Module Interface Specification for 2D-RAPP

Ziyang(Ryan) Fang

March 19, 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 $(-\infty, \infty)$
point	$\mathbb{R} \times \mathbb{R}$	ordered pair representing a 2D coordinate
configuration	$\mathbb{R}^n$	vector of joint angles, length n corresponds to the number of joints
path	sequence of configurations	ordered list of configurations representing a robot path
obstacle	(point, real)	tuple containing position (center) and radius defining a circular obstacle

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 Module	Simulation Interface Module
Behaviour-Hiding Module	Input Parameters Module Output Format Module Output Verification Module Path Planning Module Collision Detection Module Inverse Kinematics Solver Module Configuration Management Module
Software Decision Module	Plotting Module Logging and Debugging Module

Table 1: Module Hierarchy

6 MIS of Path Planning Module

6.1 Module

Path Plann

6.2 Uses

This module interacts with the following components:

- **Input Parameters Module** (for start and goal configurations, obstacles, and robot parameters)
- **Collision Avoidance Module** (to verify that planned paths are free of collisions)
- **Output Format Module** (to return and display the final planned path)
- **Data Structures Module** (for managing robot configurations, obstacles, and paths)

6.3 Syntax

6.3.1 Exported Constants

- `MAX_ITERATIONS` : $\mathbb{N}$ (Maximum iterations allowed 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
`load_params`	string	-	FileNotFound, InvalidFormat, MissingParameter, InvalidValue
`verify_params`	-	-	InvalidRobotDimensions, InvalidJointLimits, InvalidObstacleData, InvalidInitialPose, InvalidGoalPose
`getRobotParams`	-	RobotParams	-
`getObstacles`	-	Obstacles	-
`getInitialConfig`	-	$\mathbb{R}^n$	-
`getGoalConfig`	-	$\mathbb{R}^n$	-
`planPath`	$\mathbb{R}^n \times \mathbb{R}^n \times Obstacles \times RobotParams$	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

`load_params`(filePath: string):

- transition: Parameters (robot dimensions, joint limits, initial and goal configurations, obstacles) are loaded from the given file into the module's state variables.
- exception: Raises `FileNotFound` if the file does not exist; raises `InvalidFormat`, `MissingParameter`, or `InvalidValue` if any input is invalid.

`verify_params`():

- output: none
- exception: Raises `InvalidRobotDimensions`, `InvalidJointLimits`, `InvalidObstacleData`, `InvalidInitialPose`, or `InvalidGoalPose` if the loaded parameters violate constraints or logical consistency.

`getRobotParams`():

- output: $RobotParams$, a tuple containing robot dimension parameters and joint constraints.
- exception: none

`getObstacles`():

- output: A sequence of $Obstacles$, where each obstacle is represented as a tuple $(\mathbb{R}^2, \mathbb{R})$, containing the center coordinates and radius.
- exception: none

`getInitialConfig`():

- output: Initial robot configuration as a vector in $\mathbb{R}^n$.
- exception: none

`getGoalConfig`():

- output: Goal robot configuration as a vector in $\mathbb{R}^n$.
- exception: none

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

- output: A collision-free sequence of robot configurations (path) from the initial configuration ($start$) to the goal configuration ($goal$), represented as a sequence of $\mathbb{R}^n$.
- exception: Raises `NoPathFoundException` if no collision-free path exists between the provided configurations given the environment and robot constraints.

6.4.5 Local Functions

The following local functions are defined for specifying internal logic clearly. These functions are primarily for specification purposes; they may not necessarily correspond directly to explicit implementations.

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

- output: boolean, true if the given robot configuration satisfies the joint angle constraints, false otherwise.

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

- output: boolean, true if the given configuration does not result in collisions with obstacles, false otherwise.

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

- output: real, heuristic cost (estimated distance or effort) between two configurations for guiding the A* algorithm.

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

- output: real, actual distance metric used to compute the edge costs between configurations during path planning.

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)

7.3 Syntax

7.3.1 Exported Constants

None.

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
`checkCollision`	$\mathbb{R}^n \times \mathit{Obstacles} \times \mathit{RobotParams}$	boolean	InvalidConfiguration
`checkPathCollision`	sequence of $\mathbb{R}^n \times$ Obstacles $\times$ RobotParams	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):

- output: Returns `true` if the provided robot configuration collides with any obstacle; otherwise, returns `false`.
- exception: Raises `InvalidConfiguration` if the configuration provided violates joint constraints or robot parameters.

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

- 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 is improperly defined.

7.4.5 Local Functions

The following functions are local specification tools that clarify the module's collision-checking logic:
`detectSegmentCollision`(pointA: $\mathbb{R}^2$, pointB: $\mathbb{R}^2$, obstacle: ($\mathbb{R}^2$, $\mathbb{R}$)):

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

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

- output: sequence of $\mathbb{R}^2$, representing the positions of all joints and the end-effector, used for collision checking.

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.

8 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)