RBE 500 — Foundations of Robotics

Instructor: Siavash Farzan

Spring 2022

ROS Assignment 4: Closed Loop Control of Mobile Robots

4.1 Objectives

By the end of this assignment you should be able to:

- Make a TurtleBot (as a differential drive mobile robot) move in Gazebo using Twist messages
- Implement a PD controller for the mobile robot to track a trajectory
- Read the odometry messages and store the traversed trajectory for further analysis

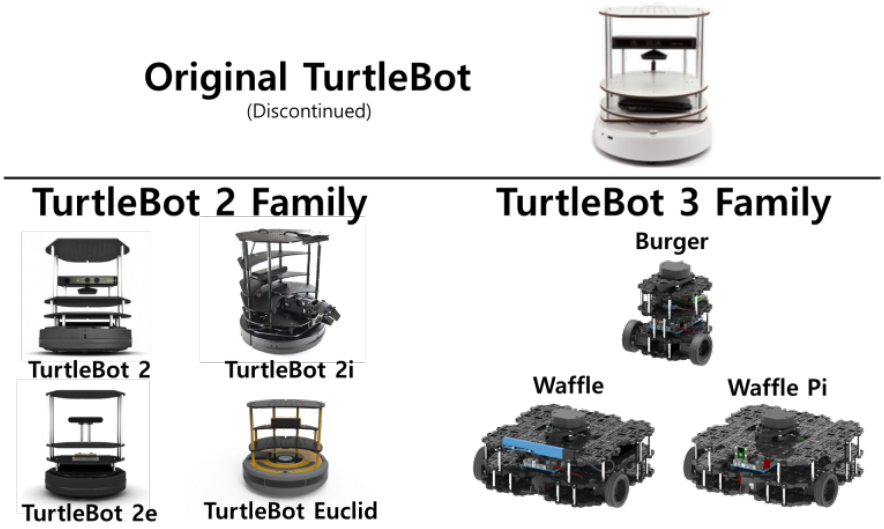
4.2 Problem Statement

In this assignment, we will start working on mobile robots*. Specifically, we are going to apply closed-loop control to track a desired trajectory by a two-wheeled mobile robot. A ROS Subscriber is included in the script to receive real-time odometry feedback from Gazebo simulator. With the `odom/pose` updates, the robot can adjust its moving direction accordingly and check if the desired waypoint has been reached.

The task is to implement a PD controller to make the robot track a *square shape* trajectory. The controller is a *velocity control*, that is, the control commands (inputs) are the linear and angular velocity of the robot (v, ω), while the desired reference provided to the controller are (x_d, y_d) position coordinates. The waypoints to visit are [4, 0], [4, 4], [0, 4] and [0, 0], and the sequence does matter. In other words, the robot should move forward 4 meters, turn left 90 degrees, move forward again 4 meters, and so on, until going back to the origin. Note that the robot is supposed to stop at the origin after completing this square movement, and the Python script should exit gracefully. The robot must complete the task in *under 4 minutes* (the reasonable time is about two minutes).

Note: It is required to use *closed-loop control* (i.e. a PD controller for the robot orientation ϕ, and at least a simple proportional controller for the robot distance) to track the trajectory. Finely tuned open-loop control script may also achieve the task, but is not allowed. All scripts will be double checked when grading manually. Penalty will apply if such script is found.

*Parts of this assignment are adapted from the course materials by Professor K. Karydis at UC Riverside.



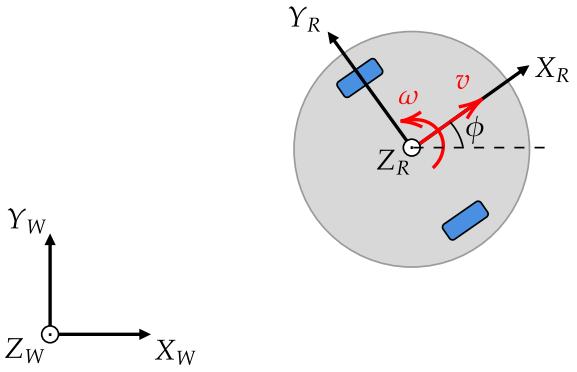
4.3 Introduction to TurtleBot

The TurtleBot3 robot is a differential wheeled robot which adopts ROBOTIS smart actuator DYNAMIXEL for driving. TurtleBot3 has three models, Burger, Waffle, and Waffle Pi, and can be customized to include more sensors (LiDars, cameras, manipulators, etc.). See the figure above for TurtleBot family.

Though multiple layers of plates/sensors can be placed on top of the robot, the kinematics of the robot can be simplified according to the property of its mobile base, which uses differential drive for locomotion. The differential mobile base has two powered wheels, located symmetrically about its center.

As users, we can send high-level commands (linear velocity v and angular velocity ω) to the robot. The mobile base will first transform the v and ω commands with respect to the robot center into the desired rotational speed of each wheel, and then control the rotational speed by a feedback control of the current of the motor that drives the wheel.

To describe the position and orientation of the robot, we attach a robot coordinate frame to it. The origin of this coordinate frame is centered between its powered wheels. The X axis of this frame is pointing forward (along the direction of the linear velocity v), the Y axis is pointing to the left, and the Z axis is pointing up. This is shown in the figure below.



4.4 Set Up TurtleBot3 in Gazebo

- First let's download and install the core ROS packages for TurtleBot3.

```
cd ~/rbe500_ros/src
git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git -b noetic-
    devel
git clone https://github.com/ROBOTIS-GIT/turtlebot3.git -b noetic-devel
cd ~/rbe500_ros
catkin_make
```

- After the correct compilation of the catkin workspace, we can download and install the TurtleBot simulator package:

```
cd ~/rbe500_ros/src
git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git -b
    noetic-devel
cd ~/rbe500_ros
catkin_make
```

- To develop our turtlebot control program, open a new terminal and go to the `rbe500_ros` workspace. We will start from a new ROS package.

```
cd ~/rbe500_ros/src
catkin_create_pkg mobilerobot_control rospy
```

- Create a new `launch` directory under the ROS package for the launch files.
  Copy and paste the script below into a new file named `gazebo.launch` inside this directory.

```
<launch>
    <arg name="model" default="burger" doc="model type [burger, waffle,
        waffle_pi]"/>
    <arg name="x_pos" default="0.0"/>
    <arg name="y_pos" default="0.0"/>
    <arg name="z_pos" default="0.0"/>

    <!-- Gazebo model spawner -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
      <arg name="world_name" value="$(find turtlebot3_gazebo)/worlds/
          empty.world"/>
      <arg name="paused" value="false"/>
      <arg name="use_sim_time" value="true"/>
      <arg name="gui" value="true"/>
      <arg name="headless" value="false"/>
      <arg name="debug" value="false"/>
    </include>

    <param name="robot_description" command="$(find xacro)/xacro --
        inorder $(find turtlebot3_description)/urdf/turtlebot3_$(arg model
        ).urdf.xacro" />

    <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-
        urdf -model turtlebot3_$(arg model) -x $(arg x_pos) -y $(arg y_pos
        ) -z $(arg z_pos) -param robot_description" />
```

```
    <!-- Publish robot state -->
    <node pkg="robot_state_publisher" type="robot_state_publisher" name="
        robot_state_publisher">
         <param name="publish_frequency" type="double" value="30.0" />
    </node>
</launch>
```

- To run the TurtleBot in Gazebo, launch the Gazebo simulator and spawn a new robot by the following command. It may take a while at the first time you open Gazebo, since it will need to download some models and world environments.

```
roslaunch mobilerobot_control gazebo.launch
```

- **Note:** If you experienced graphic issues in Gazebo, please run the following command for once. Then close all terminals and try again.

```
echo "export SVGA_VGPU10=0" >> ~/.bashrc
```

If the issue persists, please shutdown your VM, go to VM settings and allocate more resources (Processor Cores, Memory, Graphics Memory). If the issue still persists, please disable "3D Acceleration" in Display settings.

4.5 Programming Tips

i) We follow ROS conventions to use SI units. (i.e. length in meter, time in second, angle in radian). See ROS Wiki article REP 103 Standard Units of Measure and Coordinate Conventions for more information.

ii) In Gazebo, you can use `Ctrl + R` to set the robot back to the origin without the need to relaunch.

iii) The grids in Gazebo environment have the dimension of 1m x 1m. In Gazebo, you can take the visualization as feedback to tune the parameters.

iv) Note that the orientation of the robot (ϕ) ranges from $-\pi$ to π on 2D plane. When the robot turns in CCW direction and passes the direction of negative x axis, the value of ϕ will jump from π to $-\pi$. This needs to be handled properly, otherwise the robot will keep turning in place and cannot move forward.

v) The orientation of the robot (angle ϕ) is denoted by the angle θ in ROS Odometry messages.

vi) In general, PID controllers are used to track a certain target value (called setpoint), and make sure the system can converge to this target value. Note that the setpoint is a scalar, set to a certain target value.

vii) In our case, we have three variables (x, y, ϕ) to describe the 2D pose of the robot. To complete the task in this assignment, a PD controller to track ϕ is required. A simple proportional controller can be used to set the control velocity of the robot and reach a waypoint. The following is an example of how to apply feedback control algorithm to waypoint navigation problem. You may follow this algorithm to start your implementation (please feel free and encouraged to implement more advanced algorithms):

- Suppose the robot's current orientation is ϕ, the desired orientation is ϕ_d, the current position is (x, y), and the desired position is $(x_d,\ y_d)$.
- Calculate the moving direction from the difference between (x, y) and (x_d, y_d); set it as the desired orientation ϕ_d.
- Initialize a PD controller with the setpoint ϕ_d and a set of parameters K_p, K_d. Adjust the angle according to the angular velocity computed by the PD controller (while keeping the linear velocity as 0).
- Once $\phi \to \phi_d$, start moving forward at a constant speed (or using a linear velocity generated by another P or PD controller), and keep adjusting the angle (by the same PD controller designed for ϕ) and checking the remaining distance toward desired position.
- Once $(x, y) \to (x_d, y_d)$, stop and repeat the process for the next waypoint.

viii) To implement the derivative of the error for the PD controller in the code, we can replace derivative with subtraction (since it is a discrete systems):

$$u(t) = K_p e(t) + K_d \frac{e(t) - e(t - \Delta t)}{\Delta t}$$

Also, the time interval Δt can be merged into parameter K_d (If we run at 10Hz, the time interval will be 0.1 second). Therefore, we have the following equation ready, which is what you need to implement in this assignment:

$$u(t) = K_p e(t) + K_d\big(e(t) - e(t - \Delta t)\big)$$

ix) A discrete PD controller implementation in Python is provided for your information (you may or may not use it in your own implementation). To make it work, you need to understand the PD control algorithm and complete the code under the update function.

```
class Controller:
    def __init__(self, P=0.0, D=0.0, set_point=0):
        self.Kp = P
        self.Kd = D
        self.set_point = set_point # reference (desired value)
        self.previous_error = 0

    def update(self, current_value):
        # calculate P_term and D_term
        error =
        P_term =
        D_term =
        self.previous_error = error
        return P_term + D_term

    def setPoint(self, set_point):
        self.set_point = set_point
        self.previous_error = 0

    def setPD(self, P=0.0, D=0.0):
        self.Kp = P
        self.Kd = D
```

4.6 Sample Code

A sample code is provided as the starting point for your implementation. Please read carefully the provided code, and understand its functionality.

Open a new terminal and go to the ROS package created in Section 4.4. Create a `scripts` directory under the ROS package. We will start from a new python script.

```
roscd mobilerobot_control
mkdir scripts
cd scripts
touch closed_loop.py
chmod +x closed_loop.py
subl closed_loop.py
```

Copy and paste the following code into `closed_loop.py`, then save and close it.

```
#!/usr/bin/env python3

from math import pi, sqrt, atan2, cos, sin
import numpy as np

import rospy
import tf
from std_msgs.msg import Empty
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose2D

class Turtlebot():
    def __init__(self):
        rospy.init_node("turtlebot_move")
        rospy.loginfo("Press Ctrl + C to terminate")
        self.vel = Twist()
        self.vel_pub = rospy.Publisher("/cmd_vel", Twist,queue_size=10)
        self.rate = rospy.Rate(10)

        # reset odometry to zero
        self.reset_pub = rospy.Publisher("/reset", Empty, queue_size=10)
        for i in range(10):
            self.reset_pub.publish(Empty())
            self.rate.sleep()

        # subscribe to odometry
        self.pose = Pose2D()
        self.logging_counter = 0
        self.trajectory = list()
        self.odom_sub = rospy.Subscriber("odom", Odometry, self.odom_callback)

        try:
            self.run()
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
        finally:
            # save trajectory into csv file
```

```
            np.savetxt('trajectory.csv', np.array(self.trajectory), fmt='%f',
                delimiter=',')

    def run(self):
        self.vel.linear.x = 0.5
        self.vel.angular.z = 0
        self.vel_pub.publish(self.vel)
        # remove the code above and add your code here to adjust your movement
            based on 2D pose feedback

    def odom_callback(self, msg):
        # get pose = (x, y, theta) from odometry topic
        quarternion = [msg.pose.pose.orientation.x, \
                       msg.pose.pose.orientation.y, \
                       msg.pose.pose.orientation.z, \
                       msg.pose.pose.orientation.w]
        (roll, pitch, yaw) = tf.transformations.euler_from_quaternion(
            quarternion)
        self.pose.theta = yaw
        self.pose.x = msg.pose.pose.position.x
        self.pose.y = msg.pose.pose.position.y

        # logging once every 100 times
        self.logging_counter += 1
        if self.logging_counter == 100:
            self.logging_counter = 0
            # save trajectory
            self.trajectory.append([self.pose.x, self.pose.y])
            rospy.loginfo("odom: x=" + str(self.pose.x) +\
                ";  y=" + str(self.pose.y) + ";  theta=" + str(yaw))

if __name__ == '__main__':
    whatever = Turtlebot()
```

The sample code makes the robot move forward for a certain distance. Make changes to the `run` function and also implement the PD controller class to complete the task in this assignment.

4.7 Performance Testing

i) Open a terminal and launch your TurtleBot robot in Gazebo before running the script.

```
roslaunch mobilerobot_control gazebo.launch
```

ii) Once the robot is successfully spawned in Gazebo, we can test the scripts by running the `closed_loop.py` script in another terminal.

```
roscd mobilerobot_control/scripts
python3 closed_loop.py
```

iii) Once the program is complete, the trajectory is saved into a CSV file under the `scripts` directory. We provide a separate Python script to help visualize the trajectory from the saved `trajectory.csv` file. Copy and paste the following code into a new script named `visualize.py`:

```python
#!/usr/bin/env python3

import numpy as np
import matplotlib.pyplot as plt

def visualization():
    # load csv file and plot trajectory
    _, ax = plt.subplots(1)
    ax.set_aspect('equal')

    trajectory = np.loadtxt("trajectory.csv", delimiter=',')
    plt.plot(trajectory[:, 0], trajectory[:, 1], linewidth=2)

    plt.xlim(-1, 5)
    plt.ylim(-1, 5)
    plt.minorticks_on()
    plt.grid(which='both')
    plt.xlabel('x (m)')
    plt.ylabel('y (m)')
    plt.show()

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

You can then execute the scripts in the terminal by:

```
python3 visualize.py
```

which would generate a plot of the trajectory. Remember to save the resulting plot as a figure to be included in your report.

4.8 Submission

- Submission: individual submission via Gradescope.
- Due time: as specified on Canvas
- Files to submit: (please use exactly the same filename)
  - `closed_loop.py`
  - `ros4_report.pdf`
- Grading rubric:
  - 50%: implement PD controller; visit four vertices of the square trajectory with error $< 0.1\,m$. Partial credits will be given according to the number of vertices visited.
  - 10%: the script can complete the task on time and exit gracefully.
  - 20%: clearly describe your approach and explain your code in the report.
  - 20%: plot the trajectory in your report and discuss the results of different values of K_p and K_d.

4.9 Autograder

All code submissions will be graded automatically by an autograder uploaded to Gradescope. Your scripts will be tested on a Ubuntu cloud server using a similar ROS + Gazebo environment. The grading results will be available in a couple of minutes after submission.

Testing parameters are as follows.

- The tolerance for distance error is set to $0.1\,m$ (considering that this is closed-loop control).
  - For example, passing point [3.96, 3.94] is approximately equivalent to passing point [4, 4].
- The required waypoints for square trajectory are [4, 0], [4, 4], [0, 4] and [0, 0]; sequence matters.
- The time limit for the submitted script is set to 4 minutes.
  - If running properly, the task in this lab can be done in about 2 minutes, based on my testing.
  - If running timeout, the script will be terminated and a 10% penalty will apply.