Module Interface Specification for 2D-RAPP

Ziyang(Ryan) Fang

April 19, 2025

1 Revision History

Date	Version	Notes
March 19 2025	1.0	Notes
April 05 2025	2.0	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 Controll Module Data Types Module
Software Decision	Plotting Module

Table 1: Module Hierarchy

6 MIS of Path-Planning Module

6.1 Module

Path-Planning

6.2 Uses

- **Input Parameters Module** (provides start/goal configurations, obstacles, robot parameters)
- **Collision Detection Module** (verifies that candidate paths are collision–free)
- **Output Format Module** (formats the sequence of configurations for visualisation)
- **Data-Types Module**

6.3 Syntax

6.3.1 Exported Constants

`MAX_ITER` : $\mathbb{N}$ (default 10 000)

`HEURISTIC_K` : $\mathbb{R}$ (default 1.0)

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
`planPath`	`Config` × `Config` × `Obstacles` × `RobotParams`	`Path`	`PathNotFound`

6.4 Semantics

6.4.1 State Variables

None.

6.4.2 Assumptions

- The module receives complete and valid *start*, *goal*, *obstacles* and *robot parameters* from the Input-Parameters Module.
- Joint-space is treated as an n-dimensional torus (wrap-around at 360°).

6.4.3 Access Routine Semantics

`planPath(start, goal, obstacles, robotParams)`

- **output:** A feasible path $p = \langle q_0, \ldots, q_m \rangle$ such that:
  - $q_0 = start$
  - $q_m = goal$
  - $\forall i \in [0, m-1]$: $q_{i+1} \in succ(q_i)$ and $\neg\text{checkCollision}(q_i, obstacles, robotParams)$
  - The path is generated using A* with cost function: $f(q_i) = gCost(q_0, q_i) + hCost(q_i)$
- **exception:** `PathNotFound` if no feasible path is found within `MAX_ITER`.

6.4.4 Local Functions

$\texttt{succ}(q)$ returns the neighbouring configurations of q according to the lattice resolution.

$\texttt{gCost}(q_i, q_{i+1}) \;=\; \|q_{i+1} - q_i\|_2$

$\texttt{hCost}(q) \;=\; \texttt{HEURISTIC\_K}\, \|q - goal\|_2$

7 MIS of Collision-Detection Module

7.1 Module

Collision-Detection

7.2 Uses

- **Input Parameters Module** (provides *obstacles* and *robot parameters*)

7.3 Syntax

7.3.1 Exported Types

`Config` (as defined in §6)

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
`checkCollision`	`Config` × `Obstacles` × `RobotParams`	boolean	`InvalidConfig`

7.4 Semantics

7.4.1 State Variables

None.

7.4.2 Assumptions

None

7.4.3 Access Routine Semantics

`checkCollision`

- **description:** Let $P = \langle p_0, \dots, p_n \rangle$ be the joint positions obtained from `forwardKinematics`$(config, r$
  A link $\overline{p_{i-1}p_i}$ collides with obstacle $O = \langle c, r \rangle$ iff

$$d(\overline{p_{i-1}p_i}, c) \leq r \quad \text{where} \quad d = \frac{|(x_2 - x_1)(y_1 - y_c) - (y_2 - y_1)(x_1 - x_c)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

- **output:** `true` iff any link collides with any obstacle.
- **exception:** `InvalidConfig` if joint limits are violated.

7.4.4 Local Functions

`checkPathCollision` returns `true` if any configuration in the path collides with an obstacle.

`forwardKinematics` : `Config` $\times$ `RobotParams` $\rightarrow$ sequence of $\mathbb{R}^2$

8 MIS of Inverse-Kinematics Solver Module

8.1 Module

Inverse-Kinematics Solver

8.2 Uses

- **Input Parameters Module**
- **Collision Detection Module**

8.3 Syntax

8.3.1 Exported Access Programs

Name	In	Out	Exceptions
`solveIK`	`Point` $\times$ `RobotParams`	sequence of `Config`	`NoSolution`

8.4 Semantics

8.4.1 State Variables

None.

8.4.2 Assumptions

- The forward-kinematics map FK : `Config` $\rightarrow$ `Point` is continuous and differentiable.
- The target point lies inside the reachable workspace.

8.4.3 Access-Routine Semantics

`solveIK`**(Point, RobotParams)**

- **output:** A finite sequence $S = \langle q_0, \ldots, q_k \rangle$ of joint configurations such that $\|FK(q_i) - \mathit{target}\| \leq \varepsilon$ for every $q_i \in S$.
- **exception:** `NoSolution` raised if the iterative algorithm exceeds `MAX_ITER` without satisfying the tolerance.

9 MIS of Output-Verification Module

9.1 Module

Output-Verification

9.2 Uses

- **Collision Detection Module**

9.3 Syntax

9.3.1 Exported Constant

$\texttt{TOL\_ERR} = 10^{-6}$

9.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`verifyPath`	`Path` × `Obstacles` × `RobotParams`	—	`PathInvalid`, `Collision`

9.4 Semantics

9.4.1 State Variables

None.

9.4.2 Assumptions

The supplied path has already been discretised into configurations of type `Config`.

9.4.3 Access-Routine Semantics

`verifyPath`

- **description:**
  1. Check joint-limit and self-collision constraints for every $q \in$ `path`.
  2. Call `checkCollision` from the Collision-Detection Module.
- **exception:**
  - `PathInvalid` if any configuration violates step 1.

– `Collision` if step 2 reports a collision.

10 MIS of Plotting Module

10.1 Module

Plotting

10.2 Uses

None.

10.3 Syntax

10.3.1 Exported Access Programs

Name	In	Out	Exceptions
`plotPath`	`Path`	—	`PlotErr`
`plotMetrics`	`MetricTable`	—	`PlotErr`

10.4 Semantics

10.4.1 State Variables

None.

10.4.2 Environment Variables

- win : handle to the active 2-D graphics window.

10.4.3 Assumptions

The graphics back-end supports real-time rendering.

10.4.4 Access-Routine Semantics

`plotPath`

- **description:** Clears win and draws: way-points (circles), continuous trajectory (poly-line), labels for start/goal.
- **exception:** `PlotErr` if `path` is empty.

`plotMetrics`

- **description:** Replaces the contents of *win* with a bar- or line-chart of the supplied performance metrics.
- **exception:** `PlotErr` if the table is ill-formed.

11 MIS of Control Module

11.1 Module

Control

11.2 Uses

- Input-Parameters, Path-Planning, Collision-Detection, Inverse-Kinematics, Output-Verification, Plotting

11.3 Syntax

11.3.1 Exported Access Programs

Name	In	Out	Exceptions
`execute`	—	`Path`	`CtrlErr`

11.4 Semantics

11.4.1 State Variables

None.

11.4.2 Assumptions

All subordinate modules are already initialised.

11.4.3 Access-Routine Semantics

`execute`

- **output:** Returns the verified, collision-free `Path` p produced by the following explicit calls:

  1. $(init, goal, rParam, obs)$ = **Input-Parameters::**`getAll`$()$
  2. p_0 = **Path-Planning::**`planPath`$(init, goal, obs, rParam)$
  3. **Collision-Detection::**`checkCollision`$(p_0, obs, rParam)$
  4. p = **Output-Verification::**`verifyPath`$(p_0, obs, rParam)$
  5. **Plotting::**`plotPath`(p)

- **exception:** `CtrlErr` if any invoked access routine raises an exception.

12 MIS of Data-Types Module

12.1 Module

Data-Types

12.2 Uses

None. This module is imported by other modules to share common abstract data types (ADTs).

12.3 Syntax

12.3.1 Exported Types

$\texttt{Point} \equiv \mathbb{R}^2$ (Cartesian coordinate in the plane)

$\texttt{ObstacleT} \equiv \langle c : \texttt{Point},\ r : \mathbb{R} \rangle$

$\texttt{Obstacles} \equiv$ sequence of $\texttt{ObstacleT}$

$\texttt{RobotParams} \equiv \langle L : \text{sequence of } \mathbb{R},\ \mathit{jointLim} : \text{sequence of } \langle \ell : \mathbb{R},\ u : \mathbb{R} \rangle \rangle$

$\texttt{Config} \equiv \mathbb{R}^n$ (vector of joint angles)

$\texttt{Path} \equiv$ sequence of $\texttt{Config}$

$\texttt{MetricTable} \equiv$ set of key–value pairs $\langle \mathit{name} : \texttt{string},\ \mathit{value} : \mathbb{R} \rangle$

12.3.2 Exported Constants

None.

12.3.3 Exported Access Programs

None. The module only publishes type definitions.

12.4 Semantics

12.4.1 State Variables

None.

12.4.2 Assumptions

- All numeric quantities are expressed in SI units (metres, radians, seconds) unless stated otherwise.
- The dimension n in `Config` equals the number of revolute joints in the robot and is fixed at run-time by `RobotParams`.

12.4.3 Access-Routine Semantics

Not applicable – no routines are exported.

12.4.4 Local Functions

None.

13 MIS of Input-Parameters Module

13.1 Module

Input-Parameters

13.2 Uses

- **Data-Types Module** (for `Config`, `RobotParams`, `Obstacles`)

13.3 Syntax

13.3.1 Exported Types

None. All returned values use types defined in the Data-Types module.

13.3.2 Exported Constants

None.

13.3.3 Exported Access Programs

Name	In	Out	Exceptions
`getAll`	—	`Config` $\times$ `Config` $\times$ `RobotParams` $\times$ `Obstacles`	`ParamErr`

13.4 Semantics

13.4.1 State Variables

- *start* : `Config` (initial joint configuration)
- *goal* : `Config` (target joint configuration)
- *rParam* : `RobotParams` (link lengths, joint limits)
- *obs* : `Obstacles` (environment obstacles)

13.4.2 Assumptions

- All four parameters are either:
  1. successfully parsed from a user-supplied configuration file, or
  2. provided interactively through a GUI/CLI before the first call to `getAll`.
- Basic validity checks (dimension consistency, non-negative link lengths, joint limits with $\ell < u$, etc.) have already succeeded.

13.4.3 Access-Routine Semantics

`getAll()`

- **output:** the tuple (*start*, *goal*, *rParam*, *obs*).
- **exception:** `ParamErr`
  - if any of the four internal variables is *undefined*, or
  - if a post-validation step detects that the parameters are mutually inconsistent (e.g. mismatch between the length of *start* and the number of links in *rParam*).

13.4.4 Local Functions

`parseFile`(f) reads JSON/CSV input file f and initialises *start*, *goal*, *rParam*, *obs*.

`validate`(s, g, r, o) returns `true` iff the four arguments satisfy dimension and range constraints.

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.