University of California San Diego

CSE276A: Introduction to Robotics

HW1: Open-loop Control for the Omnidirectional Robot

October 20, 2023

Author	Student ID
Mazeyu Ji	A59023027
Zejia Wu	A59026752

1 Robot Calibration

In order to calibrate the four wheels of the robot, we employ the following model to characterize the mapping relationship between input *speed* parameters and the actual angular velocities ω of the wheels, as we observed the presence of dead zones, where the wheels do not rotate for small input *speed* parameters.

$$\omega = \begin{cases} a \times speed, & if \quad |speed| > s_0, \\ 0, & if \quad |speed| \leq s_0. \end{cases} \tag{1}$$

We affixed tape markers to the four wheels and recorded videos of wheel rotation under different input *speed* parameters using a mobile phone. By counting the number of wheel revolutions in slow-motion and dividing by the corresponding time, we obtained $(speed, \omega)$ data points. We individually fitted our model to the data points from four wheels, resulting in the calibration parameters. Results are shown in Figure 1.

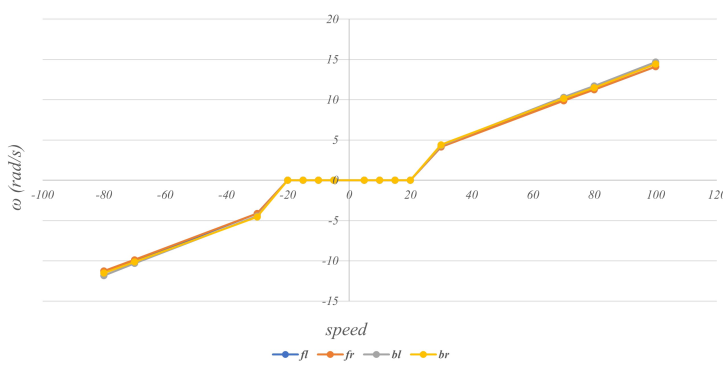

Figure 1: The fitting results of the calibration model.

2 Kinematic Model

For the kinematic model of the four Mecanum wheeled mobile robot, we followed the derivation in [1] and use the following model to mapping the angle velocities of wheels to the velocity and angle velocity of the robot,

$$\begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix} = \frac{r}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -\frac{1}{(l_x+l_y)} & \frac{1}{(l_x+l_y)} & -\frac{1}{(l_x+l_y)} & \frac{1}{(l_x+l_y)} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix}, \tag{2}$$

where r is the radius of the wheels, l_x is half of the distance between front wheels, and l_y is half of the distance between front wheel and the rear wheels. By conducting multiple measurements of wheel speeds and the forward and rotational velocities of the robot, we can solve the values of r and $l_x + l_y$. We use $l_x + l_y = 0.1354$ and $r = 0.0305$ in our code.

In the robot control process, we employ the inverse kinematic model. Firstly, we use the rotation matrix to convert velocity from the world coordinate system to the robot coordinate system.

$$\begin{bmatrix} v_x^{robot} \\ v_y^{robot} \\ \omega_z^{robot} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x^{world} \\ v_y^{world} \\ \omega_z^{world} \end{bmatrix}, \tag{3}$$

where the (x, y, θ) is the pose of robot, and $(v_x^{world}, v_y^{world}, \omega_z^{world})$ is its velocity in the world coordinate system. Then we use the calibrated inverse kinematic model to map the velocity of the robot in its local coordinate system to the input parameters *speed* of the four wheels:

$$\begin{bmatrix} a_1 \times speed_1 \\ a_2 \times speed_2 \\ a_3 \times speed_3 \\ a_4 \times speed_4 \end{bmatrix} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & -1 & -(l_x + l_y) \\ 1 & 1 & (l_x + l_y) \\ 1 & 1 & -(l_x + l_y) \\ 1 & -1 & (l_x + l_y) \end{bmatrix} \begin{bmatrix} v_x^{robot} \\ v_y^{robot} \\ \omega_z^{robot} \end{bmatrix}. \tag{4}$$

3 Control Model

The direct control variable is voltage, as it exhibits a reasonably linear relationship with torque, imparting a second-order system characteristic to the robot's control. However, due to the open-loop nature of the system, error values are unattainable, and the determination of load and friction is also a challenge. Given these circumstances, we assume that we directly control velocity, taking into account a dead zone. Velocity, in this context, maintains a predominantly linear relationship with control parameters. With this setup, our input commands directly control velocity, allowing the car to stop at the target position at any time. As a result, the system operates with negligible steady-state error and free from oscillations.

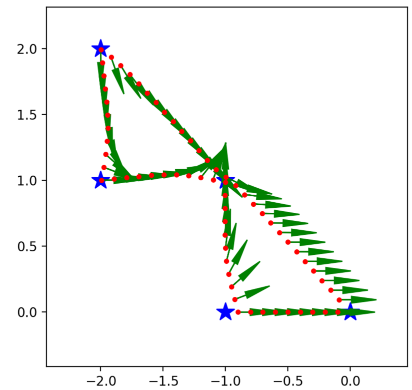

Figure 2: Simulation results of our control algorithm, where red dots represent the trajectory at each control step, and green arrows indicate the orientation of the robot.

Hence, our system relies exclusively on proportional control, which means that the output is proportional to the error. Specifically, for a simple mobile robot moving from point (x_1, y_1, θ_1) to point (x_2, y_2, θ_2), we can define **position errors** $\Delta x = x_2 - x_1, \quad \Delta y = y_2 - y_1$, and **angle error** $\Delta\theta = \theta_2 - \theta_1$. Then we define the velocity control input $v = k_v\sqrt{\Delta x^2 + \Delta y^2}$ and steering control input $\omega = k_\theta \Delta\theta$, where k_v and k_θ are the proportional gains. In this definition, the velocities are proportional to the distance and the angular error, which means that the velocity or angular velocity will be greater when the robot is far from the target or the angular error is large. As it gets closer in distance and angle, the velocity and angular velocity decrease.

Since the control process is open-loop, it is essentially a simulation of the control result. We assume a time interval Δt between two control signals. The velocity in the robot coordinate system remains constant during this period, while it changes in the world coordinate. In our code, we approximate the motion in a even smaller time intervals δt. By summing up the values happened in every δt, we obtain the overall change during Δt. This improves the accuracy of the simulation. The simulation results are shown in the Figure 2.

4 Real-world Results and Performance

We have recorded a video of the motion of the robot:

`https://drive.google.com/file/d/163lTISibHs6sfYPuCxchMxWNJBUnej2g/view?usp=sharing`

From the video, we can observe that due to the limitations of the open-loop control process, there is a significant amount of error in the actual robot running, preventing the robot from returning perfectly to the starting point. However, the trajectory of the robot approximately passes through the required waypoints, which aligns with our simulation results, demonstrating the effectiveness of our algorithm. The final stopping position of the robot exhibited an approximate 0.4 unit error in the negative y-axis direction relative to the starting point, while maintaining the same orientation.

References

[1] H. Taheri, B. Qiao, and N. Ghaeminezhad, "Kinematic model of a four mecanum wheeled mobile robot," *International journal of computer applications*, vol. 113, no. 3, pp. 6–9, 2015.

Appendix

code of mpi_openloop_ctrl.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import math
import time
import matplotlib.pyplot as plt
import numpy as np
from mpi_control import MegaPiController
import rospy
from std_msgs.msg import String


# Robot Simulator (Omnidirectional)
class RobotSimulator:
    def __init__(self, x=0, y=0, theta=0, dt=0.05,
                 k_rho=2.0, k_alpha=1.0, k_beta=1.0,
                 v_max=0.30, omega_max=40 * math.pi / 180,
                 v_min=0.10, omega_min=0 * math.pi / 180,
                 target_range_x=0.20, target_range_y=0.20, target_range_theta=0.3,
                 l=0.1354, r=0.0305, calibration_parameters=100/14.1124):
        # time step
        self.dt = dt
        # pose
        self.x = x
        self.y = y
        self.theta = theta
        # velocity(world frame)
        self.vx = 0
        self.vy = 0
        self.omega = 0
        # control parameters
        # k_alpha equals to k_beta because of the omnidirection driver
        self.k_rho = k_rho
        self.k_alpha = k_alpha
        self.k_beta = k_beta
        # velocity limit
        self.v_max = v_max
        self.omega_max = omega_max
        self.v_min = v_min
        self.omega_min = omega_min
        # parameters of the robot
        self.l = l # l_x + l_y
        self.r = r # wheel radius
        # matrix for inverse kinematics
        self.A = np.array([[1, -1, -1*l],[1, 1, l],[1, 1, -1*l],[1, -1, l]]) / r
        # parameters map real wheel velocities to input control signal
        self.calibration_parameters = calibration_parameters
        # range parameters to demetermine if the robot arrives the target pose
        self.target_range_x = target_range_x
        self.target_range_y = target_range_y
        self.target_range_theta = target_range_theta

    def move(self, vx, vy, omega):
        dt = self.dt
        # Convert velocity to robot coordinate system
        v = np.array([vx, vy])
        R = np.array([[math.cos(self.theta), -math.sin(self.theta)],
                      [math.sin(self.theta), math.cos(self.theta)]])
```

```
        R = np.linalg.inv(R)
        v_robot = np.dot(R, v)

        # update velocities
        self.vx = v_robot[0]
        self.vy = v_robot[1]
        self.omega = omega

        # # update pose
        # self.x += vx * dt
        # self.y += vy * dt
        # self.theta += omega * dt

        # # Normalize theta to be within [-2\pi, 2\pi]
        # if self.theta > 2 * math.pi:
        #     self.theta -= 2 * math.pi
        # elif self.theta < -2 * math.pi:
        #     self.theta += 2 * math.pi

        # Use a smaller dt to simulate the integration of the real pose update
        division = 100
        for _ in range(division):
            # update position
            R = np.array([[math.cos(self.theta), -math.sin(self.theta)],
                            [math.sin(self.theta), math.cos(self.theta)]])
            v_robot_real = np.dot(R, v_robot)
            self.x += v_robot_real[0] * dt / division
            self.y += v_robot_real[1] * dt / division

            # uodate direction
            self.theta += self.omega * dt / division
            # Normalize theta to be within [-2\pi, 2\pi]
            if self.theta > 2 * math.pi:
                self.theta -= 2 * math.pi
            elif self.theta < -2 * math.pi:
                self.theta += 2 * math.pi


    def move_to(self, goal_x, goal_y, goal_theta):
        error_x = goal_x - self.x
        error_y = goal_y - self.y
        direction = math.atan2(error_y, error_x)

        # The angle between the current orientation and
        # the straight line direction to the target position
        alpha = direction - self.theta
        # The angle between the straight line direction to
        # the target position and the target orientation
        beta = goal_theta - direction
        # Distance to the target position
        rho = math.sqrt(error_x ** 2 + error_y ** 2)

        # Calculate control input
        v = self.k_rho * rho
        omega = self.k_alpha * alpha + self.k_beta * beta
        # Limit speed
        if abs(v) > self.v_max:
            v = self.v_max if v > 0 else -self.v_max
        elif abs(v) < self.v_min:
            v = self.v_min if v > 0 else -self.v_min
        if abs(omega) > self.omega_max:
            omega = self.omega_max if omega > 0 else -self.omega_max
        elif abs(omega) < self.omega_min:
            omega = self.omega_min if omega > 0 else -self.omega_min

        vx = v * math.cos(direction)
        vy = v * math.sin(direction)

```

```
        # move the robot
        self.move(vx, vy, omega)

    def get_pose(self):
        return self.x, self.y, self.theta

    def get_velocity(self):
        return self.vx, self.vy, self.omega

    def _visualize_simulation(self, ax, robot, goal_x, goal_y, goal_theta):
        # move the robot in simulation
        while True:
            # Predict the robot's pose after dt time and obtain the control input
            robot.move_to(goal_x, goal_y, goal_theta)
            x, y, theta = robot.get_pose()
            rospy.loginfo('x: %.2f, y: %.2f, theta: %.2f' % (x, y, theta))
            vx, vy, omega = robot.get_velocity()
            rospy.loginfo('vx: %.2f, vy: %.2f, omega: %.2f' % (vx, vy, omega))

            # plot the pose of the robot
            pose_arrow = ax.arrow(x, y, 0.1 * math.cos(theta), 0.1 * math.sin(theta), \
                                head_width=0.2, head_length=0.2, fc='r', ec='r')
            # plot the trajectory
            ax.plot(x, y, 'r.')

            plt.pause(0.001)
            pose_arrow.remove()

            # Arriving at the target position
            if abs(x - goal_x) < self.target_range_x and \
                abs(y - goal_y) < self.target_range_y and \
                    abs(theta - goal_theta) < self.target_range_theta:
                time.sleep(1)
                break
            time.sleep(self.dt)

    def visualize_simulation(self, points_list=[[-1, 0, 0],
                                                [-1, 1, 1.57],
                                                [-2, 1, 0],
                                                [-2, 2, -1.57],
                                                [-1, 1, -0.78],
                                                [0, 0, 0]]):
        fig, ax = plt.subplots(figsize=(5,5))
        ax.set_xlim(-3, 3)
        ax.set_ylim(-3, 3)
        # Visualize the starting point and the target point
        ax.scatter(0, 0, marker='*', s=200, c='b')
        for point in points_list:
            goal_x, goal_y, goal_theta = point
            ax.scatter(goal_x, goal_y, marker='*', s=200, c='b')

        for point in points_list:
            goal_x, goal_y, goal_theta = point
            self._visualize_simulation(ax, robot, goal_x, goal_y, goal_theta)


    def robot_follow_point(self, mpi_ctrl, goal_x, goal_y, goal_theta):
        # move the robot
        while True:
            # Predict the robot's pose after dt time and obtain the control input
            self.move_to(goal_x, goal_y, goal_theta)
            x, y, theta = self.get_pose()
            rospy.loginfo('x: %.2f, y: %.2f, theta: %.2f' % (x, y, theta))
            vx, vy, omega = self.get_velocity()
            rospy.loginfo('vx: %.2f, vy: %.2f, omega: %.2f' % (vx, vy, omega))
            Omega = np.dot(self.A, np.array([vx, vy, omega])) * self.calibration_parameters
            rospy.loginfo('Omega (for wheels): %.2f, %.2f, %.2f, %.2f' % \
                            (-Omega[2], Omega[1], -Omega[0], Omega[3]))
```

```python

            mpi_ctrl.setFourMotors(-Omega[2], Omega[1], -Omega[0], Omega[3])
            # -bl, fr, -fl, br

            # Determine whether the target pose has been reached
            if abs(x - goal_x) < self.target_range_x and \
                abs(y - goal_y) < self.target_range_y \
                    and abs(theta - goal_theta) < self.target_range_theta:
                # stop robot
                # time.sleep(1)
                break
            time.sleep(self.dt)

def robot_main():
    rospy.init_node('mpi_openloop_controller')
    # Instantiate the robot
    robot = RobotSimulator()
    mpi_ctrl = MegaPiController(port='/dev/ttyUSB0', verbose=True)

    k = 0.8 # scale parameter
    points_list = np.array([[-0.5, 0, 0],
                            [-0.5, 0.5, 1.57],
                            [-1, 0.5, 0],
                            [-1, 1, -1.57],
                            [-0.5, 0.5, -0.78],
                            [0, 0, 0]])
    points_list[:,0:2] = points_list[:,0:2] * k

    # robot.visualize_simulation(self, points_list=points_list)
    while not rospy.is_shutdown():
        for point in points_list:
            goal_x, goal_y, goal_theta = point
            robot.robot_follow_point(mpi_ctrl, goal_x, goal_y, goal_theta)
            mpi_ctrl.carStop()
            time.sleep(1)


# main
if __name__ == '__main__':
    robot_main()
```