ENEE 467: Robotics Project Laboratory
Collision-Free Kinematic Motion Planning in ROS 2
Lab 5 — Part II

This lab provides a hands-on opportunity to explore the feature-rich MoveIt motion planning framework. Students will learn how motion planning requests in MoveIt are executed in a simulated environment within the Gazebo simulator, and how robot kinematic models, semantic collision information, and robot controllers connect to form a complete motion planning pipeline. After completing this lab, students should be able to:

- Use a MoveIt configuration package to interface with the motion planning framework
- Plan and execute motions via RViz on a simulated robot
- Define planning groups, poses, and collision info with SRDF
- Write collision-free joint-space planners using `pymoveit2`
- Execute trajectories via `ros2_control`
- Dynamically add objects to the planning scene
- Control robot components (e.g., grippers) with ROS 2 action clients

Motion planning computes collision-free, time-parameterized trajectories in joint or Cartesian space that move a robot from an initial to a goal configuration while respecting obstacles, joint limits, and dynamics. Unlike pure kinematics, it explicitly considers the robot's workspace and dynamic constraints and is thus critical for robots operating in semi-controlled or unstructured settings. Together, the two parts of this lab aim to equip you with the skills and tools to use modern motion planning frameworks (e.g., *MoveIt*) and to develop your own ROS 2–compliant motion planning programs.

Related Reading

- M. Spong et. al., "*Robot Modeling and Control*", John Wiley & Sons, Inc., 2006.
- PickNik Robotics, "*MoveIt Developer Concepts*." Available: Webpage.
- G. Russi, "*Robotics Part 32 — Trajectory Generation*." Available: Robotics Unveiled.
- S. Castro, "*How Do Robot Manipulators Move?*" Available: Robotic Sea Bass.

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab Questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab Exercises at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

List of Questions

List of Exercises

How to Read This Manual

Throughout the manual, boxes beginning with a **Question #** title contain questions to be answered as you work through the lab:

> **Question 0.** A question for you.

Boxes with a 🍃 symbol (like the one below) are extra theoretical details provided for your enlightenment or just general information about the lab. We recommend that you read them during the lab, but they may be skipped during the lab and revisited at a later time (e.g., while working on the lab report):

> 🍃 Some extra context and info.

Terminal commands and code blocks will be set in gray-colored boxes (usually for Python syntax)

```
import math # some important comment you should totally read
```

while terminal outputs will be set in yellow-colored boxes

```
some terminal output
```

Files and directories will be set in magenta color with a teletype font, e.g., `~/some/dir/to/a/file.txt`, code variable names and in-text Python commands or snippets will be set in blue teletype font, e.g., `a = [0, 0, 0]`, while Python type hints (and occasionally XML tags) will be set in green teletype font, e.g., `save(path: str)` signifies that the argument to the `save` function is a string. Lastly, figure captions go below figures, table captions appear above tables, and cyan text highlights a link to an external online reference.

1 Lab Procedure

1.1 Prerequisites

This lab assumes familiarity with URDF-based robot modeling and ROS 2, including rigid-body simulation, frame transforms, and forward/inverse kinematics. Prior exposure to motion planning (e.g., sampling-based search or polynomial trajectories) is helpful but not required.

1.2 Getting this Lab's Code

To begin, in a new terminal session, navigate to the `~/Labs` directory and then clone the lab repo:

```
cd ~/Labs
git clone https://github.com/ENEE467-F2025/lab-5.git
cd lab-5/
```

Verify that the lab-5 image is built on your system

```
docker image ls
```

Make sure that you see the `lab-5` image in the output:

```
lab-5-image   latest    <image_id>     <N> days ago   <image_size>GB
```

1.3 Build and Test the Required Packages

To support development of MoveIt-based programs and custom planning tools, we have provided the following seven packages:

- `ur3e_hande_planning_interfaces`: Custom task and scene management interface definitions
- `ur3e_hande_gz`: Gazebo world, models, and robot spawn configuration
- `ur3e_hande_description`: Unified URDF/XACRO and meshes for UR3e + Hand-E system
- `ur3e_hande_moveit_config`: MoveIt 2 configuration package with planners and controllers
- `ur3e_hande_moveit_scripts`: Example scripts for joint-space motion planning
- `robotiq_hande_description`: Base description package for the Robotiq Hand-E gripper
- `pymoveit2`: Instructors' fork of pymoveit2, a Python API exposing core MoveIt functionality.

From within the container, build and source your workspace:

```
cd ~/ros2_ws/
colcon build --symlink-install
source install/setup.bash
```

Then test your build by running the following command from an active container:

```
cd ~/ros2_ws/src && python3 test_docker.py
```

This should print the following output to the terminal (stop and contact your TA if it doesn't):

```
All packages for Lab 5, Part II found. Docker setup is correct.
```

2 Intro to the MoveIt Motion Planning Framework

MoveIt is a motion planning framework and collection of ROS 2 packages providing collision-free kinematic planning, hierarchical task planning, motion control, and advanced manipulator kinematics. Its robot-agnostic design allows planning for any serial-chain manipulator. While MoveIt offers many capabilities, you do not need to master every detail to use it effectively. For an overview of its main components, see the MoveIt System Architecture. There are two primary ways of interfacing with MoveIt:

i. Via the custom MoveIt `MotionPlanning` RViz panel, a convenient GUI with handy buttons for planning, saving robot states, and executing computed plans all within the RViz tool we've used in previous labs. See MoveIt Quickstart in RViz.

ii. Using low-level scripting, either through the C++ or the Python API (there are two main Python APIs here; we will use the better documented and maintained of the two).

In this lab, we'll explore both approaches, starting with the former. Since MoveIt workflows usually begin with a *MoveIt configuration package*, we provide a pre-made configuration and will now review its key components.

2.1 The Structure of a MoveIt Configuration Package

A MoveIt configuration package is a ROS 2 package that sets up the MoveIt framework for a *specific* robot. It effectively specifies the interfaces and functionalities of MoveIt to be used for a particular motion planning task. Inside the `src/ur3e_hande` folder is an **`ament_cmake`** package (`ur3e_hande_moveit_config`) containing the MoveIt configuration for the UR3e-Hand-E robot (here, the gripper used is the Robotiq Hand-E model; see link). Open the `ur3e_hande_moveit_config` folder to view its contents. You should see the following list of folder names (we will only concern ourselves with the `config` folder):

```
config  launch
```

Inspect the `config` folder and note its contents, which typically include `.yaml` files, `.rviz` files, and several XML-like files such as `.urdf.xacro` (robot description) and `.srdf` (semantic information). The exact files and layout may vary by use-case, but the essential components are (we will go over some important parts in later sections):

A. Planning configuration files specifying joint limits, planners and their **C++**-based plugins, inverse kinematics solvers, controllers, and the robot's initial pose.

B. Robot description files containing kinematic, semantic, and control-related information.

C. An RViz file that sets up an instance of RViz with the MoveIt `MotionPlanning` panel.

Question 1.

(a). Suppose you wanted to put together your own MoveIt configuration ROS 2 package. What build type (`ament_cmake` or `ament_python`) would you use and why?

(b). For each file in the `config` folder above, classify them into the categories enumerated in the list above (i.e., **A**, **B**, and **C**) using a table.

> A MoveIt configuration package is usually generated using the *MoveIt Setup Assistant*, included with the `moveit` ROS 2 meta-package (see the MoveIt Setup Assistant Tutorial). While this lab does not cover the tool, you will often need to modify configurations created with it, so knowledge of its use is valuable.

2.1.1 The Semantic Robot Description Format

By default, MoveIt will treat all links of a rigid-body kinematic chain as having the propensity to collide with any other link, unless otherwise specified via special tags in an XML-like description file defined according to the Semantic Robot Description Format (SRDF). The SRDF file is absolutely necessary for specifying a minimal MoveIt configuration package. An excerpt from the UR3e-Hand-E robot's SRDF file is provided below highlighting important tags (see the `ur3e_hande.srdf` file in the `config` folder)

```
<robot name="ur3e_hande">
    <group name="ur_manipulator">
        <joint name="shoulder_pan_joint"/>
        <joint name="shoulder_lift_joint"/>
        <joint name="elbow_joint"/>
        <joint name="wrist_1_joint"/>
        <joint name="wrist_2_joint"/>
        <joint name="wrist_3_joint"/>
        <chain base_link="base_link" tip_link="tool0"/>
    </group>
    <group name="gripper">
        <joint name="gripper_robotiq_hande_left_finger_joint"/>
        <joint name="gripper_robotiq_hande_right_finger_joint"/>
    </group>
    <group_state group="ur_manipulator" name="ur_home">
        <joint name="elbow_joint" value="0"/>
        <joint name="shoulder_lift_joint" value="-1.57"/>
        <joint name="shoulder_pan_joint" value="0"/>
        <joint name="wrist_1_joint" value="-1.57"/>
        <joint name="wrist_2_joint" value="0"/>
        <joint name="wrist_3_joint" value="0"/>
    </group_state>
    <disable_collisions link1="base_link_inertia" link2="table"
    ↪  reason="Adjacent"/>
    <disable_collisions link1="base_link_inertia" link2="wrist_1_link"
    ↪  reason="Never"/>
</robot>
```

The format should appear familiar—the SRDF file is XML-based, just like the URDF. It does not replace the URDF but merely complements it, providing semantic information necessary for collision-free motion planning. From the SRDF file, we can immediately identify a few vital tags:

```
<group></group>
<group_state></group_state>
<disable_collisions />
```

That last tag is *self-closing*, hence we do not need to specify a separate `</disable_collisions>` tag for it. Let's inspect each tag briefly.

Planning Groups and Group States: In MoveIt, a *planning group* is a collection of joints that can be controlled together, typically corresponding to a physical part of the robot, such as the arm or gripper. *Group states* refer to specific joint configurations within a planning group. In the `ur3e_hande.srdf`, two planning groups are defined: one for the arm (`ur_manipulator`) and another for the gripper (`gripper`), and a number of states for each group have also been defined. Planning groups for kinematic *chains* composed of a series of *links* are typically defined using `<joint>`, `<link>`, and/or, more compactly, using `<chain>` tags in the SRDF.

Question 2.

a. How many group states are currently defined for the UR3e? How many for the gripper? List the **names** of the available group states for each planning group.

b. Inspect the group definition for the `ur_manipulator` group. Which other tag can be used to express the same information conveyed by the `<chain>` tag in a more compact form? (**Hint**: see the `<chain>` and `<link>` definitions provided at the ROS SRDF Wiki.)

c. Add the following group states for the UR3e to the SRDF file:

```
ur_approach_cube = [-1.86057, -2.99056, 0.475617, -2.00445, 1.61112,
↪  -0.00712473]
ur_pregrip = [-1.83906, -3.10465, 0.656839, -2.02065, 1.61113,
↪  -0.00707275]
```

Note that the joints in the above arrays are listed assuming the following order of joint names: `shoulder_pan_joint`, `shoulder_lift_joint`, `elbow_joint`, `wrist_1_joint`, `wrist_2_joint`, and `wrist_3_joint`, respectively.

Collision Link Specification: The `<disable_collisions />` tag in MoveIt's SRDF file excludes specific link pairs from collision checking during motion planning, improving collision-checking efficiency. This is useful for passive robot links or parts that won't interact with the environment. Link pairs without this tag (or with it commented out) will be included in the collision check.

Question 3. `cd` to your `ros2_ws` folder, build the MoveIt config package, and source:

```
cd ~/ros2_ws/
colcon build --symlink-install --packages-select ur3e_hande_moveit_config
source install/setup.bash
```

Then run the MoveIt bringup launch file for the robot using:

```
ros2 launch ur3e_hande_moveit_scripts gz_moveit.launch.py
```

This will open RViz and Gazebo. Take a quick look at the RViz window, then terminate the launch. Edit the SRDF to *enable* collision checking for `base_link_inertia` and `shoulder_link` by commenting out the corresponding `<disable_collisions />` tags (use `Ctrl + /` in VSCode). Relaunch and note any visual changes in RViz. No figures are needed. **Remember to revert the change to the SRDF file after making note of the changes above.**

2.2 Plan & Execute Motions via the MoveIt MotionPlanning RViz Panel

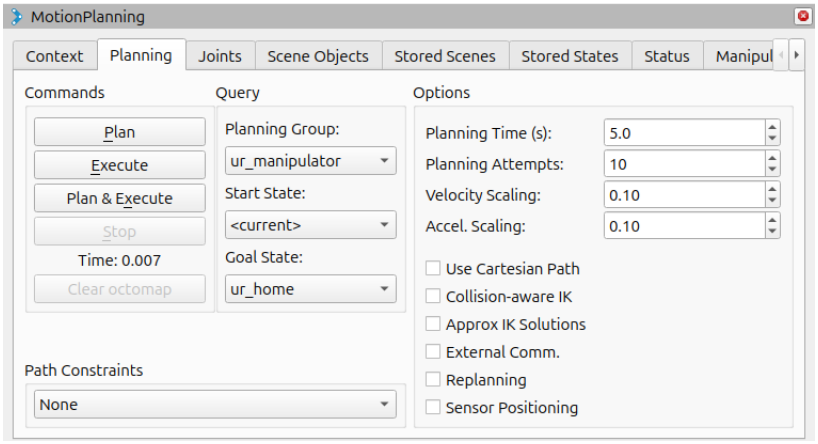


Figure 1: The MoveIt `MotionPlanning` RViz panel.

Having examined the elements of a typical MoveIt configuration, let's put it to use. Change directory to the workspace root, build with `colcon`, and source your workspace

```
cd ~/ros2_ws/
colcon build --symlink-install
source install/setup.bash
```

Then run the MoveIt bringup launch file (The `controller_manager` warning about waiting for `robot_description` can be safely ignored; it is a known issue with `ros2_control`.) To the left of the open RViz instance, you should see a custom panel (titled `MotionPlanning`; see Figure 1) containing MoveIt utilities that we will now use. The `Planning` tab contains several blocks: `Command` provides buttons to plan, execute, or do both; `Query` has dropdowns for selecting planning groups and start/goal states; and `Options` lets you configure planning parameters.

Try planning and executing a motion plan for the `ur_manipulator` group from the `<current>` start state to the `ur_test` goal state (At launch time, i.e., before any motion requests are sent to MoveIt, the `<current>` state is the robot's home configuration, `ur_home`). Note that planning and then executing using separate commands may not always succeed on the first run, whereas the `Plan & Execute` command often works on the first attempt.

Question 4.

a. Suppose we want to compute a motion plan between the saved group states `ur_home` and `ur_place` using the `MotionPlanning` RViz panel. If the robot is currently in the `ur_test` position, outline two approaches for achieving the required motion plan, assuming an RViz instance with all MoveIt planning services is already running.

b. Plan and execute a motion for the `ur_manipulator` group from the `ur_home` start state to the `ur_approach_cube` goal state (visible if you completed Question 2c). Observe the robot in Gazebo. Explain why it collides with the obstacle, and include a screenshot of the collision in your lab report.

2.3 Kinematic Constraints and MoveIt Motion Planning

So far, we've planned motions for the UR3e-Hand-E using saved group states and the `MotionPlanning` RViz panel. MoveIt computes feasible end-effector trajectories in Cartesian space, considering both position and orientation. However, it assumes the kinematic chain has enough degrees of freedom to achieve these poses; reducing DOF can limit MoveIt's ability to compute motion plans for arbitrary Cartesian goals.

> **Question 5.** Consider the kinematic model of the UR3e-Hand-E robot (a simplified URDF is provided in `simple_ur3e_hande.urdf` file within the `urdf` folder in the `ur3e_hande_description` package). Suppose the last actuated joint of the UR3e (`wrist_3_joint`), which connects `wrist_2_link` and `wrist_3_link`, along with `wrist_3_link`, were removed, and the Hand-E gripper's base (`robotiq_hande_link`) were attached directly to `wrist_2_link`.
>
> Argue for or against why it would be impossible to compute motion plans for **Cartesian** pose goals with arbitrary orientation for this modified arm-gripper system. (**Hint**: see §1.3.2 on page 13 of Prof. Spong's book here.)

3 Programmatic MoveIt-Enabled Planning via PyMoveIt2

So far, we have used MoveIt's RViz panels for manual motion planning and execution. While convenient, this interface is limited to supervised use. For more autonomous behaviors, scripting is essential. The open-source `pymoveit2` package (PyMoveIt2) provides Python bindings to MoveIt 2, exposing planning, kinematics, and scene management within ROS 2, without the complexity of C++ interfaces.

However, before learning how to programmatically command the robot, let's first see how to instantiate a MoveIt interface with PyMoveIt2. We'll focus on configuring planners and programmatic scene management, but note that the library provides many other useful capabilities. Depending on your use case, you may need to explore MoveIt's documentation and PyMoveIt's source code to discover them.

3.1 Configuring Planners

The `ur3_moveit_config.py` script in the `ur3e_hande_moveit_scripts` package demonstrates a minimal ROS 2 node that creates a MoveIt2 interface for a specific robot. As with standard MoveIt configurations, the PyMoveIt2 interface must be tied to a robot description; we first import a summarized description of the robot to create the interface:

```
from pymoveit2 import MoveIt2, MoveIt2State
from pymoveit2.robots import ur3e_hande as robot
```

We then create an interface object using the MoveIt2 constructor, passing in robot-specific data:

```
self.robot = robot
self._cb_group = ReentrantCallbackGroup() # Callback group for concurrency

# Create an instance of MoveIt2 interface
self.moveit2 = MoveIt2(
    node=self,
    joint_names=self.robot.joint_names(), # UR3e-Hand-E joint names
    base_link_name=self.robot.base_link_name(), # UR3e-Hand-E base link name
    end_effector_name=self.robot.end_effector_name(),   # UR3e-Hand-E EE name
    group_name=self.robot.MOVE_GROUP_ARM,
    callback_group=self._cb_group)
```

Notice the `planner_id` parameter

```
self.declare_parameter("planner_id", "RRTConnectkConfigDefault")
```

Here, the default value (`RRTConnectkConfigDefault`) is the RRT* *planner identifier* for the OMPL planner interface within MoveIt (see here). This setting specifies that MoveIt should use the RRT* algorithm for path generation (the same algorithm from Part I of this lab!).

Question 6. What file in the MoveIt configuration package specifies parameters for the MoveIt OMPL planner interface? List three identifiers for other planners compatible with this interface. (**Hint**: see OMPL Planning.)

3.2 The MoveIt2 Planning Scene

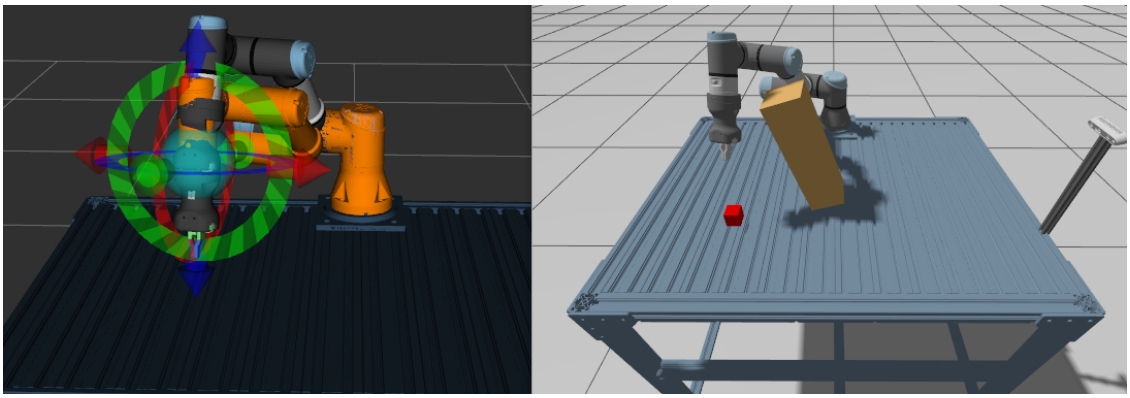


Figure 2: Without an up-to-date planning scene, collisions are inevitable.

A *planning scene* is a snapshot of the robot's environment that must be kept up to date so the planner can accurately avoid collisions as objects or the robot's workspace change (see Figure 2). Through this unit and Exercise 1, we will develop a set of tools for maintaining an updated belief of the robot's environment characterized by a MoveIt planning scene as well as how to directly modify the planning scene for collision-free motion planning (see: MoveIt Planning Scene Tutorial).

> **A Word of Caution:** In MoveIt, not all objects are equal. *Collision objects* must always be avoided, while *non-collision objects* allow intentional contact, such as during grasping, manipulation, or self-contact between robot links.

Each object in a MoveIt planning scene is a *collision object*, and the planning scene is a collection of such objects. There are typically five operations we want to be able to perform to update a MoveIt planning scene:

1. Clear the scene of objects at once
2. Load/populate the scene with a series of object in one shot
3. Selectively remove collision objects from the scene
4. Selectively add objects to the scene
5. `Move`/relocate an existing collision object by updating its position/pose

Luckily, PyMoveIt2 exposes interfaces for these functions. The `ur3e_hande_moveit_scripts` package includes `ur3_moveit_scene.py`, which implements a node for loading, clearing, and removing scene objects. Build the package by running the following command:

```
cd ~/ros2_ws/
colcon build --symlink-install --packages-select ur3e_hande_moveit_scripts
source install/setup.bash
```

3.2.1 Loading a MoveIt2 Planning Scene

In a sourced `tmux` session, split panes, and start the MoveIt launch bringup in one pane:

```
ros2 launch ur3e_hande_moveit_scripts gz_moveit.launch.py
```

Once all windows load, in another `tmux` pane, we can load several collision objects defined in a `scene.yaml` file (located in the `config` folder within the `ur3e_hande_scene_manager` `ament_python` package) by running the node (`ur3_moveit_scene_node`) and setting `load`:=`True`:

```
ros2 run ur3e_hande_moveit_scripts ur3_moveit_scene_node --ros-args -p \
load:=True
```

You should see several collision objects in green appear on the robot's workspace in RViz, and the node should indicate that three objects were loaded to the planning scene.

> **Question 7.** Try planning to the `ur_place` group state after loading the collision objects from the YAML file. In a few sentences, explain why MoveIt is able or unable to find a plan.

3.2.2 Clearing the Planning Scene

Stale objects should be cleared to compute plans with the most up-to-date information. To clear all objects, terminate the node and run the node, specifying the `clear` parameter as `True`:

```
ros2 run ur3e_hande_moveit_scripts ur3_moveit_scene_node --ros-args -p \
clear:=True
```

You should see the robot's workspace cleared of obstacles, and the node should print the following message to shell: "`Successfully cleared planning scene`."

3.2.3 Using ROS 2 Services

The above scene management approaches freeze the geometry of the objects added to the planning scene. However, we would like to have more control over the parameters of the collision object(s). One way of providing such functionality is via ROS2 services. The `ur3e_hande_planning_interfaces` package, located in the `src/ur3e_hande` folder contains the following service definitions for the last three items listed in Section 3.2: `AddObject`, `MoveObject`, and `RemoveObject`. Change directory to the workspace root, and build and source your workspace:

```
cd ~/ros2_ws
colcon build --symlink-install
source install setup.bash
```

Run an `interface show` directive using the `ros2` cli to check each service's definition like so:

```
ros2 interface show ur3e_hande_planning_interfaces/srv/AddObject
```

You should see the following output printed to the terminal:

```
# Add a new object to the planning scene
# Request
string id
string shape             # "box", "sphere", "cylinder", or "cone"
float64[] position       # [x, y, z]
float64[] quat_xyzw      # [x, y, z, w]
float64[] dimensions     # box=[x, y, z]; sphere=[r]; cylinder/cone=[h, r]
---
# Response
bool success
string message
```

Here we see that the `AddObject` service adds an object to the planning scene with the following request fields: `id`, `shape` (box, sphere, cylinder, cone), `position`, `quat_xyzw`, and `dimensions` (shape-specific). The response fields include `success` (boolean) and `message` (status string).

Question 8. What are the request and response fields of the `RemoveObject` and `MoveObject` service definitions? Why is the `position` field not required for removing scene objects?

Exercise 1. Simple MoveIt Planning Scene Manager (30 pts.)

As we saw in the examples leading up to this point, we can manually modify the robot's planning scene with ad-hoc scripts. We would like to develop a scene manager node that provides *persistence*, i.e., a single point of control that keeps track of all collision objects in the world, and *automation*, allowing easy scene resets and consistent setups for experiments. In this exercise, we will build a simple planning scene manager providing services for performing the following within the UR3e's MoveIt planning scene:

(a). adding collision objects
(b). moving collision objects

Deliverables:

Inside the `src` folder is an `ament_python` package named `ur3e_hande_scene_manager`. Complete the `scene_manager.py` node class file to add and move collision objects within a MoveIt planning scene by performing the following steps (**Note**: To reduce workload, the logic providing the service for removing objects within the `scene_manager` node has already been provided, so you DO NOT need to implement it):

Service for Adding Objects:

a. Create a class method `add_object_cb` for adding objects to the planning scene, and add the following template code to it (the `TODO` highlighted block is yours to fill in:

```
def add_object_cb(self, request, response):
    ########################################################################
    # TODO (Ex 1a): Modify obj with fields from the request argument
    ########################################################################
     obj = {}
     try:
         # TODO: 1. add collision object (obj) using class helper
         # function add_collision_object
         # TODO: 2. declare success by setting the success field of the
         # response argument to True
         response.message = f"Added object '{request.id}'"
     except Exception as e:
         response.success = False
         response.message = str(e)
     return response
```

This method should add a collision object (`obj`) using the class helper function `add_collision_object` that takes as argument the dictionary specifying the collision object to be added, `obj`. Fill in the `TODO` highlighted block within the method with the fields of the request argument as follows:

- Extract the fields `id`, `shape`, `position`, `quat_xyzw`, and `dimensions` from the incoming service request and store them as entries in the dictionary `obj`.
- Ensure that each list-valued field (e.g., `position`, `quat_xyzw`, `dimensions`) is converted to a Python list using `list()` before assignment.

b. Using the `create_service` base (`Node`) class method, i.e., `self.create_service` (see rclpy.node.Node.create_service), instantiate a class service of type `AddObject`, name `~/add_object`, and set its callback to the method you created in (a).

Service for Moving Objects:

c. Modify the class method `move_object_cb` for moving objects to a new location within the planning scene. Modify the `TODO` block in the template provided to move collision objects by calling the provided helper class method `_move_collision_object` that takes in the following arguments stored within the `request` object:

- the object's unique identifier, `id`
- the updated position that must be passed as a list `position`
- the updated orientation quaternion that must also be passed as a list using `quat_xyzw`

d. Using the same approach in (b), instantiate a class service of type `MoveObject`, name `~/move_object`, and set its callback to the class method you created in (c).

Build and Test Your Scene Manager Package:

After making your modifications, build the `ur3e_hande_scene_manager` package:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select ur3e_hande_scene_manager
source install setup.bash
```

Then split panes. In one pane, run the node:

```
ros2 run ur3e_hande_scene_manager scene_manager
```

Monitor the active terminal for a prompt indicating that the scene manager is running. Once the prompt appears, in a second `tmux` pane, check that the services are advertised:

```
ros2 service list | grep object
```

You should see the following list if your implementation is correct:

```
/scene_manager/add_object
/scene_manager/move_object
/scene_manager/remove_object
```

Launch the MoveIt bringup

```
ros2 launch ur3e_hande_moveit_scripts gz_moveit.launch.py
```

Test your node by adding a box using the `/scene_manager/add_object` service:

```
ros2 service call /scene_manager/add_object \
ur3e_hande_planning_interfaces/srv/AddObject \
"{id: 'box1', shape: 'box', position: [0.5, 0.5, 0.1065],
quat_xyzw: [0,0,0,1], dimensions: [0.1, 0.1, 0.25]}"
```

Once the box appears, the `scene_manager` node will print the following text indicating successful collision object addition:

```
Added box 'box1' at [0.5, 0.5, 0.1065]
```

Test the `/scene_manager/move_object` service by moving the box to the center of the robot's support surface ($\approx [0.0, 0.35, 0.1065]$):

```
ros2 service call /scene_manager/move_object \
ur3e_hande_planning_interfaces/srv/MoveObject \
"{id: 'box1', position: <new_pos>, quat_xyzw: [0,0,0,1]}"
```

making sure to change the `<new_pos>` placeholder in the service call to the above position.

Node Performance Evaluation

e. What happens when you add the box at a position that collides with the table? You will need to remove the previous box using the `/scene_manager/remove_object` service:

```
ros2 service call /scene_manager/remove_object \
ur3e_hande_planning_interfaces/srv/RemoveObject \
"{id: 'box1'}"
```

Verify that the `scene_manager` node confirms object removal (the following text will be printed to the terminal: "`Removed 'box1'`.") Then add a collision box at the following position: $[0.5, 0.0, 0.025]^\top$. Record your observation in your report.

f. Recall the observation from Question 4. Let's see if we can remedy that situation by adding scene awareness. Terminate all running ROS 2 processes, then, in separate `tmux` panes, run the MoveIt bringup launch file, and start your `scene_manager` node. Then add a collision object in another pane using:

```
ros2 service call /scene_manager/add_object \
ur3e_hande_planning_interfaces/srv/AddObject \
"{id: 'box1', shape: 'box', position: [0.025, 0.42, 0.165],
quat_xyzw: [0,0,0,1], dimensions: [0.1,0.1,0.35]}"
```

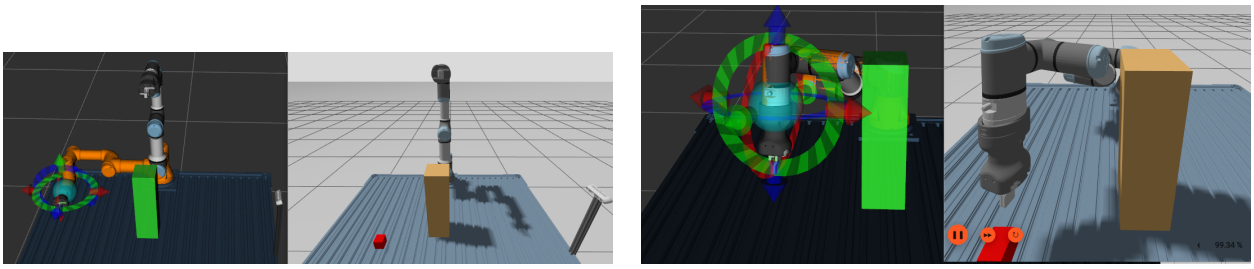
Once the box appears, try planning to the group state `ur_approach_cube` using the MoveIt `MotionPlanning` panel on RViz . You should see the robot move to the desired joint goal while avoiding the box in RViz and Gazebo (**Note**: you may need to plan a few times, as the planner may not find a path initially.). Take a screenshot of your RViz and Gazebo windows side-by-side (similar to Figure 3), and include it in your report.

Files to Submit

To receive full credit for this exercise, make sure you turn in the following files:

- Your modified `scene_manager.py` script containing a complete and fully working implementation of the above methods
- A screenshot of your `tmux` terminal showing the outputs of the scene manager node for all service calls, i.e., from "`Dynamic scene manager Node ready`" to the last printed output after exercise completion

- Screenshots of your RViz window showing the collision box at the first add position, the move position, and the position in (e). So three pictures altogether
- RViz and Gazebo window showing your planning result in (f).



(a) Robot at initial config.

(b) Robot at approach config.

Figure 3: Sample results for Exercise 1. Each subfigure shows RViz and Gazebo views with the collision object.

4 Motion Planning with the PyMoveIt2 MoveIt API

By now, we've explored a few tools for updating the state of the robot's environment or scene and played around with the MoveIt RViz `MotionPlanning` panel for planning to named group states. For complex planning applications, it behooves us to understand how to interact with MoveIt programmatically. The `ur3e_hande_moveit_scripts` package contains `ur3e_hande_joint_goal.py`, which we will sample in this unit.

4.1 Planning to Arbitrary Joint Positions

Open the `ur3e_hande_joint_goal.py` file. Inside it is a script for planning to joint goals. Terminate all running ROS 2 processes, then change directory to your workspace root, build the `ur3e_hande_moveit_scripts` package, and source your workspace:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select ur3e_hande_moveit_scripts
source install setup.bash
```

Start the MoveIt launch bringup in one `tmux` pane:

```
ros2 launch ur3e_hande_moveit_scripts gz_moveit.launch.py
```

Then run the node specifying the target joint position using the `joint_positions` parameter:

```
ros2 run ur3e_hande_moveit_scripts ur3_joint_goal_node --ros-args -p \
joint_positions:="[-0.626, -3.110, 0.681, -2.063, 1.471, -0.007]"
```

Monitor the active terminal. The node will wait for joint states from the `robot_state_publisher` and attempt to compute a trajectory once joint states are available. If a trajectory is computed, you should see purple markers in RViz animating the computed trajectory. Then the robot should begin

to move on RViz and Gazebo. The terminal running the MoveIt bringup nodes should also print the following series of messages at each planning stage:

```
Joint states are available now
Waiting for execution to start (asynchronous mode)...
Execution started.
Execution done.
```

4.2 Planning to Named Group States

We can also plan to named group states in the SRDF file contained in a MoveIt configuration to simplify motion planning by providing semantically meaningful targets that can be referenced programmatically without manually specifying joint values. Restart the MoveIt launch file if it's running. Once all windows appear, in a new `tmux` pane, run the same `ur3_joint_goal_node`, specifying a named group state using the parameter `group_state`:

```
ros2 run ur3e_hande_moveit_scripts ur3_joint_goal_node --ros-args -p \
group_state:="<named_group_state>"
```

where `named_group_state` is a named collision-free configuration (e.g., one of either `ur_home` or `ur_test`). Try running the node for these configurations. You will need to restart the node to plan to new configurations.

4.3 Controlling the Gripper

All that's nice, but what about the gripper? So far we've treated it like a decorative link, but a robot arm is only as useful as its tool. Since proper "planning" for grippers (i.e., Grasp Planning) is more involved, and no such planners currently exist for MoveIt in ROS 2, we'll set planning aside and focus on simple gripper control. Fortunately, ROS 2 offers packages outside MoveIt that make this easy. One such interface is the Parallel Gripper Action Controller from `ros2_control`, the backbone of ROS 2 robot actuation (see: ROS2 Control and Parallel Gripper Action Controller).

4.3.1 The ParallelGripperCommand ROS 2 Action

The action interface implements an `ActionServer` for executing a ROS 2 action of type `ParallelGripperCommand` for simple parallel grippers, and supports grippers that offer position only control as well as grippers that allow configuring the velocity and effort. Recall from Lab 3, that to send an action goal, we can use the following command:

```
ros2 action send_goal <action_name> <action_type> '<goal_message>'
```

With the MoveIt bringup launch file running, check the list of available actions using:

```
ros2 action list | grep gripper
```

where we have filtered the list by "grepping" for the keyword "gripper". You should see the following action printed to the terminal:

```
/gripper_action_controller/gripper_cmd
```

This tells us that the action server `/gripper_action_controller`, with action namespace `gripper_cmd`, is available (verify with `ros2 action info <action>`). We can check the action type of `/gripper_action_controller/gripper_cmd` using

```
ros2 action type /gripper_action_controller/gripper_cmd
```

which should print `control_msgs/action/ParallelGripperCommand`. Examine the action:

```
ros2 interface show control_msgs/action/ParallelGripperCommand
```

You should see the following action definition (abridged for brevity)

```
# Goal
sensor_msgs/JointState command
    string[] name
    float64[] position
---
# Result
sensor_msgs/JointState state
bool stalled
bool reached_goal
---
# Feedback
sensor_msgs/JointState state
```

We see here that the `Goal` message field (`command`) is of type `sensor_msgs/JointState`, that itself specifies the name, position, and optionally, velocity, and effort of the joint we want to control. We are now ready to control the gripper!

4.3.2 Commanding the Gripper from the Command Line

With the MoveIt bringup launch file running, control the Hand-E by sending an action goal (populating the `position` field of the goal message, `command`) with the target joint position to set:

```
ros2 action send_goal /gripper_action_controller/gripper_cmd \
control_msgs/action/ParallelGripperCommand \
"{command: {name: "gripper_robotiq_hande_left_joint", position: [0.02]}}"
```

where the value set to the `position` field must lie within the interval $[0, 0.025]$ meters). Try sending an action goal of 0.020 m. You should see the gripper open almost completely, while the active terminal should prompt you with the following message

```
Goal finished with status: SUCCEEDED
```

Question 9. Try varying the gripper position a few times. Make sure the MoveIt launch file is running as you test different gripper positions. What happens when you send a goal position outside the gripper's limits?

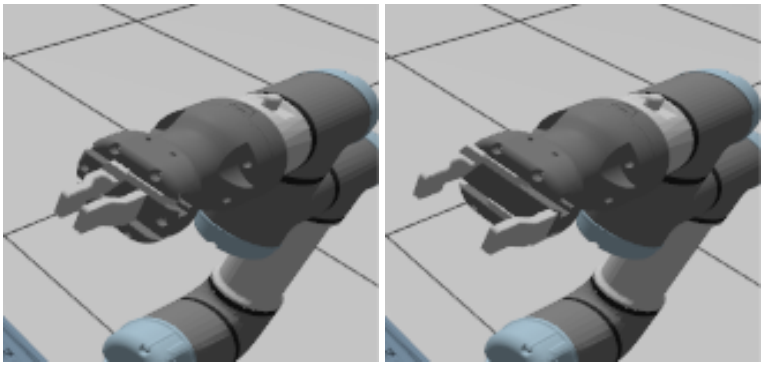


Figure 4: Robotiq Hand-E at 0.005 m (left) and 0.025 m (right), respectively.

Great, we can now command the gripper to desired positions. However, as you can imagine, the above CLI commands quickly become cumbersome if we want to integrate gripper actions into a full manipulation workflow. For example, sequencing arm motions with gripper open/close operations or reacting to sensor input requires a programmatic interface. Thus, we would like to control the gripper directly from code to define our own action primitives and extend the gripper's utility. This extension is the subject of Exercise 2.

Exercise 2. Gripper Action Client (20 pts.)

The `hande_action_client` `ament_python` package within the `src` folder contains the skeleton of a Python file (`hande_command.py`) implementing a ROS 2 node that we will finish developing. To complete this exercise, perform the following actions:

Deliverables:

Node Class Implementation:

a. Inside the `HandeCommand` `__init__` method, initialize an `ActionClient` for the action server `/gripper_action_controller/gripper_cmd` using the action type `ParallelGripperCommand`. Set the client to a class variable named `gripper_client`.

b. Create a class method `send_goal` that accepts a target `position` (meters). The method should build a `sensor_msgs.msg.JointState` message (i.e., the variable `cmd` in the snippet below) with the gripper joint name `"gripper_robotiq_hande_left_joint"` and the target position. Then assign the created `JointState` message to the `command` field of a `ParallelGripperCommand.Goal` message (stored in the variable, `goal_msg`). Use the starter template below, and modify only the highlighted blocks and/or lines:

```python
def send_goal(self,
              position: float,
              max_effort: float = 0.0,
              wait_for_server_sec: float = 5.0):

    if not self.gripper_client.wait_for_server(
        timeout_sec=wait_for_server_sec):
        self.get_logger().error("Action server not available")
        return False

    # Build the action goal
    cmd = JointState()
    cmd.position = [None] # <-- MODIFY
    cmd.name = [" "] # <-- MODIFY
    cmd.velocity = []
    cmd.effort = []

    goal_msg = ParallelGripperCommand.Goal()
    goal_msg.command = None # <-- MODIFY

    send_goal_future = self.gripper_client.send_goal_async(goal_msg)
    send_goal_future.add_done_callback(self._goal_response_callback)
    return True
```

c. Extend your node's entry point (the `main` function) to prompt the user for input using the following `if` block (add this in the `TODO`-highlighted section of the `main` function, and watch out for indentation):

```python
if not ("--open" in sys.argv or "--close" in sys.argv):
    inp = input("Gripper position (0.0-0.025 m, q to quit): ")
    if inp.lower() in ["q", "quit", "exit"]:
        break
    try:
        position = float(inp)
        if not (0.0 <= position <= 0.025):
            raise ValueError
    except ValueError:
        print("Invalid position, try again.")
        continue
```

Build & Test Your Package:

After making your modifications, build the `hande_action_client` package, and source:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select hande_action_client
source install/setup.bash
```

Then launch the MoveIt bringup (if needed), and run your node in a separate `tmux` pane:

```
ros2 run hande_action_client hande_command
```

This should prompt you for a target position with the following text:

```
Gripper position (0.0-0.025 m, q to quit):
```

Typing in a target position and pressing the `Enter` key will move the gripper to the entered position, while pressing the `q` key will terminate the node. Additionally, if the node is called with an `-open` (0.025 m) or `-close` (0.002 m) flag, it will instead execute a preset action:

```
ros2 run hande_action_client hande_command --open # to open the gripper
```

Similarly, for the `-close` preset action, use:

```
ros2 run hande_action_client hande_command --close # to close the gripper
```

In the `-open` and `-close` cases, the node should display the following prompt:

```
Goal accepted, waiting for result...
Result received.
```

Demonstrate your node working by testing the position case and the preset action cases.

Files to Submit

To receive full credit for this exercise, make sure you turn in the following files:

- Screenshots of the node output in the terminal for each test case above
- Your modified `hande_command.py` file containing a complete and fully working implementation of the above blocks and methods.

5 Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Include your solutions to all of the question boxes in the lab report. Refer to the List of Questions to ensure you don't miss any. Include your full solution to the exercises in your lab report. If you wrote any Python programs during the exercises, include these files as a separate attachment, appending your team's number and last names (in alphabetical order) to the file, e.g., **Jane Doe**, **Alice Smith**, and **Richard Roe** in **Team 90** submitting their modified `main.py` would submit the file `main_Team90_Doe_Roe_Smith.py` and the report `Lab_<LabNumber>_Report_Team90_Doe_Roe_Smith.pdf`, where `<LabNumber>` is the number for that week's lab, e.g., **0**, **1**, **2**, etc. Submit your lab report along with any code to ELMS under the assignment for that week's lab. You must complete and submit your lab report by **Friday, 11:59 PM** of the week following the lab session. See the **Files** section on ELMS for a lab report template named `Lab_Report_Template.pdf`. The grading rubric is as follows:

Component	Points
Lab Report and Formatting	10
Question 1.	5
Question 2.	5
Question 3.	5
Question 4.	5
Question 5.	3
Question 6.	2
Question 7.	5
Question 8.	5
Question 9.	5
Exercise 1.	30
Exercise 2.	20
Total	100