The disc robot we used has two types of sensors mounted, two distance sensors and one contact switch. The input from two distance sensors are mapped as state for Q-table, and the contact switch is used to approximate collisions with obstacles in the environment.

1. Packages and parameters

The platform chosen for conducting this experiment was MATLAB 2020b[[1]](#endnote-1). Robotic playground and its dependencies[[2]](#endnote-2) were used for the modeling of the disc robot and obstacle environment. The default disc robot from the package was used, and its detailed parameters were as follows: The distance between wheels of the disc robot was 0.5 meters, with wheels of radius 0.1m. The two mounted distance sensors had a position offset of [0.2 0.15] with orientation offset 45 and a position offset of [0.2 -0.15]. Both sensors had maximum range of 20 meters and minimum range of 0 meter, with a resolution of 0.01 meter and a sample time of 0.1 second. The input value will be processed to be accurate to 0.2m and with a range of 0 to 1.8 meters inclusive. The contact switch had a position offset of [0 0], with orientation offset of 0 degree. The distance for it to be triggered was set at 0.02 meter. It was much less than the radius of the disc robot itself, but this setting gave the best estimation of collision. The arena was of size 5 x 8 and the starting position of the disc robot was [-1.4 0]. In the simple setting, a block of 2 x 5 was placed in the position [0 0] of the arena. The medium setting had two blocks of size 2 x 2, placed at [0 2] and [0 -2], and the hard setting has three blocks, two of size 3 x 1 at [0 2.5] and [0 -2.5], one of size 1.5 x 2 at [0 0]. **ADD PICTURES OF THE ENVIRONMENT**

1. Implementation of Q-learning

Broadly speaking, the Q-learning algorithm requires two types of input and produces one kind of output. A reward signal is needed for punishing or rewarding the robot. The current (state, action) as well as the (state’, action’) after action was implemented was needed for the update of Q-table entry correspond to (state, action). When given a state, Q-learning will search the list of actions for the highest value and output the corresponding action.

The update function we used is as follows:[[3]](#endnote-3)

 1

In our implementation, the reward signal was provided by the contact switch. When an obstacle comes into close proximity with the contact switch, the switch will sends out the signal of 1, telling the system that a collision has happened and the agent needs to be punished. The vehicle will reverse for some time (such time can vary) to get the agent away from the obstacle. Collision detection is deactivated during such period so that it won’t stuck in an infinite loop of reversing into an obstacle.

The states were provided by the two distance sensors. They are accurate to 0.2 meters and ranged from 0 to 1.8 meters inclusive, thus 10 discrete data will be possible for each distance sensor and two sensors will provide us with 100 possible states, denoted by [leftS rightS], where leftS is the data from left sensor and rightS is the data from right sensor.

Five actions were provided for each state and will be denoted by [leftM rightM], where leftM is the torque for left motor and rightM is the torque for right motor. The five possible actions were: [100 100], goes straight; [100 -10], turn right; [-10 100], turn left; [80 -30] sharp right; [-30 80] sharp left. This gave us a 100 x 5 Q-table.

We updated the Q-table and chose an action every 0.1 second, which was the same with the sampling frequency of all our sensors. This frequency can be increased for higher precision. However, this increases requirement for computing power and training time.

The update function was a direct implementation of the function 1 above, however with some minor changes. The Q(s, a) in our update function was not really current state and action in our implementation, but the state and action chosen 0.1 second ago. Q(s’, a’) was the current state and action. And the reward was the reward signal from the last state. This is because we cannot accurately predict the next state our agent will be in; we can only update it after the next state has been reached.

The entire process was implemented in a Simulink simulation for easy monitoring, and all the variables, including the Q-table, was stored in the Simulink Data Store Memory block for convenient read and write.

1. (@manual{MATLAB,

   address = {Natick, Massachusetts},

   organization = {The MathWorks, Inc.},

   title = {{MATLAB 9.9.0.1592791 (R2020b) Update 5}},

   year = {2020}

   }) [↑](#endnote-ref-1)
2. (MathWorks Student Competitions Team (2021). Robotics Playground (https://github.com/mathworks-robotics/robotics-playground/releases/tag/20.1.4), GitHub. Retrieved March 4, 2021.) [↑](#endnote-ref-2)
3. <http://incompleteideas.net/sutton/book/ebook/the-book.html>, 6.5 [↑](#endnote-ref-3)