Trajectory Control System for Differential Drive Robots

A comprehensive ROS2-based trajectory tracking system implementing path smoothing, trajectory generation, and Pure Pursuit control for differential drive robots (TurtleBot3).

Table of Contents

Overview

This project implements a complete trajectory tracking solution for differential drive robots, addressing three key challenges in mobile robotics:

Core Features

- **Path Smoothing**: Catmull-Rom spline interpolation for C1 continuous smooth paths
- **Trajectory Generation**: Time-parameterized trajectory with configurable sampling rates
- **Trajectory Tracking**: Pure Pursuit controller with adaptive velocity control
- **Visualization**: Real-time path and robot state visualization in RViz2
- **Modular Architecture**: Clean separation of concerns with well-defined interfaces

Performance Characteristics

- Control frequency: 20 Hz (configurable)
- Position accuracy: < 10 cm goal tolerance
- Smooth velocity profiles with adaptive speed reduction on sharp turns

- Support for multiple trajectory patterns (figure-8, spiral, race track, etc.)

System Architecture

Component Diagram

```
┌─────────────────────────────────────────────────────────────┐
│              ROS2 System Architecture                      │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ┌──────────────┐      ┌──────────────────────────────┐    │
│  │  Waypoint    │─waypts─▶│  Trajectory Controller     │ │
│  │  Publisher   │    │                       │ │
│  └──────────────┘      │  ┌────────────────────────┐  │ │
│                 │ │  Path Smoother        │ │ │
│                 │ │  (Catmull-Rom Spline) │ │ │
│  ┌──────────────┐      │  └────────────────────────┘  │ │
│  │  Gazebo      │    │                       │ │
│  │  Simulation  │─odom──▶│  ┌────────────────────────┐  │ │
│  │      │◀─cmd_vel─│  │  Pure Pursuit      │ │ │
│  └──────────────┘      │ │  Controller      │ │ │
│                 │ └────────────────────────┘ │ │
│  ┌──────────────┐      └──────────────────────────────┘  │
│  │  RViz2    │◀─smooth_path──────────┘        │
│  │ Visualization│                    │
│  └──────────────┘                       │
└─────────────────────────────────────────────────────────────┘
```

Software Architecture

1. PathSmoother Class

- **Purpose**: Converts discrete waypoints into smooth, continuous trajectories
- **Algorithm**: Catmull-Rom spline interpolation
- **Key Features**:
  - C1 continuity (smooth velocity transitions)
  - Configurable sampling density
  - Handles edge cases (2 waypoints → linear interpolation)

2. PurePursuitController Class

- **Purpose**: Generates velocity commands for trajectory following
- **Algorithm**: Pure Pursuit with adaptive velocity
- **Key Features**:
  - Dynamic lookahead distance
  - Velocity limiting for safety
  - Adaptive speed reduction on curves
  - Goal tolerance checking

3. TrajectoryControllerNode Class

- **Purpose**: ROS2 node orchestrating the complete system
- **Responsibilities**:
  - Message handling (waypoints, odometry)
  - Timer-based control loop
  - Path visualization publishing
  - Parameter management

Data Flow

```
Waypoints (PoseArray)
    ↓
PathSmoother::smooth_path()
    ↓
Smooth Path (vector<Point2D>)
    ↓
PurePursuitController::compute_control()
    ↓
Velocity Commands (Twist)
    ↓
Robot Motion
```

Setup and Installation

Prerequisites

- **OS**: Ubuntu 22.04 LTS
- **ROS2**: Humble Hawksbill

- **Gazebo**: Classic 11 or Ignition
- **TurtleBot3**: Package installed

Dependencies

bash

```bash
sudo apt update
sudo apt install -y \
  ros-humble-turtlebot3* \
  ros-humble-gazebo-ros-pkgs \
  ros-humble-navigation2 \
  ros-humble-tf2-geometry-msgs
```

Package Installation

1. Create Workspace

bash

```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

2. Clone Repository

bash

```bash
git clone <your-repository-url> trajectory_control
cd ~/ros2_ws
```

3. Build Package

bash

```bash
colcon build --packages-select trajectory_control
source install/setup.bash
```

Package Structure

```
trajectory_control/
├── CMakeLists.txt
├── package.xml
├── README.md
```

```
├── config/
│   ├── controller_params.yaml     # Controller parameters
│   └── trajectory_control.rviz    # RViz configuration
├── launch/
│   └── trajectory_control.launch.py
├── src/
│   └── trajectory_controller_node.cpp
├── scripts/
│   └── waypoint_publisher.py
└── test/
    ├── test_path_smoother.cpp
    ├── test_pure_pursuit.cpp
    └── test_integration.cpp
```

Environment Setup

```bash
# Add to ~/.bashrc
export TURTLEBOT3_MODEL=burger
source /opt/ros/humble/setup.bash
source ~/ros2_ws/install/setup.bash
```

Execution Instructions

Quick Start (All-in-One Launch)

```bash
# Launch complete system with simulation
ros2 launch trajectory_control trajectory_control.launch.py
```

This single command starts:

- Gazebo simulation with TurtleBot3
- Trajectory controller node
- RViz2 visualization
- Waypoint publisher (auto-publishes after 2s delay)

Step-by-Step Launch (For Debugging)

Terminal 1: Gazebo Simulation

```bash
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_gazebo empty_world.launch.py
```

Terminal 2: Trajectory Controller

```bash
ros2 run trajectory_control trajectory_controller_node \
  --ros-args \
  -p lookahead_distance:=0.5 \
  -p max_linear_velocity:=0.5 \
  -p max_angular_velocity:=2.0
```

Terminal 3: RViz Visualization

```bash
ros2 run rviz2 rviz2 -d $(ros2 pkg prefix trajectory_control)/share/trajectory_control/config/trajectory_control.rviz
```

Terminal 4: Waypoint Publisher

```bash
# Default pattern (extended_figure8)
ros2 run trajectory_control waypoint_publisher.py

# Specific pattern
ros2 run trajectory_control waypoint_publisher.py \
  --ros-args -p pattern:=spiral

# Manual publishing (no auto-publish)
ros2 run trajectory_control waypoint_publisher.py \
  --ros-args -p auto_publish:=false
```

Available Trajectory Patterns

Pattern	Description	Waypoints	Length
`extended_figure8`	Large figure-8 pattern	25	~15m
`large_loop`	Circular path	32	~19m
`spiral`	Expanding spiral	48	~12m
`zigzag`	Zigzag across space	9	~15m
`race_track`	Straights and curves	30	~18m
`exploration`	Area coverage pattern	17	~22m
`circle`	Simple circle	24	~16m
`square`	Simple square	5	~12m

Runtime Parameter Tuning

```bash
# Adjust lookahead distance (affects smoothness)
ros2 param set /trajectory_controller lookahead_distance 0.7

# Adjust maximum velocities
ros2 param set /trajectory_controller max_linear_velocity 0.3
ros2 param set /trajectory_controller max_angular_velocity 1.5
```

Monitoring System

Check Topics

```bash
ros2 topic list
ros2 topic echo /smooth_path
ros2 topic hz /cmd_vel
```

Check Node Status

```bash
ros2 node info /trajectory_controller
```

View Logs

```bash
ros2 run rqt_console rqt_console
```

Design Choices and Algorithms

1. Path Smoothing: Catmull-Rom Splines

Why Catmull-Rom?

- **C1 Continuity**: Ensures smooth velocity transitions (no jerky movements)
- **Local Control**: Changing one waypoint doesn't affect entire path
- **Interpolating**: Path passes through all waypoints (predictable behavior)
- **Simple Implementation**: No complex matrix inversions needed

Algorithm Details

```cpp
Point = 0.5 * [
  2*P1 +
  (-P0 + P2)*t +
  (2*P0 - 5*P1 + 4*P2 - P3)*t² +
  (-P0 + 3*P1 - 3*P2 + P3)*t³
]
```

Parameters:

- `P0, P1, P2, P3`: Four control points
- `t`: Interpolation parameter [0, 1]
- `points_per_segment`: Sampling density (default: 20)

Alternative Considered: Cubic B-Splines

- **Not chosen** because path doesn't pass through waypoints
- Would require additional constraints for waypoint interpolation

2. Trajectory Tracking: Pure Pursuit Controller

Why Pure Pursuit?

- **Geometric Intuition**: Easy to understand and tune
- **Proven Performance**: Widely used in autonomous vehicles
- **Smooth Trajectories**: Natural arc-following behavior
- **Real-time Capable**: Low computational cost

Algorithm Details

```
1. Find closest point on path to robot
2. Search forward for lookahead point at distance L
3. Compute angle to lookahead point: α
4. Calculate curvature: κ = 2*sin(α) / L
5. Compute angular velocity: ω = κ * v
6. Apply velocity limits and adaptive speed control
```

Key Parameters:

- `lookahead_distance` (0.5m): Affects path following smoothness
  - Larger → smoother but less accurate
  - Smaller → more accurate but oscillatory
- `max_linear_velocity` (0.5 m/s): Safety and smoothness
- `max_angular_velocity` (2.0 rad/s): Turn rate limit
- `goal_tolerance` (0.1m): Success criteria

Adaptive Velocity Control

```cpp
if (|angular_velocity| > 0.5 rad/s) {
    reduction_factor = 1.0 - (|ω| / ω_max) * 0.5
    linear_velocity *= max(reduction_factor, 0.3)
}
```

This reduces speed on sharp turns to maintain stability.

Alternatives Considered

- **Stanley Controller**: More complex, requires precise heading tracking
- **MPC (Model Predictive Control)**: Computational overhead too high for 20Hz
- **PID Controller**: Requires separate lateral and longitudinal control

3. Software Architecture Decisions

Object-Oriented Design

- **Rationale**: Separation of concerns, testability, reusability
- **Benefits**:
  - PathSmoother can be unit tested independently
  - Controller can be swapped without changing node
  - Clear interfaces between components

ROS2 Node Structure

- **Single Responsibility**: Each node does one thing well
- **Publisher-Subscriber Pattern**: Loose coupling between components
- **Timer-Based Control**: Deterministic control loop execution

Parameter Management

- **Declarative Parameters**: All tunable values exposed as ROS2 parameters
- **Runtime Reconfiguration**: Can adjust behavior without restart
- **Validation**: Parameters checked for valid ranges

Real Robot Deployment

Hardware Requirements

- **Robot**: Any differential drive robot (TurtleBot3, custom platform)
- **Sensors**: Wheel encoders + IMU (for odometry)
- **Optional**: Lidar (for obstacle avoidance extension)
- **Computer**: Raspberry Pi 4 or equivalent (ARM/x86)

Deployment Steps

1. Hardware Setup

```bash
```

```
# On robot computer
export ROS_DOMAIN_ID=30  # Match with base station
export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp  # Better WiFi performance
```

2. Modify Launch File

```python
# Change from Gazebo to real robot
turtlebot3_bringup = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([
        PathJoinSubstitution([
            FindPackageShare('turtlebot3_bringup'),
            'launch',
            'robot.launch.py'
        ])
    ])
)
```

3. Sensor Calibration

- **Wheel Odometry**: Calibrate wheel radius and baseline
- **IMU**: Perform magnetometer calibration if using compass
- **Validation**: Drive known distance, verify odometry accuracy

4. Parameter Tuning for Real Robot

```yaml
# Real robot typically needs:
lookahead_distance: 0.3  # Smaller for tighter spaces
max_linear_velocity: 0.2  # Conservative for safety
max_angular_velocity: 1.0  # Prevent wheel slippage
control_frequency: 20.0  # Match sensor update rate
```

5. Safety Considerations

- **Emergency Stop**: Implement deadman switch or timeout
- **Collision Detection**: Monitor bumper sensors
- **Battery Monitoring**: Stop if battery low
- **Velocity Ramping**: Gradual acceleration/deceleration

Real-World Challenges and Solutions

Challenge 1: Odometry Drift

- **Problem**: Wheel slip causes position error over time
- **Solution**:
  - Fuse with visual odometry (e.g., ORB-SLAM)
  - Use external localization (AprilTags, AMCL with map)
  - Periodic re-localization at known landmarks

Challenge 2: Sensor Noise

- **Problem**: Real sensors have noise unlike simulation
- **Solution**:
  - Implement Kalman filter for state estimation
  - Low-pass filter velocity commands
  - Increase lookahead distance for smoother response

Challenge 3: Latency

- **Problem**: WiFi delays affect control performance
- **Solution**:
  - Run controller on robot (not remote PC)
  - Use wired connection for critical data
  - Implement predictive control (anticipate delays)

Challenge 4: Unmodeled Dynamics

- **Problem**: Real robot has mass, friction, motor dynamics
- **Solution**:
  - System identification to measure parameters
  - Add feedforward compensation
  - Use acceleration limits in trajectory generation

Testing Protocol for Real Robot

```bash
```

```
# 1. Static tests
ros2 topic pub /waypoints geometry_msgs/msg/PoseArray "{poses: [{position: {x: 0.5, y: 0.0}}]}" --once

# 2. Short trajectory
ros2 run trajectory_control waypoint_publisher.py --ros-args -p pattern:=square

# 3. Progressive complexity
# square → circle → figure8 → race_track

# 4. Performance metrics
ros2 bag record /odom /cmd_vel /smooth_path  # Record for analysis
```

AI Tools Used

Development Workflow

1. Claude (Anthropic) - Primary Assistant

- **Usage**: Architecture design, algorithm explanation, documentation
- **Benefits**:
  - Helped design modular architecture
  - Explained trade-offs between control algorithms
  - Generated comprehensive documentation templates
  - Debugged complex C++ template issues

2. GitHub Copilot

- **Usage**: Code completion, boilerplate generation
- **Benefits**:
  - Accelerated ROS2 publisher/subscriber setup
  - Auto-completed repetitive parameter declarations
  - Suggested error handling patterns

3. ChatGPT-4

- **Usage**: Mathematical derivations, algorithm research
- **Benefits**:
  - Derived Catmull-Rom spline equations

- Explained Pure Pursuit mathematics
- Suggested test cases and edge conditions

Best Practices with AI Tools

Effective Prompting

```
Good Prompt:
"Implement a Pure Pursuit controller for a differential drive robot.
Requirements: lookahead distance 0.5m, max velocity 0.5 m/s, handle
velocity limits, adaptive speed on curves. Use ROS2 Humble."

Poor Prompt:
"Write code for robot control"
```

Code Review with AI

- Always validate AI-generated mathematical formulas
- Test edge cases AI might not consider
- Verify ROS2 API usage against documentation
- Check for memory leaks in C++ code

Iterative Refinement

1. AI generates initial implementation
2. Human tests and identifies issues
3. AI suggests fixes with context
4. Human validates and integrates
5. Repeat until quality standards met

Obstacle Avoidance Extension

Architecture Overview

To extend this system with obstacle avoidance, we would implement a **Dynamic Window Approach (DWA)** planner integrated with the existing Pure Pursuit controller.

Proposed Architecture

```
Waypoints → Path Smoother → Global Path
                   ↓
           ┌─────────────────────────┐
Lidar Scan →     │ Local Planner (DWA) │
           │ - Obstacle detection │
           │ - Dynamic avoidance  │
           │ - Local path        │
           └─────────────────────────┘
                   ↓
           Pure Pursuit Controller
                   ↓
             Velocity Commands
```

Implementation Details

1. Obstacle Detection Module

```cpp
class ObstacleDetector {
    std::vector<Point2D> detect_obstacles(
        const sensor_msgs::msg::LaserScan& scan,
        double safety_distance = 0.5) {

        std::vector<Point2D> obstacles;

        for (size_t i = 0; i < scan.ranges.size(); ++i) {
            if (scan.ranges[i] < safety_distance) {
                double angle = scan.angle_min + i * scan.angle_increment;
                double x = scan.ranges[i] * cos(angle);
                double y = scan.ranges[i] * sin(angle);
                obstacles.push_back(Point2D(x, y));
            }
        }

        return obstacles;
    }
};
```

2. Dynamic Window Approach (DWA)

cpp

```cpp
class DWAPlanner {
public:
    struct VelocityCommand {
        double v;  // Linear velocity
        double w;  // Angular velocity
        double cost;
    };

    VelocityCommand plan(
        const Pose2D& pose,
        const Velocity& current_vel,
        const std::vector<Point2D>& obstacles,
        const Point2D& target) {

        // 1. Generate velocity samples in dynamic window
        auto velocity_samples = generate_dynamic_window(current_vel);

        // 2. Simulate trajectory for each velocity
        std::vector<VelocityCommand> candidates;
        for (auto& vel : velocity_samples) {
            auto trajectory = simulate_trajectory(pose, vel);

            // 3. Evaluate trajectory
            double heading_cost = compute_heading_cost(trajectory, target);
            double clearance_cost = compute_clearance_cost(trajectory, obstacles);
            double velocity_cost = compute_velocity_cost(vel);

            double total_cost =
                w_heading_ * heading_cost +
                w_clearance_ * clearance_cost +
                w_velocity_ * velocity_cost;

            // 4. Check collision
            if (!check_collision(trajectory, obstacles)) {
                candidates.push_back({vel.v, vel.w, total_cost});
            }
        }

        // 5. Select best velocity
        return select_best_velocity(candidates);
    }
};
```

3. Integration with Pure Pursuit

```cpp
class HybridController {
    // High-level: Pure Pursuit for global path
    // Low-level: DWA for obstacle avoidance

    void compute_control(...) {
        // Get global reference from Pure Pursuit
        auto global_target = pure_pursuit_->get_lookahead_point();

        // Check for obstacles
        auto obstacles = obstacle_detector_->detect_obstacles(scan);

        if (!obstacles.empty()) {
            // Use DWA for local avoidance
            auto cmd = dwa_planner_->plan(pose, vel, obstacles, global_target);
            linear_vel = cmd.v;
            angular_vel = cmd.w;
        } else {
            // Use Pure Pursuit when clear
            pure_pursuit_->compute_control(...);
        }
    }
};
```

Cost Functions

Heading Cost

```cpp
```

```cpp
double compute_heading_cost(
    const std::vector<Point2D>& trajectory,
    const Point2D& target) {

    auto endpoint = trajectory.back();
    double angle_to_target = atan2(
        target.y - endpoint.y,
        target.x - endpoint.x
    );
    return abs(normalize_angle(angle_to_target));
}
```

Clearance Cost

cpp

```cpp
double compute_clearance_cost(
    const std::vector<Point2D>& trajectory,
    const std::vector<Point2D>& obstacles) {

    double min_clearance = std::numeric_limits<double>::max();

    for (auto& traj_point : trajectory) {
        for (auto& obs : obstacles) {
            double dist = traj_point.distance_to(obs);
            min_clearance = std::min(min_clearance, dist);
        }
    }

    // Inverse cost: prefer larger clearance
    return 1.0 / (min_clearance + 0.1);
}
```

ROS2 Integration

Additional Topics

cpp

```
// Subscribe to Lidar
lidar_sub_ = create_subscription<sensor_msgs::msg::LaserScan>(
    "scan", 10, &callback);

// Publish obstacle markers
obstacle_pub_ = create_publisher<visualization_msgs::msg::MarkerArray>(
    "obstacles", 10);

// Publish local trajectory
local_path_pub_ = create_publisher<nav_msgs::msg::Path>(
    "local_path", 10);
```

Parameter Configuration

```yaml
obstacle_avoidance:
  enabled: true
  safety_distance: 0.5  # meters
  dwa:
    velocity_samples: 20
    angular_samples: 20
    sim_time: 2.0  # seconds
    cost_weights:
      heading: 1.0
      clearance: 2.0
      velocity: 0.5
```

Testing Strategy

Test Scenarios

1. **Static Obstacles**: Place boxes in simulation
2. **Dynamic Obstacles**: Moving pedestrians/robots
3. **Narrow Passages**: Test minimum clearance
4. **Cluttered Environment**: Multiple scattered obstacles

Evaluation Metrics

- Success rate (reaching goal without collision)
- Path efficiency (ratio of actual/optimal path length)

- Smoothness (velocity jerk)
- Computation time (must be < 50ms for 20Hz)

Testing and Quality Assurance

Test Suite Architecture

1. Unit Tests

PathSmoother Tests (`test/test_path_smoother.cpp`)

```cpp
TEST(PathSmootherTest, TwoPointsLinearInterpolation) {
    std::vector<Point2D> waypoints = {{0, 0}, {1, 1}};
    auto result = PathSmoother::smooth_path(waypoints, 10);

    EXPECT_EQ(result.size(), 11);
    EXPECT_NEAR(result[0].x, 0.0, 1e-6);
    EXPECT_NEAR(result[10].x, 1.0, 1e-6);
}

TEST(PathSmootherTest, ContinuityCheck) {
    std::vector<Point2D> waypoints = {{0,0}, {1,0}, {1,1}, {0,1}};
    auto result = PathSmoother::smooth_path(waypoints, 20);

    // Check C1 continuity (smooth derivatives)
    for (size_t i = 1; i < result.size() - 1; ++i) {
        double dx1 = result[i].x - result[i-1].x;
        double dx2 = result[i+1].x - result[i].x;
        double derivative_change = abs(dx2 - dx1);
        EXPECT_LT(derivative_change, 0.1);  // Smooth change
    }
}
```

Pure Pursuit Tests (`test/test_pure_pursuit.cpp`)

```cpp
```

```
TEST(PurePursuitTest, GoalReached) {
    PurePursuitController controller(0.5, 0.5, 2.0, 0.1);
    Pose2D pose(0.95, 0.0, 0.0);  // Near goal
    std::vector<Point2D> path = {{1.0, 0.0}};

    double v, w;
    bool reached = controller.compute_control(pose, path, v, w);

    EXPECT_TRUE(reached);
    EXPECT_NEAR(v, 0.0, 1e-6);
    EXPECT_NEAR(w, 0.0, 1e-6);
}

TEST(PurePursuitTest, VelocityLimits) {
    PurePursuitController controller(0.5, 0.3, 1.5, 0.1);
    Pose2D pose(0.0, 0.0, 0.0);
    std::vector<Point2D> path = {{1.0, 1.0}};  // 45° turn

    double v, w;
    controller.compute_control(pose, path, v, w);

    EXPECT_LE(abs(v), 0.3);  // Linear limit
    EXPECT_LE(abs(w), 1.5);  // Angular limit
}
```

2. Integration Tests

System Integration Test (`test/test_integration.cpp`)

```cpp
```

```cpp
TEST_F(IntegrationTest, CompleteTrajectoryTracking) {
  // Setup
  auto node = std::make_shared<TrajectoryControllerNode>();
  std::vector<Point2D> waypoints = {{0,0}, {1,0}, {1,1}, {0,1}};

  // Publish waypoints
  publish_waypoints(waypoints);

  // Simulate robot motion
  for (int i = 0; i < 500; ++i) {  // 25 seconds at 20Hz
    rclcpp::spin_some(node);
    update_simulated_robot();
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }

  // Verify goal reached
  auto final_pose = get_robot_pose();
  EXPECT_NEAR(final_pose.x, 0.0, 0.15);
  EXPECT_NEAR(final_pose.y, 1.0, 0.15);
}
```

3. Test Automation

CMakeLists.txt Integration

```cmake
cmake
```

```
if(BUILD_TESTING)
  find_package(ament_cmake_gtest REQUIRED)

  # Unit tests
  ament_add_gtest(test_path_smoother test/test_path_smoother.cpp)
  target_link_libraries(test_path_smoother ${PROJECT_NAME})

  ament_add_gtest(test_pure_pursuit test/test_pure_pursuit.cpp)
  target_link_libraries(test_pure_pursuit ${PROJECT_NAME})

  # Integration tests
  ament_add_gtest(test_integration test/test_integration.cpp)
  target_link_libraries(test_integration ${PROJECT_NAME})

  # Linting
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
endif()
```

Run Tests

```bash
# Build with tests
colcon build --packages-select trajectory_control

# Run all tests
colcon test --packages-select trajectory_control

# View results
colcon test-result --verbose
```

Error Handling

Input Validation

```cpp
```

```cpp
void waypoint_callback(const geometry_msgs::msg::PoseArray::SharedPtr msg) {
    // Check for empty input
    if (msg->poses.empty()) {
        RCLCPP_WARN(get_logger(), "Received empty waypoint array");
        return;
    }

    // Validate waypoint values
    for (const auto& pose : msg->poses) {
        if (std::isnan(pose.position.x) || std::isnan(pose.position.y)) {
            RCLCPP_ERROR(get_logger(), "Invalid waypoint with NaN values");
            return;
        }
        if (abs(pose.position.x) > 100 || abs(pose.position.y) > 100) {
            RCLCPP_ERROR(get_logger(), "Waypoint out of reasonable bounds");
            return;
        }
    }

    // Proceed with processing
    process_waypoints(msg);
}
```

Runtime Error Recovery

```cpp
```

```cpp
void control_loop() {
    try {
        if (!odom_received_) {
            RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 5000,
                "No odometry received, waiting...");
            return;
        }

        // Control computation
        double v, w;
        bool success = controller_->compute_control(
            current_pose_, current_path_, v, w);

        if (!success && !goal_reached_) {
            // Retry or fallback behavior
            handle_control_failure();
        }

    } catch (const std::exception& e) {
        RCLCPP_ERROR(get_logger(), "Control loop exception: %s", e.what());
        publish_zero_velocity();  // Safety stop
    }
}
```

Timeout Handling

```cpp
// Watchdog timer for odometry
void check_odometry_timeout() {
    auto now = this->now();
    if ((now - last_odom_time_).seconds() > 1.0) {
        RCLCPP_ERROR(get_logger(), "Odometry timeout - stopping robot");
        publish_zero_velocity();
        odom_received_ = false;
    }
}
```

Quality Metrics

Code Coverage

```bash
```

```bash
# Enable coverage
colcon build --packages-select trajectory_control \
  --cmake-args -DCMAKE_BUILD_TYPE=Coverage

# Generate report
lcov --capture --directory buil
```