Mission Coordination Laboratory Report Extended Multi-Robot Navigation in ROS Using PID Control and Robust Reactive Avoidance

Jérémi MAGRON, Nguyen Viet Khanh, Diyari M. Salih
M2 SAAS — Mission Coordination
University of Évry
Emails: jeremi91750@gmail.com, nguyenvietkhanh.hn@gmail.com, diyari.m.salih@gmail.com

Contents

***Abstract*—This report presents a detailed implementation and evaluation of mission coordination strategies for three mobile robots in ROS1 and Gazebo. Each robot must reach its assigned target flag while avoiding collisions with obstacles and other robots. The work follows the laboratory structure: ROS inspection and topic analysis, a basic stop controller, PID-based navigation, a timing coordination strategy, and a robust reactive strategy that combines sonar-based obstacle avoidance with robot-to-robot repulsion. We provide extended explanations in layman terms, include representative code snippets, and validate the behavior using Gazebo screenshots and terminal logs.**

***Index Terms*—ROS1, Gazebo, mission coordination, multi-robot systems, PID control, obstacle avoidance, reactive navigation**

I. Introduction

Mission coordination is the ability of multiple robots to achieve their objectives in the same environment without interfering with each other. Compared to single-robot navigation, multi-robot scenarios introduce additional challenges:

- robots can collide with each other (dynamic obstacles),
- the environment may contain static obstacles,
- timing and trajectories can conflict at crossings or shared zones.

In this laboratory, three differential-drive robots are simulated in Gazebo. Each robot must reach a unique target flag and stop safely. We implement progressively stronger strategies, because in robotics it is common to:

- start simple to validate sensing and actuation,
- then add control quality (smoothness, stability),
- then add coordination and robustness.

II. Simulation Setup and Evidence

We validate our strategies using:

- Gazebo screenshots showing start states and final states,
- runtime terminal logs confirming "reached" or "arrived" events.

To reduce whitespace and improve readability, related images are grouped into paired figures.

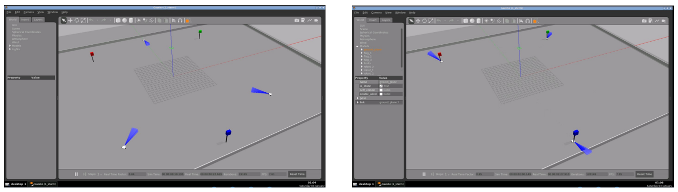


Fig. 1. Free-space experiment: (left) initial robot and flag configuration, (right) final positions after reaching targets.

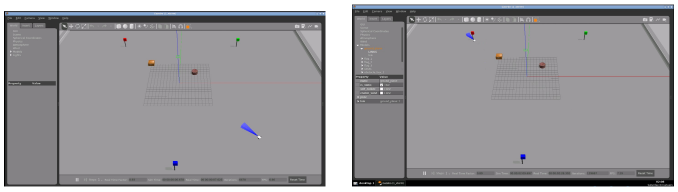


Fig. 2. Obstacle-space experiment: (left) initial configuration with obstacles, (right) final positions after navigation.

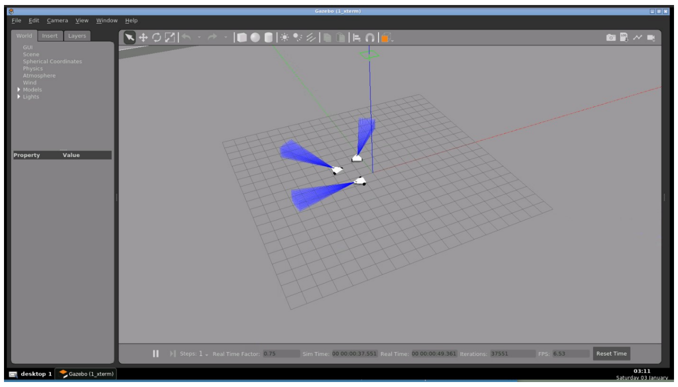
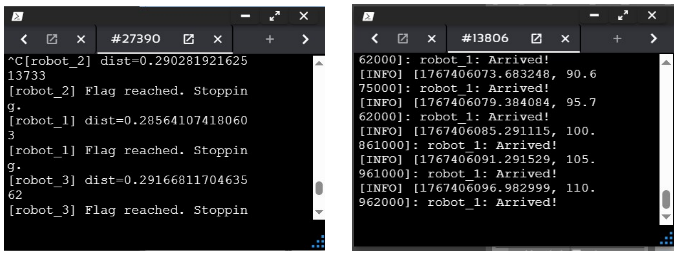


Fig. 3. Multi-robot navigation in progress: robots keep separation while moving toward their flags.



```
^C[robot_2] dist=0.290281921625
13733
[robot_2] Flag reached. Stoppin
g.
[robot_1] dist=0.28564107418060
3
[robot_1] Flag reached. Stoppin
g.
[robot_3] dist=0.29166811704635
62
[robot_3] Flag reached. Stoppin
```

```
62000]: robot_1: Arrived!
[INFO] [1767406073.683248, 90.6
75000]: robot_1: Arrived!
[INFO] [1767406079.384084, 95.7
62000]: robot_1: Arrived!
[INFO] [1767406085.291115, 100.
861000]: robot_1: Arrived!
[INFO] [1767406091.291529, 105.
961000]: robot_1: Arrived!
[INFO] [1767406096.982999, 110.
962000]: robot_1: Arrived!
```

Fig. 4. Terminal evidence: (left) free-space stop confirmations, (right) obstacle-space arrival logs.

III. Workout: ROS Theory Refresher (Layman-Friendly)

A. What is ROS?

ROS (Robot Operating System) is a robotics middleware that helps software components communicate. Instead of writing one big program, we write smaller programs called *nodes*. Each node does one job (sense, decide, or act), and nodes exchange data through standardized channels.

B. Topics vs Services

ROS provides two main communication styles:

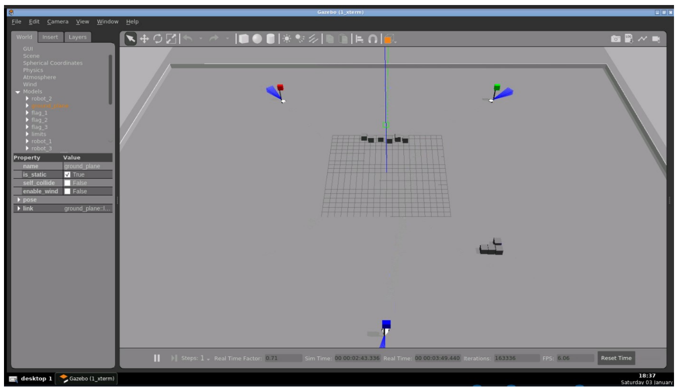


Fig. 5. Improved robust behavior: dynamic obstacle handling while navigating toward targets (reactive, still heuristic).

- **Topics (publish/subscribe)** for continuous data, like odometry or sonar.
- **Services (request/response)** for "ask a question and get an answer", like distance-to-flag.

In our lab:

- `/robot_i/odom` is a topic (continuous pose updates),
- `/distanceToFlag` is a service (distance query).

C. Why ROS is useful here

ROS is especially helpful in multi-robot simulations because:

- each robot can run the same node with a different parameter,
- each robot can have its own namespace, keeping topics separated,
- launch files can start everything consistently with one command.

IV. Theoretical Background and Equations

A. Differential-drive kinematics

Robots are controlled by:

- linear speed v (forward/backward),
- angular speed ω (rotation).

The robot pose (x, y, θ) evolves as:

$$\dot{x} = v\cos(\theta), \tag{1}$$

$$\dot{y} = v\sin(\theta), \tag{2}$$

$$\dot{\theta} = \omega. \tag{3}$$

B. Distance and heading to target

Distance to the flag:

$$d = \sqrt{(x_f - x)^2 + (y_f - y)^2}. \tag{4}$$

Desired heading angle:

$$\theta_d = \operatorname{atan2}(y_f - y, x_f - x). \tag{5}$$

C. PID control principle

PID computes a control value based on error:

$$u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \frac{de(t)}{dt}. \tag{6}$$

Why it helps (simple explanation):

- Far from target ⇒ bigger error ⇒ bigger command.
- Near target ⇒ smaller error ⇒ smaller command.
- Derivative term reduces overshoot by reacting to fast changes.

Angle normalization avoids discontinuity at $\pm\pi$:

$$e_\theta = \text{atan2}(\sin(\Delta\theta), \cos(\Delta\theta)). \tag{7}$$

V. Lab 1: Question Solutions (Expanded)

A. Q1: What do terminal numbers mean?

The terminal prints diagnostic values used by the controller:

- the robot name (e.g., `robot_1`),
- the current distance to its assigned flag (from the `/distanceToFlag` service),
- messages indicating completion (e.g., "Flag reached. Stopping.").

In other words, the terminal is showing the control loop monitoring progress.

B. Q2: What does `rostopic list` do?

`rostopic list` lists all active ROS topics. In this lab, the most relevant topics include:

- `/robot_i/odom` (odometry feedback),
- `/robot_i/cmd_vel` (velocity commands),
- `/robot_i/sonar` or similar sonar topic (distance measurement).

C. Q3–Q4: `Listening to a topic`

- **Publisher**: Gazebo (or the simulator bridge) publishes odometry.
- **Subscriber**: our control node subscribes to get robot pose.
- **Message type**: `nav_msgs/Odometry`.

D. Q5: `rostopic echo /robot_1/odom`

`rostopic echo` prints the live stream of odometry messages. This is useful because it confirms:

- the topic is active,
- the robot pose is changing when the robot moves,
- the message fields match what our code expects.

E. Q6: Move one robot to its flag and stop

The simplest working navigation strategy is:

- move forward at a constant speed,
- continuously check distance-to-flag,
- stop when close enough.

The key concept is a **threshold**: if $d < d_{\text{stop}}$, set both velocities to zero.

```
# Read distance to assigned flag from service
distance = float(robot.getDistanceToFlag())

# Default command: move forward
velocity = 2.0
angle = 0.0

# Stop zone (safe termination)
STOP_THRESHOLD = 2.0
if distance < STOP_THRESHOLD:
    print("Flag reached. Stopping.")
    velocity = 0.0
    angle = 0.0

# Publish command to /cmd_vel
robot.set_speed_angle(velocity, angle)
```

Listing 1. Q6: Distance-threshold stop controller (expanded snippet)

Why this works:

- it guarantees a stop near the target,
- it is easy to understand and debug.

Limitations:

- no steering correction,
- abrupt stop (not smooth),
- can overshoot if loop rate is slow.

F. Q7: PID controller for smooth approach

A PID controller improves realism because the robot slows down gradually as it approaches the flag. Instead of switching instantly from "fast" to "stop", the speed becomes proportional to distance error.

We use:

- **linear PID** for forward speed,
- **angular PID** for heading alignment.

```
def compute(self, error):
    now = rospy.Time.now()
    dt = (now - self.last_time).to_sec()
    if dt <= 0.0:
        dt = 1e-6

    # Integral accumulates error over time
    self.integral += error * dt

    # Derivative is how fast the error changes
    derivative = (error - self.prev_error) / dt

    # PID output
    output = (self.kp * error) + (self.ki * self.
        integral) + (self.kd * derivative)

    # Save for next iteration
    self.prev_error = error
    self.last_time = now
    return output
```

Listing 2. Q7: PID compute step (expanded, clear logic)

Heading error normalization avoids wrong turning direction:

```
angle_error = target_angle - yaw
angle_error = math.atan2(math.sin(angle_error),
    math.cos(angle_error))
```

Listing 3.]Q7: Heading error wrap to [-pi, pi]

Main benefits:

- smoother motion,
- less oscillation near the goal,
- better stopping accuracy.

G. *Q8: Timing strategy for three robots*

With three robots, collisions can happen if they move at the same time through shared areas. A simple coordination approach is to stagger their start times.

Delay policy:

$$\text{delay}(robot_i) = (i - 1) \times 5 \text{ s}. \quad (8)$$

```
robot_id = int(robot_name.split("_")[-1])  # "
    robot_2" -> 2
delay = (robot_id - 1) * 5.0

start_time = rospy.Time.now().to_sec()
while not rospy.is_shutdown():
    now = rospy.Time.now().to_sec()

    # Wait phase
    if (now - start_time) < delay:
        robot.set_speed_angle(0.0, 0.0)
        rospy.sleep(0.1)
        continue

    # After waiting, run normal PID navigation
    # (distance PID for v, angle PID for w)
    break
```

Listing 4. Q8: Timing coordination (expanded snippet)

Why it helps:

- reduces the probability that robots arrive in the same conflict region simultaneously,
- very simple to implement.

Why it is not robust:

- if one robot slows down due to an obstacle, timing assumptions break,
- it avoids collisions by schedule, not by perception.

H. *Q9: Launch file for timing strategy*

A ROS launch file is used to:

- start multiple ROS nodes simultaneously using a single command, instead of launching each node manually,
- pass configuration parameters (such as `robot_name`, control gains, or delay values) to each node at startup,
- ensure that each robot runs the same controller code while behaving differently based on its assigned parameters,
- enforce a consistent and repeatable experiment setup, where every simulation run starts in exactly the same way,
- reduce human error by avoiding forgotten nodes or incorrect startup order,
- simplify multi-robot scalability, as adding a new robot only requires duplicating a node entry in the launch file,
- allow namespace separation, ensuring that each robot publishes and subscribes to its own topics without interference,
- improve debugging and grading, since instructors can reproduce the experiment by running the same launch file,
- centralize experiment configuration, making it easier to modify parameters without changing source code.

```
<launch>
  <node pkg="evry_project_strategy" type="
      agent_PID_delay_multi.py"
        name="ctrl_robot_1" output="screen">
    <param name="robot_name" value="robot_1"/>
  </node>

  <node pkg="evry_project_strategy" type="
      agent_PID_delay_multi.py"
        name="ctrl_robot_2" output="screen">
    <param name="robot_name" value="robot_2"/>
  </node>

  <node pkg="evry_project_strategy" type="
      agent_PID_delay_multi.py"
        name="ctrl_robot_3" output="screen">
    <param name="robot_name" value="robot_3"/>
  </node>
</launch>
```

Listing 5. Q9: Launch structure for three robots (representative snippet)

VI. Lab 2: Robust Obstacle and Robot Avoidance (Expanded)

A. *Why we need a robust strategy*

Timing works only if the environment is predictable. In real robotics, unexpected events occur:

- obstacles appear,
- robots get delayed,
- paths change.

Therefore, we implement a reactive strategy that uses sensor measurements to adapt online.

B. *Strategy description*

Our robust strategy combines:

- **Goal attraction**: always pull the robot toward its flag,
- **Robot repulsion**: push away from nearby robots to avoid collision,
- **Sonar avoidance**: if a static obstacle is detected in front, enter avoidance behavior.

C. Core obstacle-avoidance logic

```
OBSTACLE_DIST = 1.5  # threshold (example)
state = "MOVING"

if sonar < OBSTACLE_DIST:
    # Obstacle is close: stop forward motion and
        rotate
    state = "AVOIDING"
    linear_vel = 0.0
    angular_vel = 1.0
else:
    # Normal navigation mode: continue toward
        goal
    state = "MOVING"
```

Listing 6. Lab 2: Sonar-triggered obstacle avoidance (expanded snippet)

D. Robot-to-robot repulsion

```
ROBOT_SAFE_DIST = 3.0
repulse_x, repulse_y = 0.0, 0.0

for other_name, other_pose in other_poses.items()
    :
    dx = my_x - other_pose.x
    dy = my_y - other_pose.y
    dist = math.sqrt(dx*dx + dy*dy)

    if dist < ROBOT_SAFE_DIST and dist > 1e-6:
        strength = (ROBOT_SAFE_DIST - dist) /
            dist
        repulse_x += dx * strength
        repulse_y += dy * strength
```

Listing 7. Lab 2: Repulsion term concept (expanded snippet)

E. Limitations (honest analysis)

Even with the improvement (Fig. 5), the method is still heuristic:

- the sonar is 1D and forward-facing, so side obstacles may not be detected early,
- potential-field style methods can create local minima (hesitation),
- turning for a fixed duration can be suboptimal for some obstacle shapes.

VII. Results and Interpretation

We interpret results using the provided evidence.

A. Free space

From Fig. 1:

- robots move to the correct flags,
- the environment does not force avoidance maneuvers,
- final stopping indicates correct threshold detection and/or arrival condition.

B. Obstacle space

From Fig. 2:

- obstacles cause deviations from straight-line paths,
- arrival is still achieved, demonstrating basic robustness,
- the stopping behavior remains stable near the goal.

C. Multi-robot interaction

From Fig. 3:

- robots maintain separation,
- coordination strategies reduce collision risk,
- avoidance behavior is visible in the sonar cones and relative trajectories.

D. Terminal logs

From Fig. 4:

- textual confirmation matches visual completion,
- distance values drop close to the stop threshold,
- logs provide reproducible proof of completion for grading.

VIII. Conclusion

This laboratory demonstrated a full progression from simple motion control to more realistic multi-robot coordination.

Key outcomes:

- Q6 implemented a safe stop controller based on distance-to-flag.
- Q7 introduced PID control for smoother, more realistic navigation.
- Q8 coordinated robots using a simple timing delay strategy.
- Q9 used launch files to deploy the multi-robot strategy reproducibly.
- Lab 2 implemented a robust reactive strategy using sonar obstacle detection and robot-to-robot repulsion.

The improved robust strategy increased dynamic obstacle handling (Fig. 5), although limitations remain due to minimal sensing and the heuristic nature of reactive avoidance.