University d'Evry Paris-Saclay

Multiagent control for UGV using ROS

Mission Coordination

Workout report

Bhargav Malasani Nagaraj, 20246517
bhargav.malasaninagaraj@etud.univ-evry.fr

Abel Mebratu Yehuala, 20245853
abel.yehuala@etud.univ-evry.fr

Instructor:
Boris KIEMA
Teewendeboris.kiema@univ-evry.fr

January 18, 2025

Contents

Project Objective

The goal of this project is to navigate three simulated robots, using ROS 1 and Gazebo, from their initial positions to their respective flags (e.g., robot1 to flag1, robot2 to flag2, and robot3 to flag3) while ensuring they avoid collisions with one another. The primary objective is to reach the designated flags as quickly as possible.

The simulated robots are characterized as follows:

- An ultrasonic sensor is embedded, with a limited range (5 meters).
- Two motorized wheels allow the robot to operate in the environment.
- Its current pose in the environment is known (position and orientation).

Introduction

This Mission Coordination Lab focuses on the implementation of strategies for coordinating multiple autonomous robots to navigate safely from a starting position to their adequate flag (e.g., robot1 going to flag1, robot2 going to flag2, and robot3 going to flag3) while avoiding any collision with another robot.

This project uses the Robot Operating System (ROS) as the middleware framework and Gazebo as the simulation platform. Each robot is equipped with an ultrasonic sensor for obstacle detection and navigation, motorized wheels for mobility, and precise position and orientation tracking within the simulation environment.

Github Repository: Misson_coordination-TP

1 Lab 1

```
roslaunch evry_project_strategy agent.launch nbr_robot :=1
```

1.1 Strategy Node

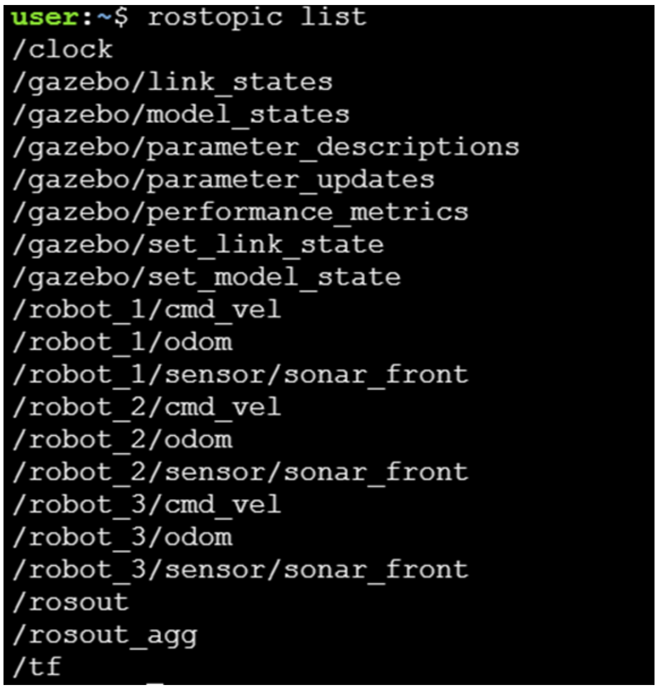
Q1: What does the terminal output represent? The terminal displays real-time logs of the robot's status. The sonar sensor consistently detects obstacles, while the flag distance values decrease progressively, indicating the robot's movement toward its target.

1.2 Visualisation and Topic Analysis

1.2.1 List Topics

```
rostopic list
```

Q2: What is the list command used for, and what is the result? The `rostopic list` command is used to inspect the available topics within a ROS system, facilitating understanding of data flow between nodes. It identifies topics related to the robot's pose (/robot_1/odom), velocity commands (/robot_1/cmd_vel), and sensor data (e.g., ultrasonic sensor topics).



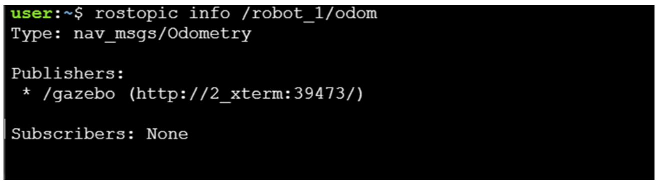
```
user:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/robot_1/cmd_vel
/robot_1/odom
/robot_1/sensor/sonar_front
/robot_2/cmd_vel
/robot_2/odom
/robot_2/sensor/sonar_front
/robot_3/cmd_vel
/robot_3/odom
/robot_3/sensor/sonar_front
/rosout
/rosout_agg
/tf
```

Figure 1: rostopic list

1.2.2 Publisher and Subscriber

Q3: Who are the publisher and the subscriber?

- **Publisher:** /gazebo. The Gazebo simulation publishes odometry data for robot_1.
- **Subscribers:** None. Currently, no ROS nodes subscribe to this topic.

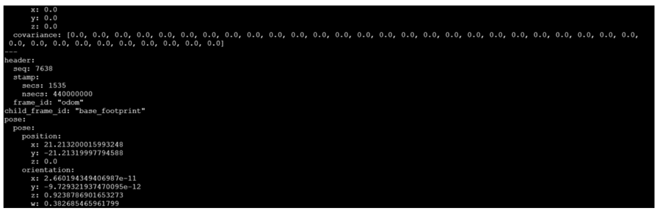


```
user:~$ rostopic info /robot_1/odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://2_xterm:39473/)

Subscribers: None
```

Figure 2: Publisher and the Subscriber

1.2.3 Topic Message Types

Q4: What type of messages are published on this topic? The /robot_1/odom topic publishes `nav_msgs/Odometry`, which includes the robot's position, orientation, and velocity in space.



```
      x: 0.0
      y: 0.0
      z: 0.0
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---
header:
  seq: 7638
  stamp:
    secs: 1535
    nsecs: 440000000
  frame_id: "odom"
child_frame_id: "base_footprint"
pose:
  pose:
    position:
      x: 21.213200015993248
      y: -21.21319997794588
      z: 0.0
    orientation:
      x: 2.660194349406987e-11
      y: -9.729321937470095e-12
      z: 0.9238786901653273
      w: 0.382685465961799
```

Figure 3: Messages published on /robot_1/odom topic

1.2.4 Echo Command

```
rostopic echo /robot_1/odom
```

Q5: What is the echo command used for, and what is the result? The `rostopic echo` command displays real-time messages from a specific ROS topic. It outputs a live stream of messages, reflecting the data being published, such as odometry details for the robot.

1.3 Moving one Robot to the corresponding path

Q6: Modify your program to move one robot safely to its corresponding flag and stop it at this position

```
        blueif distance < distance_threshold:
            velocity = 0
```

If the the distance(computed from "robot.getDistanceToFlag()") is less than the threshold the the velocity of the robot is set to zero. This makes the robot to stop at the Flag.

Q7: Implement a PID controller. It means that, when the robot is far from its goal, it moves with the highest velocity values and and as it gets closer, it slows down to a stop.

```
    blueclass PIDController:
    bluedef __init__(self, kp=0.5, ki=0.1, kd=0.1):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.sum_error = 0.0
        self.prev_error = 0.0

    bluedef compute(self, error):
        self.sum_error += error
        diff_error = error - self.prev_error
        output = self.kp * error + self.ki * self.sum_error + self.kd * d
        self.prev_error = error
        bluereturn output
```

The PID controller is implemented in the ‘PIDController‘ class with three parameters: proportional (`kp`), integral (`ki`), and derivative (`kd`) coefficients. The controller calculates the control output.

The PID controller is used to compute the robot's linear velocity based on the error, which is the difference between the robot's current distance from the target flag and a predefined threshold distance. The ‘compute‘ method updates the control output dynamically, ensuring smooth adjustments to the robot's velocity.

The ‘run_demo‘ function integrates this controller with real-time data from the robot's ultrasonic sensor and odometry. The computed velocity is passed to the robot's motion control system via the ‘set_speed_angle‘ method, enabling precise and stable navigation toward the target flag while adapting to changing conditions.

1.4 Implementing Timing Strategy

Q8: implement one of the simplest strategies: timing strategy. The timing strategy for multiple robots is implemented by introducing a delay in the start of each robot based on its identifier. In the 'run_demo' function, a delay is applied using the following line:

```
rospy.sleep(3 * blueint(robot_name[-1]))
```

This ensures that the robots do not start simultaneously, reducing the likelihood of collisions at the intersections and allowing smoother coordination. The delay is calculated as three seconds multiplied by the robot's numerical identifier, ensuring staggered starts for all robots. **Q9: Launch file for the strategy**

```
    <?xml version="1.0" encoding="UTF-8"?>

<launch>
  <arg name="nbr_robot" default="3"/>

  <!--MAIN CODE-->
  <node pkg="evry_project_strategy" type="agent_strategy1.py" name="agent_1"
    <param name="robot_name" value="robot_1"/>
  </node>

  <node pkg="evry_project_strategy" type="agent_strategy1.py" name="agent_2"
    <param name="robot_name" value="robot_2"/>
  </node>

  <node pkg="evry_project_strategy" type="agent_strategy1.py'" name="agent_3"
    <param name="robot_name" value="robot_3"/>
  </node>

</launch>
```

Here the file 'agent_strategy1.py' contains the node for the timing strategy.

2 Lab 2 : Robust strategies

2.1 Adaptive PID Navigation Strategy

2.1.1 Implementation

The robust strategy is implemented in the 'run$_demo'function, which integrates PID controllers for both line$

PID Control for Speed and Angle: The linear PID controller adjusts the robot's speed based on the distance to the target flag, while the angular PID controller manages orientation corrections using the robot's yaw angle.

Obstacle Avoidance: The strategy incorporates enhanced obstacle avoidance. If the robot detects an obstacle within 0.5 meters using its ultrasonic sensor, the linear velocity is set to zero, and angular adjustments are made to avoid collisions.

Dynamic Adjustments: Depending on the distance to the flag, specific conditions dynamically modify the angle and speed to ensure smooth navigation and target alignment.

2.1.2 Limits

- **Sensor Dependency:** The strategy heavily relies on the ultrasonic sensor for obstacle detection, making it less robust in environments with sensor noise or failure.
- **Predefined Parameters:** The PID parameters (`kp`, `ki`, `kd`) and distance thresholds are predefined, limiting adaptability to varying environments without manual tuning.
- **Complex Obstacle Scenarios:** The current approach may struggle in environments with dynamic or closely spaced obstacles, as it relies on simple distance thresholds.

2.1.3 Improvements

- **Improved Obstacle Avoidance Logic:** Replace the simple distance threshold-based obstacle avoidance with a predictive approach that calculates the robot's potential collision trajectory and adjusts both speed and angle dynamically to avoid obstacles without halting completely.
- **Enhanced Target Alignment:** Modify the angular PID control logic to incorporate a proportional adjustment to the target distance. This ensures smoother turns and avoids overshooting or oscillations near the target.
- **Path Re-Evaluation:** Add logic to periodically re-evaluate the robot's path to the target flag. This can account for changes in the environment, such as new obstacles, and provide a more robust approach to reaching the goal.
- **Error Prioritization:** Improve the PID control by introducing a weighted error calculation, prioritizing angular corrections when the robot is misaligned and prioritizing speed when alignment is accurate.