CR5 Modular Motion Controller

Technical Documentation

Robotics Engineering Team

October 4, 2025

Contents

1 Overview

This document provides comprehensive documentation for the modular CR5 robotic motion control system. The system is built on a clean, maintainable modular design following C++ best practices.

System Highlights

- **Modular Architecture:** Clean separation of concerns across specialized modules
- **ROS2 Integration:** Full compatibility with ROS2 Humble and MoveIt2
- **Safety First:** Comprehensive collision avoidance and safety mechanisms
- **Professional Code:** Following modern C++ and robotics best practices

2 System Architecture

2.1 Directory Structure

```
cr5_moveit_cpp_demo/
|-- src/
|   |-- nodes/              # Main executable entry points
|   |-- modules/            # Core modular components
|   \-- external/           # External utilities and supporting
    nodes
|-- include/modules/        # Header files for modular
    components
|-- launch/                 # ROS2 launch files
|-- CMakeLists.txt          # Build configuration
\-- package.xml             # Package metadata
```

Listing 1: Project Directory Structure

2.2 Architectural Principles

1. **Single Responsibility:** Each module handles one specific aspect of the system
2. **Dependency Injection:** Components receive dependencies through constructors
3. **Interface Segregation:** Clean, focused public interfaces
4. **Composition over Inheritance:** Flexible system composition

3 Folder Documentation

3.1 /src/nodes/ - Main Entry Points

Contains the main executable files that serve as entry points to the system.

File	Lines	Purpose
`cr5_planner_node.cpp`	75	Main application entry point for system initialization and component orchestration

Purpose: Provides system initialization, configuration, and orchestration of modular components.

3.2 /src/modules/ - Core Modular Components

Contains the main business logic organized into specialized modules.

File	Purpose
`movement_controller.cpp`	Motion planning and execution with collision avoidance
`plant_scanner.cpp`	Scanning logic, point analysis, and coverage optimization
`scene_manager.cpp`	3D obstacle management and collision object handling
`plant_processor.cpp`	Map processing, plant coordination, and system orchestration

3.3 /src/external/ - External Utilities

Contains supporting utilities and external node implementations.

File	Purpose
`fake_zed.cpp`	ZED camera simulation for testing and development
`reworked_map.cpp`	Map processing and data transformation utility
`reconnaissance.cpp`	Advanced reconnaissance and exploration functionality
`follow_target.cpp`	Dynamic target following capabilities

3.4 /include/modules/ - Header Files

Contains interface definitions and declarations for modular components.

File	Purpose
`movement_controller.hpp`	Movement controller interface and data structures

`plant_scanner.hpp`	Plant scanner interface and scanning algorithms
`scene_manager.hpp`	Scene management interface and collision objects
`plant_processor.hpp`	Plant processor interface and coordination logic

4 Module Documentation

4.1 Namespace: cr5_demo

All custom classes are organized under the `cr5_demo` namespace to:

- Prevent naming conflicts with external libraries
- Organize code logically
- Follow C++ best practices
- Maintain clean separation between custom and external code

5 Core Modules

5.1 MovementController

Files: `movement_controller.hpp/.cpp`

Purpose: Handles all robot motion planning and execution operations with comprehensive safety mechanisms.

5.1.1 Key Data Structures

```
1    struct ScanPoint {
2      geometry_msgs::msg::Pose pose;       // 3D position and
          orientation
3      bool is_boundary;                    // Indicates boundary
          point
4      bool covered;                        // Coverage status
5      double optimality;                   // Quality metric
          (0.0-1.0)
6    };
```

Listing 2: ScanPoint Structure

5.1.2 Public Interface

Function	Description
`MovementController(node)`	Constructor - initializes movement controller with ROS2 node
`moveToTargetSafely()`	Safely moves robot to target position with collision avoidance
`attemptToReachPoint()`	Attempts to reach scan point and updates coverage status
`planMovement()`	Plans robot movement using MoveIt with configured constraints
`executeMovement()`	Executes previously planned movement with safety monitoring

5.1.3 Function Details

moveToTargetSafely()

```
bool moveToTargetSafely(
MoveGroupInterface& move_group,
const geometry_msgs::msg::Point& target,
const std::string& description
)
```

Listing 3: Function Signature

- **Purpose:** Safely moves robot to target position with collision avoidance
- **Parameters:**
  - `move_group`: MoveIt interface for robot control
  - `target`: 3D target position coordinates
  - `description`: Human-readable description for logging
- **Returns:** `true` if movement successful, `false` otherwise
- **Logic:** Validates target, plans collision-free path, executes with safety checks

attemptToReachPoint()

```
bool attemptToReachPoint(
ScanPoint& scan_point,
MoveGroupInterface& move_group,
const std::string& point_description
)
```

Listing 4: Function Signature

- **Purpose:** Attempts to reach specific scan point and updates coverage status
- **Parameters:**
  - `scan_point`: Reference to scan point (modified in-place)

  - `move_group`: MoveIt interface for robot control
  - `point_description`: Description for comprehensive logging

- **Returns:** `true` if point reached successfully
- **Logic:** Plans movement, executes if feasible, marks point as covered

5.2 PlantScanner

Files: `plant_scanner.hpp/.cpp`

Purpose: Analyzes plant circumferences and manages scanning operations for optimal coverage using advanced algorithms.

5.2.1 Public Interface

Function	Description
`PlantScanner(node)`	Constructor - initializes scanner with internal movement controller
`processCircumference()`	Main scanning pipeline for plant circumference analysis
`identifyBoundaryPoints()`	Identifies critical boundary points using spatial analysis
`scanUncoveredPoints()`	Scans remaining uncovered optimal points efficiently
`createPosePointingTowardsPlant()`	Creates robot pose oriented towards plant center
`convertToScanPoints()`	Converts optimal points to internal ScanPoint structure

5.2.2 Algorithm Details

Two-Phase Scanning Strategy

1. **Phase 1 - Boundary Points:**
   - Identify critical boundary points using spatial analysis
   - Prioritize points at coverage boundaries
   - Attempt to visit all boundary points for maximum coverage
   - Track success statistics: "Visited X/Y boundary points"
2. **Phase 2 - Remaining Coverage:**
   - Identify uncovered optimal points (quality = 1.0)
   - Attempt to visit remaining high-quality points
   - Continue until coverage goals met or no reachable points remain
   - Generate comprehensive coverage reports

5.2.3 Configuration Constants

```
1    static constexpr double CAMERA_FOV_HORIZONTAL = 0.15; //
        Camera field of view
```

Listing 5: Camera Configuration

5.3 SceneManager

Files: `scene_manager.hpp/.cpp`

Purpose: Manages 3D scene obstacles and collision objects for safe robot operation.

5.3.1 Key Data Structures

```
1    struct PlantWithCenter {
2      moveit_msgs::msg::CollisionObject plant_shape;  // 3D
          collision geometry
3      geometry_msgs::msg::Point center;               //
          Calculated center point
4    };
```

Listing 6: PlantWithCenter Structure

5.3.2 Public Interface

Function	Description
`SceneManager(node)`	Constructor - initializes scene manager with logging
`addObstaclesToScene()`	Adds all map obstacles to MoveIt planning scene
`setupFloorObstacle()`	Creates safety floor obstacle for collision prevention
`clearAllObstacles()`	Removes all collision objects from planning scene

5.3.3 Scene Management Pipeline

1. **Obstacle Processing:**
   - Clear existing obstacles from planning scene
   - Process each map object for collision geometry
   - Create collision objects for non-target objects
   - Generate plant shapes with calculated centers
2. **Safety Setup:**
   - Create large horizontal floor plane as collision object

- Position at ground level to prevent dangerous movements
- Apply all changes to MoveIt planning scene

5.4 PlantProcessor

Files: `plant_processor.hpp/.cpp`

Purpose: Main coordination module that processes maps and orchestrates plant investigation with comprehensive logging.

5.4.1 Public Interface

Function	Description
`PlantProcessor(node)`	Constructor - initializes with dependent modules
`mapCallback()`	Main entry point for processing incoming map messages
`processTargetPlants()`	Processes all target plants with detailed coordination
`calculateScanCenter()`	Calculates optimal scan center for plant investigation

5.4.2 Processing Pipeline

1. **Map Validation:** Validate incoming map data and log reception
2. **Environment Setup:** Configure planning scene with obstacles
3. **Plant Filtering:** Identify target plants for investigation
4. **Trajectory Processing:** Process each plant's possible trajectories
5. **Coordination:** Orchestrate scanning operations with detailed logging

6 System Pipeline - Execution Flow

6.1 Phase 1: System Initialization

1. **Node Creation** (`cr5_planner_node.cpp:main()`)
   - Initialize ROS2 node: `cr5_moveit_cpp_planner`
   - Configure comprehensive logging and system identification
2. **MoveIt Setup**
   - Initialize `MoveGroupInterface` for `cr5_group`
   - Configure `PlanningSceneInterface` for collision management
   - Set planning parameters:

  - Planning time: 0.5 seconds maximum
  - Planning attempts: 5 maximum retries
  - Position tolerance: 2cm precision
  - Orientation tolerance: 0.1 rad ($\approx 5.7°$)

3. **Component Initialization**
   - Create `PlantProcessor` instance with dependencies
   - Create `SceneManager` instance for obstacle management
   - Setup basic floor safety obstacle for collision prevention
4. **Communication Setup**
   - Configure map subscriber with QoS settings
   - Subscribe to `reworked_map` topic with latching
   - Setup persistent map behavior for system robustness

6.2 Phase 2: Map Reception and Processing

1. **Map Reception** (`PlantProcessor::mapCallback()`)
   - Receive map message from `reworked_map` topic
   - Validate map data integrity and log comprehensive reception details
   - Store map in `current_map_` member for processing
2. **Scene Setup** (`SceneManager::addObstaclesToScene()`)
   - Clear existing obstacles from MoveIt planning scene
   - Process each map object with detailed analysis:
     - Create collision objects for non-target objects
     - Generate precise plant collision geometries
     - Calculate accurate object centers for targeting
   - Apply all obstacles to MoveIt planning scene atomically
3. **Target Plant Processing** (`PlantProcessor::processTargetPlants()`)
   - Filter map objects for target plants (`object.target == true`)
   - For each target plant:
     - Log comprehensive plant investigation start
     - Validate trajectory availability and quality
     - Process each possible trajectory with detailed tracking

6.3 Phase 3: Plant Investigation Pipeline

For each target plant and trajectory:

1. **Scan Center Calculation** (`PlantProcessor::calculateScanCenter()`)
   - Calculate optimal center point for camera targeting
   - Use different strategies for first vs. subsequent scans
   - Consider plant bounding box geometry and accessibility
2. **Circumference Processing** (`PlantScanner::processCircumference()`)
   - Convert optimal points to `ScanPoint` structures
   - Analyze point quality, reachability, and spatial distribution
   - Log detailed scanning strategy and comprehensive point statistics
3. **Boundary Point Analysis** (`PlantScanner::identifyBoundaryPoints()`)
   - Analyze spatial distribution using advanced algorithms
   - Identify critical boundary points for priority coverage
   - Mark boundary points for Phase 1 high-priority scanning
4. **Two-Phase Scanning Strategy**

   Phase 1 - Boundary Points (`PlantScanner::processCircumference()`)
   - Attempt to reach all identified boundary points systematically
   - Use `MovementController::attemptToReachPoint()` for each point
   - Track successful visits and maintain detailed coverage statistics
   - Log progress with format: "Visited X/Y boundary points"

   Phase 2 - Remaining Coverage (`PlantScanner::scanUncoveredPoints()`)
   - Identify uncovered optimal points (quality = 1.0)
   - Attempt to visit remaining high-quality points efficiently
   - Continue until coverage goals met or no more reachable points
   - Generate comprehensive final coverage reports

6.4 Phase 4: Movement Execution Details

For each scan point attempt:

1. **Pose Generation** (`PlantScanner::createPosePointingTowardsPlant()`)
   - Calculate precise robot pose at scan point location
   - Orient end-effector/camera towards plant center accurately
   - Generate complete 6DOF pose (position + orientation)

2. **Movement Planning** (`MovementController::planMovement()`)
    - Use MoveIt to plan collision-free path with obstacles
    - Apply configured tolerances and kinematic constraints
    - Validate plan quality, safety, and execution feasibility
3. **Movement Execution** (`MovementController::executeMovement()`)
    - Send planned trajectory to robot controllers safely
    - Monitor execution for completion, errors, and safety
    - Update point coverage status with detailed tracking
4. **Coverage Tracking**
    - Mark successfully reached points as covered
    - Update comprehensive scanning statistics
    - Log detailed coverage reports with visual indicators

6.5 Phase 5: Completion and Reporting

1. **Plant Investigation Completion**
    - Log final coverage statistics for each plant with details
    - Report successful trajectory completions with metrics
    - Provide comprehensive performance metrics and analysis
2. **System Status**
    - Log overall map processing completion with summary
    - Display ASCII art status indicators for visual clarity
    - Prepare system for next map reception cycle

7 Key Design Patterns

7.1 Dependency Injection

Components receive dependencies through constructors, enabling:

- Easy unit testing and mock injection
- Flexible system composition and configuration
- Clear dependency relationships and management

Example: `PlantProcessor` receives `SceneManager` and `PlantScanner`

7.2 Single Responsibility Principle

Each module has one clear, focused responsibility:

- `MovementController`: Robot motion planning and execution
- `PlantScanner`: Scanning strategy and coverage optimization
- `SceneManager`: 3D obstacle management and collision handling
- `PlantProcessor`: Overall system coordination and orchestration

7.3 Interface Segregation

- Clean, focused public interfaces in header files
- Private implementation details hidden in .cpp files
- Clear separation between interface and implementation

7.4 Composition over Inheritance

- Modules composed of other modules rather than inherited
- Enables flexible system architecture and easy modifications
- Simple to modify or replace individual components

8 Error Handling and Safety

8.1 Movement Safety

- All movements use collision-aware planning with obstacle avoidance
- Position and orientation tolerances prevent over-precision issues
- Safe retry mechanisms for failed movements with intelligent fallback

8.2 Planning Validation

- Plans validated before execution with comprehensive checks
- Timeout mechanisms prevent infinite planning loops
- Graceful degradation when targets become unreachable

8.3 Scene Safety

- Floor obstacle prevents robot from dangerous low positions
- All obstacles properly added to collision checking system
- Scene state properly managed, cleaned, and synchronized

9 Configuration and Customization

9.1 Planning Parameters

Located in `cr5_planner_node.cpp`:

```
move_group.setPlanningTime(0.5);                    // Planning timeout
move_group.setNumPlanningAttempts(5);               // Max planning attempts
move_group.setGoalPositionTolerance(0.02);          // 2cm position tolerance
move_group.setGoalOrientationTolerance(0.1);        // Orientation tolerance
```

Listing 7: MoveIt Configuration

9.2 Camera Configuration

Located in `plant_scanner.hpp`:

```
static constexpr double CAMERA_FOV_HORIZONTAL = 0.15; // Camera FOV
```

Listing 8: Camera Parameters

9.3 Safety Configuration

Floor obstacle parameters configured in `SceneManager::setupFloorObstacle()`

10 Logging and Debugging

10.1 Structured Logging

- Hierarchical log messages with clear, consistent prefixes
- ASCII art separators for visual clarity and readability
- Progress tracking with detailed statistics and metrics

10.2 Phase-Based Reporting

- Clear identification of system phases with numbered sequences
- Success/failure indicators (checkmark/warning) for immediate status recognition
- Comprehensive coverage reporting with detailed analysis

10.3 Debug Information

- Detailed movement planning logs with step-by-step tracking
- Point-by-point coverage tracking with individual status
- Error conditions clearly reported with diagnostic information