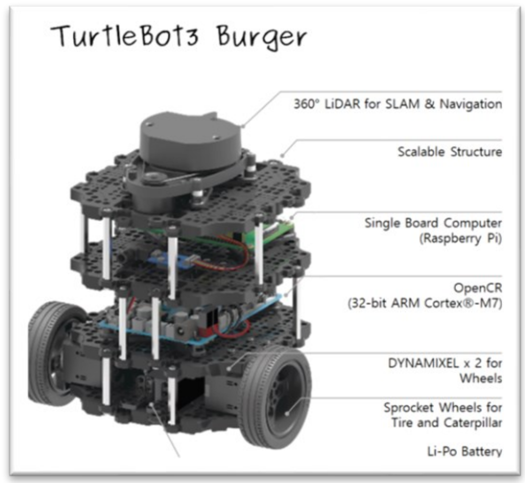
TurtleBot3 PID Controller Challenge

1. Overview & Motivation

Welcome to the TurtleBot3 PID Controller Challenge. This challenge tests fundamental control systems aspects of robotics, more specifically the implementation of a PID controller.



Mobile robotics relies heavily on precise motion control to achieve autonomous navigation tasks. The **TurtleBot3 PID Controller Challenge** focuses on implementing dual PID feedback controllers for accurate trajectory following in a simulated environment.

Why PID Control Matters in Mobile Robotics:

- **Precision**: Enables robots to reach target positions with minimal error
- **Stability**: Maintains consistent performance despite environmental disturbances
- **Adaptability**: Allows fine-tuning for different terrains and payloads
- **Foundation**: Forms the basis for more advanced control algorithms like MPC and LQR

In this challenge, participants will develop independent PID controllers for linear and angular motion, enabling a TurtleBot3 to navigate through waypoints with high accuracy and smooth trajectories.

2. Learning Objectives

By completing this challenge, participants will gain expertise in:

Technical Skills

- **ROS 2 Node Development**: Creating publishers, subscribers, and timers for real-time control
- **PID Control Theory**: Understanding proportional, integral, and derivative control mechanisms
- **Sensor Integration**: Processing odometry data for feedback control
- **Performance Analysis**: Evaluating controller behavior using quantitative metrics

Practical Applications

- **Parameter Tuning**: Systematic approaches to optimize K_p, K_i, and K_d gains
- **Robustness Testing**: Validating controller performance under various conditions
- **System Integration**: Combining multiple control loops for coordinated motion
- **Documentation**: Technical reporting with data visualization and analysis

3. Prerequisites

Software Requirements

- **Ubuntu 22.04 LTS** (recommended)
- **ROS 2 Humble** or **Iron**
- **Gazebo 11** (for Humble) or **Ignition Gazebo** (for Iron)
- **Python 3.8+** or **C++17**

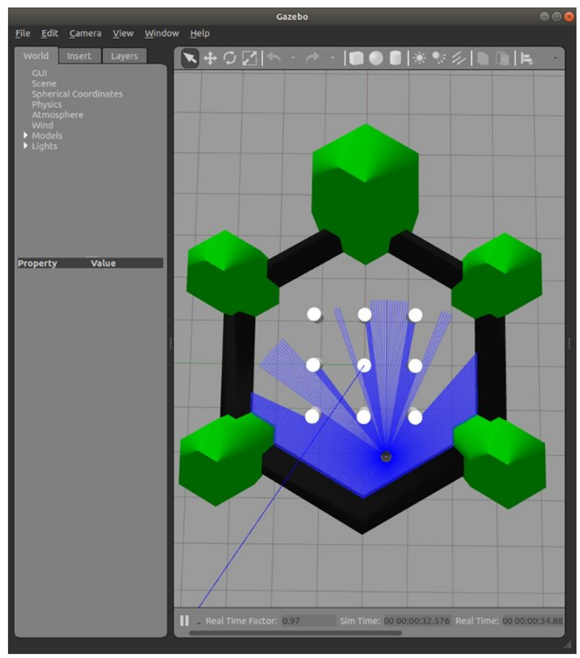
Required Packages

```
# Core ROS 2 packages
sudo apt install ros-humble-desktop-full
sudo apt install ros-humble-turtlebot3*
sudo apt install ros-humble-gazebo-*

# Additional tools
sudo apt install python3-colcon-common-extensions
sudo apt install python3-rosdep python3-argcomplete
```

TurtleBot3 Model

- **Primary**: TurtleBot3 Burger (lightweight, fast simulation)
- **Alternative**: TurtleBot3 Waffle Pi (more realistic dynamics)



Knowledge Prerequisites

- Basic ROS 2 concepts (nodes, topics, messages)
- Python programming fundamentals
- Linear algebra (vectors, coordinate transformations)
- Basic understanding of feedback control systems

4. Environment Setup

Step 1: Create Workspace

```
mkdir -p ~/turtlebot3_pid_ws/src
cd ~/turtlebot3_pid_ws
colcon build
source install/setup.bash
```

Step 2: Set Environment Variables

```
echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
echo "export
GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:~/turtlebot3_pid_ws/install/turtlebot3_gazebo/share/t
urtlebot3_gazebo/models" >> ~/.bashrc
source ~/.bashrc
```

Step 3: Launch TurtleBot3 in Gazebo

```
# Terminal 1: Launch Gazebo world
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py

# Terminal 2: Verify topics
ros2 topic list
# Expected: /odom, /cmd_vel, /scan, etc.
```

Step 4: Test Basic Movement

```
# Terminal 3: Manual control test
ros2 run turtlebot3_teleop teleop_keyboard
```

5. Challenge Tasks

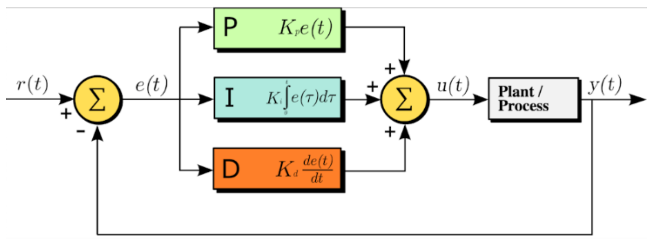
Core Node Requirements

Input/Output Specifications

- **Subscribe to**: `/odom` (nav_msgs/Odometry)
- **Publish to**: `/cmd_vel` (geometry_msgs/Twist)
- **Control Rate**: 10-20 Hz (recommended)

Dual PID Implementation

Mathematical Foundation



The PID control equation for each axis:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \cdot \frac{de(t)}{dt}$$

Where:

- $e(t)$ = error signal (setpoint - measured value)
- K_p = proportional gain
- K_i = integral gain
- K_d = derivative gain

Linear Distance PID

```
# Distance error calculation
distance_error = sqrt((target_x - current_x)² + (target_y - current_y)²)

# PID computation
linear_velocity = Kp_linear * distance_error +
                  Ki_linear * integral_distance_error +
                  Kd_linear * (distance_error - prev_distance_error) / dt
```

Angular Heading PID

```
# Angular error calculation (handle wrap-around)
angular_error = atan2(target_y - current_y, target_x - current_x) - current_yaw
```

```
angular_error = atan2(sin(angular_error), cos(angular_error))  # Normalize to [−π, π]

# PID computation
angular_velocity = Kp_angular * angular_error +
                   Ki_angular * integral_angular_error +
                   Kd_angular * (angular_error − prev_angular_error) / dt
```

Task Breakdown

Task 1: Basic Node Structure

Create a ROS 2 node with:

- Odometry subscriber callback
- Velocity publisher
- Timer-based control loop
- Parameter server integration

Task 2: Waypoint Navigation

Implement sequential navigation through **minimum 3 waypoints**:

Waypoint	X (m)	Y (m)	Tolerance (m)
WP1	2.0	0.0	0.1
WP2	2.0	2.0	0.1
WP3	0.0	2.0	0.1

Task 3: PID Parameter Tuning

Systematically tune gains using methods such as:

- **Ziegler-Nichols Method**
- **Manual Tuning** (start with $K_i = K_d = 0$)
- **Cohen-Coon Method**

Task 4: Safety & Constraints

Implement velocity limiting:

```
# Velocity constraints
MAX_LINEAR_VEL = 0.22  # m/s
MAX_ANGULAR_VEL = 2.84  # rad/s

linear_cmd = max(-MAX_LINEAR_VEL, min(MAX_LINEAR_VEL, pid_linear_output))
angular_cmd = max(-MAX_ANGULAR_VEL, min(MAX_ANGULAR_VEL, pid_angular_output))
```

6. Performance Metrics

Quantitative Targets

Metric	Linear Controller	Angular Controller	Target
Rise Time	Time to reach 90% of final position	Time to reach 90% of final heading	< 3.0 s
Overshoot	Maximum position overshoot	Maximum heading overshoot	< 10%
Settling Time	Time to stay within ±5% of target	Time to stay within ±5° of target	< 5.0 s
Steady-State Error	Final position error	Final heading error	< 5 cm / < 3°

Data Collection Requirements

- Record timestamps, positions, velocities, and errors at 10 Hz minimum using a ROS bag.
- Calculate metrics for each waypoint transition

- Generate performance plots showing:
    - Position vs. time
    - Velocity commands vs. time
    - Error signals vs. time

7. Educational Resources

Recommended Video Tutorials

- **PID Control Theory**: "Understanding PID Control" by MATLAB
- **ROS 2 Navigation**: "ROS 2 Navigation Stack Tutorial" by The Construct
- **TurtleBot3 Setup**: "TurtleBot3 Quick Start Guide" by ROBOTIS
- **Control System Design**: "Classical Control Theory" by Steve Brunton

8. Submission Requirements

Repository Structure

```
turtlebot3_pid_challenge/
├── src/
│   ├── pid_controller.py          # Main controller node
│   ├── waypoint_manager.py        # Waypoint handling
│   └── performance_analyzer.py    # Metrics calculation
├── launch/
│   └── pid_challenge.launch.py    # Launch configuration
├── config/
│   └── pid_params.yaml            # PID parameters
├── data/
│   ├── waypoint_logs.csv          # Performance data
│   └── tuning_results.json        # Parameter sweep results
├── plots/
│   ├── position_tracking.png      # Trajectory plots
│   └── performance_metrics.png    # Error analysis
├── README.md                      # Setup and usage instructions
└── TUNING_REPORT.md              # 1-2 page analysis
```

Tuning Report Requirements

Length: 1-2 pages (excluding plots)

Required Sections:

1. **Methodology**: Tuning approach and parameter selection rationale
2. **Results Table**: Final K_p, K_i, K_d values for both controllers
3. **Performance Analysis**: Quantitative metrics for each waypoint
4. **Visualization**: Minimum 2 plots showing trajectory and error evolution
5. **Discussion**: Challenges faced and potential improvements

Demo Video Specifications

- **Duration**: 2-3 minutes maximum
- **Content**: Real-time Gazebo simulation showing waypoint navigation
- **Overlay**: Display current waypoint, position error, and PID outputs
- **Quality**: 720p minimum resolution
- **Format**: MP4 or similar web-compatible format

9. Evaluation Rubric

Point Distribution (Total: 100 Points)

Category	Criteria	Points	Details
Code Correctness	Functional PID implementation	25	Node structure, PID math, ROS 2 integration
	Waypoint navigation logic	10	Sequential navigation, target detection
Performance Metrics	Meeting quantitative targets	20	Rise time, overshoot, settling time, steady-state error

	Data logging and analysis	10	Comprehensive data collection and processing
Robustness	Parameter sensitivity analysis	15	Testing with different gain values
	Edge case handling	5	Velocity limits, error bounds, safety checks
Documentation	Code comments and structure	5	Clear, maintainable code with proper documentation
	Tuning report quality	10	Technical depth, clarity, and insight

Grading Scale

- **90-100 Points**: Exceptional - Exceeds all targets with robust implementation
- **80-89 Points**: Proficient - Meets most targets with solid implementation
- **70-79 Points**: Developing - Meets basic requirements with minor issues
- **60-69 Points**: Beginning - Partial implementation with significant gaps
- **Below 60**: Incomplete - Major functionality missing

Bonus Opportunities (+5 points each)

- **Advanced Features**: Implement adaptive PID gains or feed-forward control
- **Visualization**: Real-time plotting of control signals during navigation
- **Comparison Study**: Analyze performance differences between tuning methods
- **Simulation Variations**: Test controller in multiple Gazebo worlds

Challenge Timeline: 2 weeks from announcement to submission deadline

Good luck, and remember: the best controllers are not just mathematically correct, but also robust, well-tuned, and thoroughly tested!