

# 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

<b>1</b>	<b>Revision History</b>	<b>i</b>
<b>2</b>	<b>Symbols, Abbreviations and Acronyms</b>	<b>ii</b>
<b>3</b>	<b>Introduction</b>	<b>1</b>
<b>4</b>	<b>Notation</b>	<b>1</b>
<b>5</b>	<b>Module Decomposition</b>	<b>1</b>
<b>6</b>	<b>MIS of Path-Planning Module</b>	<b>3</b>
6.1	Module . . . . .	3
6.2	Uses . . . . .	3
6.3	Syntax . . . . .	3
6.3.1	Exported Types . . . . .	3
6.3.2	Exported Constants . . . . .	3
6.3.3	Exported Access Programs . . . . .	3
6.4	Semantics . . . . .	3
6.4.1	State Variables . . . . .	3
6.4.2	Assumptions . . . . .	4
6.4.3	Access Routine Semantics . . . . .	4
6.4.4	Local Functions . . . . .	4
<b>7</b>	<b>MIS of Collision-Detection Module</b>	<b>5</b>
7.1	Module . . . . .	5
7.2	Uses . . . . .	5
7.3	Syntax . . . . .	5
7.3.1	Exported Types . . . . .	5
7.3.2	Exported Access Programs . . . . .	5
7.4	Semantics . . . . .	5
7.4.1	State Variables . . . . .	5
7.4.2	Assumptions . . . . .	5
7.4.3	Access Routine Semantics . . . . .	5
7.4.4	Local Functions . . . . .	6
<b>8</b>	<b>MIS of Inverse-Kinematics Solver Module</b>	<b>7</b>
8.1	Module . . . . .	7
8.2	Uses . . . . .	7
8.3	Syntax . . . . .	7
8.3.1	Exported Access Programs . . . . .	7
8.4	Semantics . . . . .	7
8.4.1	State Variables . . . . .	7

8.4.2	Assumptions . . . . .	7
8.4.3	Access-Routine Semantics . . . . .	7
8.4.4	Local Functions . . . . .	7
<b>9</b>	<b>MIS of Output-Verification Module</b>	<b>8</b>
9.1	Module . . . . .	8
9.2	Uses . . . . .	8
9.3	Syntax . . . . .	8
9.3.1	Exported Constant . . . . .	8
9.3.2	Exported Access Programs . . . . .	8
9.4	Semantics . . . . .	8
9.4.1	State Variables . . . . .	8
9.4.2	Assumptions . . . . .	8
9.4.3	Access-Routine Semantics . . . . .	8
<b>10</b>	<b>MIS of Plotting Module</b>	<b>10</b>
10.1	Module . . . . .	10
10.2	Uses . . . . .	10
10.3	Syntax . . . . .	10
10.3.1	Exported Access Programs . . . . .	10
10.4	Semantics . . . . .	10
10.4.1	State Variables . . . . .	10
10.4.2	Environment Variables . . . . .	10
10.4.3	Assumptions . . . . .	10
10.4.4	Access-Routine Semantics . . . . .	10
<b>11</b>	<b>MIS of Control Module</b>	<b>12</b>
11.1	Module . . . . .	12
11.2	Uses . . . . .	12
11.3	Syntax . . . . .	12
11.3.1	Exported Access Programs . . . . .	12
11.4	Semantics . . . . .	12
11.4.1	State Variables . . . . .	12
11.4.2	Assumptions . . . . .	12
11.4.3	Access-Routine Semantics . . . . .	12
<b>12</b>	<b>MIS of Data-Types Module</b>	<b>13</b>
12.1	Module . . . . .	13
12.2	Uses . . . . .	13
12.3	Syntax . . . . .	13
12.3.1	Exported Types . . . . .	13
12.3.2	Exported Constants . . . . .	13
12.3.3	Exported Access Programs . . . . .	13

12.4	Semantics . . . . .	13
12.4.1	State Variables . . . . .	13
12.4.2	Assumptions . . . . .	14
12.4.3	Access-Routine Semantics . . . . .	14
12.4.4	Local Functions . . . . .	14

### 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  $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | \dots | c_n \Rightarrow r_n)$ .

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
<b>Hardware Hiding</b>	
<b>Behaviour Hiding</b>	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
<b>Software Decision</b>	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)

### 6.3 Syntax

#### 6.3.1 Exported Types

`ObstacleT`  $\equiv \langle c : \mathbb{R}^2, r : \mathbb{R} \rangle$

`Obstacles`  $\equiv$  sequence of `ObstacleT`

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

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

`Path`  $\equiv$  sequence of `Config`

#### 6.3.2 Exported Constants

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

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

#### 6.3.3 Exported Access Programs

Name	In	Out	Exceptions
<code>planPath</code>	<code>Config</code> $\times$ <code>Config</code> $\times$ <code>Path</code> <code>Obstacles</code> $\times$ <code>RobotParams</code>		<code>PathNotFound</code>

### 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  $360^\circ$ ).

### 6.4.3 Access Routine Semantics

`planPath`

- **transition:** Returns a collision-free *feasible*<sup>1</sup> path  $p = \langle q_0, \dots, q_m \rangle$  s.t.

$$\begin{aligned} q_0 &= start \\ q_m &= goal \\ \forall i \in [0, m-1] : &\neg \text{checkCollision}(q_i, obstacles, robotParams) \end{aligned}$$

- **output:**  $p$  on success.
- **exception:** `PathNotFound` raised if no feasible path exists within `MAX_ITER`.

### 6.4.4 Local Functions

`succ( $q$ )` returns the neighbouring configurations of  $q$  according to the lattice resolution.

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

$$\text{hCost}(q) = \text{HEURISTIC\_K} \|q - goal\|_2$$

---

<sup>1</sup>The module does not guarantee optimality; any valid path is acceptable.

## 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 $\times$ Obstacles $\times$ RobotParams	boolean	InvalidConfig
checkPathCollision	Path $\times$ Obstacles $\times$ RobotParams	boolean	InvalidPath

### 7.4 Semantics

#### 7.4.1 State Variables

None.

#### 7.4.2 Assumptions

- Each obstacle is a circle  $\langle c, r \rangle$  in  $\mathbb{R}^2$ .
- Links are straight segments between joint positions produced by planar forward kinematics.

#### 7.4.3 Access Routine Semantics

checkCollision

- **transition:** Let  $P = \langle p_0, \dots, p_n \rangle$  be the joint positions obtained from `forwardKinematics(config, robotParams)`. 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.

`checkPathCollision`

- **transition:** Returns `true` iff  $\exists q \in \text{path} : \text{checkCollision}(q, \text{obstacles}, \text{robotParams})$
- **output:** boolean as above.
- **exception:** `InvalidPath` if *path* is empty or contains an `InvalidConfig`.

#### 7.4.4 Local Functions

`forwardKinematics : ConfigtimesRobotParams  $\rightarrow$  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  $FK : \text{Config} \rightarrow \text{Point}$  is continuous and differentiable.
- The target point lies inside the reachable workspace.

#### 8.4.3 Access-Routine Semantics

solveIK

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

#### 8.4.4 Local Functions

jacobian( $q$ ) returns  $J(q) = \partial FK / \partial q$ .

## 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

`TOL_ERR` =  $10^{-6}$

#### 9.3.2 Exported Access Programs

Name	In	Out	Exceptions
<code>verifyPath</code>	<code>Path</code> $\times$ <code>Obstacles</code> $\times$ <code>RobotParams</code>	—	<code>PathInvalid</code> , <code>Collision</code>

### 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`

- **transition:**
  1. Check joint-limit and self-collision constraints for every  $q \in \text{path}$ .
  2. Call `checkPathCollision` 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

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



`plotMetrics`

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

(Local helper routines such as `drawTrajectory` are omitted because they are not exported and add no specification value.)

# 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.  $(init, goal, rParam, obs) = \mathbf{Input-Parameters::getAll}()$
  2.  $p_0 = \mathbf{Path-Planning::planPath}(init, goal, obs, rParam)$
  3.  $\mathbf{Collision-Detection::checkPathCollision}(p_0, obs, rParam)$
  4.  $p = \mathbf{Output-Verification::verifyPath}(p_0, obs, rParam)$
  5.  $\mathbf{Plotting::plotPath}(p)$
- **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

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

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

`Obstacles`  $\equiv$  sequence of `ObstacleT`

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

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

`Path`  $\equiv$  sequence of `Config`

`MetricTable`  $\equiv$  set of key-value pairs  $\langle \text{name} : \text{string}, \text{value} : \mathbb{R} \rangle$

#### 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.

## 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>.