Robotics Practical: ROS Basics

MICRO-453, Vaios Papaspyros

May 18, 2020

Nom: GRESSIER Chantal SCIPER: 246488
Nom: KIMBLE Thomas SCIPER: 261204
Nom: PILLONEL Ken SCIPER: 270852

Program P14

Contents

1 Introduction 3

2 Robot model 3

3 Velocity control 4

4 Obstacle avoidance 6

5 Discussion and Conclusion 8

6 Appendix A: Robot model code in URDF 9

7 Appendix B: Control algorithm code in Python 17

8 Appendix C: Instructions for launching examples 24

1 Introduction

The aim of this practical was to go through the basics of ROS, first by designing a two-wheeled robot, then by implementing a velocity control algorithm and finally an obstacle avoidance algorithm. It is a good exercise to learn about communication between different nodes (publishers/subscribers). The design of the robot is done in the URDF format and the navigation algorithm of the robot is done thanks to Gazebo plugins through a Python code. We used Gazebo to visualize and test our solution.

2 Robot model

Our robot model is based on the existing Thymio Robot. It has a rectangular shape with two cylindrical wheels on each side, and a sphere at the front to keep the robot level. This gives us three degrees of freedom ($DOF = 3$: Tx, Ty and Rx) and a mobility of $M = 2$.

We implemented three sensors, one at front of the robot and one on each side, which are used for the obstacle avoidance algorithm. We use three to allow for accurate wall following in both left and right directions.

The model description was written in the URDF format through an XML file (Appendix A). Figure 1 shows the robot schematics and its dimensions with the following points:

- MP: *Main_Body* Link Origin
- S: Structural Sphere Origin
- RW: *Right_Wheel* Joint Origin
- LW: *Left_Wheel* Joint Origin

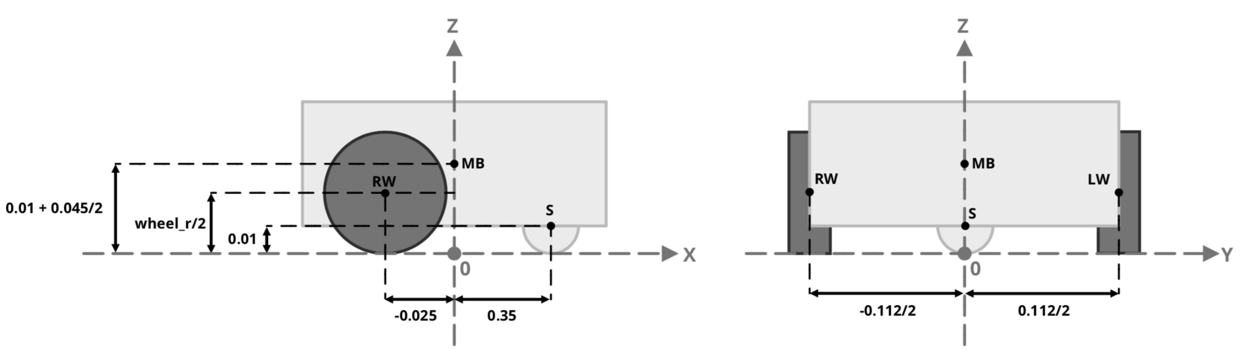
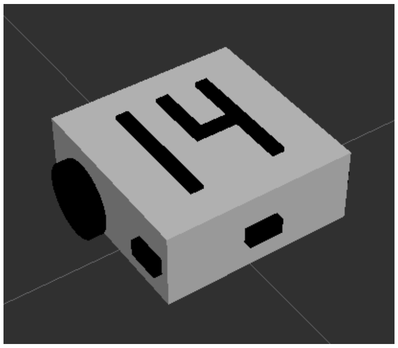
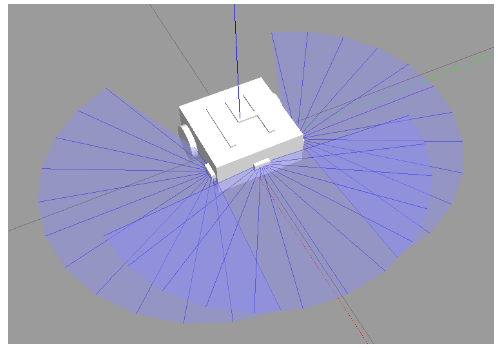


Figure 1: Robot schematics

Figure 2 shows the rendered robot in Rviz and Gazebo. Please note that we added a few custom features such as our group number and visual markers for the sensors which are not represented in the schematic found in Figure 1, these are purely visual.



(a) Robot render in Rviz



(b) Robot render in Gazebo

Figure 2: Robot model renders with sensors and custom features in Rviz and Gazebo

3 Velocity control

In order to make the robot move, we used Gazebo's differential drive plugin and computed the motion of the robot in the following code: *my_robot_control.py* (Appendix B). The velocity control algorithm is composed of two main controllers: a PD controller for the position and another PD controller for the angle. The tuned values of the PD controllers are the following:

- Position controller:
  - $K_p = 0.3$ $[s^{-1}]$
  - $K_d = 0.08$ [ ]
- Angle controller:
  - $K_p = 1.2$ $[s^{-1}]$
  - $K_d = 0.2$ [ ]

To make it move, a certain trajectory is given to the robot through waypoints by subscribing to the *goal_pos* topic where the goal positions are published by the user. Once the goal is set, the control loop is responsible for publishing the appropriate velocities to the *odom* Topic to move the robot.

At each timestep, the function *navigation()* computes the angle and distance errors between the waypoint and the robot's pose, and the *obstacle_check()* function is called to check for forward collisions.

While moving, if the norm of the current position error is larger than the norm of the old position error, the robot is stopped and the navigation is restarted from the current position of the robot. In this way, we avoid a bug in the Gazebo plugin where we found that the robot aligns itself with the x or y axis for small angles.

If the norm of the current angle error is larger than the threshold value (chosen to be equal to 0.05 rad, or $\approx 3°$), the angle controller is called to match the robot's YAW angle with the goal orientation. Otherwise, the position controller is called and the robot moves towards the set direction.

Once the position error is smaller than the threshold (chosen to be equal to 0.05 m, or 5 cm), the waypoint is considered to be reached and another goal is waited to be introduced. The two controllers are never called at the same time.

In Figure 3 and Figure 4, the linear velocity and acceleration of the robot are plotted for three different waypoints with three different distances: 0.2 $[m]$, 2 $[m]$ and 10 $[m]$. We can observe that the speed reaches the limit we set (chosen to be 0.5 $[\frac{m}{s}]$) for the two bigger distances but once the robot is getting closer to its goal, the speed decreases smoothly thanks to the PD controller. The initial acceleration peak value corresponds to a fast increase of the speed when the robot starts moving. Once the speed limit is reached, the acceleration goes to zero as the speed is kept constant and finally the acceleration becomes slightly negative as the speed is decreasing.

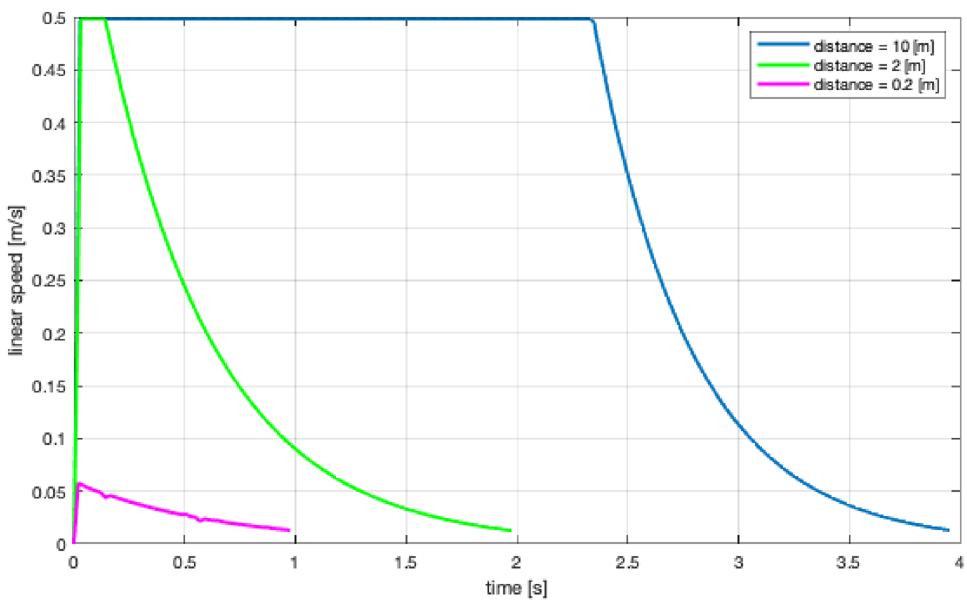


Figure 3: Linear velocity graphs for three different goal points at different distances

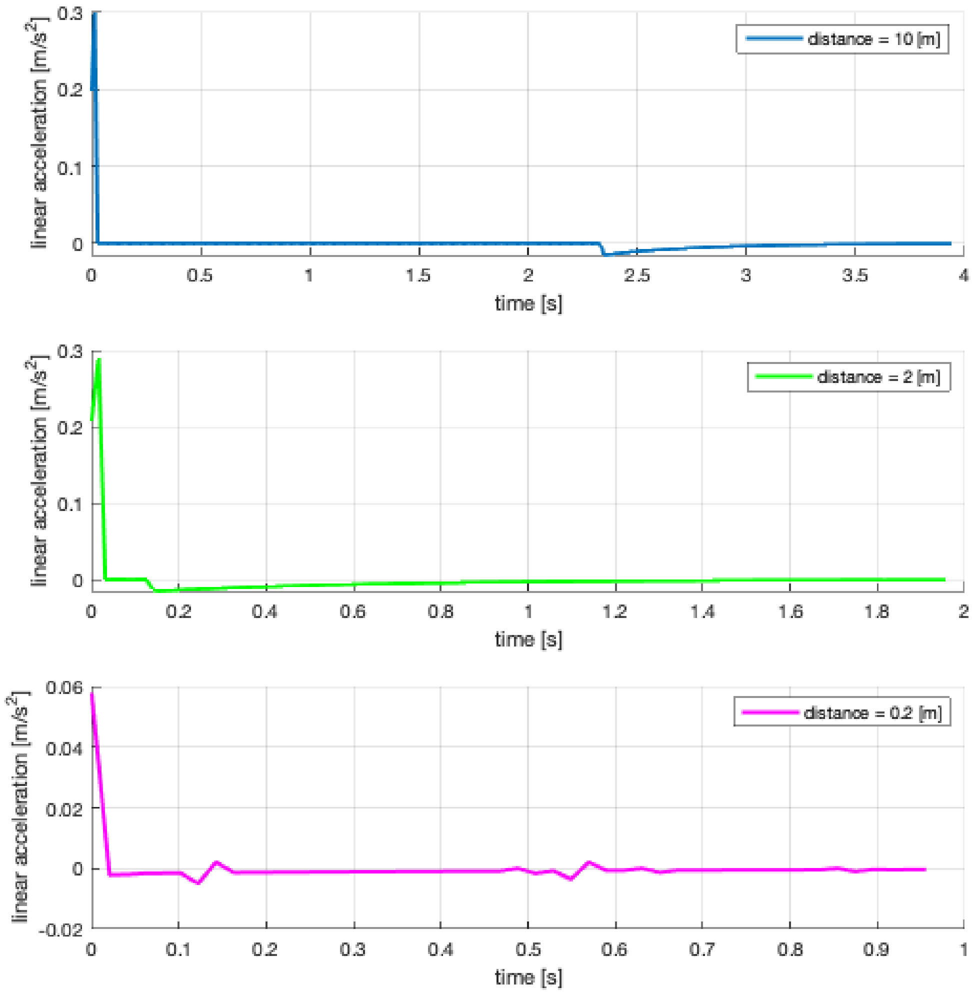


Figure 4: Linear acceleration graphs for three different goal points at different distances

4 Obstacle avoidance

We use a finite state machine to differentiate waypoint navigation and obstacle detection. Navigation is done in $STATE = 0$, and as previously mentioned, we check for obstacles at each time step. If an obstacle is detected we start a left or right wall following algorithm to avoid the obstacle. Left obstacle contouring (or right wall following) is done in $STATE = 1$, and right contouring (or left wall following) is done in $STATE = 2$. The state is chosen depending on which sensor is closer to the wall. Sensor values are read by subscribing to *laser_side/scan* topics. Figure 5 illustrates the robot's behaviour for $STATE = 2$. The avoidance is done in three parts:

1. Obstacle detected. Approach wall and turn right (or left for $STATE = 1$)

2. Left wall following (or right for $STATE = 1$)

3. If angle error is lower than zero (or greater than zero for $STATE = 1$) change to $STATE = 0$, or navigation state

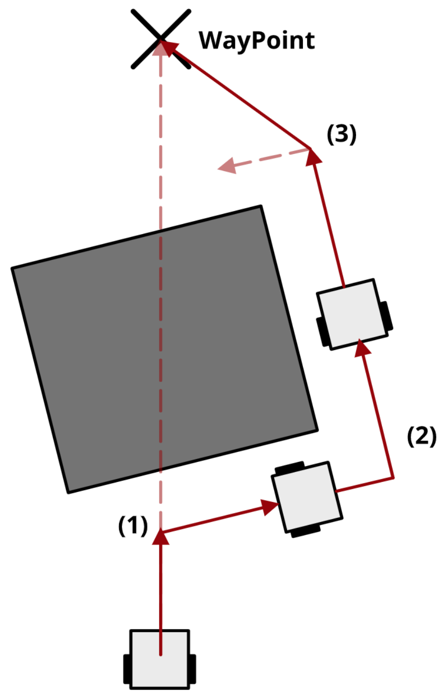


Figure 5: Obstacle avoidance schematic for $STATE = 2$ (right contouring)

This algorithm uses three sensors, one at the front, and one on each side. The one at the front is used to see if an obstacle is detected and then during obstacle avoidance to not crash into an obstacle ahead. The side sensors are used for wall following by keeping the robot at a certain distance from the wall by rotating it closer or away depending on its distance. If the side sensors do not detect anything, it is OK to turn in the wanted direction.

This algorithm can get the robot stuck in a loop, but we did not correct this because the aim of this project was to understand and implement ROS and not develop a better obstacle avoidance algorithm.

5 Discussion and Conclusion

In this practical, we have learnt how ROS works using topics to communicate between nodes. Subscriptions are used to "listen" to information and Publishers to "talk". Together they form a loosely coupled logic managed by the ROS Master, which registers the nodes. This communication allowed us to create a basic ROS model to control a two wheeled robot inspired by the Thymio.

We used the URDF (Unified Robotic Description Format) format to create a simple model, along with Gazebo's own plugins to create a differential drive and multiple sensors. By publishing and subscribing with ROS, along with the mentioned plugins, we were able to fully control and monitor the robot's behaviour. In parallel we were able to add some personal touches to our model by using different links and visual components found in URDF.

We created a Python code, controlling the robot with two separate controllers for angle and position to achieve robust waypoint navigation. We implemented a finite state machine to move in and out of an obstacle avoidance algorithm to allow for a fully autonomous system.

Despite having a bug in the Gazebo plugin, we were able to troubleshoot and create a system that goes around the problem, achieving the wanted goals.

6 Appendix A: Robot model code in URDF

```
<?xml version="1.0"?>

<robot xmlns:xacro="https://www.ros.org/wiki/xacro" name="THYMIO">

    <xacro:include filename="$(find ros_basics_2020)/urdf/macros.xacro" />
    <xacro:include filename="$(find ros_basics_2020)/urdf/materials.xacro" />


    <!-- Below you will find the rough specifications of the robot's mass and size -->
    <xacro:arg name="left_wheel_mu" default="100.0"/>
    <xacro:property name="left_wheel_mu_p" value="$(arg left_wheel_mu)"/>

    <xacro:arg name="right_wheel_mu" default="100.0"/>
    <xacro:property name="right_wheel_mu_p" value="$(arg right_wheel_mu)"/>

    <!-- radius of a single Thymio's wheel -->
    <xacro:property name="wheel_r" value="0.022"/>

    <!-- length of a single Thymio's wheel (if you think of the wheel as a cylinder,
      this would be the cylinder length) -->
    <xacro:property name="wheel_l" value="0.015"/>

    <!-- The total mass of the Thymio -->
    <xacro:arg name="mass" default="0.270"/>
    <xacro:property name="mass_p" value="$(arg mass)"/>

        <!-- Thymio's mass is distributed 20% at the wheels and 80% at the main body  -->
    <xacro:property name="body_mass" value="${mass_p * 0.80}"/>
    <xacro:property name="wheel_mass" value="${mass_p * 0.10}"/>


    <!-- Design your robot here using simple shapes (i.e., boxes, cylinders, spheres) -->

    <link name="main_body">
        <inertial>
            <mass value="${body_mass}"/>
```

```
            <!-- This is the actual size of the Thymio's main body -->
            <xacro:box_inertia m="${body_mass}" x="0.11" y="0.112" z="0.045" />

            <!-- Within this scope you need to define the inertial properties of the robot -->
            <!-- Take note that here we use the macro xacro:box_inertia defined int the macros.xacro -->
            <!-- If you are confident enough you can compute the inertia values on your own,
                 but be very careful to do it correctly -->
        </inertial>

        <!-- The collision shape is used for the physics-->
        <!-- In most cases (and especially for simple shapes) this should match
        the visual shape defined below. However, there are cases where it would
        make sense for those 2 to be different. If for example the visual shape
        is very complicated, then to accelerate the computation of the collision
        detector, we could image using a rough approximation of the visual shape. -->
        <collision name="collision_body">
            <origin xyz="0 0 ${0.045/2 + 0.01}"/>
            <geometry>
                <box size="0.11 0.112 0.045"/>
            </geometry>
            <material name="white"/>
        </collision>

        <!-- This is the shape that will be visualized by most of the simulators
        and visual tools. Some (like gazebo) might use the collision instead.
        Despite this being a "cosmetic" description, make sure it matches the
        above so that you can visually detect problems. -->
        <visual name="visual_body">
            <origin xyz="0 0 ${0.045/2 + 0.01}"/>
            <geometry>
                <box size="0.11 0.112 0.045"/>
            </geometry>
            <material name="white"/>
        </visual>
```

```

        <visual name="group_number">
            <origin xyz="0 -0.025 ${0.045 + 0.01}"/>
            <geometry>
                <box size="0.07 0.0075 0.005"/>
            </geometry>
            <material name="black"/>
        </visual>

        <visual name="group_number">
            <origin xyz="0 0.025 ${0.045 + 0.01}"/>
            <geometry>
                <box size="0.07 0.0075 0.005"/>
            </geometry>
            <material name="black"/>
        </visual>

        <visual name="group_number">
            <origin xyz="-0.0175 0 ${0.045 + 0.01}"/>
            <geometry>
                <box size="0.035 0.0075 0.005"/>
            </geometry>
            <material name="black"/>
        </visual>

        <visual name="group_number">
            <origin xyz="0 0.01 ${0.045 + 0.01}"/>
            <geometry>
                <box size="0.0075 0.0275 0.005"/>
            </geometry>
            <material name="black"/>
        </visual>

        <visual name="visual_sensor_front">
            <origin xyz="${0.11/2} 0 ${0.045/2 + 0.01}"/>
            <geometry>
                <box size="0.01 0.02 0.01"/>
            </geometry>
            <material name="black"/>
        </visual>

        <visual name="visual_sensor_left">
            <origin xyz="0.04 ${0.112/2} ${0.045/2 + 0.01}"/>
            <geometry>
                <box size="0.02 0.01 0.01"/>
            </geometry>
            <material name="black"/>
```

```
        </visual>

        <visual name="visual_sensor_right">
            <origin xyz="0.04 ${-0.112/2} ${0.045/2 + 0.01}"/>
            <geometry>
                <box size="0.02 0.01 0.01"/>
            </geometry>
            <material name="black"/>
        </visual>

        <collision name="collision_sphere">
            <origin xyz="0.035 0 0.01"/>
            <geometry>
                <sphere radius="0.01"/>
            </geometry>
            <material name="white"/>
        </collision>

        <visual name="visual_sphere">
            <origin xyz="0.03 0 0.01"/>
            <geometry>
                <sphere radius="0.01"/>
            </geometry>
            <material name="white"/>
        </visual>
    </link>

    <!-- Below you will find the joints and links for the wheels -->
    <joint name="left_wheel_joint" type="continuous">
        <parent link="main_body"/>
        <child link="left_wheel"/>
        <axis rpy="0 0 0" xyz="0 1 0"/>
        <origin xyz="-0.025 0.056 ${wheel_r}"/>
    </joint>

    <link name="left_wheel">
        <inertial>
            <mass value="${wheel_mass}"/>
            <xacro:cylinder_inertia m="${wheel_mass}" r="${wheel_r}"
                h="${wheel_l}"/>
        </inertial>

        <collision name="collision_wheel">
            <origin xyz="0 0 0" rpy="1.5708 0 0"/>
            <geometry>
                <cylinder radius="${wheel_r}" length="${wheel_l}"/>
            </geometry>
```

```
                <material name="black"/>
            </collision>

            <visual name="visual_wheel">
                <origin xyz="0 0 0" rpy="1.5708 0 0"/>
                <geometry>
                    <cylinder radius="${wheel_r}" length="${wheel_l}"/>
                </geometry>
                <material name="black"/>
            </visual>
        </link>

        <joint name="right_wheel_joint" type="continuous">
            <parent link="main_body"/>
            <child link="right_wheel"/>
            <axis rpy="0 0 0" xyz="0 1 0"/>
            <origin xyz="-0.025 -0.056 ${wheel_r}"/>
        </joint>

        <link name="right_wheel">
            <inertial>
                <mass value="${wheel_mass}"/>
                <xacro:cylinder_inertia m="${wheel_mass}" r="${wheel_r}"
                    h="${wheel_l}"/>
            </inertial>

            <collision name="collision_wheel">
                <origin xyz="0 0 0" rpy="1.5708 0 0"/>
                <geometry>
                    <cylinder radius="${wheel_r}" length="${wheel_l}"/>
                </geometry>
                <material name="black"/>
            </collision>

            <visual name="visual_wheel">
                <origin xyz="0 0 0" rpy="1.5708 0 0"/>
                <geometry>
                    <cylinder radius="${wheel_r}" length="${wheel_l}"/>
                </geometry>
                <material name="black"/>
            </visual>
        </link>

        <xacro:csensor plink="main_body" side="front" mass="0.001"
           originrpy="0 0 -1.5708" originxyz="-0.003 0 0" originjoint="$
           {0.11/2} 0 ${0.045/2 + 0.01}" />

```

```
    <xacro:csensor plink="main_body" side="left" mass="0.001" originrpy="0 0 -3.1416" originxyz="0 -0.003 0" originjoint="0.04 ${0.112/2} ${0.045/2 + 0.01}" />

    <xacro:csensor plink="main_body" side="right" mass="0.001" originrpy="0 0 3.1416" originxyz="0 0.003 0" originjoint="0.04 ${-0.112/2} ${0.045/2 + 0.01}" />

    <!-- Below you will find samples of gazebo plugins you may want to use. -->
    <!-- These should be adapted to your robot's design -->
    <gazebo reference="sensor_front">
        <sensor type="ray" name="laser_front">
            <pose>0 0 0 0 0 0</pose>
            <ray>
                <scan>
                    <horizontal>
                        <samples>13</samples>
                        <resolution>1</resolution>
                        <min_angle>-1.5</min_angle>
                        <max_angle>1.5</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <!-- You can edit adapt these to your robot's size -->
                    <min>0.001</min>
                    <max>0.2</max>
                    <resolution>0.0001</resolution>
                </range>
            </ray>
            <plugin name="laser" filename="libgazebo_ros_laser.so" >
                <topicName>laser_front/scan</topicName>
                <frameName>sensor_front</frameName>
            </plugin>
            <always_on>1</always_on>
            <update_rate>10</update_rate>
            <visualize>true</visualize>
        </sensor>
    </gazebo>

    <gazebo reference="sensor_left">
        <sensor type="ray" name="laser_left">
            <pose>0 0 0 0 0 0</pose>
            <ray>
                <scan>
                    <horizontal>
```

```
                        <samples>13</samples>
                        <resolution>1</resolution>
                        <min_angle>0.0708</min_angle>
                        <max_angle>3.0708</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <!-- You can edit adapt these to your robot's size -->
                    <min>0.001</min>
                    <max>0.2</max>
                    <resolution>0.0001</resolution>
                </range>
            </ray>
            <plugin name="laser" filename="libgazebo_ros_laser.so" >
                <topicName>laser_left/scan</topicName>
                <frameName>sensor_left</frameName>
            </plugin>
            <always_on>1</always_on>
            <update_rate>10</update_rate>
            <visualize>true</visualize>
        </sensor>
    </gazebo>

    <gazebo reference="sensor_right">
        <sensor type="ray" name="laser_right">
            <pose>0 0 0 0 0 0</pose>
            <ray>
                <scan>
                    <horizontal>
                        <samples>13</samples>
                        <resolution>1</resolution>
                        <min_angle>-3.0708</min_angle>
                        <max_angle>-0.0708</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <!-- You can edit adapt these to your robot's size -->
                    <min>0.001</min>
                    <max>0.2</max>
                    <resolution>0.0001</resolution>
                </range>
            </ray>
            <plugin name="laser" filename="libgazebo_ros_laser.so" >
                <topicName>laser_right/scan</topicName>
                <frameName>sensor_right</frameName>
```

```
            </plugin>
            <always_on>1</always_on>
            <update_rate>10</update_rate>
            <visualize>true</visualize>
        </sensor>
    </gazebo>

    <gazebo>
        <plugin name="differential_drive_controller" filename="
           libgazebo_ros_diff_drive.so">
            <alwaysOn>true</alwaysOn>
            <updateRate>20</updateRate>
            <leftJoint>left_wheel_joint</leftJoint>
            <rightJoint>right_wheel_joint</rightJoint>
            <wheelSeparation>0.11</wheelSeparation>
            <wheelDiameter>0.044</wheelDiameter>
            <!-- <wheelTorque>10</wheelTorque> -->

            <commandTopic>cmd_vel</commandTopic>
            <odometryTopic>odom</odometryTopic>
            <odometryFrame>odom</odometryFrame>

            <robotBaseFrame>main_body</robotBaseFrame>
        </plugin>
    </gazebo>

</robot>
```

7 Appendix B: Control algorithm code in Python

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

# ------------- Imports ---------------------

import rospy
import numpy as np
import time
import tf
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Point
from sensor_msgs.msg import LaserScan

# ------------- Callback Functions ----------

# Callback function to get pose
def callback_odom(data):
    # Global Pose Variables
    global POS_X
    global POS_Y
    global YAW

    # Position
    POS_X = data.pose.pose.position.x
    POS_Y = data.pose.pose.position.y

    # Orientation
    q = (data.pose.pose.orientation.x,
        data.pose.pose.orientation.y,
        data.pose.pose.orientation.z,
        data.pose.pose.orientation.w)
    e = tf.transformations.euler_from_quaternion(q)
    YAW = e[2]

# Callback function to get goal position
def callback_goal(data):
    # Global Waypoint list
    global WAYPOINTS
    WAYPOINTS.append(data)

# Callback functions to get sensor data
def callback_front_sensor(data):
    # Global Front Sensor array
    global FRONT
    FRONT = data.ranges
```

```
def callback_left_sensor(data):
    # Global Left Sensor array
    global LEFT
    LEFT = data.ranges

def callback_right_sensor(data):
    # Global Right Sensor array
    global RIGHT
    RIGHT = data.ranges

# ------------- Controller Functions ----------

def compute_errors():
    # Error computation for PD controller
    error_x = WAYPOINTS[0].x-POS_X
    error_y = WAYPOINTS[0].y-POS_Y
    yawl = YAW
    if (abs(yawl)) > 3:
        yawl = abs(yawl)
    error_angle = np.arctan2(error_y, error_x)-yawl
    if (abs(error_angle)) > 3:
        error_angle = 3.14 - yawl + error_angle%3.14
    return error_x, error_y, error_angle

def angle_controller(err_angle, old_err_angle, vel, pub):
    # Angle PD controller to orient robot correctly
    Kp = 1.2
    Kd = 0.2
    d_err_angle = err_angle - old_err_angle
    vel.angular.z = err_angle*Kp + d_err_angle*Kd
    pub.publish(vel)

def pos_controller(err_x, err_y, old_err_x, old_err_y, vel, pub):
    # Position PD controller to move robot
    Kp = 0.3
    Kd = 0.08
    d_err_x = err_x - old_err_x
    d_err_y = err_y - old_err_y
    speed = Kp*np.linalg.norm([err_x, err_y]) + Kd*np.linalg.norm([
        d_err_x, d_err_y])
    if (speed > 0.5):
        speed = 0.5
    vel.linear.x = speed
    pub.publish(vel)

def stop(vel, pub):
```

```python
    # Stops robot
    vel.linear.x = 0
    vel.angular.z = 0
    pub.publish(vel)

# ------------- Subscribe and Publish Functions ---------

# Subscribe to Topics (odom, goal_pos, sensor data)
def subscribers():
    rospy.init_node('controller', anonymous=True)
    rospy.Subscriber("odom", Odometry, callback_odom)
    rospy.Subscriber("goal_pos", Point, callback_goal)
    rospy.Subscriber("laser_front/scan", LaserScan, callback_front_sensor)
    rospy.Subscriber("laser_left/scan", LaserScan, callback_left_sensor)
    rospy.Subscriber("laser_right/scan", LaserScan, callback_right_sensor)

# Publish to Odometry Topic
def publishers():
    return rospy.Publisher("cmd_vel", Twist, queue_size=10)

# ------------- Waypoint Navigation Functions ----------

def navigation(err_x, err_y, err_angle, switch, vel, pub):
    global STATE

        # navigation state with waypoints
    if (len(WAYPOINTS) > 0) & (STATE == 0):
                # Prints
        if (switch == 1):
            rospy.loginfo("Waypoint X: %f", WAYPOINTS[0].x)
            rospy.loginfo("Waypoint Y: %f\n", WAYPOINTS[0].y)
            switch = 0

                # Errors for controllers and conditions
        old_err_x = err_x
        old_err_y = err_y
        old_err_angle = err_angle
        err_x, err_y, err_angle = compute_errors()

        if (abs(np.linalg.norm([err_x,err_y])) > abs(np.linalg.norm([old_err_x,old_err_y]))):
            stop(vel, pub)

```

```
        if (abs(err_angle)>0.05) & (abs(np.linalg.norm([err_x,err_y])) > 0.05):
                        # Angle Controller
            angle_controller(err_angle, old_err_angle, vel, pub)
        elif (abs(np.linalg.norm([err_x,err_y])) > 0.05):
                        # Position Controller
            pos_controller(err_x, err_y, old_err_x, old_err_y, vel, pub)
        else:
            stop(vel, pub)
            rospy.loginfo("Waypoint Reached\n")
            WAYPOINTS.pop(0)
            switch = 1

        # Check for obstacles
        obstacle_check(vel, pub, switch)
    else:
        if (switch == 0):
            rospy.loginfo("Waiting for Waypoint\n")
            switch = 1

    return err_x, err_y, err_angle, switch

# ------------- Obstacle Avoidance Functions -----------

def obstacle_check(vel, pub, switch):
    global STATE

    # Check front sensors and choose contour
    if (FRONT[7] < 0.2) | (FRONT[6] < 0.2):
        stop(vel, pub)
        switch = 0
        rospy.loginfo("Obstacle Detected\n")
        if(FRONT[6] > FRONT[7]):
            STATE=2
            rospy.loginfo("Right Contour Initiated\n")
        else:
            STATE=1
            rospy.loginfo("Left Contour Initiated\n")

def left_contour(vel, pub, switch):
    global STATE

    # Ideal wall distance and error
    dist = 0.13
    error = (RIGHT[7]+RIGHT[6])/2 - dist

```

```
    # Approach wall and turn right
    if (switch == 0):
        if (FRONT[7] > 0.13) | (FRONT[6] > 0.13):
            vel.linear.x = 0.1
            vel.angular.z = 0
            pub.publish(vel)
        else:
            vel.linear.x = 0
            vel.angular.z = 1
            pub.publish(vel)
            time.sleep(1.5)
            switch = 1
    else: # Wall following
        if (((error < -0.05) & (error > -0.08)) |
            (FRONT[7] < 0.12) | (FRONT[6] < 0.12) |
            ((FRONT[6] < 0.12) & (RIGHT[7] < 0.12))):
            vel.linear.x = 0
            vel.angular.z = 1
            pub.publish(vel)
        elif (((error > 0.05) & (error < 0.08)) |
            ((RIGHT[3] > 0.2) & (RIGHT[7] > 0.2))):
            vel.linear.x = 0
            vel.angular.z = -1
            pub.publish(vel)
        else:
            vel.linear.x = 0.1
            vel.angular.z = 0
            pub.publish(vel)

                # Check if obstacle is avoided
        _, _, err_angle = compute_errors()
        if (err_angle > 0):
            STATE = 0
            rospy.loginfo("Obstacle Avoided\n")

    return switch

def right_contour(vel, pub, switch):
    global STATE

    # Ideal wall distance and error
    dist = 0.13
    error = (LEFT[7]+LEFT[6])/2 - dist

    # Approach wall and turn left
    if (switch == 0):
        if (FRONT[7] > 0.13) | (FRONT[6] > 0.13):
```

```
                vel.linear.x = 0.1
                vel.angular.z = 0
                pub.publish(vel)
            else:
                vel.linear.x = 0
                vel.angular.z = -1
                pub.publish(vel)
                time.sleep(1.5)
                switch = 1
        else: # Wall following
            if (((error < -0.05) & (error > -0.08)) |
                (FRONT[7] < 0.12) | (FRONT[6] < 0.12) |
                ((FRONT[6] < 0.12) & (LEFT[6] < 0.12))):
                vel.linear.x = 0
                vel.angular.z = -1
                pub.publish(vel)
            elif (((error > 0.05) & (error < 0.08)) |
                ((LEFT[10] > 0.2) & (LEFT[6] > 0.2))):
                vel.linear.x = 0
                vel.angular.z = 1
                pub.publish(vel)
            else:
                vel.linear.x = 0.1
                vel.angular.z = 0
                pub.publish(vel)

                    # Check if obstacle is avoided
            _, _, err_angle = compute_errors()
            if (err_angle < 0):
                STATE = 0
                rospy.loginfo("Obstacle Avoided\n")

    return switch

# ------------- Algorithm -------------------

# Initializations
global STATE
WAYPOINTS = []
FRONT     = [0]*14
RIGHT     = []
LEFT      = []
POS_X     = 0
POS_Y     = 0
YAW       = 0
STATE     = 0
err_x     = 10
```

```
err_y      = 10
err_angle = 10
switch     = 0

# ROS functions
subscribers()
pub = publishers()
vel = Twist()
rate = rospy.Rate(10)

# Infinite loop
while not rospy.is_shutdown():
    if (STATE == 0):
        err_x, err_y, err_angle, switch = navigation(err_x, err_y,
            err_angle, switch, vel, pub)
    elif (STATE == 1):
        switch = left_contour(vel, pub, switch)
    elif (STATE == 2):
        switch = right_contour(vel, pub, switch)
    else:
        STATE = 0
    rate.sleep()
```

8 Appendix C: Instructions for launching examples

- Unzip Group14.zip
- Navigate to ../ros_practicals_ws on terminal
- Enter the command 'source devel/setup.sh'
- To launch Gazebo : 'roslaunch ros_basics_2020 robot_description_gazebo.launch'
- Open a new tab (2) and write 'source devel/setup.sh'
- To run the controller : 'rosrun ros_basics_2020 my_robot_control_plot.py' (make sure the Python file is executable using chmod)
- Open a new tab (3) and write 'source devel/setup.sh'
- Navigate to the source directory : 'cd /ros_practicals_ws/src/ros_basics_2020/src'
- Launch the waypoint publishing script : './publish.py' (make sure the Python file is executable using chmod)
- Observe the behavior in Tab 2 and in Gazebo. Waypoints can be modified in the publish.py script.
- You can for example add a cylinder of nominal diameter on position 1.5;1.5 (shown on Figure 6) and run a simulation using the publish.py and my_robot_control_plot.py scripts.

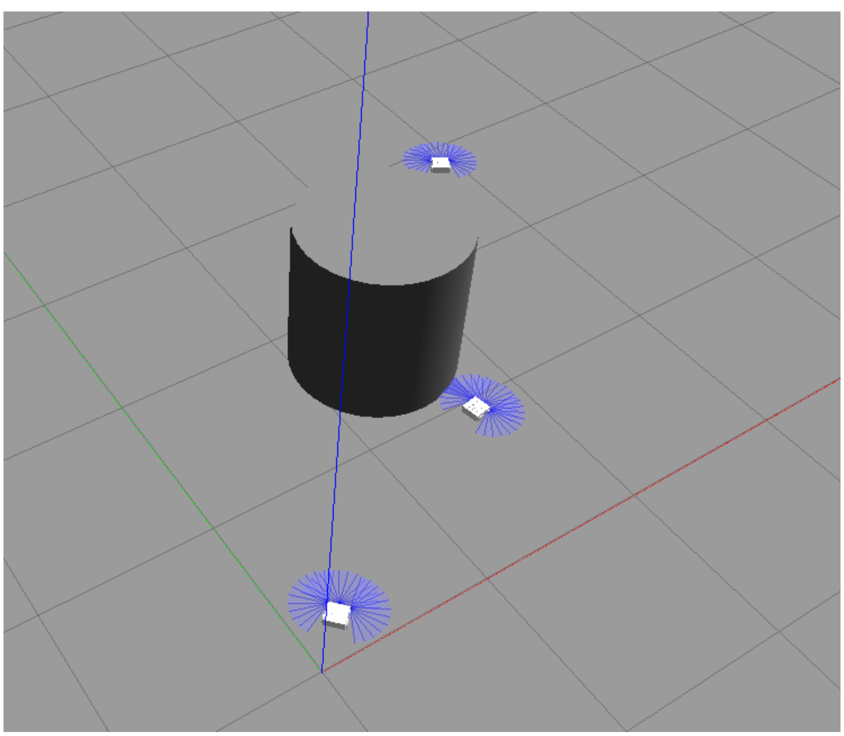


Figure 6: Obstacle avoidance and cylinder placement