Mission Coordination Lab

Name: Mostafa ELTABLAWY Student Number: 20246438
Name: Ahmed ELDESOKY Student Number: 20246092

Contents

1 Objective

The primary objective of this lab is to develop and test strategies for autonomous robot navigation in a simulated environment using ROS (Robot Operating System). The lab consists of two parts:

1. In the first part, the focus is on guiding a single robot to its designated flag in an environment without obstacles. The goal is to ensure the robot reaches the target efficiently by leveraging basic navigation techniques.

2. The second part builds upon the first by introducing obstacle avoidance. Robots must detect and respond to obstacles using ultrasonic sensors while maintaining their ability to navigate toward the designated flag safely and efficiently.

These objectives aim to improve understanding and application of robotic control, sensor integration, and strategy implementation for robust autonomous navigation.

2 Non obstacles strategy

2.1 Proposed Solution

The goal of the first strategy is to guide a single robot from its initial position to its designated flag in a simulation environment without any obstacles. The robot moves in a straight line toward the flag while maintaining its highest velocity until it reaches a predefined proximity to the target. The solution leverages the following components:

- The robot's current position and orientation are continuously monitored using the odometry data.
- The distance to the flag is retrieved via a ROS service.
- The robot's ultrasonic sensor confirms the absence of obstacles.
- Linear velocity is set to a maximum value when the distance to the flag is large and reduced to zero when the flag is reached.

2.2 Limits of the Least Robust Solution

While the proposed solution is effective in environments without obstacles, it lacks the following:

- Obstacle Avoidance: The robot does not react to obstacles that might appear unexpectedly in its path.
- Orientation Control: The robot assumes it is correctly oriented toward the flag and does not correct its heading if misaligned.
- Scalability: This solution does not account for multiple robots operating simultaneously.

2.3 Improvements

To improve the robustness of this strategy, the following enhancements can be implemented:

- Integrate obstacle detection and avoidance using the ultrasonic sensor data.
- Implement a proportional control mechanism to ensure accurate orientation toward the flag.
- Introduce coordination mechanisms to enable multiple robots to navigate simultaneously without collisions.

2.4 Explanation of the Solution and Implementation

The implementation starts by initializing the ROS environment and creating a Robot class. The class handles:

- Retrieving the robot's pose using odometry data.
- Publishing velocity commands to the robot's `/cmd_vel` topic.
- Calculating the distance to the target flag using the `distanceToFlag` service.

2.4.1 Main Steps of the Implementation

1. **Initialization**: The robot is initialized with its name, and ROS publishers and subscribers are configured.
2. **Move Toward Flag**: The robot continuously queries its distance to the flag. If the distance is greater than 2.0 meters, the robot moves at a linear velocity of 2.0 m/s.
3. **Stop at the Flag**: When the robot reaches the predefined proximity (distance less than 2.0 meters), it stops by setting both linear and angular velocities to zero.

2.4.2 Python Implementation

Below is the Python code implementing the described strategy:

Listing 1: Non-Obstacle Strategy

```
#!/usr/bin/env python3
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Range
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from evry_project_plugins.srv import DistanceToFlag

class Robot:
    def __init__(self, robot_name):
        self.robot_name = robot_name
        self.sonar = 0.0
```

```
        self.x, self.y, self.yaw = 0.0, 0.0, 0.0
        rospy.Subscriber(f"{robot_name}/sensor/sonar_front",
           Range, self.callbackSonar)
        rospy.Subscriber(f"{robot_name}/odom", Odometry, self.
           callbackPose)
        self.cmd_vel_pub = rospy.Publisher(f"{robot_name}/cmd_vel
           ", Twist, queue_size=1)

    def callbackSonar(self, msg):
        self.sonar = msg.range

    def callbackPose(self, msg):
        self.x = msg.pose.pose.position.x
        self.y = msg.pose.pose.position.y
        quaternion = msg.pose.pose.orientation
        _, _, self.yaw = euler_from_quaternion([quaternion.x,
           quaternion.y, quaternion.z, quaternion.w])

    def getDistanceToFlag(self):
        rospy.wait_for_service('/distanceToFlag')
        try:
            service = rospy.ServiceProxy('/distanceToFlag',
               DistanceToFlag)
            return service(self.x, self.y, int(self.robot_name
               [-1])).distance
        except rospy.ServiceException as e:
            rospy.logerr(f"Service␣call␣failed:␣{e}")

    def set_speed_angle(self, linear, angular):
        cmd_vel = Twist()
        cmd_vel.linear.x = min(max(linear, -2.0), 2.0)
        cmd_vel.angular.z = min(max(angular, -1.0), 1.0)
        self.cmd_vel_pub.publish(cmd_vel)

if __name__ == "__main__":
    rospy.init_node("Controller")
    robot = Robot(rospy.get_param("~robot_name"))
    rospy.sleep(3 * int(robot.robot_name[-1]))

    while not rospy.is_shutdown():
        distance = robot.getDistanceToFlag()
        if distance < 2.0:
            robot.set_speed_angle(0.0, 0.0)
            break
        robot.set_speed_angle(2.0, 0.0)
        rospy.sleep(0.5)
```

3 obstacles strategy

3.1 Proposed Solution

The second strategy extends the first by introducing obstacle avoidance capabilities. The goal remains to guide each robot to its designated flag, but now the robot must detect and respond to obstacles in its path using ultrasonic sensor data. The solution integrates the following:

- Continuous monitoring of the robot's sonar sensor for nearby obstacles.
- Dynamic adjustment of linear and angular velocities to avoid collisions.
- Proportional control (PID) to regulate speed as the robot approaches the flag.

3.2 Limits of the Least Robust Solution

The improved strategy addresses obstacle avoidance but has limitations:

- Suboptimal Navigation: The robot may take inefficient paths due to simple obstacle avoidance logic.
- Dependency on Sensor Data: Performance degrades with noisy or inaccurate sonar readings.
- Lack of Coordination: The strategy is designed for a single robot and does not account for interactions between multiple robots.

3.3 Improvements

To further enhance this solution, the following improvements are recommended:

- Implement advanced path-planning algorithms to optimize navigation.
- Integrate multiple sensors (e.g., LiDAR) for robust obstacle detection.
- Develop a multi-robot coordination mechanism to prevent collisions and improve efficiency.

3.4 Explanation of the Solution and Implementation

The implementation builds on the first strategy by incorporating obstacle detection and avoidance. The robot adjusts its trajectory when an obstacle is detected, ensuring safe navigation.

3.4.1 Main Steps of the Implementation

1. **Initialization**: Similar to the first strategy, the robot is initialized with its name, and publishers and subscribers are set up.
2. **Obstacle Detection**: The robot uses its sonar sensor to detect obstacles within a threshold distance (2.0 meters).
3. **Obstacle Avoidance**: If an obstacle is detected, the robot slows down and adjusts its angular velocity to navigate around it.
4. **Move Toward Flag**: Once the obstacle is cleared, the robot reorients and continues toward the flag.
5. **Stop at the Flag**: The robot stops when it reaches the predefined proximity to the flag.

3.4.2 Python Implementation

Below is the Python code implementing the described strategy:

Listing 2: Obstacle Avoidance Strategy

```
#!/usr/bin/env python3
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Range
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
import math
from evry_project_plugins.srv import DistanceToFlag

class Robot:
    def __init__(self, robot_name):
        self.robot_name = robot_name
        self.sonar = 0.0
        self.x, self.y, self.yaw = 0.0, 0.0, 0.0
        rospy.Subscriber(f"{robot_name}/sensor/sonar_front",
           Range, self.callbackSonar)
        rospy.Subscriber(f"{robot_name}/odom", Odometry, self.
           callbackPose)
        self.cmd_vel_pub = rospy.Publisher(f"{robot_name}/cmd_vel
           ", Twist, queue_size=1)

    def callbackSonar(self, msg):
        self.sonar = msg.range

    def callbackPose(self, msg):
        self.x = msg.pose.pose.position.x
        self.y = msg.pose.pose.position.y
        quaternion = msg.pose.pose.orientation
        _, _, self.yaw = euler_from_quaternion([quaternion.x,
           quaternion.y, quaternion.z, quaternion.w])
```

```
    def getDistanceToFlag(self):
        rospy.wait_for_service('/distanceToFlag')
        try:
            service = rospy.ServiceProxy('/distanceToFlag',
               DistanceToFlag)
            return service(self.x, self.y, int(self.robot_name
               [-1])).distance
        except rospy.ServiceException as e:
            rospy.logerr(f"Service call failed: {e}")

    def set_speed_angle(self, linear, angular):
        cmd_vel = Twist()
        cmd_vel.linear.x = max(min(linear, 2.0), -2.0)
        cmd_vel.angular.z = max(min(angular, 1.0), -1.0)
        self.cmd_vel_pub.publish(cmd_vel)

def avoid_obstacles(robot):
    sonar_distance = robot.sonar
    if sonar_distance < 2.0:
        rospy.loginfo(f"{robot.robot_name}: Obstacle detected at
           {sonar_distance}m. Adjusting course.")
        return 0.5, 3.0 if robot.yaw < 0 else -3.0
    return None

def move_to_flag(robot, kp):
    distance = robot.getDistanceToFlag()
    rospy.loginfo(f"{robot.robot_name}: Distance to flag = {
       distance}")
    if distance < 2.0:
        rospy.loginfo(f"{robot.robot_name}: Flag reached!")
        return 0.0, 0.0
    velocity = kp * distance
    return velocity, 0.0

if __name__ == "__main__":
    rospy.init_node("Controller")
    robot_name = rospy.get_param("~robot_name")
    kp = rospy.get_param("~kp", 1.0)
    robot = Robot(robot_name)

    while not rospy.is_shutdown():
        obstacle_response = avoid_obstacles(robot)
        if obstacle_response:
            velocity, angle = obstacle_response
        else:
            velocity, angle = move_to_flag(robot, kp)
        robot.set_speed_angle(velocity, angle)
        rospy.sleep(0.5)
```