Lab Report

AMARA Jugurta, NACERBEY Oussama

January 18, 2025

Objective

The objective of this project is to navigate three simulated robots in a ROS 1 and Gazebo environment from their starting positions to their respective flags (robot1 to flag1, robot2 to flag2, and robot3 to flag3) while avoiding collisions. The goal is to reach the flags as quickly as possible using robots equipped with ultrasonic sensors (range: 5 meters), motorized wheels, and real-time pose awareness (position and orientation).

VISUALISATION OF SOME CONCEPTS

Step 1: Listing ROS Topics

To list all the active ROS topics, the following command can be used:

```
$ rostopic list
```

The output displayed the list of current topics, which included:

- /clock
- /gazebo/link_states
- /robot_1/odom
- /robot_1/sensor/sonar_front
- /robot_2/odom
- /robot_3/odom

This command is used to list all the currently active ROS topics related to the robot's odometry and sensors.

Step 2: Information about a Specific Topic

The following command is used to gather information about the '/robot_1/odom' topic:

```
$ rostopic info /robot_1/odom
```

We get the following details:

- **Type:** nav_msgs/Odometry

- **Publisher:** gazebo (http://1_xterm:34375/)
- **Subscriber:** None

Q-3

- **Publisher:** The topic is published by 'gazebo', which simulates the robot and its environment.
- **Subscriber:** There are no subscribers for this topic.

Q4-The type of messages published on this topic is 'nav msgs/Odometry', which includes data about the robot's position, orientation, velocity, and covariance.

Step 3: Listening to a Topic

To view the real-time data being published on the '/robot_1/odom' topic, the following command can be used:

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

The resulting output displayed the odometry data, which includes:

- Header information (sequence number, timestamp, and frame ID).
- Pose (position and orientation) in the odometry frame.
- Twist (linear and angular velocity).
- Covariance matrices for pose and twist.

The 'rostopic echo' command allows us to view real-time messages published on a specific topic. In this case, it displayed detailed odometry information, such as:

- Robot's position.
- Robot's orientation in quaternion format.
- Linear and angular velocities.

This data is crucial for analyzing the robot's movement and understanding its behavior in the simulation.

Below is the output:

```
header:
  seq: 2120
  stamp:
    secs: 426
    nsecs: 701000000
  frame_id: "odom"
child_frame_id: "base_footprint"
pose:
  pose:
    position:
      x: -10.260697387624251
      y: 10.260689906815912
      z: 0.0
    orientation:
      x: 0.00020833214808612356
      y: -8.634811127364211e-05
      z: 0.9237899531347284
      w: 0.38289955814685434
  covariance: [...values...]
twist:
  twist:
    linear:
      x: 1.5438893755021604e-07
      y: 5.703677215837003e-07
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: -1.1861620850824004e-06
  covariance: [...values...]
---
```

MOVE ONE ROBOT: If the desired velocity is modified in the program but set to a value greater than 2, the robot will only move at its maximum velocity, which is capped at 2. This is because the robot's maximum velocity is predefined and enforced

STEP 4: Moving One Robot to Its Corresponding Flag

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

To accomplish this, we first calculate the distance from the robot to the flag. Once the robot's velocity reaches a specified threshold (0.5 m/s), we set the velocity to zero, bringing the robot to a stop.

Here is the code:

```
velocity = 2
if distance < 0.5:
    velocity = 0
```

Q7: Adaptation to real life using a PID controller

A PID (Proportional-Integral-Derivative) controller ensures smooth and precise control of the robot's movement. The PID consists of 3 elements:

- **Proportional Control (P):** this adjusts the velocity proportionally to the distance (error) to the flag, allowing faster movement when far and slower as the robot nears the flag.

- **Integral Control (I):** which accounts for accumulated error over time to ensure the robot reaches the exact position of the flag.

- **Derivative Control (D):** this Predicts and reduces overshoot by considering the rate of change of error.

The robot starts at maximum velocity when far from the target and slows down progressively as it nears the flag, ensuring safe and precise stopping.

```
# PID initialization
    intg = 0.0
    prev_e = 0.0
    dt = 0.5  # Time step

    while not rospy.is_shutdown():
        # Get robot's current distance to the flag and obstacle
           distance
        distance = float(robot.getDistanceToFlag())
        dist_obs = float(robot.get_sonar())
        print(f"{robot_name} distance to flag = {distance},
           distance to obstacle = {dist_obs}")

        # PID coefficients
        Kp = 1.0
        Ki = 0.01
        Kd = 0.1

        # PID components
        error = distance # the error
        Prop = error  # Proportional part
        intg += error*dt # Integral part
        deriv = (error-prev_e)/dt # Derivative part

        # Computing the velocity
        velocity = Kp*Prop + Ki*intg + Kd*deriv
        prev_e = error

        velocity = max(min(velocity, 4.0), -2.0)
        if distance < 0.5:
            velocity = 0
```

STEP 5: Timing Solution

Q8: Implementing the Timing Strategy

The timing strategy is a simple and effective method by leveraging timing offsets to prevent collisions while ensuring all robots reach their flags efficiently. This method lets the robots start the movement at delayed times.

Implementation Steps:

1. we assigned different start times for each robot:
   - Robot 1 starts at $t = 0$ seconds.
   - Robot 2 starts at $t = 5$ seconds.

- Robot 3 starts at $t = 10$ seconds.

This code implements the "timing strategy" by:

```
# delay for timing strategy
    robot_id = int(robot_name[-1])
    delay_t = robot_id * 2
    rospy.sleep(delay_t)
```

1. Extracting the ID for each robot from its name.
2. Calculating a delay proportional to the robot's ID
3. Pausing the robot's execution for the calculated delay rospy.sleep

Lab 2

Strategy:

1 Initialization

- we first initialise the 2 PID controllers:
  - `a_pid`: this controls the angular velocity.
  - `v_pid`: this controls the linear velocity.
- we then define the obstacle avoidance parameters for our robots:
  - `safe_d`: the minimum distance to detect and start to avoid the obstacles.
  - `avoid_V` and `avoid_A`: we also specified a Velocity and angle for avoiding obstacles.
- we get the Target flag positions for each robot that we defined at the start of the code.

```
# Initialisiing the PID controller
    a_pid = PIDController(kp=0.5, ki=0.0, kd=0.05,
       output_limit=(-1.0, 1.0))  # the control angle
    v_pid = PIDController(kp=0.5, ki=0.0, kd=0.05,
       output_limit=(0.0, 2.0))  #  the control velocity

    # we set some Obstacle avoidance parameters
    safe_d = 4     # Minimum safe distance to avoid obstacles
    avoid_V = 1.2     # we reduce the velocity when avoiding
       obstacles
    avoid_A = math.pi/2    # we set the angle of turning when
        avoiding obstacles
    flag_x, flag_y = robot.target_flag  # getting the flag
       position
```

the target Flag Positions are:

```
# Flag positions
flag_positions = {
    "robot_1": (-21.21320344, 21.21320344),# Flag 1
    "robot_2": (21.21320344, 21.21320344), # Flag 2
    "robot_3": (0, -30)                         #    Flag3
}
```

```
self.target_flag = flag_positions[robot_name]
```

we also modified the "init" function of the robot class so that for each robot will use the right flag coordinates to do its navigation by using the line shown above.

2 Main Loop

2.1 Obstacle Avoidance

The robot reads the sonar sensor to measure the distance to the nearest obstacle, then if the distance is less than `safe_d`, the robot enters obstacle avoidance mode were we use the previously predifined parameters for obstacle avoidance were the robot turns using avoiding angle(`avoid_A`) and reduce its speed (`avoid_V`). this loop also skips further execution to focus on avoiding the obstacle.

```
# we use the sonar to get the distance to the closest obstacle
        sonar_dist =robot.get_sonar()
        if sonar_dist < safe_d:  # if an obstacle is detected do:

            print(f"{robot_name} detected obstacle at distance =
               {sonar_dist}")

            # we set the avoiding angle and velocity
            velocity = avoid_V
            angle = avoid_A
            robot.set_speed_angle(velocity, angle)
            rospy.sleep(0.3)
             continue
```

2.2 Navigation to the Target Flag

We retrieve the robot's current position (`robot_x, robot_y`) and orientation (`current_angle`). The target flag coordinates (`flag_x, flag_y`) are also retrieved from the dictionary. then we calculate the relative position to the flag:

- `dx = flag_x - robot_x`
- `dy = flag_y - robot_y`

The distance to the flag (`distance_to_flag`) and the target angle (`target_angle`) are computed using trigonometric functions, and the angle error is calculated as the difference between the `target_angle` and `current_angle`.

```
# Strategy

        distance_to_flag = robot.getDistanceToFlag()
        # get robot's pose
        robot_x, robot_y, current_angle= robot.get_robot_pose()
        # the coordinates of the flag
        #flag_x = -21.21320344
        #flag_y = 21.21320344

        # Calculating dx and dy relative to the robot's position
        dx = flag_x-robot_x
        dy = flag_y-robot_y
        # calculate distance and target angle
        distance_to_flag = math.sqrt((dx)*2 + (dy)*2)
        target_angle = math.atan2((dy), (dx))
```

2.3 PID Controllers

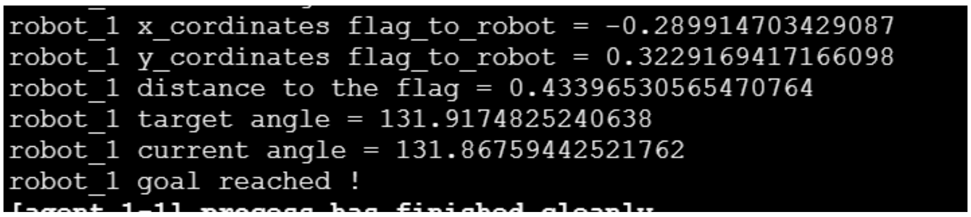
- The PID controllers compute the required adjustments for our velocity and angle:
  - `v_pid` computes the velocity based on the distance to the flag.
  - `a_pid` computes the angle based on the angle error.

```
# calculating the angle error
        angle_error = target_angle - current_angle

        # calculating the velocity and angle using the pid
        velocity = v_pid.compute(distance_to_flag)
        angle = a_pid.compute(angle_error)
```

2.4 Reaching the Flag

- If the robot is within a threshold distance (`< 0.5`) of the flag:
  - The robot stops by setting both velocity and angular speed to zero.
  - A message indicates that the goal has been reached.

```
print(f"{robot_name} x_cordinates flag_to_robot = {dx}")
        print(f"{robot_name} y_cordinates flag_to_robot = {dy}")
        print(f"{robot_name} distance to the flag = {
           distance_to_flag}")
        print(f"{robot_name} target angle = {math.degrees(
           target_angle)}")
        print(f"{robot_name} current angle = {math.degrees(
           current_angle)}")

        if abs(distance_to_flag) < 0.5:
            velocity = 0
            angle = 0
            robot.set_speed_angle(velocity, angle)
            print(f"{robot_name} goal reached !")
            break
```



```
robot_1 x_cordinates flag_to_robot = -0.289914703429087
robot_1 y_cordinates flag_to_robot = 0.3229169417166098
robot_1 distance to the flag = 0.43396530565470764
robot_1 target angle = 131.9174825240638
robot_1 current angle = 131.86759442521762
robot_1 goal reached !
```

Figure 1: The results of the robot after reaching the flag

3 Conclusion

This lab demonstrates the successful implementation of autonomous robot navigation in a simulated ROS and Gazebo environment. Using PID controllers for precise movement, obstacle avoidance for safety, and timing strategies to prevent collisions, each robot efficiently reached its designated flag.

The Github link is the following:

```
https://github.com/oussamanacerbey/
   mission_coordination_lab_report
```