MICRO-453: ROBOTICS PRACTICALS

TP11 ROS basics

June 9, 2019

CUI Mingbo

LI Weipeng

ZHOU Xiao

Contents

1 Introduction

Robot Operating System (ROS) is a robotic software platform originating from Stanford University that provides operating system-like functions for heterogeneous computer clusters. The primary design goal of ROS is to increase code reuse in the field of robotics. ROS is a distributed processing framework (aka Nodes). This allows executables to be designed separately and loosely coupled at runtime. These processes can be packaged into packages and stacks for easy sharing and distribution.

ROS has three levels of concepts:

1. `Filesystem level`: The internal structure, file structure and required core files of ROS are at this level. Understanding the ROS file system is the basis for getting started with ROS. The structure of a ROS program is a folder that is distinguished by different functions. For example, the general folder structure is: *Workspace folder -> source file space folder (src), build space folder (build) and development space folder (devel)*; (source file space folder and further feature package.)
2. `Computation graph level`: mainly refers to the communication between processes (between nodes). ROS creates a network that connects all processes, through which interactions between the network nodes are obtained, and information published by other nodes is obtained.
3. `Communication level`: mainly refers to the acquisition and sharing of ROS resources. Through an independent online community, we can share and acquire knowledge, algorithms and code, and the open source community's strong support enables ROS systems to grow rapidly.

In this practical, we will go through the basics of the ROS by some of its basic capabilities. Our experiment includes 2 parts: (1) the building of a simple 2-wheeled robot (based on the Thymio robot) with respect to the Unified Robot Description Format (URDF) used in ROS and simulated in Gazebo(Code in APPENDIX I). (2) Implementing our velocity control algorithm to move the robot by adding the differential drive to our robot. We use the Gazebo's differential drive plugin, so our robot can move in the X and Y directions. Afterward, we will have a discussion on the PID controller to control the robot for different x-y position commands (Code in APPENDIX II). And then we will test our robot to verify its stability and controllability.

2 Robot model

The robot description file we use is: Unified Robot Description Format (URDF for short). The urdf package in ROS contains a URDF C++ parser, and the URDF file describes the robot model in XML format. The URDF file can specify the appearance, dynamics, kinematics and collision behavior of the robot. And visual representation and collision model of the robot. URDF specifies robot models that primarily use two different element types: "link" and "joint"

Figure .1 .2 .3 is the three views of our robot. As can be seen from these three figures, the shape of the robot we designed is modeled after the Thymio robot. Its base link consists of three parts: a cylinder, a cuboid, and a sphere. These three assignments are linked with a "fixed" joint to ensure a complete rigid body. Since this robot only has two large wheels in the second half, we have to place a sphere on the front end of the robot to keep the entire robot body in balance. This sphere also gives us more control over this robot. The two wheels of the robot are cylindrical. The thicker wheels give the robot better stability. But too thick wheels can make turning difficult.

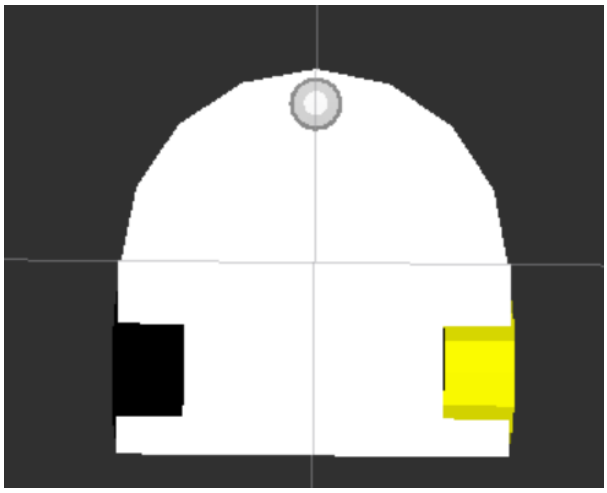

Figure 1: Bottom view of the robot

Figure 2: Side view of robot

Figure 3: Top view of the robot

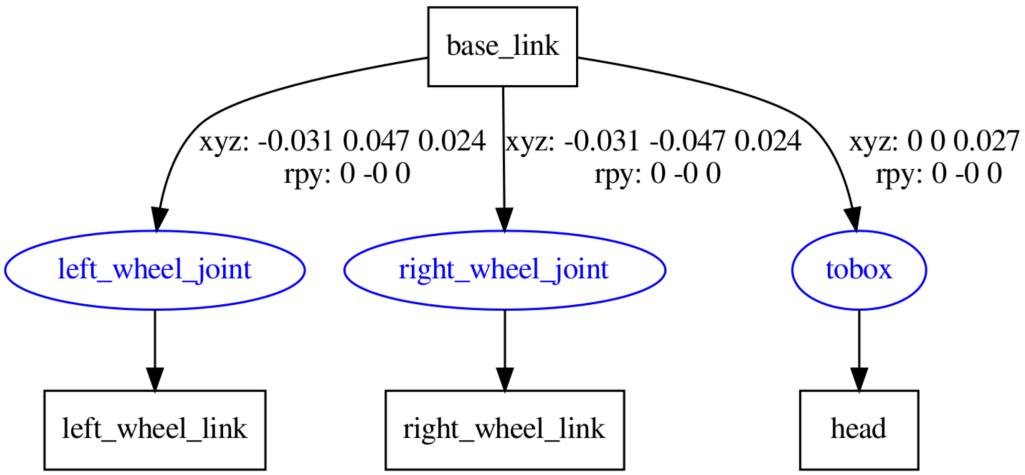

Figure 4: Graphical URDF of robotic

3 Velocity control

3.1 Localization

In this section, we subscribe to the odom topic to get information about the pose of the robot and wait for the update of position command published. There is also a listener that stores incoming goal positions in a list. Upon this, we can get yaw angle from quaternion, and also store current linear and angular velocity in a dynamic list. The positions published by the topic of goal position are also accumulated in a list.

Besides, we initialize the position coordinates of the point of our robot, as well as the initial orientation, linear speed and angular velocity, in accordance with the launch file.

After the system gets information, we will calculate essential parameters for later control. Once there is information about goal position, we calculate the magnitude of distance between current position point and goal position, as well as the angle of orientation.

3.2 Control of Speed

The linear speed is controlled by PID control. To be more specific, we use PD control to update our current linear speed according to the previous speed and distance to the goal position.

$$v_t = K_d \times v_{t-1} + K_p \times d$$

In our code, K_p is defaulted as 0.8 while K_d is defaulted as 0.82. We also set the threshold of the linear velocity to 0.2 since our robot is limited to 0.2m/s. If the absolute of calculated required speed exceed the speed threshold, we reduce it to the threshold.

3.3 Control of Angular Velocity

$$\omega_t = K'_d \times \omega_{t-1} + K'_p \times \nabla\alpha$$

We also use a PD controller to control the angular velocity, with combination of previous angular velocity and angular difference between orientation at the time and the yaw. K'_p is defaulted as 1 while K'_d is defaulted as 0.1. Also, we expect the calculated angular velocity to be within the range of $[-\pi, \pi]$, the value will plus or subtract 2π if it's outside the range. At last, we publish the linear speed and angular velocity as a Twist message to $\mathrm{cmd}_v eltopic$.

3.4 Moving

In the moving of our robot, the task is separated into 2 independent parts described above, linear speed control and angular velocity control. At first we set a threshold of distance to goal (0.05m) and angular differences (0.2rad). If the received value of angular difference exceeds the corresponding threshold, then the corresponding PD controller for the angle is activated to adjust the orientation of the robot. After that the PD control of linear speed take over and adjust the speed. Once the current goal position is approached, we delete the current goal

position and try to move to the next coordinate in the dynamic goal position list. The robot will be stopped when it approaches the final goal position in the list. All this steps of moving are implemented in a control loop so that the control of linear speed and angular velocity will be executed alternately. The detailed implementation in our code can be available in the section of appendix.

4 Discussion

4.1 Result

In the section we will discuss about the result of our algorithm with different plots. The parameters set in our code are described in the previous part. We randomly generate 3 distances to the goal position (small, medium and big) with respect to the size of our robot. The 3 distances are set as follows:

Small distance: position of goal in x-axis is 0.08 and position of goal in y-axis is 0.15.

Medium distance: position of goal in x-axis is 0.30 and position of goal in y-axis is 0.40.

Small distance: position of goal in x-axis is 1.20 and position of goal in y-axis is 1.50.

Note we set the threshold of distance to goal as 0.05, the precision can be improved with a smaller threshold.

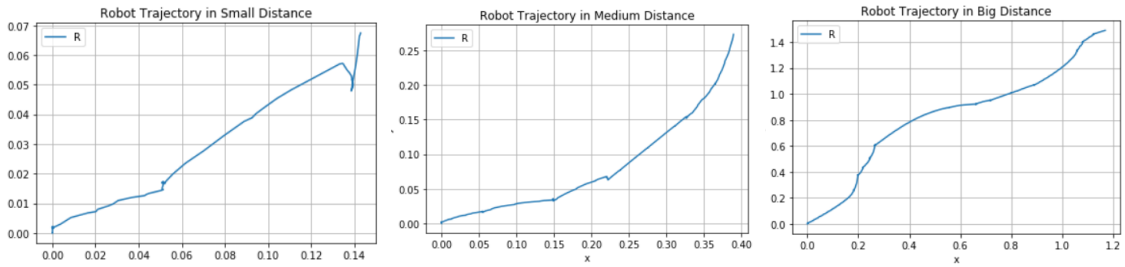

Figure 5: Trajectories of Robot in 3 Different Random Distances(left: small distance, middle: medium distance, right:big distance)

The trajectory plots with respect to different distances show that

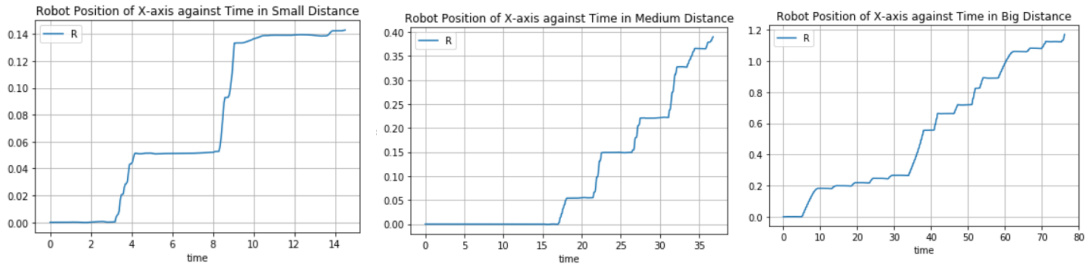

Figure 6: Robotic Position in X-axis against Time in 3 Different Random Distances(left: small distance, middle: medium distance, right:big distance)

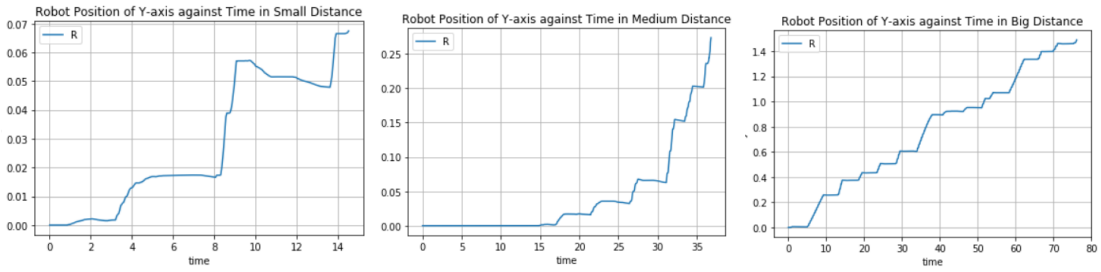

Figure 7: Robotic Position in Y-axis against Time in 3 Different Random Distances(left: small distance, middle: medium distance, right:big distance)

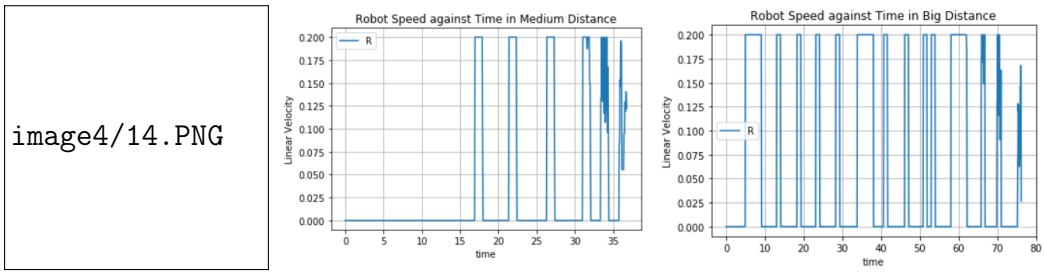

Figure 8: Magnitude of Linear Velocity of Robot against Time in 3 Different Random Distances(left: small distance, middle: medium distance, right:big distance)

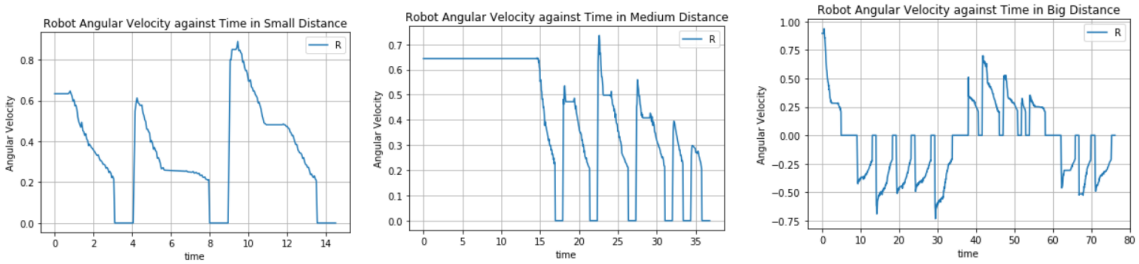

Figure 9: Magnitude of Angular Velocity of Robot against Time in 3 Different Random Distances(left: small distance, middle: medium distance, right:big distance)

Since our algorithm includes switches between 2 kinds of control, the magnitude of linear speed and angular velocity are complementary in time span, which conforms to the observation in Figure 8 and 9. Also, as the robot has a loop of changing orientation and moving forward, the positions on both axis also act like a step change. Finally, the plots show that our algorithm is robust to different ranges of goal distances although it takes some time.

4.2 Other Control Algorithm We've Tried

According to our previous algorithm described above, the control of linear speed and angular velocity are implemented separately. Although the control algorithm can reach acceptable precision, it takes quite a long time to switch between linear and angular control. We have ever tried to implement both control at the same time. That is, when the distance to the goal is larger than a first threshold (set larger), the code implements both linear and angular velocity. When the distance is smaller than the first threshold and larger than the threshold (the final allowed distance for precision), we use the previous algorithm to implement the more precise but slower control.

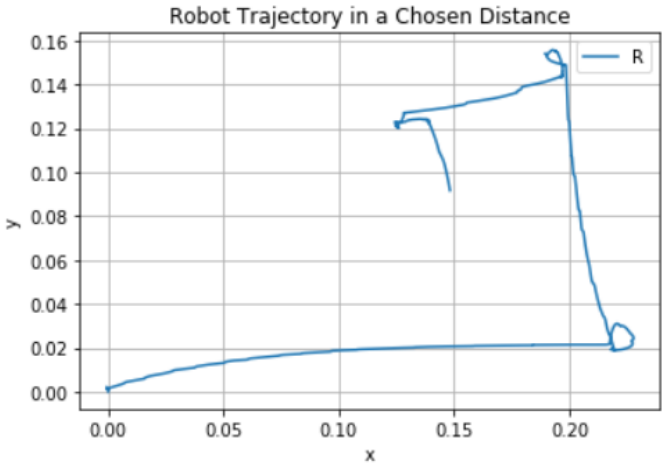

Figure 10: Trajectories of Robot in New Algorithm with a Chosen Goal Distances(x=0.15, y=0.1)

However, we find when implementing both control in our code, the control of linear speed mainly takes over. It does work quite well and saves much time when the distance is large enough or the orientation change is not large. In the case of a small distance, the robot does not have enough time to change the orientation towards the goal position before going too far away. As a result, the trajectory is not robust, which means the robot will go beyond the set goal position and then try to turn back to arrive the goal point, causing overshooting, and this may take even more time in small distance. Figure 10 shows a typical phenomenon of this type of control. Although the algorithm has better performance in bigger distance, we choose our previous algorithm in the final version of code, which suits for different ranges of goal positions.

4.3 Conclusion

In this practical, we get familiar with some basic capacities of ROS. We build a robot model by URDF and visualize it through Gazebo, and it shows that our robot is close to real Thymio robot in terms of kinetics. Also, we design different control algorithms to realize the velocity control of the robot to reach a desired goal point in simulation. After comparison of control algorithms, we make a tradeoff between robustness and time. In our future work,

we will focus on better algorithms with more precise parameter tuning and add more sensor structure to interact with environments, thus achieving more advanced functions.

5 Suggestions

We think this experiment provided good introduction about ROS, which is a very useful tool to control robot, especially for quick prototyping. Since most of us did not have access to ROS, we suggest that the experimental time should be extended appropriately. Two TP sessions are too short, and three to four sessions are more appropriate. We also suggest that installing ROS on the computers in some computer classrooms of EPFL, the installation of Ubuntu and ROS were sometimes time-consuming.

A APPENDIX I: Robot Model Code

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

<robot xmlns:xacro="https://www.ros.org/wiki/xacro" name="G30">

    <xacro:include filename="$(find ros_basics_2019)/urdf/macros.xacro"
                                        />
    <xacro:include filename="$(find ros_basics_2019)/urdf/materials.
                                        xacro" />




    <!-- Design your robot here -->
    <xacro:arg name="mass" default="0.200"/>
    <xacro:property name="mass_p" value="$(arg mass)"/>
    <xacro:property name="body_mass" value="${mass_p * 0.80}"/>
    <xacro:property name="wheel_mass" value="${mass_p * 0.10}"/>
    <xacro:property name="wheel_rad" value="0.024"/>
    <xacro:property name="wheel_width" value="0.020"/>


    <link name="base_link">
        <inertial>
            <mass value="${body_mass}"/>
            <xacro:box_inertia m="${body_mass}" x="0.055" y="0.112" z="
                                                0.045" />
        </inertial>

        <collision name="collision body">
            <origin xyz="${-0.11*0.25} 0 0.027"/>
            <geometry>
                <box size="0.055 0.112 0.045"/>
            </geometry>
        </collision>

        <visual name="visual body">
            <origin xyz="${-0.11*0.25} 0 ${0.045 - 2*0.009}"/>
            <geometry>
                <box size="0.055 0.112 0.045"/>
            </geometry>
            <material name="white"/>
        </visual>

        <collision name="collision caster">
            <origin xyz="0.035 0 0.009"/>
            <geometry>
                <sphere radius="0.009"/>
            </geometry>
        </collision>
```

```
    <visual name="visual caster">
        <origin xyz="0.045 0 0.009"/>
        <geometry>
            <sphere radius="0.009"/>
        </geometry>
        <material name="white"/>
    </visual>
</link>

<joint name="right_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="right_wheel_link"/>
    <axis xyz = "0 1 0"/>
    <origin xyz = "${- 0.11/2 + wheel_rad } -0.047 ${wheel_rad}"/>
</joint>

<link name="right_wheel_link">
    <inertial>
        <mass value="${wheel_mass}"/>
        <xacro:cylinder_inertia m="${wheel_mass}" r="${wheel_rad}"
                                              h="${wheel_width}"/>
    </inertial>

    <collision name="collision right_wheel">
        <origin xyz="0 0 0"/>
        <geometry>
            <cylinder radius="${wheel_rad}" length="${wheel_width}"
                                                  />
        </geometry>
    </collision>

    <visual name="visual right_wheel">
        <origin xyz="0 0 0" rpy="1.57075 0 0"/>
        <geometry>
            <cylinder radius="${wheel_rad}" length="${wheel_width}"
                                                  />
        </geometry>
        <material name="red"/>
    </visual>

</link>

<joint name="left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel_link"/>
    <axis xyz = "0 1 0"/>
    <origin xyz = "${- 0.11/2 + wheel_rad } 0.047 ${wheel_rad}"/>
</joint>

<link name="left_wheel_link">
    <inertial>
```

```
            <mass value="${wheel_mass}"/>
            <xacro:cylinder_inertia m="${wheel_mass}" r="${wheel_rad}"
                                                    h="${wheel_width}"/>
        </inertial>

        <collision name="collision left_wheel">
            <origin xyz="0 0 0"/>
            <geometry>
                <cylinder radius="${wheel_rad}" length="${wheel_width}"
                                                        />
            </geometry>
        </collision>

        <visual name="visual left_wheel">
            <origin xyz="0 0 0" rpy="1.57075 0 0"/>
            <geometry>
                <cylinder radius="${wheel_rad}" length="${wheel_width}"
                                                        />
            </geometry>
            <material name="yellow"/>
        </visual>

    </link>



<link name="head">
    <visual>
      <geometry>
   <cylinder length="0.045" radius="0.055"/>
      </geometry>
      <material name="white">
    <color rgba="1 1 1 0.5"/>
      </material>
    </visual>
</link>


    <joint name="tobox" type="fixed">
      <parent link="base_link"/>
      <child link="head"/>
      <origin xyz="0 0 0.027"/>
    </joint>

    <!-- Below you will find samples of gazebo plugins you may want to
                                        use. -->
    <!-- These should be adapted to your robot's design -->
    <gazebo reference="YOUR_SENSOR_LINK">
        <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>-1.571</min_angle>
                        <max_angle>1.571</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <!-- You can edit adapt these to your robot's size
                                                            -->
                    <min>0.0005</min>
                    <max>0.04</max>
                    <resolution>0.0001</resolution>
                </range>
            </ray>
            <plugin name="laser" filename="libgazebo_ros_laser.so" >
                <topicName>YOUR_ROBOT_LASER/scan</topicName>
                <frameName>YOUR_ROBOT_LASER_LINK</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>0.094</wheelSeparation>
            <wheelDiameter>0.044</wheelDiameter>
            <!-- <wheelTorque>ZZZ</wheelTorque> -->

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

            <robotBaseFrame>base_link</robotBaseFrame>
        </plugin>
    </gazebo>

</robot>
```

B APPENDIX II: Control Algorithm Code.

B.1 Talker

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Point,Pose

def talker():
  pos=rospy.Publisher('goal_pos',Point,queue_size=5)
  rospy.init_node('talker',anonymous=True)
  rate = rospy.Rate(5)
  goal_poses = [[5,5]]

  #while not rospy.is_shutdown():
  for goal_pos in goal_poses:
    point = Point()
    point.x = goal_pos[0]
    point.y = goal_pos[1]
    point.z = 0
    pos.publish(point)
    rate.sleep()

  rospy.spin()

if __name__ == '__main__':
  talker()
```

B.2 Control

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

import rospy
from tf.transformations import euler_from_quaternion
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist,Point,Pose
from math import atan2, pow, sqrt, pi

class velocityControl:

    def __init__(self):
  self.goalPoslist = []
  point = Point()
  # used for testing
        point.x = 0
        point.y = 0
  self.goalPoslist.append(point)

        rospy.init_node('velocity_controller_G30', anonymous=True)
```

```
        self.goal_pos_sub = rospy.Subscriber('/goal_pos', Point, self.
                                                stackPos)
        self.current_pose_sub = rospy.Subscriber('/odom', Odometry,
                                                self.newPosition)
        self.velocity_pub = rospy.Publisher('/cmd_vel',Twist,queue_size
                                                = 10)
#to initialize
        self.pos = Point()
        self.pos.x = 0
        self.pos.y = 0
        self.pos.z = 0.05 # in accordance with the launch file
        self.yaw = 0
self.vx = 0
self.angVel = 0
        self.rate = rospy.Rate(20)

    def stackPos(self, newPos):
#to accumulate the positions published by the topic of /goal_pos
        self.goalPoslist.append(newPos)

    def newPosition(self, odom):

        # geometry_msgs::Quaternion odom_quat = tf::
                                                createQuaternionMsgFromYaw(
                                                th);
        # odom.pose.pose.orientation = odom_quat;
        self.pos.x = odom.pose.pose.position.x
        self.pos.y = odom.pose.pose.position.y
        self.pos.z = odom.pose.pose.position.z
        orientation = odom.pose.pose.orientation
# got yaw angle from quaternion
        (_, _, self.yaw) = euler_from_quaternion([orientation.x,
                                                orientation.y,orientation.z
                                                ,orientation.w])
# store current linear velocity
self.vx = odom.twist.twist.linear.x
# store current angular velcoity
self.angVel = odom.twist.twist.angular.z;


    def distance(self, final):
        dist = pow((final.x - self.pos.x), 2) + pow((final.y - self.pos
                                                .y), 2)
        dist = sqrt(dist)
        return dist

    #PD control
    def velPID(self, final, kp = 0.82, kd = 0.8):
velocity = kd * self.vx + kp * self.distance(final)
#saturate
if(velocity > 0.2):
    velocity = 0.2
```

```
elif(velocity < -0.2):
    velocity = -0.2
      return velocity


  def orientation_angle(self, goal):
# orientation angle of current position and goal position
      angle = atan2((goal.y - self.pos.y),(goal.x - self.pos.x))
      #angle %= 2*pi
while abs(angle) > pi:
    if(angle > pi):
        angle -= 2 * pi
    elif(angle < -pi):
        angle += 2 * pi

return angle #range[-pi, pi]

  #PD controller
  def anglePID(self, goal, kd = 0.1, kp = 1):

angle_difference = self.orientation_angle(goal) - self.yaw
angle_difference = self.clip(angle_difference)
angular_vel = kd * self.angVel + kp * angle_difference
print(angle_difference)
      return angular_vel

  def clip(self,angle):
while abs(angle) > pi:
    if(angle > pi):
        angle -= 2 * pi
    elif(angle < -pi):
        angle += 2 * pi
return angle


  def move(self):

      distance_tol = 0.05
      angular_tol = 0.20 #0.08
      # define it with Twist type
      vel_control = Twist()

      while(len(self.goalPoslist)>0):

          while self.distance(self.goalPoslist[0]) > distance_tol:

              vel_control.angular.z = self.yaw

  while abs(self.clip(self.yaw - self.orientation_angle(self.
                                  goalPoslist[0]))) > angular_tol:
```

```
                    vel_control.angular.z =self.anglePID(self.
                                                goalPoslist[0])
                    vel_control.linear.x = 0 # here we seperate the
                                                task into 2
                                                parts, so we
                                                did not assign
                                                linear velocity
                    vel_control.linear.y = 0
                    vel_control.linear.z = 0

                    print(self.pos)
                    self.velocity_pub.publish(vel_control)
                    self.rate.sleep()

    print(self.pos)
                # to adjust velocity
                vel_control.linear.x = self.velPID(self.goalPoslist[0])
                vel_control.linear.y = 0
                vel_control.linear.z = 0
                self.velocity_pub.publish(vel_control)
                self.rate.sleep()
            #when approached, delete it
      self.goalPoslist.pop(0)

        # stop the vehcle when it approachs the goal position
        vel_control.linear.x = 0
        vel_control.angular.z = 0
        self.velocity_pub.publish(vel_control)
  self.rate.sleep()
  #rospy.spin()

    def loop2approach(self):
        while(True):
      self.move()

if __name__ == '__main__':

    thymio = velocityControl()
    thymio.loop2approach()
```

C APPENDIX III: provide instructions for launching examples

Our work was done with the ROS Melodic Morenia distribution in virtual enviroment. To launch our program, you should type the following command in the terminal.

```
$ roscore
```

```
$ source devel/setup.sh
```

```
$ roslaunch ros_basics_2019 robot_description_gazebo.launch
```

```
$ rosrun ros_basics_2019 velocity_control.py
```

To set the desired goal position for the testing, for example, to set the coordinates to (1,1), you can either change the parameter in line 15 and 16 of our original code in velocity_control.py (directory: <working directory>/src/ros_basics_2019/src). In the code, you can also try other paramrters like threshold and gain of PID control.

```
        point.x = 0
        point.y = 0
```

or

type the following command in the terminal:

```
$ rostopic pub /goal\_pos geometry\_msgs/Point "x: 1.0
y: 1.0
z: 0.0"
```