Universiteit Leiden
Opleiding Informatica

Design and Analysis of a

Controller for the iRobot Create

Name: Julius Koschny
Studentnr: 1417452

Date: 22/08/2016

1st supervisor: T.P. Stefanov
2nd supervisor: M.M. Bonsangue

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Design and Analysis of a Controller for the iRobot Create

Julius Koschny

Abstract

This thesis describes the design, modeling and analysis of an embedded controller for the iRobot Create [iRob]. The robot is put into an environment where it has to find a hill and climb it. The idea behind this task is to design a control algorithm that allows the robot to effectively search an area while avoiding obstacles and getting stuck.

The designed algorithm uses an array of integers to represent the environment. Using this array, the control algorithm marks where the robot has been as well as where obstacles are located. This is then used to decide where the robot has not searched yet and so to determine what direction the robot should move in.

After designing and implementing the controller, several experiments are done. Several different values are used for parameters that determine the robot's behaviour. These parameters are the drive speed, the size of the cells of the array and the difference in the angles in which the robot rotates.

The results of these experiments were somewhat inconclusive. The search times we recorder varied a lot. It would be better to repeat the experiments many times, however this could not be done due to time constraints. One conclusion we were able to draw from the experiments is that a drive speed of 100 mm/s is too much. This is because at this speed the control algorithm incorrectly detects the hill due to the high acceleration.

Acknowledgements

I would like to thank Todor Stefanov for guiding me in this project. He helped me with coming up with many of the ideas as well as solving many of the bugs I encountered during the coding process.

Also, I would like to thank my parents for pushing me to work on the project as well as helping me with some of the bugs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Robotics is becoming more and more important in today's society. Some robots are made for entertainment or research purposes. From robots that play football to androids that imitate human behaviour, modern technologies allow us to create robots that can perform various tasks with ever increasing precision and efficiency. Of course, not all robots are there just for the sake of creating them. Nowadays, robots are more and more frequently used to complete tasks that we, as humans, do not want to do. Some of the more advanced robots can efficiently mow your lawn or clean your floor. A good example of a robot used in this way is the iRobot Create [iRob]. Many people will know this circular robot as an automatic vacuum cleaner. These robots are not only made up of their hardware, they also contain software to read all the robot's sensors and drive its actuators. Part of this software is a control algorithm. This is the part that looks at all the values of the sensors and decides what the robot should do. This part is what this thesis is about.

1.1 Problem Statement

The problem addressed in this thesis is: Can we design a robust and efficient control algorithm for the iRobot Create [iRob] to complete a specific mission? The mission that this control algorithm makes the robot complete is the following: it makes the robot look for a hill in a given environment. Then it makes the robot climb the hill to its top. At the highest point the robot turns around and descends the hill. Once this is completed the robot is back on the ground. Then it terminates because its mission is complete.

During this whole mission, there are a number of constraints to take into consideration.

1. The mission should be completed as fast as possible. This constraint will also be taken as the quality measurement when evaluating the control algorithm in any experiments.

2. The robot should try to avoid collisions with obstacles as much as possible. A related constraint is that it should not show wall hugging behaviour. This means it should not repeatedly collide with obstacles to find a way around.

3. The robot should attempt to avoid wheel hazards. Meaning it always tries to keep its wheels on the ground.

4. It should avoid cliffs. While climbing or descending the hill, it is possible that the robot encounters an edge of the hill. It should not fall of this edge.

This mission and the constraints are taken from the book [JJS14]. However unlike the assignment in that book, this thesis also models the control algorithm and experiments with several parameters in order to optimise the control algorithm.

1.2 Contributions

This section will detail the work done towards solving the problem formulated in Section 1.1. These contributions consist of a few major parts. First, there is the design of a control algorithm for the iRobot Create that accomplishes the mission and satisfies the constraints. The second part consists of the modeling of the designed algorithm using finite state machines and the implementation of the control algorithm in C. The final part is the analysis and evaluation of the control algorithm by means of experiments.

The designed controller makes use of a grid to mark where the robot has been before. It also marks any obstacles in this grid. At first, the robot simply drives straight ahead. When it encounters and obstacle, the control algorithm will decide a new direction to drive in based on this grid.

The controller is modeled using finite state machines because it is also implemented in a very similar manner. The implementation in C uses a library that allows the control algorithm to easily interpret the sensor values provided by the robot's operating system. It also makes it very easy to tell the robot at what speed it should drive.

There are three experiments that have been performed to analyse and optimise the control algorithm. All of these consisted of running the control algorithm with different parameters in two different environments. The quality measurement is the time it took for the robot finish the whole mission. This means the time until the robot has found the hill, climbed it, and descended it again.

One of the parameters that we experimented on was the size of the cells of the grid. Before performing the experiments it seemed logical that sizes near the robot's diameter would yield the best results. The sizes

tried are 10 cm, 25 cm, 35 cm, 50 cm, 75 cm and 100 cm. Since the robot is around 35 cm in diameter, it was expected that this will yield the best results.

Another parameter was the size of the angles between the directions in which the robot would drive. The angles tried were 45°, 30°, 20°, 15°and 5°.

Finally, we experimented on the drive speed of the robot. By making the robot drive faster it would reach its destinations faster, however, if it drives too fast the controller will not be activated frequently enough to react to sensor input. The speeds tried were 20 mm/s, 40 mm/s, 60 mm/s, 80 mm/s and 100 mm/s.

1.3 Related Work

A lot of research has been done on subjects related to control algorithms such as the one this thesis is about. Some of those are listed below to give the reader an idea of what similar control algorithms exist for robots, how they are related to this thesis and how they are different.

1.3.1 Vector Field Histogram

The algorithm shown in the article [BK91] is an obstacle avoidance algorithm that uses the distance to surrounding objects and the location the robot is trying to reach in order to calculate the direction it needs to move. The smooth motions make this a common algorithm for implementing obstacle avoidance in robots.

Theoretically, the Vector Field Histogram could be used for the obstacle avoidance in the control algorithm presented in this thesis. However, the robot we use can only detect obstacles in front of itself. The Vector Field Histogram requires that the control algorithm has information about all obstacles surrounding the robot. So if we were to use the Vector Field Histogram, the robot would have to turn 360°in regular intervals in order to detect all surrounding obstacles. This is very time consuming, so it is not done for the control algorithm in this thesis.

1.3.2 Area Coverage

Another article closely related to this thesis is [GR01]. This article concerns itself with area coverage. This means trying to let a robot reach every location in a given area. This is useful for applications such as vacuum cleaning or lawn mowing. The algorithm presented in that paper divides the area into smaller cells. It then follows a spanning tree of the graph represented by the cells.

The algorithm presented in that article is similar to the control algorithm presented in this thesis. For searching the hill, we also need to cover much of the area the robot can drive around in. However, the hill will generally not be just one point, but it will cover a larger area. Therefore, it is likely much faster not to reach every single spot in the robot's environment. Additionally, the algorithm presented in that paper needs a

1.3.3 Multi-Robot Search

One method for searching a target using robots is shown in [PM08]. Note that this method uses multiple communicating robots to accomplish this task. The algorithm is inspired by particle swarm optimisation. The algorithm is highly effective for finding a target, but unlike this thesis, it uses multiple robots. Therefore, it cannot be used directly in this thesis.

1.4 Thesis Overview

This thesis comprises of several chapters. Chapter 2 includes some background information on the iRobot Create and the software used; Chapter 3 discusses our implementation; Chapter 4 presents our experiments; Chapter 5 draws conclusions from these experiments; Chapter 6 explains what can be done to extend this thesis.

Chapter 2

Background

In this chapter we provide some background information on the iRobot Create, because this is the robot for which the control algorithm was designed. A complete description of how the robot works can be found on the official web-page [iRob] Also, the tools used to create this control algorithm will be explained.

2.1 The iRobot Create

The iRobot Create is a circular shaped robot with a number of sensors and a few actuators. In Figure 2.1 we show a schematic view of the bottom of the robot. In Figure 2.2 we show a schematic view of the top of the robot. It has three wheels, two of which are connected to motors. This section will describe which features of the robot can be accessed with the used interfacing software. It should be noted that the experiments in Chapter 4 were done using a model of an iRobot Create and its sensors rather than a real robot. The model uses a simulator to simulate what the behaviour of the real robot would be.

The robot has a large number of sensors. These sensors are read by the robot's operating system. The interfacing software then interprets these sensor values and turns them into useful variables for the control algorithm to use.

Firstly, the interface has a variable that represents the angle relative to the starting orientation of the robot. This angle is measured in degrees and ranges from -180°to +180°. The starting orientation is defined to be 0°. This variable can be used to detect when the robot has turned by a desired angle by comparing its value to the value at the start of a turning maneuver.

Secondly, there is a variable representing the distance traveled by the robot. Its value starts at 0 and will keep increasing whenever the robot moves. It does not make a distinction between moving backward or forward.

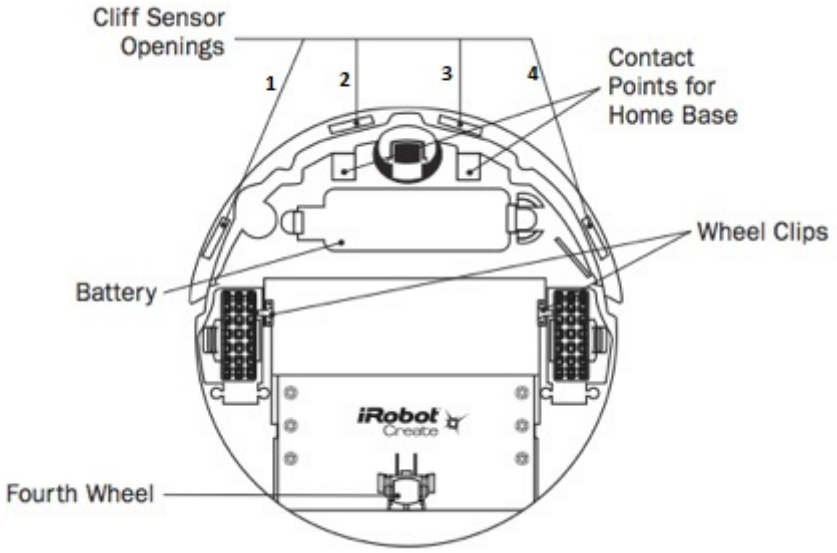


Figure 2.1: Schematic View of the Bottom of the iRobot Create [iRoa]

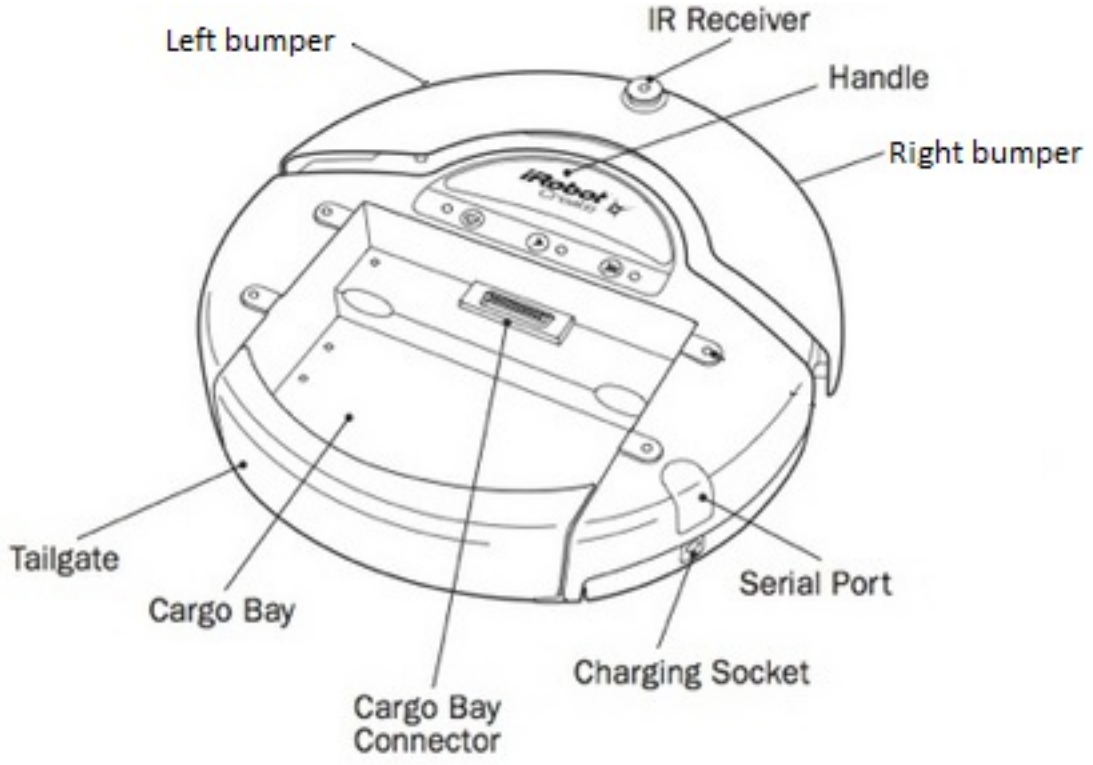


Figure 2.2: Schematic View of the Top of the iRobot Create [iRoc]

It will only increase. The values it can take are all integer values.

Thirdly, the robot has two bump sensors, as shown in Figure 2.2. These are pressure sensors. The variables corresponding to these sensors provided by the interface have Boolean values. When one of the sensors touches an object, that sensor's corresponding variable gets the value `true`. At any other time their variables have the value `false`.

Furthermore, for each wheel there is a sensor that can be used to detect whether that wheel touches the ground: one for the left motorised wheel, one for the right motorised wheel and one for the front wheel. The locations of the wheels are shown in Figure 2.1. The interface provides Boolean variables that have the value `true` when their respective wheel is not touching the ground. Otherwise, the variables have the value `false`. A fourth wheel can be added to the robot, but this is not used in this thesis.

Also, there is an infrared (IR) emitter and receiver facing forward. The receiver is capable of detecting whether there is a wall or a different obstacle in front of the robot. This can be seen in Figure 2.2 The interface has a variable of which the value can range from 0 to 4095. The higher the value is, the closer the obstacle is in front of the robot. It should be noted that this sensor does not just detect obstacles straight ahead of the robot. It also increases its value if there are objects diagonally in front of it. There is another variable that is a Boolean representation of this wall value. It will become `true` whenever the analog value is greater than a preset threshold.

Furthermore, the robot has four more of these IR emitters and receivers. These are used as cliff sensors. The values of the variables corresponding to these sensors also range from 0 to 4095. The greater the value, the closer that sensor is to the ground. These sensors also have a Boolean representation. The value of this representation will be `true` when the integer value drops below a certain threshold. The robot possesses four of these sensors. Their locations are indicated in Figure 2.1.

Another important sensor is the accelerometer. This sensor detects force in all directions. It is represented as three floating point values. It has an x direction, which is the same direction the robot is facing. The y direction is the direction perpendicular to the x direction but still horizontal. The z direction is the vertical direction. The x and y directions have as default values 0. The z direction is influenced by gravity, meaning it has a default value greater than 0.

Finally, the robot's buttons are also represented as variables by the interface. Their values are `true` when the buttons are pressed and `false` otherwise. These buttons are located at the top of the robot. It possesses a play/pause button, a forward button and a stop button. Only the play/pause button is implemented to make the robot stop moving when pressed. It will resume whatever it was doing before pausing when the button is pressed again.

The robot does have more sensors than the ones listed above. However none of those are used in our implementation. Those sensors include ones for detecting its battery state, its docking and charging state, the cargo bay inputs, sensors detecting the currents in its system and virtual wall sensors.

The actuators of the iRobot Create are very few. All it has are a left motorised wheel and a right motorised wheel. The robot's interface allows the controller to set the desired speeds for each wheel individually. The robot's operating system will then apply enough force to reach these speeds. More actuators could be added using the cargo bay, however this feature is not used in this thesis.

2.2 Software Tools

This section briefly describes the different tools used in the implementation and simulation of the control algorithm. The simulation of the control algorithm is used to conduct the experiments in Chapter 4. The modeling of the control algorithm does not use any special tools.

2.2.1 LabVIEW and CyberSim

To simulate the iRobot, a software called CyberSim [JJS14, pp. 30–40] is used. This software is based on LabVIEW [lab]. LabVIEW is a software that allows you to easily create user interfaces displaying many different inputs and outputs. In the case of CyberSim, one can see where the robot currently is and how it is moving. It also shows the values of all the sensors. This allows you to get an idea of what the robot is doing at any given time.

The CyberSim user interface is shown in Figure 2.3. In the area with label ”Environment”, it shows the environment the robot is in. It also has a circle that turns bright green when a simulation is running. To the The values of all the robot's different sensors are shown to the right, in the area labeled ”Variables”. A few buttons used for starting, pausing and stopping the simulation are shown below that in the area labeled ”Buttons”. In the top left of the user interface there is an area labeled ”Files”. This area contains three inputs. The top one allows you to select an XML file to be used as the robot's environment. The middle one contains a DLL file that contains the controller. The final one allows you to select another XML file to record a specific situation. The files used here can be remembered by clicking the ”Save Configuration” button. The final part of the interface is the area labeled ”Info”. This area shows a light to indicate whether the simulation is running and the time that has passed in the simulation. Any features that are not described are not important for this thesis.

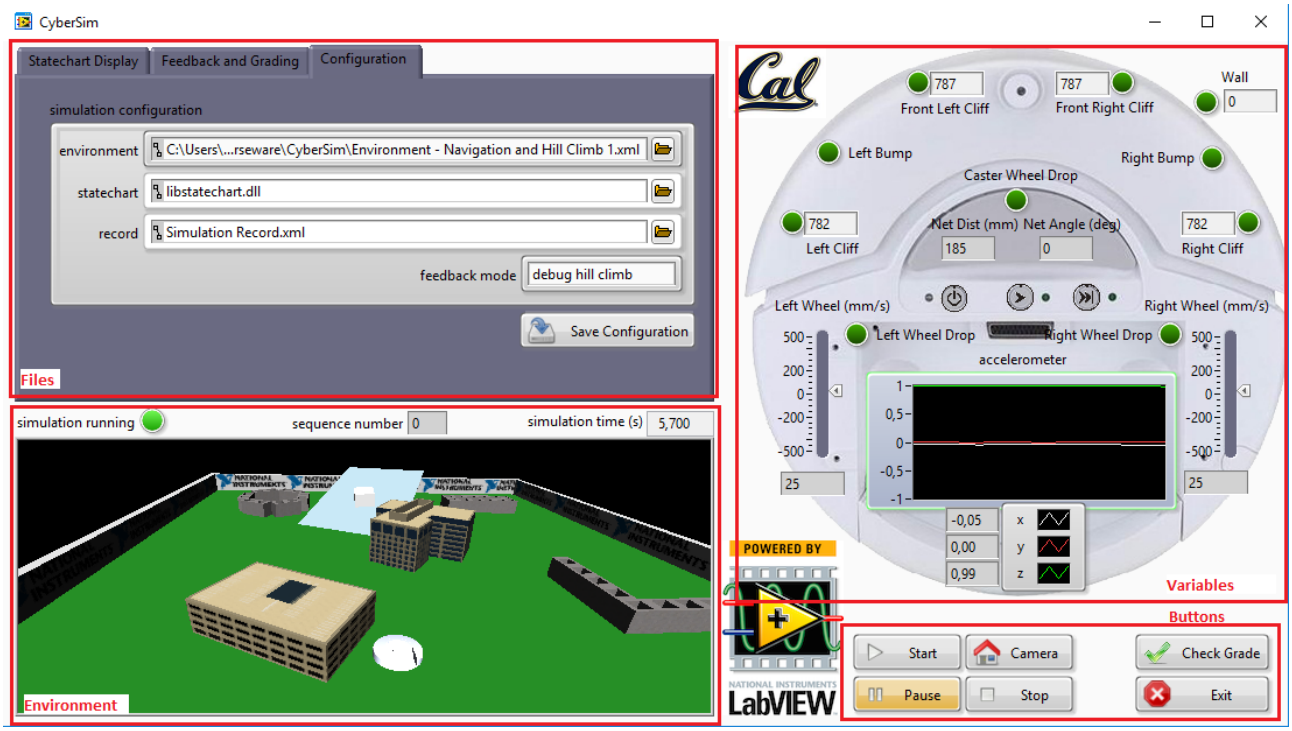


Figure 2.3: The CyberSim user interface

2.2.2 Visual Studio

The one thing that CyberSim does not show is the internal variables of the robot's control algorithm. Since the code is written in the IDE Visual Studio [vis], all the variables not shown in CyberSim's interface can be checked using Visual Studio's debugging mode. Additionally, this debugging mode enables the developer to set breakpoints. This means the developer can make the program stop at specific points in order to evaluate the state and the internal variables used by the controller.

CyberSim offers an easy way to implement the control algorithm in C and to program the robot using Visual Studio. It provides a template for a Visual Studio software project. This template contains all the necessary code for the implementation of a new controller as well as an example control algorithm to give the developer a rough idea of how to get started.

Chapter 3

Modeling, Design and Implementation

This chapter explains the design and modeling of the control algorithm. First, in Section 3.1 we explain how the different aspects of the control algorithm are modeled. In Section 3.2 we explain the design of the part of the control algorithm that makes the robot search for the hill. Finally, Section 3.3 explains the design of the part that makes the robot climb and descend the hill. Both parts of the algorithm are also illustrated using a finite state machine (FSM).

3.1 Modeling the Controller

This section explains how the controller is modeled and why certain decisions are made regarding the modeling. A large part of the control algorithm is modeled using two Moore finite state machines. Section 3.2 explains how the algorithm that makes the robot search for the hill works and shows the FSM for that algorithm. Section 3.3 explains how the algorithm that makes the robot climb the hill works. That section also shows the FSM for that part of the control algorithm.

In order to analyse the controller, we can model it as a FSM. This works very well because the implementation consists of states and transitions just like a FSM. Of course there are some ways in which the algorithm deviates from a pure FSM. We make a distinction between several aspects of the robot and its environment. Some of those will be modeled using the FSM.

The first aspect to take into consideration is the external environment. This means the area the robot can search in, the obstacles in this area, the location of the hill and the location of the robot. Most of this does not change over time, except for the location of the robot. The way we receive information about the environment is through the robot's sensors. The information from these sensors is then interpreted by the robot's operating

system and passed to the control algorithm through an interface. This is explained in detail in Section 2.1. As a result of seeing the world through sensors, initially we only have a very limited knowledge of the external environment. As the robot explores more areas, however, we receive more information, allowing us to eventually find the hill.

The second aspect are the internal variables. These variables are used to store information about the external environment. The most important variable is a two-dimensional array in which we mark all the obstacles and the locations we have already visited. Also, we calculate and store the coordinates of the robot relative to the robot's starting position. We define this starting position to be in the middle of the array. The coordinates are measured using floating point variables that represent a number of cells. In order to represent the whole controller as a pure FSM, one could simply take the cross product of all possible states this array and these coordinates can be in as well as the actual state of the robot. However, it is easy to see that this would become a massive FSM that will not serve any purpose in making it simpler to analyse the controller.

The third and final aspect is the actual finite state machine. The type we use for the model is a Moore finite state machine. This is one where the output at any given moment depends solely on the state at that moment. We use this type despite the fact that modifying the internal variables is done on a state transition. In the model, we only show the effects of sensor inputs and the control outputs to actuators. This makes the FSM a lot easier to understand than if we were to include all the changes of internal variables. A Moore FSM consists of states, transitions and actions. Every state has an action corresponding to it. In our case these actions consist of changing the speed of the wheels of the robot. For clarity, the actions shown in the FSM will be shown not as the speed of the wheels but as commands such as 'drive forward', 'turn' and so on.

3.2 The Search Algorithm

The control algorithm is split into two parts: one for finding the hill, called the search algorithm, and one for climbing it, called the climb algorithm. The search algorithm is the more complex of the two. This is because, despite having less states, it needs to modify and read more of the internal variables.

The basis of the search algorithm is a grid containing information about where the robot has been and the location of obstacles. This grid is represented as a two-dimensional array of integer variables. The robot drives straight ahead until it encounters an obstacle. When this happens, it will choose a new direction to move in based on the values of the grid variables.

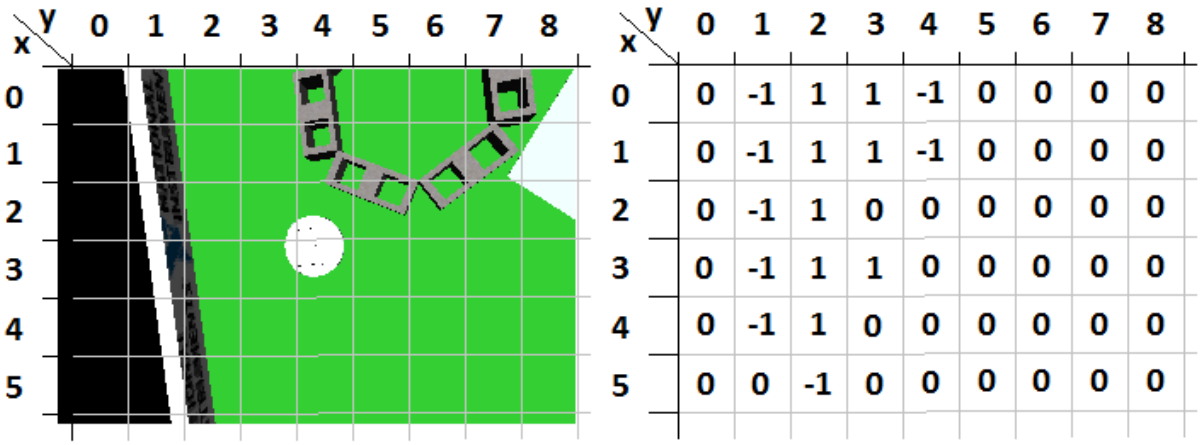


Figure 3.1: Example of an environment and a corresponding grid

3.2.1 The Grid

The grid is represented as a two-dimensional array containing integer values. If an obstacle is encountered at any point while driving, the controller will mark the grid cell at the apparent location of that obstacle with -1. Also, the values of any cells the robot has passed over will be incremented by 1. As a result, the grid will get higher and higher values the more times the robot passes through certain areas. This tells the robot that it should rather go somewhere else. An example of the grid is shown in Figure 3.1. In this figure, a part of an environment is shown on the left. On the right, the grid corresponding to this part of the environment can be seen. The cells are referenced first by their x coordinate and second by their y coordinate: (x, y). By the cells in column 1 containing -1 and the cell $(5, 2)$ we can see that the robot has detected the left wall several times. The value -1 in cells $(0, 4)$ and $(1, 4)$ shows us that the robot has also detected the obstacle in those cells. The various cells containing 1 show cells that the robot has already visited. Finally, to the left and the right of the cells with values 1 or -1, you can see a number of cells containing 0. These cells have not yet been visited.

Another important thing to note is that the robot does not know its own location or orientation relative to the hill it has to find or anything else in the environment. This means we can not just make the grid cover exactly the area the robot has to search and position the robot at the correct location. Instead, we define the robot's initial position to be at the center of the grid. Then, we calculate everything else relative to that location. We also do not initially know the locations of the boundaries of the grid. The outside walls will simply be marked the same way obstacles are marked because the robot cannot make a distinction between the two.

The cells in the grid the robot has already visited are marked whenever we transition from a state where the

```
1  if |sin(angle)| > |cos(angle)|
2      i = 0
3      while i * cellSize < distanceMoved * sin(angle) do
4          gridX = floor(startX + i)
5          gridY = floor(startY + i * tan(90 - angle))
6          if grid[gridX][gridY] != -1
7                  grid[gridX][gridY]++
8          end if
9          i++
10     end while
11 else
12     i = 0
13     while i * cellSize < distanceMoved * cos(angle) do
14         gridX = floor(startX + i * tan(angle))
15         gridY = floor(startY + i)
16         if grid[gridX][gridY] != -1
17                 grid[gridX][gridY]++
18         end if
19         i++
20     end while
21 end if
```

Figure 3.2: Pseudo-code for the marking of the grid

robot moves forward or backward. The pseudo-code for the marking of these cells can be seen in Figure 3.2. In this pseudo-code, the variable called `angle` is simply the angle along which the robot has driven, the variables `startX` and `startY` are the coordinates of the robot at the beginning of its drive maneuver. The variables `gridX` and `gridY` are the coordinates of the robot at the end of its drive maneuver, the time when this routine is called. The variable `distanceMoved` is simply the distance the robot has moved since the start of the its drive maneuver. The variable `cellSize` is the size of the cells of the grid. The calculations in lines 3 and 13 make sure that the control algorithm stops marking cells when it has marked all cells between the location of the robot at the start of the drive maneuver and the location at the end of this maneuver.

It should be noted that there are two different calculations for the coordinates of the cells to mark. In Figure 3.2, the first calculation can be seen in lines 4 and 5 while the second one is seen in lines 14 and 15. The coordinate of the axis along which the robot covers a larger distance is incremented by one for each iteration of the `while` loops. The other coordinate is then calculated based on the distance along this first coordinate and the angle in which the robot has driven. If the condition of the `if` statement in line 1 is true, we have moved more along the x-axis than along the y-axis of the grid. If the robot has moved more along the x-axis, the control algorithm goes into lines 2 to 10 of the pseudo-code. If the robot has moved more along the y-axis, the control algorithm goes into lines 12 to 20. There are two differences between lines 2 to 10 and 12 to 20. The first of these differences is the condition in lines 3 and 13. The other is the calculation of the coordinates of the cells that need to be marked. This is done in lines 4 and 5 as well as in lines 14 and 15. This is illustrated using Figure 3.3. In both grids shown in this figure, the robot has started its maneuver

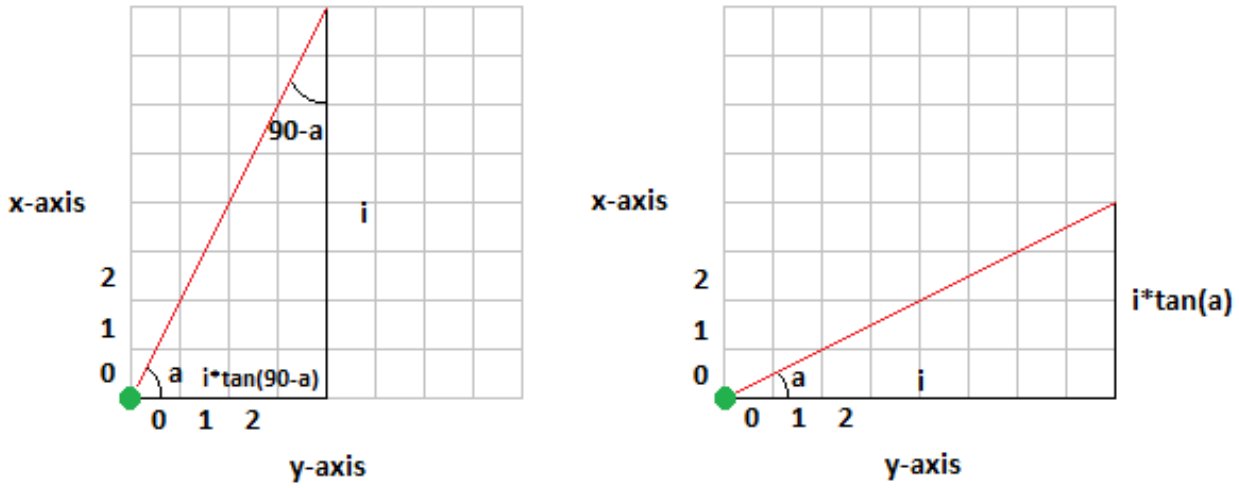


Figure 3.3: Illustration of the calculation of coordinates for marking cells

at the green spot. Then it has moved along the red diagonal. This means the variables `startX` and `startY` from the pseudo-code are the coordinates of that green spot. In Figure 3.3, the variable `angle` is replaced by a. For the coordinate of the axis along which the robot moved more, the control algorithm simply adds the variable i. For the coordinate of the axis along which the robot moved less, the control algorithm has to calculate how many cells the robot has moved if it has moved i cells along the other axis. In Figure 3.3, the left grid shows the case where the robot has moved more along the x-axis while the right grid shows the case where the robot has moved more along the y-axis. In the case shown on the right, we can use geometry to figure out that, if we moved i cells along the x-axis with an angle of a to the x-axis, then we moved $i * tan(a)$ along the y-axis. This can be seen in the pseudo-code in line 14. In the case shown on the left, simply using a for this calculation would result in the formula $i/tan(a)$ for the coordinate along the y-axis. However, if a is 90°, $tan(a)$ will be 0, so the calculation of the coordinate results in an error. Therefore, we calculate the angle opposite from a using geometry. This angle is $90° - a$. Now we can use geometry again to find out that the distance the robot moved along the y-axis is equal to $i * tan(90 - a)$. In the pseudo-code, this is shown in line 5.

3.2.2 Choosing a Direction

After encountering an obstacle, once the cells are marked, the robot will have to choose a new direction to move in, in order to avoid the obstacle. The control algorithm does this by checking the cells of the grid in several directions. What these directions are will be discussed in section 4.1. The control algorithm will loop through the angles corresponding to the different directions. For each direction it will sum up the values in

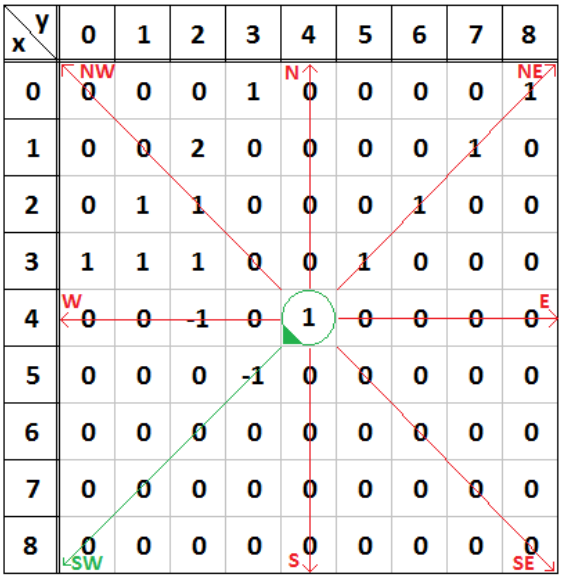


Figure 3.4: Example search grid

the cells until it encounters -1 or until it goes beyond its maximum search distance. Afterwards, if -1 is found, the sum is incremented by the highest number currently present in the grid multiplied by the number of cells it did not check within the reach. This is done since a lower number represents a better direction, and encountering -1 earlier on would otherwise result in a lower number than if the whole reach is searched.

For its new direction the robot will choose the direction that has the lowest total sum. This should work because a lower sum means the robot will probably cover more unexplored cells than if it were to choose a direction with a higher sum. As a result, every time the robot passes a cell, it will become less likely that the robot passes that same cell in the future. This means that the robot will tend to drive toward areas it has not visited yet.

The approach is illustrated using Figure 3.4. In this example, the robot is located in cell $(4, 4)$ of the grid. This is shown by the green circle. It is oriented to the SW direction. This is indicated by the green triangle facing in that direction. There it has just encountered and marked an obstacle. Since it encountered an obstacle there, it does not have to check that direction. The directions it will check are W, NW, N, NE, E, SE and S. For simplicity, we assume a maximum search distance equivalent to 4 cells. If the controller now checks the cells to the bottom, it will see four zeros. That direction will have a total sum of 0. The same goes for the N, E and S right directions. . The NW direction will yield a total sum of 1, the NE direction will yield 4. When it checks W, it will encounter -1 in cell $(4, 2)$ before finishing. So it will increase that sum depending on the number of cells it would not reach in that direction. This number of cells is multiplied by 2, as that is the highest value in the whole grid. This results in a total sum of 6 in that direction.

The pseudo-code for this algorithm can be seen in figure 3.5. The code starts by setting some values for the

```
1  bestAngle = 0
2  bestCellValue = INT_MAX
3  for each direction do
4          angle = directionNumber * angleSize + startAngle
5      currentCellValue = 0
6      totalCellValue = 0
7      xOffset = 0
8      yOffset = 0
9
10     while currentCellValue != -1 && xOffset ^ 2 + yOffset ^ 2 <= searchDistance ^ 2 do
11         if |sin(angle)| > |cos(angle)|
12                 if sin(angle) > 0
13                 xOffset++
14             else
15                 xOffset--
16             end if
17             yOffset += tan(90 - angle)
18         else
19             xOffset += tan(angle)
20             if cos(angle) > 0
21                 yOffset++
22             else
23                 yOffset--
24             end if
25         end if
26
27         currentCellValue = searchGrid[x + xOffset][y + yOffset]
28         if currentCellValue != -1
29             totalCellValue += currentCellValue
30         end if
31
32         if totalCellValue < bestCellValue
33             bestCellValue = totalCellValue
34             bestAngle = totalAngle
35         end if
36     end while
37 end for
```

Figure 3.5: Pseudo-code for choosing a direction

initial variables. It will then loop through all the possible directions. The angle corresponding to the direction is calculated. What the `angleSize` is will be explained in section 4.1. Then, the program will go through all the cells in that direction within the `searchDistance`. This distance is set to 500 cm. The reason for adding this maximum distance is that otherwise the robot would have a tendency to drive to the center of the search array. This would not make sense, since this center is always where the robot started. The calculating of the coordinates is done in the same manner as with the marking algorithm.

3.2.3 Detecting the Hill

In order to find and climb the hill, of course we need to be able to detect it. This is done using the robot's accelerometer. How the accelerometer works is explained in Section 2.1 If the robot is on the hill, it is tilted. This means gravity will affect the x and y values of the accelerometer. Also, the gravity working on the z variable of the accelerometer will be less. By evaluating $x^2 + y^2 >= threshold$, we can check whether the total tilt is enough to say we are on the hill. The threshold was determined empirically. It is set to 0.02. The reason for using this formula rather than simply the z variable of the accelerometer is because the value of the z variable seems to be very noisy. As a result, only measuring the z variable would be fairly unreliable. Using Pythagoras' theorem we can see that by using the formula shown above, we get the square of the total force of the x and y variables.

3.2.4 The Finite State Machine

The finite state machine of the Search Algorithm consists of several states and many transitions. These are shown below. The FSM is visualised in Figure 3.6. The states and transitions are described in detail below.

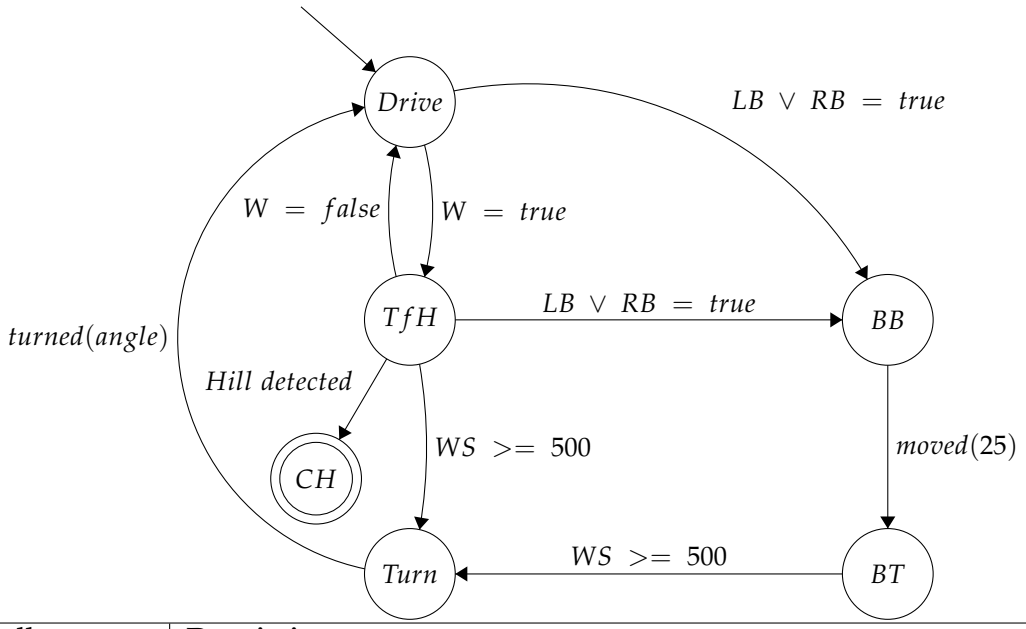
Drive Drives forward at full speed. This is the initial state. The speed at which it drives is a parameter that can be changed. This is shown in Section 4.3 as one of the experiments.

- Transitions to state *Test for Hill* if the wall sensor's Boolean variable has the value `true`.
- Transitions to state *Bump Backup* if on of the bump sensors has the value `true`.

Test for Hill When the wall sensor detects a wall, this may also be the slope of the hill rather than a wall. Therefore, we drive forward at half speed to check whether the wall sensor will keep increasing or whether the hill will be detected.

- Transitions to state *Drive* if the wall sensor's value becomes `false`.
- Transitions to state *Bump Backup* if one of the bump sensors has the value `true`.
- Transitions to state *Turn* if the analog wall signal is greater or equal to a certain threshold. This threshold is set to 500. This value was determined empirically. At this value it is always clear that the detected wall is not the hill.
- Transitions to the initial state of the Climb Algorithm if it detects the hill. See Section 3.2.3 for how the hill is detected.

Turn Turns the robot. The angle to turn is calculated using the grid. See Section 3.2.2.



State	Full name	Description
Drive	Drive	Drives forward at a constant speed
TfH	Test for Hill	Drives forward slowly to avoid bumping into a wall, may also detect the hill in this state
BB	Bump Backup	Moves backward to make room to maneuver after bumping into an obstacle
BT	Bump Turn	Turns to face the wall the robot bumped into in order to mark it in the grid
Turn	Turn	Turns towards the new direction the control algorithm has generated
CH	Climb Hill	Transitions to the Climb Algorithm

Figure 3.6: FSM of the search algorithm

- Transitions to state *Drive* when it has turned by the calculated angle.

Bump Backup When the robot bumps into an object that its wall sensor did not pick up, it will drive backwards a bit.

- Transitions to *Bump Turn* when it has moved 25 mm. At this distance, the robot's bump sensors will no longer detect the wall even after turning.

Bump Turn Turns toward the obstacle it bumped into so the obstacle can be marked in the correct location in the grid.

- Transitions to state *Turn* if the analog wall signal is greater or equal to 500. This is the same value as in the transition from state Test for Hill to state Turn.

Climb Hill The hill has been found. The robot will go to the next set of states described in Section 3.3

3.3 The Climb Algorithm

After finding the hill, the next part of the mission is to climb the hill. The algorithm for climbing the hill contains no routines as complex as the Search Algorithm. It does, however, contain significantly more states.

This is due to the fact that we have to drive around obstacles in this part of the algorithm whereas the Search Algorithm simply chooses a different direction when it hits an obstacle. Apart from the obstacle avoidance, the algorithm consists of three parts. The first part is determining which direction the robot needs to drive in to get to the top of the hill. The second part is to actually climb the hill. This is done by driving forward until the robot reaches the top of the hill. Then the robot turns around and starts the final part of the algorithm. The robot drives back down until the it reaches the bottom of the hill. Detecting the bottom of the hill is done exactly the opposite way of detecting the hill itself as shown in Section 3.2.3. This means the robot will drive down the hill until the hill is no longer detected. Once this is the case, the robot has completed its mission.

3.3.1 Determining the Up Direction

The first thing that the climb algorithm has to do, once the hill has been detected, is to find out in which direction the top of the hill is. To figure this out, we make use of the robot's accelerometer. The robot turns and checks in which direction it measures the greatest acceleration in the x variable of the accelerometer. How this accelerometer works is explained in Section 2.1. If the robot is tilted, gravity will no longer work only on the z variable of the accelerometer. It will also work on the x and y variables. We want to find out in which direction the gravity acceleration shown by the x variable is the greatest, because this variable shows the acceleration in the same direction the robot is facing. To find out in which direction this is variable has the greatest value, the control algorithm makes the robot turn clockwise first. While turning, it saves the greatest value it detects of the x variable. It also saves the direction in which it detects this value. As soon as it detects that the value of the x variable of the accelerometer is lower than this greatest value it has stored, the control algorithm stops the robot from turning clockwise and makes it turn counter-clockwise instead. Then it again saves in which direction it detects the greatest value of the x variable as well as the corresponding value of the x variable. Also, the robot again stops turning once the x variable is decreasing. After searching both clockwise and counter-clockwise, the control algorithm checks in which of the two turning maneuvers it detected a greater value of the x variable. Then it saves the direction corresponding to the greater of these two values as the direction in which the robot needs to drive to get to the top of the hill.

3.3.2 Obstacle Avoidance

While climbing the hill, the robot will also have to avoid an obstacle. However, since the main focus of this thesis is the algorithm that searches for the hill, this has been done in a rather simple way. Whenever the robot detects an obstacle, it will turn 60°away from that obstacle. This value was determined empirically. The avoidance seemed to work best at this value. After turning away from the obstacle, the robot drives forward for 200 mm. Then the control algorithm makes the robot turn back up. Once this is done, the robot resumes

its normal hill climbing. This way of avoiding obstacles means that the robot can only deal with one obstacle at a time. If another obstacle is encountered while the robot is busy avoiding one, it will most likely not be able to complete its mission. A good thing, however, is that this way of avoiding obstacles also works when the robot encounters cliffs that are not at the top of the hill. It can avoid these cliffs in the same way that it avoids obstacles. While going down, the control algorithm uses the same way of avoiding obstacles as it does while going up.

3.3.3 The Finite State Machine

The FSM for the Climb Algorithm is visualised in Figure 3.7. The states and transitions of this FSM are explained in detail below.

Search Up Right Turns clockwise and remembers the angle at which the greatest value is detected in the x variable of the accelerometer. See Section 3.3.1.

- Transitions to state *Search Up L* if the value of the x variable decreases.

Search Up Left Turns counterclockwise and remembers the angle at which the greatest value is detected in the x variable of the accelerometer. See Section 3.3.1.

- Transitions to state *Turn Up* if the value of the x variable decreases.

Turn Up Turns clockwise toward the top of the hill.

- Transitions to state *Drive Up* if it has reached the direction determined as the direction it needs to drive to to get to the top of the hill. See Section 3.3.1.

Drive Up Drives forward at full speed.

- Transitions to state *Turn Down* if either the front left or the front right cliff-sensor has the value `true`. These sensor values mean that the robot has reached the top of the hill.
- Transitions to state *Up Avoid Backup* if any of the left bump, right bump, left cliff or right cliff sensors have the value `true`.

Turn Down Rotates clockwise.

- Transitions to state *Drive Down* if it has turned to the direction it needs to drive in to get to the bottom of the hill. See Section 3.3.1.

Drive Down Drives forward at full speed.

- Transitions to state *Done* if the hill is no longer detected.

- Transitions to state *Down Avoid Backup* if any of the left bump, right bump, left cliff or right cliff sensors have the value `true`.

Up Avoid Backup Drives backward.

- Transitions to state *Up Avoid Turn* if it has moved 25 mm. This is the same value as in state Bump Backup of the Search Algorithm.

Up Avoid Turn Rotates away from the obstacle. This means it will rotate clockwise if the obstacle or cliff is detected on the left of the robot and counter-clockwise if it is detected on the right of the robot.

- Transitions to state *Up Avoid Drive* if it has turned 60°as explained in Section 3.3.2.

Up Avoid Drive Drives forward at full speed.

- Transitions to state *Turn Up* if it has moved 200 mm. This is also explained in Section 3.3.2.

Down Avoid Backup Drives backward.

- Transitions to state *Down Avoid Turn* if it has moved a set distance. This distance is 25 mm as explained in Section 3.3.2.

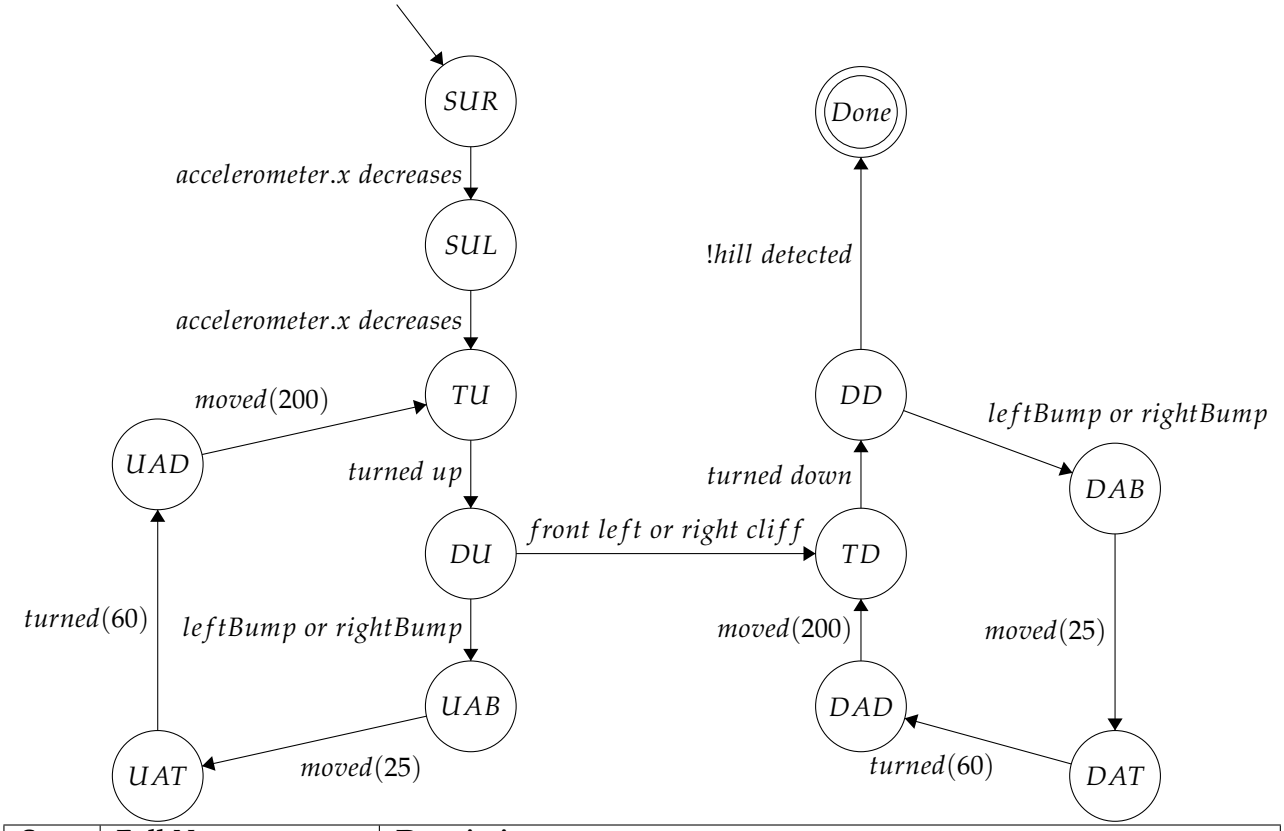
Down Avoid Turn Rotates away from the obstacle in the same way as in state Up Avoid Turn.

- Transitions to state *Down Avoid Drive* if it has turned 60°as explained in Section 3.3.2.

Down Avoid Drive Drives forward at full speed.

- Transitions to state *Turn Down* if it has moved 200 mm as explained in Section 3.3.2.

Done This is the final state. It is reached when the robot has completed its mission. In this state the robot will stop moving.



State	Full Name	Description
SUR	Search Up Right	Searches for the direction the robot needs to drive in when driving up by turning clockwise
SUL	Search Up Left	Searches for the direction the robot needs to drive in when driving up by turning counter-clockwise
TU	Turn Up	Turns toward the direction determined in states Search Up Right and Search Up Left
DU	Drive Up	Drives toward the top of the hill
UAB	Up Avoid Backup	Moves backward after bumping into an obstacle or detecting a cliff
UAT	Up Avoid Turn	Turns 60°away from the obstacle or cliff it detected
UAD	Up Avoid Drive	Drives forward 200 mm to avoid the detected obstacle or cliff.
TD	Turn Down	Turns 180°away from the direction determined in states Search Up Right and Search Up Left
DD	Drive Down	Drives toward the bottom of the hill
DAB	Down Avoid Backup	Equivalent to Up Avoid Backup, but is reached while driving down
DAT	Down Avoid Turn	Equivalent to Up Avoid Turn, but is reached while driving down
DAD	Down Avoid Drive	Equivalent to Up Avoid Drive, but is reached while driving down
Done	Done	The robot has completed its mission and stops moving

Figure 3.7: FSM of the climb algorithm

Chapter 4

Experiments

This chapter presents the experiments we have performed. The experiments were performed in order to analyse for which values of certain parameters the control algorithm performs best. After analysing for these values, we optimise the control algorithm by choosing the values for which the control algorithm completes its mission the quickest. In all of the experiments, the time it took the robot to find the hill, climb it and descend it is taken as the measure of quality. For each of the parameters we experimented on, the algorithm was run a number of times with different values in those parameters. The parameters we experimented on are the size of the angles the robot can turn in, shown in Section 4.1, the size of the cells in the grid, shown in Section 4.2, and the speed at which the robot drives, shown in Section 4.3.

In order to perform the experiments, first we set the values for the parameters. The exact values are shown in the sections about the individual experiments. After setting these values, a breakpoint is set so CyberSim will be interrupted as soon as the robot has completed its mission. Next, we start CyberSim. Then, we choose an environment. After this, the "start" button is clicked. CyberSim will then start simulating the robot's behaviour in the given environment with our control algorithm. As soon as the robot has finished its mission, the simulation will be interrupted. We can then read the time it took the robot to find, climb and descend the hill. This time is used to measure the quality of the control algorithm with the parameters used in that experiment.

All the experiments presented in this chapter have been executed in two different environments. The first one is called *Navigation and Hill Climb 1* (Figure 4.1) in CyberSim and the second *Navigation and Hill Climb 3* (Figure 4.2). The only actual difference between these two environments is the starting location of the robot. It would have been better to have multiple different environments in order to get more reliable results, but because there were no other environments available, only these two were used.

Many of the experiment results shown in the next few sections have an asterisk next to the search duration.

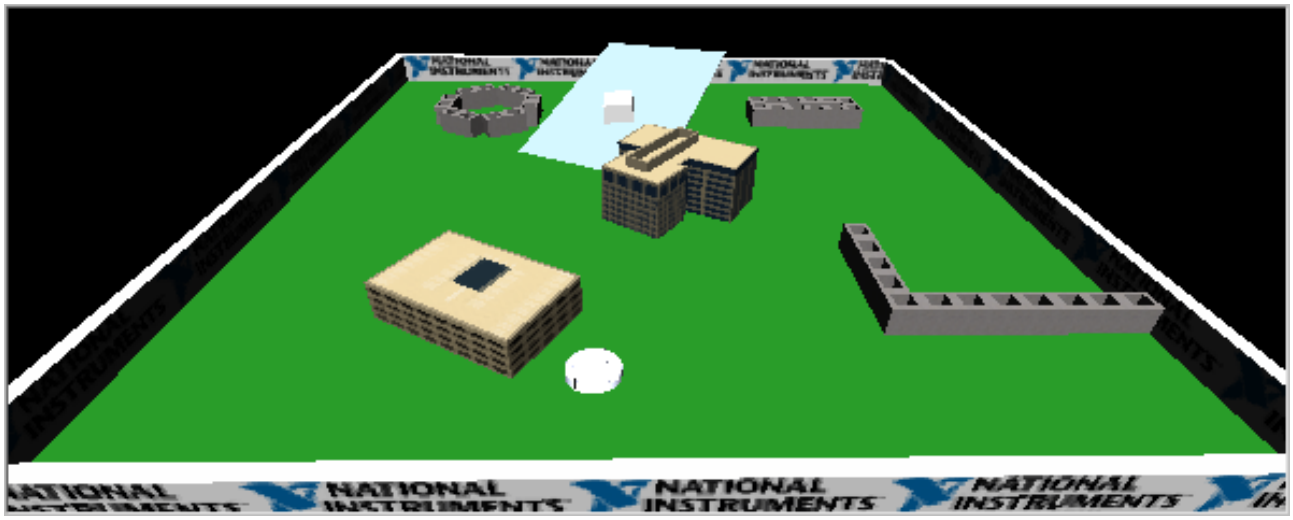


Figure 4.1: Environment - Navigation and Hill Climb 1

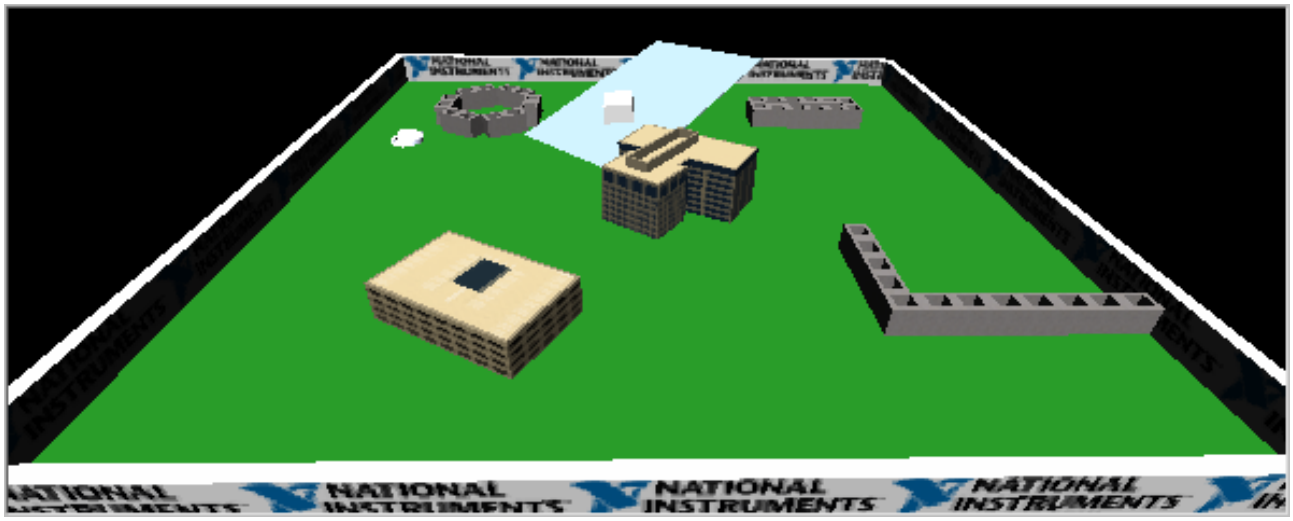


Figure 4.2: Environment - Navigation and Hill Climb 3

This indicates that, while the control algorithm did manage to find the hill in those experiments, it failed to climb and descend it properly. Therefore these experiments only show the time it took to find the hill. In all of these experiments where the climbing and descending of the hill failed, the robot started climbing the hill very close to the sides of the hill. The reason why this problem was not solved in this thesis is due to time constraints. Because the experiments tend to take a long time, we decided to focus on running as many experiments as possible. This meant there was no time to address these hill climbing issues. This decision was made because this thesis focuses on the searching of the hill.

Also, a few of the results are missing. This is because the robot did not find the hill in those experiments at all. There were two causes for this. In three of the experiments, the search took longer than $1.5 * 10^4$ seconds. Around this point, CyberSim shows an error message saying it ran out of memory. The failed experiments shown in Section 4.3 failed for a different reason, as described in that section.

Angle Size [°]	Nr. of Directions	Search Duration 1 [s]	Search Duration 3 [s]
5	72	2146*	5341*
15	24	—	6061
20	18	655.8*	1453*
30	12	$1.384 * 10^4$	1539
45	8	616.0	2551*

Table 4.1: Results of the angle size experiment

4.1 Angle Size

The first parameter to experiment on was the size of the angles the robot can drive in. In the code, this is represented as an integer number counting the number of directions the robot can drive in. To calculate the size of these angles, simply divide 360°by the number of directions. For example if the number of directions is 8, the size of the angles is $360° / 8 = 45°$.

The angles that were used are 45°, 30°, 20°, 15°and 5°. It is expected that bigger angles will result in lower precision. However it may occur that at very small angles the robot will take considerable time to calculate where it will go next. For the drive speed parameter, 40 mm/s was used during all these experiments. The reason is that, at this speed, the robot did not seem to have any unnecessary collisions with obstacles. For the cell size, the value 350 mm was used because this also the diameter of the robot.

The results of this experiment can be found in table 4.1. In this table, the first column shows the size of the angle, the second column shows the corresponding number of directions, the third column shows the time it took the robot to find the hill using this angle in environment *Navigation and Hill Climb 1* and the final column shows this time in the environment *Navigation and Hill Climb 3*.

In the table there does not seem to be a clear pattern emerging from the data. The recorded times do not give the impression that the duration of the search increases or decreases by changing the size of the angles. It should also be noted that search duration seems to vary a lot even if the same values are used for the parameters. To say anything with any confidence, we would need to repeat the experiments many times. Now, due to time constraints, each experiment has only been executed twice.

4.2 Cell Size

The second experiment was on the size of the cells in the search grid. It was expected that sizes near the robot's own size would yield the best results. The reason for this is that, for cells bigger than the robot, the position of obstacles and the marking of where we have already been would be very inaccurate. On the other hand, if the cells are too small the robot would only every mark those cells in the middle of its track. This means a large number of cells will remain unmarked even though they have already been searched.

Cell Size [mm]	Search Duration 1 [s]	Search Duration 3 [s]
100	2888	4123
250	$1.288 * 10^4$	4403
350	655.8*	1453*
500	—	856.9
750	1212	—

Table 4.2: Results of the cell size experiment

The values used for the cell size were 100 mm, 250 mm, 350 mm, 500 mm and 750 mm. The movement speed in all of the experiments was 40 mm/s because, at this speed, the robot did not seem to have any unnecessary collisions with obstacles. The angle size was 20°, as this value seemed to work well in the experiment in Section 4.1.

The results are shown in table 4.2. The first column shows the size of the cells, the second column shows the time it took the robot to find the hill using this angle in environment *Navigation and Hill Climb 1* and the final column shows this time in the environment *Navigation and Hill Climb 3*. In the third row, with the cell size of 350 mm, the results of the experiment from Section 4.1 with an angle size of 20°are shown as a reference point.

At first glance it might seem that, due to the two low values shown for the cell sizes of 500 mm and 750 mm, a higher cell size is better. However, it should be noted that two of the results are missing. This is because the robot did not find the hill in these experiments. It is likely that this behaviour occurs because the robot cannot mark its obstacles with enough precision anymore at these cell sizes. So it would seem that a cell size greater than the robot's own size is not beneficial for the search duration. Of course, to be certain we would need to repeat the experiments several times, as these experiments have only been executed twice.

4.3 Drive Speed

The final experiment done concerns itself with the drive speed parameter. We looked at several different values for this parameter. These values were: 20 mm/s, 40 mm/s, 60 mm/s, 80 mm/s and 100 mm/s. It was expected that higher speeds will result in a better search duration. However, at a certain point it might occur that the controller is not called frequently enough to react to sensor input, thus decreasing the algorithm's effectiveness. For the other parameters that we experimented on, we took the values that yielded the best results in previous experiments. For the angle size that is 20°. For the cell size that is 350 mm.

The results of this experiment can be found in table 4.3. The first column shows the drive speed while, the second column shows the time it took the robot to find the hill using this angle in environment *Navigation and Hill Climb 1* and the final column shows this time in the environment *Navigation and Hill Climb 3*.

Drive Speed [cm/s]	Search Duration 1 [s]	Search Duration 3 [s]
20	1021	7858*
40	655.8*	1453*
60	1307	1733
80	1893	147.3
100	—	—

Table 4.3: Results of the drive speed experiment

The values in this figure seem to vary a lot without an obvious trend. Therefore we cannot say for certain whether the drive speed affects the search duration. To solve this, these experiments would have to be repeated many times. Due to time constraints, these experiments have been done only twice each. One important thing to note though, is that the search algorithm failed both of the times at a drive speed of 100 mm/s. This is due to the fact that, at this speed, the acceleration when colliding with an obstacle will be so big that the control algorithm believes it has found the hill. Therefore, it would be best not to either use lower drive speeds, or change the method of detecting the hill to something that does not detect it incorrectly.

4.4 Other Findings

While observing the execution of the experiments, a number of times the robot seemed to exhibit unexpected behaviour. As a result, it is possible that the robot took longer to find the hill than needed. These unexpected behaviours have been analysed and are explained in this section. All the sensor variables mentioned in this section are explained in Section 2.1.

First of all, a number of the sensor values seem to be unreliable. We do not know whether this is due to bugs in the simulator or actually a real effect of which we do not know the cause. An example of these unreliable sensor value is the values of the robot's bumpers. While, most of the time, the bumpers seem to work correctly, there have been a number of occasions on which the control algorithm received a `true` value on one of its bumpers while there was no obstacle in front of it. It should be noted that in all of these cases the robot did have an obstacle right behind it. Another sensor that seems to have unreliable values is the infrared sensor to the front of the robot. This sensor is used to detect obstacles in front of the robot. The distance at which this sensor detects obstacles seems to vary significantly. This means that, when marking obstacles in the grid, we can never be certain of the distance at which we need to mark an obstacle when we detect one using this sensor. The final sensors that seem to have unreliable values are the wheeldrop sensors. When the simulation of the robot shows the robot with one wheel in the air, the variable corresponding to that wheel should indicate that that wheel is not touching the ground. However, the control algorithm did not always receive the correct value for these wheeldrop variables. Luckily, this does not occur very often.

Despite these issues, the search algorithm managed to find the hill in the majority of the experiments. How-

ever, it would probably have been faster if these issues had not occurred.

Chapter 5

Conclusions

As shown in Chapter 4, most of the experiments need to be repeated many more times to draw reliable conclusions from them. The reason for this is that the time it takes to search the hill can vary a lot even for experiments with the same values for the parameters. No clear trends emerged from our experiments, but perhaps more experiments will make the results more clear. It would also be beneficial to use a simulator other than CyberSim, as that simulator seems to have its shortcomings.

Additionally, it would be beneficial to experiment on more parameters as well as more combinations of parameter values. In this thesis, the experiments covered only three of the many different parameters used in the implementation of the control algorithm.

Chapter 6

Future Work

This thesis has analysed several parameters of the control algorithm. Of course, there is much more that can be done to analyse and optimise it. Therefore, this chapter shows a few things that can be done to elaborate on this thesis.

6.1 Experiment in More Environments

As was said in the conclusions, the results from the experiments may be somewhat unreliable due to the small number of different environments they were run in. It would be best to verify the results by running the same experiments in many different environments.

6.2 Experiment on More Parameters

While we only experimented on three parameters in this thesis, it would be interesting to see what other parameters can be improved. One example of such a parameter would be the search distance. This is the distance the robot will look ahead while choosing a direction to drive in.

Appendix A

Getting Started

This appendix serves as a guide for the installation of all the software required to run the proposed controller for the iRobot Create. It shows how to reproduce the experiments that were done in this thesis.

A.1 Install LabVIEW

To install LabVIEW, download either .exe file on the website `http://www.ni.com/download/labview-development-syste` `5314/en/`. Once the download is complete, install it.

A.2 Install Visual Studio

First, download Visual Studio. This can be done at `https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx`. It is recommended to download the community version. Once the download is complete, install it.

A.3 Extract CyberSim

To use CyberSim with the proposed controller for the iRobot Create, extract the tarball that is provided with this thesis. This tarbal contains the source code for the controller in the folder `controller/src/C Statechart/libstatechart Visual Studio Solution/`. The rest of the files and folders located in the `controller/` folder are part of CyberSim itself and do not need to be accessed to run the experiments. By opening the file located at `controller/src/C Statechart/libstatechart Visual Studio Solution/libstatechart.vc`

in Visual Studio, the relevant files for project can be accessed. If you have opened this Visual Studio Solution, you can see the Solution Explorer to the left of the interface. Expand ”libstatechart” in this explorer. Then expand the ”C Statechart” folder. In this folder, the file containing the code of the proposed controller is located. This file is called ”irobotNavigationStatechart.c”.

A.4 Running the Experiments

If you have opened the Visual Studio project, you can compile the project by clicking the first option in the ”Build” menu, called ”Build Solution”, or by pressing the ”F7” button. Once this is done, the proposed controller for the iRobot Create can be used simply by clicking the ”Local Windows Debugger” button. After clicking this button, CyberSim will be started. This may take a while. Once it has started, click the ”start” button. This will start the simulation with our controller.

In order to reproduce the experiments done in this thesis, several parameters can be changed. These parameters are located at lines 96, 97 and 98 of the file `irobotNavigationStatechart.c`. How to access this file is explained in Section A.3. These parameters can be changed to the values used in the experiments. The first parameter, located at line 96, is called `DRIVE_SPEED`. This parameter simply determines the speed at which the robot drives in mm/s. This parameter is explained in Section 4.3. The parameter located at line 97, called `N_DIRECTIONS`, represents the number of directions the robot can drive in. This is explained in detail in Section 4.1 The parameter `CELL_SIZE`, which is located at line 98, is the size of the cells of the grid in cm. This is explained in Section 4.2. The values we used for each of the different parameters can be found in the respective sections.

In order to make CyberSim stop when the robot has completed its mission, set a break point at the line of code where the controller switches to the state `DONE`. This can be done by going to line 681 and clicking to the left of the line number. When you do this, a red circle should appear where you clicked.

The final things needed to reproduce the experiments from this thesis are the different environments. Once CyberSim has started, a different environment can be selected in the top left of the user interface. In Figure 2.3, this can be done using the first text area in the area labeled ”Files”. In the provided tarball, the environments are located in the folder `controller/CyberSim/`. For the experiments, two different environment files have been used. These files are called `Environment - Navigation and Hill Climb 1.xml` and `Environment - Navigation and Hill Climb 3.xml`. To reproduce the experiments, enter the path to one of these files in the text area mentioned before.

Once you have changed the parameters to the desired values and selected the desired environment, click on the ”start” button in the button area of the interface shown in Figure 2.3. This will start the simulation. It

should be noted that you are not guaranteed to get the exact same results as the average numbers shown in Chapter 4. This is because the simulator is not deterministic. Some small variations may occur. If these variations make the difference between colliding with a wall and not colliding, it can change the duration of the missions significantly.

Bibliography

[BK91] J. Borenstein and Y. Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, Jun 1991.

[GR01] Yoav Gabriely and Elon Rimon. Spanning-tree based coverage of continuous areas by a mobile robot. *Annals of Mathematics and Artificial Intelligence*, 31(1):77–98, 2001.

[iRoa] Bottom view schematic of the iRobot Create. `https://www.engadget.com/2006/11/29/irobot-create-roomba-hacking-for-the-everyman/`. Accessed: 18/8/2016, Picture has been edited.

[iRob] iRobot Create owner's guide. `http://www.irobot.com/filelibrary/create/Create%20Manual_Final.pdf`. Accessed: 15/08/2016.

[iRoc] Top view schematic of the iRobot Create. `https://www.engadget.com/2006/11/29/irobot-create-roomba-hacking-for-the-everyman/`. Accessed: 18/8/2016, Picture has been edited.

[JJS14] E.A. Lee J.C. Jensen and S.A. Seshia. *An Introductory Lab in Embedded and Cyber-Physical Systems*. `http://leeseshia.org/lab`, 1 edition, 2014. Version 1.70.

[lab] LabVIEW official website. `http://www.ni.com/labview/`. Accessed: 15/08/2016.

[PM08] Jim Pugh and Alcherio Martinoli. *Distributed Adaptation in Multi-robot Search Using Particle Swarm Optimization*, pages 393–402. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[vis] Visual Studio product guide. `http://download.microsoft.com/download/4/E/1/4E1938CA-2146-4F83-8526-6271D298904D/Visual_Studio_2012_Product_Guide_17_02_14.pdf`. Accessed: 15/08/2016.