École Polytechnique Fédérale de Lausanne

Robotics Practicals (MICRO-453)

Lecturers: Vaios Papaspyros, Rafael Barmak

Topic 11:

ROS Basics

Group 3 - Program P11

Francesco Nonis *SCIPER: 300540*

Erfan Etesami *SCIPER: 337448*

Hugo Miranda Queiros *SCIPER: 336706*

Spring 2022

Contents

1 Introduction

Robot Operating System (ROS) is an open-source environment that allows designers to develop robots and build robotic applications more efficiently by letting them reuse and share various tools. Thanks to the notion of nodes and topics, ROS allows different processes to communicate and share information through messages. Each node is representative of a unique process and can either publish or subscribe to a topic. More specifically, a node can publish a set of data, named messages, and pass them through a channel called topic. On the other hand, a node can also subscribe to a topic, retrieve the messages sent through it, and use them for its own purpose. This structure allows having different processes working independently whilst being able to share information.

Through the *ROS Basics* practical sessions, we first designed a simplified version of the Thymio robot in the URDF file format. This design consists of the main body of Thymio, its caster wheel, differential drive system, and seven proximity sensors. The main goal of this practical was to make this robot model capable of waypoint following and obstacle avoidance. These control strategies were implemented in ROS and simulated with Gazebo. Finally, we tested the codes we had developed for the simulation on a real Thymio robot in order to compare the control performance in both cases.

The remainder of this report is organized as follows: The details of the robot model are explained in section 2. Section 3 details the underlying principles of the waypoint-following control algorithm. Section 4 is devoted to the obstacle avoidance strategy. The comparison of the control performance in simulation and real-life is provided in section 5. Section 6 summarizes and concludes this report. Appendix A contains information about files and commands to run the simulated path used in this report. This section also provides information about another simulation (Thymio in the box) that reveals the robustness of the developed obstacle avoidance routine.

2 Robot Model

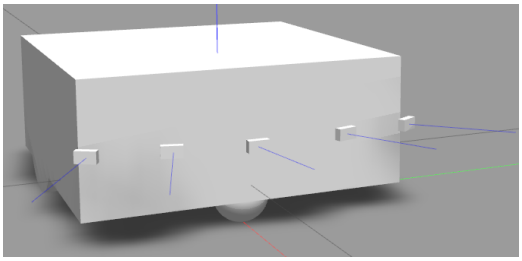
The design of the Thymio robot is done using the provided Xacro file which was used then to auto-generate an URDF file. Thymio is approximated by a box as its main body, two cylinders on its sides as the wheels of the differential drive system, and a semi-sphere on its bottom as the caster wheel. Since it was not possible to generate a semi-sphere in the URDF file format, a full sphere whose center is aligned on the bottom of the box was created instead. Therefore, half of the sphere goes inside the body. Besides, the caster wheel is modeled as a separate link which is connected to the main body with a *fixed* joint.

The box representing the Thymio's main body has the dimensions of $0.11 \times 0.112 \times 0.045$ $[m]$ and its mass is of 210.6 $[g]$ (78% of Thymio's mass). The two cylinders which are representative of the wheels have a radius of 0.022 $[m]$, a width of 0.015 $[m]$ and a mass of 27 $[g]$ (10% of the total mass). Finally, the sphere used as the caster wheel is of radius of 0.0095 $[m]$ and of mass of 5.4 $[g]$ (2% of the Thymio's mass). Thymio is considered to have a mass of 270 $[g]$. It is important to note that the right placing of the wheels and the caster wheel is crucial to ensure the stability of the robot and preventing it from tilting. After a careful investigation, the center of the wheels is placed symmetrically on 0.02 $[m]$ behind and 0.01 $[m]$ below the center of the box. The center of the caster wheel is placed on the bottom of the box, on the origin of the y axis and 0.045 $[m]$ in front of the center of the box.

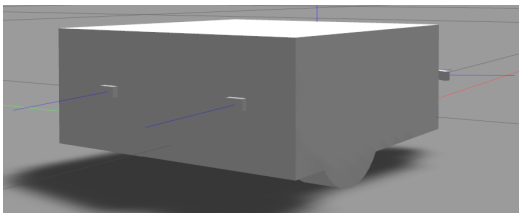
Thymio also has seven horizontal proximity sensors (five on the front and two on the back). Including these sensors in the design is essential for having the obstacle avoidance routine. To mimic the curvy shape in front of the Thymio robot, the front sensors are considered to have a difference of 15° with respect to their neighbors such that the front center sensor is parallel to the front face of the box and the rest of front sensors are titled outward. The coordinates of the sensors are summarized in the Table 1. It is noteworthy that each sensor is assumed to have a mass of 1 $[g]$. The final model of the robot is shown in Fig. 1.

Table 1: Coordinates of the horizontal sensors

Sensor ID	**x, y, z coordinates**	**roll, pitch, yaw coordinates**
0 (front left)	$+0.0600,\ +0.0575,\ 0$	$\pi,\ 0,\ +\frac{2\pi}{3}$
1 (front mid-left)	$+0.0650,\ +0.0325,\ 0$	$\pi,\ 0,\ +\frac{7\pi}{12}$
2 (front center)	$+0.0700,\ +0.0000,\ 0$	$\pi,\ 0,\ +\frac{\pi}{2}$
3 (front mid-right)	$+0.0650,\ -0.0325,\ 0$	$0,\ 0,\ -\frac{7\pi}{12}$
4 (front right)	$+0.0600,\ -0.0575,\ 0$	$0,\ 0,\ -\frac{2\pi}{3}$
5 (back right)	$-0.0560,\ -0.0300,\ 0$	$0,\ 0,\ +\frac{\pi}{2}$
6 (back left)	$-0.0560,\ +0.0300,\ 0$	$0,\ 0,\ +\frac{\pi}{2}$



(a) Front view of the robot model



(b) Back view of the robot model

Figure 1: The final model of Thymio

3 Waypoint Following

To implement the waypoint-following routine, two PD controllers are designed to correct for both the angular ($\frac{rad}{s}$) and linear velocities ($\frac{m}{s}$) separately. Those velocities are published through the *set_velocities* topic. The PD controller responsible for the forward velocity use the distance from the current position of the robot to the current waypoint as error. The PD controller for the angular velocity takes into account the difference between the robot's orientation and the desired orientation (the one which would make the robot point to the current waypoint) as error. The control terms used in simulation are summarized in the Table 2, and the pseudo-code of the waypoint-following control algorithm is given in Fig. 2.

Table 2: Control terms (simulation)

Linear velocity	**Angular velocity**
$K_{p-fwd} = 0.8, K_{d-fwd} = 0.1$	$K_{p-ang} = 1.5, K_{d-ang} = 0.1$

Note that since all the angles are given in the interval $[-\pi, \pi]$, we recompute the difference between the desired orientation and the actual orientation of the robot by adding or subtracting 2π to its value when the absolute value exceeds $-\pi$ or π respectively. This method helps using `math.atan2()` function and keeps the difference in angle in $[-\pi, \pi]$ while avoiding any issue resulted from the jump in angle which would compromise the functionality of the designed PD controllers.

Waypoint reaching strategy

```
Kp_fwd ← 0.8
Kp_angle ←1.5
Kd_fwd ← 0.01 * F                    #F is the sampling frequency
Kd_ang ← 0.01 * F

goal_angle ← atan2(goal_y – robot_pose_y,  goal_x– robot_pose_x)  #compute the desired robot orientation

distance ← ||goal  – robot position||_L2

angle_difference ← goal_angle – robot_orientation

If angle_difference > π
   angle ← angle - 2π
Else if angle_difference  < - π
   angle ← angle + 2π

speed_forward ← Kp_fwd * distance + Kd_fwd * (distance – previous_distance)
speed_angular ← Kp_angle * goal_angle + Kd_angle * (goal_angle – previous_goal_angle)

previous_distance ← distance
previous_goal_angle ← goal_angle

return distance, goal_angle
```

Figure 2: The pseudo-code of the waypoint-following algorithm

4 Obstacle Avoidance

The implemented obstacle avoidance procedure can be summarized as follows: Whenever an obstacle is detected the robot will move in the opposite direction and rotate accordingly based on the sensor which returns the maximum value. Therefore, the direction of the rotation depends on the orientation of a sensor which is the closest to the obstacle, and the sign of linear speed depends on whether the front or back sensors detects an obstacle. To have a successful switching from the permanent waypoint-following routine to the temporary obstacle avoidance strategy, the state machine presented in Fig. 3 is used.

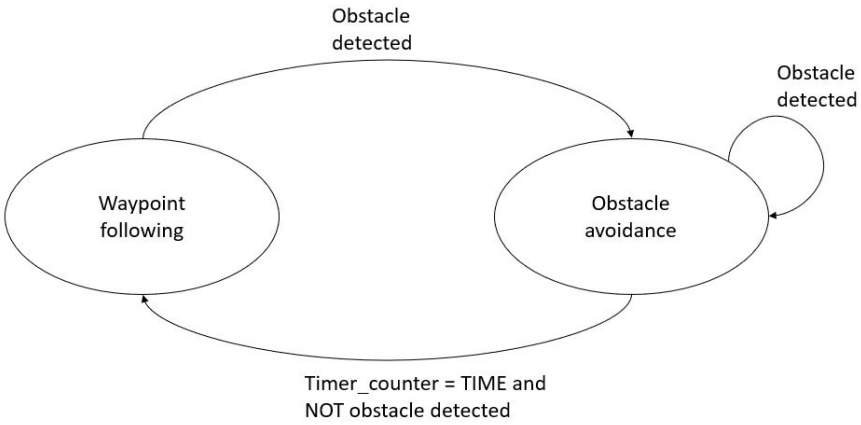


Figure 3: State machine

As depicted in Fig. 3, whenever an obstacle is detected, the robot switches to the obstacle avoidance state. Here, a timer starts counting and is incremented until it reaches a predefined constant (TIME $= 5$, which for a frequency of F $= 10$ [Hz], corresponds to a period of 0.5 seconds) which was determined empirically. When this constant is reached and there is not an obstacle anymore, the robot switches back to its waypoint-following routine. The pseudo-code of the obstacle avoidance algorithm for simulation is shown in Fig. 4(a).

```
min_val← 10
Idx_min ← 0
SENSOR_THRESH ← 0.03
for i in range(size(nbr_sensors))                    #determine the value of the closest sensors to the obstacle
      if  sensor(i) < min_val
          min_val ← sensor(i)
          Idx_min = i

If min_val <= SENSOR_THRESH
       state = STATE_OBS                             #change the state of the robot to Obstacle avoidance
       timer ← 0
       front_speed ← 0.035
       rotate_speed ← 1

      if idx_min == 0                                          #front left most sensor
         update_speed(-front_speed, -rotate_speed)

      else if idx_min == 1                                     #front left sensor
         update_speed(-front_speed, -rotate_speed/2)

     else if idx_min == 2                                      #front middle sensor
         update_speed(-front_speed, -rotate_speed/2)

      else if idx_min == 3                                     #front right sensor
         update_speed(-front_speed, rotate_speed/2)

      else if idx_min == 4                                     #front right most sensor
         update_speed(-front_speed, rotate_speed)

      else if idx_min == 5                                     #back right sensor
         update_speed(front_speed, -rotate_speed/2)

      else if idx_min == 6                                     #back_left sensor
         update_speed(front_speed, -rotate_speed/2)
```

(a) Obstacle avoidance pseudo-code for simulation

```
max_val← 0
Idx_max ← 0
SENSOR_THRESH ← 3500
for i in range(size(nbr_sensors))                    #determine the value of the closest sensors to the obstacle
      if  sensor(i) > max_val
          max_val ← sensor(i)
          Idx_max = i

If max_val => SENSOR_THRESH
       state = STATE_OBS                             #change the state of the robot to Obstacle avoidance
       timer ← 0
       front_speed ← 0.035
       rotate_speed ← 2

      if idx_max == 0                                          #front left most sensor
         update_speed(-front_speed, -rotate_speed)

      else if idx_max == 1                                     #front left sensor
         update_speed(-front_speed, -rotate_speed/2)

     else if idx_max == 2                                      #front middle sensor
         update_speed(-front_speed, -rotate_speed/2)

      else if idx_max == 3                                     #front right sensor
         update_speed(-front_speed, rotate_speed/2)

      else if idx_max== 4                                      #front right most sensor
         update_speed(-front_speed, rotate_speed)

      else if idx_max == 5                                     #back right sensor
         update_speed(front_speed, -rotate_speed/2)

      else if idx_max == 6                                     #back_left sensor
         update_speed(front_speed, -rotate_speed/2)
```

(b) Obstacle avoidance pseudo-code in real-life experiment

Figure 4: Obstacle avoidance pseudo-code

5 Simulation and Real-life Comparison

To test the validity and performance of the simulated robot model and control algorithms, a similar procedure is used on a real setup of camera and Thymio robot. The main difference in real-life experiment is that the

horizontal sensors of Thymio are of proximity kinds whose values should be mapped to the metric unit; necessary steps in this regard are taken in the supplementary files. It is also noteworthy that due to the friction in the real environment between Thymio and the table as well as internal frictions between joints, we had to increase the angular velocity of the robot in the real experiment compared to the simulation. Moreover, due to the problem of scaling generated by the real camera, we also had to multiply, in our code, the real path coordinates that we measured, by constants *kx* and *ky* so that they match exactly with their real value in space.

As before, two PD controllers are implemented to correct for the angular and linear velocities of the robot published through the *set_velocities* topic. The control terms in the real-life experiments are described in the Table 3. As can be seen, only the p term of the controller responsible for angular velocity is increased due the issues discussed above. Additionally, because of the inaccuracy of the the robot model used in simulation, the fact of having different controllers compared to those in simulation is completely reasonable. Note that the waypoint-following and obstacle avoidance routine are similar to the simulation case and their associated pseudo-code is shown in Fig. 2 and Fig. 4(b). For obstacle avoidance, the only change in commands was to increase, in the real experiment, the angular velocity from 1 to 2 $[\frac{rad}{s}]$ while avoiding an obstacle. This change allowed for better avoidance in the real-life experiment.

Table 3: Control terms (real-life)

Linear velocity	**Angular velocity**
$K_{p-fwd} = 0.8, K_{d-fwd} = 0.1$	$K_{p-ang} = 5, K_{d-ang} = 0.1$

Note that the pseudo-code for the obstacle avoidance routine in real-life differs a little from that of the simulation. More specifically, one small change is made in the sensor threshold condition to determine whether an obstacle is detected. As mentioned before, this is because the values measured by Thymio's sensors in real-life do not have the unit of distance (m) as in the simulation. Moreover, the distance values returned by sensors in simulation decrease as the robot gets closer to the obstacle. However, the actual Thymio's proximity sensors measure the reflected infrared light such that the closer an obstacle is, the larger their values would be. Therefore, to adapt the code from simulation to the real-life experiment, both the threshold condition and its value is changed. More concretely, the value of the sensors in simulation are checked to be smaller than a threshold (0.03); whereas in the actual robot, they are checked to be larger than the value 3500. It is also noteworthy that the `update_speed()` function of the pseudo-code is responsible for publishing the linear and angular velocities on the *set_velocities* topic with the aim of updating the speed of the robot.

The actual setup in which the real Thymio robot had to follow a trajectory was a rectangle of dimensions 80×60 $[cm]$ where a rectangular obstacle of dimensions 3.3×6.5 $[cm]$ was placed in its middle. The height of camera with respect to this environment was about 82 $[cm]$; therefore, we changed accordingly the value in the simulation launch file. The simulated and real desired trajectory followed by the robot model and real Thymio respectively is shown in Fig. 5.

One can also see in Fig. 6 the evolution of the real and simulated orientation of the robot through time. The strong correlation between the both cases for the trajectory and the orientation can clearly be seen on both figures. The differences between the simulated version of the followed trajectory and that of real-life in some parts of the path is related to the inherited errors in the robot model used in simulation. For instance, as discussed in detail in section 2, the shape of Thymio and its differential drive is approximated mainly by a box and two simple cylinders in simulation. Besides, the sensor placement in simulation is different from the real setting of actual Thymio.

Finally, concerning the orientation, one can see that the curve of the real orientation is quasi the same as the simulated one but is spread along the time axis. This spread can be explained by the fact that the simulation is faster than the real robot to finish the path. This significant difference in time is mainly due to the fact that, in real life, there is friction between the table and the robot and also in all the joints, which are not taken into

account during the simulation; and that the real path was not as optimal as the simulated one. Moreover, the fact that we did not increase the linear speed commands and gains in the real experiment to compensate for the friction, combined with the previous approximations and errors in measurement of the camera, are enough to explain the differences visible in Fig. 6 and why the real robot was slower.

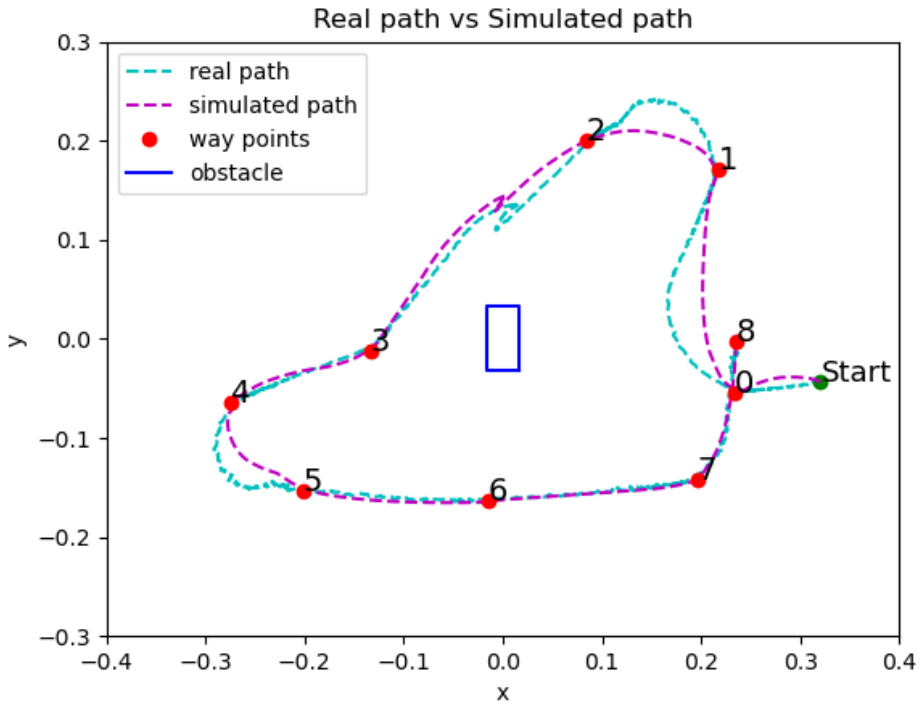
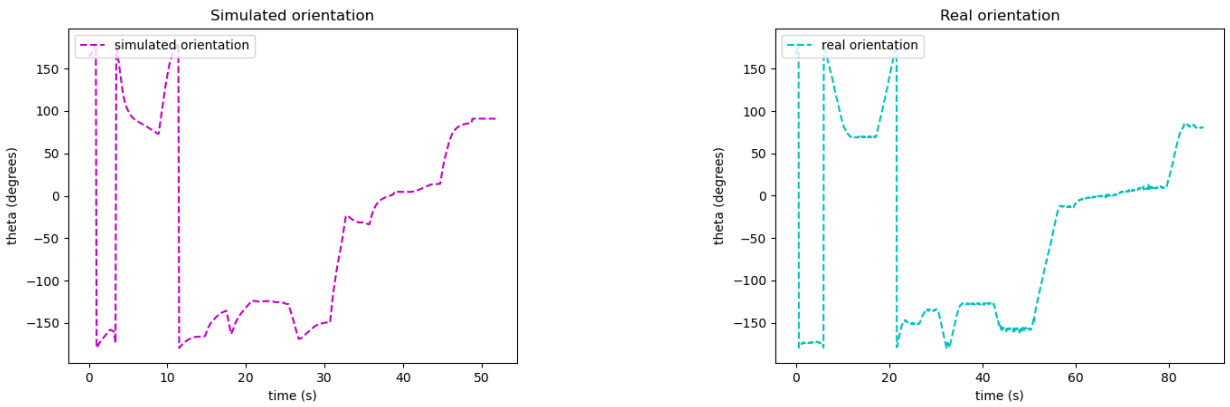


Figure 5: Simulated and real-life trajectory of the robot



(a) Simulated orientation

(b) Real orientation

Figure 6: Simulated and real-life orientation of the robot

6 Conclusion

The goal of this practical was for us to get familiar with ROS by implementing some control and obstacle avoidance strategies on the Thymio robot. We enjoyed how easy it was to use a similar piece of code for both the simulation and real-life experiment. For example, thanks to the notion of topics and the environment

provided by ROS, we managed to compute the desired linear and angular velocities needed to feed into the robot without actually having to do the conversion from linear and angular speeds to the actual commands. Even though the code was easy to reuse on the real Thymio and the trajectories followed by the robot in both the simulation and real situation are quite close to each other, there are still some discrepancies in between them. This results from the fact that the model of Thymio used in simulation is just an approximation. Apart from the difference in the shape of the actual Thymio and the designed model, there are also a lot of unknown parameters, such as friction coefficient or the exact mass distribution of Thymio. To compensate for these errors, the control parameters were retuned for the real-life experiment.

7 Appendix A: Robustness of the Obstacle Avoidance Algorithm

To run the simulated path that we used to produce the plots of this report, you need to download the *src_path.zip* file and after building and sourcing your workspace, you just need to type the command below.

```
roslaunch ros_basics_control simu_thymio.launch
```

We also provide a test of our obstacle avoidance algorithm where Thymio is put inside a small box. In that test, one can observe the robustness of the control routines we developed as the robot model follows well the given waypoints despite the small size of the box. To run this test, you need to download the *src_obs.zip* file and after building and sourcing your workspace, you just need to type the command below.

```
roslaunch ros_basics_control simu_thymio.launch
```