

# REPORT ON TURTLEBOT SIMULATION GOALS

This report summarizes the approach, assumptions, algorithm choices, and potential improvements for the implemented TurtleBot simulation goals.

**goal1\_control\_turtle.py:** This script implements a basic turtle control system using a Proportional-Integral-Derivative (PID) controller. The script begins by initializing a ROS node, subscribing to the `/turtle1/pose` topic to receive the turtle's current position and orientation, and publishing to the `/turtle1/cmd_vel` topic to control the turtle's movement. A PID class is defined to handle the control logic. The update method of this class calculates the control output based on the current error (difference between the turtle's current position and the target position), the integral of the error (accumulated error over time), and the derivative of the error (rate of change of the error). The script then calculates the distance and angle errors to the goal. These errors are fed into the PID controller to generate linear and angular velocity commands. These commands are then published to control the turtle's movement. The script includes a loop that continues until the turtle reaches the goal (distance error is below a threshold) or the ROS node shuts down. Finally, it publishes data to ROS topics for visualization using `rqt_plot`, allowing for monitoring of error, control input, and position over time. The random generation of start and goal positions ensures variability in each run. The `rqt_plot` is added in the video for Goal1. 📺 Goal1.mp4

**goal2\_make\_grid.py:** This script extends the functionality of `goal1_control_turtle.py` by adding acceleration and deceleration limits to the turtle's motion. It introduces `max_acceleration` and `max_deceleration` parameters to constrain the rate of change of the linear velocity. The script defines a list of waypoints representing a grid pattern. The core logic remains similar to Goal 1, using a PID controller to guide the turtle to each waypoint. However, before publishing the velocity commands, the script calculates the allowed velocity change based on the acceleration and deceleration limits. This ensures that the turtle's velocity doesn't change too abruptly. The `rotate` function handles the turtle's rotation to the correct orientation at each waypoint. The script iterates through the waypoints, moving the turtle to each one and then rotating to face the next waypoint before proceeding. The addition of acceleration and deceleration limits makes the turtle's movement more realistic and smoother. The `rqt_plot` is added in the video for Goal2 📺 Goal2.mp4

**goal3\_rotate\_turtle\_in\_circle.py:** This script focuses on making the turtle move in a continuous circular path. It initializes a ROS node and publishes velocity commands to the `/turtle1/cmd_vel` topic. The script calculates the required linear and angular velocities to achieve circular motion using the formula  $\omega = v / r$ , where  $\omega$  is the angular velocity,  $v$  is the linear velocity, and  $r$  is the radius of the circle. These velocities are then published to the `/turtle1/cmd_vel` topic. The script also publishes the turtle's real pose to the `/rt_real_pose` topic and a noisy version of the pose (with added Gaussian noise) to the `/rt_noisy_pose` topic at regular intervals. The noise is added to simulate sensor inaccuracies. 📺 Goal3.mp4

**goal4\_chase\_turtle\_fast.py:** This script simulates a chase scenario between two turtles: a "Robber Turtle" (RT) and a "Police Turtle" (PT). RT moves in a circle using the logic similar to Goal 3. PT is spawned after a delay and uses the real-time pose of RT (obtained from the `/rt_real_pose` topic) to chase it. PT's velocity is dynamically adjusted based on the distance to RT. The script uses a `chase_rt` function that calculates the direction and speed needed to intercept RT. The PT's speed is limited by `max_speed`, and acceleration and deceleration are controlled to make the movement smoother. The chase continues until PT is within a certain distance of RT. The use of threading allows RT to move continuously in its circle while PT simultaneously chases it. 📺 Goal4.mp4

**goal5\_chase\_turtle\_slow.py:** In this goal, the Police Turtle (PT) moves at half the speed of the Robber Turtle (RT), requiring a more strategic approach to catching it. To achieve this, an ambush strategy is implemented, where PT predicts RT's future position based on its circular trajectory and moves towards a calculated interception point. This reduces unnecessary chasing and compensates for PT's speed limitation. A simpler alternative could have been direct pursuit, where PT continuously follows RT's current position; however, this would have resulted in inefficient chasing and prolonged the capture time. 📺 Goal5.webm

**goal6\_chase\_turtle\_noisy.py:** In this scenario, PT must chase RT while only receiving noisy position updates every 5 seconds. To counteract the measurement noise, a Kalman Filter is used to estimate RT's actual position by combining past measurements and motion dynamics. This filtering helps PT make more accurate decisions, despite unreliable observations. An alternative approach could have been moving average smoothing or assuming RT follows a perfect circular path, but these methods would be less adaptive to variations in RT's movement. Although the technique seems sophisticated the PT was not able to chase RT due to the noisy data.

📺 Goal6.mp4

### **Assumptions:**

Unimpeded Movement : It was assumed that the turtles could move freely without any obstacles or collisions.

Instantaneous Communication (Goal 4, 6): For Goals 4 - 6, it was assumed that the communication between the turtles (i.e., the transmission of RT's pose to PT) was instantaneous and without delay.

### **Algorithm/Tool Choices:**

ROS (Robot Operating System): ROS was used as the framework for building the simulation. Its features for publishing and subscribing to topics, creating nodes, and managing services were crucial for the project.

PID Controller (Goals 1, 2): A PID controller was chosen for its effectiveness in controlling systems with continuous feedback. It provided a robust way to regulate the turtle's movement towards the goal.

Simple Control Loop (Goal 3): A simple control loop was sufficient for Goal 3 because the desired behavior (circular motion) was straightforward to implement.

### **Potential Improvements (Given More Time):**

More Sophisticated Control Algorithms: Explore more advanced control algorithms, such as reinforcement learning (RL), to handle more complex scenarios and improve performance.

Dynamic Environment: Create a dynamic environment with moving obstacles or changing goal positions to test the robustness of the control algorithms.

In the Goal 1, performance variation for various gain values was not implemented due to time crunch and lack of experience in using dynamic reconfigure for the gain variations.

For Goal 4, a 30+ unit radius is impractical in Turtlesim's 11x11 grid, as it exceeds the environment's boundaries. Even a 5-unit circle nearly fills the space, leaving PT with little room to maneuver without hitting the edges. Additionally, PT's random spawn point may place it outside a feasible chasing area, further restricting pursuit. Maintaining a 3-unit capture threshold in such a small world makes smooth chases nearly impossible. This setup is designed for larger environments, and a  $\leq 5$ -unit radius with a 0.5 unit threshold would be more realistic.