Universidade de Aveiro Departamento de Engenharia Mecânica
2020

Sidney Daniel Delgado Duarte Lopes

Integration of a Webots Mobile Robot Simulation in ROS

Projeto em Engenharia de Automação

Keywords ROS framework, Webots simulator, mobile robot navigation, maze-like environments

Abstract A mobile robot system is under development to explore innovative machine learning solutions to localization and navigation problems. The system consists of a small-size mobile robot and a ROS-based framework for intelligent control and simulation. The main objectives of this project are twofold: First, to integrate a robot simulator in the ROS-based framework. The selection of the simulator should consider aspects such as the benefits of using open source software, the advantages of cross-platform development and the possibilities of multiple programming languages (*e.g.*, C/C++, Java, Python). Second, to develop the software modules needed to be able to control, simulate and analyse the behaviour of the mobile robot. This report describes the efforts dedicated to the software tools, namely the use of the ROS framework and the Webots simulator, as well as their interconnection. The implementation of a hierarchical reinforcement learning algorithm is discussed, as well as the application for robot navigation is a maze-like environment.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context and background

There is a growing interest in the development of autonomous navigation technologies for application in domestic and urban environments. On the one hand, simulation tools assume an essential role since a well-designed robotics simulator makes it possible to rapidly test algorithms and evaluate performance in many realistic scenarios. Webots, V-REP and Gazebo stand out among the free simulation software tools used todays in robotics. On the other hand, machine learning provides a set of computational useful tools for robot navigation, such as neural networks (NN) and reinforcement learning (RL). RL is a computational approach to learning by trial-and-feedback where an agent tries to maximize the total amount of reward it receives when interacting with the environment to achieve a goal.

In the past few years, several RL algorithms have been proposed for navigation control of autonomous mobile robots [refs]. In particular, hierarchical RL algorithms have drawn the attention of researchers as an effective solution to several challenges in robot RL such as coping with large state-action spaces (*i.e.*, curse of dimensionality problem) and the impractical number of interactions required for learning a policy (*i.e.*, due to sample inefficiency). A promising approach is to investigate how RL can be employed for robotic navigation tasks based on a hierarchical structure that decomposes the problem into basic sub-tasks coordinated by a higher level.

Following the latest progress, a mobile robotics system is being developed to explore innovative machine learning solutions to localization and navigation problems. The system is composed of a small mobile robot based on the Alphabot2 platform and a ROS-based framework for intelligent control and simulation. The mobile robot carries on board an Arduino Uno Plus micro-controller, a belt of 6 infrared sensors (Sharp GP2Y0A41SK0F) distributed around its perimeter, 5 infrared ground sensors (ITR20001/T) to solve path following tasks and 1 ultrasound distance sensor (HC-SR04).

The communication between the host computer and the remote mobile robot is established through a wireless system, such as the sensorial information flows from the remote mobile robot to the host computer and the high-level commands are sent in the opposite direction. In what concerns the software development, the project adopted the ROS middleware since it provides multi-platform support, distributed programming, real-time execution and an increasing number of compatible products and simulation tools.

1.2 Objectives

The main objectives of this work are twofold: First, to integrate a robotic simulator in the ROS-based framework. The selection of a dynamic simulator should consider aspects such as the benefits of using open source software, the advantages of cross-platform development and the possibilities of multiple programming languages (*e.g.*, C/C++, Java, Python). Second, to create the software packages needed to control, simulate and analyse the behaviour of a mobile robot. The robot should be able to perform navigation experiments in maze-like environments by exploring learning approaches.

The second part of the work follows in line with a dissertation (Diogo, 2019) that had investigated the use reinforcement learning for a maze-like navigation tasks using a hierarchical structure. This dissertation gave rise to several ROS software packages, written in C/C++, allowing to carry out preliminary experiments with the real mobile robot. Within the scope of the PEA, the hierarchical structure was completely revised, and new ROS packages were added to help the interpretability of the results.

1.3 Report outline

This project report is organized into five chapters, including the introduction:

- Chapter 2 introduces the key concepts, main features and functionalities associated with the software tools used thorough this work, namely the ROS middleware and the Webots dynamic simulator.
- Chapter 3 provides a detailed step-by-step guide for interfacing ROS with Webots with examples of a collision avoidance controller for a mobile robot in a maze-like environment.
- Chapter 4 describes and evaluates the software packages developed towards the mobile robot navigation in the maze using a reinforcement learning controller.
- Chapter 5 presents the main conclusions of this project.

Chapter 2

ROS Framework and Webots Simulator

This chapter describes the software tools used thorough this work, including the ROS development environment and the robotic simulator. Section 2.1 provides an overview of how ROS works both on the side of operating system features and the set of user contributions to high-level functionalities (*i.e.*, ROS packages). Section 2.2 explains the fundamental concepts of the robotics simulator Webots needed to perform the mobile robot navigation in a maze-like environment. This section focuses on the modelling of the robot and its environment, as well as on programming the robot controller.

2.1 Robot Operating System (ROS)

Developing a robot system requires a collection of software tools on the computer side such as software drivers, reusable software components and simulation tools. The ROS framework allows gathering all these components tools, helping the development of robotic applications. ROS was initially developed by the Stanford AI Laboratory in 2007 and it is currently maintained by the Open Source Robotics Foundation. ROS provides hardware abstraction, low-level device control, inter process communication (message-passing based on publisher/subscriber mechanisms) and package management. Therefore, this middleware allows soft-

ware developers to focus on the specific purpose of their application, making it easier to design, create and simulate a robotic model, as well as to interfacing it into real hardware.

2.1.1 An overview of how ROS works

ROS facilitates the interaction of hardware and software using a modular architecture where all processes are connected in a network. A ROS application consists of several processes, eventually running on different host computers, in a peer-to-peer topology where each of the nodes of the network works both as a client and as a server, without the need for a central server. All nodes can access this network to interact with other nodes, *i.e.*, to transmit data to the network and to read the information other nodes are sending. Figure 2.1 illustrates an example of two ROS nodes communicating through a ROS topic.

The ROS nodes are executable programs that subscribe to topics to receive information, perform computations, and publish data for other nodes to use. The communication between nodes occurs through the passing of messages, *i.e.*, a data structure comprising typed fields. This message types includes standard types such as integer, floating point, boolean, arrays, as well user-defined types. A node sends a message by publishing it to a given topic (name used to identify the content of a message). A node interested in the data needs to subscribe to the appropriate topic. A single node may publish and/or subscribe to multiple topics and there may be multiple nodes publishing and subscribing to a single topic.

The peer-to-peer topology requires some sort of search mechanism to allow processes to find each other at runtime. There is a ROS Master that makes possible a peer-to-peer communication among nodes, providing naming and registration services to the rest of the nodes in the ROS system. In the specific context of a mobile robot system, the interconnection between nodes facilitates a modular development where different tasks runs at the same time (*e.g.*, monitoring the robot state, data processing algorithms and commanding the robot).

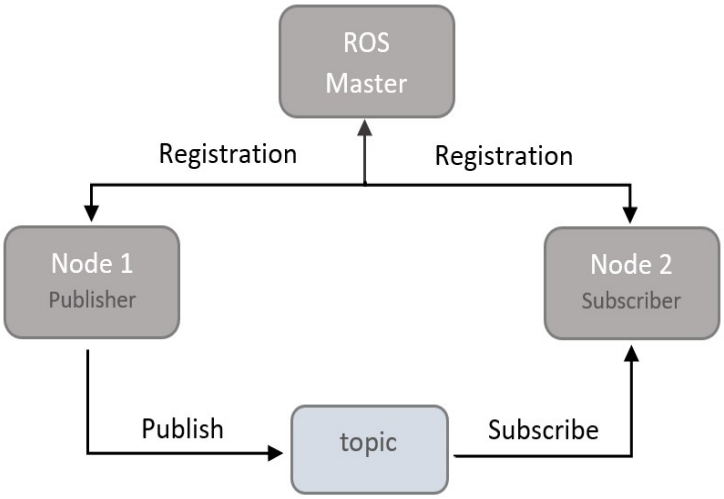

Figure 2.1: Node 1 and Node 2 are communicating with each other through a ROS topic; Node 1 is publishing information on a topic, while Node 2 subscribes to this topic to receive this same information.

2.2 Webots Simulator

The list of open source simulation tools is quite large which makes it difficult to decide which is the right tool for mobile robots. Among those open-source tools that have achieved some degree of maturity, it is possible to mention Gazebo and Webots (a comparative analysis of current robotics simulators can be found elsewhere https://en.m.wikipedia.org/wiki/Robotics_simulator). Although Gazebo has become one of the primary tools used by the ROS community, Webots became an alternative to consider since it has been released as free open source software in December 2018.

Webots is a 3D simulation platform developed by Cyberbotics which offers support for Windows, Linux and Apple platforms. This simulation software provides a rapid prototyping environment for modeling, programming and simulating various types of robots using the Open Dynamics Engine (ODE) [4] library for realistic physics simulation of rigid bodies. Perhaps because it was until recently a commercial product, it provides an easy to use graphical interface and the support for multiple programming languages (*e.g.*, C/C++, Python, Java, Urbi, Matlab). It is also worth mention its compatibility with external libraries such as Open CV

and the integration of a complete ROS programming interface (see "Using ROS with Webots"). The possibility of transferring the behaviour developed in simulation to a real robot is a characteristic of Webots that was designed as sensor-based simulator for mobile robotics [10].

In Webots you can program and simulate robots, vehicles and biomechanical systems. It contains a large number of sensors, actuators, controller program examples and robot models, that can be modified. A simulation in Webots embodies two concepts:

- The World.
- The Controller.

2.2.1 World

World is a file that contains all the information of the environment like position, orientation, geometry and appearance of the objects and the robots. It also has the information of how the different objects interact with each other, containing their gravity, friction and masses properties. Worlds are organized as hierarchical structures where objects can contain other objects, in VRML97 format . A world is stored in a file having the *".wbt"* extension.

The objects in the world file are called Nodes and they are organized hierarchically in a *Scene Tree*. A node can contain sub-nodes.

The Scene Tree

The scene tree contains contains the information that describes a simulated world, including the objects which are the Webots Nodes. Each node contains fields, which in turn can contain values (text strings, numerical values) or other nodes.

2.2.2 Controller

A controller is a computer program that controls a robot specified in a world file. Webots supports programming languages like C, C++, Java, Python or MATLAB,

thus controller programs can be written in any of these languages, making it very flexible.

Typically, Webots launches automatically the robot controller specified in the controller field of each Robot node, but if that field is set to *<extern>* it is possible to manually launch a controller, (for example from ROS as I will explain further) and Webots will try to connect to the controller.

2.2.3 Library

Webots has an extensive selection of models of robots, sensors, actuators and objects.

We can add nodes from the *Scene Tree* like **Solid** node which represents a rigid body, that is a body in which deformation can be neglected. The distance between any two given points on a rigid body remains constant in time regardless of external forces exerted on it. The **Solid** node is the most important base node in Webots from which many other nodes derive.

Another base node that can be added is the **Robot** node which can be used as basis for building a robot. It can contain, as well, nodes as was mentioned previously. In the **Robot** node we can add other nodes like: **Accelerometer**, **Camera** and **DistanceSensor**.

Aside those nodes, Webots contains other nodes that are called **PROTO** nodes that are defined in a PROTO file. You can make your own PROTO files but Webots contains standard PROTO nodes like *robots, vehicles and objects* that can be modified but should usually not be modified by the user. Basically, the PROTO files are objects that were created and can now be reused.

2.2.4 Webots simulation examples

To have some understanding of how Webots works, here follows a brief demonstration with two examples. In both examples was created a world with an *e-puck* robot . *E-puck* is a mini mobile robot developed by GCtronic and EPFL that has, among other devices, infrared sensors for proximity and light measurements, an

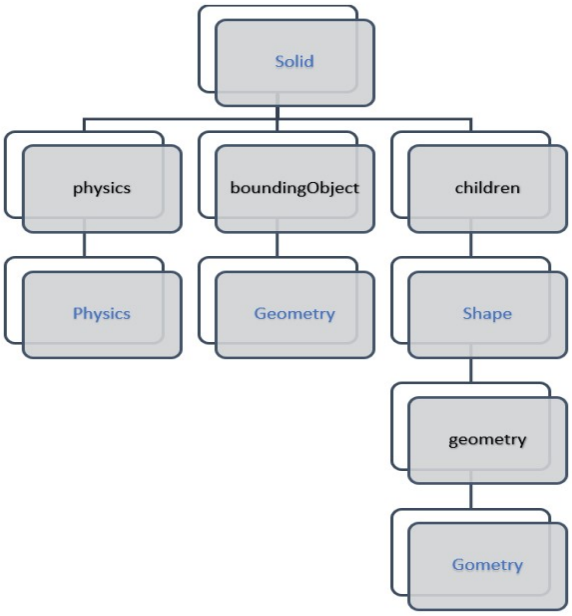

Figure 2.2: Solid model example.

accelerometer, a camera and ground sensors. It can be found in the PROTO nodes contained in webots. The purpose of both examples was to test the behaviour of the robot using different controllers in different environments. The controllers were written based on Python programming language.

To create a new project, from the Wizards menu, you select the *New Project Directory* menu item and then choose the name for the project directory and the name of the world file. To create a controller program, from the Wizards menu, you select the *New Robot Controller* menu item and then choose the language for the controller program and name it.

Maze environment and an avoid-collision controller

In the first example was created a maze-like environment and a controller program aimed to make the robot avoid obstacles.

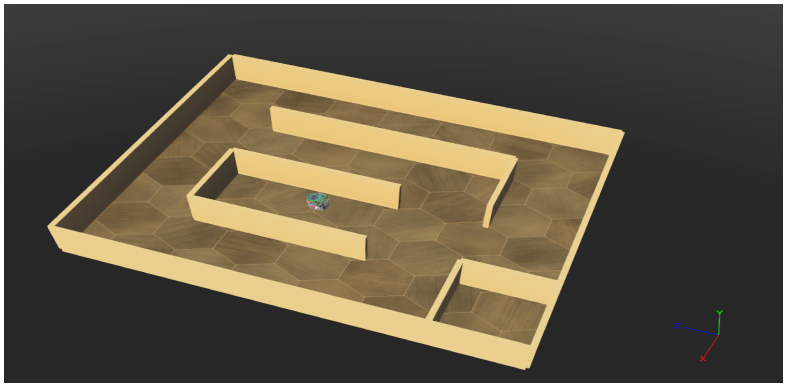

Figure 2.3: Maze.wbt.

1. **The Maze environment** :

   - All the walls were created from the **Solid** node, having a **Shape** sub-node where a box geometry was added, and the appearance was changed.
   - The floor was also created from the **Solid** node, having a **Shape** sub-node where a plane geometry was added and, in the appearance field, was added an appearance from a PROTO node in Webots.
   - The last node added was the *e-puck* robot from the PROTO nodes. One of the robots fields is the controller, and it is where the controller program will be associated with the robot.

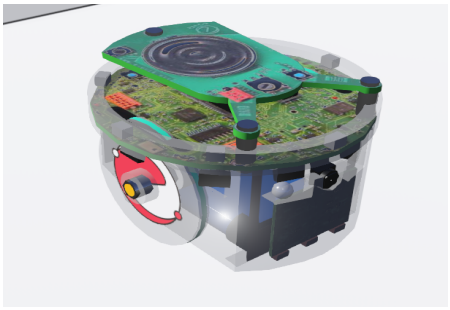

Figure 2.4: e-puck from Webots.

2. **The avoid-collision controller**

The controller made to avoid collisions is very simple. Taking into consideration the figure 2.6 that shows the values of the sensors against distance, the algorithm 1 was made:

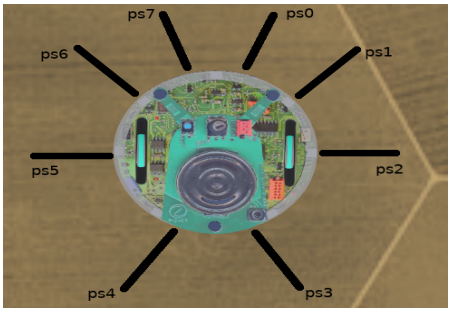

Figure 2.5: e-puck's sensors locations.

Algorithm 1 The avoid-collision controller

1: **if** one of the three infrared sensors at the front and on the right side measures values greater than 80 **then** ▷ there is an obstacle on the right
2: the velocity of the left wheel motor is reduced and the right wheel motor velocity is increased
3: **else if** the same occurs to one of the three infrared sensors at the front and on the left side **then** ▷ there is an obstacle on the left
4: the velocity of the right wheel motor is reduced and the left wheel motor velocity is increased
5: **else** ▷ no obstacles are detected
6: the wheel motor velocities maintain equal
7: **end if**

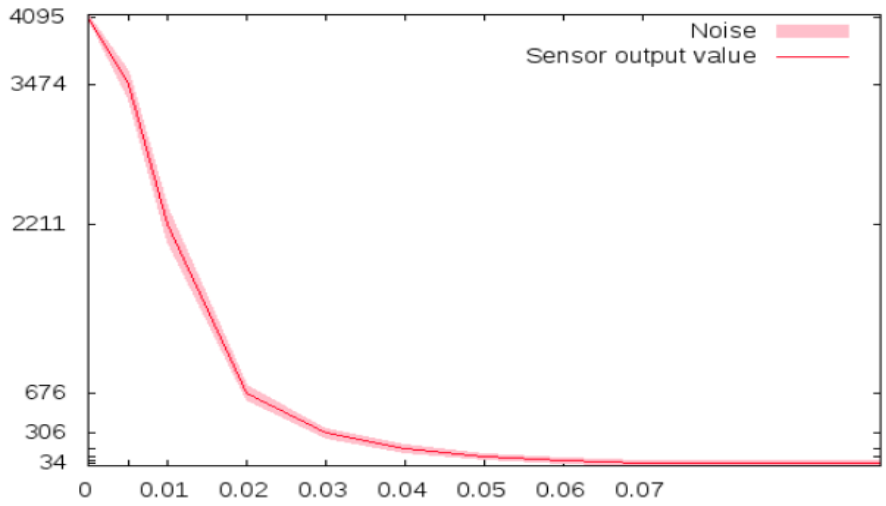

Figure 2.6: Sensor response versus obstacle distance.

Environment with a black line and a controller to follow it

For this example, was created an environment with a black line on the floor, and a controller program to make the robot follow the line. In this case, the e-puck robot uses the three infrared sensors pointing to the ground that it has at its front.

1. **The black line environment**

   I changed a few things for this environment in comparison to the maze environment. I removed the walls in the middle and changed the floor.

   Instead of using a PROTO node in the appearance field, it is possible to add a **PBRAppearance** node that specifies a physically-based visual appearance of a node, and in its fields you can specify the base color, the roughness of the material's surface, how metallic the material's surface is, among other things. One of the **PBRAppearance** node fields is *baseColorMap* and in it you can add an **ImageTexture** node where you can include an image. And that is what I did, adding an image that I made for this simulation.

   The new controller was associated with the robot, in the controller field of the robot.

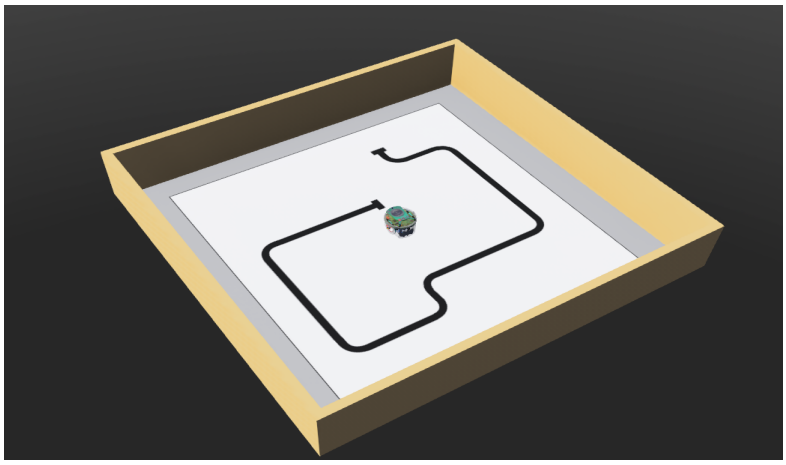

Figure 2.7: Follow_Black_Line.wbt.

2. **The follow-the-line controller**

   The algorithm to follow the line can be described as the following:

   It is not a robust algorithm, but in most cases, it will keep following the black line after finding it.

Algorithm 2 The follow-the-line controller

1: **if** the right sensor is the only one that detects the black line **then**
2: the velocities of the wheel motors should be adjusted in such a way that the robot turns right
3: **else if** the left sensor is the only one that detects the black line **then**
4: the velocities of the wheel motors should be adjusted in such a way that the robot turns left
5: **else**
6: the wheel motor velocities maintain equal
7: **end if**

Chapter 3

Interaction between ROS and Webots

This chapter provides a tutorial on how to use ROS with Webots using a solution named **Custom ROS controller**. The ROS-Webots integration is the key element of the present project around which the machine learning algorithms will be developed. A detailed description of the required steps for interfacing ROS with Webots will be provided in the following sections.

3.1 ROS-Webots integration

There are two ways to integrate the ROS framework with the Webots simulator (see Webots documentation):

1. **Standard ROS Controller**:
    - Webots has a ROS controller as one of its default controllers, that can be used on any robot in Webots and acts as a ROS node which creates ROS services or topics based on the Webots functions (robots, motors, sensors, etc). For instance, if there is a distance sensor in a robot, it will publish to a topic for the sensor values.
    - To use this controller, you have to change the controller field of the Webots robot node to *ros*.

- You will only have to build your nodes to communicate to the services and topics that this controller provides.

2. **Custom ROS Controller**:

   - In this way, you have to build your Webots controller that will be, like the other controller, a ROS node using Webots and ROS libraries.
   - Building your Webots controller gives you total control over the data that is published from Webots.
   - The custom ROS controller can be written in both C++ and Python. In Python you have to import the ROS libraries (with rospy) and the Webots libraries (controller).

The Robot Controller is used to control everything that it is related to the robot. The controllers implemented in Webots need to import its library, as in figure 3.1, to get the functions that control the robot. If you are building a Webots controller to use as a Custom ROS Controller, you will need to import the library as well.

```
#include <webots/Robot.h>
```

(a) C++

```
from controller import Robot
```

(b) Python

Figure 3.1: Importing Controller library.

3.2 Example of ROS integration to Webots

For the *Standard ROS Controller*, there is a *webots_ros* package that can be installed, and in it contains some ROS nodes that you can run to see the simulations. Those nodes give good support to start building your nodes and how to use Webots API functions, and interact with the services and topics that the *Standard ROS Controller* generates. Unfortunately, the nodes in the *webots_ros* package are only in C++ language.

For this example, I created a simple ROS package that contains some ROS nodes, made with Python, and the interaction with Webots was made through

the second way (**Custom ROS Controller**). Through it, will be explored some details on how ROS works and how the **Custom ROS Controller** interacts with Webots.

Once I have presented some examples of Webots simulations, I started from one of them, *i.e.*, the maze example made in 2.2.4 with the avoid-collision controller. The goal was to start the simulation and to control it through ROS.

This example is organized into three parts:

1. **How to write the ROS nodes**: explanation code of the *Webots Controller* node and the *avoid-collision controller* node.
2. **Creating and Building the ROS package**: explanation of how to build a simple ROS package, containing launch files.
3. **Running ROS nodes**: some procedures to run the ROS nodes and environment variables that must be set so that the node that communicates with Webots can run properly.

3.2.1 The Webots Controller and the avoid-collision controller nodes

Next follows the explanation of the code made for the Webots controller that is also a ROS node. In this node it is possible to understand how we can use the Webots API functions.

Algorithm used to create the Webots Controller ROS node :

Detailed explanation of some parts of the code 3.1 implemented for this algorithm 3:

- The *rospy* is a Python client library for ROS that is needed for writing nodes.
- The *controller* is used to get the functions that control the robot and Webots environment.
- *Data* is a message type file that a created to be used by the topics.

Algorithm 3 The Webots Controller ROS node

```
1: Get the robot used in Webots
2: Get the time Step (in milliseconds) of the simulation, specified in the wbt file
3: Get the motors used by the robot
4: Initialize the devices (infrared sensors), using the names that were specified in
   the wbt file
5: Initialize the ROS node
6: Subscribe to the motor topic and create the sensors topic that the node will
   publish to
7: while the simulation is running and the node isn't closed do
8:     read the sensor values
9:     publish the sensor values to the sensors topic
10:    set the velocities of the robot's motors
11: end while
12: if the topic motor receives news messages then
13:    the callback function, associated with the motor topic, is invoked
14:    get the massage published to the motor topic
15:    changes the motor's velocities
16: end if
```

- *robot* creates the Robot instance.
- *timeStep* is the value returned by the basicTimeStep field of the Webots WorldInfo node, and it defines the duration of the simulation step executed by Webots.
- *getMotor(arg)* returns motor associated to the robot, with the name equals to arg.
- *getDistanceSensor(arg)* returns distance sensor associated to the robot, with the name equals to arg.
- *rospy.init_node* creates a ROS node with the name *e_puck_maze* which is the one that connects to Webots, being the Webots controller.
- *rospy.Subscriber('motor', Test , callback)* declares that the e_puck_maze node **subscribes** to the topic "motor" which is of type "Datat.msg". Callback is invoked when it receives new messages.
- *rospy.Publisher('sensors', Test , queue_size =10)* declares that the node **publishes** to the topic "sensors" which is of type "Data.msg".

- In the while loop, the sensors values are obtained using *getValue()* and they are published to the *sensors* topic.
- The motors are actuated by *setVelocity()*, with the values published to the *motor* topic.

```
#!/usr/bin/env python

import rospy
from controller import Robot
from webots_example.msg import Data
import os

def callback(data):
    global left_velocity
    global right_velocity
    global message

    velocities = []
    for i in range(2):
        velocities.append(data.data[i])

    left_velocity = velocities[0]
    right_velocity = velocities[1]

# create the Robot instance.
robot = Robot()
# returns the value of the basicTimeStep field of the WorldInfo
    node
timeStep = int(robot.getBasicTimeStep())

# Motors
left = robot.getMotor('left wheel motor')
right = robot.getMotor('right wheel motor')

# initialize devices
ps = []
psNames = [
    'ps0', 'ps1', 'ps2', 'ps3',
    'ps4', 'ps5', 'ps6', 'ps7',
    'gs0', 'gs1', 'gs2'
]
```

```
36
37 for i in range(11):
38     ps.append(robot.getDistanceSensor(psNames[i]))
39     ps[i].enable(timeStep)
40
41 # turn on velocity control for both motors
42 left.setPosition(float('inf'))
43 right.setPosition(float('inf'))
44 left_velocity = 0
45 right_velocity = 0
46 left.setVelocity(left_velocity)
47 right.setVelocity(right_velocity)
48
49 robot.step(timeStep)
50 rospy.init_node('e_puck_maze', anonymous=True)
51
52 # Motor topic
53 robot.step(timeStep)
54 rospy.Subscriber('motor', Data, callback)
55
56 # Sensors topic
57 pub = rospy.Publisher('sensors', Data, queue_size=10)
58
59 # feedback loop: step simulation until receiving an exit event
60 while robot.step(timeStep) != -1 and not rospy.is_shutdown():
61
62     # send sensors outputs to sensors topic
63     sv = [];
64     for i in range(11):
65         sv.append(ps[i].getValue())
66     sensor_data = Data()
67     sensor_data.data = sv
68     pub.publish(sensor_data)
69
70     left.setVelocity(left_velocity)
71     right.setVelocity(right_velocity)
```

Listing 3.1: e_puck_maze.py

The other ROS node made in this example is a node that subscribes to the *sensors* topic to get the values of the sensors, executes the algorithm 1 and publishes the result at the *motor* topic.

Here follows the algorithm:

Algorithm 4 The avoid-collision controller ROS node

1: Initialize the ROS node
2: Subscribe to the *sensors* topic and create the *motor* topic that the node will publish to
3: **if** the topic *sensors* receives news messages **then**
4: the callback function, associated with the *sensors* topic, is invoked
5: the algorithm 1 to avoid collisions is executed
6: the motor's velocities are published to the *motor* topic
7: **end if**

```
#!/usr/bin/env python

import rospy
from webots_example.msg import Data

def callback(data):
    global MAX_SPEED
    global pub

    # read sensors outputs
    psValues = []
    for i in range(8):
        psValues.append(data.data[i])

    # detect obstacles
    right_obstacle = psValues[0] > 80.0 or psValues[1] > 80.0 or
   psValues[2] > 80.0
    left_obstacle = psValues[5] > 80.0 or psValues[6] > 80.0 or
   psValues[7] > 80.0

    # initialize motor speeds at 50% of MAX_SPEED.
    leftSpeed  = 0.5 * MAX_SPEED
    rightSpeed = 0.5 * MAX_SPEED
    # modify speeds according to obstacles
    if left_obstacle:
        # turn right
        leftSpeed  += 0.5 * MAX_SPEED
        rightSpeed -= 0.5 * MAX_SPEED
    elif right_obstacle:
        # turn left
        leftSpeed  -= 0.5 * MAX_SPEED
```

```
30          rightSpeed += 0.5 * MAX_SPEED
31
32      vel = [leftSpeed,rightSpeed]
33      vel_data = Data()
34      vel_data.data = vel
35      pub.publish(vel_data)
36
37  MAX_SPEED = 6.28
38
39  rospy.init_node('avoid_collision_controller', anonymous=True)
40  pub = rospy.Publisher('motor', Data, queue_size=10)
41  rospy.Subscriber("sensors", Data, callback)
42  rospy.spin()
```

Listing 3.2: e_puck_avoid_collision_controller.py

After writing the nodes, it is necessary to make them executable. You can do this by going to their directory via terminal and writing the command:

```
chmod +x e_puck_avoid_collision_controller.py
```

Listing 3.3: Making the file executable.

3.2.2 Creating and Building the ROS package

I decided to make a package to put the ROS nodes I created, and to create launch files that are common in ROS, once they make easy to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

1. The package was created with the name *webots_example*, in the source space directory of the catkin workspace, using the *catkin_create_pkg*. Its dependencies are *rospy* and *std_msgs* and, once created, generated a *CMakeLists.txt* file and a *package.xml* file. The *CMakeLists.txt* file describes how to build the code and where to install it to, and the *package.xml* file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.

   ```
   catkin_create_pkg webots_example std_msgs rospy
   ```

   Listing 3.4: Creating the package.

2. In the package folder, was created a launch folder with two launch files. One of the launch files is called *Maze.launch* that is responsible for launching the *e_puck_maze* node and the launch file called *Webots.launch* that is responsible for launching Webots. In the first launch file is specified msgthe location of the *wbt* file and the node, which is the Webots controller (e_puck_maze.py). The *Webots.launch* file was copied from the *webots_ros* package and altered the package field.

3. After create the launch files, I created a msg folder with a *msg* file. This is the message type used by the topics, and was named *Data.msg*.

```
float64[] data
```

Listing 3.5: Data.msg.

4. The nodes created were copied into the package source folder, along with the *webots_launcher.py* copied from the *webots_ros* package.

5. Some changes were made in *CMakeList.txt*:

   - *message_generation* was added to *find_package* so I could generate messages.
   - At *catkin_package()*, the line with CATKIN_DEPENDS was uncommented and added *message_runtime*.
   - At *add_message_files* was added the msg file created.
   - *generate_messages* was uncommented.
   - The Python scripts and launch files were marked for installation.

6. Some lines were uncommented in *package.xml*:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

Listing 3.6: Changes in package.xml.

7. For last, I went to the catkin_workspace and build the package.

```
catkin_make
```

Listing 3.7: Build package.

3.2.3 Running the nodes

A change was made in the *wbt* file to connect to an external controller. When the controller field of the robot is set to *<extern>*, no controller is launched while Webots controller is not launched manually on the same computer. When the Webots controller is launched, it tries to connect to the <extern> robot controller.

Before start the simulation, you must set or extended some environment variables. In the workspace, you have to source the workspace setup with the command:

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

Listing 3.8: Source ROS environment.

To use an external Webots controller, the WEBOTS_HOME, the controller language and the LD_LIBRARY_PATH must be set.

```
export WEBOTS_HOME=/usr/local/webots
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$WEBOTS_HOME/lib/
   controller
export PYTHONPATH=$PYTHONPATH:$WEBOTS_HOME/lib/controller/python27
```

Listing 3.9: Environment Variables.

After set the environment variables, you just have to run the command 3.10 and it will initiate the *e_puck_maze* node and open the *Maze.wbt* world from the figure 2.3.

```
roslaunch webots_example Maze.launch
```

Listing 3.10: Roslaunch.

In a new terminal that you have to re-source the workspace setup and set the environment variables, you run the *e_puck_avoid_collison_controller.py* with the command 3.11.

```
rosrun webots_example e_puck_avoid_collison_controller.py
```

Listing 3.11: Avoid_collision_controller node.

After running the *e_puck_avoid_collision_controller.py*, the e_puck robot stars moving and avoiding collisions. In the figure 3.2 it is possible to see the interaction between the nodes.

Figure 3.2: ROS graph.

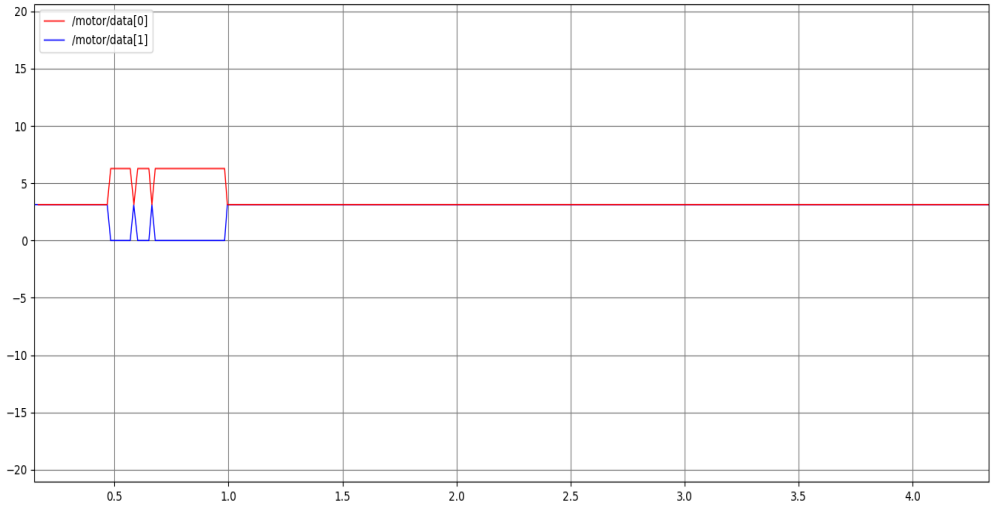

Figure 3.3: Motors values versus time.

In the figure 3.3 it is possible to see the change of the motors values when the robot encounters an obstacle. When there isn't an obstacle, the motors have the same velocity that is 3.14 rad/s. If there is an obstacle, the velocity of one motor is increased and the other decreased, depending on which side was the obstacle.

Chapter 4

Reinforcement Learning for Robot Navigation

This chapter describes the development of a robot controller based on a reinforcement learning approach adopted in [1]. The navigation problem was solved using a hierarchical structure with two levels. The high-level task selects one of the low-level tasks (or primitive actions) in order to navigate in a maze-like environment consisting of corridors, corners and doors. All sub-problems in this two-level hierarchy are solved through a tabular Q-learning algorithm [11]. Another particularity of the hierarchical control is the existence of checkpoints (topological nodes) placed at the arrival at (or departure from) a specific door. The high-level task must decide which is the optimal decision to be taken at the discrete checkpoint, while the lower-level is responsible for the robot's behavior between two checkpoints.

4.1 Hierarchical implementation

This section describes the ROS-based Phyton implementation of the Hierarchical Reinforcement Learning algorithm adapted from a previous ROS package written in C/C++ [1]. As mentioned above, HRL structure is decomposed into two levels:

- The lower-level: it is in charge of dividing the maze into distinct sub-tasks.

The sub-tasks chosen for this maze (4.2) are "Right turn on a door", "Left turn on a door", "Going Forward on a door" and "Corridors and Corners". Thus, this level is responsible for the learning of how the robot should perform these sub-tasks.

- The higher-level: it is in charge of making the decision, choosing the sub-task it needs to execute at a given time. It takes command when it detects a checkpoint.

A ROS node was made to perform an algorithm implemented for this HRL. It aims to determine the actions that the robot should take, knowing the learning tables obtained by this learning.

Algorithm 5 Perform Learning ROS node

```
initialize the ROS node
load the learning tables
subscribe to the sensors_data topic
if the topic sensors_data receives new messages then
    the callback function, associated with the sensors_data topic, is invoked
    obtain the sensors distances and the checkpoint
    determine the levels of each sensor
    calculate the state
    if checkpoint ≠ 0 then                                   ▷ Higher-level
        execute the algorithm that chooses the sub-task
        perform the sub-task                                 ▷ Lower-level
        determine the action
    else                                                     ▷ Lower-level
        perform the sub-task related to "Corridors and Corners"
        determine the action
    end if
end if
```

Another ROS node was implemented to communicate with Webots and perform a simulation.

Algorithm 6 Data publication ROS node

```
get the robot used in webots
get the time step of the simulation
get the motors of the robot
initialize the sensors of the robot
initialize the ROS node
subscribe to the action topic
indicate the topic sensor_data that it will publish to
while the simulation is running do
    read the sensors values
    perform the algorithm 7
    convert the sensor values to distances, in cm
    determine checkpoint
    publish the sensors values and the checkpoint to sensor_data topic
    set the velocities of the robot's motors
end while
if the topic action receives new messages then
    the callback function, associated with the action topic, is invoked
    obtain the action
    change the motor's velocities
end if
```

4.2 Topological navigation

4.2.1 Implementation of the checkpoints

Topological navigation uses a topological representation of the environment where navigational processes happen. A topological map describes the environment with topological nodes, thus providing a map of connectivities between the nodes in the environment. These nodes will be called checkpoints, and they represent discrete areas of the environment to help the robot know where it is.

In this case, the topological structure (*i.e.*, the discrete checkpoints) consists of barcodes placed in the environment, that represent discrete areas such as hallway intersections. These barcodes allow scanning in both directions for differentiate between arrival and departure situations.

The barcodes in figure 4.1 were implemented in the world file 2.2.4, and each one consists of 5-bars of 1.7 cm width that occupy the width of the corridor.

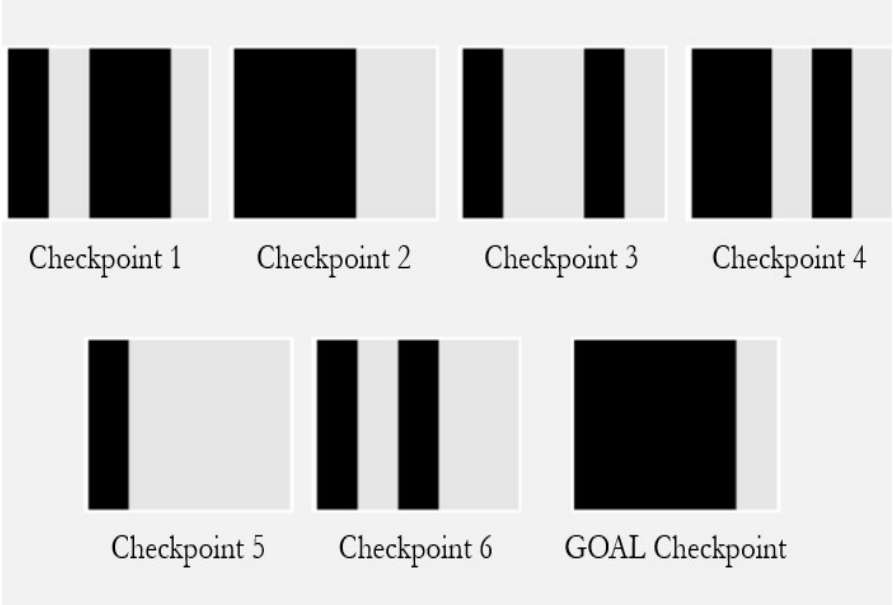

Figure 4.1: Barcodes placed on the ground environment.

The inclusion of the barcodes in the *Maze.wbt* file, in Webots, was made using the **Solid** node with a **Shape** that contains a **Plane** geometry, and adding the figure of the barcode in the appearance field of the **Shape**. In total, was made seven **Solid** nodes, that correspond to each barcode in figure 4.1.

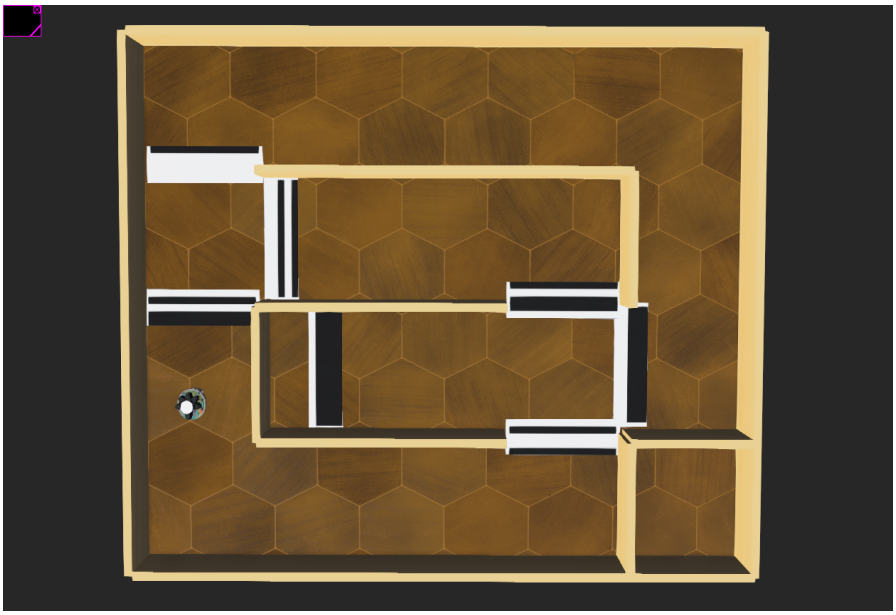

Figure 4.2: Maze.wbt with checkpoints.

For the reading of the checkpoints, the robot uses an infrared ground sensor,

that outputs values as the figure 2.6 shows, and to convert these readings to determine the checkpoints, the algorithm 7 was implemented.

Algorithm 7 Barcode reader

```
1: if white or black colours are detected then
2:     start reading
3:     while other colour is not detected do
4:         continue reading
5:     end while
6:     n ← size(codes)/5
7:     for i = 1 to 5 do
8:         positions_of_codes_to_use ← n × i − n/2
9:     end for
10:    for i = codes_to_use do
11:        if i ≥ 700 then                          ▷ white barcode
12:            barcode = 1
13:        else                                     ▷ black barcode
14:            barcode = 0
15:        end if
16:    end for
17:    convert binary code number to decimal
18:    compare decimal number to determine the checkpoint
19: end if
```

4.2.2 Evaluation of the algorithm for checkpoints reading

Some tests were made to demonstrate how competent the algorithm 7 is. The method chosen for the tests was to put the robot navigating in the maze, using an algorithm similar to the algorithm 2 but added the condition of the robot turning around itself if it detects an obstacle in front.

Is was added six infrared sensors, Sharp GP2Y0A41SK0F, to the *e-puck* robot, to measure distances in the six directions (left, front left, right, front right, front and rear). Webots contains some predefined commercialized infrared sensors models, and, conveniently, the one I needed was one of them.

In figure 4.3, it is possible to see the percentage of right and wrong readings, among two hundred readings, using different thresholds to determine if there is an obstacle. The thresholds were chosen according to figure 4.4, being the distance from the obstacles between eleven and six centimeters.

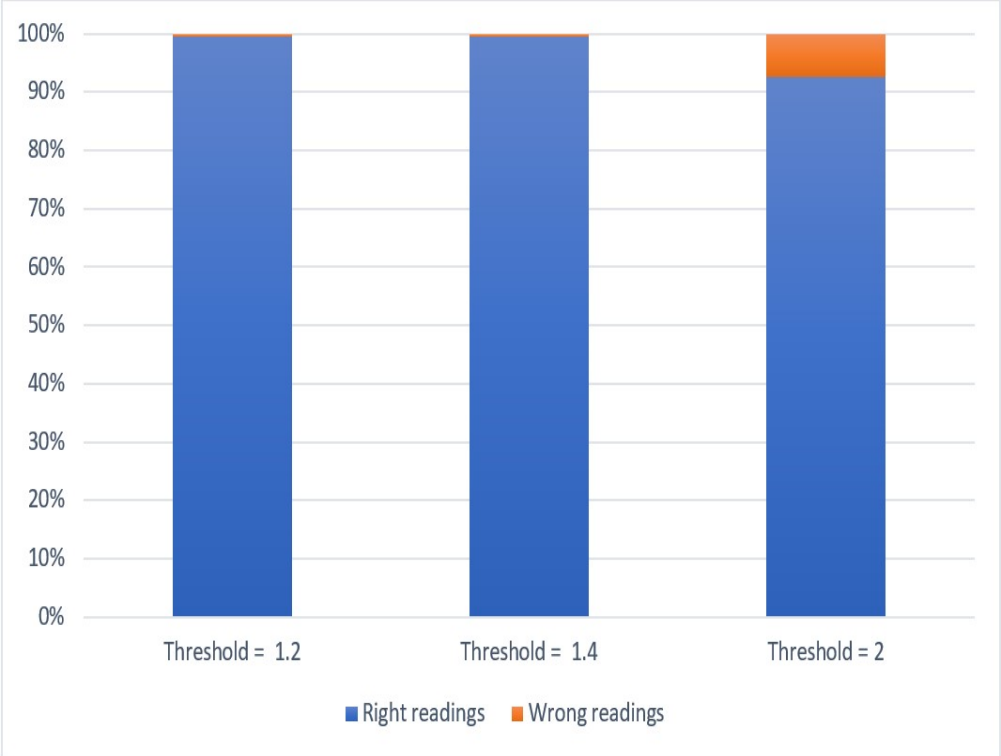

Figure 4.3: Percentage of right and wrong readings, in 200 readings, with different thresholds.

From the figure 4.3 we observe that the algorithm 7 determines the checkpoints, with success, almost every time when the the thresholds are 1.2 and 1.4. With a higher threshold, which corresponds to a lower distance to find an obstacle, the algorithm becomes less effective. That is because the robot can be closer to the walls and, in certain conditions, it will find the obstacles while reading the barcodes and apply different velocities to the motors, resulting in a bad reading. With the possibility of being closer to the walls, sometimes the robot will approach the areas where the barcodes are, very diagonally and will find an obstacle before finishing the reading, which will result in a bad reading as well. The algorithm can read barcodes in diagonal but, if there is a lot of changing of direction, while the reading is occurring, it will not be successful.

In figure 4.5, it is possible to see that increasing the velocity of the motors, the algorithm becomes weak to determine the checkpoint. The reasons are quite similar to the previous mentioned, for instance, if the robot changes direction while reading the barcodes, these changes become more accentuated when the velocity is increased. Additionally, with more speed the sensors do less reading, ruining

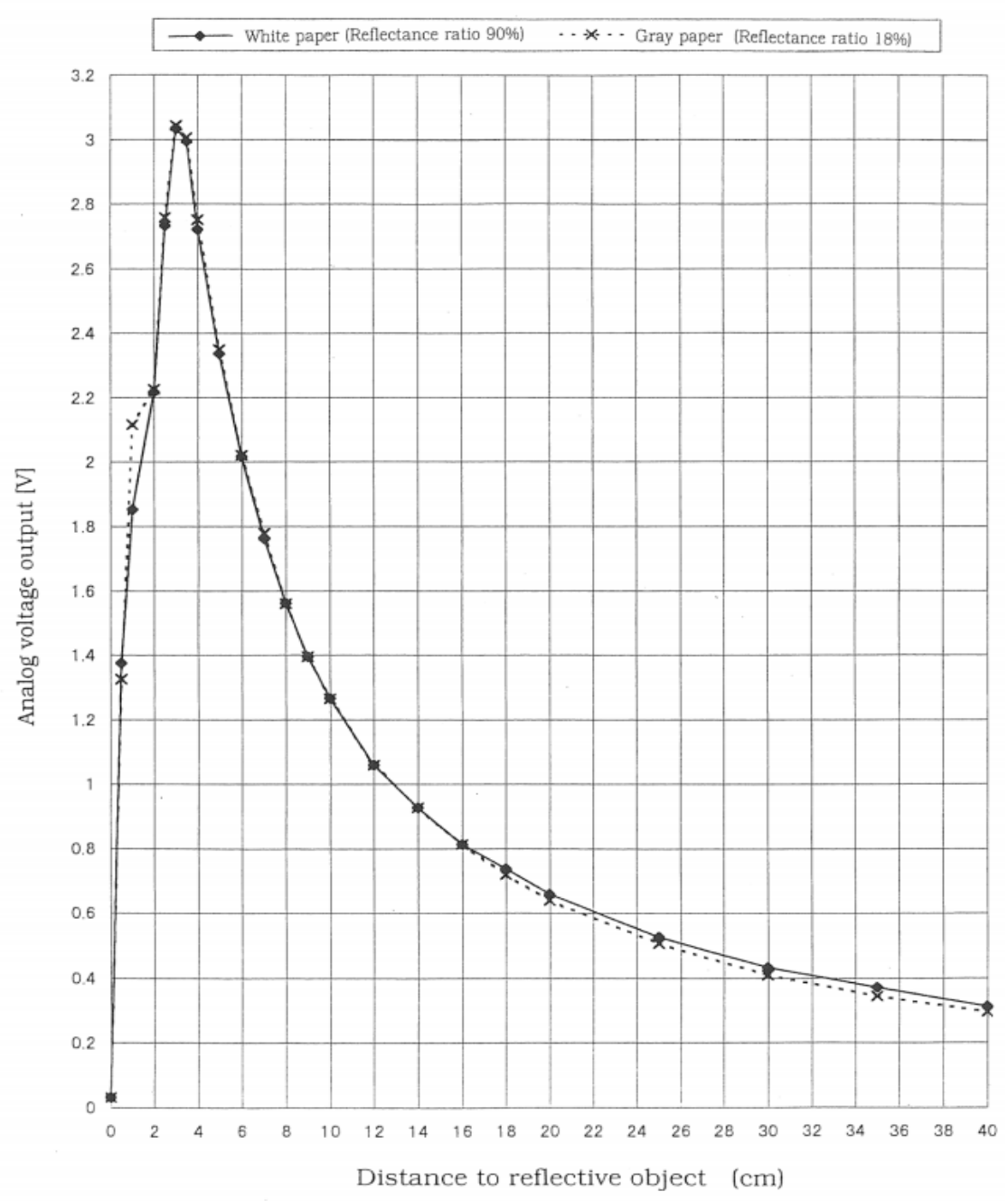

Figure 4.4: Output voltage corresponding to the distance.

the interpretation of the barcodes.

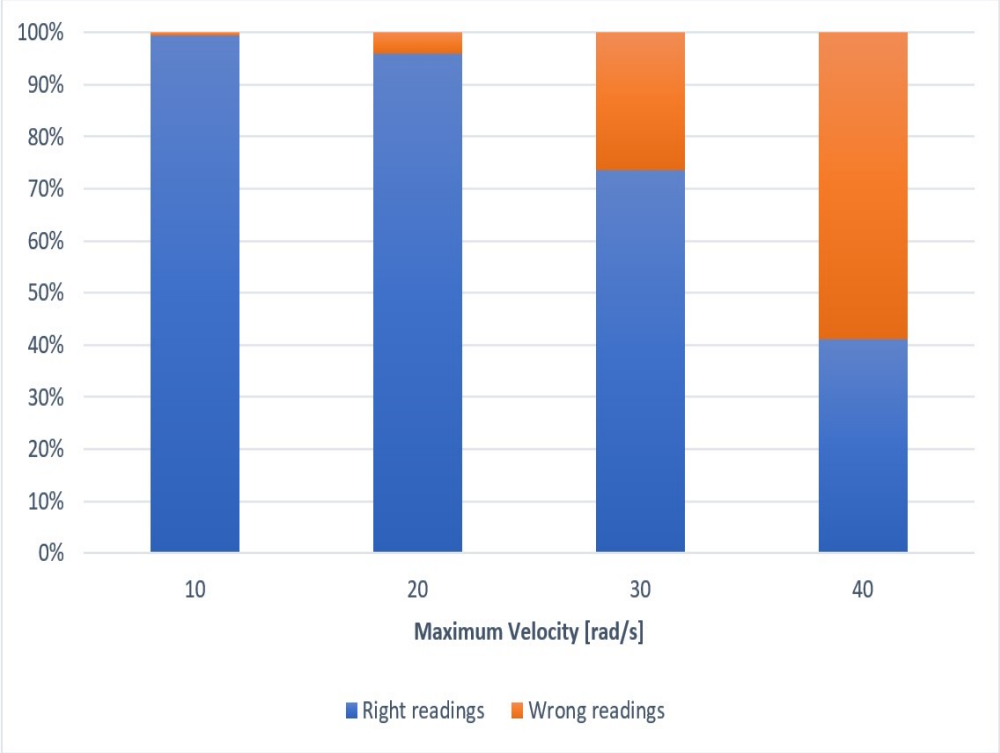

Figure 4.5: Percentage of right and wrong readings, in 200 readings, with different velocities.

4.3 Experiments and results

Having the ROS-based implementation of the HRL, some experiments were made with the intention of understand whether the robot reach the GOAL, using the RL learning tables generated in [1]. The results were not as expected, since the robot struggled to navigate in the maze and did not reach the goal. In fact, it sometimes hits the walls. Although, it could sometimes cross a hallway, and even turn in a corner.

Through debugging, it was possible to observe that when the robot reaches a checkpoint and identifies it with success, the Higher-level selected the right subtask, so the implementation appears to be done as desired. To demonstrate this, it is presented next some figures with some trajectories, and some outputs on the terminal, that will help explain the behaviours of the robot.

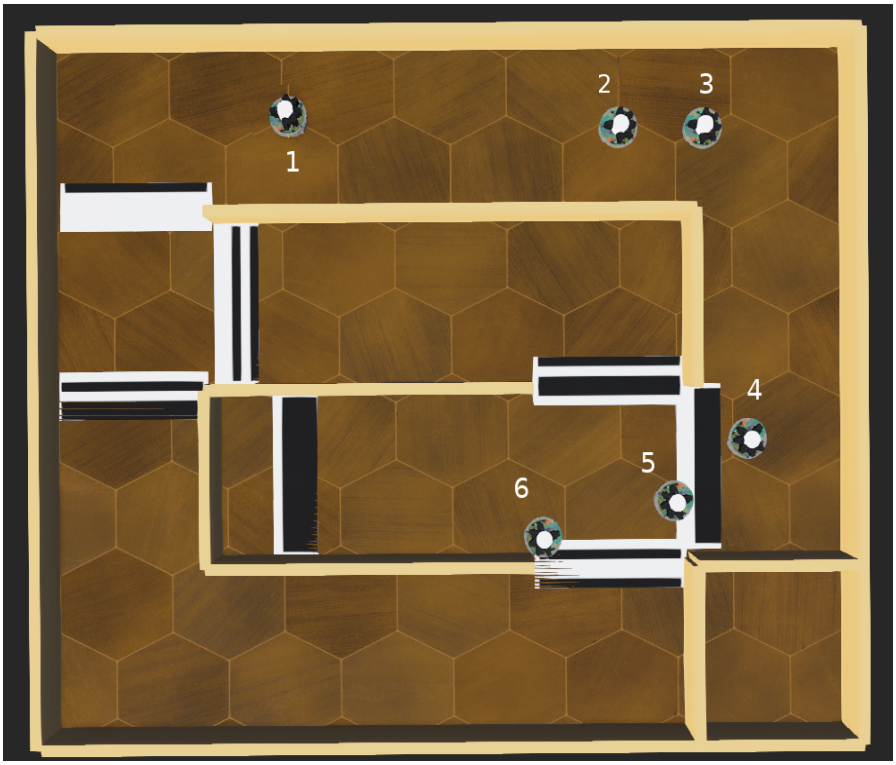

Figure 4.6: First trajectory example.

The trajectory represented in 4.6, begins with the robot in the position one, and ends at six. From the figure 4.7 we can see that, in the first robot's position, the level of each sensor is as expected. To understand that, it is necessary to know how the levels of the sensors are determined. The sensor levels definition is represented in table 4.1.

Measured Distances (cm)	**Level**
$0 \leq d < 7$	0
$7 \leq d < 13$	1
$13 \leq d < 20$	2
$d \geq 20$	3

Table 4.1: Discretization of measured distances.

For the right comprehension of the sensor levels, it is also necessary to know that the output follows the following sequence: front sensor, back sensor, left sensor, right sensor, left-front sensor and right-front sensor. The actions represent the velocities applied to the robot's motors.

$$00 \rightarrow forward$$

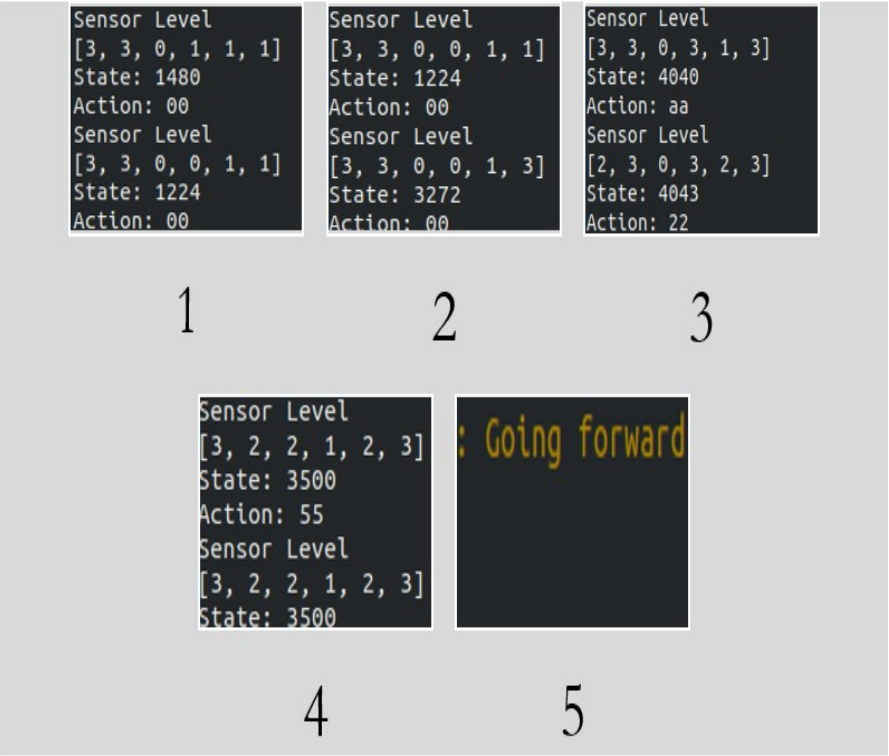

Figure 4.7: Terminal Outputs of the first trajectory.

$$11 - 44 \rightarrow turnright$$

$$55 - 88 \rightarrow turnleft$$

$$aa \rightarrow turnright, inhisaxis$$

$$bb \rightarrow turnleft, inhisaxis$$

$$99 \rightarrow backward$$

As was previously mentioned, the sensors levels are as expected, once the front sensor is pointed to the end of the hallway, the sensor returns a distance that belongs to the level 3. The same occurs to the back sensor and is easy to see the other levels are well determined. The action for this case is to go forward. When the robot reaches the second position, the right-front sensor is not detecting the right wall anymore, as can be confirmed in figure 4.7 (2), on the last sensor level output. At the third position, it begins to turn right, being the first actions "aa" and "22". The robot is able to turn, without hitting, and repeats the same actions when it reaches the forth position.

When passing through the checkpoint, it recognizes it with success, and the higher-level determines that it should perform the sub-task of "Going Forward", and that is what it is represented on the fifth output, with a warning. From this example, it appears that the robot behaves as we want it, but if we look at the sixth position, it hit the wall. That is because it was told to go in front when it was not completely pointed to forward.

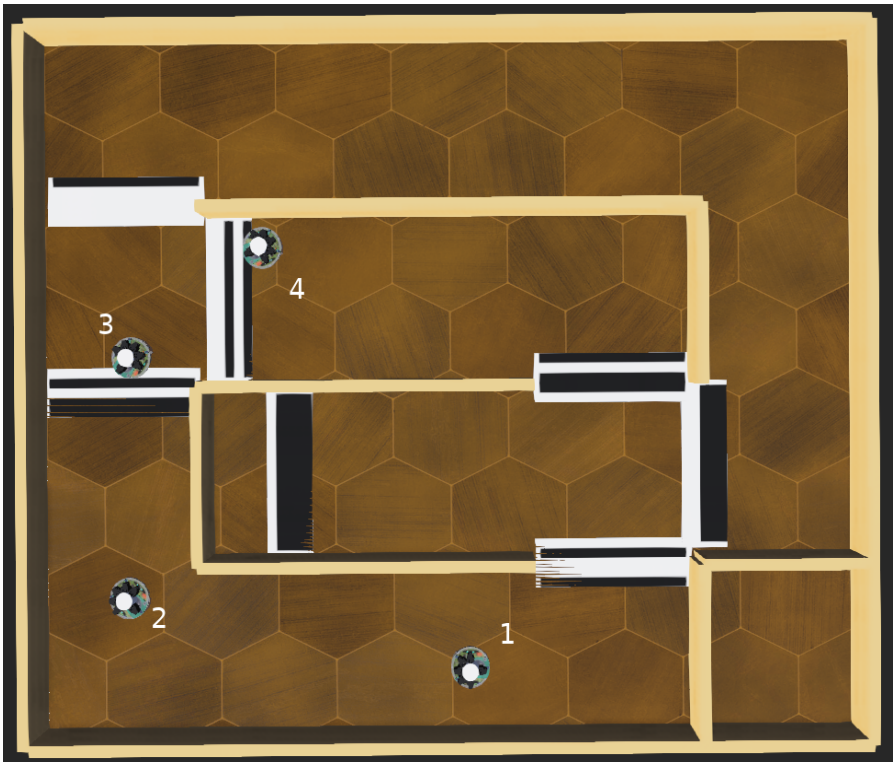

Figure 4.8: Second trajectory example.

In the second example (figure 4.8), the robot also performed the trajectory with success for some time, but failed to achieve the final goal. In position 3, the Higher-level determined that it should perform the sub-task "Go Right", but after that it began to determine actions that would make the robot turns to the left. It only made it because it approached to the checkpoint 4 diagonally, hence making the robot to be already pointed to the right side. The example ends with the robot hitting the wall. In other simulations, it did not perform the same actions as in the examples presented, and it appears to struggle with the sub-task "Go Left". Yet, it is clear that it tries to make the expected trajectories, only knowing the learning tables.

In this context, Webots helped a lot in the comprehension of the results, hence it is possible to pause the simulation and move the robot, if it pleased, giving a good flexibility to detect flaws and attack the problem from different perspectives.

Chapter 5

Conclusions

This project aimed at the development of a set of computational tools for integration into a mobile robot system under development. At the end of this project, it is possible to say that the objective of selecting robot simulator and integrating it in a ROS-based framework was achieved. On the one hand, the ROS framework provides added value for future developments in which machine learning algorthims will be exploited. On the other hand, the Webots simulator is a powerful open source software that can be programmed with multiple languages, giving infinite possibilities to simulate a lot of robots in the desired environments. It allows building the desired robot from scratch or using (and adapt) the various robots and sensors among other things, that it contains in its library. The simulations can accurately simulate real world environments, especially if the physics properties are well filled.

For the second part, it was created a ROS package having in mind a previous dissertation work [1]. The implementation of the hierarchical structure was revised and new functionalities were added to help the interpretability of results. From the experiments, it was proved that the HRL was implemented with success despite the problems found while running the experiments. The results show that the higher-level was determining the right sub-tasks when achieving the checkpoints, and the robot proved to have learned to do some navigation in the maze. The problem may reside in the use of not optimal learning tables for the adopted mobile robot. Nevertheless, the problem found through the simulations seems to be the synchronization between the ROS node and Webots. It is possible to

synchronize ROS and Webots in such a way that if the simulation speed, in Webots, is increased or decreased, while the ROS node is able to go along with that changing speed. These mechanisms must be better explored in future work, together with the assessment of effectiveness in transferring knowledge from simulations to the real robot.

References

[1] Silva, D. (2019). Mobile robot navigation using reinforcement learning. Master dissertation, University of Aveiro.

[2] Webots: robot simulador [Online]. Available: https://www.cyberbotics.com/

[3] Gazebo [Online]. Available: http://gazebosim.org/

[4] Open Dynamics Engine [Online]. Available: http://www.ode.org/

[5] ROS.org, Documentation – ROS Wiki [Online]. Available: http://wiki.ros.org/

[6] Martinez, A., and Fernández, E. (2013). Learning ROS for Robotics Programming. Packt Publishing.

[7] Michieletto, S., Ghidoni, S., Pagello, E., Moro, M., Menegatti, E. (2014). Why teach robotics using ROS? Journal of Automation Mobile Robotics and Intelligent Systems, 8(1):60-68.

[8] Ivaldi, S., Peters, J., Padois, V. and Nori, F. (2014). Tools for simulating humanoid robot dynamics: A survey based on user feedback," 2014 IEEE-RAS International Conference on Humanoid Robots, Madrid, 2014, pp. 842-849, doi: 10.1109/HUMANOIDS.2014.7041462.

[9] de Meneses, and Y.L., Michel, O. (1998). Vision sensors on the Webots simulator. In: Heudin, J.C. (ed.) Virtual Worlds. pp. 264–273. Springer Berlin Heidelberg, Berlin, Heidelberg.

[10] Michel O. (1998). Webots: Symbiosis between virtual and real mobile robots. In: International Conference on Virtual Worlds. pp. 254–263. Springer.

[11] Sutton, R., and Barto, A. (1998). Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA.

[12] Kober, J., Bagnell, J, and Peters, J. (2013). Reinforcement learning in robotics: A survey. The International Journal of robotics Research, 32(11):1238-1274.