System Verification and Validation Plan for 2D-RAPP

Ziyang(Ryan) Fang

February 25, 2025

Revision History

Date	Version	Notes
Feb 24	1.0	Notes

Contents

List of Tables

[Remove this section if it isn't needed —SS]

List of Figures

[Remove this section if it isn't needed —SS]

1 Symbols, Abbreviations, and Acronyms

Symbol	Description
q	Joint angle (rad)
$\dot{q}$	Joint angular velocity (rad/s)
$\ddot{q}$	Joint angular acceleration (rad/s^2)
L	Link length (mm)

This document describes the Verification and Validation (V&V) plan for our software system, which addresses the path-planning and inverse-kinematics (IK) verification tasks for 2D robotic manipulators. The document outlines our approach, objectives, and the tests to be performed to ensure that the system meets the specified requirements. A roadmap of the plan is as follows:

- Section 1 covers general information about the software, its purpose, and the primary V&V objectives.
- Section 2 provides details on the V&V plan, the team, and the methods and tools used.
- Section 3 describes the system-level tests for both functional and non-functional requirements.
- Section 4 focuses on unit-level testing, including scope, test coverage, and traceability to modules.

2 General Information

2.1 Summary

The software under test is a **2D Robot Arm Path-Planning and IK Solver** for manipulators with circular obstacles. The system:

- Represents the manipulator's joints and links in a distance-geometric framework.
- Uses path-planning algorithms (e.g., A*) and inverse-kinematics solvers to find feasible joint angles.
- Ensures collision-free paths around circular obstacles.

2.2 Objectives

The primary objectives of the V&V process are:

- Build confidence in software correctness by systematically testing the functional requirements (e.g., path planning, collision avoidance).

- Demonstrate acceptable performance, including computational efficiency and accuracy of solutions.
- Verify reliability, so that the solution can be used for 2D manipulator control scenarios with minimal failures.

Out of scope:

- Extensive *usability* testing. We will focus on correctness and performance rather than user interface design.
- Verification of external libraries (e.g., numerical solvers). We assume they have been validated by their providers.

2.3 Challenge Level and Extras

Challenge Level: General (per agreement with the course instructor). **Extras:**

- A brief user manual for researchers/engineers.
- Potential code walkthroughs and extended design documentation, as time and resources permit.

2.4 Relevant Documentation

- **Software Requirements Specification** (SRS) : This defines the required functionalities for the 2D robot arm path planner, including input constraints, output specifications, and performance requirements.
- **Other Project Documents** (MG, MIS, etc.): These documents describe the design and detailed modules of the system. They are relevant for deriving unit tests and ensuring that design decisions align with requirements.
- **Code Reference**: An example is the A* toroidal grid code for obstacle navigation, which will be referenced or adapted for verifying path-planning correctness.

Fang (2025)

3 Plan

This section outlines the multiple stages of the verification and validation (V&V) process. First, the V&V team is introduced. Then, verification plans for the SRS, design, V&V plan, and implementation are described. Finally, automated testing and verification tools are briefly discussed.

3.1 Verification and Validation Team

The following personnel will be involved in the verification and validation of the 2D Robot Arm Path Planning system:

- **Ziyang Fang**: The author of the program. Responsible for the creation of the V&V plan, implementation of the tests, and analysis of results.
- **Dr. Spencer Smith**: The project supervisor. Responsible for reviewing the V&V plan, test cases, and overall validation of the system.
- **Alaap Grandhi**: The domain expert. Responsible for reviewing the V&V plan, ensuring scientific correctness, and verifying test coverage.

3.2 SRS Verification Plan

To ensure the SRS is complete, correct, and consistent, we will use the following verification methods:

- **Supervisor Review**: Dr. Spencer Smith will review the SRS for completeness and clarity. A structured checklist will be provided to guide feedback.
- **Domain Expert Review**: Alaap Grandhi will review the SRS to ensure technical accuracy and feasibility.
- **Issue Tracker**: Feedback from reviews will be logged as GitHub issues, and necessary modifications will be made accordingly.
- **Peer Feedback**: Classmates will provide additional feedback to identify any unclear or missing requirements.

3.3 Design Verification Plan

- **Classmate / Peer Design Review**: We will have scheduled walkthrough sessions where classmates provide feedback on the design documents (MG, MIS).
- **Design Checklists**: We will ensure that each requirement in the SRS is addressed at the design level. Checklists will cover topics like modularity, clarity, and traceability to requirements.

3.4 Verification and Validation Plan Verification Plan

- **Internal Review**: The same team members will review this V&V plan for completeness and feasibility.
- **Mutation of Plan**: Introduce hypothetical changes (mutations) to test if the plan remains robust and comprehensive.

3.5 Implementation Verification Plan

- **Unit Tests**: We will write unit tests for each major component, automated by a suitable framework (e.g., `pytest` for Python).
- **Static Analysis**: Use a linter (e.g., `flake8`), possibly additional tools like `pylint` or `mypy`, to verify code standards and catch potential errors.
- **Code Walkthroughs**: During final presentations or peer sessions, we will review code segments dealing with core algorithms (IK solver, collision detection, path planning).

3.6 Automated Testing and Verification Tools

- **Continuous Integration (CI)**: GitHub Actions is used to run automated test suites on every pull request and commit to ensure code reliability.
- **Unit Testing Framework**: The project uses `pytest` for unit testing, which provides powerful fixtures and easy test case management.
- **Coverage Analysis**: The tool `coverage.py` is used to measure code coverage, ensuring critical components are well-tested.

- **Static Code Analysis**: `flake8` is used to check for style violations and potential errors in Python code.
- **Type Checking**: `mypy` is used to enforce type annotations and detect type inconsistencies.
- **Performance Profiling**: `cProfile` is used for runtime performance analysis to identify bottlenecks in critical functions.
- **Memory Profiling**: `memory-profiler` helps track memory usage during execution to detect leaks and optimize performance.
- **Automated Dependency Management**: `pip-tools` is used to manage dependencies and ensure a stable development environment.

3.7 Software Validation Plan

- **Supervisor/Stakeholder Review**: If an external stakeholder is available, we will present a demo (Rev 0) to validate the requirements match real-world needs.
- **Comparison with Known Benchmarks**: Use small or known scenarios to validate the solution's correctness against expected paths/angles.

4 System Tests

System-level tests will address functional requirements (such as collision avoidance and path planning success) and nonfunctional requirements (such as performance and accuracy). These tests will be conducted after individual modules pass unit tests to ensure the entire system functions as expected.

4.1 Tests for Functional Requirements

This section defines tests for key functional areas of the **2D Robot Arm Path Planning System**, ensuring that the system correctly computes collision-free paths and valid inverse kinematics (IK) solutions. The test cases are derived from the functional requirements specified in the Software Requirements Specification (SRS).

4.1.1 Path Planning and Obstacle Avoidance

These tests ensure that the system can generate valid and collision-free paths from the start to the goal configuration. **Reference: SRS Sections on Path Planning and Collision-Free Paths.**

Collision-Free Path Generation

1. **Test ID: T1**

   Control: Automatic

   Initial State:

   - Defined robotic arm parameters (link lengths, joint limits)
   - Known obstacle positions and sizes
   - Initial and goal joint configurations

   Input:

   - Initial joint angles q_{init}
   - Goal position $(x_{\text{goal}}, y_{\text{goal}})$
   - Obstacle positions and radii

   Expected Output:

   - A valid sequence of joint angles $q(t)$ ensuring obstacle avoidance
   - No intersection between the arm and any obstacle

   Test Case Derivation: Ensures that the system correctly computes a path from the initial to the goal state while maintaining a safe clearance from obstacles.

   How Test Will Be Performed:

   - Use predefined obstacle environments
   - Execute the path-planning function and validate results
   - Visually inspect plotted trajectories
   - Check computational efficiency

2. **Test ID: T2**

   Control: Automatic

   Initial State: Same as T1

   Input:

   - Multiple circular obstacles blocking the direct path

   Expected Output:

   - The planned trajectory avoids all obstacles
   - The trajectory remains kinematically feasible

   Test Case Derivation: Ensures the system can handle more complex environments and still find feasible paths.

   How Test Will Be Performed:

   - Introduce additional obstacles in varying configurations
   - Validate computed trajectories

4.1.2 Inverse Kinematics (IK) Solver Validation

These tests verify that the IK solver correctly computes joint angles for a given end-effector position. **Reference: SRS Section on Forward Kinematics and Inverse Kinematics.**

Feasibility of IK Solutions

1. **Test ID: T3**

   Control: Automatic

   Initial State: Defined robotic arm parameters

   Input:

   - A reachable target position $(x_{\text{goal}}, y_{\text{goal}})$

   Expected Output:

   - A valid set of joint angles (θ_1, θ_2)

- The computed position matches the expected end-effector position

Test Case Derivation: Ensures that inverse kinematics computations return correct and feasible joint angles.

How Test Will Be Performed:

- Compute joint angles using inverse kinematics
- Validate against forward kinematics
- Check if solutions respect joint limits

4.2 Tests for Nonfunctional Requirements

4.2.1 Performance and Computational Efficiency

These tests ensure that the system computes paths within acceptable time limits and does not exceed memory constraints. **Reference: SRS Section on Accuracy and Performance.**

1. **Test ID: N1**

   Type: Performance, Automatic

   Initial State: None

   Input:

   - A large number of obstacles (e.g., 50+)
   - Randomized goal positions

   Expected Output:

   - Execution time statistics
   - Peak memory usage report

   How Test Will Be Performed:

   - Profile the system while computing paths
   - Measure time-to-solution for varying complexity levels
   - Compare against baseline performance requirements

2. **Test ID: N2**

   Type: Scalability, Automatic

   Initial State: None

   Input:

   - Increasing number of degrees of freedom (DOF) in the robotic arm (e.g., from 2-DOF to 6-DOF)

   Expected Output:

   - System performance degradation analysis
   - Success rate of path planning for different DOF levels

   How Test Will Be Performed:

   - Measure computation time for different arm configurations
   - Evaluate feasibility of solutions at high DOF levels

4.3 Traceability Between Test Cases and Requirements

The traceability matrix below maps test cases to the corresponding requirements from the SRS to ensure complete coverage.

Table 1: Traceability Matrix for System Tests

Requirement	**Test Case(s)**	**Comments**
FR1: Single Obstacle Avoidance	T1	Ensures safe navigation around obstacles
FR2: Multiple Obstacles	T2	Tests complex environment handling
FR3: Solve Feasible IK	T3	Verifies correct IK computations
NFR1: Accuracy	N1	Validates end-effector positioning accuracy
NFR2: Performance	N2	Ensures computational efficiency

5 Unit Test Description

Unit testing will occur once the detailed designs are finalized (see the MIS). Below is the general approach for module-level testing.

5.1 Unit Testing Scope

All modules described in the MIS are considered in-scope, except for any third-party libraries. Priority is given to:

- IK Solver Module
- Collision Detection Module
- Path Planner Module

We assume external math/optimization libraries (e.g., for matrix operations) are already tested by their providers.

5.2 Tests for Functional Requirements

5.2.1 Collision Detection Module Tests

1. **Test ID**: U1
   Type: Functional, Automatic
   Initial State: Environment with at least one circular obstacle
   Input: A line segment representing a robot arm link and an obstacle defined by center position and radius
   Expected Output: Boolean indicating whether a collision occurs
   Test Case Derivation: Verify collision detection under different scenarios:

   - No collision (link does not intersect obstacle)
   - Tangential collision (link touches the edge of the obstacle)
   - Full overlap (link completely inside the obstacle)

   How Test Will be Performed: Simulate known geometries and compare algorithm outputs with analytical results.

5.2.2 IK Solver Module Tests

1. **Test ID**: U2
   Type: Functional, Automatic
   Initial State: 2-link robotic arm with fixed link lengths
   Input: Target position in Cartesian coordinates

Expected Output: Set of joint angles that place the end-effector at the target
Test Case Derivation: Verify correctness under different conditions:

- Target reachable within workspace
- Target at singularity points (e.g., fully extended or folded)
- Target unreachable (should return failure)

How Test Will be Performed: Compare solver results to analytical solutions and verify feasibility of output joint angles.

5.2.3 Path Planner Module Tests

1. **Test ID**: U3
   Type: Functional, Automatic
   Initial State: Defined workspace with static obstacles
   Input: Start and goal positions, obstacle map
   Expected Output: A sequence of joint angles forming a valid path from start to goal
   Test Case Derivation: Ensure correct path generation under:

   - No obstacles (should return a direct path)
   - Single obstacle blocking the direct path (should generate a detour)
   - Multiple obstacles with narrow passage (should generate an optimal feasible path)

   How Test Will be Performed: Compare generated paths with expected solutions in predefined test environments.

5.3 Tests for Nonfunctional Requirements

5.3.1 Performance and Scalability Tests

1. **Test ID**: U4
   Type: Performance, Automatic
   Initial State: Environment with varying numbers of obstacles
   Input: Multiple path-planning queries with increasing workspace complexity
   Expected Output: Execution time and memory usage statistics

How Test Will be Performed: Use a profiler to measure performance overhead and scalability under high computational loads.

5.3.2 Numerical Stability Tests

1. **Test ID**: U5
   Type: Accuracy, Automatic
   Initial State: Workspace with no obstacles
   Input: Multiple IK queries with floating-point precision limits
   Expected Output: Consistent and numerically stable joint angles
   How Test Will be Performed: Compare solver results with high-precision analytical solutions and detect anomalies in floating-point calculations.

5.4 Traceability Between Test Cases and Modules

Table 2: Traceability Matrix for Unit Tests

Module	Test Case(s)	Comments
Collision Detection	U1, U4	Checking correctness and performance
IK Solver	U2, U5	Ensuring correctness and numerical stability
Path Planner	U3, U4	Validates A* search efficiency and scalability

References

Ziyang (Ryan) Fang. Software requirements specification for 2d robot arms for path planning. https://github.com/FangZiyang/CAS741-Ryan/blob/main/docs/SRS/SRS.pdf, February 2025. Accessed: 2025-02-10.

6 Appendix

Any symbolic constants referenced in the test cases (like tolerances or maximum obstacle counts) can be listed here for easier maintenance.

6.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

6.2 Usability Survey Questions?

If we conduct a small-scale usability survey, we can place the questionnaire or script here.