Robotic Practicals
Topic 11: Robot Operating System basics

Group 20, Vuk Pajovic, Darko Lukic, Leonardo Cencetti

Spring Semester 2018/2019

1 Introduction

This report will present the results obtained during the two sessions of the Robotics Practicals TP 11. The content of the practical were the following: first, a basic introduction of Robot Operating System (ROS); second, the modeling of a simple approximation of the Thymio robot and the control architecture to do way-point navigation, and third, the implementation of a simple "vision-based" (IR sensors) obstacle avoidance algorithm. The following sections will analyze in more details the approaches taken to the problems and the results obtained.

2 Robot Model

The model of the robot was created using the measurements of the Thymio robot as reference, in order to have an accurate approximation of the kinematic and dynamic behaviour of the original robot. The model was then written in a XACRO file using XML to define the single component, their styling and relationships with the rest of the body. Several approximations were taken regarding the shape of the core blocks, which were approximated with parallelepipeds. Two versions of the Thymio model were created, one for the Velocity Control objective (lacking the distance sensors) and one for the Obstacle Avoidance (with the two additional distance sensors). The distance sensors were positioned on the frontal edges of the main body, at an angle of 0.5 radians, which allows the robot to sense obstacles on the full frontal semi-sphere. Sections 6.1 includes the XACRO code for the final model of the wheeled robot, used for Obstacle Avoidance, which is represented in Fig. 1.

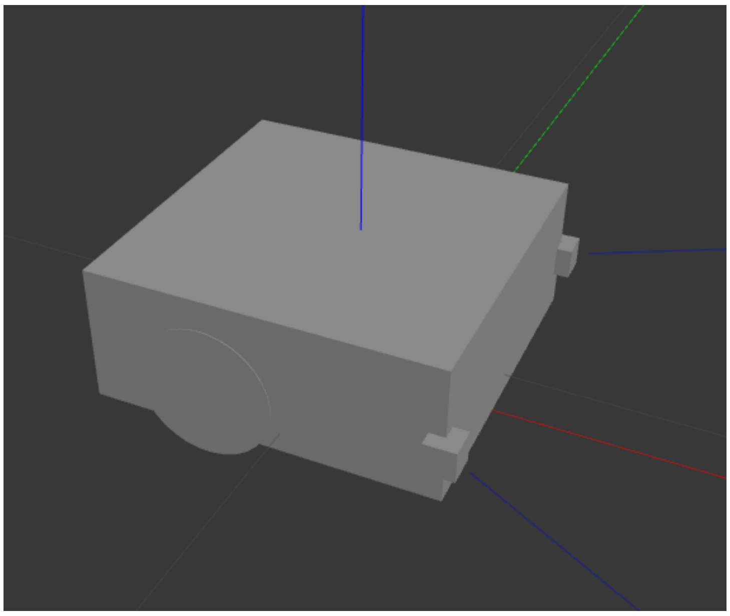

Figure 1: Final model of the Thymio robot with two distance sensors on the frontal corners

3 Velocity Control

In this section we will describe the control algorithm implemented to perform waypoint navigation. The code was written in Python and is reported in Section 6.2. The controller architecture relied on a three-state state machine, whose diagram is presented in Fig. 4. The diagram of the ROS network, with all the topic nodes and data streams, is reported in Fig. 6 The "ROTATION" state is triggered when the heading direction error (between the target and the robot heading direction) is greater than 0.02 radians (based on odometry). In this case, the robot stops and switches to a PD controller to correct the error ($k_p = 0.05, k_d = 0.01$). Particular attention had to be taken to account for the "Euler Singularity Problem", swapping the sign of the angles when crossing the sagittal plane. Once the angle error is under the threshold, the state machine is switched to the "FORWARD" state. For this state, the velocity is again set with a simple PD controller ($k_p = 0.05, k_d = 0.01$), using the position error with respect to the target (based on odometry) as input. To avoid weird behaviours (robot accelerating too fast and lifting the front), the acceleration is limited using a ramp function for the control command with $slope = 0.01$, represented in figure 2. The maximum velocity was also limited to 0.08 m/s to avoid reaching unrealistic speeds: the results of such limit can be observed in Fig. 3. While in this state, the state machine checks continuously the angle error and the position error, and switches to either the "ROTATION" or "STOP" states respectively if one of the two violates the boundary condition. Trivially, the "STOP" is triggered once the position error is below 0.05 meters, and sets the velocity of the motors to zero. In this state, the algorithms keeps checking for updates in the target position, and reverts back to "FORWARD" state if a different target is published.

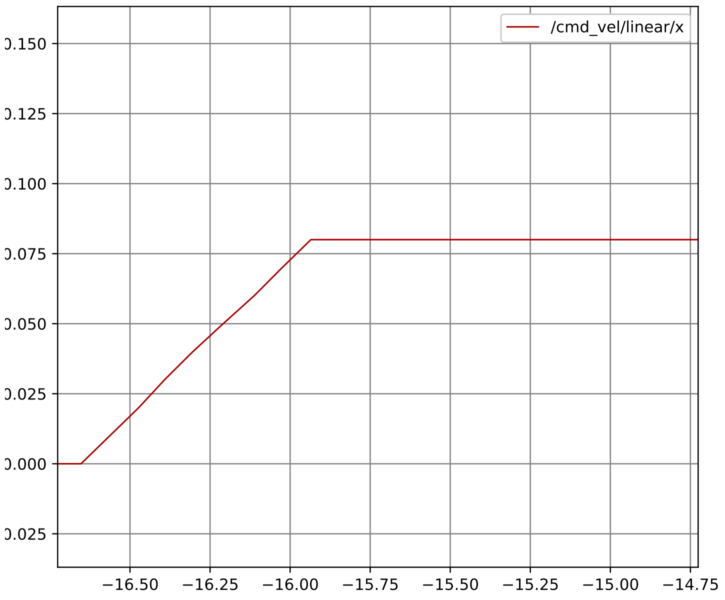

Figure 2: Ramp function used to limit the acceleration of the robot. The plot is acquired with *rqt_plot* during simulation in Gazebo.

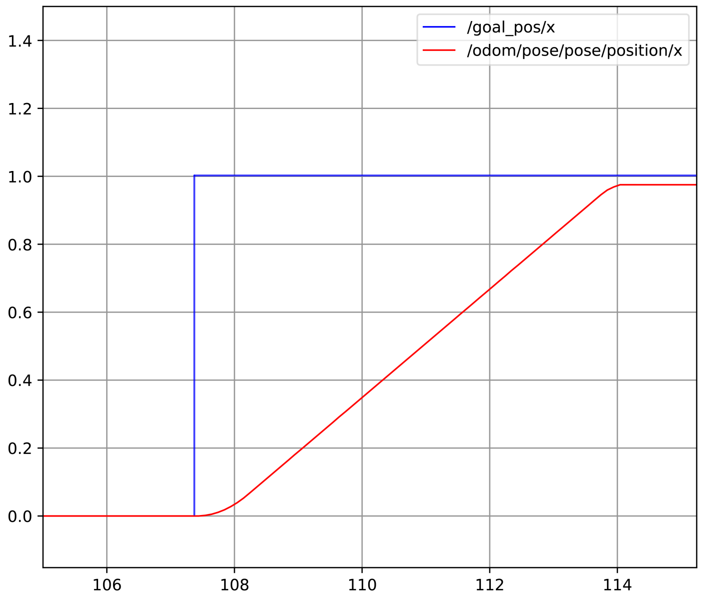

Figure 3: Robot position with respect to the target position over time. The results of the acceleration limit can be observed in the beginning and ending parts of the ramp as smoothed angles. The plot is acquired with *rqt_plot* during simulation in Gazebo.

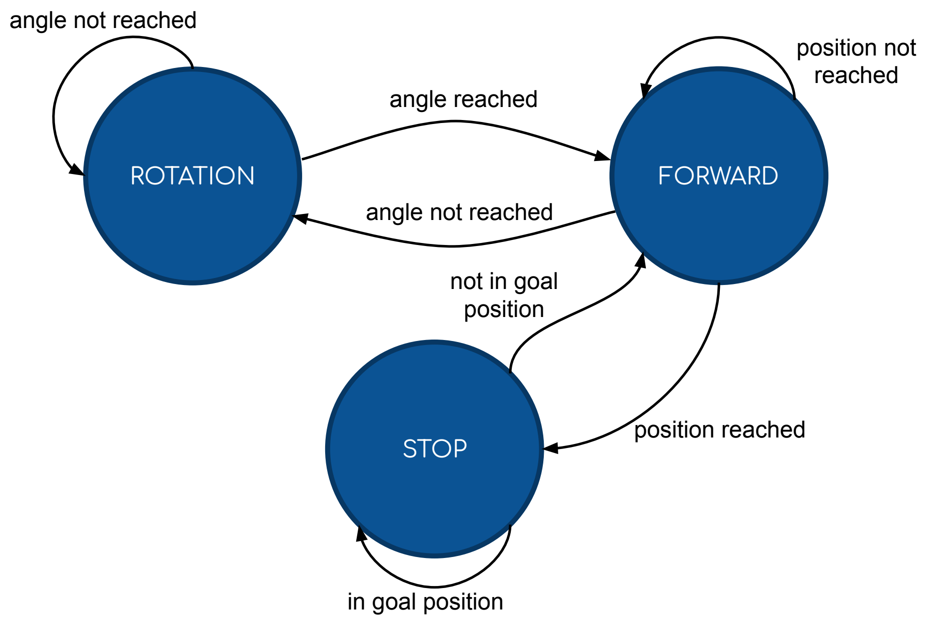

Figure 4: FSM for reaching the goal position

4 Local Obstacle Avoidance

4.1 Sensor modeling

Additional joints were added in .xacro file to create the left and right laser distance sensor, located on the left and right edges of the front face of the robot ans shown in Fig.5. The sensors are visualized as a small box while their functionality is implemented using the *libgazebo_ros_laser.so* plugin for Gazebo.

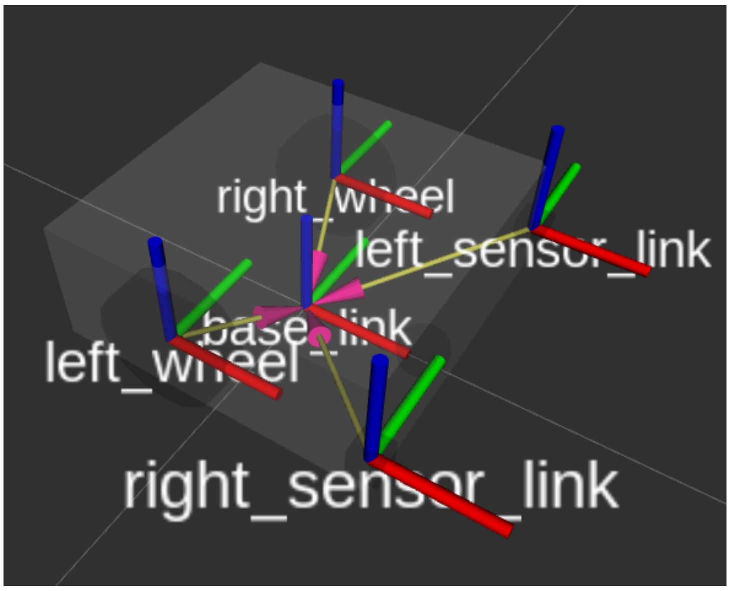

Figure 5: Model of the robot with the visible joints

This plugin allows a detailed configuration of the laser sensor parameters, e.g. scan range, scan resolution, angle range, angle resolution, noise configuration and a sample rate. In this case, for the sake of simplicity, a single laser ray for each sensor was used, and the range limited from a minimum of 1cm to a maximum of 2m.

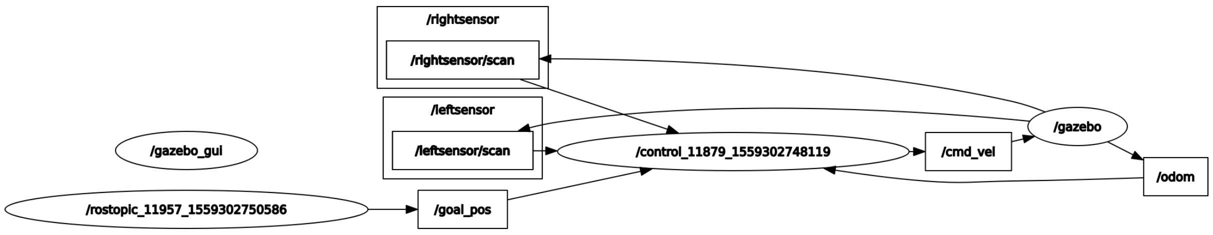

Figure 6: Nodes and topics in our system (visualized using *rqt_graph*)

In the empirical analysis of the performance of the local obstacle avoidance algorithm, it was identified that the angle of the laser ray is the most determinant factor and, after several trials, it was set to 0.5 radians. Using such angle, the algorithm is able to detect obstacles in front of the robot, as well as obstacles on the left and right sides. With this configuration the robot is able to precisely compute the correction angle and avoid the obstacle.

4.2 Obstacle avoidance algorithm

As stated before, the implemented obstacle avoidance algorithm is relatively simple. The main goal of this part of the TP was to test the gazebo plug-in, and to experiment with different designs modeled with Gazebo studio. For what regards the algorithm structure, the chosen strategy was to, first, read continuously the values from the two sensors. Then, if the measurement of one the sensors is less than the safety distance, the state machine is triggered into the state "AVOIDANCE". The used safe distance threshold was set to 0.5 meters, but lower values were also tested providing good values if the maximum speed was changed accordingly. Once in the state "AVOIDANCE", the vehicle starts turning away from the sensor which has the lowest value of the two. Once both sensors don't return values below the threshold, meaning that the way is clear, the state machine enters the state called "ESCAPE", moving forward for the next three seconds. If again at least one of the sensors detects an obstacle closer than the threshold, the state machine is returned to the "ESCAPE" state. Finally, if during the "ESCAPE" state no obstacle is detected (all distance measurements at maximum range), the state machine is restored to the "ROTATION" state, resuming waypoint navigation. The full state machine is described in the Fig. 7.

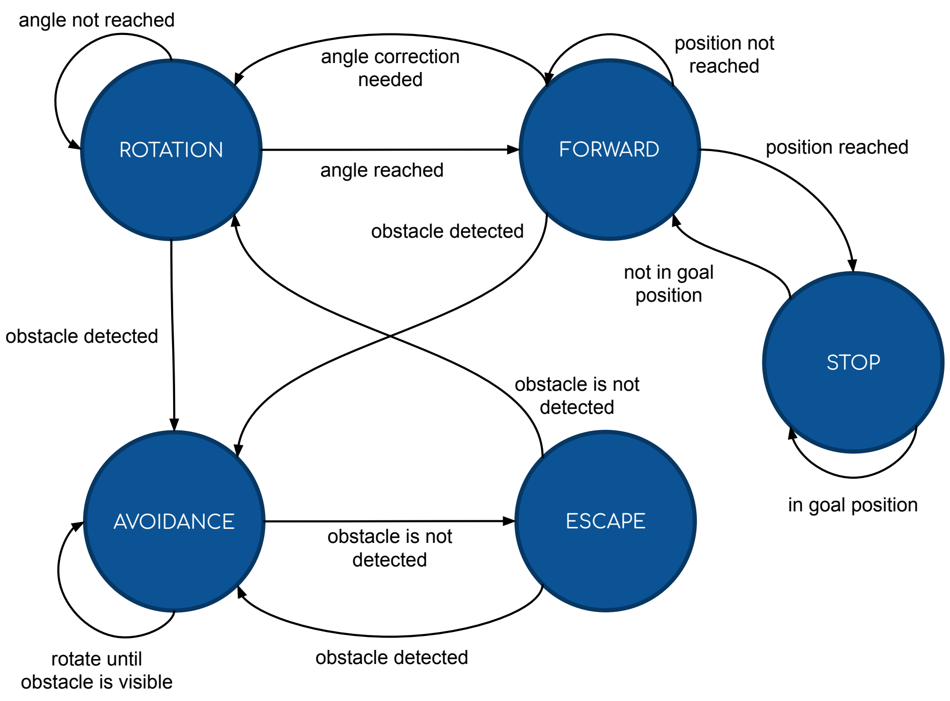

Figure 7: Obstacle Avoidance FSM

4.3 Side note

The implementation of the obstacle avoidance algorithm used for the second part is very simple. More robust (but more complex) solutions, that were not explored in detail as out of the scope of this TP, could have been the*bug*, *potential field*, and *vector field histogram* algorithms[1]. Few additional implementations of obstacle avoidance algorithms are available through the ROS community[2].

5 Discussion

This report has presented the results obtained during the two practical sessions, divided in two sections. Regarding the velocity control and waypoint navigation, the chosen approach proved to be more than effective, achieving a precision of 0.05m consistently, and even higher less reliably. Experiments with the P controller alone showed large overshoots in both position and heading control. With the addition of the D component the robot could reliably reach the designated target position. Since the results were accurate enough, the addition of the I component was not contemplated. Due to limitation in the controller-simulator interface, not allowing the direct control of single wheel speed, it was not possible to implement sophisticated trajectory-planning algorithms both for the first and second part. Regarding the Obstacle Avoidance section, the approach was again to implement a simple-but-reliable algorithm, augmenting the state machine implemented for the first part. The naive implementation of the algorithm performed well enough across the task it was designed for, and was tested against obstacles of different size and shape, always performing as supposed. The only possible limitation of this approach, which was not tested thoroughly, is the inability of avoiding concave obstacles, where just turning away from the sensor with the highest measurement is not optimal. Additionally, also here no complex trajectory planning was performed, for the same reasons as before.

[1]https://pdfs.semanticscholar.org/519e/790c8477cfb1d1a176e220f010d5ec5b1481.pdf
[2]http://wiki.ros.org/nav_core

6 Appendix

6.1 Appendix A

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="https://www.ros.org/wiki/xacro" name="SpesBot">
    <xacro:include filename="$(find ros_basics_2019)/urdf/macros.xacro" />
    <xacro:include filename="$(find ros_basics_2019)/urdf/materials.xacro" />
    <xacro:property name="width" value="0.11" />
    <xacro:property name="length" value="0.112" />
    <xacro:property name="height" value="0.045" />
    <xacro:property name="body_mass" value="${0.27 * 0.8}" />
    <xacro:property name="wheel_mass" value="${0.27 * 0.1}" />
    <xacro:property name="wheel_thickness" value="0.015" />
    <xacro:property name="wheel_radius" value="0.022" />
    <xacro:property name="sensor_size" value="0.01" />
    <!-- Design your robot here -->
    <!--
    <link name="base_link"></link>
    <joint name="robot_footprint_joint" type="fixed">
        <origin xyz="0 0 0" rpy="0 0 0" />
        <parent link="base_link"/>
        <child link="chassis" />
    </joint>
    -->
    <link name="base_link">
        <inertial>
            <mass value="${body_mass}"/>
            <xacro:box_inertia m="${body_mass}" x="${length}" y="${width}"
                                                 z="${height}" />
        </inertial>
        <collision name='collision'>
            <origin xyz="0 0 ${0.045 - 2 * 0.009}" />
            <geometry>
                <box size="${length} ${width} ${height}"/>
            </geometry>
        </collision>
        <visual name='chassis_visual'>
            <origin xyz="0 0 ${0.045 - 2 * 0.009}" />
            <geometry>
                <box size="${length} ${width} ${height}"/>
            </geometry>
            <material name="white"/>
        </visual>
        <!-- Cylinder -->
        <collision name='collision_cylinder'>
            <origin xyz="${length/2.5} 0 0.005" />
            <geometry>
                <sphere radius="0.009"/>
            </geometry>
        </collision>
        <visual name='cylinder'>
            <origin xyz="${length/2.5} 0 0.005" />
            <geometry>
                <sphere radius="0.009"/>
            </geometry>
            <material name="red"/>
        </visual>
    </link>
    <joint name="footprint_left_sensor" type="fixed">
        <origin xyz="${length/2} ${width/2} ${height/2}" />
        <parent link="base_link"/>
        <child link="left_sensor_link" />
        <axis xyz="0.0 1 0.0"/>
    </joint>
    <link name="left_sensor_link">
        <visual name="left_sensor_visual">
            <origin xyz="0 0 0" />
            <geometry>
                <box size="${sensor_size} ${sensor_size} ${sensor_size}"/>
            </geometry>
            <material name="red"/>
        </visual>
        <collision name='left_sensor_collision'>
            <origin xyz="0 0 0" />
            <geometry>
                <box size="${sensor_size} ${sensor_size} ${sensor_size}"/>
            </geometry>
        </collision>
    </link>
    <joint name="footprint_right_sensor" type="fixed">
        <origin xyz="${length/2} ${-width/2} ${height/2}" />
        <parent link="base_link"/>
        <child link="right_sensor_link" />
        <axis xyz="0.0 1 0.0"/>
    </joint>
    <link name="right_sensor_link">
        <visual name="right_sensor_visual">
            <origin xyz="0 0 0" />
            <geometry>
                <box size="${sensor_size} ${sensor_size} ${sensor_size}"/>
            </geometry>
            <material name="red"/>
        </visual>
    </link>
    <link name="left_wheel">
        <inertial>
            <mass value="${wheel_mass}"/>
            <origin xyz="0 0 0" rpy=" 0 ${pi/2} ${pi/2}"/>
            <xacro:box_inertia m="${body_mass}" x="${length}" y="${width}"
                                                 z="${height}" />
        </inertial>
        <collision name="left_wheel_collision">
            <origin xyz="0 0 0" rpy=" 0 ${pi/2} ${pi/2}"/>
            <geometry>
                <cylinder length="${wheel_thickness}" radius="${wheel_radius}"/>
            </geometry>
        </collision>
        <visual name="left_wheel_visual">
            <origin xyz="0 0 0" rpy=" 0 ${pi/2} ${pi/2}"/>
            <geometry>
                <cylinder length="${wheel_thickness}" radius="${wheel_radius}"/>
            </geometry>
            <material name="black" />
        </visual>
    </link>
    <link name="right_wheel">
        <inertial>
            <mass value="${wheel_mass}"/>
            <origin xyz="0 0 0" rpy=" 0 ${pi/2} ${pi/2}"/>
            <xacro:box_inertia m="${body_mass}" x="${length}" y="${width}"
                                                z="${height}" />
        </inertial>
        <collision name="left_wheel_collision">
            <origin xyz="0 0 0" rpy=" 0 ${pi/2} ${pi/2}"/>
            <geometry>
                <cylinder length="${wheel_thickness}" radius="${wheel_radius}"/>
            </geometry>
        </collision>
        <visual name="left_wheel_visual">
            <origin xyz="0 0 0" rpy=" 0 ${pi/2} ${pi/2}"/>
            <geometry>
                <cylinder length="${wheel_thickness}" radius="${wheel_radius}"/>
            </geometry>
            <material name="black" />
        </visual>
    </link>
    <joint name="footprint_left_wheel" type="continuous">
        <origin xyz="-${width/6} ${wheel_thickness/2.1-width/2} ${height/2-0.005}"
                                                                rpy="0 0 0" />
        <parent link="base_link"/>
        <child link="left_wheel" />
        <axis xyz="0.0 1 0.0"/>
    </joint>
    <joint name="footprint_right_wheel" type="continuous">
        <origin xyz="-${width/6} ${-wheel_thickness/2.1+width/2} ${height/2-0.005}"
                                                                rpy="0 0 0" />
        <parent link="base_link"/>
        <child link="right_wheel" />
        <axis xyz="0.0 1 0.0"/>
    </joint>
    <gazebo>
        <plugin name="differential_drive_controller"
                                    filename="libgazebo_ros_diff_drive.so">
            <alwaysOn>true</alwaysOn>
            <updateRate>10</updateRate>
            <leftJoint>footprint_left_wheel</leftJoint>
            <rightJoint>footprint_right_wheel</rightJoint>
            <wheelSeparation>${width - wheel_thickness}</wheelSeparation>
            <wheelDiameter>${wheel_radius}</wheelDiameter>
            <commandTopic>cmd_vel</commandTopic>
            <odometryTopic>odom</odometryTopic>
            <odometryFrame>odom</odometryFrame>
            <robotBaseFrame>base_link</robotBaseFrame>
        </plugin>
    </gazebo>
    <gazebo reference="left_sensor_link">
        <sensor type="ray" name="left_sensor">
            <pose>0 0 0 0 0 0</pose>
            <ray>
                <scan>
                    <horizontal>
                        <samples>1</samples>
                        <resolution>1</resolution>
                        <min_angle>0.5</min_angle>
                        <max_angle>0.5</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <min>0.01</min>
                    <max>2.0</max>
                    <resolution>0.01</resolution>
                </range>
                <noise>
                    <type>gaussian</type>
                    <mean>0.0</mean>
                    <stddev>0.01</stddev>
                </noise>
            </ray>
            <plugin name="laser" filename="libgazebo_ros_laser.so">
                <topicName>leftsensor/scan</topicName>
                <frameName>left_sensor_link</frameName>
            </plugin>
            <always_on>1</always_on>
            <update_rate>10</update_rate>
            <visualize>true</visualize>
        </sensor>
    </gazebo>
    <gazebo reference="right_sensor_link">
        <sensor type="ray" name="right_sensor">
            <pose>0 0 0 0 0 0</pose>
            <ray>
                <scan>
                    <horizontal>
                        <samples>1</samples>
                        <resolution>1</resolution>
                        <min_angle>-0.5</min_angle>
                        <max_angle>-0.5</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <min>0.01</min>
                    <max>2.0</max>
                    <resolution>0.01</resolution>
                </range>
                <noise>
                    <type>gaussian</type>
                    <mean>0.0</mean>
                    <stddev>0.01</stddev>
                </noise>
            </ray>
            <plugin name="laser" filename="libgazebo_ros_laser.so">
                <topicName>rightsensor/scan</topicName>
                <frameName>right_sensor_link</frameName>
            </plugin>
            <always_on>1</always_on>
            <update_rate>10</update_rate>
            <visualize>true</visualize>
        </sensor>
    </gazebo>
</robot>
```

6.2 Appendix B

```
#!/usr/bin/python

"""
£ rosrun ros_basics_2019 control.py

£ rostopic pub /cmd_vel geometry_msgs/Twist "linear:
  x: 0.1
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
"""


import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Point, Twist
from nav_msgs.msg import Odometry
from sensor_msgs.msg import LaserScan
from time import sleep
import signal
import sys
import tf
import numpy as np


ANGLE_P = 1
ANGLE_D = 0.1
POS_P = 1
POS_D = 0.1
ACC_SLOPE = 0.01
ROTATION_EPS = 0.02
ROTATION_TRANS_EPS = 0.1
DISTANCE_EPS = 0.05
SAFE_REGION = 0.1
TOP_SPEED = 0.08
```

```python
STATE_ANGLE = 0
STATE_FORWARD = 1
STATE_STOP = 2
STATE_AVOIDANCE = 3
STATE_ESCAPE = 4

odom = Odometry()
goal = Point()
state = STATE_FORWARD
range_left = np.inf
range_right = np.inf

pub_scan = rospy.Publisher('scan', LaserScan, queue_size=1)

def signal_handler(sig, frame):
    sys.exit(0)

def goal_pos_cb(pose):
    """
    £ rostopic pub /goal_pos geometry_msgs/Point "x: 5
y: 0
z: 0"
    """
    global goal
    global state
    goal = pose
    state = STATE_ANGLE
    print(f'Goal received: {goal}')
    rospy.loginfo(pose)

def odom_cb(pose):
    global odom
    odom = pose
    """
    £ rostopic pub /odom nav_msgs/Odometry "???
    """
    # rospy.loginfo(pose)

def laser_left_cb(laser):
    pub_scan.publish(laser)
    global range_left
    range_left = laser.ranges[0]

def laser_right_cb(laser):
    global range_right
    range_right = laser.ranges[0]

def get_euler(odom):
    quaternion = (
        odom.pose.pose.orientation.x,
        odom.pose.pose.orientation.y,
        odom.pose.pose.orientation.z,
        odom.pose.pose.orientation.w
    )
    return tf.transformations.euler_from_quaternion(quaternion)
```

```python
def get_pose(odom):
    return np.array([
        odom.pose.pose.position.x,
        odom.pose.pose.position.y,
        odom.pose.pose.position.z
    ])


def print_status(target_angle, target_pose, current_pose, euler):
    if False:
        return

    print(f'State {state}')
    print(f'[Angle] Target: {np.round(target_angle, 2)}, current {np.round(euler[2], 2)}, diff {np.r
    print(f'[Position] Target: {np.round(target_pose, 2)}, current {np.round(current_pose, 2)}, diff


def listener():
    global state

    prev_pos_err = 0
    prev_angle_err = 0
    prev_speed = 0
    avoiding_counter = 0

    pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)

    rospy.init_node('control', anonymous=True)
    rospy.Subscriber('goal_pos', Point, goal_pos_cb)
    rospy.Subscriber('odom', Odometry, odom_cb)
    rospy.Subscriber('leftsensor/scan', LaserScan, laser_left_cb)
    rospy.Subscriber('rightsensor/scan', LaserScan, laser_right_cb)



    while True:
        target_pos = np.array([goal.x, goal.y, goal.z])
        target = Twist()

        euler = get_euler(odom)
        current_pose = get_pose(odom)

        diff = target_pos - current_pose
        target_angle = np.arctan2(diff[1], diff[0])

        # Calculate angle error
        angle_err = (euler[2] - target_angle)
        if angle_err > np.pi:
            angle_err -= 2 * np.pi
        elif angle_err < -np.pi:
            angle_err += 2 * np.pi

        if state == STATE_ANGLE:
            # PD controller
            target.angular.z = angle_err * ANGLE_P
            target.angular.z += (angle_err - prev_angle_err) * ANGLE_D
            prev_angle_err = angle_err

            # Change state
            if range_right < SAFE_REGION or range_left < SAFE_REGION:
                target.linear.x = 0
                state = STATE_AVOIDANCE
            elif abs(target_angle - euler[2]) > ROTATION_EPS:
                state = STATE_ANGLE
            elif np.linalg.norm(diff) > DISTANCE_EPS:
                state = STATE_FORWARD

        elif state == STATE_FORWARD:
            pos_err = np.linalg.norm(target_pos - current_pose)

            # PD controller
            target.angular.z = 0
            target.linear.x = pos_err * POS_P
            target.linear.x += (pos_err - prev_pos_err) * POS_D
            prev_pos_err = pos_err

            # Slope (throttle the acceleration)
            if target.linear.x - prev_speed > ACC_SLOPE:
                target.linear.x = prev_speed + ACC_SLOPE
            prev_speed = target.linear.x

            if target.linear.x > TOP_SPEED:
                target.linear.x = TOP_SPEED

            # Change state
            if range_right < SAFE_REGION or range_left < SAFE_REGION:
                target.linear.x = 0
                state = STATE_AVOIDANCE
            elif np.linalg.norm(diff) < DISTANCE_EPS:
                prev_speed = 0
                state = STATE_STOP
            elif abs(target_angle - euler[2]) > ROTATION_TRANS_EPS:
                prev_speed = 0
                state = STATE_ANGLE

        elif state == STATE_STOP:
            # Turn off the motors
            target.linear.x = 0
            target.angular.z = 0

            # Change state
            if np.linalg.norm(diff) > 0.1:
                state = STATE_FORWARD

        elif state == STATE_AVOIDANCE:
            target.angular.z = 0.2 if range_right > range_left else -0.2

            # Change state
            if range_right > SAFE_REGION * 2 and range_left > SAFE_REGION * 2:
                avoiding_counter = 0
                prev_speed = 0
                state = STATE_ESCAPE

        elif state == STATE_ESCAPE:
            target.linear.x = 0.1
            target.angular.z = 0

            # Slope (throttle the acceleration)
            if target.linear.x - prev_speed > ACC_SLOPE:
                target.linear.x = prev_speed + ACC_SLOPE
            prev_speed = target.linear.x

            avoiding_counter += 1

            # Change state
            if range_right < SAFE_REGION * 2 or range_left < SAFE_REGION * 2:
                target.linear.x = 0
                state = STATE_AVOIDANCE
                avoiding_counter = 0
            elif avoiding_counter > 30:
                state = STATE_ANGLE


        # print_status(target_angle, target_pos, current_pose, euler)
        pub.publish(target)
        sleep(0.1)

    rospy.spin()

if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler)

    listener()
```

6.3 Appendix C

This appendix is not needed as ROS Melodic was used.

6.4 Full Source Code

https://gitlab.com/lukicdarkoo/ros-differential-drive