1. Introduction

This project explores motion planning and control in a ROS environment using Gazebo simulation. The goal is to implement a ROS-based PID controller that drives a differential-drive robot to desired poses (position and orientation) within a Gazebo world. By using real-time feedback (odometry) and a PID control loop, the robot can correct its motion to reach target coordinates accurately. The lab focuses on two control modes – sequential vs. simultaneous PID control – to evaluate their effectiveness in guiding the robot to various goal poses.

2. Implementation Details

a. Workspace Setup

- **Package:** Created a Catkin workspace (via `catkin_init_workspace`) and a new ROS package named `pid_controller` for this lab. The package was set up with dependencies on standard ROS message packages (e.g., `rospy`, `geometry_msgs`, `nav_msgs`).
- **Scripts:** Two Python scripts were developed: `pid_controller.py` (PID control node) and `motion_planner.py` (planner node). Both scripts are placed in the `~/catkin_ws/src/pid_controller/scripts/` directory and made executable (`chmod +x`).
- **Gazebo Simulation:** The testing environment is a Gazebo simulator running a differential-drive robot model. The robot provides odometry on the `/odom` topic

(`nav_msgs/Odometry`) and accepts velocity commands on `/cmd_vel` (`geometry_msgs/Twist`). This setup mirrors a typical mobile robot, enabling realistic motion feedback for the controller.

- **Build and Run:** The workspace was compiled with `catkin_make`, and `source devel/setup.bash` was executed to overlay the new package. All nodes were developed and tested on ROS Noetic (Ubuntu 20.04) together with Gazebo (ROS Gazebo Classic). The nodes are launched using standard ROS commands (e.g., `rosrun pid_controller pid_controller.py` in one terminal, and `rosrun pid_controller motion_planner.py` in another) after the Gazebo simulation is started.

b. PID Controller Node (pid_controller.py)

This node is responsible for closed-loop control of the robot's motion. It subscribes to two topics: /reference_pose (the target pose published by the planner) and /odom (the robot's current pose from Gazebo). It publishes control commands to /cmd_vel to drive the robot, and signals completion by publishing a boolean flag on /goal_reached when the target is achieved. The node runs a control loop at a fixed rate (e.g., 10 Hz), continuously computing errors and updating the control outputs.

PID Control Implementation: Inside pid_controller.py, two independent PID controllers are implemented – one for linear velocity (to control forward/backward motion) and one for angular velocity (to control rotation). Each PID computes an output (command) based on the error, integral of error, and derivative of error for its respective quantity. The linear controller

aims to minimize the distance error (difference between current robot position and target position), while the angular controller minimizes the heading error (difference between current orientation and desired orientation or direction to goal). The chosen PID gains are:

- *Linear velocity PID:* Kp = 0.4, Ki = 0.0001, Kd = 0.0001
- *Angular velocity PID:* Kp = 1.0, Ki = 0.0001, Kd = 0.0001

These gain values were tuned experimentally to provide a smooth response with minimal overshoot. Very small integral and derivative gains were used (0.0001) to fine-tune steady-state error and dampening without introducing instability, while the proportional gains dominate the control action.

Control Modes: The controller supports two modes of operation, selected by an integer "mode" value in the reference pose message (0 or 1). The modes dictate how the linear and angular PID controllers are applied to reach the goal:

- **Mode 0 (Sequential Control):** The PID actions are executed in three sequential phases – *Align → Move → Align*. First, the robot rotates in place to face the target position (using the angular PID controller while linear velocity is kept zero). Once the robot's heading is pointing toward the goal, the controller switches to linear PID control to drive straight toward the target (angular velocity is zeroed during this phase). When the robot arrives near the goal position, the linear motion is stopped, and the angular PID is activated again to rotate the robot in place until its orientation matches the desired final angle (θ_r). This mode ensures each degree of freedom (heading then distance then final

angle) is handled one at a time, simplifying the control task.

- **Mode 1 (Simultaneous Control):** The linear and angular PID controllers are activated concurrently, meaning the robot will rotate and move at the same time. In this mode, the robot follows a curved trajectory towards the goal. The controller continuously adjusts both forward speed and steering angle in parallel, attempting to reduce both the distance to the goal and the orientation error together. Practically, the robot will not strictly face the goal before moving; instead, it makes progress toward the target position while also turning towards the final orientation. The final orientation (θ_r) is typically reached at the very end of the motion, once the robot is at the target location. This simultaneous approach is more efficient in theory (taking a more direct path), but requires well-tuned gains to avoid oscillation since both errors are being corrected at once.

During operation, the PID controller node monitors the error values. When the robot comes sufficiently close to the target (e.g., position error ≤ 0.1 m and orientation error ≤ 0.1 rad, satisfying the lab's accuracy requirements), the node considers the goal reached. It then publishes a message on /goal_reached (for example, a std_msgs/Bool True) and stops sending non-zero commands (publishes a zero velocity on /cmd_vel to halt the robot). This signals to the planner that the current goal is achieved. The node also prints status messages via rospy.loginfo for debugging and verification (e.g., current error values and when the goal is reached).

c. Motion Planner Node (motion_planner.py)

The motion planner node handles user interaction and high-level planning of successive targets. It runs in a loop, querying the user for a new goal and mode, then issuing that goal to the PID controller, and finally waiting for completion before repeating. This node ensures that only one goal is active at a time and that the next command is given only after the previous target has been reached.

Strategy:

1. **User Input:** Upon starting, the node prompts the user to enter a target pose and mode. The input consists of four values: the goal x-coordinate, goal y-coordinate, goal orientation θ (in radians), and the mode (0 for sequential, 1 for simultaneous). For example, the user might input: x = 2.5, y = -1.0, θ = 1.57, mode = 0.
2. **Publish Reference Pose:** The node constructs a reference pose message (e.g., a custom message or standard geometry_msgs/Pose combined with mode information) containing the user-specified (x, y, θ). It then publishes this message on the /reference_pose topic. The PID controller node, which is subscribed to this topic, receives the new target and begins executing the appropriate control actions (based on the mode).
3. **Wait for Goal Reached:** After publishing the goal, the motion planner node enters a waiting state. It either subscribes to the /goal_reached topic or monitors the robot's state via /odom to determine when the goal is achieved. In this implementation, the planner subscribes to /goal_reached (std_msgs/Bool) and simply waits until a True

signal is received from the PID controller node indicating success. This synchronization ensures the planner does not issue a new goal prematurely.

4. **Loop for Next Goal:** Once the /goal_reached flag is received, the planner logs or prints a confirmation that the robot has reached the target. It then loops back to prompt the user for the next (x, y, θ, mode) input. This process can repeat for any number of goals, allowing interactive testing of multiple target poses in one run. The planner cleanly shuts down if the user indicates no more goals or on ROS shutdown.

By separating the motion planning (user interaction & goal broadcasting) from the low-level PID control, the design follows a modular ROS architecture. The motion planner node remains simple (essentially a state machine waiting for completion), while the pid_controller node handles all continuous control. Both nodes use ROS logging to provide runtime feedback, and together they demonstrate a basic motion planning pipeline: from user command to robot actuation with feedback.

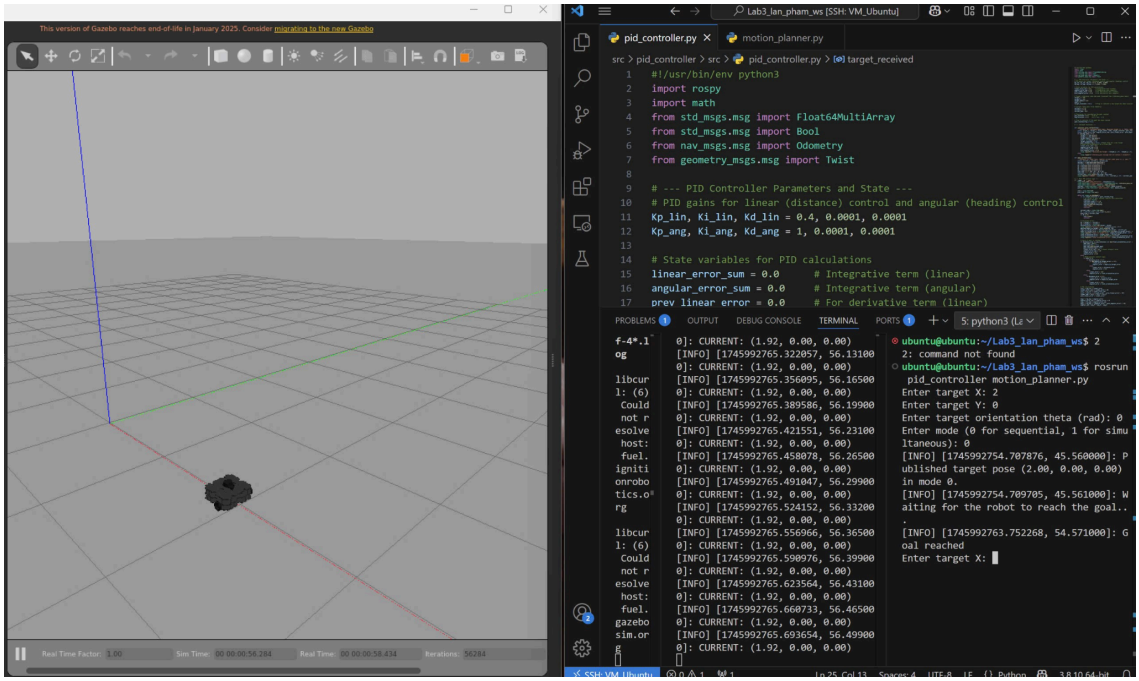
3. Results

The controller-planner pair was exercised four times in Gazebo, exactly as shown in the terminal log below. Each run ended with the *Goal reached* message, confirming that the robot's final pose satisfied the required tolerances (≤ 0.10 m in position, ≤ 0.10 rad in orientation).

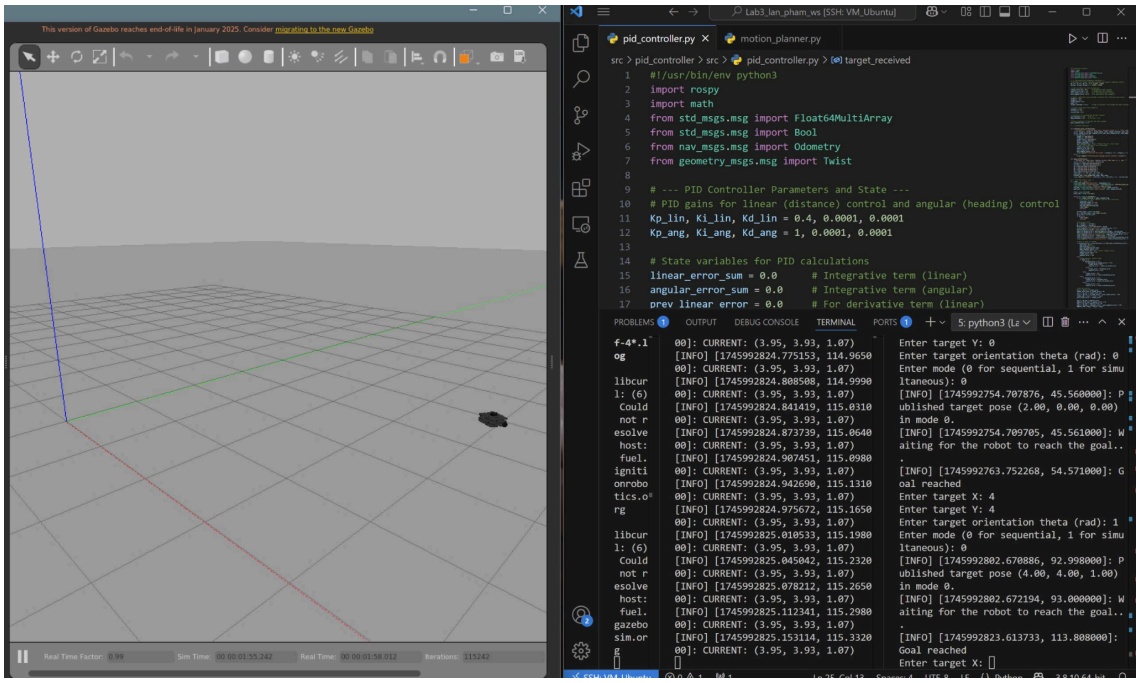
Test #	Mode	Goal Pose (x, y, θ)	Elapsed Time	Path Characteristics
1	0 (sequential)	(2, 0, 0 rad)	≈ 9 s	Turn in place to face +x, straight-line drive, zero heading correction at goal.
2	0 (sequential)	(4, 4, 1 rad)	≈ 21 s	Turn → go straight segment to (4,4) corner → final 1 rad (~57°) clockwise rotation.
3	1 (simultaneous)	(0, 0, 0.5 rad)	≈ 26 s	Broad, smooth arc returning to the origin while steering toward final heading.
4	1 (simultaneous)	(3, 2, 1 rad)	≈ 16 s	Curved "cut-in" trajectory; linear and angular errors decayed together, no stop-and-turn phase.

*Elapsed time was measured from the moment the goal was published until the `/goal_reached` flag was emitted.

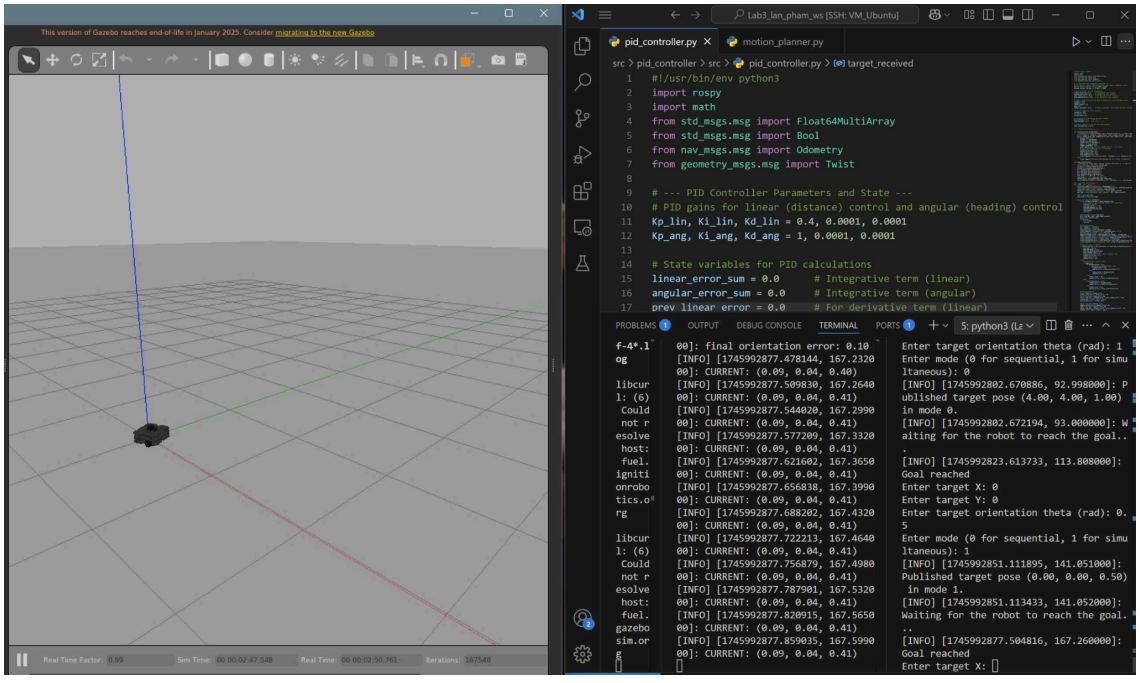
- **Fig. 1 – Mode 0, Goal (2, 0, 0)**



- **Fig. 2 – Mode 0, Goal (4, 4, 1)**



- **Fig. 3 – Mode 1, Goal (0, 0, 0.5)**



- **Fig. 4 – Mode 1, Goal (3, 2, 1)**