Module Interface Specification for 2D-RAPP

Ziyang(Ryan) Fang

March 20, 2025

1 Revision History

Date	Version	Notes
2025 March 19	1.0	Notes

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at https://github.com/FangZiyang/CAS741-Ryan/blob/main/docs/SRS/SRS.pdf.

Contents

3 Introduction

The following document details the Module Interface Specifications for **2D Robot Arm Path Planning**

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/FangZiyang/CAS741-Ryan.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by 2D-RAPP.

Data Type	**Notation**	**Description**
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in $(-\infty, \infty)$
natural number	$\mathbb{N}$	a number without a fractional component in $[1, \infty)$
real	$\mathbb{R}$	any number in $(-\infty, \infty)$

The specification of 2D-RAPP uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, 2D-RAPP uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	Level 2
Hardware Hiding	
Behaviour Hiding	Input Parameters Module Output Format Module Output Verification Module Inverse Kinematics Solver Module Configuration Management Module Path Planning Module Collision Detection Module
Software Decision	Plotting Module

Table 1: Module Hierarchy

6 MIS of Path Planning Module

6.1 Module

Path Planning Module

6.2 Uses

This module interacts with the following components:

- **Input Parameters Module**: Provides start and goal configurations, obstacles, and robot parameters.
- **Collision Detection Module**: Ensures that planned paths avoid obstacles.
- **Output Format Module**: Handles the formatting and visualization of the planned path.

6.3 Syntax

6.3.1 Exported Constants

- `MAX_ITERATIONS` : $\mathbb{N}$ (Maximum iterations for path search, default: 10,000)
- `HEURISTIC_FACTOR` : $\mathbb{R}$ (Scaling factor for heuristic function in A* search, default: 1.0)

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
`planPath`	$\mathbb{R}^n \times \mathbb{R}^n \times \mathit{Obstacles} \times \mathit{RobotParams}$	sequence of $\mathbb{R}^n$	NoPathFoundException

6.4 Semantics

6.4.1 State Variables

None. The module does not maintain any persistent state.

6.4.2 Environment Variables

None. The module does not interact with external devices.

6.4.3 Assumptions

- The robotic arm operates in a 2D plane.
- The environment is known beforehand and does not change dynamically.
- Obstacles are represented as circles with given positions and radii.
- The A* algorithm is used for path search in joint space.

6.4.4 Access Routine Semantics

`planPath`(start: $\mathbb{R}^n$, goal: $\mathbb{R}^n$, obstacles: *Obstacles*, robotParams: *RobotParams*):

- **transition**: Computes an optimal path from *start* to *goal* in joint space using A* search, ensuring that the path avoids obstacles.
- **output**: *path* is a valid and optimal sequence of robot configurations in $\mathbb{R}^n$, satisfying the following properties:

$$\text{isOptimalPath}(p) \equiv \forall q \in \text{validPaths}(start, goal), \quad \text{cost}(p) \leq \text{cost}(q)$$

$$\text{validPath}(p) \equiv \bigwedge_{i=0}^{|p|-1} \text{collisionFree}(p_i, p_{i+1}) \land p_0 = start \land p_{|p|} = goal$$

$$\text{cost}(p) = \sum_{i=0}^{|p|-1} \text{distance}(p_i, p_{i+1})$$

$$\text{heuristic}(q) = \min_{q' \in \text{neighbors}(q)} \text{distance}(q', goal)$$

- **exception**: Raises `NoPathFoundException` if no valid collision-free path exists from *start* to *goal*.

6.4.5 Local Functions

`heuristicCost`(config1: $\mathbb{R}^n$, config2: $\mathbb{R}^n$):

- output: real, heuristic cost for A* search.

`distance`(config1: $\mathbb{R}^n$, config2: $\mathbb{R}^n$):

- output: real, actual distance between configurations.

7 MIS of Collision Detection Module

7.1 Module

Collision Detection

7.2 Uses

This module interacts with:

- **Input Parameters Module** (to obtain obstacle definitions and robot dimensions)
- **Path Planning Module** (to provide collision checks for planned paths)
- **Inverse Kinematics Solver Module** (to verify joint configurations are collision-free)
- **Configuration Management Module** (to access stored robot and obstacle data)
- **Logging and Debugging Module** (to record collision detection results for debugging)

7.3 Syntax

7.3.1 Exported Constants

None.

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
`checkCollision`	ADT angle info $\mathbb{R}^n \times \mathit{Obstacles} \times \mathit{RobotParams}$	boolean	InvalidConfiguration
`checkPathCollision`	sequence of ADT angle info $\mathbb{R}^n$ $\times$ Obstacles $\times$ Robot-Params	boolean	InvalidPath

7.4 Semantics

7.4.1 State Variables

None. This module does not maintain any persistent state.

7.4.2 Environment Variables

None. This module does not interact with external devices or environments.

7.4.3 Assumptions

- Obstacles are defined as circles with a given position and radius.
- Robot arm links are modeled as line segments between joint positions.
- The environment and obstacle positions remain static during collision detection.

7.4.4 Access Routine Semantics

`checkCollision`($config : \mathbb{R}^n$, $obstacles : Obstacles$, $robotParams : RobotParams$):

- transition: Computes whether the given configuration results in a collision by applying forward kinematics to determine the position of each joint and link. The joint positions are calculated as:

  $$p_0 = (0, 0), \quad p_i = p_{i-1} + L_i \begin{bmatrix} \cos(\theta_i) \\ \sin(\theta_i) \end{bmatrix}, \quad \forall i \in \{1, ..., n\}$$

  where: - p_i is the position of the i-th joint, - L_i is the link length, - θ_i is the joint angle.

  For each obstacle $O_k = (x_k, y_k, r_k)$, the minimum distance between each robot link segment and the obstacle is computed:

$$d = \frac{|(x_2 - x_1)(y_1 - y_k) - (y_2 - y_1)(x_1 - x_k)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

where (x_1, y_1) and (x_2, y_2) are endpoints of a robot link. A collision occurs if $d \leq r_k$.

- output: Returns `true` if any link collides with an obstacle; otherwise, returns `false`.
- exception: Raises `InvalidConfiguration` if the configuration violates joint constraints or is out of the robot's feasible workspace.

`checkPathCollision`$(path : sequence\ of\ \mathbb{R}^n,\ obstacles : Obstacles,\ robotParams : RobotParams)$:

- transition: Iterates over a given discrete sequence of configurations $path = [q_1, q_2, ..., q_m]$, checking each configuration q_i using `checkCollision`. The path is validated using:

$$\forall q_i \in path, \quad \texttt{checkCollision}(q_i, obstacles, robotParams) = false$$

  If any configuration q_i fails this check, the path is marked as invalid.

- output: Returns `true` if any configuration in the provided path results in a collision; otherwise, returns `false`.
- exception: Raises `InvalidPath` if the provided path contains invalid configurations or does not conform to robot motion constraints.

7.4.5 Local Functions

The following functions are local specification tools that clarify the module's collision-checking logic:
`forwardKinematics`$(config : \mathbb{R}^n,\ robotParams : RobotParams)$:

- output: A sequence of joint positions representing the locations of all joints and the end-effector, used for collision checking.

`detectSegmentCollision`$(pointA : \mathbb{R}^2,\ pointB : \mathbb{R}^2,\ obstacle : (\mathbb{R}^2, \mathbb{R}))$:

- output: `true` if the line segment defined by `pointA` and `pointB` intersects the obstacle; `false` otherwise.

8 MIS of Input Parameters Module

8.1 Module

Input Parameters Module

8.2 Uses

This module interacts with:

- **Configuration Management Module** (for validating and managing stored parameters)
- **Path Planning Module** (to provide robot configuration, obstacles, and joint limits)
- **Collision Detection Module** (to verify that loaded parameters do not lead to invalid configurations)

8.3 Syntax

8.3.1 Exported Constants

None.

8.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`loadParams`	ADT	-	FileNotFound, InvalidFormat, MissingParameter, InvalidValue
`getRobotParams`	-	ADT	-
`getObstacles`	-	ADT	-

8.4 Semantics

8.4.1 State Variables

- $RobotParams$: Stores the dimensions and joint constraints of the robot.
- $Obstacles$: A set of obstacles represented as tuples $(center, radius)$.
- $InitialConfig$: The starting configuration of the robot in joint space.
- $GoalConfig$: The goal configuration of the robot in joint space.

8.4.2 Environment Variables

- **inputFile**: sequence of ADT, where each ADT corresponds to a parameter read from an external file.

8.4.3 Assumptions

- The input file contains properly formatted parameters.
- The input file follows a specific order in listing parameters.

8.4.4 Access Routine Semantics

`loadParams`(filePath: string):

- transition: Loads robot parameters, obstacles, and configurations from the specified file.
- exception: Raises `FileNotFound` if the file does not exist. Raises `InvalidFormat`, `MissingParameter`, or `InvalidValue` if any input is invalid.

`getRobotParams`():

- output: Returns $RobotParams$, a structure containing the robot's dimensions and joint constraints.

8.4.5 Local Functions

`parseParameter`:

- output: Extracts numerical values from the input string and converts them to the appropriate type.

9 MIS of Inverse Kinematics Solver Module

9.1 Module

Inverse Kinematics Solver

9.2 Uses

This module interacts with:

- **Input Parameters Module** (to obtain robot kinematic parameters and constraints)
- **Path Planning Module** (to generate valid joint-space trajectories)
- **Collision Detection Module** (to ensure the computed inverse kinematics solutions do not result in collisions)

9.3 Syntax

9.3.1 Exported Constants

None.

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
`solveIK`	$point\mathbb{R}^2 \times RobotParams$	Seq(ADT(AngleInfo))	NoSolutionException

9.4 Semantics

9.4.1 State Variables

None. This module does not maintain any persistent state.

9.4.2 Environment Variables

None. This module does not interact with external devices.

9.4.3 Assumptions

- The forward kinematics function FK is well-defined and differentiable.
- The workspace coordinates provided are within the robot's reachable region.
- The Jacobian matrix is not singular when computing inverse kinematics.

9.4.4 Access Routine Semantics

`solveIK`(target: $\mathbb{R}^m$, robotParams: RobotParams):

- transition: Computes the joint configuration q that satisfies the inverse kinematics equation:

  $$FK(q) = target$$

  where $FK : \mathbb{R}^n \to \mathbb{R}^m$ is the forward kinematics function of the robot. In the case where FK^{-1} is not explicitly defined, an iterative numerical method is used to approximate q.

  Given an initial guess q_0, the update rule follows Newton-Raphson iteration:

  $$q_{k+1} = q_k + J(q_k)^{-1}(target - FK(q_k))$$

  where $J(q) = \frac{\partial FK(q)}{\partial q}$ is the Jacobian matrix.

  If $J(q)$ is singular or near-singular, a damped least-squares (DLS) method is applied:

  $$q_{k+1} = q_k + J(q_k)^T (J(q_k)J(q_k)^T + \lambda^2 I)^{-1}(target - FK(q_k))$$

  where λ is a small damping factor.

- output: A sequence of valid joint-space solutions q in $\mathbb{R}^n$ such that:

  $$\|FK(q) - target\| \leq \epsilon$$

  where ϵ is a predefined tolerance for numerical accuracy.

- exception: Raises `NoSolutionException` if no valid joint-space solution exists within the given iteration limit.

9.4.5 Local Functions

`forwardKinematics`(q: $\mathbb{R}^n$, robotParams: RobotParams):

- output: Returns the corresponding workspace coordinates $FK(q)$.

`jacobianMatrix`(q: $\mathbb{R}^n$, robotParams: RobotParams):

- output: Computes the Jacobian matrix $J(q) = \frac{\partial FK(q)}{\partial q}$, used in numerical inverse kinematics solutions.

`numericalIK`(target: $\mathbb{R}^m$, robotParams: RobotParams):

- output: Uses an iterative method (such as Newton-Raphson) to compute an approximate inverse kinematics solution.

10 MIS of Output Verification Module

10.1 Module

Output Verification Module

10.2 Uses

This module interacts with:

- **Path Planning Module** (to verify the correctness of planned paths)
- **Collision Detection Module** (to check if the final path avoids collisions)
- **Input Parameters Module** (to access the input parameters and thresholds for validation)

10.3 Syntax

10.3.1 Exported Constants

- `ADMIS_ER` $= 1 \times 10^{-6}$ (Tolerance threshold for acceptable numerical errors)

10.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`verifyOutput`	$path : ADT(Path), obstacles : ADT(Obstacles), robotParams : ADT(RobotParams)$	-	PATH_INVALID, COLLISION_DETECTED

10.4 Semantics

10.4.1 State Variables

None. This module does not maintain any persistent state.

10.4.2 Environment Variables

None. This module does not interact with external devices.

10.4.3 Assumptions

- The computed path is a discrete sequence of configurations in $\mathbb{R}^n$.
- Obstacles are represented as closed regions in $\mathbb{R}^2$ or $\mathbb{R}^3$.
- The verification module ensures that the planned path is both valid and collision-free.
- The energy conservation check ensures numerical accuracy.

10.4.4 Access Routine Semantics

`verifyOutput`$(path, obstacles, robotParams)$:

- transition: Validates the computed path to ensure:

$$\forall q_i \in path, \quad q_i \text{ satisfies joint constraints and does not intersect obstacles.}$$

- output: If the path passes all verification checks, the module does not return an explicit output.
- exception: Raises:
  - `PATH_INVALID` if any configuration q_i violates kinematic constraints.
  - `COLLISION_DETECTED` if any segment in the path intersects an obstacle:

$$\exists i, \quad detectCollision(q_i, obstacles) = \text{true}.$$

10.4.5 Local Functions

`detectCollision`$(q, obstacles)$:

- output: Returns `true` if q results in a collision with any obstacle; otherwise, returns `false`.

`computePathError`$(path)$:

- output: Computes the total deviation in path smoothness:

$$\sum_{i=0}^{n-2} ||q_{i+1} - q_i||.$$

11 MIS of Plotting Module

11.1 Module

Plotting

11.2 Uses

This module interacts with:

- **Output Format Module** (to format the the data that is correctly structured before plotting)

11.3 Syntax

11.3.1 Exported Constants

None.

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
`plot`	$pointR^2$, $RobotParams$, $Obstacles$	-	-

11.4 Semantics

11.4.1 State Variables

None. This module does not maintain any persistent state.

11.4.2 Environment Variables

- `win`: 2D graphical window displaying the plot.

11.4.3 Assumptions

- The plotting module receives validated input data.
- The visualization environment supports real-time rendering.
- The plotted path and metrics remain static during execution.

11.4.4 Access Routine Semantics

`plotPath`(*path*):

- transition: Reads the planned path and generates a 2D trajectory plot using Python's Matplotlib.
- output: Displays the robot's motion along the planned path, with:
  - Waypoints marked as circles.
  - The robot's trajectory visualized as a continuous line.
  - A legend showing the start and goal positions.
- exception: Raises `PLOT_ERROR` if path data is empty or not formatted correctly.

`plotMetrics`(*metrics*):

- transition: Reads performance metrics (e.g., execution time, path length) and generates a bar chart or line graph.
- output: Displays a graphical comparison of metrics with labeled axes.
- exception: Raises `PLOT_ERROR` if metric data is missing or improperly formatted.

11.4.5 Local Functions

The following helper functions are used internally for plotting:
`drawTrajectory`(path):

- Reads the sequence of waypoints and plots them on a 2D plane.

`addAnnotations`(plot):

- Adds titles, labels, and legends to the generated plot.

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

12 Appendix

[Extra information if required —SS]

Appendix — Reflection

[Not required for CAS 741 projects —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?
4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?
5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)
6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)