System Verification and Validation Plan for 2D-RAPP

Ziyang(Ryan) Fang

April 17, 2025

Revision History

Date	Version	Notes
Feb 24	1.0	Notes
April 15	2.0	Notes

Contents

List of Tables

List of Figures

1 Symbols, Abbreviations, and Acronyms

symbol	description
A	Assumption
DD	Data Definition
GD	General Definition
GS	Goal Statement
IM	Instance Model
LC	Likely Change
PS	Physical System Description
R	Requirement
SRS	Software Requirements Specification
TM	Theoretical Model
IK	Inverse Kinematics
FK	Forward Kinematics
A*	A-star Pathfinding Algorithm
DOF	Degrees of Freedom
EE	End-Effector
2D-RAPP	2D Robot Arm Path Planning

This document describes the Verification and Validation (V&V) plan for our software system, which focuses on collision-free path planning for a 2D robotic manipulator in an environment with circular obstacles. The system generates a valid trajectory from a given start configuration to a goal configuration while ensuring that the robotic arm does not collide with obstacles. The verification and validation process aims to assess the correctness, efficiency, and robustness of the path-planning algorithm and collision detection mechanism. 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 System** that computes collision-free trajectories for a robotic manipulator with circular obstacles. The system:

- Discretizes the robot's joint space and represents possible configurations on a toroidal grid.
- Uses the A* algorithm to compute a valid trajectory from the start configuration to the goal configuration in joint space.
- Detects and avoids collisions by checking intersections between the robotic arm links and circular obstacles.

2.2 Objectives

The primary objective of this Verification and Validation (V&V) plan is to ensure the correctness, reliability, and efficiency of the 2D Robot Arm Path-Planning System. This system computes collision-free trajectories for a robotic manipulator navigating around circular obstacles. The specific objectives are:

- **Ensure system-level correctness and visualization integrity**: Confirm that the system initializes properly, and visualizes planned paths as expected. The system's UI and plotting components will be manually inspected to ensure correct rendering of obstacles and trajectories.
- **Validate the effectiveness of A* path planning in joint space**:Verify that the "A* Algorithm" finds valid paths when they exist and alerts when a goal is unreachable.
- **Ensure robustness of collision detection**: Confirm that obstacle avoidance is correctly implemented and that no invalid paths are generated.
- **Assess computational efficiency**: Measure execution time and memory usage to ensure the system performs well under different obstacle configurations.

Out of Scope:

- **Usability Testing**: The system is evaluated based on correctness and performance, not user experience or interface design.
- **Real-world Hardware Validation**: The plan verifies software functionality but does not include hardware implementation or physical robot testing.
- **External Library Verification**: The system assumes that third-party numerical and optimization libraries (e.g., NumPy) are already tested by their providers.

This plan prioritizes correctness, efficiency, and collision-free navigation, acknowledging that usability and physical testing are beyond the current project scope.

2.3 Challenge Level and Extras

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

- A brief user manual for researchers/engineers.

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.(Fang, 2025c)
- **Other Project Documents** (MG (Fang, 2025a), MIS (Fang, 2025b), 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.(code link).

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.Feedback and suggestions were documented in GitHub Issue. GitHub Issue example.
- **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 (Fang, 2025a), MIS (Fang, 2025b)).
- **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 (`pytest` for Python).
- **Static Analysis**: Use a linter (`flake8`), possibly additional tools like `pylint` or `mypy`, to verify code standards and catch potential errors.
- **Code Walkthroughs**: During the final presentation, we demonstrated the software's execution using example scenarios, walking through key modules such as path planning and collision detection, and explaining how they work together to generate a valid trajectory.

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**. The goal is to ensure that the system correctly computes collision-free paths and generates feasible inverse kinematics (IK) solutions using a **distance-geometric representation**.

The test cases are derived from the functional requirements specified in the Software Requirements Specification (SRS).

4.1.1 Path Planning and Obstacle Avoidance

This section ensures that the system can generate valid and collision-free paths from the start to the goal configuration under different environmental constraints.

Collision-Free Path Generation

1. **Test ID: T1 – Basic Collision-Free Path Planning**

   Control: Automatic

   Initial State:

   - A 2-joint planar robotic arm with link lengths [1, 1] (meters).

- Three circular obstacles placed at coordinates:
  - (1.75, 0.75) with radius 0.6 m
  - (0.55, 1.5) with radius 0.5 m
  - (0, -1) with radius 0.25 m
- Initial joint configuration: $q_{\text{init}} = [10°, 50°]$
- Goal joint configuration: $q_{\text{goal}} = [58°, 56°]$

Input:

- Discretized joint indices: start = (10, 50), goal = (58, 56)
- Link lengths = [1, 1] m
- Obstacles = `[[1.75, 0.75, 0.6], [0.55, 1.5, 0.5], [0, -1, 0.25]]`
- Joint limits = $[-180°, 180°]$ for each joint

Output:

- A sequence of joint angle vectors $q(t) \in \mathbb{R}^2$ from start to goal
- Each intermediate configuration is free of collision (validated by segment-circle intersection test)
- The end-effector trajectory avoids all obstacles

Test Case Derivation:

- Uses `Example 1` from the dataset.
- Validates that the A* algorithm on the toroidal grid finds a solution (non-empty path).
- Checks that all configurations in the returned path pass collision detection using `detect_collision()`.

How Test Will Be Performed:

- Load Example 1 data into the system.
- Generate a 100x100 occupancy grid using `get_occupancy_grid()`.
- Run `astar_torus()` with the given start and goal.
- For each node in the path:

  - Convert indices to radians
  - Apply joint angles to update arm configuration
  - Check for collisions
- Log whether path is found, path length, and collision-free status.

2. **Test ID: T2 – Path Planning with Multiple Obstacles**

   Control: Automatic

   Initial State:

   - A 2-link planar robotic arm with link lengths [1, 1].
   - Initial joint angles $q_{\text{init}} = (40^\circ, 40^\circ)$.
   - Goal joint angles $q_{\text{goal}} = (60^\circ, 60^\circ)$.
   - Joint limits: $[-180^\circ, 180^\circ]$ for both joints.
   - Three circular obstacles:
     - Obstacle 1: center at (1.0, 1.0), radius 0.6
     - Obstacle 2: center at (-1.2, 0.3), radius 0.3
     - Obstacle 3: center at (0.0, -1.0), radius 0.5

   Input:

   - Discretized joint space of size $M = 100$.
   - Obstacle configuration and initial/goal joint angles as above.

   Output:

   - A feasible joint trajectory $q(t) \in \mathbb{R}^2$ avoiding all obstacles.
   - No intersection between any arm segment and obstacles at all time steps.

   Test Case Derivation:

   - Demonstrates the algorithm's capability in environments with tight spaces and multiple obstacle clusters.
   - Confirms the planner generates alternative collision-free routes when the direct path is blocked.

   How Test Will Be Performed:

- Use Example 4 from the implemented test suite in ‘examples.py‘.
- Run the planner to compute a trajectory from start to goal.
- Visually verify the trajectory with obstacle overlay using the GUI animation.
- Ensure no error or collision flags are raised during planning.

4.1.2 Inverse Kinematics (IK) Solver Validation

This section verifies that the system's inverse kinematics solver correctly computes feasible joint configurations for a given end-effector position.

Feasibility of IK Solutions

1. **Test ID: T3 – Basic IK Feasibility**

   Control: Automatic

   Initial State:

   - A robotic arm with n joints and known link lengths.

   Input:

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

   Output:

   - A valid set of joint angles $(q_1, q_2, ..., q_n)$ that achieve the goal position.

   Test Case Derivation:

   - Ensures that inverse kinematics computations return correct and feasible joint angles.
   - Verifies that the end-effector reaches the expected position.

   How Test Will Be Performed:

   - Compute joint angles using the distance-based graph method.
   - Validate against forward kinematics results.

- Check if solutions respect joint limits and geometric constraints.

2. **Test ID: T4 – IK for Complex Configurations**

   Control: Automatic

   Initial State:

   - A robotic arm with n joints and predefined joint constraints.

   Input:

   - A target position that may have multiple valid joint configurations.

   Output:

   - The solver returns multiple possible configurations.
   - The solver prioritizes the most optimal (e.g., minimal joint movement).

   Test Case Derivation:

   - Ensures that the solver can handle redundancy in joint configurations.
   - Verifies that the system can optimize movement efficiency.

   How Test Will Be Performed:

   - Run multiple test cases with different initial configurations.
   - Compare the generated solutions and their efficiency.

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 – Path Planning Performance Under High Obstacle Density**

   Type: Performance, Automatic

   Initial State:

   - A 2-link robotic arm with link lengths [1, 1] and joint limits of $[-180°, 180°]$ for each joint.

   Input:

   - A joint space discretization of $M = 100$.
   - 20 circular obstacles randomly placed within a 2m x 2m workspace.

   Expected Output:

   - Average path planning time per query (measured in seconds).
   - Maximum and average memory usage during grid generation and A* search.

   How Test Will Be Performed:

   - Use Python's `time` and `memory_profiler` modules to collect runtime and memory data.
   - Compare against baseline target: path planning should finish in under 10 second.

2. **Test ID: N2**

   Type: Scalability, Automatic

   Initial State: None

   Input:

   - Increasing number of degrees of freedom (DOF) in the robotic arm (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:

- Modify the number of joints in the robotic arm (2–6 DOF).
- For each DOF level, run the planner on a fixed obstacle set and log:
  - Execution time.
  - Path success or failure.
- Analyze whether the planner remains efficient and reliable at higher DOFs.

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 **Module Interface Specification (MIS)**). This section outlines the general approach to module-level testing, ensuring that core functionalities perform correctly and efficiently.

5.1 Unit Testing Scope

All modules described in the **MIS** are in-scope for unit testing, except for any third-party libraries. The following modules have been identified as the highest priority:

- **Inverse Kinematics (IK) Solver** – Computes joint angles for a given target position.
- **Collision Detection Module** – Ensures that paths do not intersect obstacles.
- **Path Planner Module** – Generates valid paths based on obstacles and goal configurations.

External libraries for mathematical operations (e.g., matrix solvers) are assumed to be tested by their providers and are excluded from unit testing.

5.2 Unit Testing Strategy

The testing strategy follows a combination of:

- **Black-box testing:** Verifying correct outputs given specific inputs.
- **White-box testing:** Checking internal logic using test coverage tools.
- **Edge-case testing:** Ensuring robustness for boundary conditions.
- **Automated testing:** Using scripted tests for reproducibility via continuous integration (CI) with GitHub Actions.

Unit tests will be developed in Python, following the **pytest** framework, with assertions to check correctness.

5.3 Tests for Functional Requirements

5.3.1 Collision Detection Module Tests

1. **Test ID: U1 – Basic Collision Detection**

   Type: Functional, Automatic

   Initial State: Environment with at least one circular obstacle

   Input:

- A line segment representing a robot arm link.
- An obstacle defined by center position and radius.

Expected Output:

- Boolean indicating whether a collision occurs.

Test Case Derivation:

- Ensures correct detection under different cases:
  - No collision (link does not intersect obstacle)
  - Tangential collision (link touches the obstacle's boundary)
  - Full overlap (link completely inside the obstacle)

How Test Will Be Performed:

- Simulate known geometries.
- Compare algorithm outputs with analytical results.

—

5.3.2 IK Solver Module Tests

1. **Test ID: U2 – IK Feasibility Check**

   Type: Functional, Automatic

   Initial State: 2-link robotic arm with fixed link lengths

   Input:

   - A target position in Cartesian coordinates.

   Expected Output:

   - A set of joint angles that place the end-effector at the target.

   Test Case Derivation:

   - Verifies correctness under different conditions:
     - Target reachable within workspace.
     - Target at singularity points (fully extended or folded).

– Target unreachable (should return failure).

How Test Will Be Performed:

- Compute joint angles using the IK solver.
- Compare results against analytical solutions.
- Verify feasibility of the computed joint angles.

5.3.3 Path Planner Module Tests

1. **Test ID: U3 – Path Generation in Obstacle-Free Environment**

   Type: Functional, Automatic

   Initial State: Defined workspace with no obstacles

   Input:

   - Start and goal positions.

   Expected Output:

   - A direct path connecting start and goal.

   Test Case Derivation:

   - Ensures path planning works correctly when there are no obstacles.

   How Test Will Be Performed:

   - Run the path-planning function.
   - Check whether the generated path follows a direct trajectory.

5.4 Tests for Nonfunctional Requirements

5.4.1 Performance and Computational Efficiency

1. **Test ID: N1 – Performance Under Heavy Obstacle Load**

   Type: Performance, Automatic

   Initial State:

- A workspace with no obstacles (baseline).

Input:

- A varying number of obstacles (e.g., 10, 50, 100).

Expected Output:

- A table summarizing execution time for different obstacle densities.

Test Case Derivation:

- Ensures system performs within acceptable runtime limits even in complex environments.

How Test Will Be Performed:

- Profile the system using `cProfile`.
- Measure time-to-solution for varying complexity levels.

5.5 Traceability Between Test Cases and Modules

Table 2: Traceability Matrix between Modules and Tests

Test ID	Module	Requirement Covered	Reference
U1	Collision Detection	Ensures collision-free motion	SRS Section 4.2.3, 4.2.4
U2	IK Solver	Computes valid joint angles	SRS Section 4.3.2
U3	Path Planner	Generates valid paths	SRS Section 4.3.3, 4.3.4
N1	Performance	Ensures efficient execution	SRS Section 5.1

Summary: This document outlines unit test cases for the key modules in the 2D Robot Arm Path Planning System. The testing strategy incorporates functional tests for correctness, performance tests for efficiency, and numerical tests for stability, ensuring that the system is robust and scalable.

References

Ziyang Fang. Module guide for 2d robotic arm path planning system. Technical report, McMaster University, Department of Computing and Software, March 2025a. URL https://github.com/FangZiyang/CAS741-Ryan/blob/main/docs/MG/MG.pdf. Accessed: 2025-04-14.

Ziyang Fang. Module interface specification for 2d robotic arm path planning system. Technical report, McMaster University, Department of Computing and Software, March 2025b. URL https://github.com/FangZiyang/CAS741-Ryan/blob/main/docs/MIS/MIS.pdf. Accessed: 2025-04-14.

Ziyang Fang. Software requirements specification for 2d robotic arm path planning system. Technical report, McMaster University, Department of Computing and Software, February 2025c. URL https://github.com/FangZiyang/CAS741-Ryan/blob/main/docs/SRS/SRS.pdf. Accessed: 2025-04-14.

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.