3) with the sensor based planner in table (4) many observations is observed. The model based planner has a success rate of finding rooms at 100 % with and average travelled distance of 46.9 meters. Compared to the sensor based planner with a success rate of 64.3 % and average distance on 27.9 meters, the model based performs very well in finding rooms, but it comes with a cost of distance travelled compared to the sensor based. It is clear from figure (48a) that when the sensor based planner performs well, it outperforms the model based planner in distance travelled.

Modelbased					
	Successrate (%)	Distance traveled (m)	Time (s)	Approx velocity (m/s)	Distance to obstacle (m)
Room 1	100	63.087	76.474	0.825	1.990
Room 2	100	43.331	52.589	0.824	1.876
Room 3	100	44.450	58.178	0.764	1.886
Room 4	100	57.150	76.660	0.745	2.157
Room 5	100	50.800	68.931	0.737	1.809
Room 6	100	23.576	31.978	0.737	1.822
Room 7	100	41.920	51.427	0.815	2.127
Room 8	100	25.400	33.170	0.766	2.401
Room 9	100	31.044	36.554	0.849	2.752
Room 10	100	44.450	52.626	0.845	2.435
Room 11	100	37.355	35.697	1.046	2.628
Room 12	100	59.851	69.724	0.858	2.966
Room 13	100	57.029	67.993	0.839	3.094
Room 14	100	76.906	87.035	0.884	2.893
Average	**100**	**46.882**	**57.074**	**0.824**	**2.345**

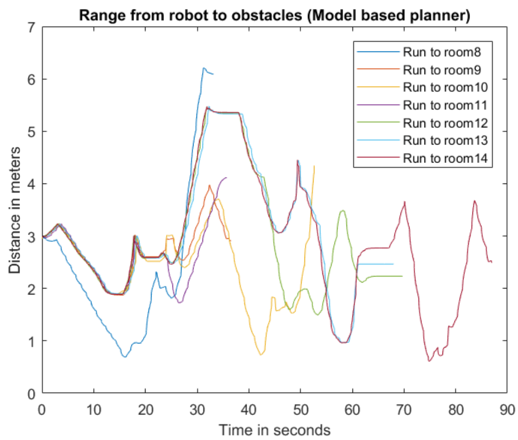
Table 3: Table over test data for the model based planner

Sensorbased					
	Successrate (%)	Distance traveled (m)	Time (s)	Approx velocity (m/s)	Distance to obstacle (m)
Room 1	0	-	-	-	-
Room 2	100	31.044	87.956	0.353	1.613
Room 3	100	23.283	79.409	0.293	1.683
Room 4	100	38.806	135.232	0.287	1.316
Room 5	100	28.222	415.227	0.068	1.033
Room 6	100	8.467	28.323	0.299	3.050
Room 7	0	-	-	-	-
Room 8	100	23.283	79.528	0.293	1.724
Room 9	100	23.989	83.720	0.287	1.193
Room 10	100	38.806	123.614	0.314	1.432
Room 11	100	35.278	102.771	0.343	1.320
Room 12	0	-	-	-	-
Room 13	0	-	-	-	-
Room 14	0	-	-	-	-
Average	**64.286**	**27.909**	**126.198**	**0.282**	**1.596**

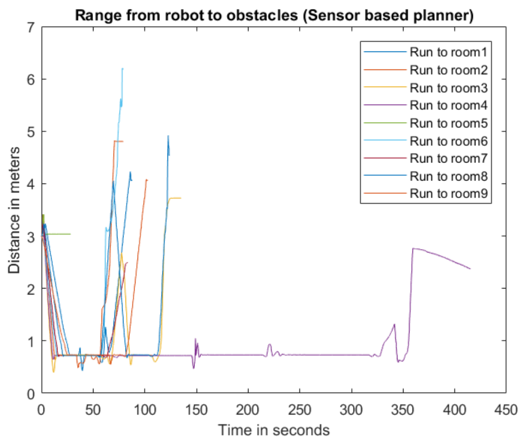
Table 4: Table over test data for the sensor based planner



(a) Illustration of the distance to the closest obstacle for room 1-7 for the model based planner

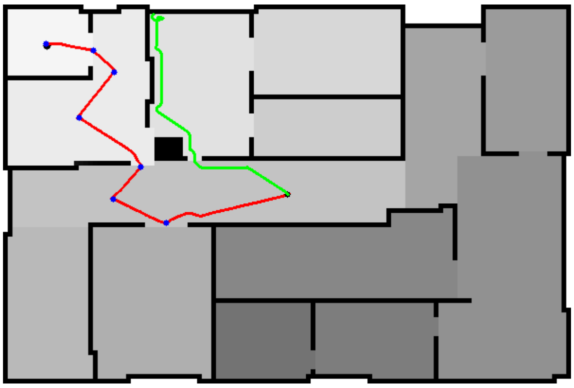


(b) Illustration of the distance to the closest obstacle for room 8-14 for the model based planner

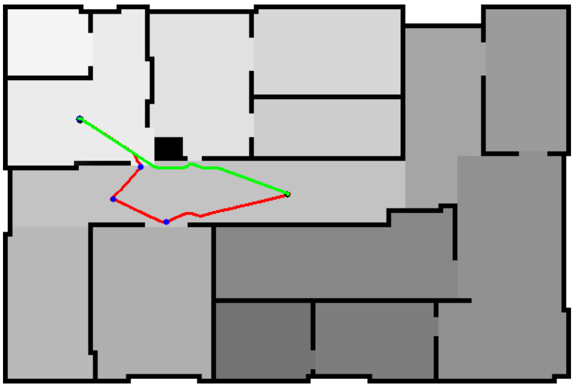


(c) Illustration of the distance to the closest obstacle for room 1-14 for the sensor based planner

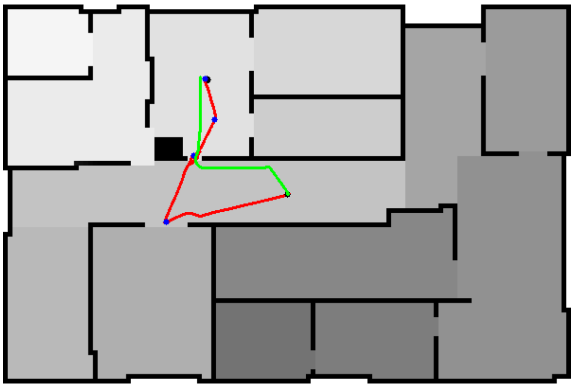
Figure 46: Illustration of the distance to the closest obstacle for room 1-14 for both planners



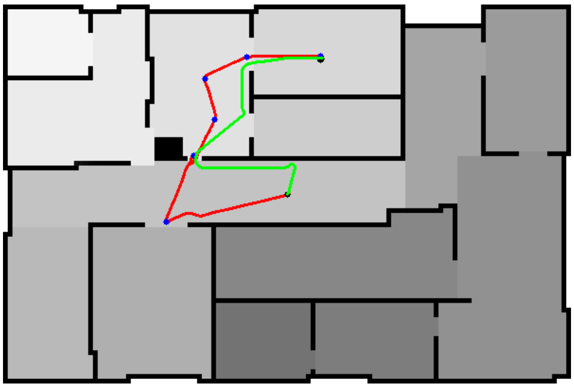
(a) Illustration of the run from the origin to room 1



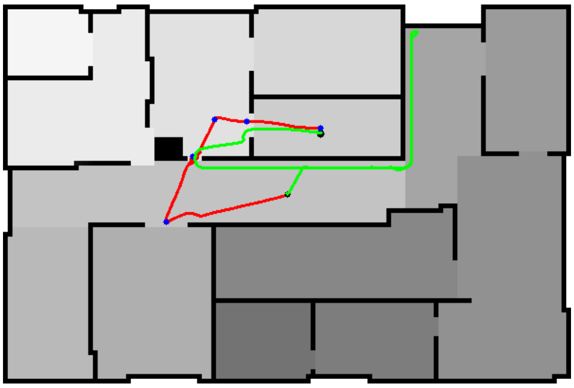
(b) Illustration of the run from the origin to room 2



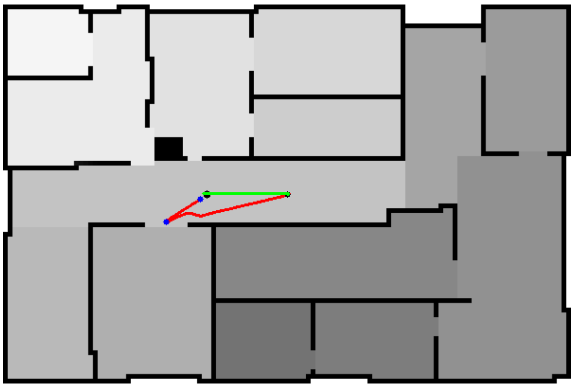
(c) Illustration of the run from the origin to room 3



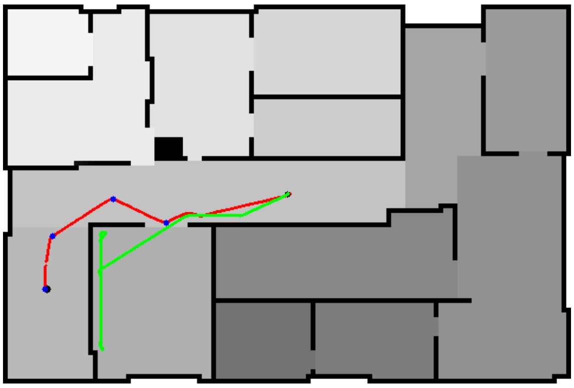
(d) Illustration of the run from the origin to room 4



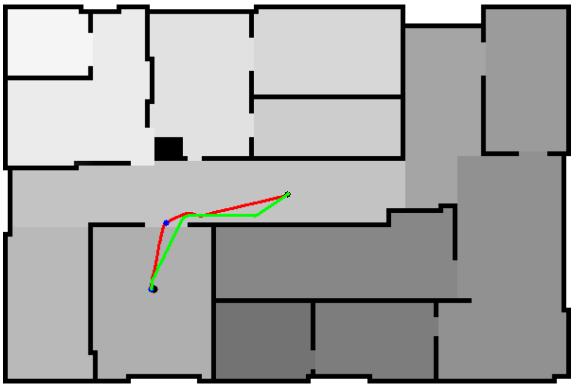
(e) Illustration of the run from the origin to room 5



(f) Illustration of the run from the origin to room 6



(g) Illustration of the run from the origin to room 7



(h) Illustration of the run from the origin to room 8

Figure 47: Illustration of the run from the origin to room 1-8 for both the model based (red line) and the sensor based (green line)

(a) Illustration of the run from the origin to room 9

(b) Illustration of the run from the origin to room 10

(c) Illustration of the run from the origin to room 11

(d) Illustration of the run from the origin to room 12

(e) Illustration of the run from the origin to room 13

(f) Illustration of the run from the origin to room 14

Figure 48: Illustration of the run from the origin to room 9-14 for both the model based (red line) and the sensor based (green line)

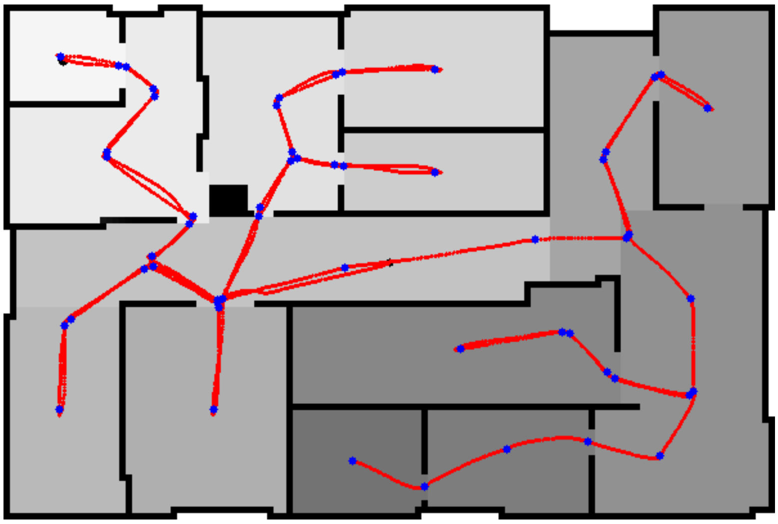


Figure 49: Illustration of the run visiting all rooms starting from the origin for the model based planner

Conclusion

The model based planner is more reliable than the sensor based planner in that it has a success rate of finding rooms at 100 % compared to 64.3 % for the sensor based. However the sensor based planner has an average distance travelled about 27.9 meters compared to the model based about 46.9 meters. It can therefore be concluded that the sensor based planner outperforms the model based in certain scenarios, but the model based is more reliable in finding rooms.

A.7 Model based planners effectiveness to collect marbles

The purpose of this test is to show the effectiveness of the fuzzy control to make the collect marbles.

Description of test

The robot is placed in the origin of the map and set to visit all rooms starting from room 1 to 14. In this test the model based planner is used as motion planning. The sensor data from the robot to the closest obstacles is fetched so that is can be visualised when the robot collects a marble.

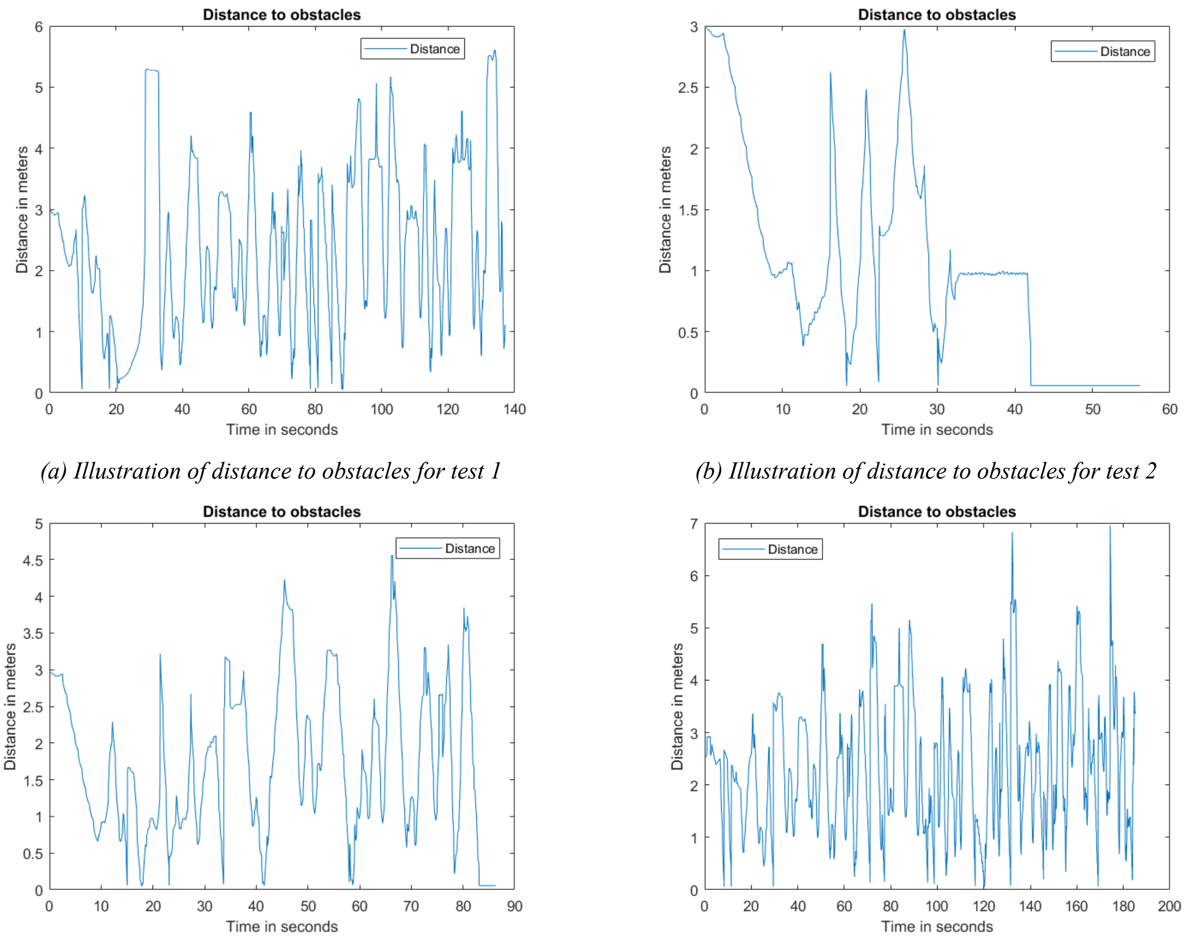
Test parameters

- World used	bigworld	
- Speed of model planer	$U = \{-2\ to\ 2\}$	(Universe of discourse)
- Number of tests	10	
- Number of marbles	20	

Data

Seen from figure (51f), test 10 is the only one when the robots actually visits all rooms. This was primarily due to a problem with a simulation problem Gazebo. Many times when the robot picked up marbles Gazebo crashed. This problem made it rather difficult to conduct useful tests about the effectiveness of collecting marbles. Figure (50b) shows this case. One can see that after 42 seconds the distance to an obstacle is continuously close to zero meaning that Gazebo have crashed and the test had to be aborted. From the same graph one can see the marbles been collected because of the spike very close to zero. It can also be visualised on the path on the robot, where red dot indicates a marble have been found.

Model based										
	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
Marble 1	9.801	18.274	14.988	8.323	16.987	7.315	17.195	13.937	10.469	6.290
Marble 2	18.047	22.438	17.533	11.381	20.727	64.122	21.065	36.695	21.509	22.489
Marble 3	20.430	30.108	23.129	29.490	29.189	86.813	36.784	107.336	39.083	29.828
Marble 4	78.501	42.058	33.680	71.099	-	89.096	39.808	110.907	98.340	48.259
Marble 5	80.805	-	41.228	95.845	-	-	42.896	135.985	-	78.271
Marble 6	87.962	-	58.009	98.683	-	-	-	-	-	79.921
Marble 7	88.036	-	83.116	116.301	-	-	-	-	-	87.543
Marble 8	-	-	-	120.020	-	-	-	-	-	93.739
Marble 9	-	-	-	131.455	-	-	-	-	-	97.356
Marble 10	-	-	-	169.128	-	-	-	-	-	101.262



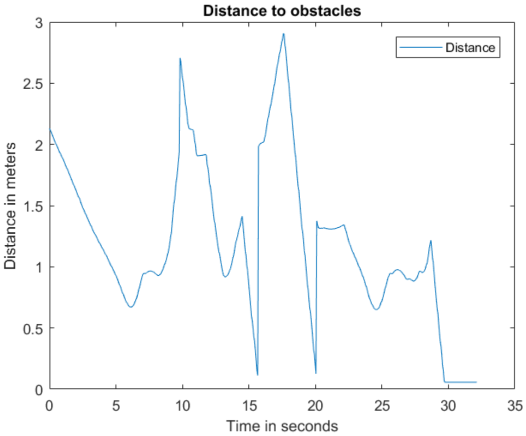
(a) Illustration of distance to obstacles for test 1

(b) Illustration of distance to obstacles for test 2

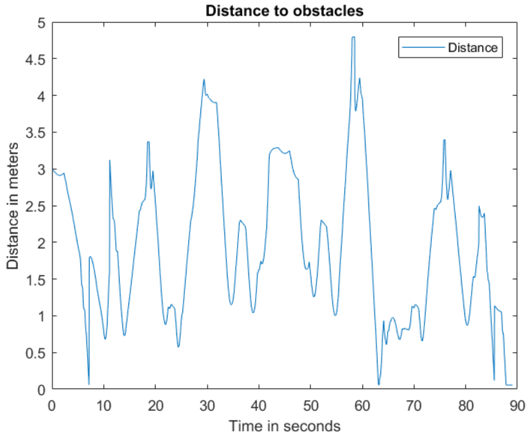
(c) Illustration of distance to obstacles for test 3

(d) Illustration of distance to obstacles for test 4

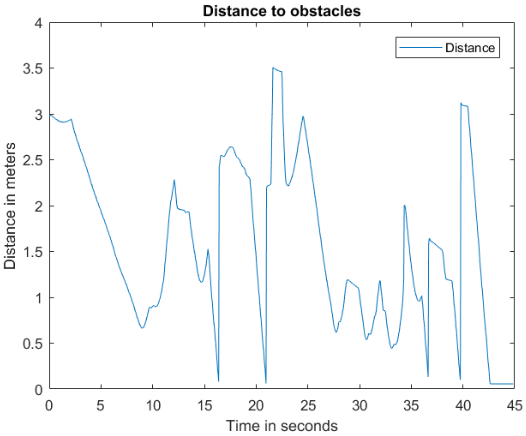
Figure 50: Illustration of distance travelled for test 1-4, where the robots visits all 14 rooms while collecting marbles



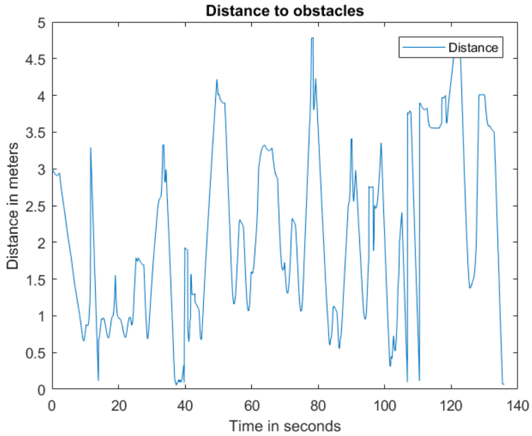
(a) Illustration of distance to obstacles for test 5



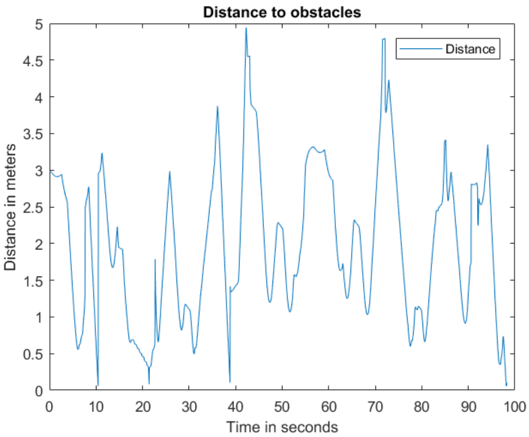
(b) Illustration of distance to obstacles for test 6



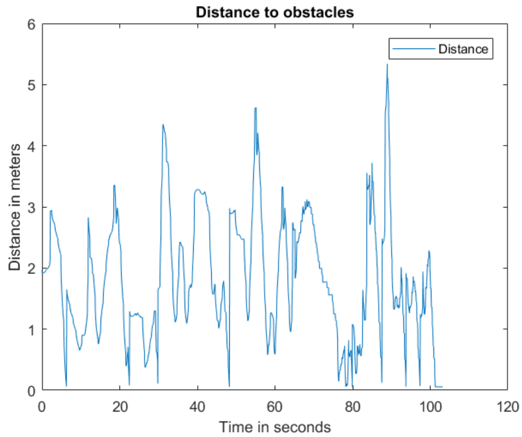
(c) Illustration of distance to obstacles for test 7



(d) Illustration of distance to obstacles for test 8

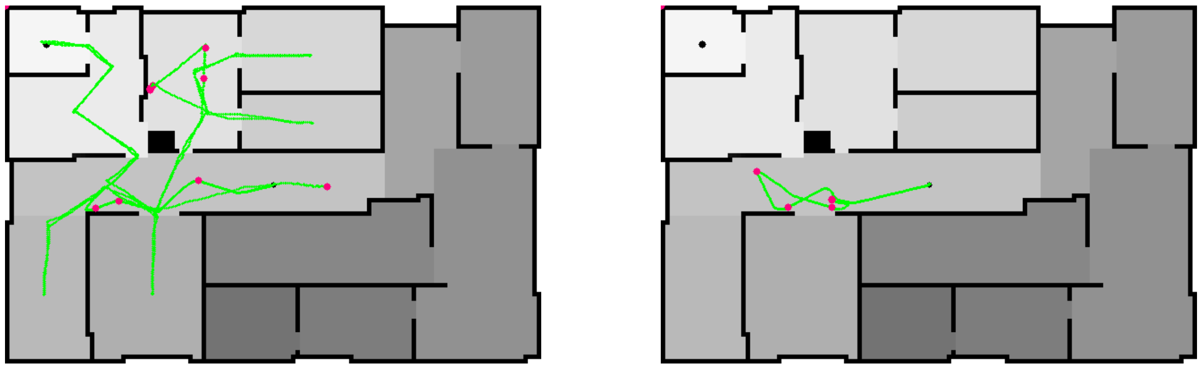


(e) Illustration of distance to obstacles for test 9



(f) Illustration of distance to obstacles for test 10

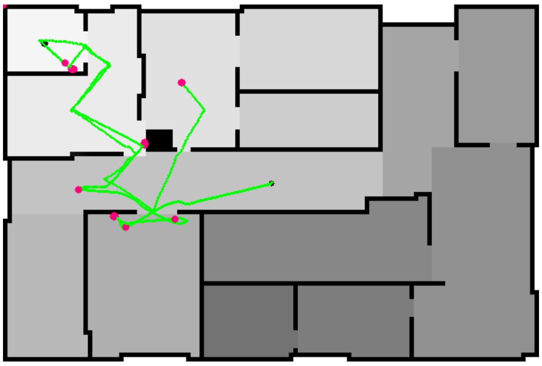
Figure 51: Illustration of distance to obstacles for test 5-10



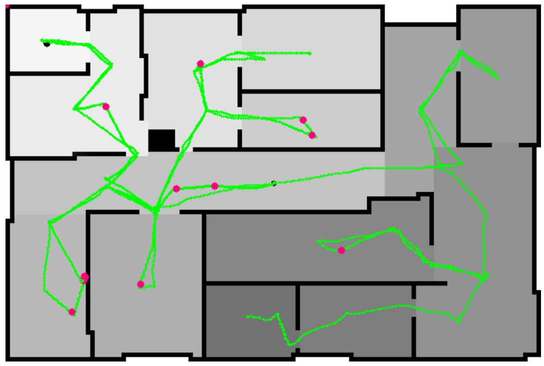
(a) *Illustration of distance travelled for test 1, where the robots visits all 14 rooms while collecting marbles*

(b) *Illustration of distance travelled for test 2, where the robots visits all 14 rooms while collecting marbles*

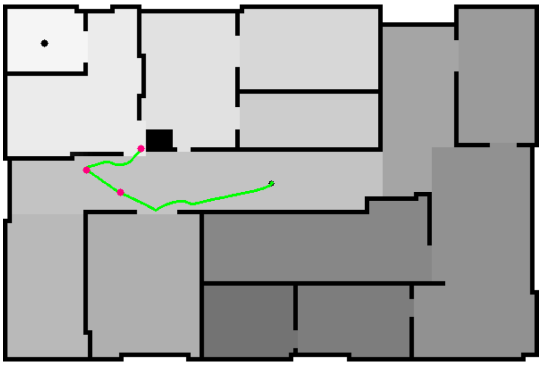
Figure 52: Illustration of distance travelled for test 1-2, where the robots visits all 14 rooms while collecting marbles



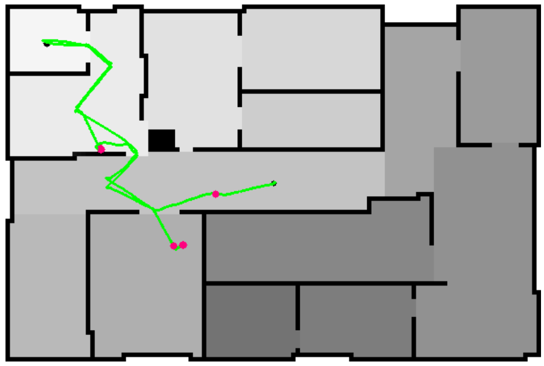
(a) Illustration of distance travelled for test 3, where the robots visits all 14 rooms while collecting marbles



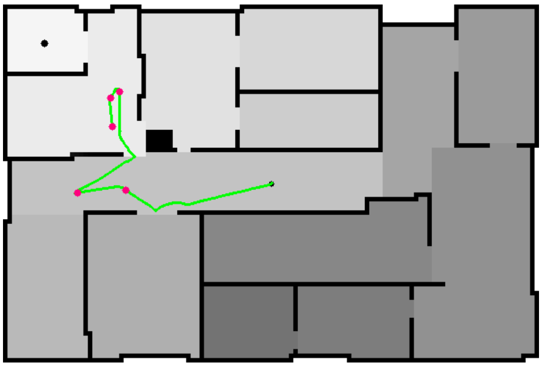
(b) Illustration of distance travelled for test 4, where the robots visits all 14 rooms while collecting marbles



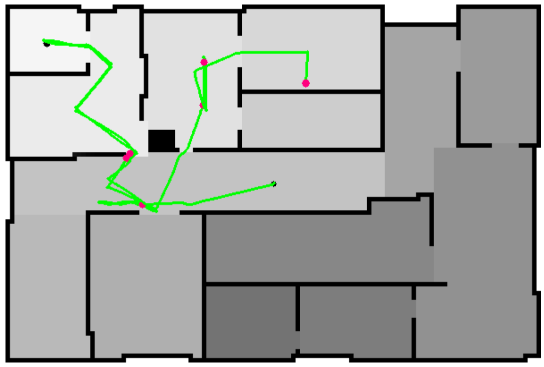
(c) Illustration of distance travelled for test 5, where the robots visits all 14 rooms while collecting marbles



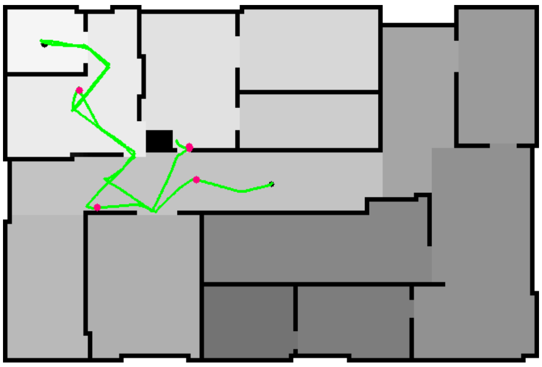
(d) Illustration of distance travelled for test 6, where the robots visits all 14 rooms while collecting marbles



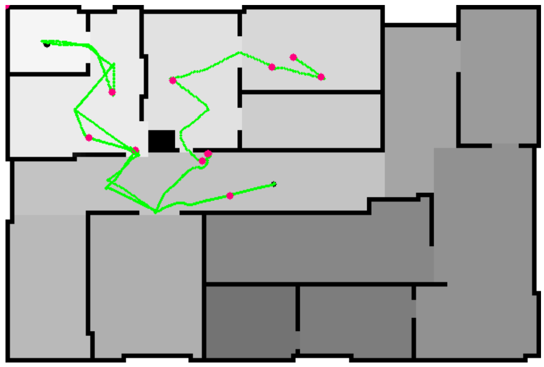
(e) Illustration of distance travelled for test 7, where the robots visits all 14 rooms while collecting marbles



(f) Illustration of distance travelled for test 8, where the robots visits all 14 rooms while collecting marbles



(g) Illustration of distance travelled for test 9, where the robots visits all 14 rooms while collecting marbles



(h) Illustration of distance travelled for test 10, where the robots visits all 14 rooms while collecting marbles

Figure 53: Illustration of distance travelled for test 3-10, where the robots visits all 14 rooms while collecting marbles

Conclusion

It can be concluded that the fuzzy controller works and that the robot is able to find marbles. Due to problems with Gazebo, only one of ten test lead to a complete search of marbles in all rooms. In test ten the robot was able to find 10 out of 20 marbles with a success rate of 50 %. But because the robot only made through all rooms once, no average of the effectiveness of finding marbles can be concluded.

A.8 Room based probability of marbles spawning

The purpose of this test is to determine the probability of a marble spawning in each of the 14 rooms described in section 3.1.

Description of test

This test was done by conduction a total of 50 tests, where the position of the 20 marbles in the Gazebo environment are saved resulting in a total of 1000 samples.

These marbles was then mapped to one of the 14 rooms using the class `map_class`. The total amount of marbles found in each room can be seen in table (5). This data was then divided by the total number of marbles, to find the probability.

Due to the fact that the rooms are not the same size, this probability was divided by the size of the room (number of pixels in the map) and the normalised. The result can be seen in table (6).

Test parameters

- World used	bigworld
- Number of spawned marbles	20
- Number of tests	50

Data

Distribution of marbles	
Total number of marbles found in room 1	5
Total number of marbles found in room 2	65
Total number of marbles found in room 3	75
Total number of marbles found in room 4	52
Total number of marbles found in room 5	72
Total number of marbles found in room 6	158
Total number of marbles found in room 7	17
Total number of marbles found in room 8	118
Total number of marbles found in room 9	100
Total number of marbles found in room 10	22
Total number of marbles found in room 11	75
Total number of marbles found in room 12	160
Total number of marbles found in room 13	47
Total number of marbles found in room 14	34

Table 5: Table of how the marbles are distributed in the 14 rooms based on all 50 tests

Probability of marbles	
Probability of marbles found in room 1	0.012232
Probability of marbles found in room 2	0.061057
Probability of marbles found in room 3	0.078610
Probability of marbles found in room 4	0.059975
Probability of marbles found in room 5	0.117993
Probability of marbles found in room 6	0.098801
Probability of marbles found in room 7	0.019629
Probability of marbles found in room 8	0.099063
Probability of marbles found in room 9	0.114518
Probability of marbles found in room 10	0.027633
Probability of marbles found in room 11	0.044562
Probability of marbles found in room 12	0.130379
Probability of marbles found in room 13	0.072915
Probability of marbles found in room 14	0.062541

Table 6: Table of how the probabilities are distributed in the 14 rooms with room size taken into account

Conclusion

It can be concluded that the highest number of marbles was found in room 12 closely followed by 6 and 8. But due to the size of the rooms, the highest probability is found in room 12, followed by 5 and 9.

A.9 Estimate for path lengths

The purpose of this test was to find an estimate for the distances between the rooms in order to have a distance punishment for the Q-learning.

Description of test

This test was conducted using Geogebra a cas tool for geometry and algebra. An image of the environment bigworld was loaded into the program. The width of the image was set to 8.4 cm.

Based on this lines was drawn by hand between the centroids of the rooms navigating any obstacles. This can be seen in figure (54).

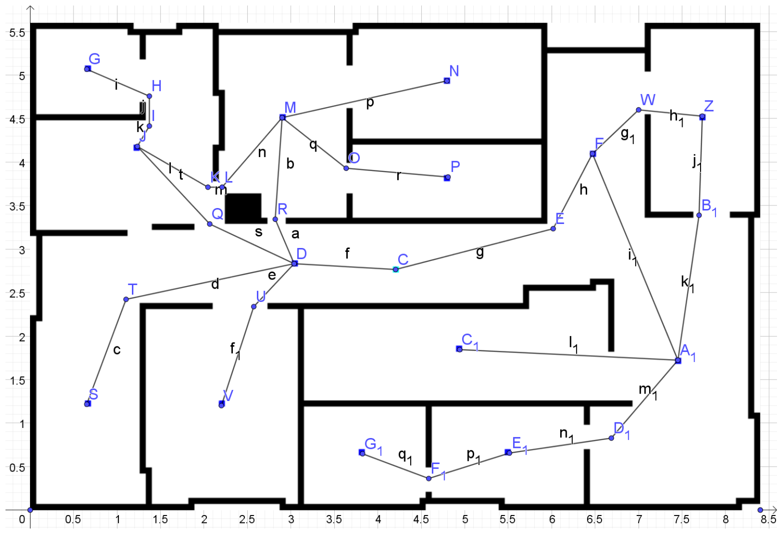


Figure 54: Illustration of paths in geogebra

Geogebra calculates the length of each line. Based on this the total path length from one room to another was found. This can be seen in table gegegepgkek

Test parameters

- World used bigworld
- Length of world 8.4 cm

Data

Distribution of marbles			
Distance from room 1 to 2	-2.78	Distance from room 2 to 3	-4.32
Distance from room 2 to 6	-4.58	Distance from room 3 to 4	-3.88
Distance from room 3 to 5	-4.24	Distance from room 3 to 6	-3.46
Distance from room 6 to 7	-6.54	Distance from room 6 to 8	-3.76
Distance from room 7 to 9	-8.02	Distance from room 9 to 10	-2.94
Distance from room 9 to 11	-5.14	Distance from room 10 to 11	-5.64
Distance from room 11 to 12	-5.02	Distance from room 11 to 13	-4.72
Distance from room 13 to 14	-3.56		

Table 7: Distances between rooms

A.10 The impact of ϵ on Q-learning performance

The purpose of this test is to show how different values of ϵ influences the performance of Q-learning.

Description of test

This test was done by performing 100 trials of each selected value of ϵ for a range of episodes. The test was conducted using the world ”5-room world” seen on figure (55).
The distance punishments are bases on those found in the test in appendix(A.9). The distance punishments was scaled with a factor of 1.2 to make the distance a bigger factor in final path.
The probabilities used for each room was found in the test in appendix (A.8). These probabilities was divided by the maximal value and scaled by a factor of 20. This was done to ensure that the total reward for entering a room the first time would be positive.

The initial state for all tests was set to room 3 in order to ensure greatest number of possible paths.

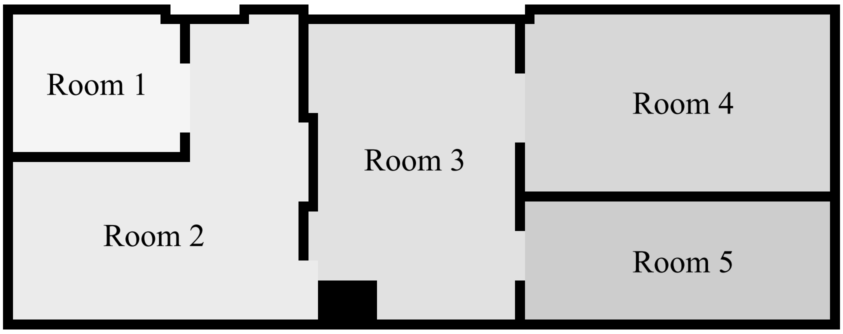


Figure 55: Illustration of ”5-room world”

Test parameters

In the following tables, the parameters for the test can be seen. In order not to make the agent act completely randomly, it have been chosen to make the test on values of ϵ between 0.01 and 0.5.

- World used	5-room world
- Initial room	room 3
- Probabilities based on	50 tests
- Number of tests	100
- Scaling factor distance	1.2
- Scaling factor reward	20
- Learning rate α	0.1
- Discount factor γ	0.9

Tested values of ϵ	
0.01	0.15
0.025	0.2
0.05	0.3
0.075	0.4
0.1	0.5

Table 8: Table of tested the values of ϵ. Ranging from 1 % to 50 %

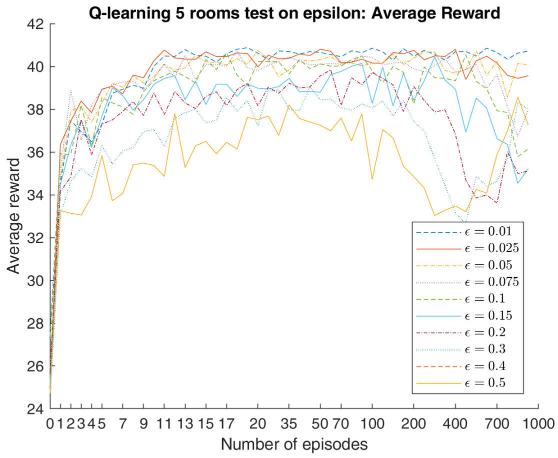
Distance punishments	
Start to room 3	0
Room 1 to room 2	-1.668
Room 2 to room 3	-2.592
Room 3 to room 4	-2.328
Room 3 to room 5	-2.544

Table 9: Table of the distance punishments. The distances are found in the test in appendix (A.9)

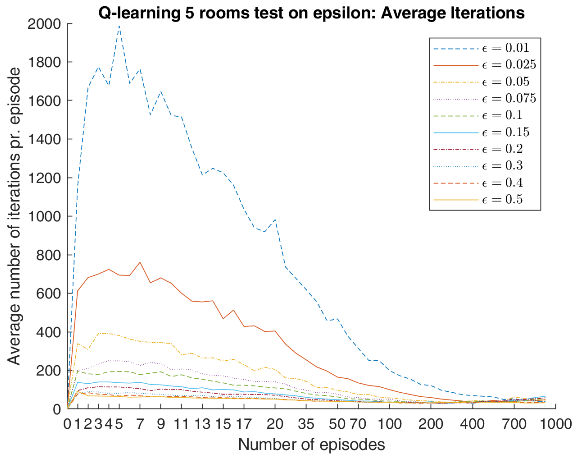
Data

In figure (56a) the average reward for all tests can be seen. It can be seen that the lower the value of ϵ the higher average reward, meaning that the policy will converge to the optimal policy.

In figure (56b) the average number of iterations per episoden can be seen. It can be seen that the higher the value of ϵ the lower the average number of iterations will be. Given the fact that the more random the agent acts, the faster the agent will search alternative paths to the policy, and find its way through the environment.



(a) *Plot of how the average reward develops as a function of the number of episodes for each value of* ϵ



(b) *Plot of how the average number of episodes develops as a function of the number of episodes for each value of* ϵ

Figure 56: Plots of both the average reward and average number of iterations pr. episode for each value of ϵ

A value of 0.05 have been chosen as the best compromise between a high average reward and a low average number of iterations per episode.

Conclusion

It can be concluded that the smaller the value of ϵ the higher the average reward will be for a given number of episodes.
It can as well be concluded that the higher the value of ϵ the lower the number of iterations per episode will be.
The value of 0.05 was chosen as the best compromise.

A.11 The impact of α on Q-learning performance

The purpose of this test is to show how different values of α influences the performance of Q-learning.

Description of test

This test was done by performing 100 trials of each selected value of α for a range of episodes. The test was conducted using the world 5-room world seen on figure (57).
The distance punishments are bases on those found in the test in appendix (A.9). The distance punishments was scaled with a factor of 1.2 to make the distance a bigger factor in final path.
The probabilities used for each room was found in the test in appendix (A.8). These probabilities was divided by the maximal value and scaled by a factor of 20. This was done to ensure that the total reward for entering a room the first time would be positive.

The initial state for all tests was set to room 3 in order to ensure greatest number of possible paths.

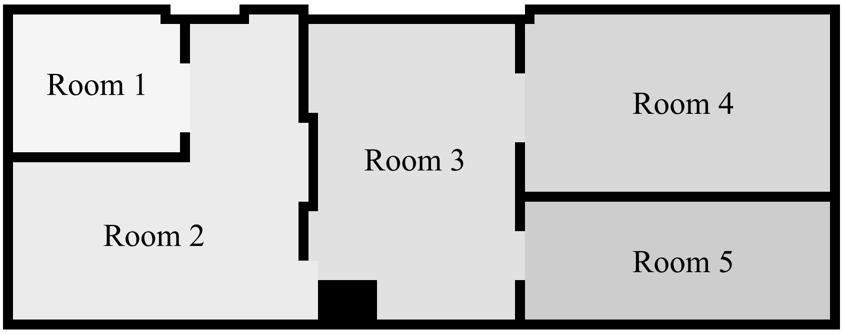


Figure 57: Illustration of "5-room world"

Test parameters

In the following tables, the parameters for the test can be seen. It have been chosen to make the test on values of α between 0.01 and 0.5.

- World used	5-room world
- Initial room	room 3
- Probabilities based on	50 tests
- Number of tests	100
- Scaling factor distance	1.2
- Scaling factor reward	20
- Randomness factor ϵ	0.05
- Discount factor γ	0.9

Tested values of α	
0.01	0.15
0.025	0.2
0.05	0.3
0.075	0.4
0.1	0.5

Table 10: Table of tested the values of α. Ranging from 0.01 to 0.5

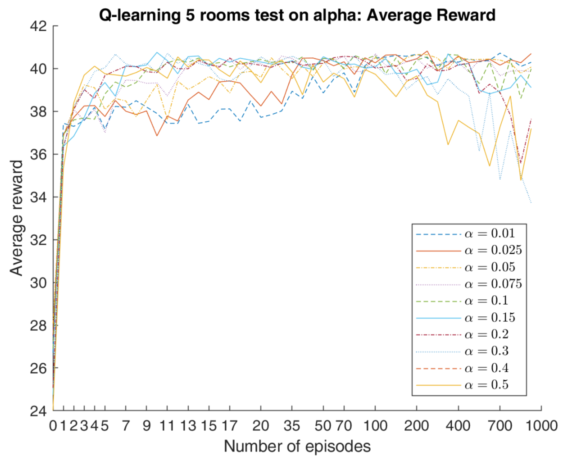
Distance punishments	
Start to room 3	0
Room 1 to room 2	-1.668
Room 2 to room 3	-2.592
Room 3 to room 4	-2.328
Room 3 to room 5	-2.544

Table 11: Table of the distance punishments. The distances are found in the test in appendix (A.9)

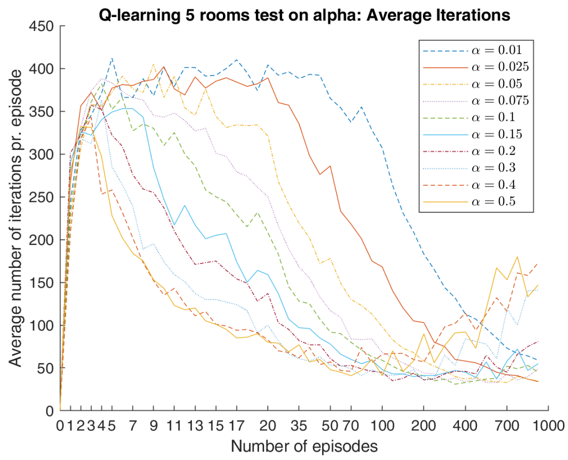
Data

In figure (58a) the average reward for all tests can be seen. It can be seen that the lower the value of α the higher average reward, meaning that the policy will converge to the optimal policy. Whereas a higher value of α will result in a lower average reward. It would also seem like the average reward are oscillating more.

In figure (58b) the average number of iterations per episoden can be seen. It can be seen that the higher the value of α the lower the average number of iterations will be. This is due to the fact, that the lower the learning rate are, the slower the algorithm learns, and will therefore need more steps to get to the goal.



(a) Plot of how the average reward develops as a function of the number of episodes for each value of ϵ



(b) Plot of how the average number of episodes develops as a function of the number of episodes for each value of α

Figure 58: Plots of both the average reward and average number of iterations pr. episode for each value of α

A value of 0.025 have been chosen as the best compromise between a high average reward and a low average number of iterations per episode.

Conclusion

It can be concluded that the smaller the value of α the higher the average reward will be for a given number of episodes. It can as well be concluded that the higher the value of α the lower the number of iterations per episode will be.
The value of 0.025 was chosen as the best compromise.

A.12 The impact of γ on Q-learning performance

The purpose of this test is to show how different values of γ influences the performance of Q-learning.

Description of test

This test was done by performing 100 trials of each selected value of γ for a range of episodes. The test was conducted using the world 5-room world seen in figure (59).
The distance punishments are bases on those found in the test in appendix (A.9). The distance punishments was scaled with a factor of 1.2 to make the distance a bigger factor in final path.
The probabilities used for each room was found in the test in appendix (A.8). These probabilities was divided by the maximal value and scaled by a factor of 20. This was done to ensure that the total reward for entering a room the first time would be positive.

The initial state for all tests was set to room 3 in order to ensure greatest number of possible paths.

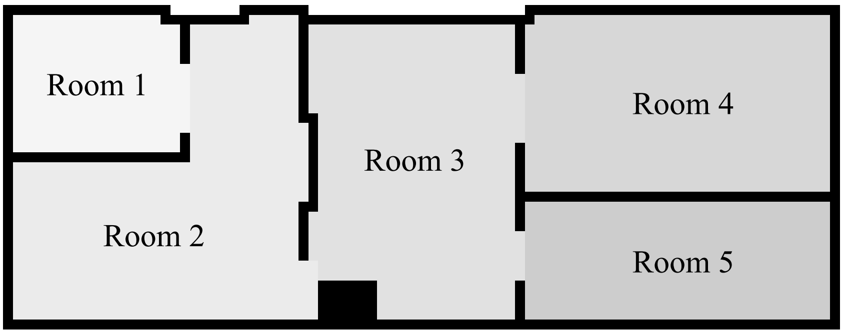


Figure 59: Illustration of "5-room world"

Test parameters

In the following tables, the parameters for the test can be seen. It have been chosen to make the test on values of γ between 0.99 and 0.5.

- World used	5-room world
- Initial room	room 3
- Probabilities based on	50 tests
- Number of tests	100
- Scaling factor distance	1.2
- Scaling factor reward	20
- Randomness factor ϵ	0.05
- Learning rate α	0.025

Tested values of γ	
0.99	0.85
0.975	0.8
0.95	0.7
0.925	0.6
0.9	0.5

Table 12: Table of tested the values of γ. Ranging from 0.99 to 0.5

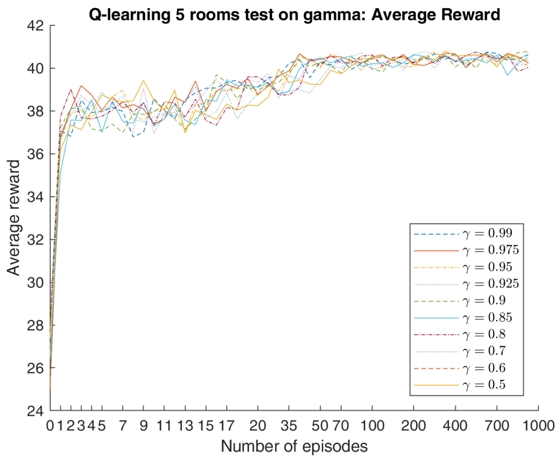
Distance punishments	
Start to room 3	0
Room 1 to room 2	-1.668
Room 2 to room 3	-2.592
Room 3 to room 4	-2.328
Room 3 to room 5	-2.544

Table 13: Table of the distance punishments. The distances are found in the test in appendix (A.9)

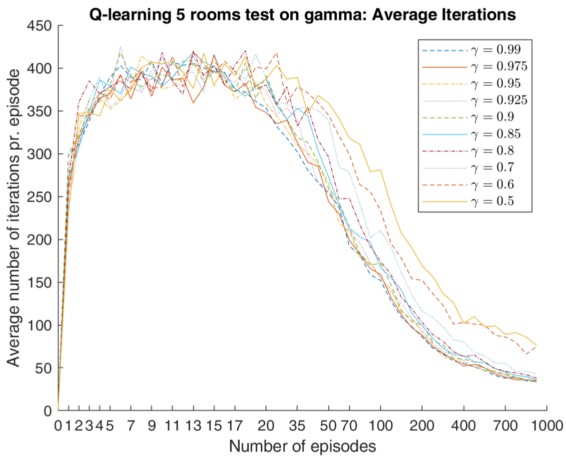
Data

In figure (60a) the average reward for all tests can be seen. It can be seen that the higher the value of γ the higher average reward. Whereas a lower value of γ will result in a lower average reward.

In figure (60b) the average number of iterations per episoden can be seen. It can be seen that the higher the value of γ the lower the average number of iterations will be.



(a) Plot of how the average reward develops as a function of the number of episodes for each value of ϵ



(b) Plot of how the average number of episodes develops as a function of the number of episodes for each value of α

Figure 60: Plots of both the average reward and average number of iterations pr. episode for each value of γ

A value of 0.99 have been chosen as the best compromise between a high average reward and a low average number of iterations per episode.

Conclusion

It can be concluded that the higher the value of γ the higher the average reward will be for a given number of episodes.
It can as well be concluded that the higher the value of γ the lower the number of iterations per episode will be.
The value of 0.99 was chosen as the best compromise.

A.13 Best path based on Q-learning

The purpose of this test was to find the optimal path for covering 5-room world in the best possible way

Description of test

This test consisted of 5 subtest, where each test was started in different rooms. This was to find out, which room was the most optimal to start in. Each subtest consisted of 10 trials, where all parameters where the same.
The test was conducted using the world 5-room world seen in figure (61).
The distance punishments are bases on those found in the test in appendix (A.9). The distance punishments was scaled with a factor of 1.2 to make the distance a bigger factor in final path.
The probabilities used for each room was found in the test in appendix (A.8). These probabilities was divided by the maximal value and scaled by a factor of 20. This was done to ensure that the total reward for entering a room the first time would be positive.
All tests ran for 2000 episodes, with ϵ varying from 0.5 to 0.05. ϵ was started on 0.5 and reduced by 0.05 for each 200 episodes, enabling the algorithm to make larger learning steps in the beginning.

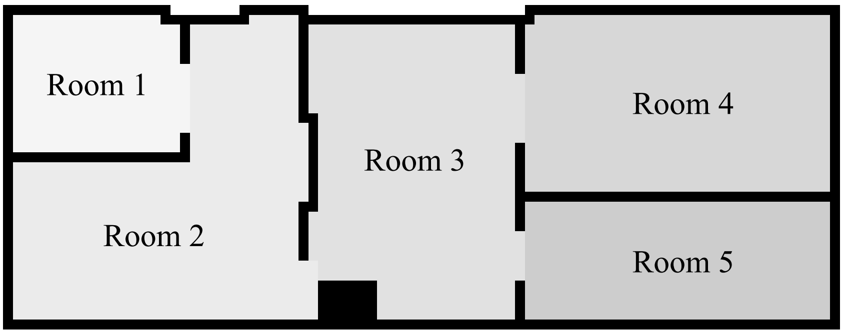


Figure 61: Illustration of "5-room world"

Test parameters

In the following tables, the parameters for the test can be seen.

- World used	5-room world
- Probabilities based on	50 tests
- Number of tests	10
- Scaling factor distance	1.2
- Scaling factor reward	20
- Randomness factor ϵ	0.5, 0.45, 0.4, 0.35, 0.3, 0.25, 0.2, 0.15, 0.1, 0.05
- Learning rate α	0.025
- Discount factor γ	0.99

Distance punishments	
Start to room 3	0
Room 1 to room 2	-1.668
Room 2 to room 3	-2.592
Room 3 to room 4	-2.328
Room 3 to room 5	-2.544

Table 14: Table of the distance punishments. The distances are found in the test in appendix (A.9)

Data

In the following tables, the result from those tests can be seen. Looking at the average reward from those 5 tests, it can be seen that the best average was achieved in the test starting in room 1. The best case was found to be was found to be both start in room 1 and 5.

Start in room 1		
Trial	**Path**	**Reward**
1	0, 1, 2, 3, 5, 3, 4	44.2521
2	0, 1, 2, 3, 5, 3, 4	44.4681
3	0, 1, 2, 3, 4, 3, 5	44.2521
4	0, 1, 2, 3, 5, 3, 4	44.2521
5	0, 1, 2, 3, 5, 3, 4	44.2521
6	0, 1, 2, 3, 5, 3, 4	44.2521
7	0, 1, 2, 3, 5, 3, 4	44.4681
8	0, 1, 2, 3, 4, 3, 5	44.2521
9	0, 1, 2, 3, 5, 3, 4	44.2521
10	0, 1, 2, 3, 5, 3, 4	44.2521
Average Reward		44.3001
Best	0, 1, 2, 3, 5, 3, 4	44.4681

Table 15: All 10 paths from test started in room 1

Start in room 2		
Trial	**Path**	**Reward**
1	0, 2, 3, 5, 3, 4, 3, 2, 1	39.3322
2	0, 2, 3, 5, 3, 4, 3, 2, 1	39.3322
3	0, 2, 3, 5, 3, 4, 3, 4, 3, 2, 1	34.6762
4	0, 2, 3, 5, 3, 4, 3, 5, 3, 5, 3, 2, 1	29.1562
5	0, 2, 3, 4, 4, 3, 5, 3, 4, 3, 2, 1	34.6762
6	0, 2, 3, 4, 3, 5, 3, 4, 3, 4, 3, 4, 3, 2, 1	25.3642
7	0, 2, 3, 5, 3, 4, 3, 5, 3, 2, 1	34.2442
8	0, 2, 3, 5, 3, 4, 3, 5, 3, 3, 2, 1	34.2442
9	0, 2, 3, 4, 3, 5, 3, 5, 3, 2, 1	34.2442
10	0, 2, 3, 5, 5, 3, 4, 3, 5, 3, 4, 3, 2, 2, 1	29.5882
Average Reward		33.4858
Best	0, 2, 3, 5, 3, 4, 3, 2, 1	39.3322

Table 16: All 10 paths from test started in room 2

Start in room 3		
Trial	**Path**	**Reward**
1	0, 3, 5, 3, 2, 1, 2, 3, 4	39.9921
2	0, 3, 4, 3, 2, 1, 2, 3, 5	40.2081
3	0, 3, 2, 1, 2, 3, 5, 3, 4	39.9921
4	0, 3, 5, 3, 4, 3, 2, 1	41.9241
5	0, 3, 2, 1, 2, 3, 4, 3, 5	40.2081
6	0, 3, 2, 1, 2, 3, 5, 3, 4	39.9921
7	0, 3, 2, 1, 2, 3, 5, 3, 4	39.9921
8	0, 3, 2, 1, 2, 3, 5, 3, 4	39.9921
9	0, 3, 4, 3, 5, 3, 2, 1	41.9241
10	0, 3, 4, 3, 5, 3, 4, 3, 0, 3, 2, 1	37.2681
Average Reward		40.1493
Best	0, 3, 5, 3, 4, 3, 2, 1	41.9241

Table 17: All 10 paths from test started in room 3

Start in room 4		
Trial	**Path**	**Reward**
1	0, 4, 3, 5, 3, 2, 1	44.2521
2	0, 4, 3, 2, 1, 2, 3, 5	42.5361
3	0, 4, 3, 2, 3, 5, 3, 4, 3, 4, 3, 2, 1	29.7562
4	0, 4, 3, 2, 1, 2, 3, 5	42.5361
5	0, 4, 3, 5, 3, 2, 1	44.2521
6	0, 4, 3, 2, 3, 5, 3, 2, 1	39.0681
7	0, 4, 3, 5, 3, 2, 1	44.2521
8	0, 4, 0, 4, 3, 2, 1, 2, 3, 5	42.5361
9	0, 4, 3, 5, 3, 2, 1	44.2521
10	0, 4, 3, 5, 3, 2, 1	44.2521
Average Reward		41.7693
Best	0, 4, 3, 5, 3, 2, 1	44.2521

Table 18: All 10 paths from test started in room 4

Start in room 5		
Trial	**Path**	**Reward**
1	0, 5, 3, 2, 1, 2, 3, 4	42.5361
2	0, 5, 3, 2, 1, 2, 3, 4	42.5361
3	0, 5, 3, 5, 5, 3, 4, 3, 2, 1	39.3801
4	0, 5, 3, 2, 1, 2, 3, 4	42.5361
5	0, 5, 3, 2, 1, 2, 3, 4	42.5361
6	0, 5, 3, 2, 1, 1, 2, 3, 2, 3, 4	37.3521
7	0, 5, 3, 2, 1, 2, 3, 4	42.5361
8	0, 5, 3, 4, 3, 2, 1	44.4681
9	0, 5, 3, 4, 3, 2, 1	44.4681
10	0, 5, 3, 4, 3, 2, 1	44.4681
Average Reward		42.2817
Best	0, 5, 3, 4, 3, 2, 1	44.4681

Table 19: All 10 paths from test started in room 5

Conclusion

It can be concluded that the best average reward was found in the test starting in room 1. It can also be concluded that the best path overall was found in either room 1 or 5, where the best case in both tests had the same reward.
The best path was found to be either of the following two.

Initial room	**Path**	**Reward**
1	0, 1, 2, 3, 5, 3, 4	44.4681
5	0, 5, 3, 4, 3, 2, 1	44.4681