ReworkedMap Node Documentation

Path Planning and Trajectory Optimization for Robot Vision Systems

ROS 2 Dobot CR5 Project

October 4, 2025

Contents

1 Introduction

The `reworked_map` node is a critical component of the Dobot CR5 robot vision system that processes environmental maps and generates optimal camera positioning trajectories for object scanning. This node receives raw environmental data, calculates potential camera positions around target objects, and evaluates their feasibility based on workspace constraints and obstacle avoidance.

1.1 Purpose

The primary purpose of this node is to:

- Process incoming environmental maps from vision sensors
- Generate circular trajectories around target objects
- Evaluate trajectory points for reachability and collision avoidance
- Publish optimized maps with feasible camera positions

1.2 System Architecture

The node operates as a middleware component between raw sensor data (`Boing` topic) and motion planning systems (`reworked_map` topic), ensuring that only viable camera positions are considered for robot motion planning.

2 Global Configuration Parameters

2.1 Trajectory Parameters

```
                double radius = 0.3;                    // Circumference
                   radius in meters
                int num_points = 64;                     // Number of points
                    per circumference
                double field_of_vision_z = 0.2;          // Camera field of
                   view height in meters
                double tolerance = 0.05;                 // Tolerance for
                   field of view exceeding
```

Listing 1: Global Configuration Variables

2.1.1 Radius Parameter

The `radius` parameter defines the distance from the object center at which camera positions are generated. A radius of 0.3 meters ensures:

- Adequate distance for collision avoidance
- Sufficient field of view coverage
- Optimal image quality for object analysis

2.1.2 Point Density

The `num_points` parameter (64) provides sufficient angular resolution (5.625° between points) for smooth trajectory planning while maintaining computational efficiency.

2.1.3 Field of Vision

The `field_of_vision_z` parameter determines the vertical coverage area of the camera, affecting how multiple circumferences are stacked for tall objects.

3 Class Architecture

3.1 MapProcessorNode Class

```
                class MapProcessorNode : public rclcpp::Node
                {
                        public:
                        MapProcessorNode();
                        private:
                        rclcpp::Subscription<custom_messages::msg::Map
                           >::SharedPtr map_subscriber_;
                        rclcpp::Publisher<custom_messages::msg::Map>::
                           SharedPtr map_publisher_;
                        void mapCallback(const custom_messages::msg::Map
                           ::SharedPtr msg);
                };
```

Listing 2: MapProcessorNode Class Definition

3.1.1 Constructor Logic

The constructor initializes the node with specific Quality of Service (QoS) configurations:

- **Subscriber QoS**: `KeepLast(1)` with `transient_local()` durability ensures the node receives the most recent map even if it starts after the publisher
- **Publisher QoS**: Similar configuration ensures downstream nodes can receive the latest processed map regardless of timing

3.1.2 Callback Mechanism

The `mapCallback` function orchestrates the entire map processing pipeline:

1. Receives incoming map data
2. Creates a local copy for processing
3. Invokes the map modification pipeline
4. Publishes the processed result
5. Generates visualization output

4 Core Processing Functions

4.1 generateOptimalCircumferences Function

```
void generateOptimalCircumferences(custom_messages::msg::Object& plant) {
    std::vector<custom_messages::msg::Point> plant_centers = calculateCenters(plant);
    double angle_increment = 2.0 * M_PI / num_points;

    for (auto target_center : plant_centers) {
        std::vector<custom_messages::msg::OptimalPoint> circumference_size(num_points);
        custom_messages::msg::Circumference trajectory;
        trajectory.circumference = circumference_size;

        for (int j = 0; j < num_points; ++j) {
            double angle = j * angle_increment;
            trajectory.circumference[j].x = target_center.x + radius * cos(angle);
            trajectory.circumference[j].y = target_center.y + radius * sin(angle);
            trajectory.circumference[j].z = target_center.z;
            trajectory.circumference[j].optimality = 1.0;
        }
        plant.possible_trajectories.push_back(trajectory);
    }
}
```

Listing 3: Circumference Generation Logic

4.1.1 Mathematical Foundation

The function implements parametric circle generation using trigonometric functions:

$$x = x_{center} + r\cos(\theta) \tag{1}$$
$$y = y_{center} + r\sin(\theta) \tag{2}$$
$$z = z_{center} \tag{3}$$

Where θ ranges from 0 to 2π in increments of $\frac{2\pi}{n}$ for n points.

4.1.2 Multi-Level Strategy

For tall objects, multiple circumferences are generated at different heights, ensuring complete coverage of the object's vertical extent.

4.2 calculateCenters Function

```cpp
        std::vector<custom_messages::msg::Point> calculateCenters(custom_messages::msg::Object& plant) {
            custom_messages::msg::Point low_left = plant.shape.low_left;
            custom_messages::msg::Point top_right = plant.shape.top_right;

            double height = top_right.z - low_left.z;
            std::vector<custom_messages::msg::Point> centers;

            if (height <= field_of_vision_z + tolerance) {
                // Single center for small objects
                custom_messages::msg::Point center;
                center.x = top_right.x - (top_right.x - low_left.x) / 2.0;
                center.y = top_right.y - (top_right.y - low_left.y) / 2.0;
                center.z = top_right.z + (0.2) * field_of_vision_z;
                centers.push_back(center);
            } else {
                // Multiple centers for tall objects
                int num_divisions = height / field_of_vision_z;
                if (height - (num_divisions * field_of_vision_z) > tolerance) {
                    num_divisions++;
                }

                for (int i = 0; i < num_divisions; i++) {
                    custom_messages::msg::Point center;
                    center.x = top_right.x - (top_right.x - low_left.x) / 2.0;
                    center.y = top_right.y - (top_right.y - low_left.y) / 2.0;

                    if (i != 0) {
                        center.z = top_right.z - (i + 0.5) * field_of_vision_z;
                    } else {
                        center.z = top_right.z + (0.25) * field_of_vision_z;
                    }
                    centers.push_back(center);
                }
            }
            return centers;
        }
```

Listing 4: Center Calculation Logic

4.2.1 Adaptive Height Strategy

The function implements an adaptive strategy for objects of varying heights:

Small Objects ($h \leq f_{vision} + \epsilon$):

- Single circumference positioned above the object
- Center calculated at geometric centroid of bounding box
- Height offset ensures camera can capture entire object

Tall Objects ($h > f_{vision} + \epsilon$):

- Multiple circumferences stacked vertically
- Division count: $n = \lceil \frac{h}{f_{vision}} \rceil$
- Each circumference covers a specific vertical segment
- Top circumference positioned above object for complete coverage

4.2.2 Height Calculation Logic

The vertical positioning follows a top-down approach:

$$z_{center}(i) = \begin{cases} z_{top} + 0.25 \cdot f_{vision} & \text{if } i = 0 \text{ (topmost)} \\ z_{top} - (i + 0.5) \cdot f_{vision} & \text{if } i > 0 \end{cases} \tag{4}$$

4.3 checkWorkspace Function

```
                    void checkWorkspace(custom_messages::msg::Object& plant,
                        const custom_messages::msg::Map& map) {
                            custom_messages::msg::BoundingBox ws_limits =
                                map.work_space;
                            custom_messages::msg::Point low = ws_limits.
                                low_left;
                            custom_messages::msg::Point top = ws_limits.
                                top_right;

                            for (auto& trajectory : plant.
                                possible_trajectories) {
                                    for (auto& point : trajectory.
                                        circumference) {
                                            if (point.optimality != 0.0) {
                                                    if (point.x < low.x ||
                                                        point.x > top.x ||
                                                    point.y < low.y || point
                                                        .y > top.y ||
```

```cpp
                point.z < low.z || point.z > top.z) {
                    point.optimality = 0.0; // Mark as non-optimal
                }
            }
        }
    }
}
```

Listing 5: Workspace Validation Logic

4.3.1 Boundary Validation

This function implements a critical safety mechanism by validating that all generated trajectory points fall within the robot's operational workspace. The validation checks:

$$\begin{aligned} \text{Valid} = (x_{low} \leq x \leq x_{high}) \wedge & \quad (5) \\ (y_{low} \leq y \leq y_{high}) \wedge & \quad (6) \\ (z_{low} \leq z \leq z_{high}) & \quad (7) \end{aligned}$$

4.3.2 Conservative Marking

Points outside the workspace are permanently marked as non-optimal (`optimality = 0.0`), ensuring they won't be reconsidered in subsequent processing stages.

4.4 pointsInsideOtherObjects Function

```cpp
void pointsInsideOtherObjects(custom_messages::msg::Object& plant, const custom_messages::msg::Map& map) {
    for (auto& object : map.objects) {
        for (auto& trajectory : plant.possible_trajectories) {
            for (auto& point : trajectory.circumference) {
                if (point.optimality != 0.0) {
                    custom_messages::msg::Point low = object.shape.low_left;
                    custom_messages::msg::Point top = object.shape.top_right;
                    if (point.x >= low.x && point.x <= top.x &&
                                                        point.y >= low.y
                                                            && point.y
                                                           <= top.y &&
                                                        point.z >= low.z
                                                            && point.z
                                                           <= top.z) {
                                                                point.
                                                                   optimality
                                                                    =
                                                                   0.0;
                                                                   //
                                                                   Mark
                                                                   as
                                                                   non-
                                                                   optimal
                                                        }
                                                }
                                        }
                                }
                        }
                }
```

Listing 6: Collision Detection Logic

4.4.1 Collision Avoidance Strategy

This function implements a comprehensive collision detection algorithm that:

- Iterates through all objects in the environment
- Checks each trajectory point against object bounding boxes
- Marks points inside any object as non-optimal
- Includes self-collision detection (camera positions inside target object)

4.4.2 Bounding Box Intersection

The collision detection uses axis-aligned bounding box (AABB) intersection:

$$\text{Collision} = (x_{obj,low} \leq x \leq x_{obj,high}) \wedge \tag{8}$$
$$(y_{obj,low} \leq y \leq y_{obj,high}) \wedge \tag{9}$$
$$(z_{obj,low} \leq z \leq z_{obj,high}) \tag{10}$$

5 Processing Pipeline

5.1 modifyMap Function

```
                void modifyMap(custom_messages::msg::Map& map) {
                        for (auto& object : map.objects) {
                                if (object.target) {
                                        generateOptimalCircumferences(
                                           object);
```

```
                                                checkWorkspace(object, map);
                                                pointsInsideOtherObjects(object,
                                                     map);
                                      }
                            }
                  }
```

Listing 7: Map Processing Pipeline

5.1.1 Sequential Processing

The map modification follows a strict sequential order:

1. **Generation**: Create initial circumferences with all points marked optimal
2. **Workspace Validation**: Remove points outside operational boundaries
3. **Collision Detection**: Remove points intersecting with obstacles

5.1.2 Target Object Filtering

Only objects marked with `target = true` are processed, allowing selective trajectory generation for objects of interest while ignoring static obstacles.

6 Optimization Strategy

6.1 Binary Optimality Model

The current implementation uses a binary optimality model:

$$\text{optimality}(p) = \begin{cases} 1.0 & \text{if point is reachable and collision-free} \\ 0.0 & \text{if point violates constraints} \end{cases} \tag{11}$$

6.2 Constraint Hierarchy

The optimization follows a hierarchical constraint structure:

1. **Hard Constraints**: Workspace boundaries (safety-critical)
2. **Collision Constraints**: Obstacle avoidance (safety-critical)
3. **Soft Constraints**: Could include view quality, accessibility, etc.

7 Visualization and Debugging

7.1 Grid Visualization

The node includes a built-in visualization system that generates a 2D ASCII representation of the trajectory points:

```
                const int grid_size = 41;
                char grid[grid_size][grid_size];
                double scale = (double)(grid_size - 1) / (2.0 * radius);
                int center_grid = grid_size / 2;

                for (auto& circumference : optimal_points.possible_trajectories) {
                        for (const auto& point : circumference.circumference) {
                                int grid_x = static_cast<int>(round(center_grid + (point.x) * scale));
                                int grid_y = static_cast<int>(round(center_grid + (point.y) * scale));

                                if (grid_x >= 0 && grid_x < grid_size && grid_y >= 0 && grid_y < grid_size) {
                                        if (point.optimality == 1.0) {
                                                grid[grid_y][grid_x] = '1';
                                        } else {
                                                grid[grid_y][grid_x] = '0';
                                        }
                                }
                        }
                }
```

Listing 8: Visualization Logic

7.1.1 Coordinate Transformation

The visualization transforms 3D world coordinates to 2D grid coordinates using:

$$x_{grid} = \text{center} + x_{world} \cdot \text{scale} \tag{12}$$

$$y_{grid} = \text{center} + y_{world} \cdot \text{scale} \tag{13}$$

Where `scale` ensures the circumference fits within the grid boundaries.

8 Performance Considerations

8.1 Computational Complexity

The algorithm's computational complexity is:

$$O(n \cdot m \cdot p \cdot o) \tag{14}$$

Where:

- n = number of target objects
- m = number of circumferences per object
- p = number of points per circumference (64)
- o = number of objects for collision checking

8.2 Memory Usage

Memory requirements scale with:

- Object count in the environment
- Circumference density (points per circle)
- Number of vertical levels for tall objects

8.3 Optimization Opportunities

Potential optimizations include:

- Spatial indexing for collision detection
- Early termination for obviously invalid points
- Parallel processing for independent circumferences
- Adaptive point density based on object size

9 Future Enhancements

9.1 Dynamic Radius Calculation

Current implementation uses a fixed radius. Future versions could implement adaptive radius based on object dimensions:

```
                    double calculateDynamicRadius(const custom_messages::msg::Object& plant) {
                        double width = plant.shape.top_right.x - plant.shape.low_left.x;
                        double depth = plant.shape.top_right.y - plant.shape.low_left.y;
                        double max_dimension = std::max(width, depth);
                        return std::max(max_dimension / 2.0 + 0.2, 0.3); // Safety margin + minimum
                    }
```

Listing 9: Proposed Dynamic Radius Calculation

9.2 Advanced Optimality Metrics

Beyond binary classification, future versions could implement:

- View quality assessment
- Lighting condition evaluation
- Camera angle optimization
- Multi-objective optimization

9.3 Path Smoothing

Integration of trajectory smoothing algorithms to ensure feasible robot motion between optimal points.

10 Integration with Robot Control

10.1 Interface Specifications

The node interfaces with the broader robot control system through:

- **Input**: `Boing` topic (raw environmental map)
- **Output**: `reworked_map` topic (processed map with trajectories)

10.2 QoS Configuration

The `transient_local` durability ensures:

- Late-joining subscribers receive the latest map
- System resilience to temporary node failures
- Consistent data availability across restarts

11 Error Handling and Robustness

11.1 Input Validation

The node should validate:

- Map structure completeness
- Object bounding box validity
- Workspace boundary consistency

11.2 Graceful Degradation

In case of errors, the system should:

- Log detailed error information
- Publish partial results when possible
- Maintain system stability

12 Conclusion

The `reworked_map` node represents a sophisticated trajectory planning system that bridges environmental perception and robot motion planning. Its modular design, comprehensive constraint handling, and optimization strategies make it a critical component for autonomous robot vision systems.

The node's ability to generate collision-free, workspace-compliant camera trajectories enables safe and effective object scanning operations while maintaining computational efficiency suitable for real-time robotic applications.