Universiteit Leiden
Opleiding Informatica

Design and Simulation of an Embedded Controller for the iRobot Create robot by Integrating ROS in Gazebo

Name: Jasper Oostdam

Date: 27/08/2016

1st supervisor: Dr. Ir. T.P. Stefanov

2nd supervisor: Dr. W.A. Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

Leiden University

Niels Bohrweg 1

2333 CA Leiden

The Netherlands

Design and Simulation of an Embedded Controller for the iRobot Create robot by Integrating ROS in Gazebo

Jasper Oostdam

Abstract

In this thesis, we design and implement a controller for the iRobot Create robot by integrating Robot Operating System (ROS) in the robot simulator Gazebo. The thesis consists of two main parts. The first part describes the implementation of the robot and the controller in ROS and Gazebo. By choosing the proper Gazebo sensors and adding ROS plugins to them, we show how we can accurately simulate the robot and control it using ROS. In the second part, we present an algorithm that is used by our controller to make iRobot Create find a hill in an unknown environment, climb the hill, and drive back down. We conclude that the algorithm does what it is supposed to do and we find out which configuration of the algorithm makes the robot find the hill as fast and efficient as possible. Afterwards, we present an addition to the algorithm to try and improve its performance. Although the controller works, there is much room for improvement and further development which we discuss as well.

Acknowledgements

I would like to thank my friends and family for their amazing support, both during this project and before that. I also want to give a special thanks to my second reader, Walter Kosters, for his detailed and helpful feedback that helped me improve this thesis. But most of all, I want to thank my supervisor, Todor Stefanov, for his ideas, help and support that made this entire project possible!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Robots have become more and more important in our lives and come in all shapes and sizes, from small robots in our homes to big robots in factories. Well-known examples of household robots are, e.g., vacuum cleaner or lawn mower robots. Another example is the iRobot Create, which is designed for robot development. Something that all robots have in common, is that they have to be programmed somehow so they can achieve whatever goal they were designed for. Especially for robots that have to perceive and act in their environments, programming these robots can be very complicated. In Artificial Intelligence (AI), such robots — or anything else that can perceive and act in its environment — are called agents. Robotics is a very important topic within Artificial Intelligence, as seen in, e.g., [2, Chapter 25]. There are many issues for these AI agents, such as finding an optimal path through their environment, for which robust control algorithms must be designed. Those algorithms read and process the sensor data from the agent and send control signals to the agent's actuators, such as arms or wheels. Examples of practical problems that can be solved in this way, are making a vacuum cleaner clean a room as fast as possible, or making a lawn mower evenly mow an entire lawn.

1.1 Problem statement

The goal of this thesis is to create an embedded robot controller for the iRobot Create, so that the robot can navigate through an unknown environment to accomplish some mission. The goal for this controller is to make the robot find a hill that is located somewhere in the environment, along with other objects. Once it finds the hill, it must climb the hill to the top, turn around, and drive back down. When it reaches the bottom of the hill and it is back on the ground plane, the mission is finished.

Instead of using an actual version of the robot, we use the robot simulator Gazebo [3] to simulate it in different environments.

For the controller, we use ROS, or Robot Operating System [4]. ROS provides libraries and tools to help software developers create robot applications. The controller is a ROS package written in `C++` code. ROS can be integrated into Gazebo, so that the robot in Gazebo can be controlled using ROS commands. The reason ROS is used to control the robot, instead of using just Gazebo, is that Gazebo only supports very low level commands. For example, we can change the coordinates or pitch of the robot, but not much more. Using ROS, we can make our controller use more high level commands, such as driving or turning in a given direction. This makes controlling the robot much easier, as ROS easily turns our high level commands into more complicated commands for, e.g., the robot's wheels.

1.2 Thesis contributions

The main contributions of this thesis are:

- We modified the existing Gazebo model for the iRobot Create to include the bumper and IMU (Inertial Measurement Unit) sensor, the ROS plugins to be able to read all of the sensors, and the ROS plugin to allow ROS to control the robot's movement. The sensors that are usually on the real robot but which were not used in this project, such as the wheel drop sensors or omnidirectional IR sensor, were not included in the model file.

- We created a control algorithm for the iRobot Create with ROS, which uses the cliff sensors, wall sensor, bumper sensors and IMU sensor to observe its environment and accomplish its mission. The robot does not know anything about the objects inside the environment, including the hill. It does know the size of the environment and its starting position inside the environment. It can also track how far it has travelled and in which direction the robot has turned.

1.3 Thesis overview

In Chapter 2, we will quickly look at other algorithms for similar robots and tasks. Before explaining how our implementation of our control algorithm works, we will go through some background information about the iRobot Create and the software that was used in Chapter 3. Chapter 4 explains how the robot is implemented in Gazebo and what the basis of our ROS controller is. The control algorithm itself is then explained in more detail in Chapter 5 and evaluated in Chapter 6 with some experiments. Finally, Chapter 7 will go over this evaluation and explain what did and what did not work well in our implementation. This chapter also looks at future work that can be done to further improve our implementation.

This thesis was written as bachelor project at LIACS, Leiden University, supervised by Todor Stefanov and Walter Kosters.

Chapter 2

Related Work

An important part of our control algorithm is making the robot find the hill, which in general we can call searching for an object within an unknown environment. With unknown environment, in this case, we mean that we do not know anything about the locations, shapes, etc. of the objects within the environment. In this chapter, we discuss some existing algorithms for robot navigation in an unknown environment.

2.1 Covering algorithm

The first way to find an object in an unknown environment, is by simply making sure the robot visits every part of the environment. This is also known as a coverage algorithm [5]. In this way, at some moment in time, the robot must run into the object.

A possible way of implementing this, is by making the robot zigzag through the environment, first up and down, and if necessary from left to right and vice versa. While doing so, it should avoid objects by moving around them. In this way, the robot will visit the entire environment and from different directions. Sooner or later the robot must run into the hill in a direction such that the robot can drive up the hill. Though this algorithm is reliable, it can also be very slow.

Various algorithms that are based on this approach have already been published. Their goal is to improve the algorithm, covering the environment as fast and efficient as possible. Most of these algorithms use uniform grid maps or occupancy grids to keep track of where objects are located [6]. An algorithm that comes closer to our implementation is [7]. It uses an extended version of this occupancy grid to store more information about the environment and where the robot has already travelled. The biggest downsize of this kind of algorithm is that they may require a lot of storage and computation.

Our algorithm differs from [6] and [7] in the data that is stored in the grid cells. Where [6] uses two variables to indicate the probabilities of the cell being occupied and empty, and [7] uses values to indicate the state of the cells (not visited, empty and cleaned, empty but not cleaned, etc.), our algorithm keeps track of whether a cell contains an object, and otherwise how often the robot has visited that specific cell. The reason for that is that we want our robot to stay away from known objects while also trying to minimize the amount of times the robot visits the same place, so that the robot explores more unknown areas rather than already known areas.

Another similar possibility is to use topological maps instead of occupancy grids [8]. A topological map is a graph in which the nodes represent objects in the environment known as landmarks, and the edges represent the connectivity between them.

2.2 Wall following algorithm

Wall following algorithms are well-known and simple algorithms for robot navigation [9]. In such an algorithm, the robot uses some sensor such as an infrared (IR) sensor or an ultrasonic sensor to follow walls at a certain distance. When the robot finds an obstacle, it will follow the edges of the object like it would with a wall, until it can follow its original course again. When it drives into a wall, it will turn and start following that wall instead.

This kind of algorithms are useful for navigation through environments such as mazes, but is not suitable for the kind of environments and mission we consider. In our case, the robot would only keep following the edges of the environment trying to find a path, while it should traverse the environment searching for the hill.

2.3 Virtual Force Field algorithm

A very different approach to finding a target and avoid objects in an environment is the Virtual Force Field algorithm or VFF [10]. It uses a positive or attractive force to guide the robot towards the target, while objects exert negative or repulsive forces to guide the robot away from obstacles. The algorithm also uses some kind of occupancy grid, called a certainty grid, in which each cell contains a certainty value. That value indicates how likely an object occupies that cell. To determine in which direction to move, a small window moves along with the robot. Each occupied cell exerts a repulsive force to the robot, and these forces are added to create a single repulsive force that tries to push the robot away from obstacles. At the same time, the target exerts an attractive force that tries to pull the robot closer to the target. From those two forces, a resultant force is calculated, which determines the next direction of the robot.

Though this algorithm is quite smooth and accurate, it requires many sensors to scan the area around the robot for obstacles. To actually implement this as an embedded controller on a robot would be much more complicated than the earlier discussed methods. Other downsides are that the robot could get stuck if the resultant force becomes zero (this can occur, e.g., when there is an object directly between the robot and the target), and it can cause unstable behaviour in narrow pathways.

An algorithm based on VFF, called Vector Field Histogram (VFH) [11], was build to overcome some of the downsides of VFF.

Chapter 3

Background

In this chapter, we look into three major components of the project in more detail: the iRobot Create robot, the Gazebo simulator, and ROS. This background information is necessary to better understand some definitions used in this thesis and to have a better overall understanding of the software we used.

3.1 iRobot Create

The iRobot Create [12] is a robot specifically designed for educators, students and developers who want to program or even build their own robot. It is based on the Roomba [13] vacuum cleaner from the same company, iRobot, and was designed for robotics development. The robot can be customized with both hardware and software and it can be programmed and controlled in several ways, such as directly from a laptop or with a microcontroller.

In this thesis, we use a virtual variant of the base version of the robot and focus on programming it. Besides the standard base and wheels, the robot also features some sensors that we use to monitor the robot and its environment. The sensors are also visualized in Figure 3.1:

- 4 cliff sensors to detect cliffs to make sure the robot does not fall off a surface (left, left front, right front and right).
- 1 wall sensor to detect objects ahead of the robot to avoid collisions.
- 2 bumper sensors to detect collisions with other objects (left and right).
- 1 IMU sensor (not shown in Figure 3.1) to detect the robot's orientation (not on the real robot by default but it can be attached).

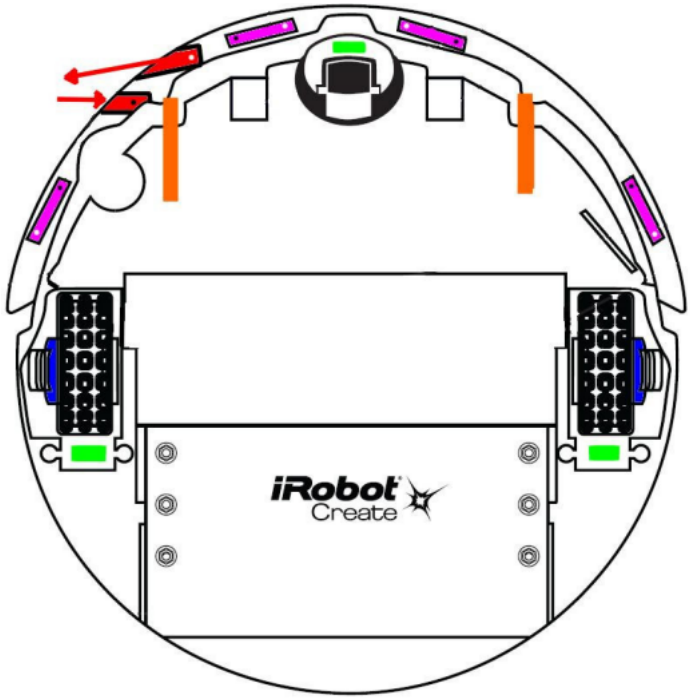

Figure 3.1: Bottom view of the iRobot Create with the wall sensor (red), cliff sensors (purple), bumper sensors (orange), wheel drop sensors (green) and wheel motors (blue). Source: [1].

3.2 Gazebo

Gazebo [3] is a robot simulator that allows to accurately design, simulate and test robots in various environments. It uses the SDF file format [14], which is an XML format used to describe objects and environments in robot simulators and was originally developed specifically for Gazebo. Below we quickly go through some of the XML tags that are important to understand and show what kind of environments we use.

3.2.1 XML tags

To better understand the XML tags below, we use a simplified version of our SDF model file:

```
1  <?xml version="1.0" ?>
2  <sdf version="1.4">
3    <model name="diffdrive">
4      <static>false</static>
5      <!-- Base -->
6      <link name="chassis">
7        <!-- Physics -->
8        <inertial>
9          <pose>0.001453 -0.000453 0.029787 0 0 0</pose>
10         ...
11         <mass>2.234000</mass>
12       </inertial>
```

```
13        <collision name="collision">
14          <pose>0 0 0.047800 0 0 0</pose>
15          <geometry>
16            <cylinder>
17              <radius>0.16495</radius>
18              <length>0.061163</length>
19            </cylinder>
20          </geometry>
21        </collision>
22        <visual name="visual">
23          <pose>0 0 0.047800 0 0 0</pose>
24          <geometry>
25            <mesh>
26              <uri>model://create/meshes/create_body.dae</uri>
27            </mesh>
28          </geometry>
29        </visual>
30        ...
31        <sensor name="wall_sensor" type="ray">
32          ...
33          <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
34            ...
35          </plugin>
36        </sensor>
37        ...
38      </link>
39      ...
40      <!-- IMU plugin -->
41      <plugin name="imu_plugin" filename="libhector_gazebo_ros_imu.so">
42        ...
43        <updateRate>20.0</updateRate>
44      </plugin>
45    </model>
46  </sdf>
```

The `<inertial>` tags (lines 8–12) describe some of the physical properties of the robot, including the mass. This allows Gazebo to accurately simulate the physics and behaviour of the robot in an environment, such as gravity.

The `<collision>` tags (lines 13–21) are used for calculations, for example, to determine if two objects are colliding with each other. These collision elements are also used by sensors to "see" objects in the environment. They do not determine how objects are displayed in the simulator, for that we have the `<visual>` elements (lines 22–29).

Those elements are usually more detailed, and allow us to use visual models such as the iRobot Create model.

The `<sensor>` tags (lines 31–36) are used for the different sensors and have several parameters to adjust the sensors. In this thesis, two of these parameters are important to know. The `<update_rate>` (line 43) determines how often the sensor outputs its data. The `<plugin>` tags (lines 33–35 and 41–44) allow us to add plugins to these sensors. In our case, we need ROS plugins that allow Gazebo and ROS to communicate with each other. Some plugins are used separately from sensors, such as the IMU plugin above (lines 41–44) which simulates an IMU sensor by itself.

3.2.2 Environment

The environments in which the robot operates are built in a similar way as the robot in SDF files. To simplify the environments, we build them as squares by inserting 4 rectangular boxes as boundaries so that the robot will stay within the set enclosure. Within an environment, we will place some objects from the Gazebo model database that the robot has to avoid. One of these objects is the hill, which the robot should drive onto instead of avoiding it when the robot approaches the hill from the front. The objects within the environment are static, meaning the robot cannot move them when it drives into them. One of the two environments used in our experiments is shown in Figure 3.2.

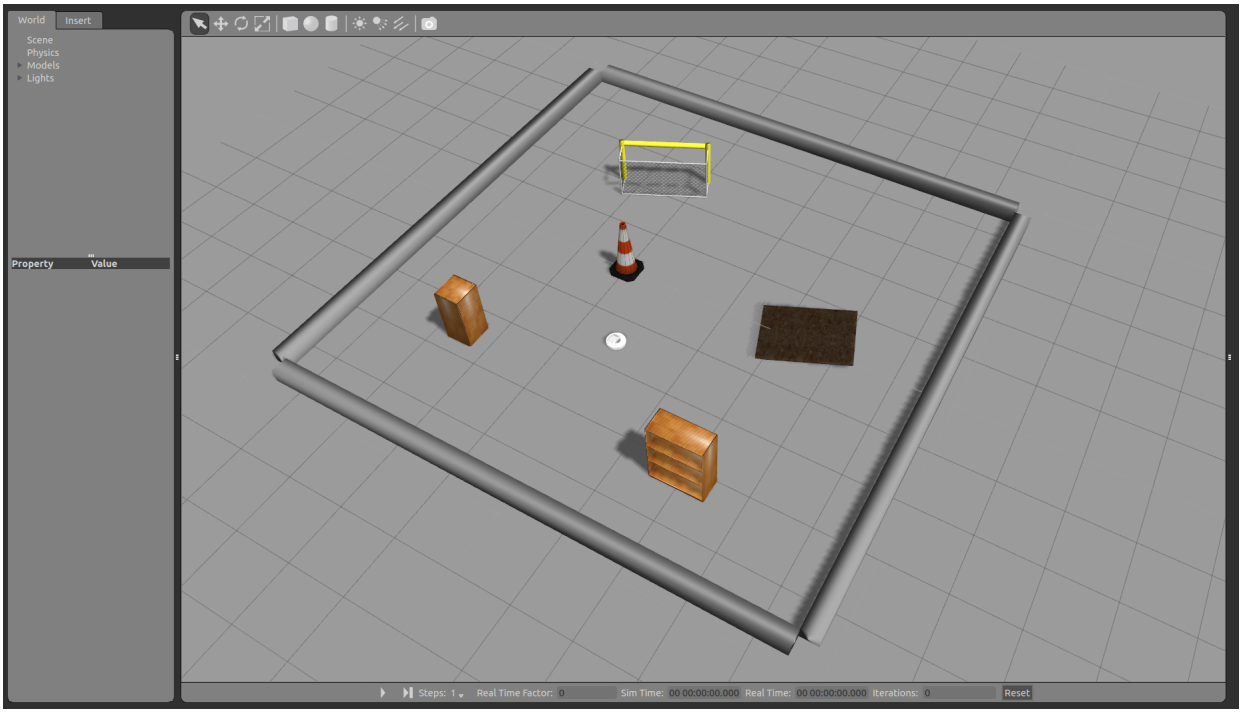

Figure 3.2: Screenshot of an environment in Gazebo, including the iRobot Create model (near the middle, below the construction cone) and the hill (near the right side of the environment).

3.3 ROS

Both Gazebo and ROS [4] use messages to send and receive data. A message is basically a data structure holding the data values from, e.g., sensors or plugins. They often contain a header with a timestamp for identification and one or more variables or other data structures, which can, e.g., be arrays, vectors or even other messages or data structures.

There are many pre-defined message types, but it is also possible to define custom message types. An example of one of these pre-defined messages types is the `gazebo_msgs/ContactState` message:

```
1 string info                                # text info on this contact
2 string collision1_name                     # name of contact collision1
3 string collision2_name                     # name of contact collision2
4 geometry_msgs/Wrench[] wrenches            # list of forces/torques
5 geometry_msgs/Wrench total_wrench          # sum of forces/torques in every DOF
6 geometry_msgs/Vector3[] contact_positions  # list of contact position
7 geometry_msgs/Vector3[] contact_normals    # list of contact normals
8 float64[] depths                           # list of penetration depths
```

The message starts with a string with information as opposed to a header, because this message is usually included in another bigger message (`gazebo_msgs/ContactsState`) that already has a header. The message then contains two more strings and a float array. It also contains two other message types, `geometry_msgs/Wrench` and `geometry_msgs/Vector3`.

Messages are exchanged over named buses called topics, which are meant for unidirectional, streaming communication. They are basically queues with a queue size to determine how many messages can be held and a name to identify it. Each topic only supports one single type of messages. Messages are published onto topics by ”publishers” and received by ”subscribers”.

Each publisher has a topic over which it publishes, a message type for that topic and a queue size for the topic. These parameters are set when creating the publisher. After it is created, it can publish messages of the chosen type onto the chosen topic. When the queue is full, the first message in the queue is discarded. There can be multiple publishers publishing onto the same topic, in which case every subscriber on that topic will receive the messages from all of those publishers.

When a new message arrives on a topic, it is received by all subscribers on that topic. A subscriber subscribes to a single topic, and every ROS cycle, it checks if there is a message available in the topics queue. If there is, the callback function of that subscriber is called. This function has the message as its only parameter, and is used to process the data from this message. For example, when our robot controller receives new sensor data, it must process this data and decide what action to take based on that data, which happens in such a callback function.

In order for Gazebo to be able to communicate with ROS we use special ROS pluins. We need them because Gazebo publishes its data over Gazebo topics, but our robot controller only has access to ROS topics. The plugins work like this:

1. First, we include a ROS plugin in a Gazebo sensor. In this way, when the sensor publishes its data, it automatically sends the message to the plugin.

2. The plugin extracts the data from the Gazebo message, and stores it in a new ROS message. In some cases, the two messages have a different structure, so the plugin must first convert them into the new data types.

3. The plugin then publishes the ROS message onto a ROS topic, so that, e.g., our robot controller can receive and process them.

It also works the other way around, e.g., for the plugin that allows our robot controller to move the robot around. In that case, the plugin actually receives a ROS message and uses it to tell Gazebo how to move the robot.

Chapter 4

Implementation

Before we go into details about our robot controller itself in this chapter, we explain how the controller gets its data from the Gazebo simulator. First, we explain how we implemented the sensors from the iRobot Create in Gazebo. Then we show how the sensor data is sent to the controller over ROS. This also includes the data that is sent back to the robot in a similar way, to make the robot move and turn.

4.1 Gazebo

Our implementation of iRobot Create uses a wall sensor, four cliff sensors, two bumper sensors and an IMU sensor. Each of these sensors is explained in more detail below.

4.1.1 Wall and cliff sensors

For the wall sensor and cliff sensors we use Gazebo's ray sensor. This sensor works by taking a minimum and maximum angle, and dividing the resulting angle in one or multiple rays width equal widths. For each ray, the distance between the origin of the ray and the first object it hits is measured and stored in an array with range data. If that distance is smaller than the set minimum range or bigger than the maximum range it is ignored. In that case, the maximum range is returned as distance in the `ranges[]` array. The ROS plugin that was used for these sensors was the `gazebo_ros_laser` plugin [15].

In order to mimic the IR sensor that is used for the wall and cliff sensors on the real robot, we give the ray sensor a small minimum and maximum angle and use only one ray.

This way, we ensure that the array with range data always contains exactly one entry. This single entry then holds the distance between the sensor and the closest object in front of the sensor.

The wall sensor is located at the front right side of the robot aiming forward. As the name suggests, it is used to detect walls and other objects in front of the robot. It will keep returning its maximum range, which is set to 0.5 m, until a wall or object is detected. In that case it returns the distance between the sensor and that object. Figure 4.1 shows how the returned data is interpreted. Objects that are detected within the red areas would in real life not be detected since they are not in the effective range of the sensor. We are only interested in the objects detected within the green area (the ones within the minimum and maximum range).

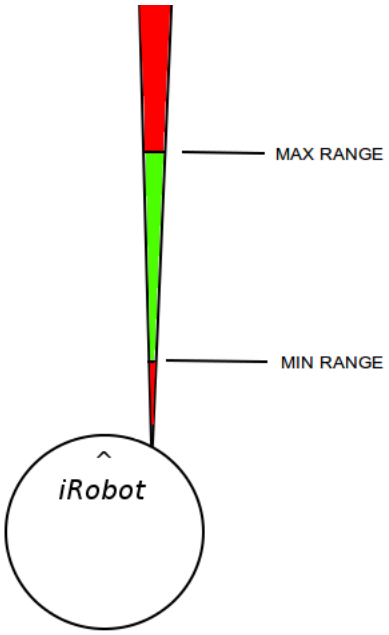

Figure 4.1: Simplified view of the effective range of the wall sensor on the iRobot Create (top view, arrow indicated direction of robot).

The cliff sensors are located at the left, left front, right front and right sides of the robot and aim downward. As opposed to the wall sensors, we do not directly take the minimum and maximum ranges into account. This is because the cliff sensor almost constantly detects the ground within those ranges, and we are only interested in the cases in which it does no longer detect the ground. For this, we declare some minimum threshold that we use instead of the minimum and maximum ranges. When the sensor detects an object below this threshold, we assume it is the surface it is driving on. As soon as the sensor detects an object above this threshold, i.e., it no longer detects the ground, we know that the sensor has detected a cliff. This minimum threshold is set a little higher than the height of the sensor above the ground. This is because when the robot accelerates and pitches up, the cliff sensor is temporarily higher above the ground while there is no actual cliff underneath the robot. Whether the detected range is more than the sensor's maximum range or not does not matter, as long as it is more than the minimum threshold it counts as a cliff.

Figure 4.2 shows how the returned data is interpreted. Objects that are detected within the red and orange areas cannot be detected just like with the wall sensor. We are only interested in the objects detected within the orange and green areas (the ones outside of the minimum range). Objects in the yellow area can be detected, but are ignored since they are not above the minimum threshold.

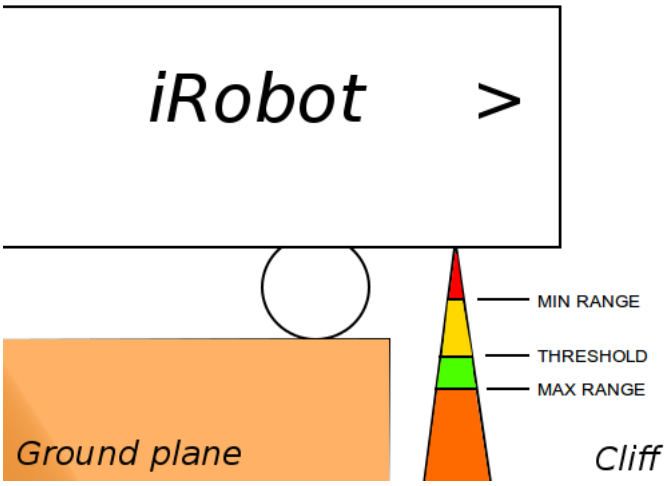

Figure 4.2: Simplified view of the effective range of a cliff sensor on the iRobot Create (right view, arrow indicates direction).

4.1.2 Bumper sensors

Gazebo has a contact sensor that does exactly what we are looking for to implement our bumper sensors: detecting contact between the robot and another object. For this, the contact sensor checks if the collision element from the robot's chassis has contact with any other collision element that is not part of the robot. Note that contact between the wheels and the ground, for example, are not registered because the wheels have different collision elements than the chassis itself, and only contacts with the collision element of the chassis are detected. The sensor returns an array with each contact it measures at the moment the sensor is read. This means that often, more than one contact is returned, even if it looks like there is only one point of contact in the simulator. In this case, the contacts will most likely have very similar values. Each of the returned contact points must be taken into account when simulating the bumper sensors, because it is possible that multiple bumpers are pressed simultaneously.

The downside of this type of sensor is that it can only detect collisions for entire collision elements. This means that, for example, one cannot directly implement multiple contact sensors to detect collisions on different parts of the same collision element.

Our robot has two bumper sensors: one on the front left and one on the front right. To determine if one of those bumpers is pressed, we can use the contact normals of each contact, which is one of the parameters sent with every message for every contact. A contact normal is a normalized vector that points from the point of contact to the center of the robot chassis, which is decomposed into an x- and y-vector. By negating this vector, in other words, by making it point into the opposite direction, we can determine at which absolute angle (in the world, compared to the robot) the contact takes place. For this, within the controller, we calculate $\arctan(-y/-x)$ for each of the contacts. Note that with the minus signs in front of y and x, we negate the vectors. Since such an obtained angle is in radians, we convert it to degrees by dividing it by 180 and multiplying it by π. We also invert the angle by multiplying it by -1 because the `atan2()` `C++` function, which we use to calculate the arctan, gives a positive angle for a contact at the left side of the robot and a negative angle for the right side, while out controller uses negative values for left angles and positive values for right angles. Now, we know where along the robot's edge the contacts take place. The orientation of the robot itself, however, is not yet taken into account. To do that, we simply subtract the robot's angle of orientation, which is always stored as a variable in the controllers class. Using our own defined function `getDegrees()` we ensure that the obtained values are within the range [0;359]. Now, we have the relative contact angles. For each contact, we check whether it causes a bumper to be pressed by checking if the angle lays between certain values:

- left bumper: between 270 and 360 degrees,
- right bumper: between 0 and 90 degrees.

The ROS plugin that was used for these sensors was a modified version of the `gazebo_ros_bumper` plugin [15]. Because the plugin was not properly adding a timestamp to each massage, we made a small modification to the plugin to make it create proper headers. Why this was necessary is explained in Section 4.2.1.

4.1.3 IMU sensor

The IMU sensor, or Inertial Measurement Unit, is a device that measures the robots linear acceleration and angular speed, and also the orientation that can be calculated with these accelerations and speeds. The linear acceleration in this case is also known as the specific force or g-force, and is measured using accelerometers. They measure the non-gravitational accelerations (relative to free-fall: proper acceleration). This means that if the robot is in free-fall, the accelerometers measure no acceleration, as opposed to a situation with coordinate acceleration, in which case an acceleration of around 9.80 m/s due to gravity is measured. To understand better how these accelerometers work, we look at a simple example. Imagine the accelerometer as a small mass that is connected to, in this case, the robot by a spring. When the robot is in free-fall, the same external forces apply to both the robot and the mass, and no acceleration is measured.

But when the robot is standing still on the ground, the ground exerts a g-force on the robot that is the equal and opposite force of the gravity acting on the robot (Newton's third law). Because the mass is still "hanging" from the spring inside the accelerometer instead of resting on some surface, the gravity is the only force acting on the mass and the mass is pulled downwards. The accelerometer then measures this as an upward acceleration from the robot along the z-axis.

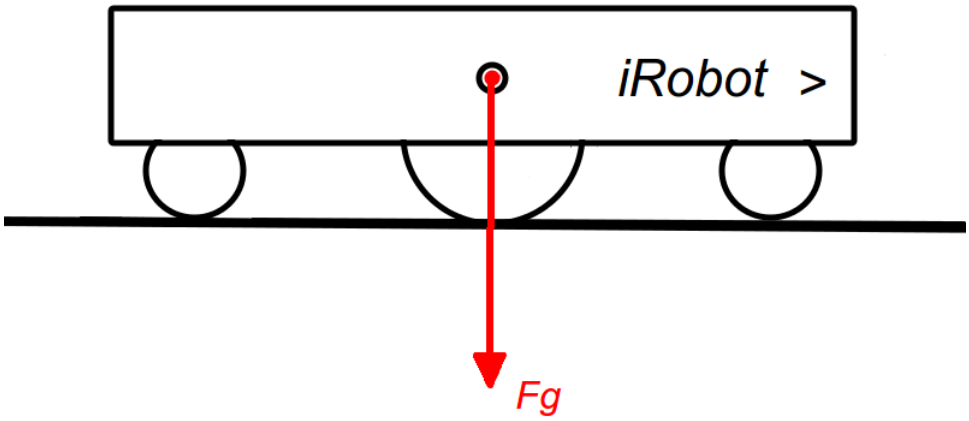

Figure 4.3: Simplified view of the gravitational force (Fg) acting on the iRobot Create on a flat surface.

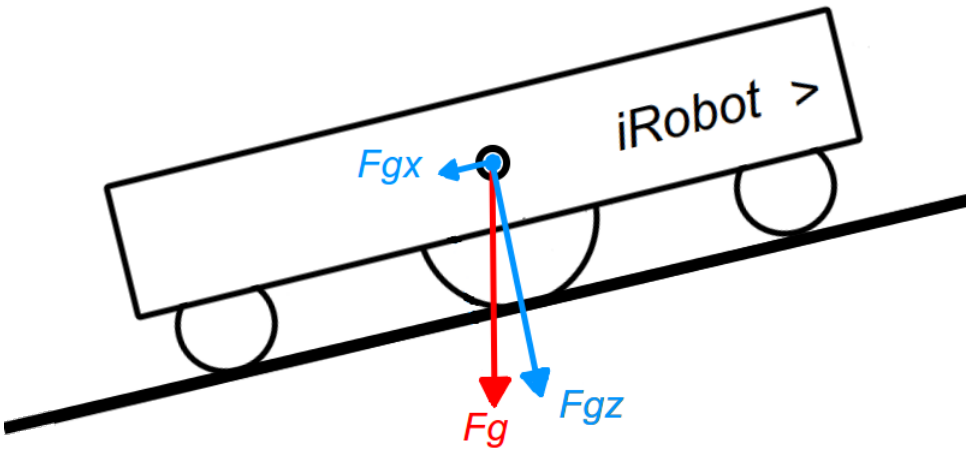

Figure 4.4: Simplified view of the gravitational force (Fg) acting on the iRobot Create on a hill, divided into an x-component (Fgx) and z-component (Fgz).

As long as the robot is on the ground and not moving, the inverse gravity is the only force that is measured by the accelerometers. If the robot is flat on the ground, this force (Fg) is measured only along the z-axis (Figure 4.3). However, when the robot is pitching up along the y-axis or rolling around the x-axis, this force is broken down in not just a z-component, but also an x- and y-component. (Figure 4.4, there is no y-component in this situation because the roll is zero). Therefore, using the accelerations along the x-, y- and z-axis measured by the accelerometers, we can calculate the pitch and roll angles from the robot as described by Equations 28 and 29 in [16]. This will be necessary for the hill detection, as will be explained in Chapter 5.

The ROS plugin that was used for the IMU sensor is the `hector_gazebo_ros_imu` plugin [17]. This is a modified version of the original `gazebo_ros_imu_plugin` [15]. The reason why we have used this one instead of the original one, is that the original one was using coordinate accelerations, while our implementation required the proper accelerations used by the modified variant of the plugin.

4.1.4 Differential drive controller

The differential drive plugin [15] is the ROS plugin that takes the commands from our robot controller, and converts them into commands for the wheel motors. This allows us to simply let our controller tell the robot to move or turn at a certain speed, and the differential drive controller automatically controls the wheels to make the robot move as we want.

The differential drive plugin subscribes to the `cmd_vel` topic and expects `Geometry::Twist` ROS messages. These messages have two main components: a linear speed and an angular speed component. Each of those components is broken down into an x-, y- and z-component.

The differential drive plugin is only interested in two of these sub-components: the linear x speed and the angular z speed. Basically, our controller generates three kinds of messages:

- Move forward or backward by providing a positive or negative speed respectively, in m/s (max. 0.5 m/s), and setting angular z to 0.
- Turn left or right by providing a negative or positive turning speed respectively, in rad/s, and setting linear x to 0.
- Stop moving at all by setting both linear x and angular z to 0.

The controller has several variations of these messages for various purposes, which will be explained in more detail in Section 4.2.2. Note that all we have to do is set the desired speeds, and the plugin does the rest. The plugin also publishes the position and the orientation of the robot in the world. Because the real iRobot Create always knows its orientation, we use this published orientation in our algorithm as we will explain later.

4.2 ROS

The controller for the robot is a ROS package that gets the sensor data as input over ROS topics, uses this input to determine what action to take, and then outputs a ROS message to the differential drive plugin of the robot to make it move. In this section, we describe some aspects of the ROS side of the implementation. The control algorithm itself will be covered in Chapter 5.

The main part of the controller consists of a `C++` class, of which we create an instance right after ROS is initialized. The `ros::spin()` function will then make sure that 20 times per second (which is the looprate we have chosen), the callback function is called, which receives the sensor data and uses the member variables and functions to determine what to do next.

4.2.1 Synchronized callback

As we have explained in Chapter 3, every ROS subscriber has its own callback function. When a new message is published onto the subscribers topic, the callback function is called to process that message. However, our controller has eight subscribers for the different sensors, which means we would need eight separate callback functions.

But how do we make all sensor data available to the controller at once, and how do we ensure that the different pieces of data originate from the same moment in time? For that, ROS offers the ApproximateTime policy [18]. This policy matches incoming messages on a set of different topics using their relative timestamps. This means that during every ROS cycle, a set of messages is created containing exactly one message from each selected topic with approximately the same timestamp. This set of messages is then passed to a single, synchronized callback function as a set of parameters. In this way, we end up with one single function that has access to the readings from all sensors, and we make sure that those readings were from (approximately) the same time.

The ApproximateTime policy uses the timestamps of each message to determine in which order the messages were sent, and thereby determining which messages were sent at approximately the same time. This means that every message should have a unique and non-zero timestamp for the policy to be able to match the message to messages on other topics. The original `gazebo_ros_bumper` plugin [15] had a bug that caused it to give every message from the bumper sensor the same zero timestamp. This made it impossible for the ApproximateTime policy to match any of the messages from this sensor to the messages from the other sensors. Because of that, the entire callback function would never be called as it could never form a complete set of messages. To fix this, we manually added a timestamp to each message by taking the source code of the original plugin and adding the line:

```
this->contact_state_msg_.header.stamp = ros::Time::now();
```

The modified `cpp` file of the plugin is included in the software project and is compiled together with the controller.

The callback function is the most important function of the class. Firstly, it has to process the sensor data. The way the sensor data is interpreted has already been explained in Section 4.1. It checks if any of the bumpers are pressed, if the wall sensor and cliff sensors detect a wall or cliff respectively, and it calculates the roll and pitch of the robot based on the IMU sensor data. Secondly, it uses some extra local and member variables to help deciding which action to take next. Finally, it uses all of this data to actually determine what action to take. This includes, if necessary, changing the robot's states, direction, etc. This is explained in more detail in Chapter 5.

4.2.2 Other functions

Apart from the callback function, the class contains some other functions, we have implemented, that are important for the control algorithm. We will briefly discuss them here:

- `bool isAccelerating(double current_acceleration)` determines if the robot is accelerating. As explained earlier, the accelerometers in the robot measure the robots proper acceleration. This means that the measured acceleration is not necessarily zero if the robot is not accelerating, unlike with coordinate acceleration. Instead, if the robot is not accelerating, the acceleration in the forward (x) direction is constant. Using an array that remembers a number of previous acceleration readings, we can check if the robot is accelerating or decelerating. For this, the current acceleration is added to the array, the average acceleration over this array is calculated and then the current acceleration is compared against the average acceleration. This function is used because acceleration or deceleration can influence the pitch of the robot, and in some cases it is necessary to determine if a change in pitch is caused by this, or because the robot is driving up the hill.

- `bool isPitchingUp(int current_pitch)` is used in addition to the previous function. It determines if the robot is pitching up or not. Again, this is necessary to determine if a robot is really driving up a hill or that the increase in pitch is because of acceleration.

- `void nextDirection()` is the function used by the control algorithm to determine in which direction to continue when the robot detects an object. It is explained in more detail in Chapter 5.

- `void drive()` is used to determine which drive command must be send to the robot. We have implemented the following commands, as ROS messages:

    - drive forward with the current set speed

- drive backward with the current set speed
- turn left with a pre-set turning speed
- turn right with a pre-set turning speed
- turn left with a pre-set turning speed (slower than the regular turn message, used on the hill for more accuracy)
- turn right with a pre-set turning speed (slower than the regular turn message, used on the hill for more accuracy)
- stop moving

Which command is sent depends on the current state of the robot.

Chapter 5

iRobot Create Control Algorithm

Our control algorithm is somewhat based on the vacuum cleaner algorithm of Ulrich [7], which uses a bitmap to remember which areas have been cleaned already or not, and where several kinds of objects are located. In this chapter, we show what kind of grid map we use, and how our control algorithm uses the grid to determine where to move. We will also explain the different possible states for the robot and how they are used to avoid objects, find and climb the hill, etc.

5.1 Grid

In this section, we explain how the grid works that is used by the control algorithm. Since we have defined the environment to be a square, we can easily divide it into a grid of cells. Figure 5.1 shows how an environment is divided into cells of, in this case, 0.5 m by 0.5 m. If there is no object in a cell, the cell's value corresponds to the number of times the robot has driven over that cell. Otherwise, the cell is given the value -1 to indicate an object, even if the object does not occupy the entire cell. This negative value is never increased or overwritten once it has been set. Note that this is not necessary in this implementation, because objects are static and cannot move into or out of a cell.

Usually with occupancy maps, the robot determines its location with odometry data. This means that it uses information such as the wheel speeds and angular velocity to calculate how far and in which directions the robot has travelled. It would also have to know the size of the environment and the starting position to use these calculations to determine where in the environment the robot is located. Another option is to use known landmarks to determine the robot's location, such as the algorithm in [19]. With multiple strategies for this already described, and to simplify our algorithm, we use the actual coordinates of the robot within Gazebo.

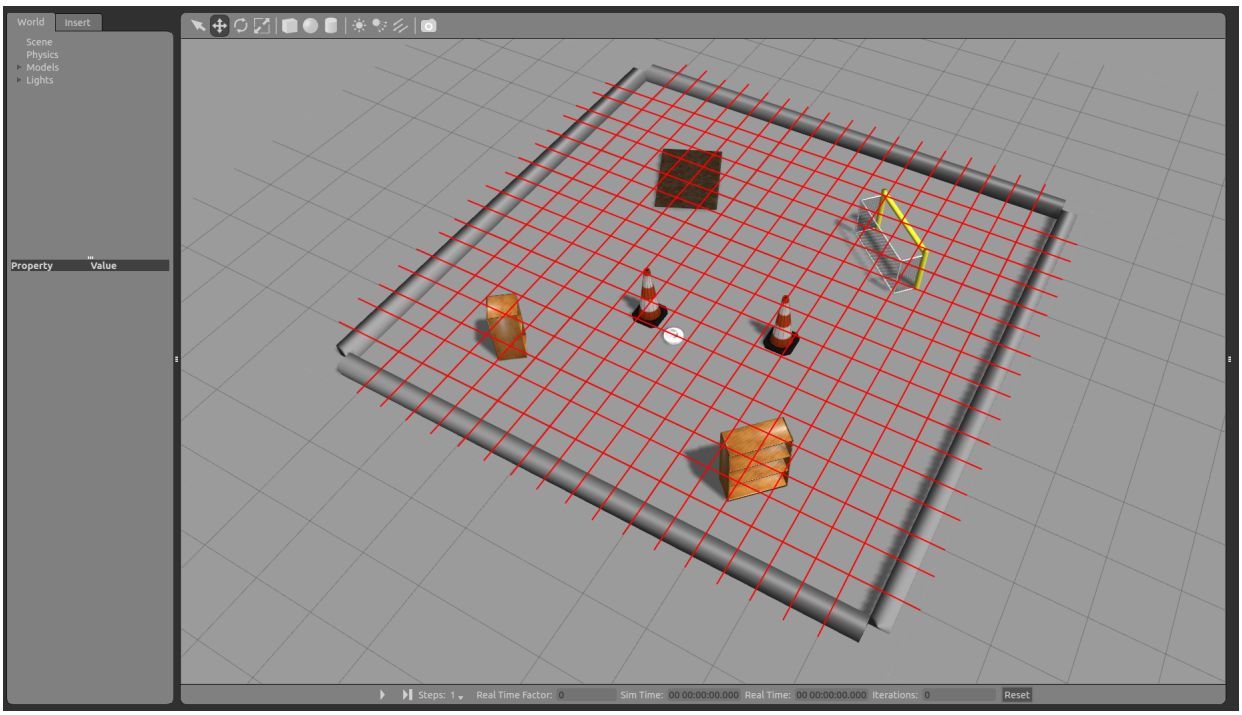

Figure 5.1: Environment in Gazebo of 9.0 m by 9.0 m divided into cells of 0.5 m by 0.5 m.

These are published by the differential drive plugin along with other odometry data, and can be directly used to calculate where in the grid the robot is located.

5.2 Determining the next action

To keep track of what the robot is doing and to determine what to do next, the robot has two different types of states. The first and most important type of state is the overall state of the robot, and it indicates where in the process of finding the hill the robot is at a given moment. Examples of possible overall states are driving forward, turning left to avoid an object, ascending on the hill etc. All possible overall states are listed in Table 5.1 and visualized in the Finite State Machine (FSM) in Figure 5.2.

The second type of state is called the avoid state and consists of two parts:

- The direction in which the robot is avoiding an object. For example, when the robot detects an object on the right, it has to move around it on the left. This information is used by, e.g., the `drive()` function to determine in which direction the robot should turn.
- The location of the robot: on the ground, ascending on the hill, descending on the hill, or at the bottom of the hill (on the ground, but the hill was detected). This information is used when changing states and makes it possible to reduce the amount of overall states.

For example, avoiding an object while ascending or descending on the hill follows the same steps, but after the object avoidance is done, the state should switch back to either `HILL_ASCEND` or `HILL_DESCEND` depending on whether the robot is ascending or descending the hill. Instead of using duplicate states for these steps to distinguish between ascending and descending, we combine the states and use the avoid state to keep track of whether the robot was ascending or descending, so we can switch back to the proper state.

The avoid state, the overall state, stored variables and sensor readings are used together to determine what action to take next. From now on, we refer to overall states as just ”states”. During every ROS cycle, the controller generally performs the following steps, depending on the current state. First of all the controller determines the robot's location in the grid and updates the grid accordingly. This means that if the robot was not already on the current cell in the previous cycle, the visit count of that cell is increased by 1, or if an object is detected the cell is given the value -1. It also reads all the sensors and calculates information such as the pitch and roll. After that, it determines the next state as shown in the FSM in Figure 5.2 (the FSM starts in state ”fwd” and ends in state ”finish”).

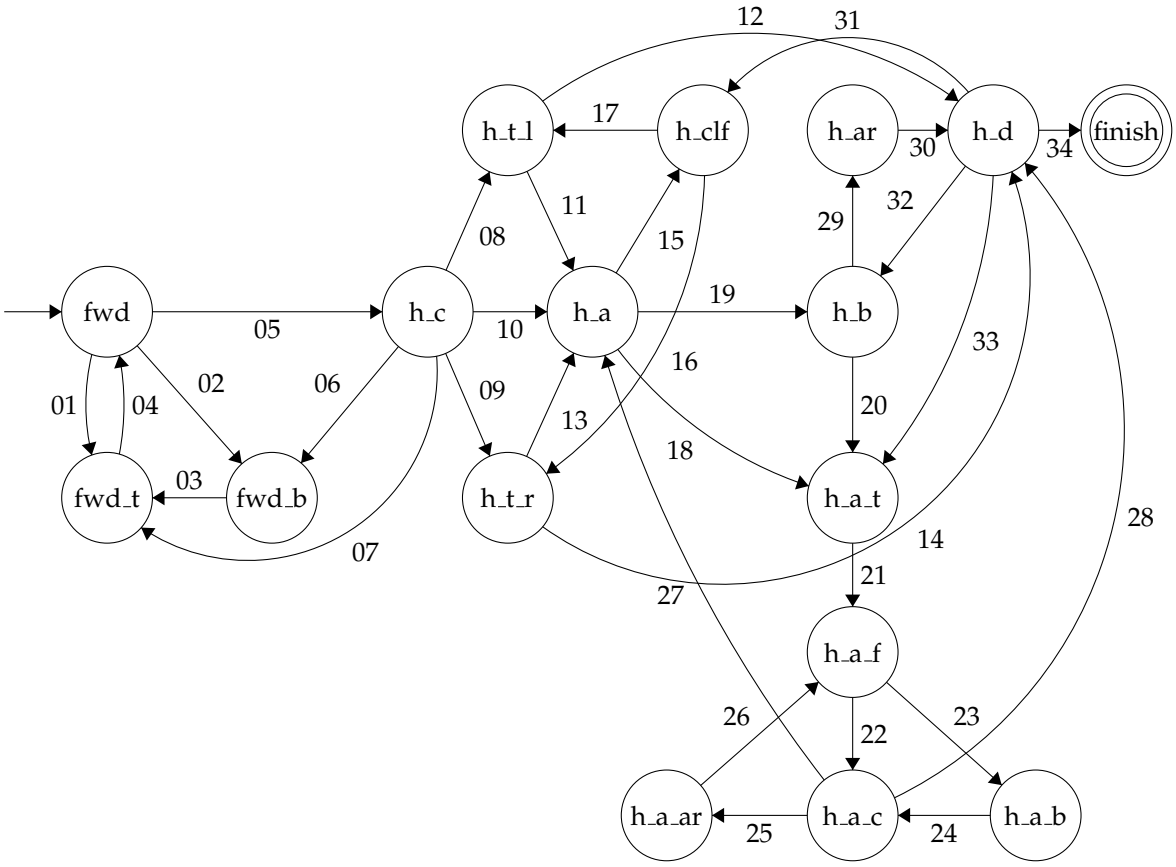

Figure 5.2: The FSM containing all robot states except the stop states.

The FSM represents the control algorithm in the following way:

- Each node represents one of the states the robot can be in.
- Each transition represents a sequence of actions that is taken during every ROS cycle:
    - The controller checks if certain conditions, on which the state has to change, have been met already. For example, when the robot is turning, it must stop turning if the desired angle has been reached. If this condition is not met yet, it simply does not change the states or variables and continues in the current state.
    - If that is the case, the controller changes the current state of the robot. In most cases, this will be a "stop" state. Stop states create small windows between two different states in which the robot can decelerate.
    - The controller also determines what should be the next state. During the next cycle, when the robot is in a "stop" state, the state is switched to this new state.
    - When the robot has detected an object, it should update the avoid state accordingly, depending on where the robot is and where the object was located. For example, when the robot is on the ground and detects an object on the left, the avoid state is given the value "`GROUND_RIGHT`".

      This indicates that the robot is avoiding an object on the ground by turning to the right, which is important information for, e.g., the `drive()` function (to determine in which direction to turn) or when choosing the next state (to, e.g., distinguish between avoiding while ascending or descending on the hill, which both use the same overall states). The avoid state should also be restored to a non-avoiding state when the object avoidance is complete.
    - For some states, the robot should update a variable. For example, when the robot must start turning, the desired next angle should be set in the `next_angle` variable. And when the robot has moved back far enough, the variable `dist_back`, which keeps track of how far the robot has moved back, should be reset to 0.

  The transitions and their conditions are also listed in Table 5.2.
- The transitions from each state to itself — the situations in which the robot does not change its state — are not drawn to simplify the FSM.
- The "stop" states are also excluded from the FSM, as they are already included in the transitions.

The FSM can be divided into several compoments:

- The robot starts on the ground, moving forward (`fwd`). When it detects a wall, it will turn to the left (`fwd_t`). When it detects a cliff or hits an object with its bumper, it drives away backward first (`fwd_b`), and then turns depending on the location of the object.
- When the pitch becomes bigger than 0, the robot knows it is driving up the hill (`h_c`). If it approaches the hill directly from the front (the roll is 0), it continues to climb the hill (`h_a`). Otherwise, it first turns left (`h_t_l`) or right (`h_t_r`) until the roll is 0, which means that the robot is facing toward the hilltop.
- The robot continues to climb until it detects the top of the hill with its left- or right-front cliff sensor, the left or right edge of the hill with its left- or right cliff sensor, or an object with its wall or bumper sensors.
    - In the first case, it drives backward (`h_b`), turns around (`h_ar`) and starts driving down the hill (`h_d`). When it reaches the bottom of the hill (the pitch becomes 0), the mission is finished and the controller shuts down. Note that the second and third case below for avoiding cliffs or objects can also apply when the robot is driving down the hill.
    - In the second case, it drives backward (`h_c`) and then restores its orientation so it faces toward the top of the hill again.
    - In the third case it starts the object avoidance on the hill. If the object was detected with the bumper sensor, it first drives backward (`h_b`).
- The object avoidance on the hill is slightly more complicated then on the ground. It first turns as usual (`h_a_t`) and then drives toward the edge of the hill until it detects a cliff or an object (`h_a_f`). If the object was detected with a bumper or cliff sensor, the robot first backs away (`h_a_b`). The robot then turns 90 degrees so that it is facing toward the top of the hill and checks if there is an object or cliff (`h_a_c`). If there is, it will turn 90 more degrees (`h_a_ar`) and drive toward the other edge of the hill. Otherwise, it will continue to climb up or down the hill.

After this, the `drive()` function is called. This function, based on the current state and avoid state, sends the proper commands to the robot to make it move or turn as we have explained in Section 4.2.2. Which command is sent for each state is shown in Table 5.1.

State:	Full name:	Command:
fwd	FORWARD	Drive forward
fwd_b	FORWARD_BACK	Drive backward
fwd_t	FORWARD_TURN	Turn left or right*
h_c	HILL_CLIMB	Drive forward
h_a	HILL_ASCEND	Drive forward
h_t_l	HILL_TURN_LEFT	Turn left
h_t_r	HILL_TURN_RIGHT	Turn right
h_clf	HILL_CLIFF	Drive backward*
h_b	HILL_BACK	Drive backward
h_a_t	HILL_AVOID_TURN	Turn left or right*
h_a_f	HILL_AVOID_FWD	Drive forward
h_a_b	HILL_AVOID_BWD	Drive backward
h_a_c	HILL_AVOID_CHECK	Turn left or right*
h_a_ar	HILL_AVOID_ARND	Turn left or right*
h_ar	HILL_AROUND	Turn left
h_d	HILL_DESCEND	Drive forward
finish	FINISH	Stop
**	STOP_FWD	Drive backward
**	STOP_BWD	Drive forward
**	STOP_TURN	Stop

Table 5.1: All possible states and the drive commands that are sent to the robot when the robot is in that state.
* = based on the avoid state
** = not shown in the FSM (as explained earlier in this section)

Transition:	Condition:	Function calls:
01	wall detected	`nextDirection()`
02	bumper or cliff detected	`updateNextAngle(**)`
03	travelled backward far enough	
04	turn complete and robot on ground	
05	pitch > 0	
06	bumper or cliff detected	
07	wall detected	`updateNextAngle(`-90 `degrees)`
08	roll > 0	
09	roll < 0	
10	when no other transition is chosen	
11	roll $= 0$ while ascending	
12	roll $= 0$ while descending	
13	roll $= 0$ while ascending	
14	roll $= 0$ while descending	
15	left or right cliff detected	
16	travelled backward, cliff on left side	
17	travelled backward, cliff on right side	
18	wall detected	`updateNextAngle(`-90 `degrees)`
19	bumper or front cliff detected	
20	travelled backward, left/right edge of hill detected	`updateNextAngle(*)`
21	turn complete	
22	wall detected	`updateNextAngle(*)`
23	bumper or cliff detected	
24	travelled backward	`updateNextAngle(*)`
25	bumper, cliff or wall detected	`updateNextAngle(*)`
26	turn complete	
27	no obstacle detected while ascending	
28	no obstacle detected while descending	
29	travelled backward, top edge of hill detected	`updateNextAngle(+180 degrees)`
30	turn complete	
31	cliff	
32	bumper detected	
33	wall detected	`updateNextAngle(`-90 `degrees)`
34	pitch $= 0$	

Table 5.2: The transitions from the FSM in Figure 5.2,the conditions on which they take place and the function calls that take place during the transition.
* = depends on avoid state, can be -90 or +90
** = depends on avoid state, can be -90, +90, -45 or +45 degrees

5.3 Determining the next direction

In this section we will focus on how the algorithm makes the robot find the hill. As we have already discussed, the environment is represented as a grid that keeps track of where the robot has already been and where objects were detected. This grid is used by the controller to determine in which direction to continue when an object is detected. When an object is detected while the robot is on the ground (see transition 01 in Table 5.2), it calls the function `nextDirection()` which uses the algorithm from Algorithm 1 to choose the robot's next direction.

```
1  lookahead ← 3.00;
2  directions [total angles∗2];
3  for i ← 1 to total angles do
       // check left angle
4      directions [i ∗ 2 − 2].dir ← 'l';
5      draw line of length lookahead;
6      for cell along this line do
           // Cells outside grid count as objects
7          if cell is outside environment then directions [i ∗ 2 − 2].objects ++;
8          else
9              if cell value = −1 then directions [i ∗ 2 − 2].objects ++;          // Object in cell
10             else if cell value = 0 then directions [i ∗ 2 − 2].not_visited ++;  // Cell not visited
11             else directions [i ∗ 2 − 2].visit_count += cell .visit_count;       // Cell visited
12             |
13         end
14     end
       // check right angle
15     directions [i ∗ 2 − 1].dir ← 'l';
16     draw line of length lookahead;
17     for cell along this line do
18         if cell is outside environment then directions [i ∗ 2 − 1].objects ++;
19         else
20             if cell value = −1 then directions [i ∗ 2 − 1].objects ++;          // Object in cell
21             else if cell value = 0 then directions [i ∗ 2 − 1].not_visited ++;  // Cell not visited
22             else directions [i ∗ 2 − 1].visit_count += cell .visit_count;       // Cell visited
23             |
24         end
25     end
26 end
27 best_dir ←directions [0];
28 for i ← 1 to size of directions −1 do
29     if directions [i].not_visited > best_dir .not_visited then best_dir ←directions [i];
30     else if directions [i].not_visited = best_dir .not_visited then
31         if directions [i].objects < best_dir .objects then best_dir ←directions [i];
32         else if directions [i].objects = best_dir .objects then
33             if directions [i].visit_count < best_dir .visit_count then best_dir ←directions [i];
34         end
35     end
36 end
37 update the direction and avoid_state of the robot;
```

Algorithm 1: Find next direction

To better understand how the algorithm works, we break down the pseudo code from Algorithm 1:

- The robot draws two virtual lines at a certain angle, one to the left (line 5) of the robot and one to the right (line 16). This angle is defined at the beginning of the controller and in our initial implementation it is set to 30 degrees. The length of the line is called the lookahead and is set to 3.00 m. That means the robot draws the two lines at -30 and +30 degrees with a length of 3 meters.

- The robot then checks which cells the line goes through, and for each of the angles it sums how many of those cells are occupied by an object (lines 7, 9, 18 and 20), how many are not visited yet by the robot (lines 10 and 21), and how many times the remaining cells have been visited together (lines 11 and 22).

  To determine which cells the line crosses, we use an existing algorithm called the Bresenham-based supercover line algorithm [20]. The total amounts are stored in a struct array `directions` for each angle, along with whether the angle is left or right and the angle itself (in degrees relative to the robot, lines 4 and 15).

- The above step is repeated for -60 and +60 degrees, then for -90 and +90 degrees and so on, until -150 and +150 degrees (the for-loop in lines 3–26). Note that 0 degrees is not taken into account because we have just detected an object in that direction. Neither is 180 degrees, because that is where we were coming from. This adds up to a total amount of $(180/angle) - 1$ different angles, and a total of $totalangles * 2$ checks (per angle X, one check for the left side ($-$X), and one for the right side (+X)).

- When all of these angles have been checked, the best one is selected (lines 27–36). This is done by taking the one with the most unvisited cells, the least amount of objects, or the lowest total visit count, in that order of priority. This angle is then determined to be the next direction for the robot.

Figure 5.3 illustrates a possible scenario for the iRobot Create to encounter in an environment. In this case:

- The blue angles are the angles that are checked by our algorithm, the red angles (0 and 180 degrees) are ignored.

- For example, for the angle -30 degrees, four cells are checked: C3, C2, B2 and B1. Two of these cells contain an object and might contain the value -1, but only if the robot has actually already detected and marked those locations.

- The wall sensor, which is located at the right front of the robot, will detect the object (the range is represented by the green line) and set the value of cell D3 to -1. Note that although the object is also located in cells C2, D2 and C3, the wall sensor cannot detect this yet.

- The length of the lines represent the lookahead. Cells further away are ignored by our algorithm.

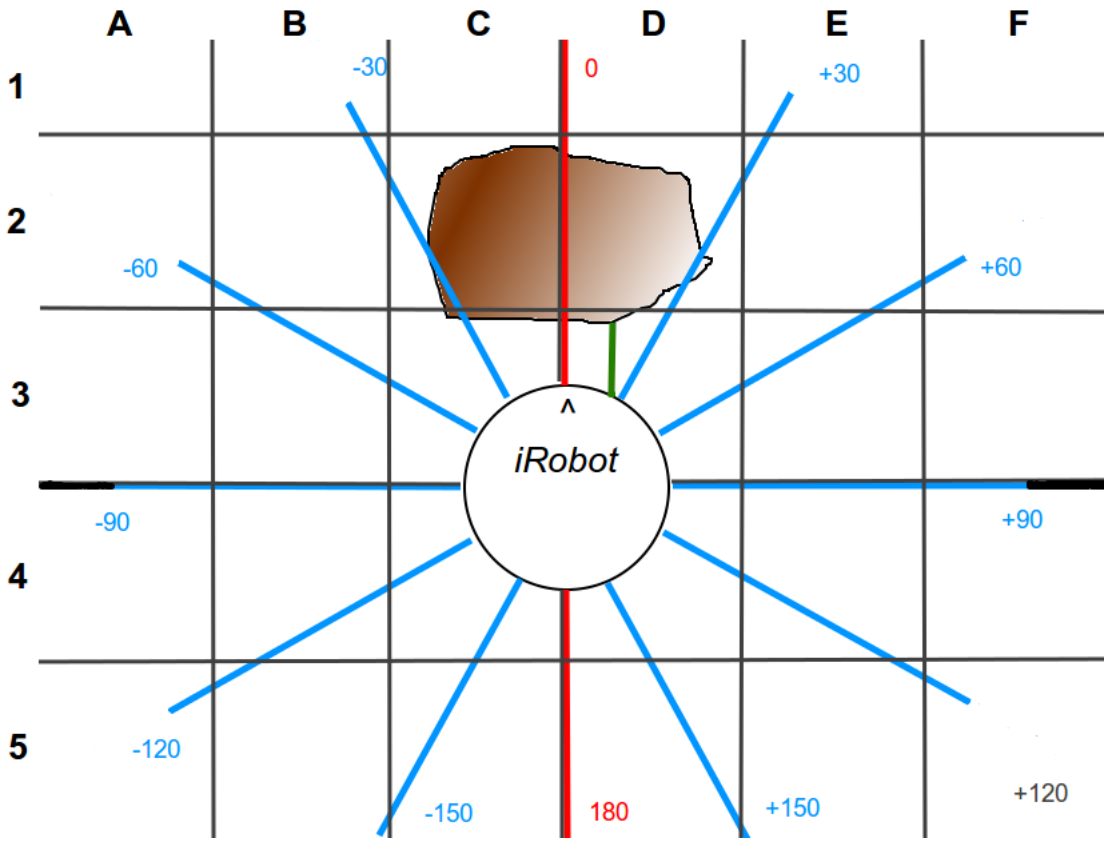

Figure 5.3: A top view illustration of the angles that are checked by the robot when an object is detected.

Because every time when we change the direction we choose the direction that the robot has visited the least, it will eventually have covered the entire grid. Since, it drives both up, down, left and right (relatively), at some moment it will drive onto the hill at an angle at which it can actually climb it. In Chapter 6, we evaluate experimentally how effective and efficient our algorithm is in finding the hill.

Chapter 6

Evaluation

Our iRobot Create control algorithm depends on a few parameters, such as the size of the lookahead and the amount of angles to check when determining the next direction. In this chapter, we examine how efficient our algorithm is and how different values of these parameters affect the performance of our algorithm. Then, we present a slightly different version of our algorithm and see whether the changes offer any improvement in efficiency.

6.1 Experiments

In this section, we experimentally examine the effects of the length of the lookahead, the angles to check when changing direction and the size of the individual grid cells on the efficiency of our algorithm. All distances in this chapter are measured in meters (m), all angles in degrees (°), and all (mission) times in minutes:seconds (min:s). The experimental set-up is shown in Figure 6.1, showing Gazebo in the background just after the iRobot Create has finished its mission. The left terminal was used to launch Gazebo and shows the command used to do so, and the right terminal shows the experimental output of our iRobot Create controller.

When we refer to columns of tables in this section, we refer to the columns of Tables 6.1, 6.2, 6.3, 6.4, 6.5 and 6.6. For each experiment, two values will be measured:

- The amount of time it takes the robot to accomplish its mission (in minutes and seconds).
- The average number of times the robot visits a single cell (excluding cells with objects) given by the formula

$$\frac{TotalVisitCount}{TotalAmountOfCells - TotalObjectCount}$$

where the *TotalVisitCount* is the sum of the number of times the robot has visited each individual cell, *TotalAmountOfCells* is the total number of cells in the grid, and the *TotalObjectCount* is the total number of cells in which the robot has detected an object.

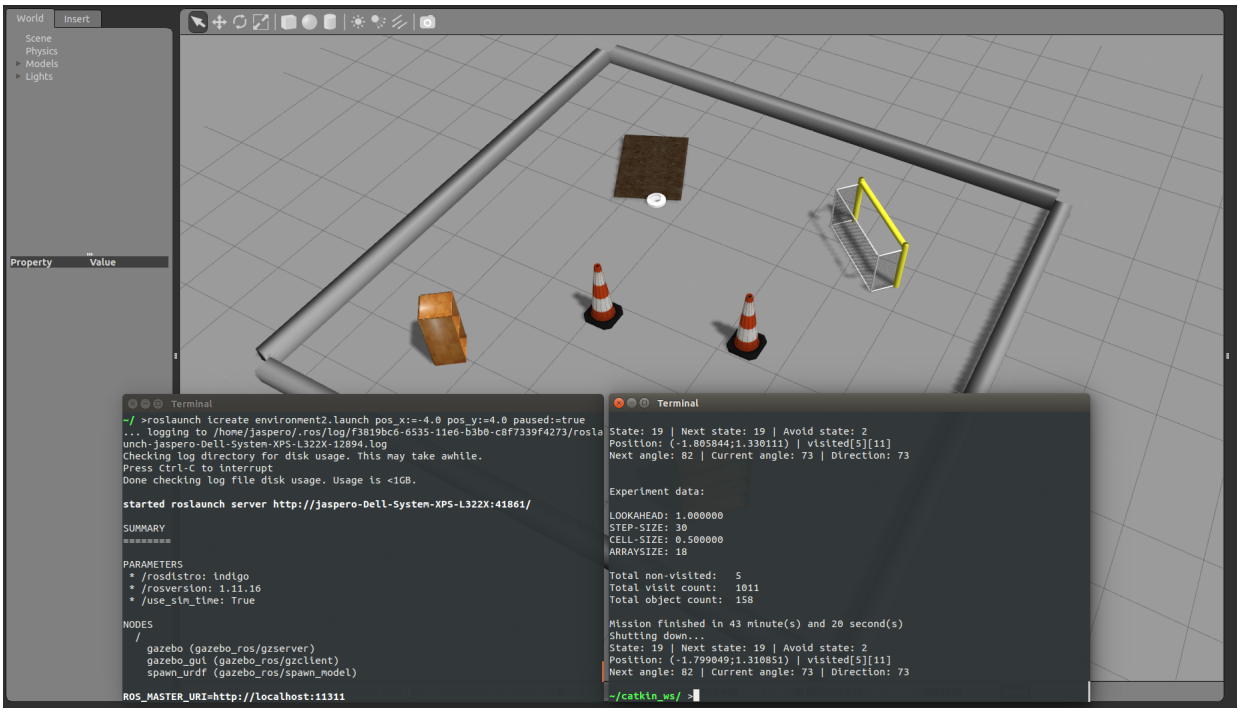

Figure 6.1: Experimental set-up: Gazebo running in the background, one terminal in which Gazebo runs and one terminal in which the controller just finished its execution.

The results in the six tables mentioned above are obtained through the following steps:

1. We select multiple, different values for each of the parameters, which are listed in the first column of each table. For each of these values, we launch Gazebo with our first environment and run the controller. For Section 6.1.1 in combination with the first environment, the starting coordinates of the robot are always $(-0.5\ ,\ -3.1)$ and for the other sections and environments always $(0.0\ ,\ -1.5)$. These coordinates were chosen randomly while ensuring the robot would not start at the position of another object. The starting position had to be adjusted in Section 6.1.2 because the robot would always find the hill after its first turn, making the experiment not very representative, and has been kept the same in all following experiments.

2. Step 1 is repeated four times, giving a total of five experiments. The highest, lowest and average measured values of the mission time and average visit count from these five experiments are displayed in the second and third columns respectively of Tables 6.1, 6.3 and 6.5.

3. Steps 1–2 are then repeated with a different starting position, with the coordinates $(-4.0\ ,\ 4.0)$, and the results are shown in the fourth and fifth columns respectively of Tables 6.1, 6.5 and 6.3.

4. The averages of the results from steps 1–3 are then calculated and stored in the fifth and sixth columns of Tables 6.1, 6.3 and 6.5.

5. Steps 1–4 are repeated for our second environment, of which the results are shown in Tables 6.2, 6.4 and 6.6.

6.1.1 Length of lookahead

The first parameter we check is the lookahead. This parameter determines how far around the robot we examine the values of the grid cells when changing direction. Initially, it was set to 3 meters. We expect a larger value to make our algorithm perform better in terms of mission time and average visit count because it will also take grid cells into account that are far away from objects and would normally not be used. However, it could also make the robot travel in a direction where it soon runs into an object because there are unvisited cells far behind that object. The results are displayed in Table 6.1 and Table 6.2. For each value, the tables show the lowest measured, highest measured and average mission time and average visit count. The same also applies for all other tables in this chapter.

When looking at the results, a few things immediately stand out. First of all, the lowest and highest measured mission times and average visit counts are in most cases very far apart. For example, with a lookahead of 1.00 m in Table 6.1, the lowest measured mission times are 5:24 and 1:16, while the highest measurements were 33:04 and 16:27. Although this seems strange for a deterministic algorithm, we can find an explanation in the simulator itself. Just like in the real world, the wheels motors and the sensors do not behave perfectly. There is always some noise or deviation causing the robot to behave slightly different each time. Even though these differences might be very small, they can build up over time, and even the smallest deviation can mean the difference between just finding or just missing the hill.

Secondly, what is also interesting, is that a larger lookahead does not necessarily mean that our algorithm performs better. We can see that a lookahead of 6.00m does indeed have a smaller average mission time and average visit count than the lookaheads from 1.00 m through 3.00 m for both environments. For the second environment, as seen in Table 6.2, the lookaheads of 7.50 m and 9.00 m perform even better than the one of 6.00 m. However, for the first environment and a lookahead of 9.00 m, the mission time is over 7 and a half minutes, and the average visit count is almost the same as for a lookahead of 2.00 m. This is most likely because, as we assumed, a larger lookahead takes more cells into account and therefore could increase the algorithm's performance. However, at some point, the increased lookahead might take objects into account that are far away from the robot, ignoring that direction though the hill might still be in between the robot and that object. Or the robot might even steer away from the hill because it thinks it is an object.

This is true especially for a lookahead of 9.00 m, because that is the length of the square area (environment) in which the robot moves around. Our algorithm will therefore often register the edges of the environment, even if the robot is far away.

After all, we can say that the lookahead does have an effect on the efficiency of the algorithm, even though there are still major differences between each experiment. The lookahead should not be too small so the algorithm can take enough cells into account, but not too large either, because it might take cells into account that are so far away from the robot that their values are not very important. For the following experiments we continue with a lookahead of 6.00 m.

6.1.2 Size of angle

The second algorithm parameter we look at, is the (step-size of the) angle at which the robot checks the values of the grid cells in the algorithm. We initially set this value to 30 degrees. We assume that a smaller value increases the accuracy because overall, more grid cells are taken into account. In this way the robot has a greater chance of covering the whole grid and in a shorter amount of time. However, if the angle would be too small, the robot might make a lot of minor direction adjustments instead of less larger adjustments. Since, each adjustment consumes both time for computation and for physically turning, this could actually slow the algorithm down. The results are displayed in Table 6.3 and Table 6.4.

The results show the same big differences in times to finish and average visit count as the results of the lookahead experiments, so the step-size of the angle apparently does not help to improve the consistency (i.e., reduce the differences between the different measurements) of the algorithm.

In other words, the time in which the algorithm accomplishes its mission and the average visit counts per cell vary quite much between the experiments. We can see that the average time to finish does improve with a smaller step-size as we expected in the first environment (7:09 for the 10 degree step-size, 9:07 for the 45 degree step-size). However, the second environment shows the opposite when we look at the averages in the last two columns. The average mission time for that environment is the highest for the 10 degree step-size. The average of the lowest mission times that were measured for this environment however were indeed smaller for the 10 and 15 degree step-sizes (1:24 and 1:44 respectively) than for the 30 and 45 degrees step-sizes (3:07 and 2:14 respectively).

Although the impact of the angle step-size varies between the two starting positions and environments, the results do show that generally a lower step-size decreases the time it takes the robot to finish its mission. For the following experiments we continue with a step-size of 10 degrees.

6.1.3 Size of grid cells

The third and final parameter, we examine, is the size of the individual grid cells. These were initially 0.5 m by 0.5 m. We expect that a smaller cell size, and therefore a higher amount of cells in the grid, improves the performance of the algorithm.

However, the smaller the cells and the more cells in a grid, the higher the storage and computational costs will be. It is also possible that many cells remain unchecked for a long time because these cells would fall between those that are examined by the algorithm.

The results are displayed in Table 6.5 and Table 6.6. Since the cells are squares, the size in the first column of these tables is both the length and the width. The second value in this column of these tables — the value between the parentheses — shows how many cells the grid of 9.0 m by 9.0 m contains with each cell size. Note that because we have already obtained experimental data for the 0.5 m cell size, 6.00 m lookahead and 10 degrees step-size during the experiments in the previous section, we can copy these values into these tables.

When looking at the data from the tables, we can see that the cell size does indeed have an effect on the performance of the algorithm and most of our assumptions were right. As we can see, the smaller cell sizes significantly reduce the average visit count because there are simply many more cells that can be visited. The fact that the smallest experiment contained 3600 cells in the grid, as opposed to 324 in our initial configuration, might be an explanation to why the smallest cells from our experiment do not lead to the best results. Although they seem to slightly outperform the big ones of 0.5 m, the middle size of 0.3 m seems to produce the best results we have seen so far in any of the experiments. The difference between the lowest value and highest value is relatively small, and the highest measured value is less than 9 minutes, where most other experiments in other sections had over 10 minutes.

This shows that the algorithm performs quite consistently for both positions. This cell size seems to be a good balance between having enough cells to accurately change direction and having too many cells to effectively use.

		Position 1		Position 2		Average	
Lookahead (m):		Time (min:s):	Avg. visits:	Time (min:s)	Avg. visits:	Time (min:s):	Avg. visits:
1.00	Low	5:24	0.72	1:16	0.18	3:20	0.45
	High	33:04	4.77	16:27	2.03	24:45	3.40
	Avg	12:49	1.80	7:41	1.02	10:15	1.41
2.00	Low	2:04	0.28	2:24	0.34	2:14	0.32
	High	17:06	2.30	19:23	2.77	18:15	2.54
	Avg	9:41	1.20	6:44	0.95	8:13	1.08
3.00	Low	1:07	0.14	6:23	0.87	3:45	0.51
	High	21:55	3.12	13:13	1.87	17:34	2.50
	Avg	7:26	1.03	10:17	1.42	8:52	1.23
6.00	Low	2:36	0.33	4:09	0.39	3:23	0.36
	High	5:32	0.74	11:04	1.67	8:18	1.21
	Avg	4:00	0.53	7:55	1.13	5:58	0.83
7.50	Low	0:44	0.08	2:39	0.34	1:42	0.21
	High	26:14	2.39	7:45	1.22	17:00	1.81
	Avg	8:49	0.94	4:25	0.65	6:37	0.80
9.00	Low	0:42	0.07	1:45	0.26	1:14	0.17
	High	24:46	3.54	17:26	2.38	21:06	2.96
	Avg	8:09	1.13	7:01	0.97	7:35	1.05

Table 6.1: Mission times and average visit counts for different values for the lookahead on different starting positions (first environment).

		Position 1		Position 2		Average	
Lookahead (m):		Time (min:s):	Avg. visits:	Time (min:s)	Avg. visits:	Time (min:s):	Avg. visits:
1.00	Low	7:40	0.99	4:03	0.37	5:51	0.68
	High	18:47	2.10	43:20	6.09	31:03	4.09
	Avg	11:36	1.49	15:24	2.03	13:30	1.76
2.00	Low	1:13	0.17	1:56	0.17	1:35	0.17
	High	30:02	4.08	34:21	4.68	32:11	4.38
	Avg	9:14	1.21	13:36	1.73	11:25	1.47
3.00	Low	3:56	0.48	3:16	0.38	3:36	0.43
	High	28:34	3.77	11:51	1.47	20:12	2.62
	Avg	14:33	1.85	6:38	0.80	10:35	1.33
6.00	Low	5:58	0.69	2:48	0.42	4:23	0.56
	High	10:54	1.58	19:44	1.46	15:19	1.52
	Avg	7:41	1.05	8:34	0.93	8:08	0.99
7.50	Low	0:45	0.10	3:44	0.19	2:14	0.15
	High	5:49	0.79	10:04	1.46	7:56	1.13
	Avg	3:24	0.45	5:59	0.76	4:41	0.61
9.00	Low	0:58	0.11	2:01	0.25	1:30	0.18
	High	12:07	1.63	9:57	1.24	11:02	1.44
	Avg	4:19	0.56	5:19	0.61	4:49	0.59

Table 6.2: Mission times and average visit counts for different values for the lookahead on different starting positions (second environment).

Step-size angle (deg):		Position 1 Time (min:s):	Position 1 Avg. visits:	Position 2 Time (min:s)	Position 2 Avg. visits:	Average Time (min:s):	Average Avg. visits:
10	Low	1:06	0.13	5:40	0.60	3:23	0.37
	High	10:09	1.55	12:30	1.88	11:20	1.72
	Avg	5:45	0.79	8:32	1.23	7:09	1.01
15	Low	3:46	0.52	1:15	0.20	2:31	0.36
	High	20:55	2.51	19:01	2.56	19:58	2.54
	Avg	8:43	1.14	7:10	0.98	7:57	1.06
30	Low	5:58	0.69	2:48	0.42	4:23	0.56
	High	10:54	1.58	19:44	1.46	15:19	1.52
	Avg	7:41	1.05	8:34	0.93	8:08	0.99
45	Low	1:26	0.15	3:03	0.44	1:24	0.30
	High	24:43	3.95	14:40	2.07	19:42	3.01
	Avg	10:33	1.57	7:40	0.96	9:07	1.27

Table 6.3: Mission times and average visit counts for different values for step-size angle on different starting positions (first environment).

Step-size angle (deg):		Position 1 Time (min:s):	Position 1 Avg. visits:	Position 2 Time (min:s)	Position 2 Avg. visits:	Average Time (min:s):	Average Avg. visits:
10	Low	1:36	0.23	1:11	0.15	1:24	0.19
	High	12:23	1.50	22:27	2.58	17:25	2.04
	Avg	7:46	0.89	11:24	1.27	9:35	1.13
15	Low	1:37	0.21	1:50	0.14	1:44	0.18
	High	9:48	1.25	8:23	0.71	9:05	0.98
	Avg	4:58	0.66	3:35	0.32	4:21	0.49
30	Low	3:59	0.58	2:15	0.20	3:07	0.39
	High	12:19	1.38	10:29	1.37	11:24	1.38
	Avg	8:37	1.07	6:07	0.71	7:22	0.89
45	Low	2:26	0.31	2:01	0.19	2:14	0.25
	High	6:39	0.77	14:18	1.80	10:29	1.28
	Avg	4:35	0.57	6:44	0.82	5:39	0.70

Table 6.4: Mission times and average visit counts for different values for step-size angle on different starting positions (second environment).

Cell size (m):		Position 1		Position 2		Average	
(Cells in grid):		Time (min:s):	Avg. visits:	Time (min:s)	Avg. visits:	Time (min:s):	Avg. visits:
	Low	1:10	0.05	3:08	0.13	2:09	0.09
0.15	High	13:43	0.54	16:41	0.69	15:12	0.62
(3600)	Avg	5:04	0.21	6:51	0.29	5:58	0.25
	Low	1:41	0.13	1:26	0.13	1:34	0.13
0.3	High	7:43	0.61	7:42	0.59	7:43	0.60
(900)	Avg	4:47	0.37	4:43	0.35	4:45	0.36
	Low	1:06	0.13	5:40	0.60	3:23	0.37
0.5	High	10:09	1.55	12:30	1.88	11:20	1.72
(324)	Avg	5:45	0.79	8:32	1.23	7:09	1.01

Table 6.5: Mission times and average visit counts for different sizes of the grid cells on different starting positions (first environment).

Cell size (m):		Position 1		Position 2		Average	
(Cells in grid):		Time (min:s):	Avg. visits:	Time (min:s)	Avg. visits:	Time (min:s):	Avg. visits:
	Low	1:19	0.05	4:39	0.19	2:59	0.12
0.15	High	5:00	0.23	19:14	0.58	12:07	0.41
(3600)	Avg	3:08	0.14	10:21	0.37	6:44	0.26
	Low	1:42	0.34	1:46	0.12	1:44	0.13
0.3	High	8:48	0.60	8:22	0.63	8:35	0.62
(900)	Avg	4:33	0.34	3:53	0.29	4:13	0.32
	Low	1:36	0.23	1:11	0.15	1:24	0.19
0.5	High	12:23	1.50	22:27	2.58	17:25	2.04
(324)	Avg	7:46	0.89	11:24	1.27	9:35	1.13

Table 6.6: Mission times and average visit counts for different sizes of the grid cells on different starting positions (second environment).

6.2 Improvements

In this section, we present a modification to our control algorithm to try to improve the algorithm performance. Our control algorithm only checks the cells on the checked angles, as discussed on Sections 5.3 and 6.1.3. With this method, it is possible that some cells that are within the lookahead distance are ignored, because they do not lay on one of the checked angles. To make our algorithm take more cells into account, we add a new section to the algorithm that uses 4 windows of cells: the top left, top right, bottom left and bottom right windows (Figure 6.2). The modified algorithm is shown in Algorithm 2.

```
1  lookahead ← 6.00;
2  step-size ← 10;
   // Amount of angles to check within one window
3  total_angles ← (90/step-size ) − 1;
   // Amount of indices that fit in the length/width of one window
4  indices_to_check ← WINDOW_SIZE / CELL_SIZE;
5  directions [total_angles ];
6  windows [4];
7  windows [0].angle ← 270 // top left;
8  windows [1].angle ← 0 // top right; windows [2].angle ← 180 // bottom left;
9  windows [3].angle ← 90 // bottom right;
   // Check windows
10 for d ← 0 to 3 do
11     for i ← 0 to indices_to_check do
12         if top left or bottom left window then x − index to check = x − index robot + j;
13         else x − index to check = x − index robot + i;
14         for j ← 0 to indices_to_check do
15             if top left or top right window then x − index to check = x − index robot − j;
16             else x − index to check = x − index robot + i;
               // Cells outside grid count as objects
17             if cell is outside environment then windows [d].objects ++;
18             else
19                 if cell value = −1 then windows [d].objects ++;                    // Object in cell
20                 else if cell value = 0 then windows [d].not_visited ++;           // Cell not visited
21                 else windows [d].visit_count += cell .visit_count;                 // Cell visited
22
23             end
24         end
25     end
26 end
   // Choose best window
27 best_window ←windows [0];
28 for i ← 1 to 3 do
29     if windows [i].not_visited > best_window .not_visited then best_window ←windows [i];
30     else if windows [i].not_visited = best_window .not_visited then
31         if windows [i].objects < best_window .objects then best_window ←windows [i];
32         else if windows [i].objects = best_window .objects then
33             if windows [i].visit_count < best_window .visit_count then best_window ←windows [i];
34         end
35     end
36 end
```

```
   // Check angles within best window
37 for i ← 1 to total_angles do
38 |  directions.angle ← best_window.angle − ((i + 1) ∗ step-size );
39 |  draw line of length lookahead;
40 |  for cell along this line do
   |  |  // Cells outside grid count as objects
41 |  |  if cell is outside environment then directions [i].objects ++;
42 |  |  else
43 |  |  |  if cell value = −1 then directions [i].objects ++;                    // Object in cell
44 |  |  |  else if cell value = 0 then directions [i].not_visited ++;           // Cell not visited
45 |  |  |  else directions [i].visit_count += cell .visit_count;                 // Cell visited
46 |  |  |
47 |  |  end
48 |  end
49 end
   // Choose best angle
50 best_dir ←directions [0];
51 for i ← 1 to size of directions −1 do
52 |  if directions [i].not_visited > best_dir .not_visited then best_dir ←directions [i];
53 |  else if directions [i].not_visited = best_dir .not_visited then
54 |  |  if directions [i].objects < best_dir .objects then best_dir ←directions [i];
55 |  |  else if directions [i].objects = best_dir .objects then
56 |  |  |  if directions [i].visit_count < best_dir .visit_count then best_dir ←directions [i];
57 |  |  end
58 |  end
59 end
60 update the direction and avoid_state of the robot;
```

Algorithm 2: Find next direction ("window method").

The names of the windows refer to their position relative to the robot's position in the environment, their locations relative to the robot do not change when the robot rotates .Instead of checking angles around the robot as discussed in Section 5.3, our algorithm performs the following steps:

1. First, we choose the best of the 4 windows in the same way as we would choose the best angle: by taking the one with the most unvisited cells, the least amount of objects, or the lowest total visit count, in that order of priority.

2. Then, to determine the exact angle within this window, we use the "old" method of drawing virtual lines and checking angles as described in Section 5.3. However, instead of checking all angles, we now only check the angles within the chosen window. We also do not choose angles relative to the robot, but to the environment, with 0 degrees being the initial direction of the robot.

   This is also visible in Figure 6.2, where the step-size of the angles is 30 degrees, and the direction of the robot is 300 degrees. Notice that we still do not check the angles directly in front of and behind the robot, like we discussed in Section 5.3, because in front of the robot would be some object that caused us to change direction in the first place, and we do not want the robot to travel back along the exact same way as it just travelled.

For example, imagine that in the situation in Figure 6.2, the top left (yellow) window has the most unvisited cells. Instead of checking the angles 270 and 330 degrees (+30 and -30 relative to the robot's direction), 240 and 0 etc., we check the angles within the top left window. In this case, those angles are 210, 240 and 270 degrees. For the top right window, those angles would be 330 and 0 degrees (330 is the direction of the robot and is ignored as discussed in step 2 above), for the bottom right window 30, 60 and 90 degrees and for the bottom left window 150 and 180 degrees.

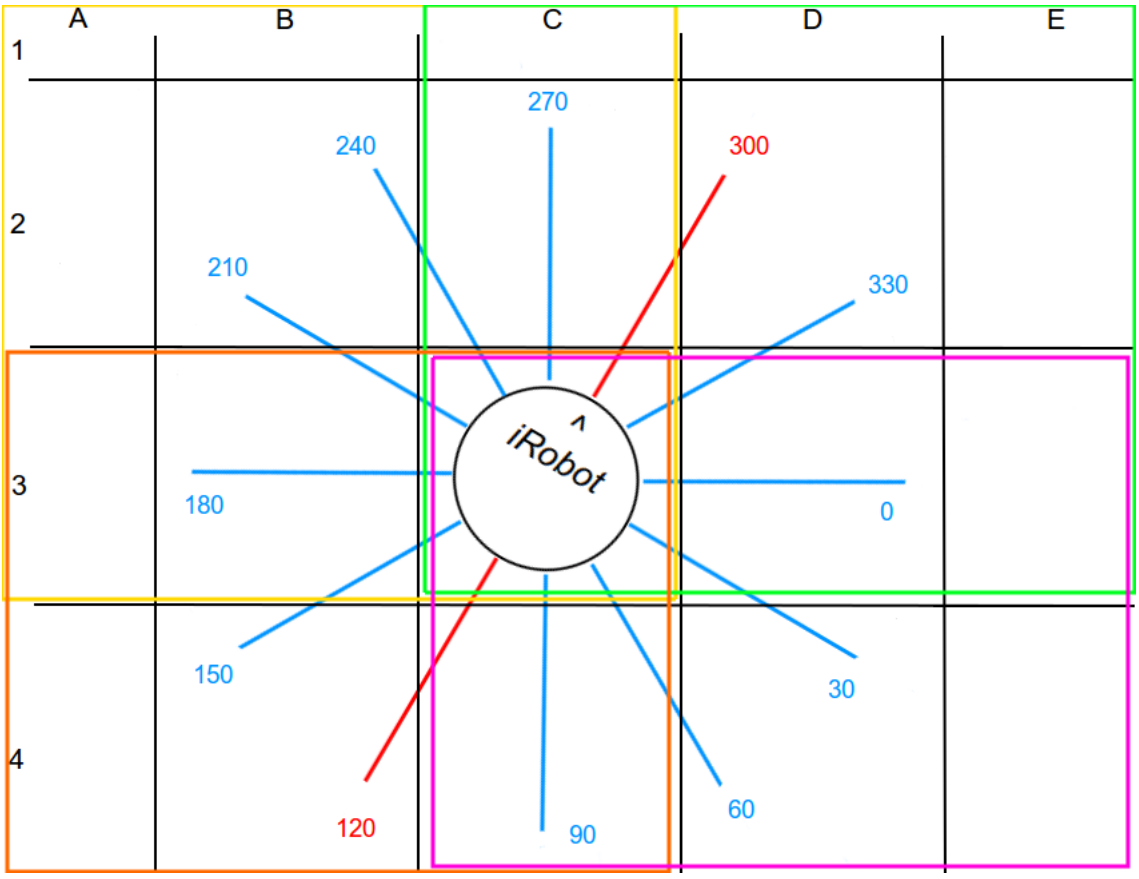

Figure 6.2: Simplified view of the "window method" with the top left (yellow), top right (green), bottom left (orange) and bottom right (purple) windows.

6.2.1 Experiments

In this section, we experimentally evaluate the new addition to our algorithm. The experimental set-up will be the same as explained in Section 6.1. The parameter that we will investigate in this section is the length and width of the windows. Note that because the windows are squares, the length and width of the windows are equal, which is why we refer to them as the ”window size” in Tables 6.7 and 6.8. We expect that a bigger window will make our algorithm perform better, because the algorithm takes more cells into account and therefore has a higher chance of finding unvisited areas, just like for a larger lookahead as seen in Section 6.1.1. However, the experiments in Section 6.1.1 also showed that if the lookahead size becomes too large, the algorithm actually performs worse. We expect a similar pattern in the results of the experiments in this section.

Because more cells are taken into account with this window method than with the old method of just checking angles around the robot, we expect that our algorithm will perform better overall in this section than in the previous sections of this chapter. We use the optimal parameter values that we obtained from the experiments in Sections 6.1.1 (a lookahead of 6.00 m), 6.1.2 (an angle step-size of 30 degrees) and 6.1.3 (a cell size of 0.3 m). The results of the experiment are displayed in Tables 6.7 and 6.8.

As we can see in Tables 6.7 and 6.8, the window size does have a significant effect on the mission time and average visit count as expected. The average mission times for a window size of 6.00 m are 4:10 and 3:20 for the first and second environment respectively, while the 3.00 m window size had average mission times of 11:31 and 7:33 for these environments. However, a larger window size does not necessarily lead to lower mission times or average visit counts. For both environments, the algorithm performs better for a 6.00 m window size than for a 9.00 m window size. We already expected this based on the effect of the size of the lookahead.

The results also show that the algorithm does perform slightly better with the window method than without it with a window size of 6.00 m. Tables 6.5 and 6.6 show average mission times of 4:45 and 4:13 for the parameter values that we used in this section, for the first and second environment respectively. Tables 6.7 and 6.8, in this section, show average mission times of 4:10 and 3:20 for these same environments.

Window size (m):		Position 1 Time (min:s):	Position 1 Avg. visits:	Position 2 Time (min:s)	Position 2 Avg. visits:	Average Time (min:s):	Average Avg. visits:
3.00	Low	4:29	0.42	8:21	0.63	6:25	0.53
	High	17:55	1.27	18:49	1.61	18:22	1.44
	Avg	9:26	0.67	13:36	1.09	11:31	0.88
6.00	Low	1:55	0.16	1:06	0.09	1:31	0.13
	High	8:00	0.65	13:56	0.93	10:58	0.79
	Avg	4:13	0.34	4:06	0.30	4:10	0.32
9.00	Low	1:34	0.09	1:06	0.10	1:20	0.10
	High	22:19	1.59	5:09	0.45	13:44	1.02
	Avg	8:47	0.67	3:26	0.30	6:06	0.49

Table 6.7: Mission times and average visit counts for different values for window size on different starting positions (first environment).

Window size (m):		Position 1 Time (min:s):	Position 1 Avg. visits:	Position 2 Time (min:s)	Position 2 Avg. visits:	Average Time (min:s):	Average Avg. visits:
3.00	Low	3:31	0.23	1:52	0.14	2:41	0.19
	High	14:04	0.97	15:24	1.07	14:44	1.02
	Avg	8:11	0.60	6:56	0.49	7:33	0.55
6.00	Low	1:00	0.07	1:44	0.11	1:22	0.09
	High	6:46	0.59	8:29	0.73	7:38	0.66
	Avg	3:10	0.26	3:30	0.28	3:20	0.27
9.00	Low	1:34	0.13	2:15	0.15	1:54	0.14
	High	18:00	1.29	5:43	0.52	11:51	0.91
	Avg	7:48	0.55	3:29	0.28	5:38	0.42

Table 6.8: Mission times and average visit counts for different values for window size on different starting positions (second environment).

6.3 Software bugs

During these experiments, we found two software bugs that cause the robot to get stuck or drive around in an infinite loop, making it impossible for the robot to finish its mission. Although these experiments are not taken into account in the tables in Chapter 6 — after all, the mission could not be finished so no mission time could be measured — we discuss them in this section, because it is important to know they exist when using our controller. Both bugs seem to be part of either Gazebo or one of the ROS plugins, and not part of our controller.

- The first bug occurs at seemingly random moments and locations, both on the ground or on the hill. One of the cliff sensors suddenly starts to return its max range, as if it detects a cliff, even at times the robot is on the ground plane and in no way can detect any cliff. Since the robot thinks it has detected one of the edges of the hill, it drives back, turns left or right (based on which cliff sensor reacted), and tries to drive forward again. However, because the sensor keeps giving the same maximum reading, the robot gets stuck in an infinite loop of driving backward and turning, never finishing its mission. Because the ROS messages that are published by the ROS plugin for the cliff sensors already contain the wrong readings, we assume that either within Gazebo something goes wrong causing the sensor to give wrong readings, or the ROS plugin somehow makes an error when converting the Gazebo data into an ROS message.

- The second bug occurs when the robot is driving upward on the hill and detects a cliff. For some reason the robot does not always decelerate in the same way. Usually, it gradually slows down as it should, but sometimes it slows down much more rapidly. This can cause the robot to suddenly pitch down and the front wheel to end up off the side of the hill. When the robot then moves backward to avoid the cliff, the front wheels acts as a hook, keeping the robot in place although the robot thinks it is actually moving backward. When the robot then tries to turn away from the cliff, the robot is still too close to the cliff, and the left or right wheel drives off the hill. Sometimes the robot can recover from this with the remaining wheel, but otherwise it will remain stuck on the hill. Since, these different speeds of deceleration occur in situations in which the controller sends the same stop commands to the robot, it is not yet clear why the robot sometimes behaves so differently. It is possible that the Gazebo model or the simulator itself sometimes glitches a little bit causing this behaviour, or that the robot somehow does not get the proper stop commands at the right time.

Chapter 7

Conclusions and Future Work

In this thesis, we presented an embedded robot controller for the iRobot Create robot in Gazebo, using Robot Operating System (ROS). The mission of the controller was to make the iRobot Create find a hill in an unknown, enclosed environment within Gazebo, drive up the hill, turn around, and then drive back down. We first presented our implementation of the robot for the Gazebo simulator, and of our controller as an ROS package. Then, we presented our control algorithm, which is used by the controller to control the robot and make the robot accomplish its mission. Finally, our control algorithm was experimentally evaluated. A slightly improved version of our algorithm was also presented and experimentally evaluated.

As the experiments have confirmed, the robot always finds the hill and finishes its mission in a limited amount of time — that is, if the controller is not affected by one of the bugs mentioned in Section 6.3. The highest measured time for finishing this mission was 43 minutes and 20 seconds, but the differences between the lowest and highest values were significant. As we already discussed in Chapter 6, this is most likely because of small differences between simulations in Gazebo. Sensor noise and slightly varying (wheel) speeds can build up over time, causing the robot to behave differently. For example, a minor change in the robot's orientation can mean that instead of finding the hill, the robot just misses it and keeps searching for a while before finding it again.

From the experiments, we can see that all parameters we varied have a measurable effect on the algorithm's performance. A lookahead of 6 meters, a step-size for checking angles of 10 degrees, and a cell size of 0.3 by 0.3 meters lead by far to the best algorithm results, and gave a relatively consistent algorithm performance. From the 20 experiments, we have conducted with these settings, whenever the hill was found, it was found in between 1:26 and 8:48 minutes. From the window method we presented in Section 6.2, a window size of 6 meters slightly increased the algorithm's performance.

By now, we can say that the combination of Gazebo and ROS is not yet entirely perfect. One of the most obvious reasons for this is that ROS plugins and packages are often created by third parties, and can easily contain small bugs or imperfections. However, both Gazebo and ROS have very active communities, which helps when trying to fix bugs or get other kinds of help. For most plugins, the source code is also available, making it possible to extend or fix them like we did with the bumper plugin. The implementation of existing robots can be a little tricky because the sensors on these robots must be somehow implemented using the available Gazebo sensors and ROS plugins. Of course, it is also possible to develop a custom ROS plugin, although this would require some more experience with ROS. This ability to easily build a custom robot or plugin is also the biggest strength of the Gazebo and ROS combination. They allow to simulate almost any kind of robot, as long as one knows how to implement it.

Looking at the conclusions in this chapter, it is obvious that still there is a lot to be investigated or to be more extensively tested. Although our control algorithm often provides a satisfactory result, the large differences in time to completion deserve some more attention. It might be worth investigation why exactly these differences occur and if the controller can be optimized to reduce them, to make the controller more stable. Even re-running the experiments with more different values for the parameters, more starting positions or environments and even simply more experiments in general, might already lead to an alternative and slightly better configuration for the controller. What might be more interesting, is to explore how the parameters affect each other. For this, the parameter space — all possible combinations of values for all parameters — should be explored. Optimization techniques, such as genetic algorithms, can be used to do this, and find the optimal parameter settings.

It is also worth investigating the two bugs mentioned in Section 6.3. Since these bugs can cause the robot to get stuck and never finish its mission, the algorithm's reliability will be significantly higher when these bugs are fixed.

Something that we have not investigated very thoroughly is the "window" method that we introduced in Section 6.2. The experiments in Section 6.2.1 showed a slightly increased algorithm performance compared to the experiments in Section 6.1.3, which makes it interesting to further look into this method of navigation. An experiment that might also be interesting, is running the controller in an environment without any objects other than the hill and the robot, and see how this affects our algorithm's performance. Another experiment could be to make our controller choose a random new direction when the robot detects an object, and compare the results (such as the time it takes to finish the mission) with the performance of our algorithms from Sections 5.3 and 6.2.

Bibliography

[1] "Getting to know the iRobot Create - iRobot Create Bottom View – CSSE 120Rose Hulman Institute of Technology." https://www.rose-hulman.edu/Users/faculty/young/CS-Classes/archive/csse120-old/csse120-old-terms/200920robotics/Slides/02-Robots.pdf. Accessed: 16-08-2016.

[2] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Prentice Hall, 3 ed., 2010.

[3] "Gazebo homepage." http://www.gazebosim.org. Accessed: 16-08-2016.

[4] "ROS homepage." http://www.ros.org. Accessed: 10-07-2016.

[5] M. Bosse, N. Nourani-Vatani, and J. Roberts, "Coverage algorithms for an under-actuated car-like vehicle in an uncertain environment," in *Proc. 2007 IEEE Int. Conf. on Robotics and Automation*, pp. 698–703, 2007.

[6] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *IEEE Computer Magazine*, vol. 22, no. 6, pp. 46–57, 1989.

[7] I. Ulrich, "Autonomous vacuum cleaner," *Robotics and Autonomous Systems*, vol. 19, no. 3–4, pp. 233–245, 1997.

[8] S. C. Wong and B. A. MacDonald, "A topological coverage algorithm for mobile robots," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 1685–1690, 2003.

[9] T. Yata, L. Kleeman, and S. Yuta, "Wall following using angle information measured by a single ultrasonic transducer," in *Proc. IEEE Int. Conf. on Robotics and Automation, vol. 2*, pp. 1590–1596, 1998.

[10] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for fast mobile robots," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 19, no. 5, pp. 1179–1187, 1989.

[11] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, 1991.

[12] "iRobot website - iRobot Create product page." http://www.irobot.com/About-iRobot/STEM/Create-2.aspx. Accessed: 10-07-2016.

[13] "iRobot website - Roomba product page." http://www.irobot.com/For-the-Home/Vacuuming/Roomba.aspx. Accessed: 16-08-2016.

[14] "SDFormat specification." http://sdformat.org/spec. Accessed: 16-07-2016.

[15] "Gazebo plugins in ROS." http://gazebosim.org/tutorials?tut=ros_gzplugins&cat=connect_ros. Accessed: 16-08-2016.

[16] M. Pedley, "Tilt sensing using a three-axis accelerometer." https://www.nxp.com/files/sensors/doc/app_note/AN3461.pdf, 2013. Accessed: 13-07-2016.

[17] "ROS Wikia – hector_gazebo_plugins." http://wiki.ros.org/hector_gazebo_plugins. Accessed: 16-08-2016.

[18] "ROS wiki - message_filters/ApproximateTime." http://wiki.ros.org/message_filters/ApproximateTime. Accessed: 11-07-2016.

[19] F. Chenavier and J. L. Crowley, "Position estimation for a mobile robot using vision and odometry," in *Proc. IEEE Int. Conf. on Robotics and Automation, vol. 3*, pp. 2588–2593, 1992.

[20] "Bresenham-based supercover line algorithm." http://eugen.dedu.free.fr/projects/bresenham/. Accessed: 15-07-2016.

[21] "Which combination of ROS/Gazebo versions to use." http://gazebosim.org/tutorials/?tut=ros_wrapper_versions. Accessed: 16-07-2016.

[22] "Ubuntu install of ROS Indigo." http://wiki.ros.org/indigo/Installation/Ubuntu. Accessed: 16-07-2016.

[23] "Installing and configuring your ROS environment." http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment. Accessed: 16-07-2016.

Appendix A

Getting Started

In this appendix we go through all the necessary steps to install Gazebo and ROS and use the controller. For some steps we will refer to existing tutorials from both Gazebo and ROS since they already have clear and detailed instructions on, e.g., the installation. The ROS package with the controller comes with this thesis.

A.1 Software versions

The software versions we have used in this project are:

Gazebo version 2.2.6

ROS distribution indigo, version 1.11.16

Ubuntu 14.04 LTS

Note that the robot model and controller are designed to work with these software versions. The project has not been tested yet on other versions, and we cannot guarantee that the robot will behave as expected. The instructions below are therefore for ROS indigo + Gazebo 2.X. Also worth mentioning is that ROS indigo only supports Ubuntu 13.10 and 14.04. If you have another version, either up- or downgrade to 14.04 or try whether the controller works on other software versions [21].

A.2 Installing ROS and Gazebo

To install both ROS indigo and Gazebo 2, we can use the ROS repository which automatically installs Gazebo alongside ROS.

The following commands were obtained directly from the ROS indigo installation page for Ubuntu [22]. Execute them in this order in a terminal to install ROS indigo:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
    main" > /etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0
    xB01FA116
$ sudo apt-get update
$ sudo apt-get install ros-indigo-desktop-full
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/indigo/setup.bash" >> (*$\sim$*)/.bashrc
$ source (*$\sim$*)/.bashrc
$ sudo apt-get install python-rosinstall
```

Now that ROS and Gazebo are installed, run the following command to install the ROS packages that will make ROS and Gazebo actually work together:

```
$ sudo apt-get install ros-indigo-gazebo-ros-pkgs ros-indigo-gazebo-ros-
    control
```

To test if the installation was successful, we run the following commands, which should use ROS to start Gazebo:

```
$ roscore
$ rosrun gazebo_ros gazebo
```

A.3 Set up a workspace

Now that ROS and Gazebo are installed, we have to set up a workspace. The instructions below are from the ROS environment tutorial [23] and will show how to set up a Catkin workspace which is used by ROS:

```
$ mkdir -p ~/catkin_ws/src
```

```
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

With the environment set up, we can now build the workspace:

```
$ cd ~/catkin_ws/
$ catkin_make
```

If these commands were successful, there should now be a `build` and `devel` folder inside the workspace. We now source the generated `setup.bash` file from the `devel` folder with the command

```
$ source devel/setup.bash
```

If everything went well, the `ROS_PACKAGE_PATH` variable should now include the location of the workspace. This can be checked with:

```
$ echo $ROS_PACKAGE_PATH
```

To make sure that this variable always includes the location of workspace, one can add the source command to `.bashrc`. This can be done manually or use the following command:

```
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

The ROS environment has now been set up. The `catkin_make` command can from now on be used to build the environment after any changes have been made to, e.g., one of the ROS packages in the `src` folder. Note that this command should always be executed from the root of the workspace (`~/catkin_ws`).

A.4 Build the ROS package

With our environment all set up, it is time to copy our ROS package into the workspace. To do this, place the folder `icreate`, that comes with this thesis, into the source folder of the workspace (`~/catkin_ws/src`). Then, we build it with the following commands:

```
$ cd ~/catkin_ws/
$ catkin_make
$ source devel/setup.bash
```

A.5 Usage of files

To run the controller, we perform three simple steps:

1. Start the ROS master with the command:

   ```
   $ roscore
   ```

2. Start Gazebo with the environment of choice. The environment is determined by the launch file we launch Gazebo with. The default launch file is `environment.launch`.

   To use this file, execute:

   ```
   $ roslaunch icreate environment.launch
   ```

   We can also change the robots initial position. With the default environment size of 9 m by 9 m, the initial coordinates must be between -4.5 and 4.5. Example:

   ```
   $ roslaunch icreate environment.launch pos_x:=-4.0 pos_y:=4.0
   ```

3. Finally, start the controller by running:

   ```
   $ rosrun icreate controller
   ```

The important files for our implementation are:

- The launch files load an environment and spawn the robot in that environment. By default the robot is spawned at the coordinates (0,0), but these coordinates can be changed when running the launch file.
- The world files contain the different environments in which the robot has been tested. They contain both the borders of the environment and the objects within, including the hill. The robot itself is not spawned with the rest of the world, but separately by the launch file.
- The model file `model-4.sdf` contains the model of the robot itself.
- The file `controller.cpp` is the controller itself.
- The file `gazebo_ros_bumper_v2.cpp` contains the modified version of the original `gazebo_ros_bumper` plugin. It is compiled together with the controller.
- The files `gazebo_ros_bumper.h` and `gazebo_ros_utils.h` are required for compiling the modified plugin.
- The file `controller_v2.cpp` is the modified version of our controller, as presented in Section 6.2. It can be launched in the same way as our original controller, the only difference is the name of the executable:

  ```
  $ rosrun icreate controller_v2
  ```