ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

ROS BASICS REPORT

Sim2Real Development for Thymio with ROS

CHUANFANG NING (SCIPER: 320662)

JIANHAO ZHENG (SCIPER: 323146)

YUJIE HE (SCIPER: 321657)

14th April 2021

Contents

1 Introduction 2

2 Thymio Model Design 3
- 2.1 Building a Virtual Thymio Robot Model with URDF . . . 3
- 2.2 Sensor Integration . . . 4
- 2.3 Differential Drive Integration . . . 5
- 2.4 Texturing . . . 5

3 Developing Autonomous Navigation Algorithms for Thymio 6
- 3.1 OOP Programming Paradigm . . . 6
- 3.2 PID Control . . . 7
- 3.3 Waypoint Following . . . 8
- 3.4 Obstacle Avoidance . . . 9

4 Experiments & Discussion 12
- 4.1 Implementation Details . . . 12
- 4.2 Robot Performance Evaluation & Analysis . . . 12
- 4.3 Discussion . . . 13
  - 4.3.1 Sim2Real Challenges . . . 13
  - 4.3.2 Ways of Improving . . . 15

5 Conclusion 16

6 Appendix 17
- 6.1 Instructions for Launching Examples . . . 17
- 6.2 Development on different ROS distributions . . . 19
- 6.3 Setting up Dynamic Reconfigure for Parameters Tuning . . . 20

1 Introduction

During the second semester of the Robotics Practicles module, we are tasked to create an artifact of a mobile robot with ROS. The project is supervised by Vaios Papaspyros and Rafael Barmak from MOBOTS group at EPFL. We conduct the simulation in Gazebo during the first weeks. After that, We transfer our code to a physical robot, working on a real Thymio interacting with its surroundings on the third session.

The goal of this practical is to get an understanding of the Robot Operating System (ROS) by carrying out a project with a Thymio robot. In this project, we will be building up a robot model with URDF, utilize ROS communication functionalities, and implementing a trajectory following algorithm with obstacle avoidance using the data acquired. Throughout the course, we also explore the development paradigms and good practices adopted by the ROS community.

The Thymio robot will be placed in a limited environment with obstacles randomly, and it should start to move autonomously after setting waypoints. The robot should keep moving towards the next goal while avoiding obstacles until the waypoint list is empty.

The remainder of this report is structured as follows: Section 2 covers the virtual model design of the Thymio robot, including the addition of main mechanical characteristics, the integration of sensors and differential drives, and texturing. Section 3 introduces the details of the proposed MKCT tracker. Section 4 shows implementation details, experiment results, and discussion on the Sim2Real challenges and ways of improving. Section 5 summarizes the overall implementation, simulation, and real-world tests. As supplementary materials, Section 6 includes instructions for launching files, development on different ROS distributions, and other implementation details.

2 Thymio Model Design

The first step of ROS practical is to build a Thymio robot model to propose and verify algorithms in the simulation environment. All our functions like trajectory following and obstacle avoidance will be tested on this model in simulation first. This virtual model should represent the mechanical structure and sensor deployment corresponding to its real-world counterpart.

2.1 Building a Virtual Thymio Robot Model with URDF

Unified Robot Description Format (URDF) is a popular XML format in ROS for representing different elements in a robot model. There are different ways to get models in URDF format. It can be preferable for more complex models to export URDF directly from CAD applications like SolidWorks. In this pratical, we focus on some basic ideas in ROS, and the robot model is created from scratch using a xacro file with a simplified collada model[1].

The rigid body parameters of the robot is represented with links and joints in a URDF. Links are basic shapes that make up the robot. For each link, we have to define its visual shape, collision shape, and transformations. We have used both basic primitives and detailed collada model (i.e., *.dae file) as links in our models. Two cylinders are used to represent the simplified model for the motorized wheels at the back. In contrast, a sphere is used to replace a cylinder-shape for the caster wheel to realize better turning and realistic representation similar to the real Thymio.

For the main body, the collada model has three advantages rather than using a simple box primitive. Firstly, a better preview can be offered in the simulator to indicate where the robot is heading. Secondly, the proximity sensors on Thymio can be distributed on an arc instead of a line to be easier to arrange these sensors on a model with a smoother contour. Finally, the more realistic model can avoid unrealistic collisions when Thymio navigates near the wall in the simulated experimental setup, narrowing the gap between the simulation and the real test. As shown in Fig. 1 and Fig. 2, the main body model was firstly drafted in Inventor by intersecting a circle and rectangle. Then it was extruded by the height of the robot, scaled to meter, and exported as a collada file, i.e., `Thymio.dae`. In the urdf xacro file, both the link's visual and collision elements refer to this file directly.

After finishing the wheels and body of the Thymio robot, we need to define their relative position by adding joints to the URDF. Joint elements can refer to flexible and inflexible joints, both of which describe the relative relationship between different links. We use a continuous link between wheels and body so that the wheels can rotate in the simulation. As observed, putting the caster wheel inside the main body link will lead the robot to touch the ground with the wheel directly, which resulted in more friction during robot motion, and the robot would bounce in high-speed situations(>0.2m/s). To avoid this unwarranted phenomenon, we defined

[1] https://wiki.ros.org/urdf#Components

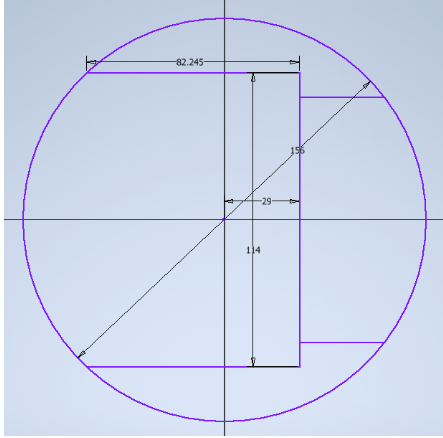


FIGURE 1: Scratch of main body

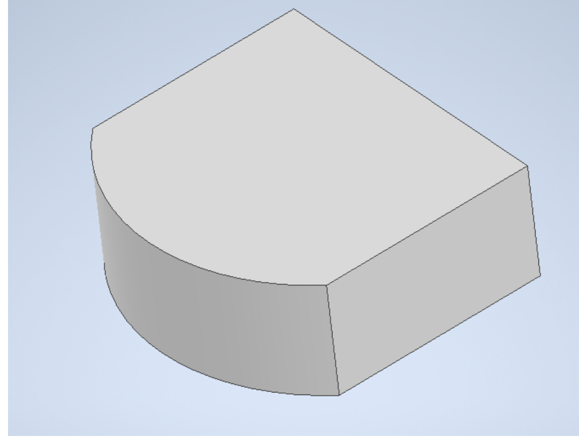


FIGURE 2: Preview of main body

the caster wheel as a separate link and connected it with the main body using a continuous joint.

As a result, Fig. 3 visualizes the topography between different link parts of Thymio.

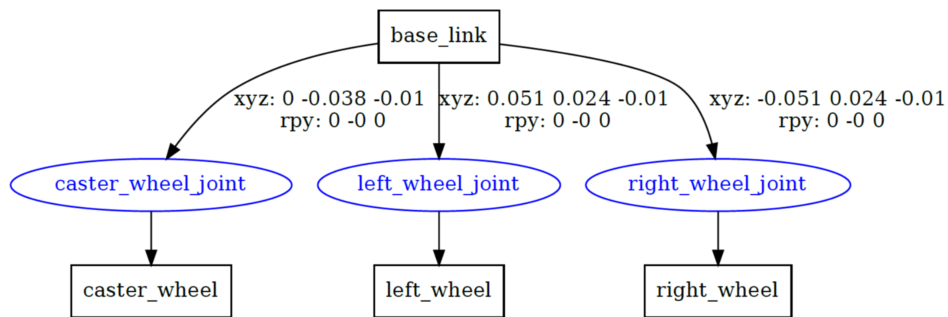


FIGURE 3: Visualisation of robot urdf structure

2.2 Sensor Integration

After constructing the robot's main body, the proximity sensors are added for receiving information of the environment with the LaserScan message. The integration of sensors includes adding sensor models to the main body and configuring sensor parameters.

Since the Thymio robot has all horizontal proximity sensors sharing the same geometry, we use xacro macro to reduce the redundancy of code to define similar shapes. Most of the geometric properties of proximity sensors were already defined in a macro `custom_sensors`. We adjust the transformation of link and joint elements in the input parameters for required sensors to integrate into the robot model in a few lines.

After configuring the sensors' physical property, their corresponding parameter can be defined in the Gazebo configuration file. The laser sensor available in the Gazebo simulator is employed to represent the proximity sensors on Thymio. From the practical experience last semester, the proximity sensor of Thymio has a very close detection range and a very narrow

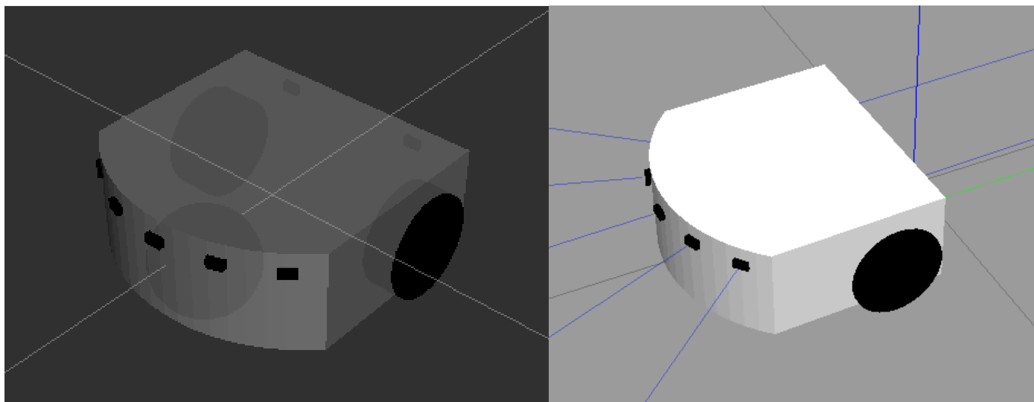
field of view (FOV). Therefore, the detection range is set to 10 cm with 1 mm resolution, and the FOV is limited to a single line like a laser to simulate the least sensitive sensor. It is assumed that if our proposed algorithms could work with this sensor in simulation, the Thymio robot would have more chance to behave correctly with real sensors in the on-site test.

2.3 Differential Drive Integration

After achieving a simple robot model with sensors, the differential drive is implemented to generate different linear and angular speeds on the Thymio robot. The Gazebo simulator has a basic plugin that can implement differential drives given the geometry of the wheels and odometry of the robot. Again we added this plugin in the Gazebo configuration file by passing the wheel link and wheel parameters we defined in the previous part to the arguments. This plugin could control both wheels automatically as soon as it received new commands from the command topic. The command topic cmd_vel accepts messages of type geometry_msg/Twist. We first tested the robot model's functionality by publishing linear and angular speed manually to the topic. In the implementation of obstacle avoidance and trajectory following, this task would be accomplished by a publisher to advertise speed calculated by our controller.

2.4 Texturing

The last step in the robot design part is applying materials to provide intuitive visualization, which can be defined in two ways. The first way is used for simple visualization of the robot in RVIZ. We can define the material element by using the visual tag of urdf file, where we can use pre-defined material directly by referring to `ros_basics_models`. The second way is used for visualization in Gazebo. We can use existing materials in Gazebo material library by assigning the respective tag to the link in Gazebo separately. Fig. 4 shows the textured robot model in RVIZ and Gazebo. As observed, when spawning the model in a pre-configured environment, we should adjust the position to avoid the collision carefully so that the robot would not perform out of control when it is inside another collision object.



(A) Preview in RVIZ (B) Preview in Gazebo

FIGURE 4: Visualizing textured Thymio robot preview

3 Developing Autonomous Navigation Algorithms for Thymio

3.1 OOP Programming Paradigm

The topic subscriber/publisher and server/client are the must-have for most ROS packages. One common solution is to define subscribers and publishers within the main function, where each subscriber/server will have a corresponding callback function separately. However, this will bring maintainability and reliability issues when subscribing topics and services grow and variables between different threads sharing in different scope. For example, if we want to use the robot pose (`x`, `y`, and `yaw`) in the main control functions of the program, we have to declare them in the global scope, which is far from optimal.

To make the project structure more readable and easy for development, we have utilized object-oriented programming (OOP) as our developing paradigm. In the following part, `thymio_control_pnode_simu.py` is taken as an example.

The `__init__()` function of is the constructor of the proposed class. We initialize the parameters and ROS setup, which encapsulate three main parts as follows:

1. Robot state and PID control, which includes the robot pose, error variables, max limit, and PID parameters
2. Sensing and obstacle avoidance, including proximity sensor variables, obstacle avoidance parameters, and threshold constants.
3. ROS setup, which initializes the node, subscriber, publisher, client, server of each type of required information, and their corresponding message objects.

In this way, we can use the attributes in all the member methods and variables after creating an instance of the class. Especially when designing high-level control functions such as `run_with_obstacle_avoidance(self)`, we can implement it without hacking the information from other nodes with global variables. For example, the related code snippets for the subscription to topic `robot_pose` can be implemented as follows:

```
class ThymioController:
    def __init__(self):
        # ...
        self.pose = SimplePoseStamped() # initialize message object
        self.robot_pose_sub = rospy.Subscriber('robot_pose', SimplePoseStamped, self
            .robot_pose_cb) # initialize subscriber
    def robot_pose_cb(self, simple_pose_stamped):
        """subscribe to the robot_pose topic to get the robot's current pose"""
        self.pose = simple_pose_stamped
        self.x = simple_pose_stamped.pose.xyz.x
        self.y = simple_pose_stamped.pose.xyz.y
        self.yaw = simple_pose_stamped.pose.rpy.yaw
```

where we use the these robot pose as the attribute of the class instead of a global variable while no need to worry about their scope.

As a result, the main function of the program can be reduced to few lines. All the variables and ROS functionalities are initialized while creating an instance of `ThymioController` class.

```
if __name__ == '__main__':
    pi = math.pi
    tc = ThymioController()
    try:
        tc.run()
    except rospy.ROSInterruptException:
        pass
```

By using classes, it will bring usability and maintainability at the same time: it will be easier to create reusable blocks that allow for scaling more efficiently; No more global variables which have to declare before a specific function; no more "hacking" to make variables communicate between each other. To summarize, OOP is not the only solution but a suitable solution for a three-week compact development cycle and multi-person cooperation, which brings convenience for the following control and obstacle avoidance development and testing.

3.2 PID Control

Two PID controllers are implemented to control the forward velocity **v** and the angular velocity **w** respectively.The pseudo-code in Algorithm 1 briefly explain the logic of the controller:

Algorithm 1: PID controller

Input: The current error between the desired and current angle (position): `err`;
The previous error between the desired and current angle (position): `prev_err`;
The integral (I) part of the PID control loop: `integrator`;
The PID parameters: `K_p, K_i, K_d`;
The maximal output value: `u_max`.

Output: The integral (I) part of the PID control loop: `integrator`; The computed output: `u`.

```
1 integrator = integrator + dt * (err + prev_err) / 2
2 derivative = (err - prev_err) / dt
3 u = Kp * err + Ki * integrator + Kd * derivative
4 if (u > u_max) and (K_i != 0) then
5 |   u = u_max
6 |   integrator = integrator - dt * (err + prev_err) / 2
```

where Line 4 - 6 explains the input saturation process. When the computed input exceeds the preset limit, the input will be saturated, and the integrator will not accumulate the error in this loop. That anti-windup operation can prevent the instability from mistakenly adding the large error to the integrator when the input is saturated.

For the PID controller of the angular velocity, we use the Ziegler-Nichols method to tune the parameters, as shown in Fig. 5. Thanks to the rqt_plot tools, we measure the ultimate gain $\mathbf{K_u}$ to be 7 and the oscillation period $\mathbf{T_u}$ to be 1.17s. The rqt_plot tools is discussed in Appendix 6.3.

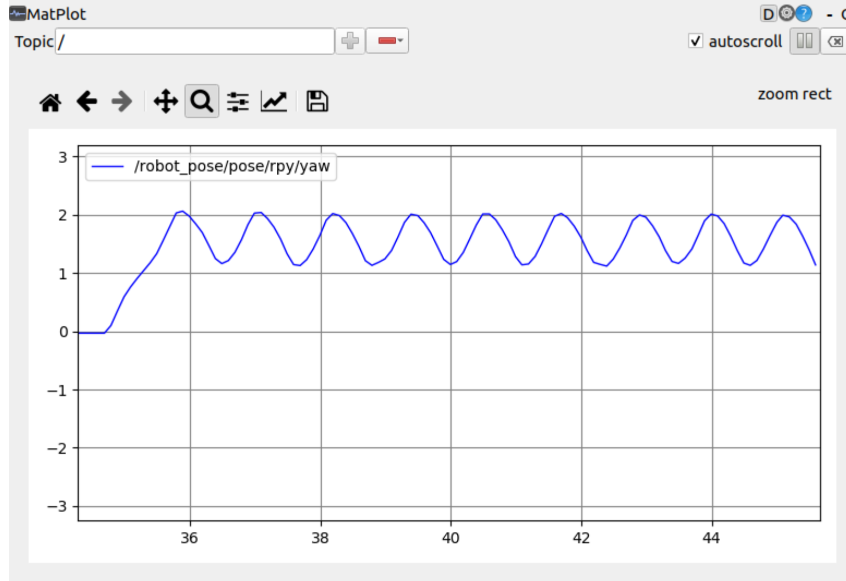


FIGURE 5: The plot of yaw angle when using the ultimate gain $\mathbf{K_u}$

After trying all the types in table 1 and further tuning, we finally find the parameters having the best performance for angular velocity **w** is $[K_p, K_i, K_d] = [3, 0, 0]$.

Control Type	K_p	K_i	K_d
P	$0.5K_u$	-	-
PI	$0.45K_u$	$0.45K_u/T_u$	-
PD	$0.8K_u$	-	$0.1K_uT_u$
PID	$0.6K_u$	$1.2K_u/T_u$	$0.075K_uT_u$

TABLE 1: Ziegler–Nichols method

Although Thymio can run as fast as 0.2m/s, the limitation of the forward speed **v** in our controller is set to be 0.05m/s so that Thymio will not behave too aggressive to detect obstacles and stop in time. Due to the low max speed, we use a P-controller with conservative parameters, which is $[K_p, K_i, K_d] = [2, 0, 0]$.

3.3 Waypoint Following

In terms of way following, we first check the existence of a goal point from the service **current_waypoint**. If not, Thymio will stop. Else, we will send the current pose of Thymio, which we get from the **robot_pose**, to the service **check_waypoint_reached** to check whether the goal is reached. If reached, Thymio will stop, and we set the integrator and previous error of both controllers to be 0. If the robot is still on the way, both PID controllers will compute a velocity and send it to Thymio. Note that if the difference between the current yaw angle and the desired one is more than $\pi/2$, the forward distance will be 0.

The pseudo-code of our way-following algorithm can be summarized as Algorithm 2.

Algorithm 2: Way following

```
1  if there is no goal then
2  |  set v,w to be 0
3  else
4  |  if reach the goal then
5  |  |  set v,w to be 0
6  |  |  reset prev_err and integrator to be 0 (for both angle control and distance
   |  |    control)
7  |  else
8  |  |  compute the error of the yaw angle and the error of the distance
9  |  |  if error of yaw angle is greater than pi/90 then
10 |  |  |  set v to be 0 and use Algorithm 1 to compute w
11 |  |  else
12 |  |  |  use Algorithm 1 to compute both v and w
13 |  |  end
14 |  end
15 end
16 update: prev_err = err (for both angle control and distance control)
17 publish v and w to the topic set_velocities
```

3.4 Obstacle Avoidance

In this part, an obstacle avoidance algorithm is implemented so that the Thymio robot could still do the trajectory following even if there are obstacles in its way. A pledge algorithm for the left wall following is chosen, whose logic is described as in Algorithm 3

Fig, 6 visualizes the state space diagram: Robot will switch between 3 modes, i.e., **turning_left**, **going_forward** and **turning_right**, when obstacle avoidance activates.

The robot is in **turning_left** mode by default once it enters obstacle avoidance. In this mode, it tries to turn left until obstacles no more block the robot. Then the robot would go forward for a small distance and try turning right back to its original direction towards the target. If there are still obstacles ahead, the robot will continue to turn left until it gets into a position where the target direction is no more blocked. Concretely, the detailed actions that the robot will be performing are explained in steps as shown in Fig. 7, where the procedure can be described step by step as follows:

Step 1: The robot always turns its head in the direction of the target when the data is accessible.

Step 2: The robot keeps heading towards the object unless it detects an obstacle ahead.

Step 3: The robot enters obstacle avoidance and initializes itself in left turning mode.

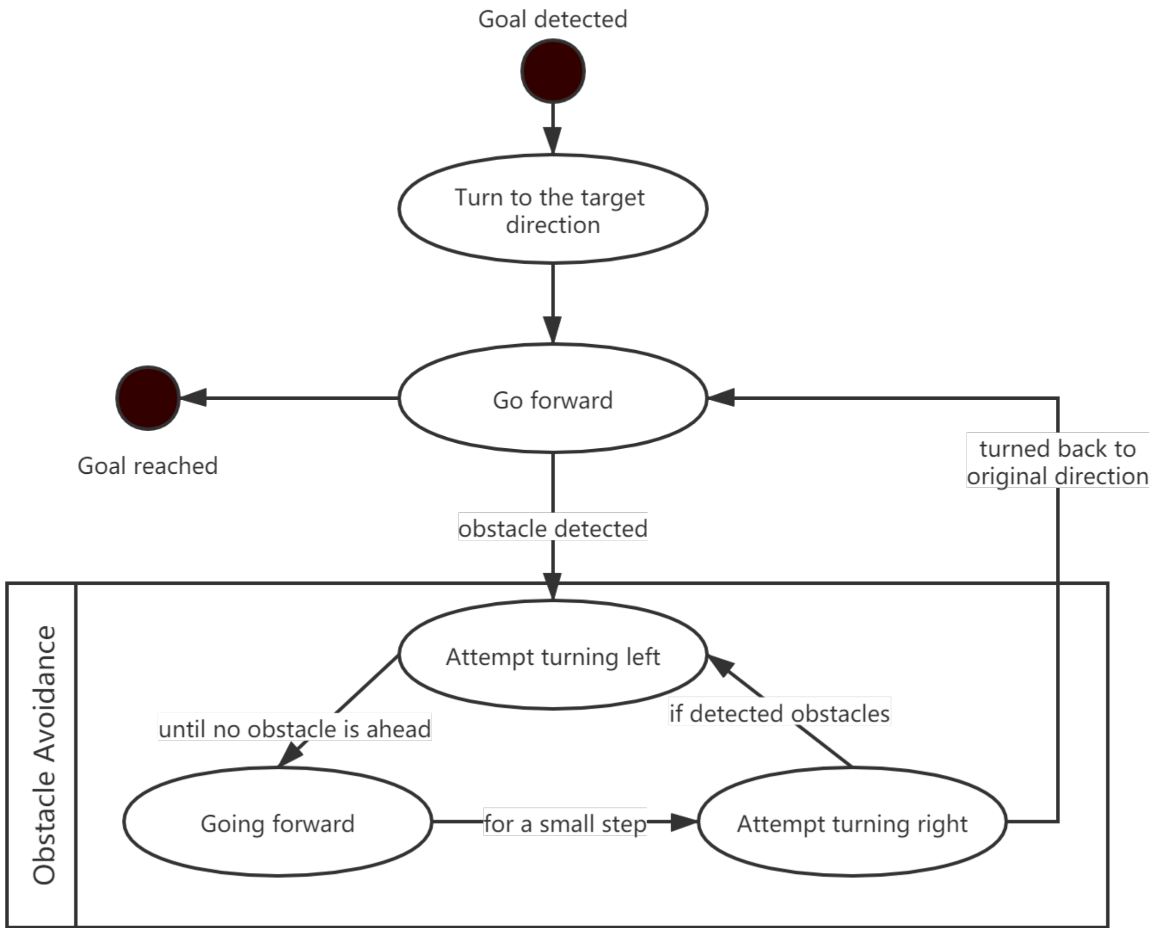


FIGURE 6: Simple state-space diagram of obstacle avoidance

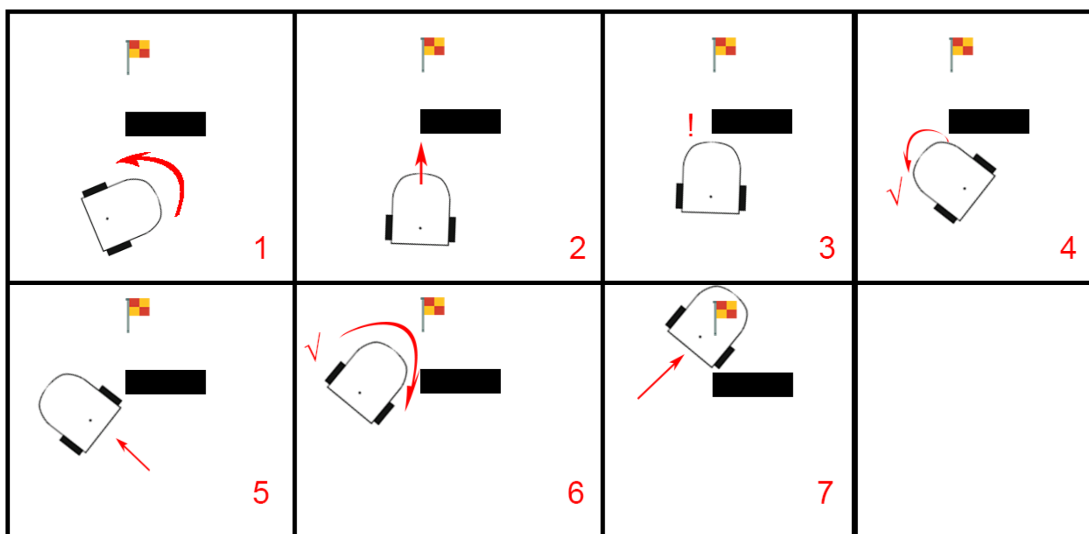


FIGURE 7: Robot actions in steps in trajectory following with obstacle avoidance

Step 4: The robot turns left until the proximity sensor value drops beyond a threshold, meaning there are no obstacles ahead in the current direction.

Step 5: The robot moves forward for a small distance and enters **_turning_right** mode.

Step 6: The robot keeps turning right in the turning_right mode until it is back to its initial direction when it entered obstacle avoidance or there is obstacle ahead. For the second case, the robot goes back to step 4 and tries moving into another direction to safely turn back without collisions.

Step 7: The robot exits obstacle avoidance and goes to step 2. It keeps detecting obstacles while heading towards the target. On running into obstacles, it will enter obstacle avoidance again. If the robot has reached the target position, the next point in the trajectory following queue will be passed as the new object, and the robot starts from step 1.

Algorithm 3: Left-wall following

```
1  while there is goal and not reach the goal do
2      if not obstacle avoidance then
3          turn to the direction towards goal
4          if not obstacles ahead then
5              use Algorithm 2 to go towards goal
6          else
7              stop; enter obstacle avoidance; enter left_turning mode; set counter to be 0
8          end
9      else
10         if left-turning then
11             try turning left; add degrees to counter; enter going_forward mode
12         if going-forward then
13             try going forward enter right_turning mode
14         if right-turning then
15             if not obstacle ahead then
16                 try turning right; subtract degrees from counter
17             else
18                 enter left_turning mode
19             end
20         if counter is 0 again then
21             exit obstacle avoidance
22     end
23 end
```

4 Experiments & Discussion

4.1 Implementation Details

We combine the PID controller and the Left-wall following described in section 3 to control Thymio to follow the given trajectory with obstacle avoidance. In the real experiment, a Lego building block is put in the origin as an obstacle, and the given waypoints are illustrated in Fig. 8. In the simulation, we use the same waypoints as those in the real experiment, and we set a square obstacle, which has a similar shape as the Lego building block, to be the obstacle. Moreover, we try some other scenes to test our control algorithm. Recordings of these simulations and the real experiment is uploaded here.

The key parameters used in our algorithm are listed in Table. 2.

Parameters	Value
Maximal forward velocity	0.05 m/s
Maximal angular velocity	π rad/s
PID parameters for position control	$[K_p, K_i, K_d] = [2, 0, 0]$
PID parameters for yaw angle control	$[K_p, K_i, K_d] = [3, 0, 0]$
Threshold for real experiment (starting from sensor 0-4)	$[3420, 34000, 3480, 3460, 3520]$
Threshold for simulation (starting from sensor 0-4)	$[0.025, 0.03, 0.03, 0.03, 0.025]$

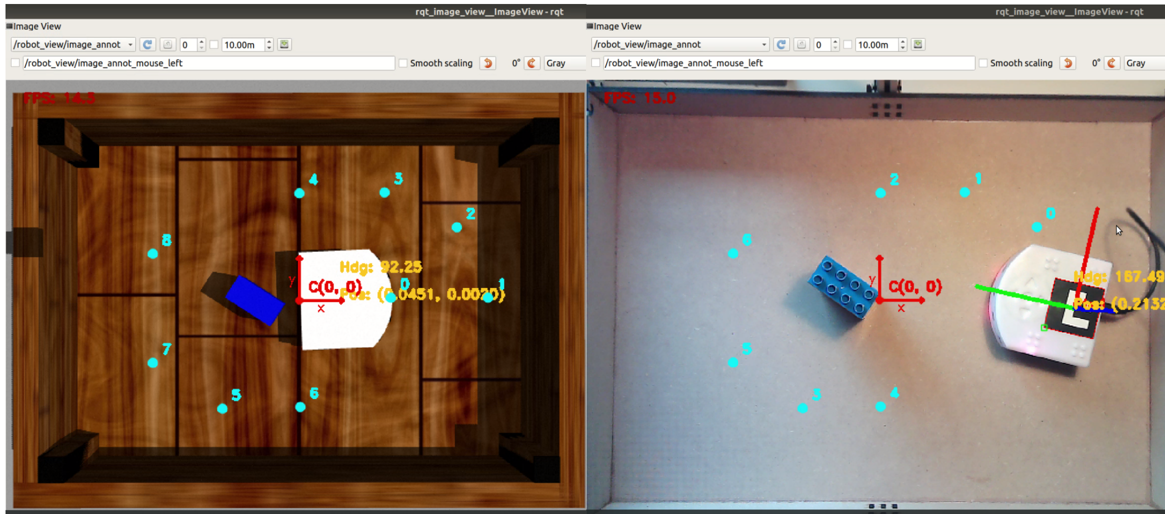
TABLE 2: Measurement of sensor values at the distance of 3cm in real world

4.2 Robot Performance Evaluation & Analysis

To have a better simulation, we make a scene, illustrated in Fig, 8, similar to that used in the real experiment. In the simulation, Thymio perfectly tracks all the waypoint. When moving from point 4 to point 5 in the simulation, Thymio can successfully detect the blue obstacle and bypass it without any collision. Furthermore, we try several other scenes to test our algorithm. In most experiments, Thymio can reach the waypoints without touch any obstacles. But, sometimes Thymio takes a longer path, and that will be discussed in section 4.3.2.

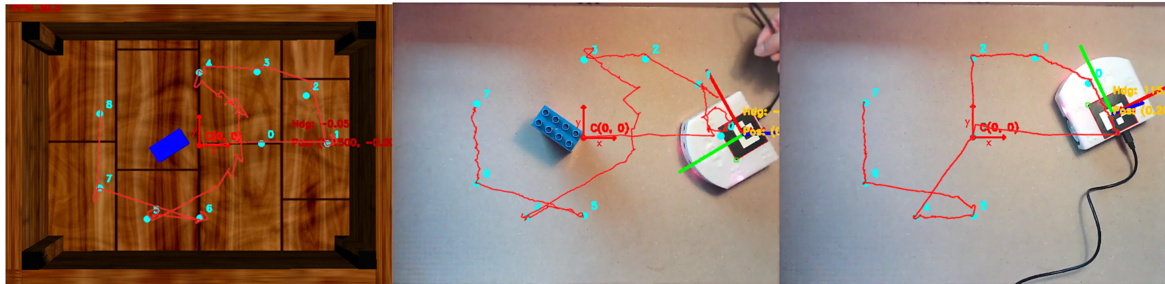
In the real experiment, Thymio can follow the desired trajectory and do obstacle avoidance despite two issues.

One is that Thymio will mistakenly enter local avoidance when moving to point 1 in Fig. 8 (A). That is because point 1 is to close to the border of the space. Moreover, the camera cannot accurately detect the position of the Thymio, which makes the Thymio needs to reach is even closer to the border. When Thymio moves towards point 1, it will go too close to the wall, and the sensor value will exceed the threshold so that Thymio start to do obstacle avoidance.



(A) Simulation test in Gazebo

(B) Real-world test on the campus



(C) Trajectory with obstacle avoidance in simulation

(D) Trajectory with obstacle avoidance on campus

(E) Trajectory without obstacle avoidance on campus

FIGURE 8: The tested environment and trajectory of the Thymio robot in Gazebo and on-site testbench. Example videos of the performance can be found here.

The other issue is that Thymio slightly touches the obstacle one time during the experiment. This phenomenon sometimes also happens in the simulation when using the control code implemented in the real experiment. One explanation is that we miss the values from right-sided sensor 3 and 4 on the if-condition of step 2 in Algorithm. 3. After adding the two sensors, Thymio won't hit a similar obstacle in the simulation. Though we don't have the chance to test our algorithm on campus due to the COVID difficulties, we think the corrected algorithm can perform much better in the real experiment as it does in the simulation.

4.3 Discussion

4.3.1 Sim2Real Challenges

Although simulation in Gazebo can provide an excellent platform to verify the proposed algorithm and exploit the robot's potential without breaking our robot during exploration. However, these methods that work well on simulation may fail to generalize on a real robot. After extensive experiments and discussion, several key challenges, including **different scaling**

FIGURE 9: Thymio trajectory comparison between real and simulation

of sensor value, **less accurate odometry** and **impact from the cable**, are summarized as follows:

1. The feedback value from five front proximity sensors of the real Thymio has different scaling than that in simulation. According to our measurement, the sensor value is approximately 3400 when an obstacle is 3 cm in front of the sensor. The measured value of each sensor in the real world is listed in table 2. Moreover, the sensor property behaves differently in the real robot compared to the simulation: in reality, the value will increase when approaching the obstacle, while the value will decrease in Gazebo.

2. The output of robot pose in real experiments is not accurate and continuous, especially near the border. By recognizing the aruco marker on top of the robot, we could quickly get the robot's pose information. However, the precision is greatly influenced by radial/tangential distortion of the camera and the marker visibility variation caused by the power cable. In these situations, the robot may keep navigating centimeters away from the goal position and miss the ground-truth. Therefore, we managed to deal with this challenge: On the one hand, we tried to ensure that the cable is flat and avoid occlusion caused by bending and overlapping. On the other hand, we calibrated the provided camera with OpenCV 9×6 chessboard and the following command:

```
rosrun camera_calibration cameracalibrator.py --size 9x6 --square 0.020
    image:=/robot_view/image_raw camera:=/robot_view
```

3. The stiff cable connecting the robot will impact the motion of the robot when navigating near the border. With limited space near the drawer walls, the power cable will inference the robot rotation greatly as the robot could be controlled to articulate for a given time. It poses a challenge to our left-wall following strategy since Thymio needs to turn to the

original orientation to exit obstacle avoidance. In the on-campus test, we lifted the cable to mitigate the inference so that our Thymio can navigate smoothly. For future improvements, using a wireless dongle to connect the Thymio robot can reduce the effect considerably.

As a result, the trajectory comparison of our Thymio robot in simulated and real environment is shown as in Fig. 9.

4.3.2 Ways of Improving

One drawback of our obstacle avoidance is that Thymio will choose to turn left and follow the border for a long time if it meets a scene shown in Fig. 10. The example video showing the performance of Thymio when encountering such a scene can be found here. The corresponding video is named as **"thymio_simulation_corner_case"**. Although Thymio can finally reach the goal, this is not the best path. The global optimal solution is to turn right and bypass the obstacle. One way to improve the algorithm is to exploit the complete map information to Thymio and implement a global planning algorithm such as A*.

FIGURE 10: Corner case of the obstacle avoidance algorithm

5 Conclusion

In this project, we first developed the model of the Thymio by editing the urdf.xacro files. Both basic primitives and detailed collada file exported from SolidWorks are used to describe our model. With a collada file, we could achieve high-precision modeling of the robot. After that, we also add the proximity sensors so that the environment's information with the Laser Scan message can be available. Moreover, the texture is added to the model so that our robot looks more vivid for intuitive visualization.

OOP is utilized as our developing paradigm to make the project more readable and easy for development. We implement the classic PID controller for the waypoint following algorithm. We use the Ziegler-Nichols method to tune the parameters. Left-wall following algorithm is introduced to realize the obstacle avoidance. Both simulations and on-campus experiments show that our controller behaves well on waypoint following and obstacle avoidance. Still, real-world experiments pose some challenges when transferring code from simulation, such as different sensor value scaling, the error of the measured position, and the cable's impact. Maybe a wireless Thymio can be used for the on-campus experiment in the future to eliminate the disturbance from the cable.

6 Appendix

6.1 Instructions for Launching Examples

ROS launch files are powerful tools to start master and multiple nodes with preset parameters without having to issue commands one-by-one every time to bring convenience in the process of algorithm tuning, test, and evaluation. Therefore, we mainly create and modify the following launch files to boost our development during this project.

1. `thymio_ctrl_cli.launch`

   AFor the on-campus test, we create a separate control code snippet, which could avoid code conflicts caused by the on-site adjustment of parameters. To ensure one-click convince, we use if and unless attributes in the launch file to switch between the simulation and real-world test code as follows:

   ```
   <group if="$(arg is_simu)">
     <node name="thymio_control_pnode" pkg="ros_basics_exercise" type="
         thymio_control_pnode_simu.py" output="screen"/>
   </group>
   <group unless="$(arg is_simu)">
     <node name="thymio_control_pnode" pkg="ros_basics_exercise" type="
         thymio_control_pnode_real.py" output="screen"/>
   </group>
   ```

   where `is_simu` argument is the condition to determine when launching the files.

2. `set_simu_waypoints_obstacle.launch`

   To quantitatively compare performance in both the simulation and on-campus test, we build a blue lego-like model, i.e., `ros_basics_lego_block.urdf` compatible with the the prepared waypoint lists as shown in Fig. 11.

   In this launch file, `path_to_follow.py` is first launched as node "create_trajectory". In the meantime, the obstacle model is placed using `spawn_model` with several default

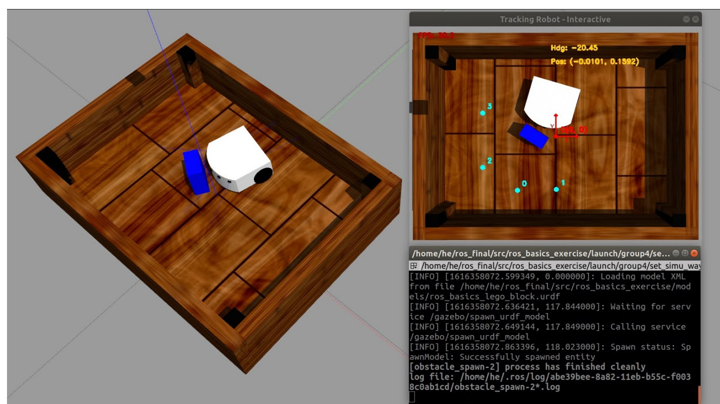


FIGURE 11: Using `set_simu_waypoints_obstacle.launch` simulation test

arguments including the obstacle model path and pose (x, y, and yaw coorespond to `-x`, `-x`, and `-Y` separately) in the simulator.

As a result, the standard simulation test for waypoints following and obstacle avoidance can be simplified by issuing the following commands:

```
roslaunch ros_basics_control simu_thymio.launch
roslaunch ros_basics_exercise set_simu_waypoints_obstacle.launch
```

3. `tune_with_rqt.launch`

rqt is a software framework of ROS that implements the various GUI tools in the form of plugins. In our implementation, we integrate three rqt plugins into the launch file, which helps our algorithm development and experimental analysis.

First plugin is dynamic_reconfigure [2]. After the preliminary implementation of proposed algorithms, we further utilize this plugin by creating a `.cfg` file to tune the key parameters as shown in Fig. 13. For the detailed description of setting up dynamic reconfigure is referred to Apprendix 6.3.

Second plugin is rqt_plot [3]. When tunning the PID parameters as shown in Table 2, the real-time plotting function can provide intuitive information, including settling time and overshooting, oscillation period, and ultimate gain.

As shown in Fig. 13, the designed script for the reconfiguration will print out the current main parameters, including max speed limit, PID parameters, and operating mode in the terminal. With PyQtGraph as the back-end, real-time pose information from topic `/robot_pose` is shown in two different plotting windows. For more details, example video of using rqt_plot and rqt_reconfigure can be found here.

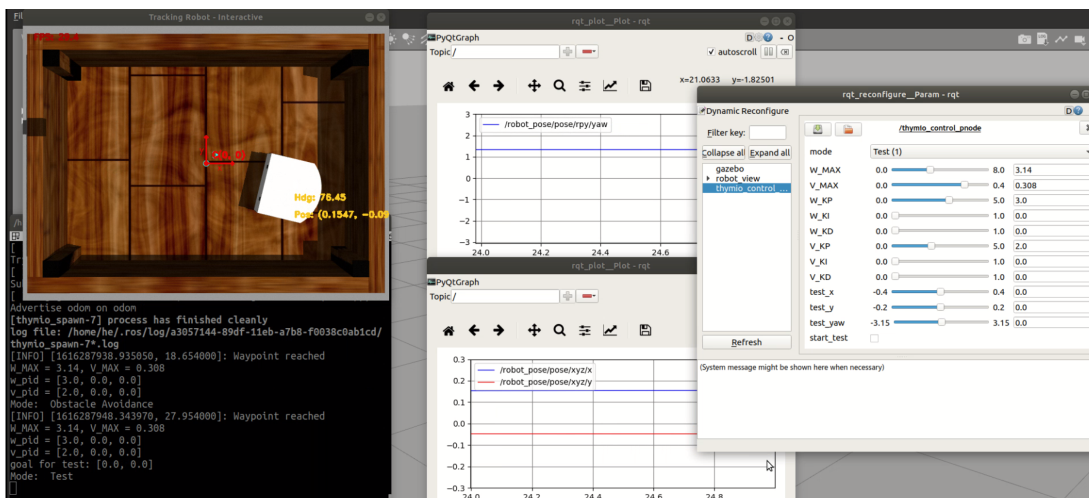


FIGURE 12: Using `tune_with_rqt.launch` for algorithms tuning and visualization

[2] https://wiki.ros.org/rqt_reconfigure
[3] https://wiki.ros.org/rqt_plot

Third plugin is rqt_image_view [4]. By including topic `/robot_view/image_annot` published by `/thymio_simu_tracking_node`, we can visualize the output annotated images for further evaluation.

In this launch file, three arguments with the prefix `use_` correspond to three plugins are set to switch on and off with the help of if attribute flexibly for different usage. As a result, simulation with rqt tools can be activated by issuing the following commands:

```
roslaunch ros_basics_control simu_thymio.launch
roslaunch ros_basics_exercise tune_with_rqt.launch
```

4. `view_with_rosbag.launch`

To qualitatively verify the robot performance from rosbag files, this launch include `rosbag play` command and proposed `tune_with_rqt.launch`. Similarly, the default parameters regarding the play rate, rosbag path, and bag name are set for flexible use. The example of using this launch file is shown as Fig. 8. As a result, view the rosbag with rqt_image_view can be activated by issuing the following commands:

```
# 1. default (visualize rosbag in simulation scene)
roslaunch ros_basics_exercise set_simu_waypoints_obstacle.launch
# 2. visualize rosbag in real-world scene
roslaunch ros_basics_exercise set_simu_waypoints_obstacle.launch is_simu:=
    false
# 3. visualize with other bags in real world
roslaunch ros_basics_exercise view_with_rosbag.launch is_simu:=false
    real_bag_name:=thymio_real_without_obstacle
```

6.2 Development on different ROS distributions

We use Ubuntu 20.04 in the VM with ROS Noetic distribution and the personally installed Ubuntu 18.04 with ROS Melodic distribution for algorithm development and debugging in the simulation phase.

During the on-campus verification phase in the third week, we mainly use the personal computer with Ubuntu 18.04 and ROS Melodic distribution to connect the device and perform the camera calibration to achieve final real-world testing.

In terms of on-site verification, we first checked the operational status of the provided camera by using `guvcview`[5]. When connecting to the Thymio robot, the python library `thymiodirect`[6] is required to ensure the use of `thymio_intf_node.py`. Due to visible deviation, we calibrated the provided camera by using OpenCV 9×6 chessboard.

Although the default Python versions of ROS Noetic and Melodic distributions are different, the main python code, either from the main codebase or our proposed one, is written compatible

[4] https://wiki.ros.org/rqt_image_view
[5] http://guvcview.sourceforge.net/
[6] https://pypi.org/project/thymiodirect/

with both versions. Therefore, the overall simulation and test can work smoothly using non-default settings.

6.3 Setting up Dynamic Reconfigure for Parameters Tuning

After adding related dependency in `CMakeLists.txt` and `package.xml`, the proposed `ThymioController.cfg` is proposed with following structures:

1. an mode parameter defined with integer type whose value is set by an enum, which includes `Obstacle_Avoidance(0)` and `Test(1);`
2. eight parameters defined with double type to cover main arguments for PID controller of Thymio;
3. three parameters defined with double type to denote the goal position when using the `Test` mode;
4. `start_test` defined with bool type to activates the robot when using the `Test` mode.

where Fig. 13 shows the `dynamic_reconfigure` dashboard for parameter tuning.

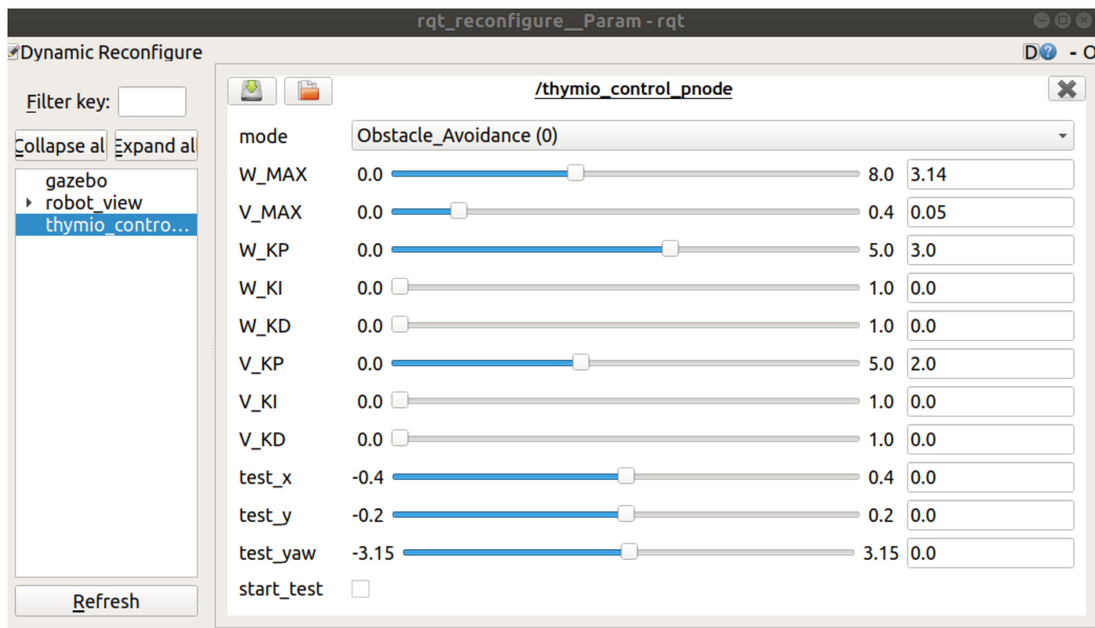


FIGURE 13: Using `tune_with_rqt.launch` for algorithms tuning and visualization

When implementing with main control function, the proposed cfg file and the dynamic reconfigure is imported as follows:

```
from dynamic_reconfigure.server import Server
from ros_basics_exercise.cfg import ThymioControllerConfig
```

Following the OOP-style, the dynamic reconfigure service and parameter set `dynamic_param` is initialized within the `__init__()` method while callback function `reconfig()` is enclosed

as instance method to increase the usability and readability.

As a result, we can easily adjust the parameter for better qualitative and quantitative performance using the proposed `tune_with_rqt.launch`.