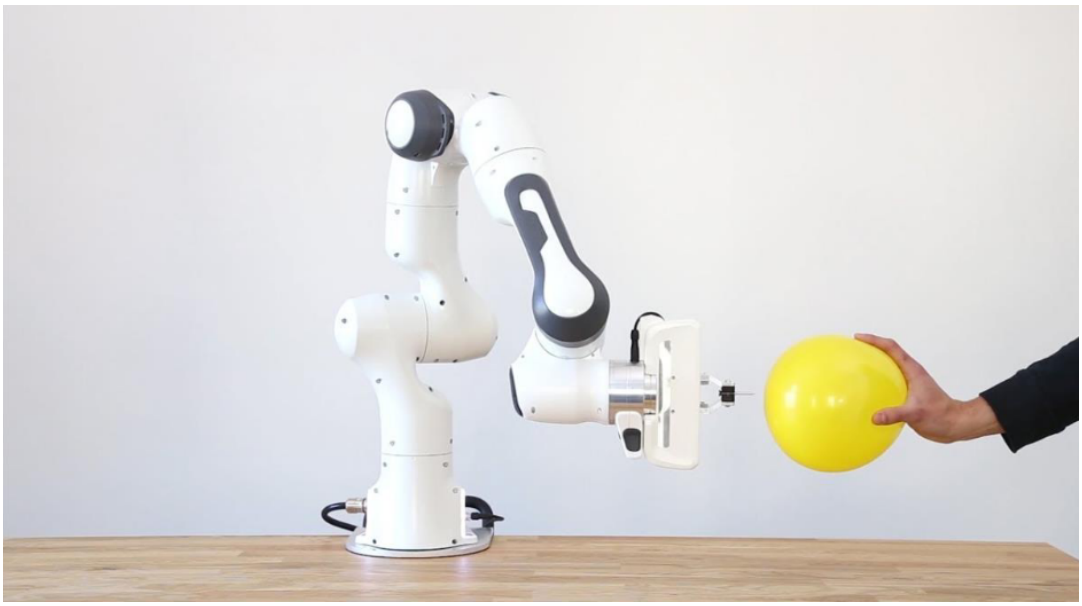
COMP0129: Robotic Sensing, Manipulation and Interaction

Labs

Instructors: *Eddie Edwards, Francisco Vasconcelos*
TAs: Kefeng Huang, Bowie (Heiyin) Won



Lab 2 - MoveIt

Date: Week 21
Goal: ROS Services, MoveIt, and Gazebo practice

This lab is an introduction to MoveIt in ROS. We will be using c++ for this lab and will start with a recap of ROS services in c++. We will simulate our robot in RViz and a physics simulator called Gazebo. We will finish by using a ROS package called MoveIt to move our robot and pick items.

Part 1: Background

1. ROS Services

Services are in the form of srv files and are used to request / reply information between nodes defined by a pair of messages: a request and a response message (http://wiki.ros.org/Services).

Services have an associated service type that is the package resource name of the .srv file. The service type is the package name + the name of the .srv file, (e.g. my_srvs/srv/PolledImage.srv has the service type my_srvs/PolledImage).

Service overview:

1. ROS node offers a service under a string name, defined as an srv file
2. roscpp converts these srv files into C++ source code and creates three classes: service definitions, request messages, and response messages.
3. Client calls the service by sending the request message and awaiting the reply
4. Nodes can only make service calls if both the service type match.

1.1 Command line tools

rossrv – displays information about .srv data structures:

$ rossrv **show**	Show service description
$ rossrv **list**	List all services
$ rossrv **package**	List services in a package
$ rossrv **packages**	List packages that contain services

rosservice – lists and queries ROS Services:

$ rosservice **call** /srv_name srv-args	call the service with the provided args
$ rosservice **find**	find services by service type
$ rosservice **node** /srv_name	name of the node that provides a particular service
$ rosservice **info** /srv_name	print information about service
$ rosservice **list**	list active services
$ rosservice **type** /srv_name	print service type

1.2 roscpp .srv files:

A service description file consists of a *request* and a *response* msg type, separated by '---'. This builds directly upon the ROS msg format to enable *request/response* communication between nodes. Messages could be of various types (see: http://wiki.ros.org/msg), but include built-in types like:

- bool, int32, int64, float32, float64, string
- arrays which are converted to std::vector (add [] to type, eg int32[] or string[])

roscpp converts srv files into C++ source code and creates three classes:

1. service definitions
2. request messages
3. response messages.

The .srv file format:

my_package/srv/Foo.srv →

- my_package::Foo
- my_package::Foo::Request
- my_package::Foo::Response

For information on how to write the individual classes follow the ROS tutorial on: http://wiki.ros.org/roscpp/Overview/Services

1.3 Example .srv files:

Here are some example service files – we will see more of them later! The filename is written at the top in italics.

add_collision.srv

```
string object_name
geometry_msgs/Point centre
geometry_msgs/Vector3 dimensions
geometry_msgs/Quaternion orientation
---
bool success
```

Request

Response

set_arm.srv

```
geometry_msgs/Pose pose
---
bool success
```

set_gripper.srv

```
float64 finger_distance
---
bool success
```

Note the use of additional, non-built-in types from the “geometry_msgs” package.

2. MoveIt

MoveIt is a primary source of the functionality for manipulation in ROS. MoveIt it is a ROS package that could be used for controlling the joints of the robot, performing path planning, accounting for collisions etc. It builds on the ROS messaging and build systems and utilizes some of the common tools in ROS like the ROS Visualizer (RViz) and the ROS robot format (URDF).

The quickest way to get started using MoveIt is through its RViz plugin. RViz is the primary visualizer in ROS and an incredibly useful tool for debugging robotics. The MoveIt RViz plugin allows you to setup virtual environments, create start and goal states for the robot interactively, test various motion planners, and visualize the output.

For information for the MoveIt Package refer to: https://ros-planning.github.io/moveit_tutorials/

2.1 Installation

Firstly it would be helpful to check if MoveIt is already installed:

```
$ rospack list | grep moveit
```

In case you do not have MoveIt installed already, install MoveIt for ROS Noetic, Ubuntu 20.04:

```
$ sudo apt install ros-noetic-moveit
```

For further information on how to operate move it follow the following tutorial: https://ros-planning.github.io/moveit_tutorials/doc/quickstart_in_rviz/quickstart_in_rviz_tutorial.html

3. Gazebo

Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.

Typical uses of Gazebo include:

- testing robotics algorithms,
- designing robots
- performing regression testing with realistic scenarios

For a better understanding on how Gazebo works refer to: https://gazebosim.org/tutorials

3.1 Installation

If you do not have gazebo installed, open the terminal and type the following:

```
$ curl -sSL http://get.gazebosim.org | sh
```

To check if it runs:

```
$ gazebo
```

Next, we want to install gazebo_ros. This is a ROS package that can interface with Gazebo and allow us to use both ROS and Gazebo at the same time.

```
$ sudo apt-get install ros-noetic-gazebo-ros-pkgs ros-noetic-gazebo-ros-control
```

For more information for the Gazebo ROS Package refer to:
http://gazebosim.org/tutorials?tut=ros_overview

Part 2: The moveit_tutorial package

This section goes through the template code for the lab, provided in the moveit_tutorial package.

This package defines a class to provide a series of ROS services which interface with MoveIt to move the Franka Emika Panda arm in simulation, using RViz and Gazebo.

1. Download the package

To start, make sure you have setup your COMP0129 workspace as directed in the first lab, cloning the following repo: https://github.com/COMP0129-UCL/comp0129_s24_labs.git

```
git clone https://github.com/COMP0129-UCL/comp0129_s24_labs.git --recurse-submodules
```

If you already have this workspace set up then you may need to update it, navigate to it (eg $ cd comp0129_s24_labs) and then run:

```
git pull
```

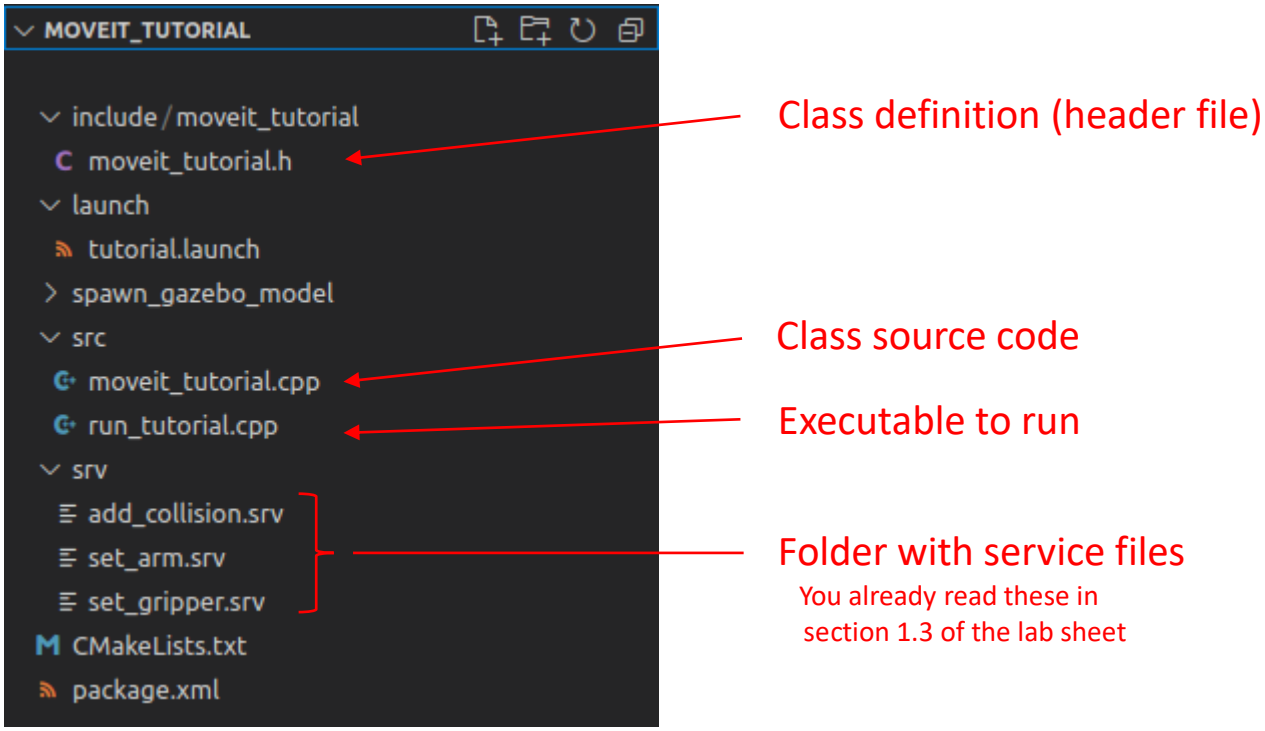
Now check that you can find the lab code, it should be in the '**src/labs/**' directory.

You should now see a folder called **'moveit_tutorial'**, this contains the template code for this lab.

Taking a look at the new package, 'moveit_tutorial', it has the following folder/file structure:

- **include**: contains the header files for our c++ class
- **launch**: contains launch file for the tutorial to run
- **spawn_gazebo_model:** spawn a model into gazebo (you don't need to look at this)
- **src:** contains the c++ source files, one for our class, one to run the node
- **srv:** contains the .srv service files
- **CMakeLists.txt:** builds the class into a library, and links to the node executable
- **Package.xml:** contains information and dependencies

Below is an image of the file structure with all of the **key files** that are useful to read to complete all the exercises **labelled with red arrows**:



2. SrvClass class

This is the class we define in this package which implements a series of ROS services. Next we will go through the key files.

2.1 moveit_tutorial.h

This is the header file which declares the functions and variables that make up the class. At the top of the file are a series of "#includes" which import libraries for us to use. This includes the services that are created within this package (note how there is one for each .srv file):

```
#include <moveit_tutorial/set_arm.h>
#include <moveit_tutorial/set_gripper.h>
#include <moveit_tutorial/add_collision.h>
```

Next, we find the class definition for our class called “SrvClass”. Class variables can be accessed from anywhere within the SrvClass namespace.

The class definition contains two groups of functions:

1. Service callback functions
2. MoveIt functions

The three service callback functions are linked to services we advertise and trigger when someone calls the service. Service callbacks should always return a bool.

```
bool
setArmCallback (moveit_tutorial::set_arm::Request &request,
   moveit_tutorial::set_arm::Response &response);

bool
setGripperCallback (moveit_tutorial::set_gripper::Request &request,
   moveit_tutorial::set_gripper::Response &response);

bool
addCollisionCallback (moveit_tutorial::add_collision::Request &request,
   moveit_tutorial::add_collision::Response &response);
```

Notice each callback has two inputs: request and response. The type of these all follow the same pattern:

- moveit_tutorial::srv_file_name::Request
- moveit_tutorial::srv_file_name::Response

These are automatically generated types from our .srv files, and we import them with the corresponding “#include moveit_tutorial/srv_file_name.h” lines.

The “&” means “reference to”. If a function argument is passed by reference, then if you change the variable inside the function it changes everywhere. In c++ the default is making local copies of each function argument, so we use “&” to prevent this. This helps stop unnecessary copying, in case request or response are big data structures.

Then, there are three MoveIt functions. These are the functions which move the arm and the gripper, and trigger path planning and collision avoidance – all using MoveIt:

```
bool
moveArm(geometry_msgs::Pose target_pose);

bool
moveGripper(float width);

void
addCollisionObject(std::string object_name, geometry_msgs::Point centre,
   geometry_msgs::Vector3 dimensions, geometry_msgs::Quaternion orientation);
```

It is convenient to separate the callbacks from the MoveIt implementation because it adds flexibility in combining different functions.

The addCollisionObject function puts a new collision object into the MoveIt planning scene. The planning scene is used when planning robot paths and motions, so if we add a collision object here then MoveIt will automatically avoid this collision object. For example, a common collision object to add is a ground plane, so ensure the robot does not crash into the ground.

2.2 moveit_tutorial.cpp

This is the source file containing the code for each of the functions of the class.

The constructor is first and is run once when an instance of the class is initialised. It should prepare the class and get it ready. Here, we see that the constructor advertises all the services that this class will offer, and links them to the callbacks.

```
set_arm_srv_ = nh_.advertiseService(service_ns + "/set_arm",
   &SrvClass::setArmCallback, this);

set_gripper_srv_ = nh_.advertiseService(service_ns + "/set_gripper",
   &SrvClass::setGripperCallback, this);

add_collision_srv_ = nh_.advertiseService(service_ns + "/add_collision",
   &SrvClass::addCollisionCallback, this);
```

The keyword “this” is a pointer to the class itself, similar to using “self” in Python. The “&” once again means “reference to”, but now we are using it to pass a reference to a function.

Following this is the code for each of the functions declared in the header file, the three callbacks and then the three MoveIt functions. Read through to see how they work.

2.3 run_tutorial.cpp

This is a very simple file which initialises ROS, creates a class instance, and then spins. The class instance sets up and advertises the service callbacks, then “ros::spinOnce” checks if anyone has made a service request – if they have then the callbacks in the class instance are triggered.

Part 3: Exercises

Exercise 1. Check to make sure you have the following packages in your workspace:

- ☐ labs/moveit_tutorial
- ☐ panda_description
- ☐ panda_moveit_config
- ☐ realsense_gazebo_plugin
- ☐ rpl_panda_with_rs

```
$ cd  ~/comp0129_s24_labs/src
$ ls
$ cd labs
$ ls
```

Exercise 2. Run the package and experiment with services:

1. Compile and source the workspace
2. Run the lab package with $ roslaunch moveit_tutorial tutorial.launch
3. You should see gazebo and RViz start up
4. The panda arm should be upright in both windows, in the same pose
5. After a short delay, you should see a green cube spawn into gazebo in front of the robot, but not into RViz

```
$ cd ..
$ catkin build
$ source devel/setup.bash
$ roslaunch moveit_tutorial tutorial.launch
```

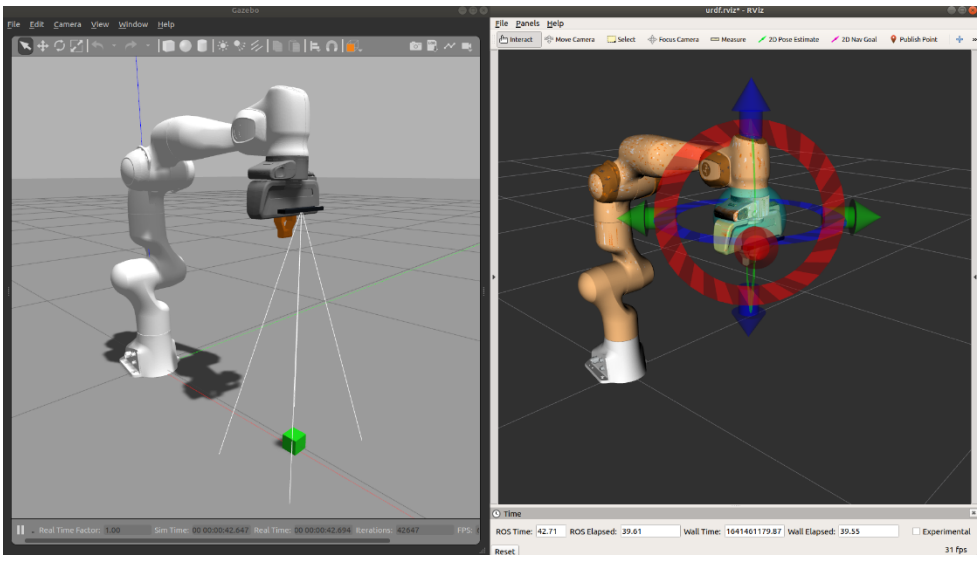


Figure 1: Gazebo (right) and RViz (left) both showing the robot in the same pose. You may see some extra panels/menus which are hidden here

6. Open and source a new terminal window, and list all available ROS services:

```
$ source devel/setup.bash
$ rosservice list
```

7. scroll to /moveit_tutorials and you should see the three services from the source file
    - TIP: filter the list and show only the services from the moveit_tutorial package:

```
$ rosservice list moveit_tutorial
```

8. Call the /set_gripper service using “rosservice call” with finger distance 0.08m
    - TAB can be used to autocomplete service names and service request messages
    - Type in: $ rosservice call /moveit_tutorials/set_grip
    - Press TAB and the service name should autocomplete to /set_gripper
    - Press TAB again and the service message should autocomplete in the terminal
    - Now use the LEFT and RIGHT arrow keys to navigate and fill in the service request message

```
$ rosservice call /moveit_tutorial/set_gripper "finger_distance: 0.08"
```

9. Next try the /set_arm service
    - Use a pose of (0.4, 0.4, 0.4)
    - Use an orientation of (-1, 0, 0, 0)

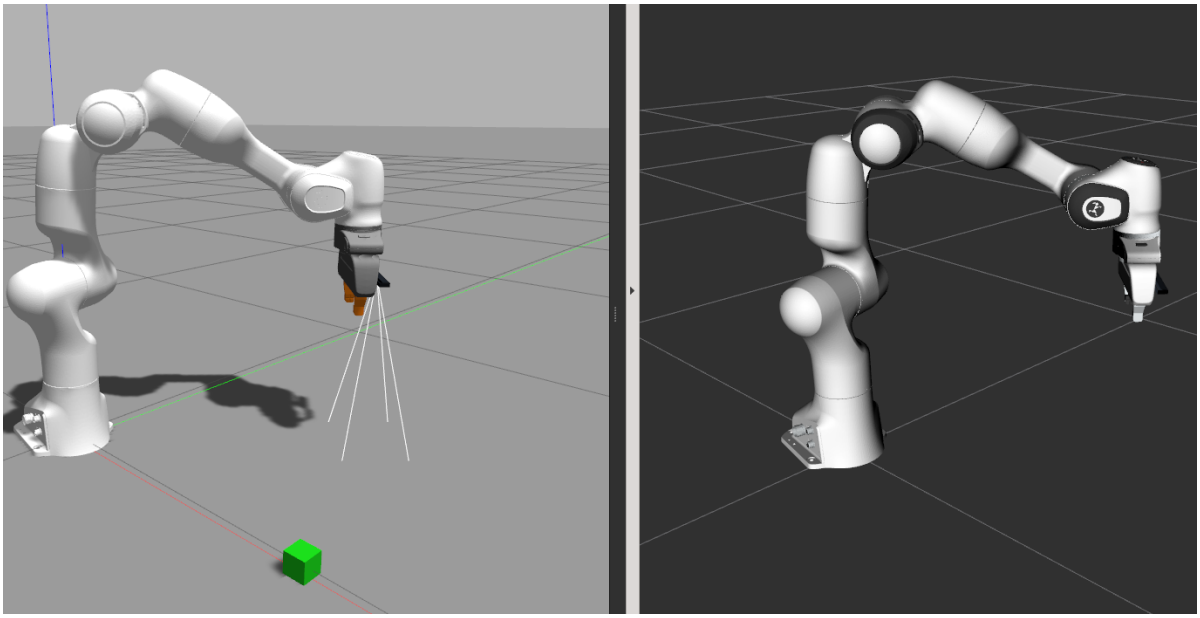


Figure 2: The robot arm moved using the /set_arm service

Exercise 3. Complete the **addCollisionCallback:**

1. Go to line 55 in moveit_tutorial.cpp, see that there is a // TODO
2. Use the addCollisionObject function to get the service to work
3. Check the add_collision .srv file to understand the request
4. Recompile and rerun:

```
$ catkin build
$ source devel/setup.bash
$ roslaunch moveit_tutorial tutorial.launch
```

5. Try calling the /add_collision service now
    - Add a cube called “cube” with dimensions 1m, at centre point (1, 0, 0)
    - The cube is input into the planning scene, which MoveIt uses for collision avoidance. Hence, the cube should appear in RViz because RViz displays the planning scene. The cube will not appear in Gazebo because it does not access the planning scene and is just a physics simulator

```
$ rosservice call /moveit_tutorials/add_collision + SPACE + TAB
```

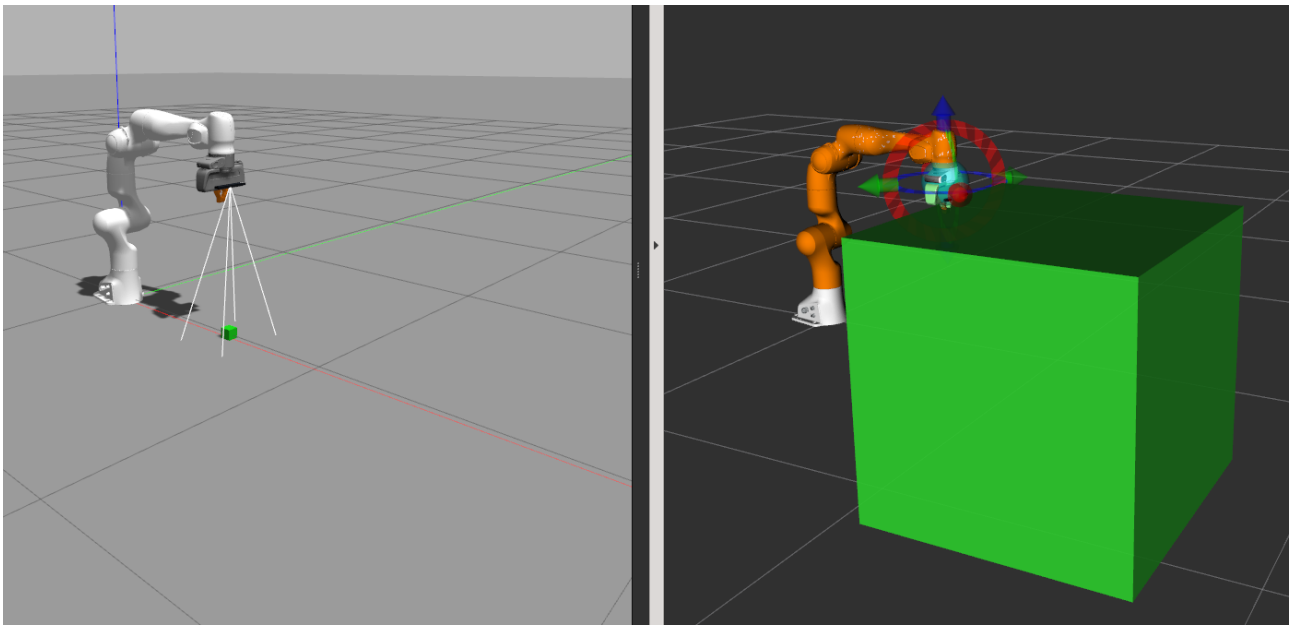


Figure 3: Using the /add_collision service to add a large cube into the MoveIt planning scene. Note that it does not appear in Gazebo (left), as it is just a physics simulator and is separate from MoveIt

Exercise 4. Add a new service called **remove_collision:**

1. Write a new service and add it into the SrvClass
2. Get it to remove a collision object from the planning scene
    - name: “/remove_collision”

- srv input: std_msgs/String object_name
- srv output: bool success

3. Read the addCollisionObject function on line 122 moveit_tutorial.cpp for a hint
4. There are a number of places you need to tell ROS/c++ about the new service:
   - It needs a new service file in the srv folder (see srv/add_collision.srv)
   - CMakeLists.txt needs to know about the service file (see line 61 in CMakeLists.txt)
   - The header for this service that is built by $ catkin build needs to be included in the moveit_tutorial.h header file (see lines 22-24 in moveit_tutorial.h)
   - The service needs to be advertised in the constructor with nh_.advertise(…) (see lines 14-19 in moveit_tutorial.cpp)
   - It needs a ros::ServiceServer defined in the header (see lines 63-65 in moveit_tutorial.h)
   - It needs a callback to be written to implement the service
   - The callback needs to be declared in the moveit_tutorial.h header file
5. Recompile, run, and test if your new service works

Exercise 5. Pick up the green cube in Gazebo:

1. Now we want to create a new service, called "/pick" that will pick up the green cube in Gazebo
2. The location of the cube is (0.6, 0, 0)
3. The side length of the cube is 40mm, but try just closing the gripper all the way
4. Have the service take as input a geometry_msgs/Point
5. Write the service callback function as well as a MoveIt function called:
   - SrvClass::pick(geometry_msgs/Point grasp_point) {…}
6. Use the moveArm and moveGripper functions
   - Be careful! MoveArm is not calibrated to move the end effector to the desired point (instead it moves panda_link8 to the specified point)
   - You will need to experiment and add a z offset in the pick function, as well as adjust the orientation by pi/4 about the z axis (ie roll=0, pitch=0, yaw=pi/4)
7. Run your solution and pick up the cube!

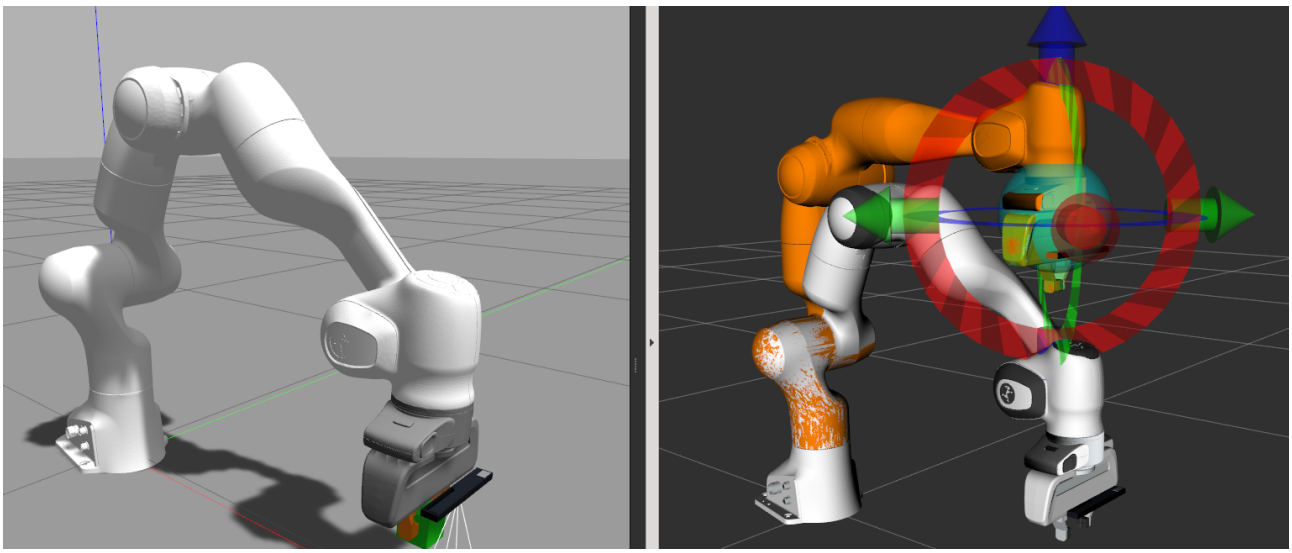


Figure 4: The green cube has been picked up by the arm. Gazebo (left) detects the contacts and forces between the gripper and the cube. RViz does not see the cube but displays the robot pose