Flytbase Assignment Report For Robotics Intern Role

Submitted by: Athrva Kulkarni || 8010353482 || athrvakulkarni11@gmail.com

Video Submissions: FlytBase Assignments Videos
GitHub Repo : https://github.com/athrvakulkarni11/FlytBase-Assignments.git

Assignment Report

Go to Results

1. Goal-wise Solution Breakdown

Goal 1: Control Turtle with PID

Objective

The goal is to control a turtle in the ROS TurtleSim environment using a PID (Proportional-Integral-Derivative) controller. The turtle starts at a random position and needs to navigate to a predefined goal in the 2D environment. The PID controller ensures the turtle reaches the goal as quickly as possible without overshooting or oscillating.

Code Description

The code implements a ROS2 node (`TurtleBot`) that controls the turtle's movement by subscribing to its pose and publishing velocity commands. The node uses a PID controller to compute linear and angular velocities based on the current pose and the goal position.

Detailed Code Breakdown

1. **Initialization (`__init__` method)**:
    - Declares dynamic parameters for the PID controller, maximum speed, goal position, and tolerance.
    - Sets up a publisher (`/turtle1/cmd_vel`) to control the turtle's movement and a subscriber (`/turtle1/pose`) to receive the turtle's pose updates.
    - Initialises a timer to execute the control loop (`move_towards_goal`) every 0.1 seconds.
2. **Dynamic Parameter Setup (`initialize_variables` method)**:

    - Initialises the PID control variables, including proportional, integral, and derivative gains for both linear and angular velocity control.
    - Sets default values for the goal position and tolerance.
    - Resets error terms to ensure the PID controller starts from a clean state.
3. **PID Controller (`pid_controller` method)**:
    - Computes the control output based on the current error (`error`), accumulated error (`error_sum`), and the change in error (`last_error`).
    - Adjusts the linear and angular velocities using the PID formula:
        - **Proportional (P)**: Adjusts based on the current error.
        - **Integral (I)**: Accounts for the accumulated error to reduce steady-state error.
        - **Derivative (D)**: Predicts future error based on the rate of change.
    - Returns the control output and updates the accumulated error to avoid windup.
4. **Control Logic (`move_towards_goal` method)**:
    - Computes the distance to the goal and the required steering angle.
    - Uses the PID controller to compute linear and angular velocities.
    - Normalises the angular velocity to keep it within the range of `[-pi, pi]`.
    - Caps the velocities to the specified maximum linear and angular speeds.
    - Publishes the velocity commands to the turtle. If the turtle is within the goal tolerance, it stops and logs the "Goal reached!" message.
5. **Parameter Callback (`parameters_callback` method)**:
    - Provides a mechanism to update parameters at runtime. If the `reset_controller` parameter is set to `True`, it triggers the reset logic to reinitialize the controller.

Summary

This code demonstrates a comprehensive implementation of a PID controller for navigating a turtle in the TurtleSim environment. By incorporating dynamic parameter reconfiguration, logging, reset features, and a modular design, the code provides flexibility and ease of use, making it an effective solution for controlling the turtle's movement towards a goal.

Goal 2: Make a Grid

Objective

The goal is to control the turtle in the TurtleSim environment to move along a predefined grid pattern. The movement should adhere to realistic acceleration and deceleration limits, simulating the natural behaviour of a vehicle. The turtle must align itself to the grid points using a PID controller while avoiding abrupt changes in speed. The primary aim is to ensure smooth navigation across the grid with precise stopping at each point.

Code Description

The code implements a ROS2 node (`TurtleBotPIDController`) that uses a PID controller to navigate the turtle through a series of grid points. It incorporates realistic movement constraints by introducing maximum acceleration and deceleration limits. The turtle aligns itself to each grid point and moves towards it while adhering to the set limits.

Detailed Code Breakdown

1. **Initialization (`__init__` method)**:
    - Declares dynamic parameters for PID control, speed limits, acceleration/deceleration constraints, and other control settings.
    - Initialises the publisher to send velocity commands (`/turtle1/cmd_vel`) and subscriber to receive pose updates (`/turtle1/pose`).
    - Creates a control loop timer (`control_loop`) that runs every 0.1 seconds to control the turtle's movement.
2. **Variable Initialization (`initialize_variables` method)**:
    - Initialises all dynamic parameters, including the PID gains, maximum speed, acceleration limits, and tolerances for goal-reaching.
    - Defines the grid points (`grid_goals`) that the turtle will navigate to.
    - Resets error terms for the PID controller to ensure smooth and controlled movement.
3. **PID Controller (`pid_controller` method)**:
    - Calculates control outputs using the PID formula:
        - **Proportional (P)**: Adjusts based on the current error.
        - **Integral (I)**: Accumulates errors over time to address steady-state errors.
        - **Derivative (D)**: Predicts future error based on the rate of change.
    - Returns the control output and the updated accumulated error, ensuring smooth navigation.
4. **Control Loop (`control_loop` method)**:
    - This is the main control logic executed periodically to align the turtle to the grid points and move it toward the next point.
    - **Alignment Phase**:
        - Calculates the heading error to align the turtle to the direction of the next grid point using angular PID control.
        - If the turtle is within the angular tolerance (`angle_tolerance`), it stops rotating and transitions to the movement phase.
    - **Movement Phase**:
        - Uses linear PID control to move the turtle towards the current goal while ensuring smooth acceleration.
        - Calls `limit_acceleration` to ensure the turtle's speed changes gradually, avoiding abrupt acceleration or deceleration.
    - **Transition to Next Goal**:
        - Once the turtle reaches a grid point within the defined `goal_tolerance`, it transitions to the next goal in the grid.
5. **Acceleration Limiting (`limit_acceleration` method)**:

    - Limits the rate of change of velocity to avoid abrupt movements. Ensures that the turtle accelerates or decelerates smoothly within the defined limits (`max_acceleration`, `max_deceleration`).
6. **Parameter Callback (`parameters_callback` method)**:
    - Provides a mechanism for dynamic parameter updates. Adjusts control parameters and logs the changes if logging is enabled.

Summary

This code controls the turtle in the TurtleSim environment to navigate through a series of grid points using a PID controller. It introduces acceleration and deceleration constraints to simulate realistic vehicle behaviour, ensuring smooth navigation and precise stopping at each grid point. By incorporating dynamic parameter reconfiguration, logging, reset features, and modular methods, the code provides a flexible and robust solution for controlled movement in a grid pattern.

Goal 3: Rotate the Turtle in a Circle

Objective

The goal is to control a turtle in the TurtleSim environment to continuously move in a circular path. Additionally, the turtle's pose should be published periodically, both as the actual pose and with added Gaussian noise, to simulate sensor inaccuracies. The implementation should allow dynamic reconfiguration of movement parameters like speed, radius, and noise level, while providing smooth circular motion with acceleration constraints.

Code Description

The code implements a ROS2 node (`TurtleCircles`) that makes the turtle move in a circular path while providing control over its linear and angular speeds. It includes the capability to add Gaussian noise to the turtle's pose to simulate real-world sensor data inaccuracies. The node publishes both the real and noisy poses at regular intervals, supporting dynamic parameter reconfiguration and a reset feature to restart the turtle's motion.

Detailed Code Breakdown

1. **Initialization (`__init__` method)**:
    - Declares dynamic parameters for controlling the turtle's motion, including `linear_speed`, `angular_speed`, `radius`, `noise_std_dev`, and `max_acceleration`.
    - Sets up publishers for velocity commands (`/turtle1/cmd_vel`), real poses (`/rt_real_pose`), and noisy poses (`/rt_noisy_pose`).
    - Initialises pose subscribers to receive the turtle's pose updates (`/turtle1/pose`).
    - Creates timers:

        - One for controlling the turtle's circular motion (`move_in_circle`), executed every 0.1 seconds.
        - Another for publishing poses (`publish_pose`), executed every 5 seconds.
2. **Parameter Initialization (`initialize_variables` method)**:
    - Initializes or reinitializes all dynamic parameters and internal state variables.
    - Sets the turtle's current speed to zero, ensuring a smooth start to the circular motion.
    - Logs the initial parameters if logging is enabled.
3. **Parameter Callback (`parameters_callback` method)**:
    - Provides a mechanism for updating parameters dynamically.
    - Updates internal variables based on parameter changes and logs the changes if `enable_logging` is `True`.
    - Resets the motion if the `reset_motion` parameter is set, allowing for the reinitialization of the turtle's movement.
4. **Pose Update (`on_pose_update` method)**:
    - Updates the current pose of the turtle as received from the `/turtle1/pose` topic.
    - The updated pose is used in the control loop and pose publishing.
5. **Acceleration Limiting (`limit_acceleration` method)**:
    - Limits the rate of change of velocity to avoid abrupt accelerations, ensuring smooth circular motion.
    - Adjusts the turtle's linear and angular velocities incrementally within the specified acceleration limits (`max_acceleration`).
6. **Circular Movement (`move_in_circle` method)**:
    - The main control loop for moving the turtle in a circle.
    - Computes target linear and angular speeds for circular motion and adjusts the current speeds using `limit_acceleration`.
    - Publishes velocity commands to move the turtle, ensuring that the circular motion is smooth and within the specified parameters.
7. **Pose Publishing (`publish_pose` method)**:
    - Publishes both the real and noisy poses of the turtle at regular intervals (every 5 seconds).
    - The real pose is published on the `/rt_real_pose` topic.
    - Adds Gaussian noise to the turtle's pose and publishes it on the `/rt_noisy_pose` topic, simulating noisy sensor data.
    - Logs the published pose information if logging is enabled.

Summary

This code enables the turtle in the TurtleSim environment to move in a continuous circular path with configurable speed and radius. It includes a mechanism to publish the turtle's pose both as real and noisy data, simulating sensor noise. By incorporating dynamic reconfiguration, logging, and reset features, the code offers a flexible and robust framework for circular motion control. The modular design allows for easy adjustment of parameters,

smooth acceleration control, and detailed logging of the turtle's behaviour, making it an effective solution for simulating realistic robotic movement in a 2D space.

Goal 4: Chase the Turtle Fast and Goal 5: Chase Turtle Slow

Objective

The objective of Goal 4 and 5 is to simulate a chase scenario in the TurtleSim environment where a "Robber Turtle" (RT) moves in a circular path, and a "Police Turtle" (PT) attempts to catch it. RT follows a predefined circular trajectory while PT spawns at a random location 10 seconds later and chases RT using i qts periodically published real position. PT must use the information from the RT's past movements to predict and intercept its future path efficiently. The chase is considered successful when the distance between PT and RT is less than or equal to 0.5 units.

Note: Code For Goal 4 and Goal 5 Are Exactly Same In My Case

Code Descriptions

1. Robber Turtle (RobberTurtle) Code

Class Overview: The `RobberTurtle` class simulates the RT moving in a circular path. It periodically publishes both the real and noisy poses of the turtle to specific ROS2 topics. The turtle stops moving when PT catches it.

Code Breakdown:

1. **Initialization (`__init__` method)**:
    - Declares dynamic parameters like `circle_radius`, `angular_velocity`, `noise_stddev`, and `enable_logging`.
    - Creates publishers for velocity commands, real pose (`/rt_real_pose`), and noisy pose (`/rt_noisy_pose`).
    - Sets up subscribers to listen to the turtle's current pose (`/turtle1/pose`) and catch events (`/catch_event`).
    - Starts timers for continuous circular motion (every 0.1 seconds) and pose publishing (every 5 seconds).
2. **Parameter Initialization (`initialize_variables` method)**:
    - Initialises dynamic parameters and state variables like `caught` to determine if RT should stop.
    - Logs initialization if logging is enabled.
3. **Parameter Callback (`parameters_callback` method)**:
    - Handles dynamic parameter updates and resets the turtle's state if the reset flag is set.
4. **Movement Control (`move_in_circle` method)**:

    - Commands RT to move in a circular path with constant speed and radius unless it is caught or reset.
    - Publishes velocity commands to make the turtle follow the circle.
5. **Pose Publishing (`publish_poses` method)**:
    - Publishes both real and noisy poses of RT every 5 seconds.
    - Adds Gaussian noise to the real pose to simulate noisy sensor data.
6. **Stop Movement (`stop_moving` method)**:
    - Stops the turtle when a catch event is received from PT, indicating that the chase is over.

2. Police Turtle (PoliceTurtle) Code

Class Overview: The `PoliceTurtle` class spawns a second turtle (PT) in the TurtleSim environment to chase RT. It uses RT's periodic pose information to predict its future positions and tries to catch RT by intercepting its circular path.

Code Breakdown:

1. **Initialization (`__init__` method)**:
    - Declares dynamic parameters including `catch_threshold` and `enable_logging`.
    - Sets up publishers for velocity commands (`/turtle2/cmd_vel`) and catch events (`/catch_event`).
    - Subscribes to RT's real pose (`/rt_real_pose`) and PT's pose (`/turtle2/pose`).
    - Initialises state variables and sets up timers:
        - One for spawning PT after 10 seconds.
        - Another for the control loop (every 0.1 seconds) to chase RT.
2. **Parameter Initialization (`initialize_parameters` method)**:
    - Initialises and resets all dynamic parameters and state variables like `pt_has_caught_rt` and `pattern_detected`.
    - Logs the reset status if `reset_controller` is set.
3. **Update Callbacks (`update_rt_pose` and `update_pt_pose` methods)**:
    - Update RT's real pose and PT's current pose, respectively.
    - Uses RT's pose history to detect its circular motion pattern.
4. **Spawning PT (`spawn_pt` and `spawn_callback` methods)**:
    - Spawns PT at a random location 10 seconds after RT starts moving.
    - Initiates the chase once PT is successfully spawned.
5. **Movement Control (`control_loop` method)**:
    - Main control loop for PT's movement.
    - Uses RT's past poses to detect its circular pattern and predict its future positions.
    - Calls `move_to_target` to intercept RT based on the predicted future positions.
6. **Move to Target (`move_to_target` method)**:
    - Commands PT to move toward the predicted position of RT.

    - Uses proportional control to set PT's linear and angular velocities based on the distance and angle to the target.
7. **Pattern Detection (`detect_circular_motion` method)**:
    - Analyses RT's past 10 poses to identify its circular motion.
    - Uses least squares fitting to estimate the circle's center and radius, aiding in the prediction of RT's future positions.
8. **Future Position Prediction (`predict_rt_future_position` method)**:
    - Predicts RT's future position on the circle based on its angular velocity and the time ahead (e.g., 5 seconds).
9. **Event Handling (`send_catch_event` and `stop_turtle` methods)**:
    - Publishes a catch event and stops PT when it catches RT.

Summary

In Goal 4 and 5, the Robber Turtle (RT) moves in a circular path while the Police Turtle (PT) spawns after a delay and attempts to catch RT by analyzing its movement patterns. RT publishes its real pose every 5 seconds, which PT uses to detect RT's circular motion and predict future positions. PT employs a simple proportional control mechanism to intercept RT, and the chase is successful when PT comes within a 0.5-unit distance of RT.

Both the `RobberTurtle` and `PoliceTurtle` codes are equipped with dynamic reconfiguration capabilities, detailed logging options, and reset features, making the simulation adaptable and insightful. The modular design ensures that each component of the chase behavior, from circular movement to pattern detection, is clearly defined and maintainable.

Goal 6: Chase the Turtle with Noisy Data

Objective

In Goal 6, the objective is to simulate a more challenging chase scenario in the TurtleSim environment where the "Police Turtle" (PT) attempts to catch the "Robber Turtle" (RT). In this setup, RT publishes a noisy version of its pose every 5 seconds, simulating sensor inaccuracies. PT, which moves at half the speed of RT, must use an estimator (Extended Kalman Filter, EKF) to predict RT's future positions accurately and intercept it. The chase is considered successful when PT comes within a 1.5-unit distance of RT.

Code Descriptions

1. Robber Turtle (RobberTurtle) Code

Overview: The `RobberTurtle` code is the same as in Goal 4. RT moves in a circular path, periodically publishing its real and noisy poses to specific ROS2 topics. This noisy pose introduces challenges for PT in predicting RT's trajectory accurately.

2. Police Turtle (PoliceTurtle) Code

Class Overview: The `PoliceTurtle` class represents PT, which tries to catch RT using its noisy pose data. It utilizes an Extended Kalman Filter (EKF) to filter the noisy pose data and predict RT's future positions based on a detected circular motion pattern.

Code Breakdown:

1. **Initialization (`__init__` method)**:
    - Declares dynamic parameters such as `catch_threshold`, `pt_linear_speed`, `prediction_time_ahead`, and more.
    - Sets up publishers for velocity commands (`/turtle2/cmd_vel`) and catch events (`/catch_event`).
    - Subscribes to RT's noisy pose (`/rt_noisy_pose`), RT's real pose (`/rt_real_pose`), and PT's pose (`/turtle2/pose`).
    - Initializes the Extended Kalman Filter (EKF) for noisy pose estimation.
    - Starts timers for spawning PT after 10 seconds and the control loop (every 0.1 seconds).
2. **EKF Initialization (`initialize_ekf` method)**:
    - Configures the EKF with state transition and measurement matrices to estimate RT's pose based on noisy sensor data.
    - Sets up the covariance matrix (`P`), measurement noise (`R`), and process noise (`Q`).
3. **Noisy Pose Update and EKF Update (`update_rt_noisy_pose` method)**:
    - Updates RT's noisy pose using the data received on the `/rt_noisy_pose` topic.
    - Uses EKF to filter this noisy data and predict the actual position of RT.
    - Stores the predicted pose in a history buffer for circular pattern detection.
4. **Real Pose Update for Catch Checking (`update_rt_real_pose` method)**:
    - Updates RT's real pose for checking the catch condition in the control loop.
    - Ensures PT stops the chase upon successfully catching RT.
5. **Spawning PT (`spawn_pt` and `spawn_callback` methods)**:
    - Spawns PT at a random location after a delay of 10 seconds.
    - Initiates the chase once PT is successfully spawned.
6. **Control Loop (`control_loop` method)**:
    - The main control loop for PT's movement.
    - Uses EKF-filtered data to detect RT's circular motion and predict its future positions.
    - Calls `move_to_target` to move PT toward the predicted future position of RT.
7. **Move to Target (`move_to_target` method)**:
    - Commands PT to move toward the predicted target position using proportional control.
    - Computes linear and angular velocities to guide PT smoothly towards the predicted point.
8. **Circular Motion Detection (`detect_circular_motion` method)**:

    - Analyses RT's past poses (filtered by EKF) to detect its circular motion pattern.
    - Uses least squares fitting to identify the circle's centre and radius, aiding in future position prediction.
    - Applies a smoothing filter (Savitzky-Golay) to reduce noise in the historical pose data before detecting the pattern.
9. **Future Position Prediction (`predict_rt_future_position` method)**:
    - Predicts RT's future position on the circle using the detected motion pattern and estimated angular velocity.
    - Calculates the future x and y coordinates based on the circle's properties.
10. **Event Handling (`send_catch_event` and `stop_turtle` methods)**:
    - Sends a catch event to indicate that PT has caught RT.
    - Stops PT by setting its velocities to zero when the chase is complete.
11. **Reset and Parameter Callback (`parameters_callback` method)**:
    - Handles dynamic parameter updates and resets PT's state if the reset flag is set.

Summary

In Goal 6, the challenge is to track and intercept the Robber Turtle (RT) when it publishes noisy pose data, simulating real-world sensor inaccuracies. The Police Turtle (PT) uses an Extended Kalman Filter (EKF) to estimate RT's true pose from noisy measurements. PT further detects RT's circular motion pattern to predict its future positions and intercepts RT efficiently, despite moving at half the speed of RT.

The use of EKF and pattern recognition makes this setup a complex estimation and control problem. Both the `RobberTurtle` and `PoliceTurtle` codes are designed to handle dynamic parameter changes, logging, and resetting. The modular approach ensures clarity in managing the various components of the chase behavior, including noise filtering, motion detection, and control strategies.

---

2. Results and Visualisations:

Figure 1 shows the execution of Goal 1 with a fully tuned PID controller, which was dynamically adjusted using rqt dynamic reconfiguration.

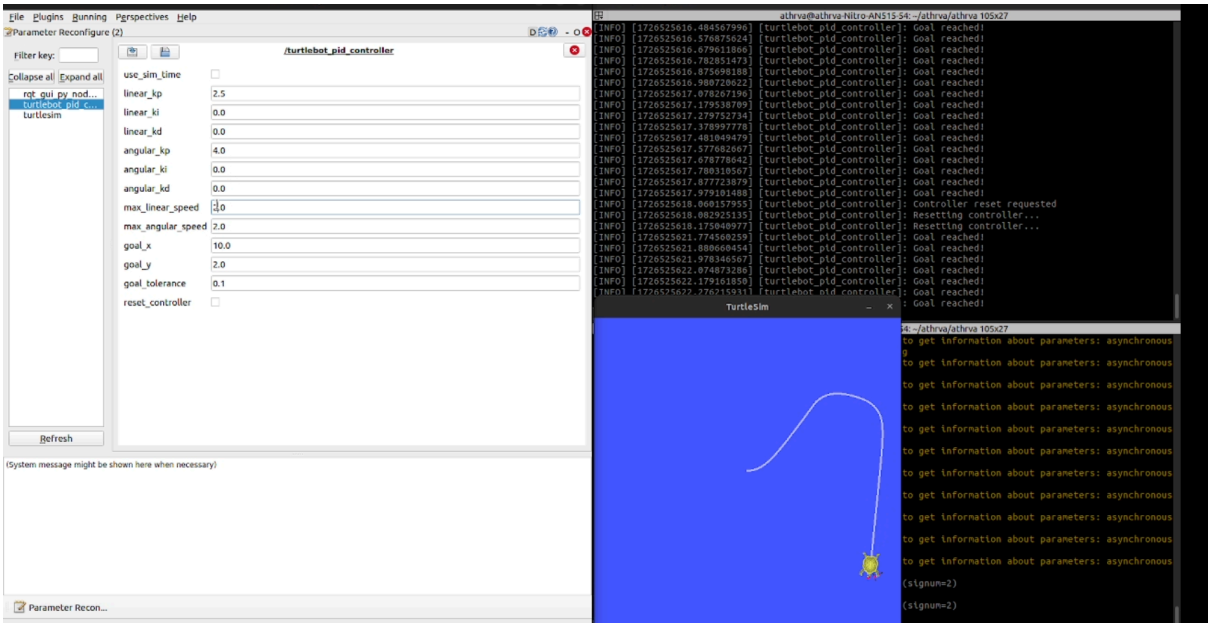


Figure 2 shows the execution of Goal 2 with a fully tuned PID controller,With Exact Grid Pattern

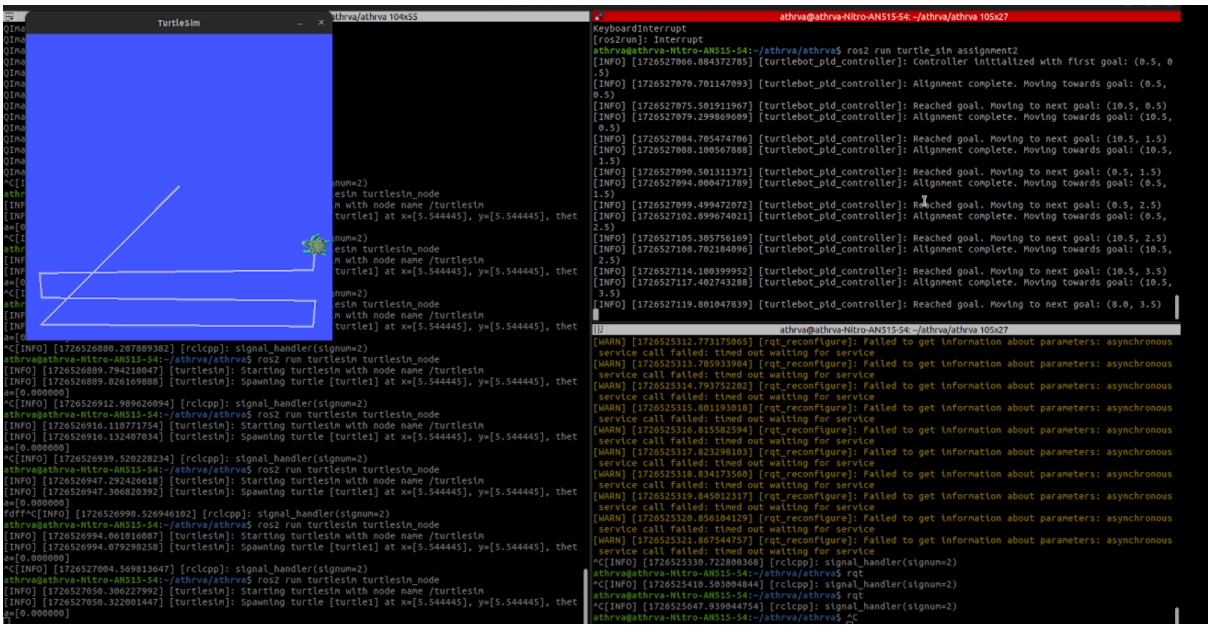


Figure 3 shows the execution of Goal 3 /Robber Turtle Also Publishing Of rt_real_pose and rt_noisy_pose

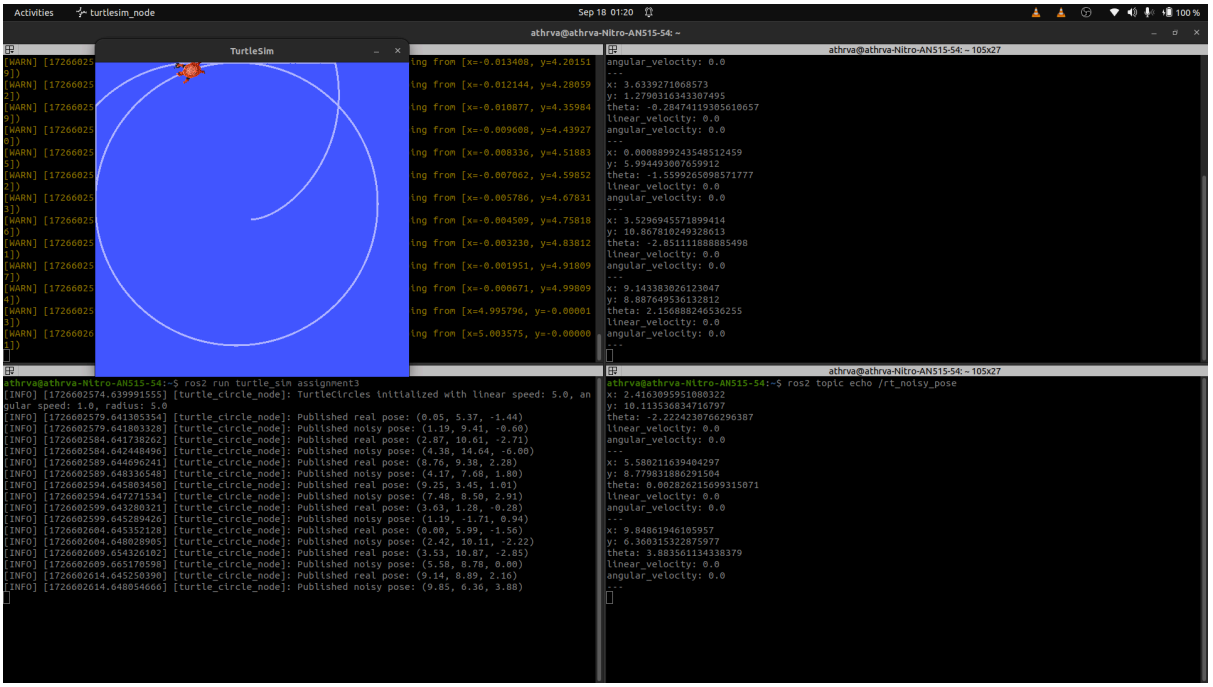


Figure 4 shows the execution of Goal 4 and 5 Robber Turtle and Police Turtle , With PT predicting The Future Position of RT to Catch It

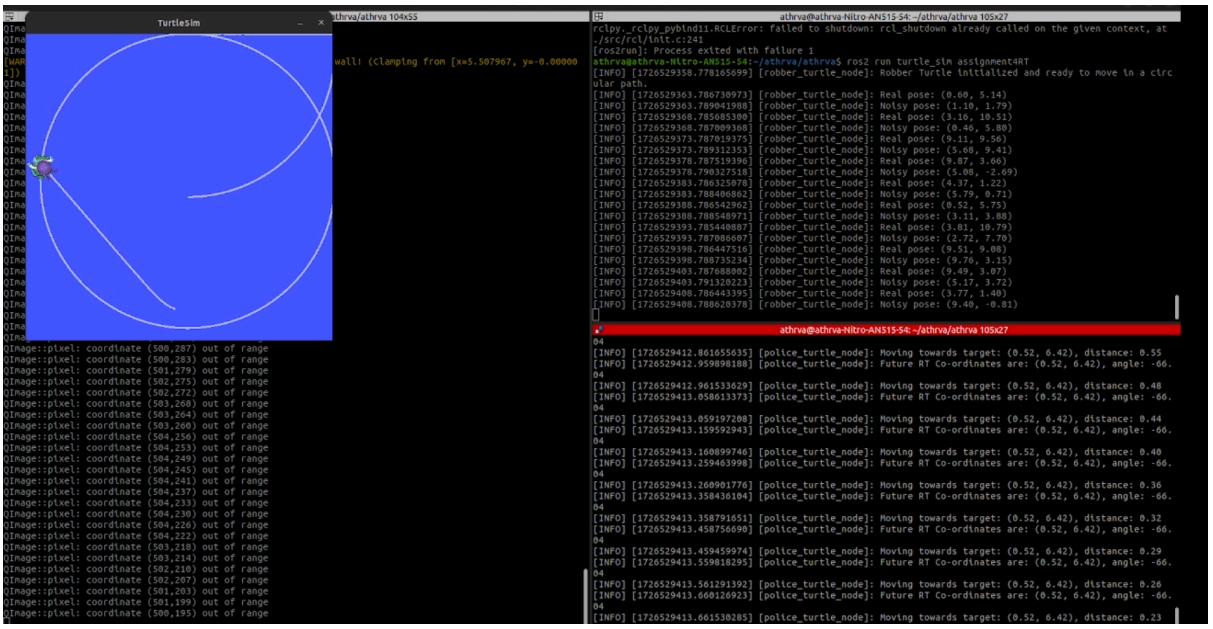


Figure 5 shows the execution of Goal 6 Robber Turtle and Police Turtle , With PT predicting The Future Position of RT to Catch It , Had to Use Various Methods to Reduce Noise

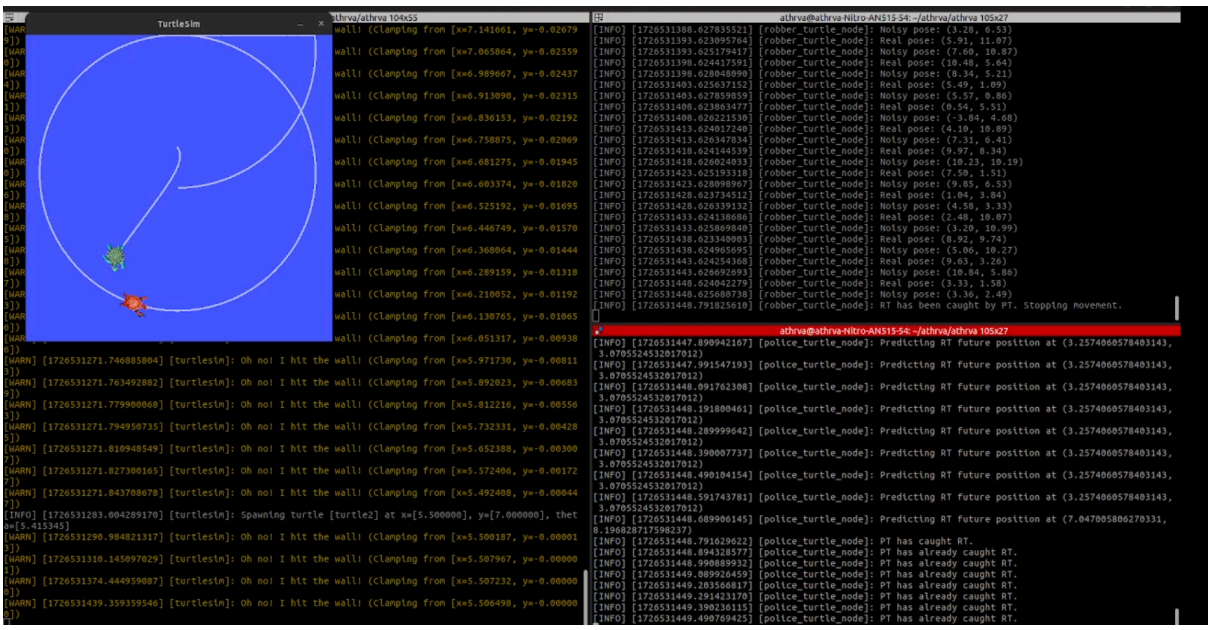


Figure 6 Shows Dynamic Reconfiguration Setup For Tuning PID's ,Code Handling , Changing Velocity Turning Logging On/Off , Resetting Controller

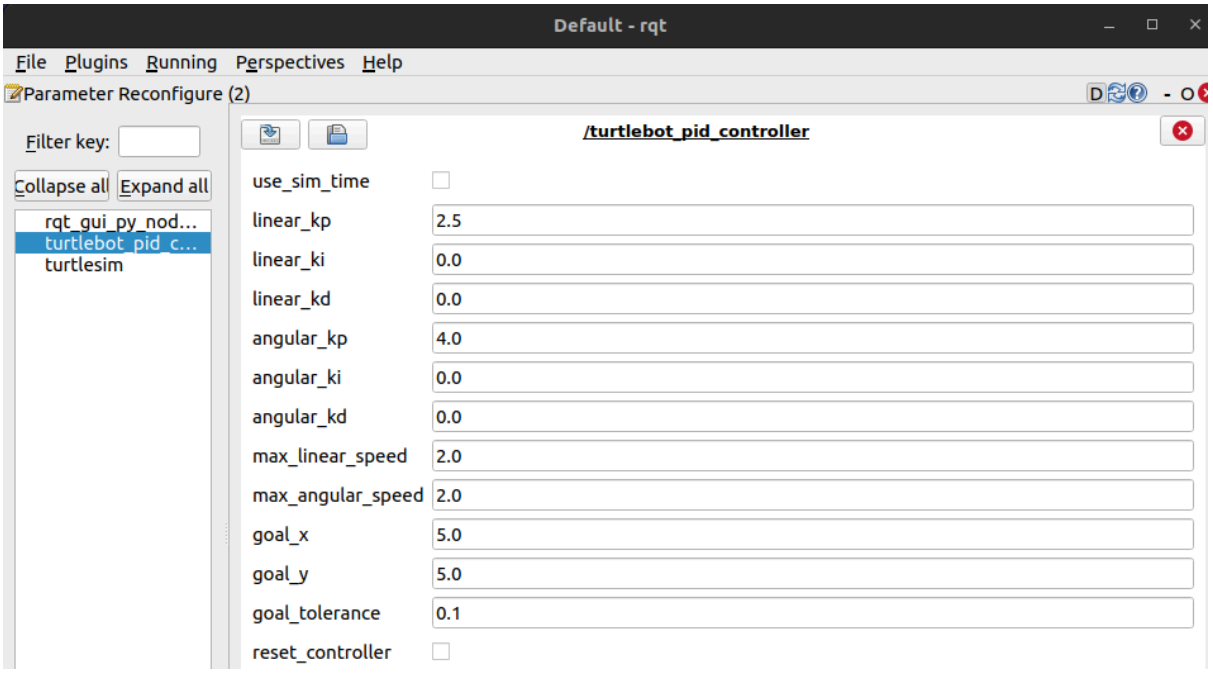
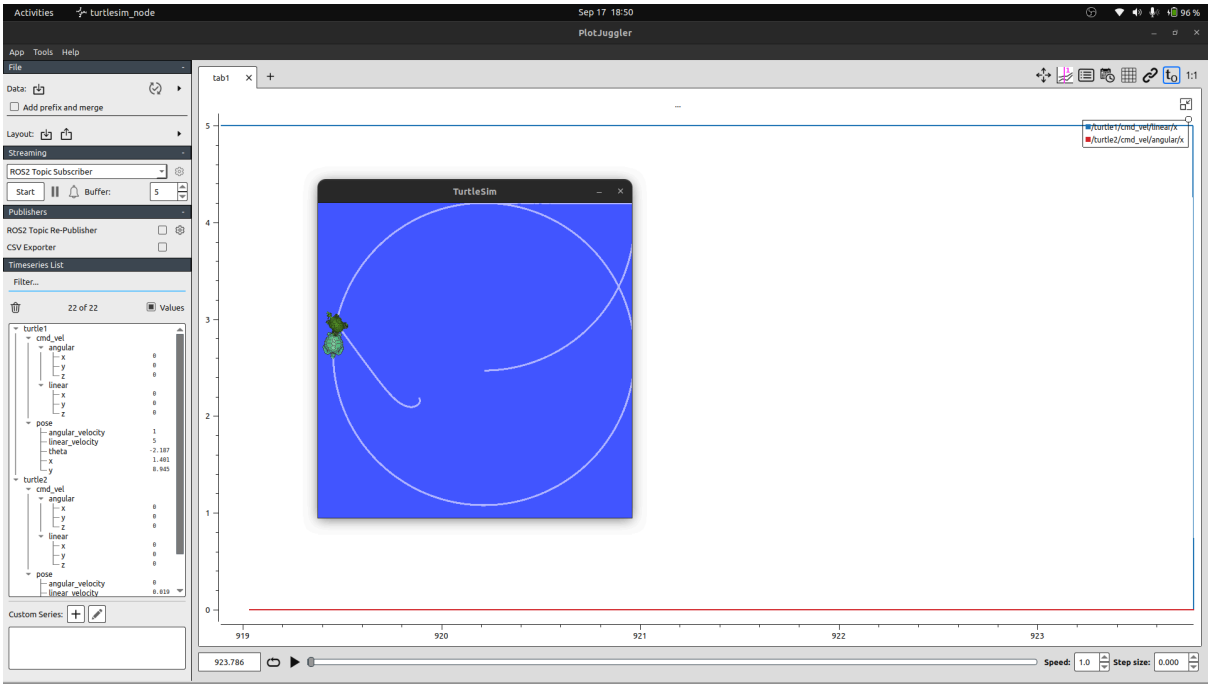
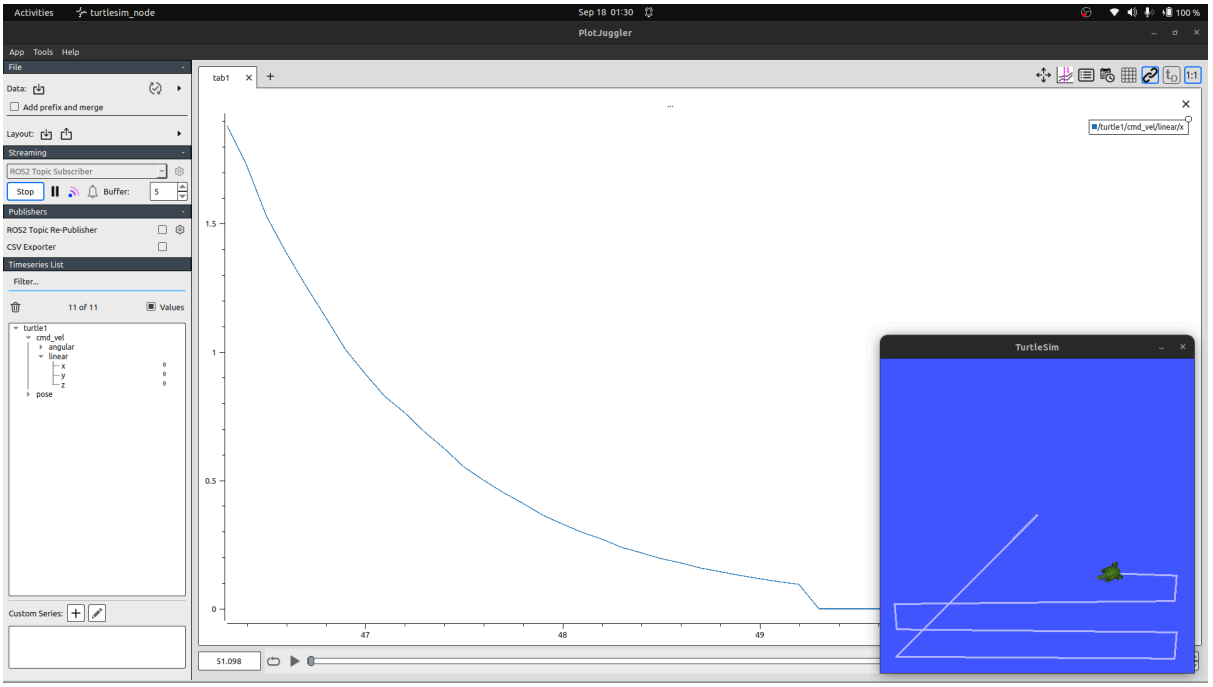


Figure 7 and 8 Show PlotJuggler Variations For The Velocities For Goal 2 and Goal 4





3.Key Features Across All Goals

1. **Dynamic Reconfiguration**:
    - The system allows real-time tuning of various parameters such as PID gains, movement speeds, noise levels, and reset flags through ROS2 dynamic reconfiguration. This adaptability enables fine-tuning of the turtles' behaviours without restarting the nodes.
2. **Logging**:
    - Each node includes extensive logging capabilities that can be enabled or disabled dynamically. Logs provide detailed insights into the system's internal state, including control actions, pose updates, detection events, and decision-making processes, aiding in debugging and performance evaluation.
3. **Reset Functionality**:
    - A reset feature allows reinitialization of all control variables and dynamic parameters to their default values. This is useful for re-running simulations, testing different scenarios, and maintaining consistent initial conditions.
4. **Modular Design**:
    - The codebase follows a modular design approach, separating key functionalities into well-defined methods and classes. This makes the code easy to understand, maintain, and extend for further development or experimentation
    -

4. Improvements and Future Work

- **Enhanced Estimation Techniques**: Implementing more advanced filters, such as the Unscented Kalman Filter (UKF), could improve robustness against noise and nonlinear motion.
- **Optimised Path Planning**: For Goals 5 and 6, integrating advanced path-planning algorithms like A* or D* could improve PT's efficiency in catching RT.
- **Adaptive Control**: Implementing adaptive PID control could enable real-time adjustment of gains based on environmental conditions and noise levels.

Given more time, further work could include dynamic noise modeling, enhanced path-planning algorithms, and adaptive control mechanisms for more robust pursuit strategies.

Conclusion

This project successfully demonstrated the application of control theory, motion planning, and noise management in a 2D TurtleSim environment using ROS2. Through dynamic reconfiguration, modular design, and noise estimation techniques, the solutions effectively addressed progressively complex challenges in control, estimation, and tracking.