Final work

Planning and Implementation of Robotic Systems

Final Report

Authors: Andrés Alberto Castro Peñaranda

Iván Barrachina Sabariego

Professors: Jan Rosell

Cristina Lampón

Date: December 24, 2024

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
Industrial de Barcelona

Contents

List of Figures

1 Abstract

This project consists on the simulation of a robotic bartender scenario in Kautham where the MADAR mobile manipulator prepares and serves drinks in a bar environment. The setup combines a bar scene modeled in Blender with the MADAR robot, defined using a URDF file. The robot navigates the bar, picks up bottles, prepares drinks, and delivers them to designated areas. A Task and Motion Planning (TAMP) framework coordinates high-level task planning using PDDL and low-level motion planning for robot navigation and manipulation.

2 Introduction

2.1 Project Conceptualization

The primary objective of this project is to simulate a robotic bartender scenario, where a mobile manipulator named *MADAR*, which is shown below in Figure 1 is tasked with preparing and serving drinks in a bar environment. This work brings together two key elements: the bar scenario itself, which was modeled using the *Blender* software, and the MADAR robot, whose structure and capabilities were defined in a *URDF* (Unified Robot Description Format) file. The combination of these elements forms a rich testbed for exploring task and motion planning (TAMP) solutions.

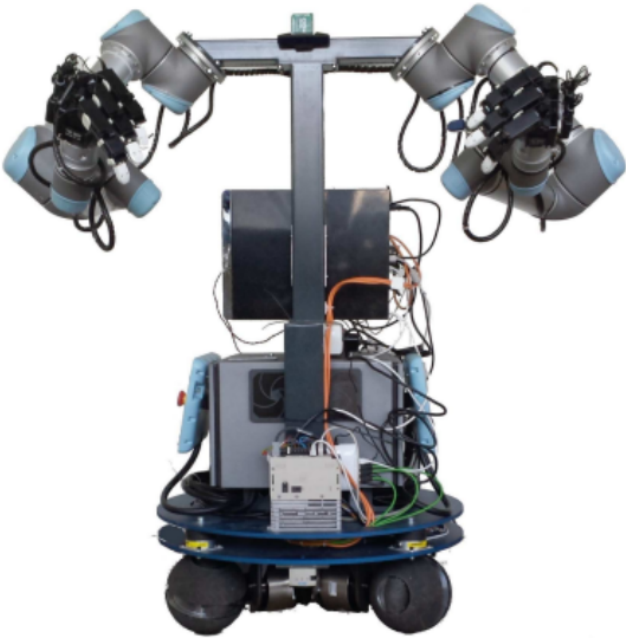


Figure 1: Picture of the real MADAR robot

From a conceptual standpoint, the robot must:

- Navigate the bar environment without collisions.
- Approach designated sections where glasses and bottles are stored.
- Pick up bottles, prepare drinks, and place them at target sections or customer tables.

The interplay between high-level task planning and low-level motion planning serves as a cornerstone of this final project. High-level *PDDL* (Planning Domain Definition Language) descriptions manage the sequence of tasks (like *move*, *pick*, and *place*), while low-level motion planners (in *Kautham*, for instance) generate feasible robot trajectories to execute each step.

2.2 Framework in Task and Motion Planning

The TAMP framework integrates a classical planner for generating a sequence of symbolic actions and a motion planner to ensure each action is geometrically feasible. In the bar scenario:

- The **task planning** layer is responsible for deciding *what* the robot should do next: e.g., "move to shelf," "pick up bottle," "prepare a drink," and "deliver the drink" to a target table.
- The **motion planning** layer computes *how* the robot performs each action: e.g., planning safe navigation paths around obstacles or generating inverse kinematics-based trajectories for picking a glass.

This layered approach allows a modular and scalable solution, where each symbolic action is independently grounded in a feasible motion plan. Consequently, we benefit from:

1. A clear **domain definition** in *PDDL*, describing relevant predicates (e.g., `(at robot location)`, `(holding hand object)`) and actions (e.g., `move`, `pick`, `place`).
2. A **tampconfig** file (or an equivalent domain file) that bridges the symbolic actions to low-level controls and robot joint configurations.
3. A set of **motion planners** (e.g., *OMPL* or *Kautham* RRT-based approaches) to ensure continuous feasibility of each action.

2.3 Blender Bar Scene and MADAR Robot URDF

Creating the bar environment in *Blender* was the first step in establishing a realistic simulation. The bar scene consists of:

- A shelf where multiple beverage bottles and glasses are stored.
- Serving tables or zones where the robot must place or pick up items.
- Enough free space for the MADAR robot to navigate.

The 3D models were exported from *Blender* to a suitable format that can be visualized in *Kautham* or integrated into planning tools.

Parallel to constructing the bar, we developed the **MADAR robot's URDF**. This description encapsulates:

- The robot's kinematic structure, including links for the base, arms, and end-effectors.
- Joint definitions specifying their limits, degrees of freedom, and hierarchy.
- Appropriate collision and visual meshes to ensure accurate simulation.

A repository for MADAR's URDF and Gazebo implementation was found at:

> https://gitioc.upc.edu/robots/madar_description/-/tree/gazebo_implementation?ref_type=heads

This resource helped accelerate the setup by providing a base structure upon which further modifications could be made for the bar scenario.

With a complete bar environment and a properly defined mobile manipulator, the project transitions to defining the domain and problem in *PDDL*, along with specialized motion planning strategies. The ensuing sections describe how these layers integrate and highlight our methodology, results, and conclusions.

3 Implementation and methodology

This section describes the practical steps taken to build the bar scenario, configure the MADAR robot, and integrate all components into a cohesive task and motion planning (TAMP) pipeline. The work involved three major strands:

1. Modeling the bar environment and integrating it into simulation.
2. Defining the MADAR robot in URDF, providing controllers, and generating feasible robot movements.
3. Constructing a TAMP framework that uses a PDDL domain for symbolic actions and a `tampconfig` setup for low-level motion planning queries.

3.1 Bar Environment and Scenario Definition

Scenario URDF. While the bar environment was primarily created in *Blender* (see Figure 2), it was also necessary to export and refine a simplified `URDF` (Simulation Description Format) representation for collision checking and simulation. The scenario URDF contains:

- Several **collision geometries**, typically simple boxes or cylinders, to approximate tables, shelves, and walls.
- Basic **visual meshes** to preserve some realism while maintaining real-time simulation performance.
- A **reference coordinate system** placed near the bar's center, ensuring consistent transforms for the robot's base and any placed objects.

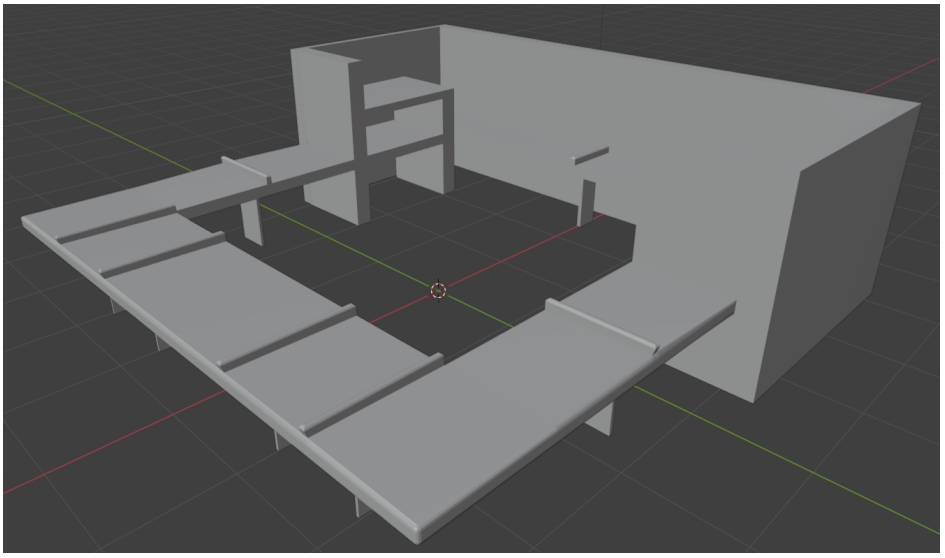


Figure 2: Picture of the real MADAR robot

In combination with the environment's URDF, we also assigned **semantic labels** to key areas (e.g., *Shelf*, *Prep-Table*, *Table1*, etc.), which are referenced in the PDDL problem file.

3.2 MADAR Robot URDF and Controllers

MADAR URDF. The MADAR robot's geometry and kinematic structure are defined in a `URDF` (Unified Robot Description Format). Inspired by the repository linked in the introduction, we customized:

- **Base links** and **chassis joints** for omnidirectional driving.
- Two **manipulator arms** with 6 *DoFs* each, enabling pick-and-place tasks around the bar.
- **End-effector** descriptions for both arms, suitable for grasping glasses and bottles.

All the mentioned customizations were implemented on a single URDF file, instead of using macros like in the provided link because kautham is only able to load URDF files.

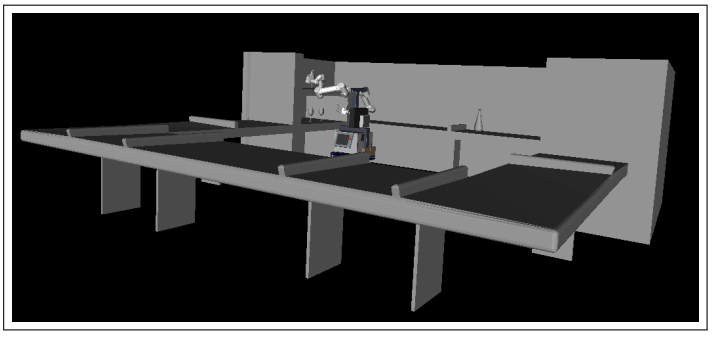


Figure 3: MADAR robot URDF, loaded in kautham gui.

Controllers and Joints. In parallel with the URDF definition, a set of **joint controllers** (written as XML control sets) were created, mapping each joint to high-level commands. These controllers make it possible to:

- **Move the base** by just being able to control (x, y, yaw).
- **Position the arms** by setting *DoFs* in the (x, y, yaw) that the platform should have in order to be at a location and just being able to control the joints of a UR arm.
- **Open and close grippers** setting the *DoFs* in the (x, y, yaw) that the platform should have in order to be at a location,and the *DoFs* of the arms to be in the section of the location and just being able to control the joints of a hand arm.

We developed specialized controls for:

Move Robot A *mobile base* control set with 3 *DoFs* (x, y, yaw).

Arm Approach and Retract Control sets that define suitable *joint angles* for picking or retracting, used in the `prepare drink` , `aproach arm` and `retractarm` actions.

Hand Pick and Place Another set of control parameters specific to finger closures, used in the `pick`, `prepare drink` and `place` actions.

3.3 PDDL Actions and TAMP Pipeline

Overview. A *PDDL domain* was constructed to specify the symbolic side of the task. This domain includes actions such as:

- `move`: Move the robot's base between zones (home, shelf, etc.).
- `approach_section`: Move an arm from a retracted position to a section where objects or glasses reside.
- `pick`: Grasp an object (bottle or glass) using a free hand.
- `place`: Release an object at a desired section.
- `retractarm`: Move the manipulator back to a safe position near the robot's body.
- `prepare_drink`: A higher-level compound action, leveraging sub-actions to pick up a bottle, pour into a glass, and place the bottle back.

Each of these symbolic actions is mapped onto a set of low-level *joint configurations* in our TAMP configuration (`tampconfig`) file. For instance, `pick` actions reference "grasp controls," `place` actions reference "release controls," and `move` uses a "transit" or "transfer" path that Kautham's motion planner validates.

Implementation Steps.

1. **Domain File.** We enumerated *predicates* (e.g., `(at robot location)`, `(holding hand object)`) and *actions* with preconditions/effects.
2. **Problem File.** We defined the initial state (robot at home, glasses on shelf) and goals (e.g., deliver a glass to a table).

3. **tampconfig File.** For each PDDL action, we specified the relevant *controls* (joint angles for arms, base positions, etc.) so that the geometric planner could ground the symbolic steps.

Actions Implementation and Scripts. We prepared **Python scripts** for each action, consistent with the domain:

- **MOVE.py**: Moves the robot base if arms are retracted.
- **PICK.py**: Transits the manipulator to the desired object, grasps it, and starts a `<Transfer>` block for the path logging.
- **PLACE.py**: Concludes the `<Transfer>` block, placing the object in a new section.
- **APPROACHSECTION.py**: Moves the robot arms from retracted configuration to a configuration that aproaches the arm to a section in the current location.
- **RETRACTARM.py**: Moves the robot arm from configuration that aproaches the arm to a section in the current location to retracted configuration at home of the robot.
- **PREPAREDRINK.py**: Combines picking the bottle, moving to the glass, pouring (implicitly), and returning the bottle. This script is invoked by the `prepare_drink` action in the domain.

Each script leverages **Kautham's C++/Python API** to set queries, generate paths, and write them in `.xml` files using specialized `<Transit>` or `<Transfer>` tags, consistent with the project's logging format.

4 Motion planning integration

In this section, we describe how the motion planning layer complements and grounds the high-level symbolic actions from the TAMP approach. We begin with an overview of the motion planning workflow, followed by details of the specific queries for mobile navigation and grasping, and conclude with a benchmarking study of planning performance.

4.1 Motion planning workflow

Motion planning plays a pivotal role in ensuring each high-level PDDL action is geometrically feasible. In our scenario, we rely on the *Kautham Project* (or any other chosen planner) to generate collision-free paths. The integration steps can be summarized as follows:

1. **Action invocation**: When a symbolic action (e.g., `move`, `pick`, or `place`) is selected by the TAMP solver, we consult the `tampconfig` file to retrieve joint controls and relevant initial/goal states.
2. **Planner setup**: The motion planning script (e.g., `MOVE.py`, `PICK.py`) configures the robot controls in Kautham, sets the collision environment, and defines the query (init → goal).
3. **Path generation**: The planner uses sampling-based algorithms (e.g., *RRTConnect*, *PRM*, or *Lazy-RRT*) to find a path in configuration space that avoids collisions with the bar environment, the MADAR robot's own geometry, and any held objects if in `transfer` mode.

4. **Path logging and execution**: Once a valid path is found, it is written into a `<Transit>` or `<Transfer>` block in our task file, and the simulation environment (Gazebo or similar) is updated with the new joint states.

4.2 Motion planning Queries

Since we handle both **mobile navigation** (driving the robot base around the bar) and **manipulation** (moving arms to pick or place objects), we created two primary query types:

4.2.1 Mobile Domain Queries

For actions like `move` the robot must navigate from one location to another while keeping arms retracted. Our **mobile domain queries** revolve around:

- A *2D or 3D* base pose space, depending on whether we consider (x, y, yaw) or a full 6D pose.
- Collision constraints that treat the bar's floor plan, tables, and shelves as static obstacles.
- Possibly dynamic constraints (if humans or other objects move around) but not included in our base scenario.

Typical examples:

```
Query 1: from (x=0.5,y=0.5,theta=0.0) to (x=0.2,y=0.8,theta=1.57)
Query 2: from (home) to (shelf)
```

Kautham's sampling-based approach finds a collision-free path. If found, we record it as a *Transit* path. If the robot is holding an object, the domain changes to a *Transfer* path, though for base motion alone we still handle straightforward collisions.

4.2.2 Grasping Domain Queries

Actions like `pick`, `place`, or `prepare_drink` require arm motion planning:

- **Approach section** queries: In which the manipulator's end-effector moves from a retracted position to a target section (SHELF_1, PREP-SECTION1, etc.), ensuring no collisions with the bar structure or the robot's own body.
- **Grasp configuration** queries: In which the end-effector or *hand* transitions from an open state to a closed state around an object, verifying that the final joint angles do not collide with surrounding items.

Whenever an object is attached, the environment updates to reflect that object as part of the robot's geometry, ensuring the next motion plan accounts for new collision shapes.

4.3 Benchmarking results

To evaluate the performance of our motion planning integration, we ran several tests focusing on planning time, path length, and success rate under two typical scenarios: the *Move* and *Approach_section* actions,

which correspond to the most relevant mobile and grasping domain actions.

This required creating benchmarking `.xml` files that called the different planner definition files, executed the simulations and created resulting log files that could be turned into database files, which were later processed by external applications like the online `plannerarena.org` to extract statistical results from the output of the simulations.

4.3.1 Mobile domain benchmarking results

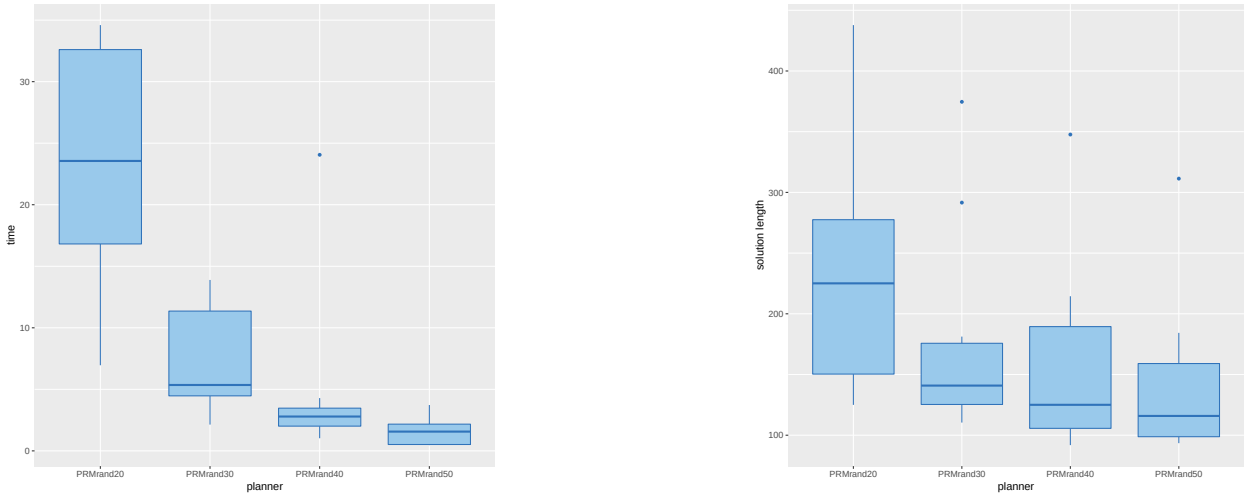
First, we performed 10 trials of `move` actions among *HOME* and the *(0.3, 0.75, 0.5)* point for each of the candidate planners (PRM with four different sampling sources, RRTconnect and LazyRRT) with different tuneable parameters, with all the trials being limited to a maximum computation time of 120 seconds. For the PRMs, four iterations of the distance threshold parameter, which determines the maximum distance between two samples to be connected, will be tested. The tested values for each planners, along standard, static values for the others, are determined using the Kautham GUI interface to find the working range for the parameter that is different for each planner. For the RRTs, the Range, that determines the maximum length of the tree edges, parameter will be tested.

The results for each planner are shown next:

- **PRM (random)** (found a solution 100% of the times)

Parameter	**Assigned value**
BounceDistanceThreshold	5
BounceSteps	10
Distance threshold	20-30-40-50
MaxNearestNeighbors	5
MinExpandTime	5
MinGrowTime	5
SimplifySolution	0

Table 1: Simulation parameters for the PRMrandom benchmarking



(a) Planning time.

(b) Solution length.

Figure 4: Results for the PRM planner with Random sampling.

The distance threshold is set to 50, as it offers the shortest solution with the shortest planning time. This solution yields a result as follows:

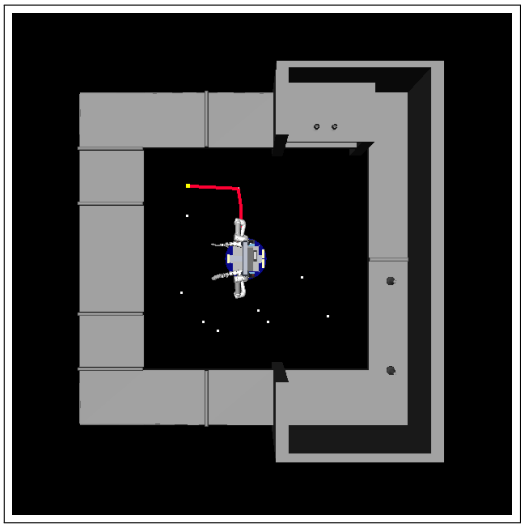
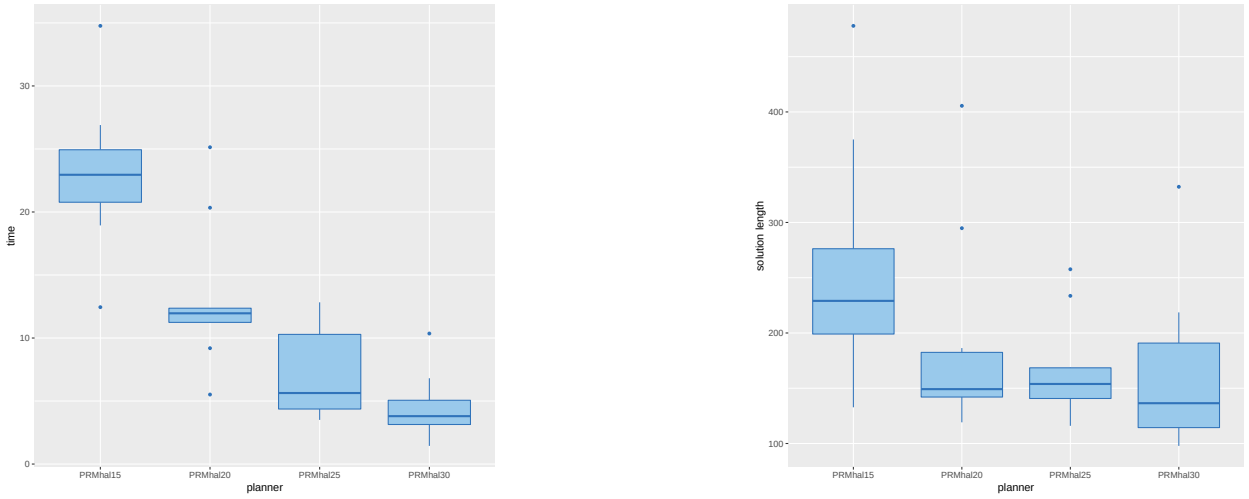


Figure 5: Solution path for the PRMrandom planner for a distance threshold of 50.

- **PRM (Halton)** (found a solution 100% of the times)

Parameter	Assigned value
BounceDistanceThreshold	5
BounceSteps	20
Distance threshold	15-20-25-30
MaxNearestNeighbors	10
MinExpandTime	5
MinGrowTime	5
SimplifySolution	0

Table 2: Simulation parameters for the PRMhalton benchmarking



(a) Planning time.

(b) Solution length.

Figure 6: Results for the PRM planner with Halton sampling.

The distance threshold is set to 30, as it offers the shortest solution with the shortest planning time. This solution yields a result as follows:

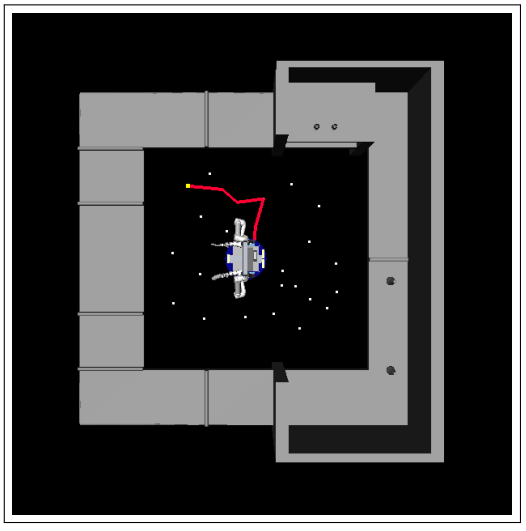
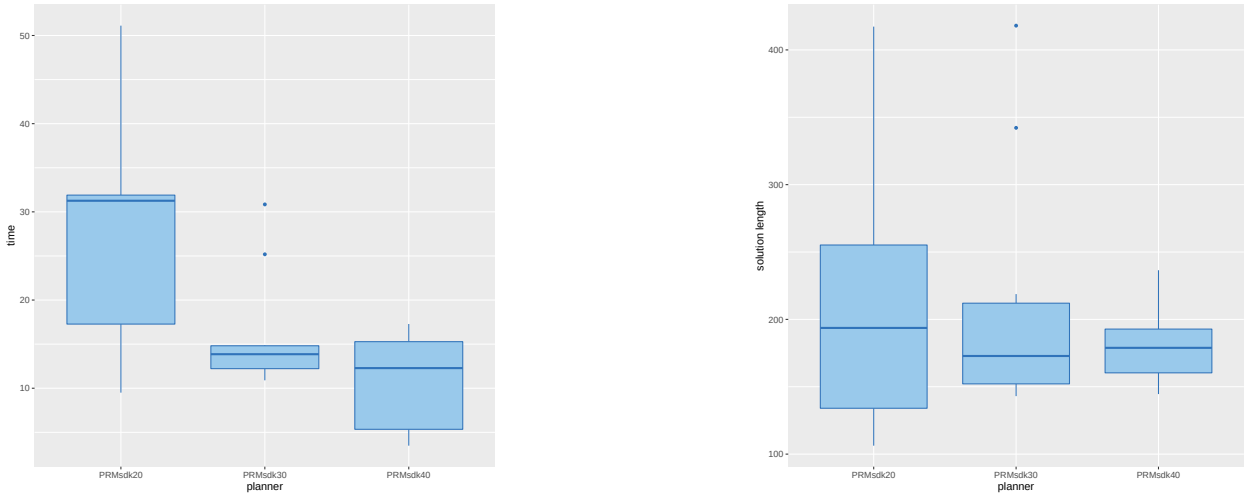


Figure 7: Solution path for the PRMhalton planner for a distance threshold of 30.

- **PRM (SDK)** (found a solution 100% of the times)

Parameter	**Assigned value**
BounceDistanceThreshold	5
BounceSteps	10
Distance threshold	20-30-40
MaxNearestNeighbors	5
MinExpandTime	5
MinGrowTime	5
SimplifySolution	0

Table 3: Simulation parameters for the PRMsdk benchmarking



(a) Planning time.

(b) Solution length.

Figure 8: Results for the PRM planner with SDK sampling.

The distance threshold is set to 30, as it offers the best solution length to planning time relation. This solution yields a result as follows:

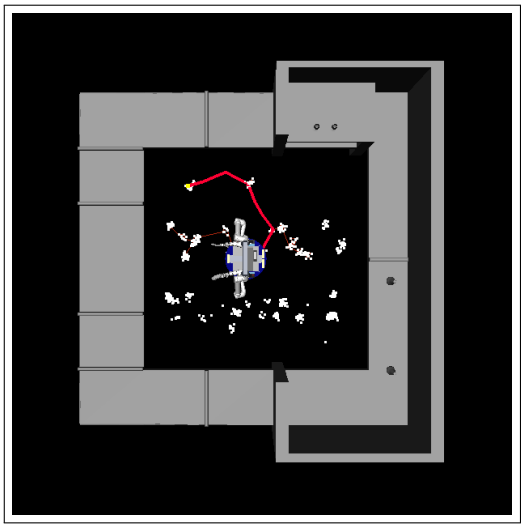
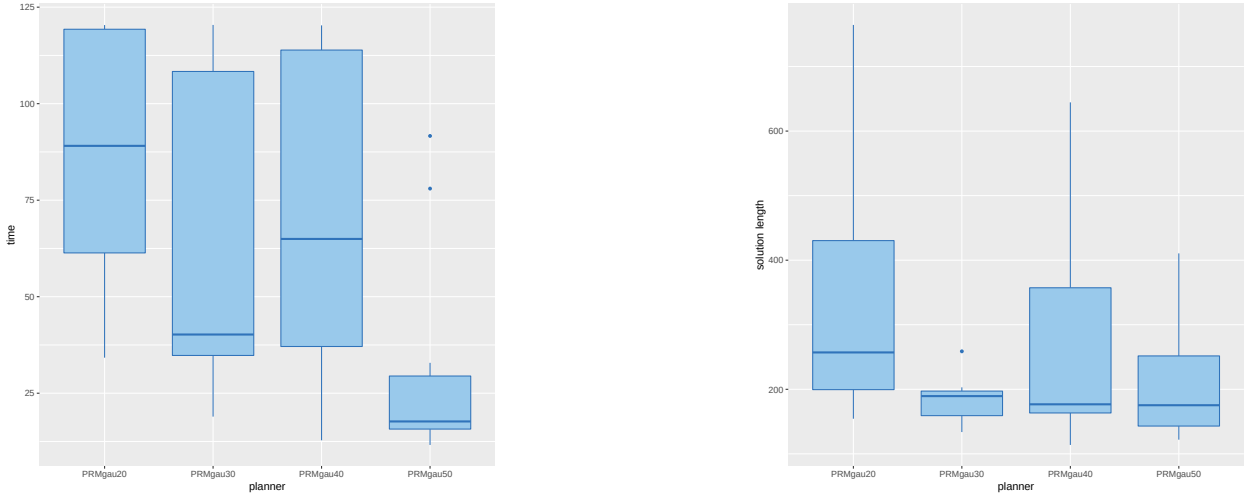


Figure 9: Solution path for the PRMsdk planner for a distance threshold of 30.

- **PRM (Gaussian)** (did **NOT** find a solution 100% of the times)

Parameter	**Assigned value**
BounceDistanceThreshold	20
BounceSteps	20
Distance threshold	20-30-40-50
MaxNearestNeighbors	20
MinExpandTime	10
MinGrowTime	10
SimplifySolution	0

Table 4: Simulation parameters for the PRMgauss benchmarking



(a) Planning time.

(b) Solution length.

Figure 10: Results for the PRM planner with Gaussian sampling.

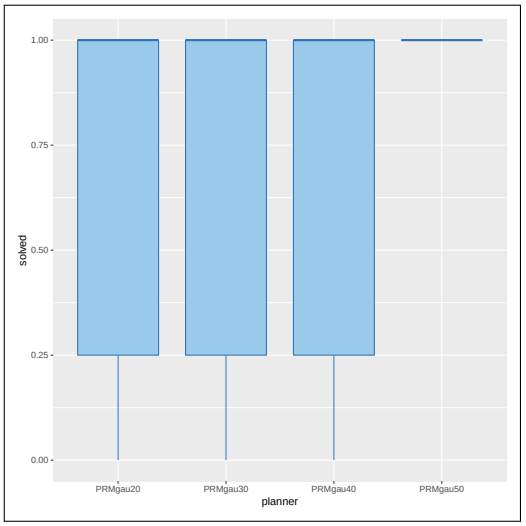


Figure 11: Success rate for the PRM planner with Gaussian sampling.

The distance threshold is set to 50, as it offers one of the best solution length to planning time relations, while being the only one that makes it possible to find a solution within the stated time goal. This solution yields a result as follows:

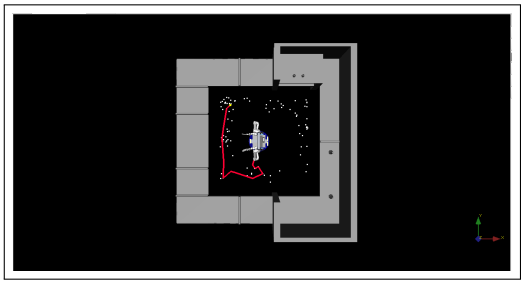
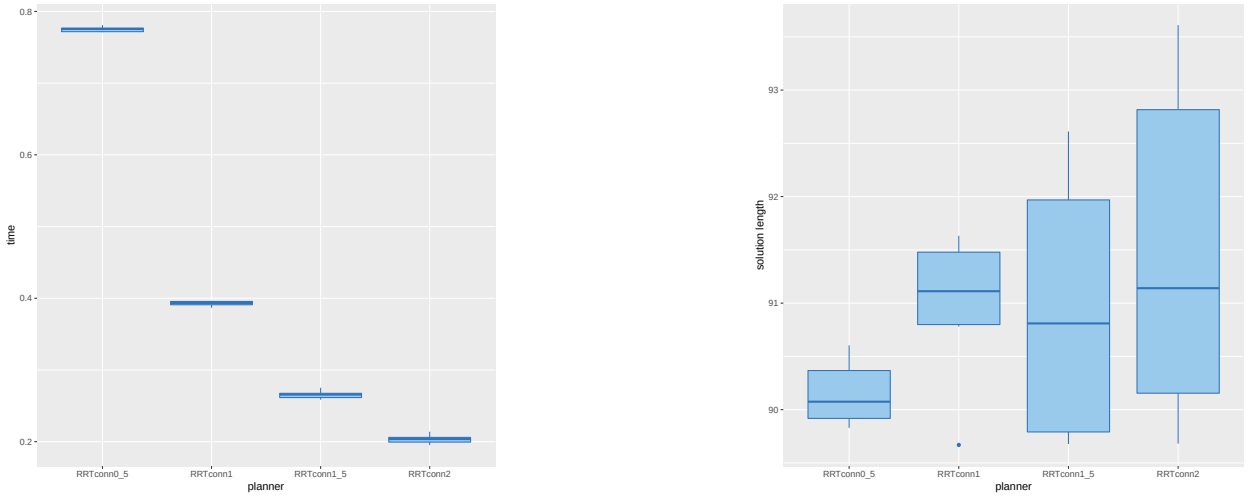


Figure 12: Solution path for the PRMgauss planner for a distance threshold of 50.

- **RRTconnect**

Parameter	**Assigned value**
Range	0.5-1-1.5-2
SimplifySolution	0

Table 5: Simulation parameters for the RRTconnect benchmarking



(a) Planning time.

(b) Solution length.

Figure 13: Results for the RRTconnect planner.

The Range parameter is set to 1, as it offers the best solution length to planning time relation. This solution yields a result as follows:

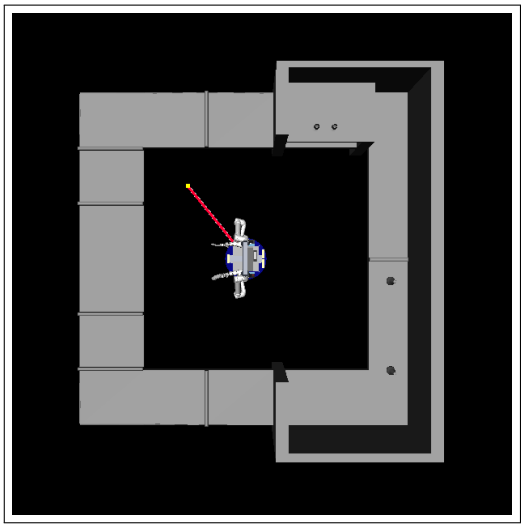
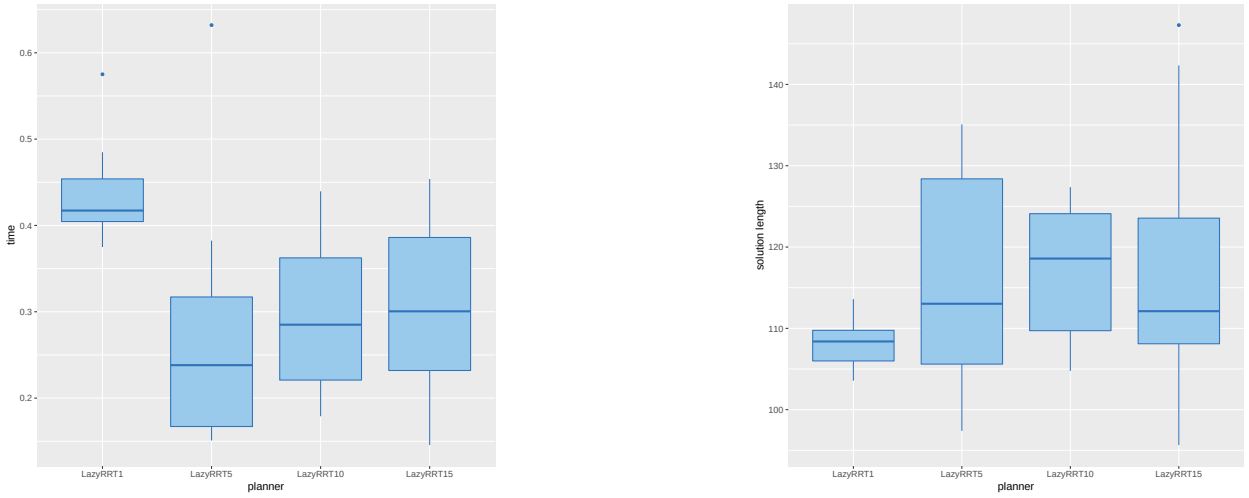


Figure 14: Solution path for the RRTconnect planner for a Range of 1.

- **LazyRRT**

Parameter	**Assigned value**
Range	1-5-10-15
SimplifySolution	0

Table 6: Simulation parameters for the LazyRRT benchmarking



(a) Planning time.

(b) Solution length.

Figure 15: Results for the LazyRRT planner.

The Range parameter is set to 5, as it offers the best solution length to planning time relation. This solution yields a result as follows:

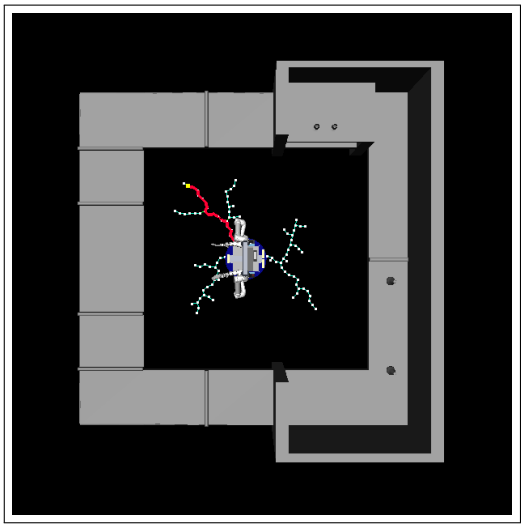
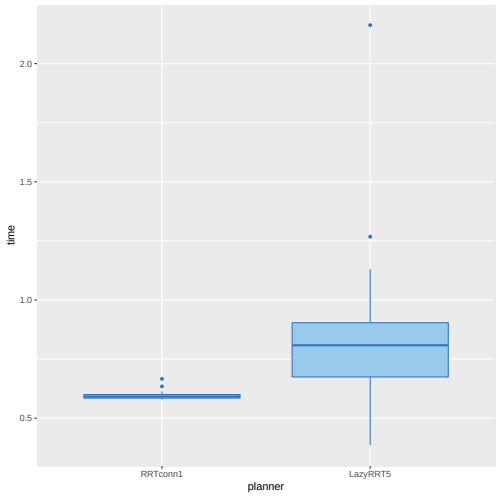
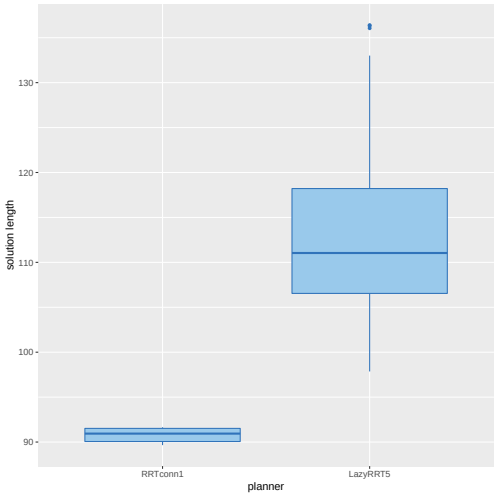


Figure 16: Solution path for the LazyRRT planner for a Range of 5.

Since the PRM planners take a significative greater amount of time to plan a solution on the open scenario of the bar, they will be discarded for the rest of the benchmarking, focusing on the two RRT planners that are more adequate for this problem. A second benchmarking, this time with 30 trials per planner, has been conducted in order to test the previously stated optimal parameters of both planners against each other. The following results have been obtained:



(a) Planning time.



(b) Solution length.

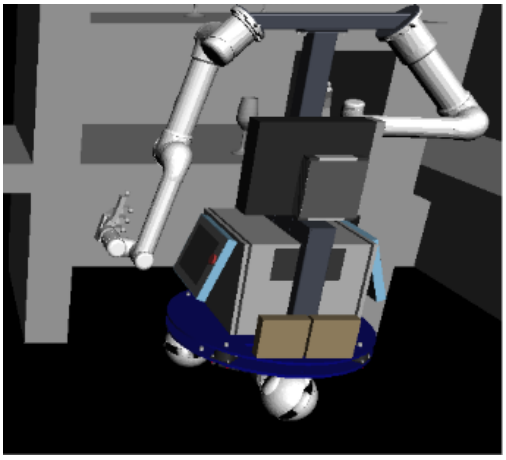
Figure 17: Results for the crossed RRT planner comparison for the Move action.

It can be concluded that, for the Mobile Domain, the RRTconnect planner is the fastest and the one that provides simpler solutions for the robot.

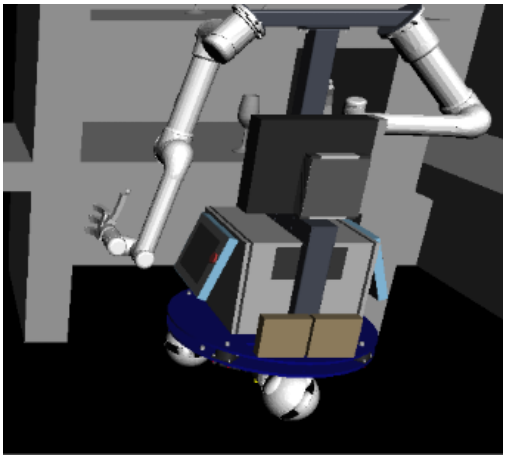
4.3.2 Grasping Domain benchmarking results

For the `Approach_section` action we tested the previous RRT planners in a position in which the end-effector of the left robotic arm approaches from a retracted position to a target section. We measured:

- **Reachability Success Rate**: Cases where the final grasp pose was reachable without collisions.
- **Planning Time for Arm Motions**: Time to find a collision-free approach path.
- **Length of the Solution**



(a) Retracted position.



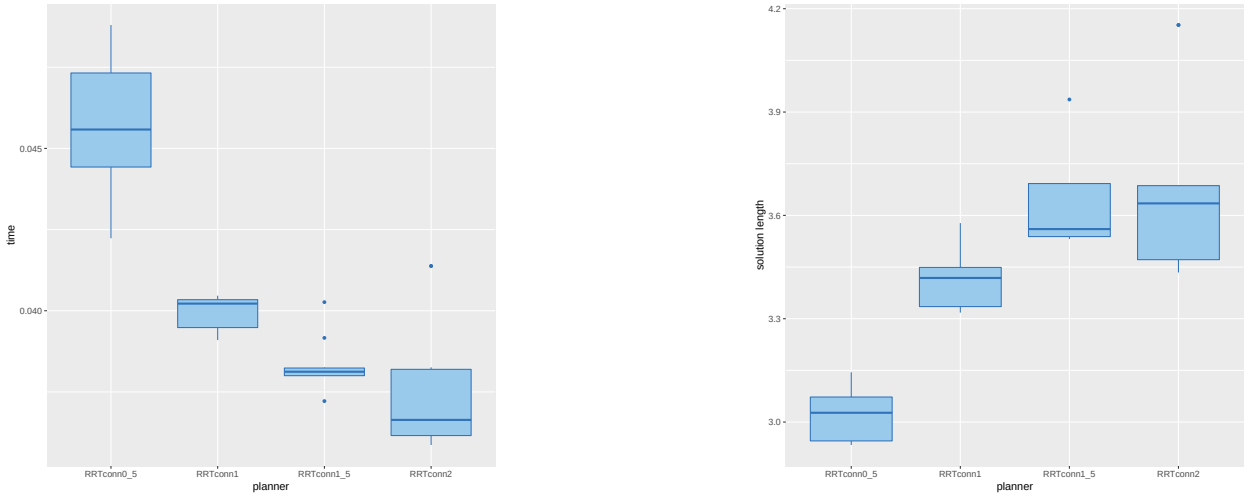
(b) Target position.

Figure 18: Initial and goal position of an `Approach_section` action.

The benchmarking, as previously done for the `Move` action, consisted on a 10 iteration test for each planner, in which the four initial planner parameters were checked in order to observe possible changes in the optimal

configuration of the planners, followed by a 30 iteration experiment in which the performance of the optimal configurations were compared. The following results were obtained:

- **RRTconnect planner:**



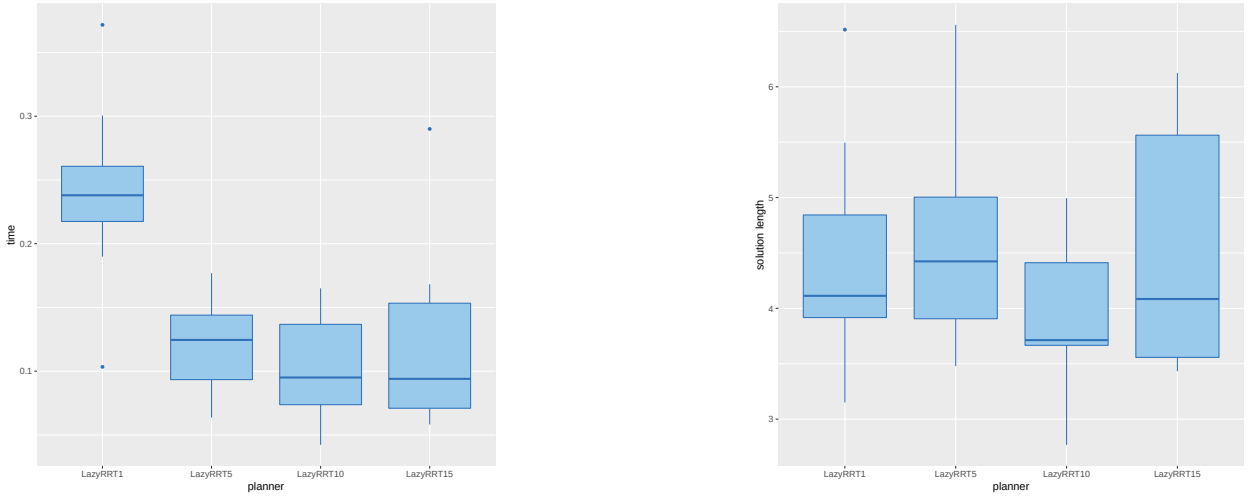
(a) Planning time.

(b) Solution length.

Figure 19: Results for the RRTconnect planner.

We checked that the same results previously obtained apply for this action.

- **LazyRRT planner:**



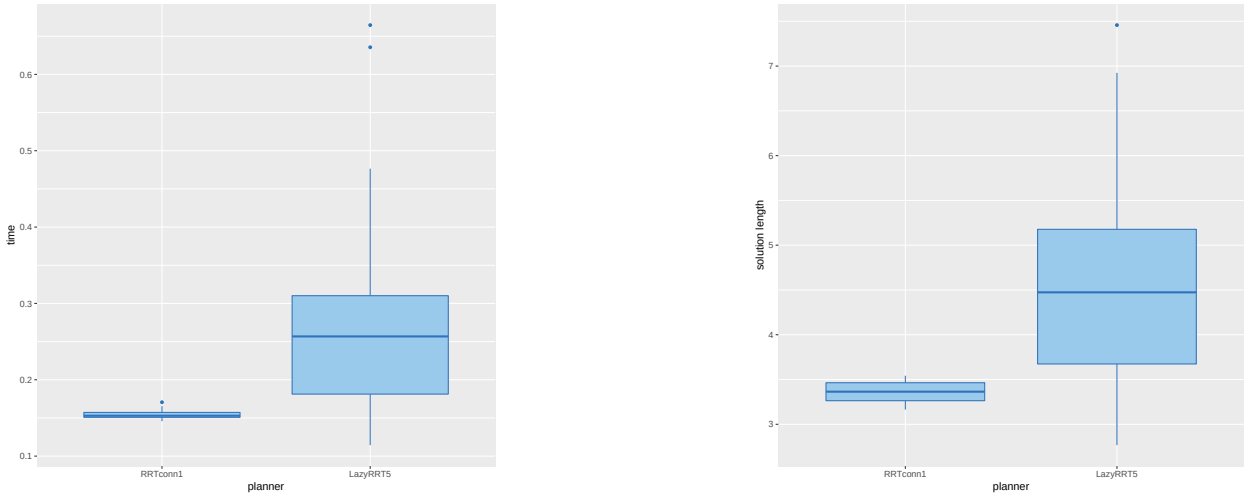
(a) Planning time.

(b) Solution length.

Figure 20: Results for the LazyRRT planner.

We checked that the same results previously obtained apply for this action.

As done for the previous action, the optimal planner configurations are compared, which yields the following result:



(a) Planning time.

(b) Solution length.

Figure 21: Results for the crossed RRT planner comparison for the Approach Section action.

4.4 Motion planning conclusions

Finally, we concluded that all the motion planning had to be performed using a RRTconnect planner with the following configuration:

Parameter	Assigned value
Range	1
SimplifySolution	0

Table 7: Parameter selection for the selected motion planner.

5 Task Planning Integration

5.1 pddl files structure

We organize our PDDL solution into two primary files:

- **Domain file** (`robot_bartender.pddl`): Defines the *types*, *predicates*, and *actions* needed to capture the robot's possible tasks (e.g., move, pick, place, prepare_drink).
- **Problem file** (`deliver-drink.pddl`): Declares the *objects* in the scenario, the *initial states* (e.g., where the robot and glasses are), and the *goal conditions* (e.g., a filled glass at a table).

5.1.1 pddl domain file

The domain file, called `robot_bartender.pddl`, enumerates key **predicates** and **actions** for a bartender scenario with a mobile manipulator. Table 8 recaps important predicates, while Table 9 outlines the primary actions (including `move`, `approach_section`, `retractarm`, `pick`, `place`, and `prepare_drink`).

Predicate	Arguments	Interpretation
`(at ?r - robot ?l - location)`	`?r, ?l`	Robot `?r` is at location `?l`.
`(holding ?h - hand ?o - object)`	`?h, ?o`	Hand `?h` is holding object `?o`.
`(obj-at ?o - object ?s - section)`	`?o, ?s`	Object `?o` is on section `?s`.
`(hand-at ?h - hand ?s - section)`	`?h, ?s`	Hand `?h` is positioned at section `?s`.
`(section-of ?s - section ?l - location)`	`?s, ?l`	Section `?s` belongs to location `?l`.
`(free-hand ?h - hand)`	`?h`	`?h` is free (not holding anything).
`(connected ?l1 - location ?l2 - location)`	`?l1, ?l2`	`?l1` is connected to `?l2`.
`(arm-retracted ?h - hand)`	`?h`	The manipulator arm for `?h` is retracted.
`(glass-empty ?o - object)`	`?o`	Glass `?o` is empty.
`(glass-filled ?o - object)`	`?o`	Glass `?o` is filled (drink ready).

Table 8: Key domain predicates for the bartender scenario.

Action	Preconditions	Effects
`move(?r, ?from, ?to)`	• `(at ?r ?from)` • `(connected ?from ?to)` • `(arm-retracted LEFT-HAND)` • `(arm-retracted RIGHT-HAND)`	• `(not (at ?r ?from))` • `(at ?r ?to)`
`approach_section(?r, ?h, ?l, ?s)`	• `(at ?r ?l)` • `(section-of ?s ?l)` • `(arm-retracted LEFT-HAND)` • `(arm-retracted RIGHT-HAND)`	• `(hand-at ?h ?s)` • `(not (arm-retracted ?h)`
`retractarm(?r, ?h, ?l, ?s)`	• `(at ?r ?l)` • `(hand-at ?h ?s)`	• `(not (hand-at ?h ?s))` • `(arm-retracted ?h)`
`pick(?r, ?h, ?o, ?s)`	• `(free-hand ?h)` • `(hand-at ?h ?s)` • `(obj-at ?o ?s)`	• `(not (free-hand ?h))` • `(holding ?h ?o)` • `(not (obj-at ?o ?s))`
`place(?r, ?h, ?o, ?s)`	• `(holding ?h ?o)` • `(hand-at ?h ?s)`	• `(free-hand ?h)` • `(obj-at ?o ?s)` • `(not (holding ?h ?o))`
`prepare_drink(?r, ?o, ?s)`	• `(at ?r PREPARATION-TABLE)` • `(obj-at ?o ?s)` • `(glass-empty ?o)` • `(free-hand LEFT-HAND)` • `(free-hand RIGHT-HAND)` • `(arm-retracted LEFT-HAND)` • `(arm-retracted RIGHT-HAND)` • `(section-of ?s PREPARATION-TABLE)`	• `(not (glass-empty ?o))` • `(glass-filled ?o)`

Table 9: Domain actions (`move`, `approach_section`, `retractarm`, `pick`, `place`, and `prepare_drink`).

5.1.2 pddl problem file

The `deliver-drink.pddl` file introduces the **objects** (robot, hands, locations, sections, glasses, etc.), **initial conditions** (robot at HOME with arms retracted, glasses on shelves, etc.), and **goal** (e.g., a filled glass at a specific table). For instance:

```
(define (problem deliver-drink)
```

```
  (:domain robot_bartender)
  (:objects
    MADAR - robot
    HOME SHELF PREPARATION-TABLE TABLE1 ...
    LEFT-HAND RIGHT-HAND - hand
    ...
    glass1 glass2 glass3 glass4 - objectPlanning
  )
  (:init
    (at MADAR HOME)
    (free-hand LEFT-HAND)
    (free-hand RIGHT-HAND)
    (arm-retracted LEFT-HAND)
    (arm-retracted RIGHT-HAND)
    ...
    (obj-at glass1 SHELF_1)
    (glass-empty glass1)
    ...
  )
  (:goal
    (and
      (obj-at glass1 table2-left)
      (glass-filled glass1)
    )
  )
)
```

These definitions ensure the symbolic solver knows which steps might be feasible (e.g., you can `pick glass1` from `SHELF_1` because it is at that location).

5.2 Tampconfig file structure

To connect the discrete (symbolic) planning with continuous motion planning, we have a file like the following:

```
<?xml version="1.0"?>
<Config>
  <Problemfiles>
      <pddldomain name="ff-domains/madar_world.pddl" />
      <pddlproblem name="ff-domains/travel-to-shelf" />
      <kautham name="OMPL_RRTconnect_madar_case_move_a_b.xml" />
      <directory name="/demos/OMPL_geo_demos/madar_bar/"/>
```

```
      <graspit name="DUMMY NO GRASPIT TO BE USED HERE"/>
  </Problemfiles>
  <States>

      <!-- Initial state with robot at HOME position -->
      <Initial>
          <!-- Robot's initial configuration (controlfile can be adjusted as per your setup) -->
            <Object name="GLASS1" kthname="glass1">60 110 73.5 0 0 1 0</Object> <!-- orientation in ax
            <Object name="BOTTLE" kthname="bottle">130.0 -20.0 43.5 0 0 0 0</Object> <!-- orientation
          <Robot name="MADAR" controlfile="controls/madar_R2.cntr"> 0.5 0.5 0.5 </Robot>
      </Initial>
  </States>

  <Actions>
      <!-- Move actions between HOME and SHELF -->
      <Move robot="MADAR" region_from="HOME" region_to="SHELF">
          <Rob> 0 </Rob>
          <Cont> controls/madar_R2.cntr </Cont>
          <InitControls> 0.5 0.5 0.5 </InitControls>
          <GoalControls> 0.75 0.75 0.25 </GoalControls>
      </Move>

<Approach_section robot="MADAR" hand="LEFT-HAND" location="SHELF" section="SHELF_1">
    <Rob> 0 </Rob>
    <Cont> controls/madar_left_arm.cntr </Cont>
    <!-- Base X, Y, Z, followed by 6 arm joint angles -->
    <InitControls> 0.45 0.58 0.13 0.5 0.5 0.7 </InitControls>
    <GoalControls> 0.24 0.395 0.274 0.620 0.774 0.632</GoalControls>
</Approach_section>

<Pick robot="MADAR" hand="LEFT-HAND" object="GLASS1" section="SHELF_1">
    <Rob>0</Rob>
    <Obj>glass1</Obj>
    <Link>82</Link>
    <Cont>controls/hand_pick/madar_left_hand.cntr</Cont>
    <Regioncontrols>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</Regioncontrols>
    <Graspcontrols>
        <grasp grasp="1">0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</grasp>
    </Graspcontrols>
```

```
</Pick>

<Retractarm robot="MADAR" hand="LEFT-HAND" location="SHELF" section="SHELF_1">
    <Rob> 0 </Rob>
    <Cont> controls/madar_left_arm.cntr </Cont>
    <InitControls> 0.24 0.395 0.274 0.620 0.774 0.632 </InitControls>
    <GoalControls> 0.45 0.58 0.13 0.5 0.5 0.7 </GoalControls>
</Retractarm>

<Place robot="MADAR" hand="LEFT-HAND" object="GLASS1" section="PREP-SECTION1">
    <Rob>0</Rob>
    <Obj>glass1</Obj>
    <Cont>controls/hand_place/left_hand_prep_1.cntr</Cont>
    <Regioncontrols>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</Regioncontrols>
    <Graspcontrols>
      <grasp grasp="1">0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</grasp>
    </Graspcontrols>
</Place>

<Prepare_drink robot="MADAR" glass="GLASS1" section="PREP-SECTION1">
    <Rob>0</Rob>
    <Obj>bottle</Obj>
    <Link>82</Link>
    <ArmCont>controls/arm_aproach/left_prep1.cntr</ArmCont>
    <HandCont>controls/hand_pick/madar_prepare_1.cntr</HandCont>
    <InitControls>0.45 0.58 0.13 0.5 0.5 0.7</InitControls>
    <GoalControls>0.293 0.451 0.447 0.534 0.65 0.628</GoalControls>
    <Regioncontrols>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</Regioncontrols>
    <Graspcontrols>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</Graspcontrols>
    <GlassPosition>0.229 0.391 0.447 0.5 0.737 0.861</GlassPosition>
</Prepare_drink>

  </Actions>
</Config>
```

Within the `<Actions>` block, each domain action (`move`, `pick`, `prepare_drink`, etc.) has a corresponding XML tag specifying *which robot index* (e.g. `<Rob>0</Rob>`), the *control file* for that action, plus `InitControls`, `GoalControls`, and `Graspcontrols` if relevant.

5.3 task planning solution

Given these domain and problem definitions, and the TAMP bridging, an example high-level plan to deliver a drink might look like:

```
1) move MADAR HOME SHELF
2) approach_section MADAR LEFT-HAND SHELF SHELF_1
3) pick MADAR LEFT-HAND glass1 SHELF_1
4) retractarm MADAR LEFT-HAND SHELF SHELF_1
5) move MADAR SHELF PREPARATION-TABLE
6) prepare_drink MADAR glass1 prep-section1
7) move MADAR PREPARATION-TABLE TABLE2
8) approach_section MADAR LEFT-HAND TABLE2 table2-left
9) place MADAR LEFT-HAND glass1 table2-left
10) retractarm MADAR LEFT-HAND TABLE2 table2-left
```

Throughout these steps:

- The PDDL domain ensures preconditions (e.g. arms retracted before `move`).
- The TAMP file passes `InitControls` and `GoalControls` to motion scripts, implementing safe paths for the base or arm.

5.4 task planning conclusions

By combining PDDL domain actions with a `tampconfig` XML bridging layer, we achieve a **cohesive** Task and Motion Planning solution. Each discrete action from the domain (like `approach_section` or `prepare_drink`) is grounded in the real geometry of the scene:

- **Reduced complexity**: The domain remains a concise symbolic definition, while the TAMP file handles numeric robot joint controls and path queries.
- **Collision-free**: Each step is validated by a motion planner before finalizing, ensuring feasibility.
- **Extendability**: We can easily add new sections, objects, or actions (like `deliver_tray`) by defining domain logic and TAMP tags.

6 Conclusions

This project demonstrated a complete task-and-motion planning (TAMP) pipeline applied to a realistic bar-tending scenario. By combining a carefully modeled bar environment in *Blender* with the MADAR robot's URDF and controllers, we showed how task-level actions can be mapped to geometrically valid trajectories for mobile manipulation tasks.

Key insights include:

- **Holistic Integration:** The workflow spanned from low-level robot description (URDF, controllers) and bar geometry, to high-level PDDL domain/problem files for symbolic reasoning, all seamlessly bridged by the `tampconfig` structure.
- **Extensibility of Actions:** Custom actions like `prepare_drink` and domain constraints (e.g., `arm-retracted` for safe navigation) allowed flexible expansions of the scenario. In principle, new objects and tasks (cleaning, restocking) can be added with minimal extra code.
- **Reliance on Motion Planning Accuracy:** The quality of the motion planner, collision approximations, and the manipulator's degrees of freedom strongly influenced feasibility rates. Careful simplifications of collision geometry sped up planning without compromising correctness.
- **Modular Architecture:** Each action script (e.g., `MOVE.py`, `PICK.py`, `PLACE.py`) is independent yet orchestrated by the domain logic. This modularity streamlines debugging and fosters maintainability as the environment or robot evolves.

Future Improvements. Potential next steps involve:

- *Advanced Perception:* Incorporating sensor feedback and object-detection modules to handle uncertain object poses.
- *Dynamic Environments:* Allowing the environment or the tasks to change mid-plan, necessitating online re-planning or partial plan repair.
- *Multi-Robot Collaboration:* Extending the domain to multiple agents could increase throughput in large-scale bar scenarios.
- *Human-Robot Interaction:* Integrating user commands or gestures for a more natural bar service experience.

By demonstrating robust pick-and-place actions, drink preparation, and a flexible TAMP structure, we have laid the groundwork for a wide spectrum of future research and real-world applications where robots autonomously interact with structured environments to perform service tasks.