Note:

1. This project is from the course, ROS Manipulation in 5 Days on Robot Ignite Academy

: https://www.robotigniteacademy.com/en/course/ros-manipulation-in-5-days/details/

2. Any contents of the project belong to Robot Ignite Academy except for the sample solution written by Samwoo Seong. I.e. I don't own any of the project contents

3. Any work throughout the project is for learning purpose

4. The solution written by Samwoo Seong shouldn't be used to pass the project on this course

Samwoo Seong

samwoose@gmail.com

Project, ROS Manipulation in 5 Days

July 21, 2020

1.0 Objectives

- Create MoveIt! package and Python script so that RB1 robot will be able to grab a cube, play with it, and put it back onto a table.

-(Bonus) Develop a MoveIt! package and Python script so that Fetch robot will be able to perform similar grasping task, but with help of perception.

1.1 Abstract and Motivation

A robotic arm with a gripper has been used in many situations such as manufacturing process. Furthermore, with the robotic arm we can expect them to work in dangerous environments where humans cannot approach easily. It is good to start with a simple application such as moving around a small object with a robotic arm on RB1. Then, the principle behind this basic application can be applied to more complex problems.

1.2 Approach, Procedures, and Sample Solution.

Thanks to ROS MoveIt!, we can create a package that is capable of motion planning and grapping based on our robot configuration without building everything from scratch. The key assumption of using MoveIt! is having a URDF (or XACRO file) of interest of robot on our hands. e.g., “rb1_robot_mico_3fg.urdf.xacro” Before, we jump into building our MoveIt! package, the whole procedure can be done by following 3 major steps.

1. Create the MoveIt! package given a xacro file of RB1 is available
2. Connect the MoveIt! package with the simulation
3. Write up a Python script for a simple grasping task

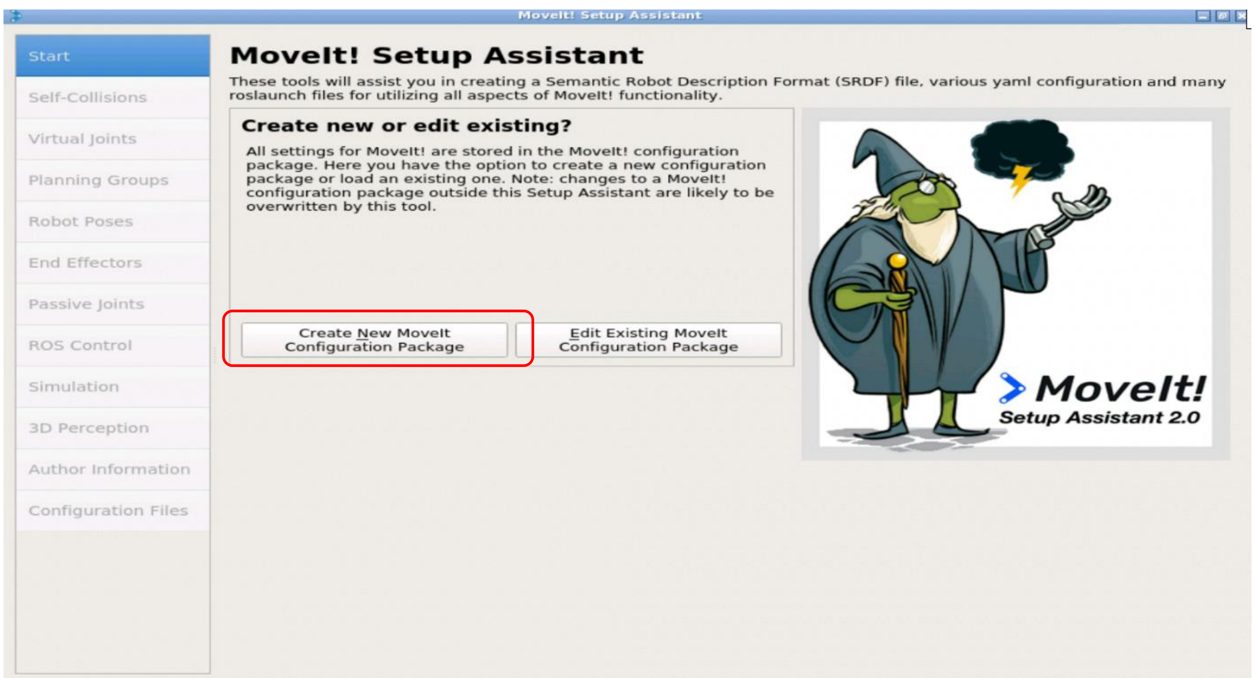
<1. Create the MoveIt! package for RB1>

(1) Launch MoveIt Setup Assistant.

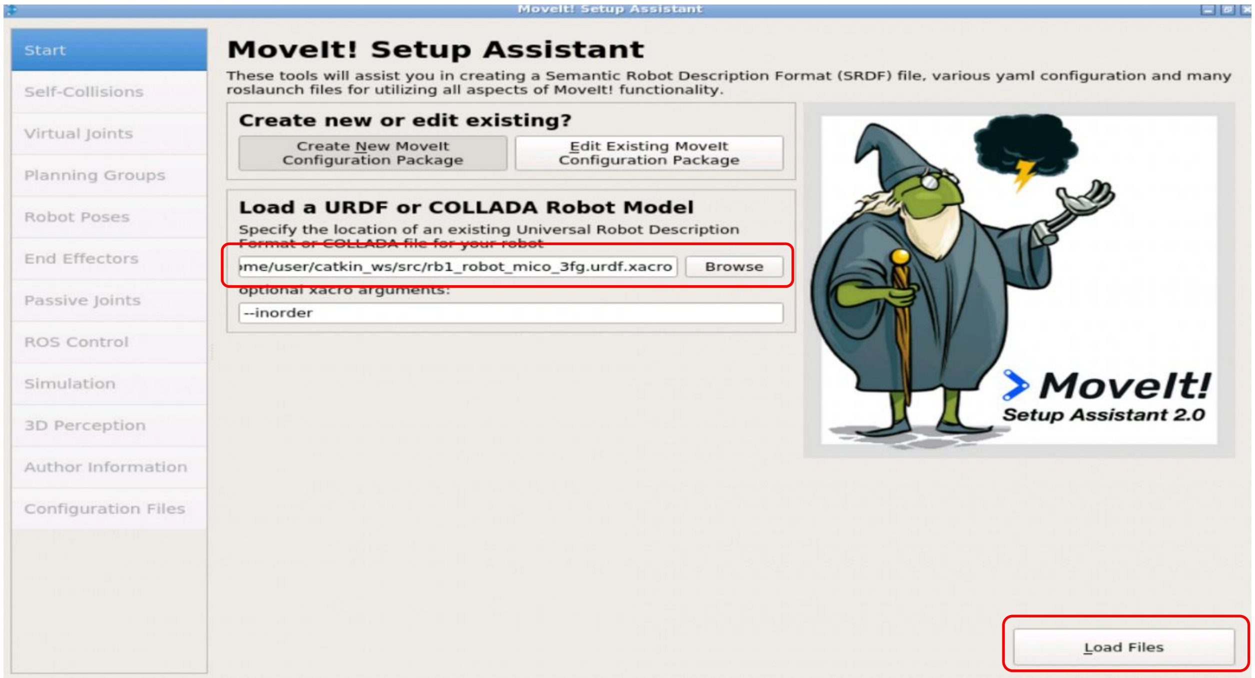
```
user:~$ roslaunch moveit_setup_assistant setup_assistant.launch
```

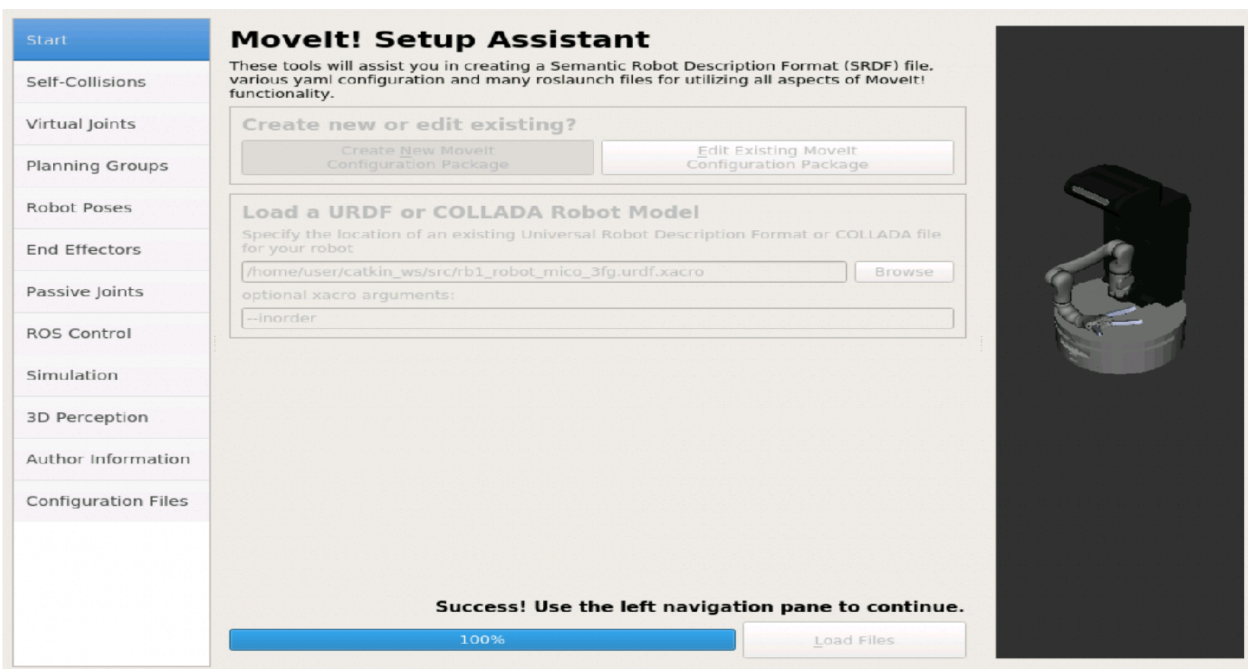
(2) Open “graphical interface”

(3) Click “Create New MoveIt Configuration Package”



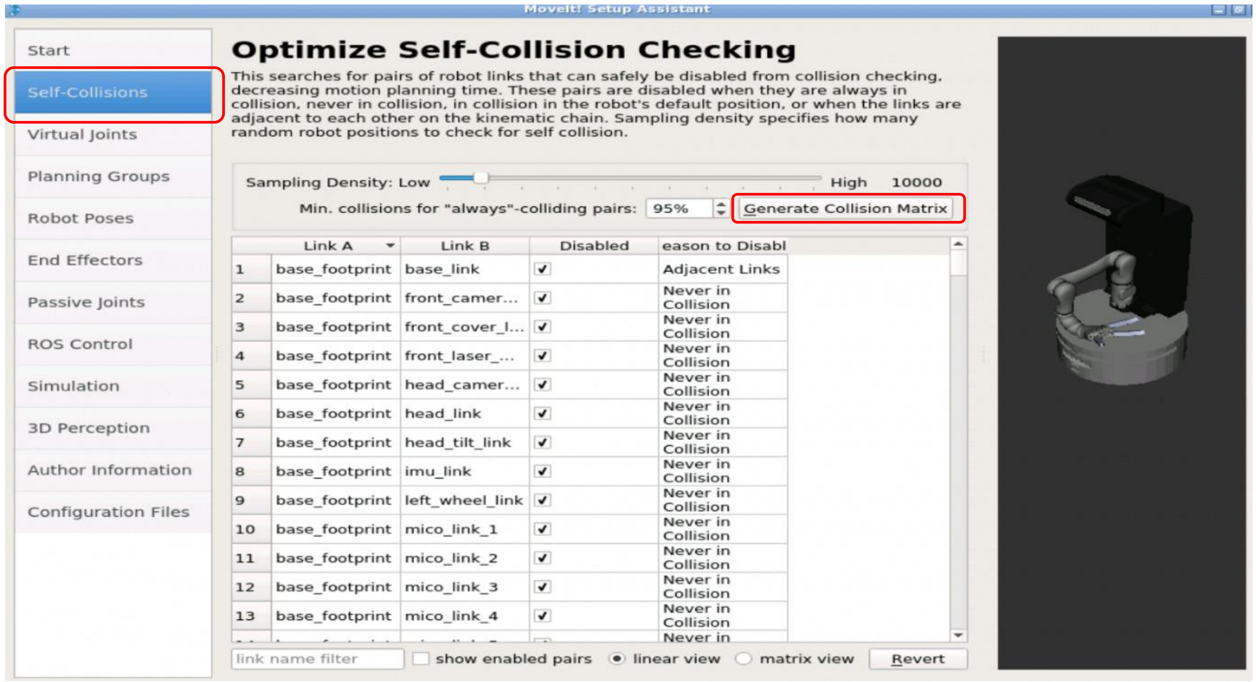
(4) Browse the XACRO File (rb_robot_mico_3fg.urdf.xacro) we create at the previous part and load the file.





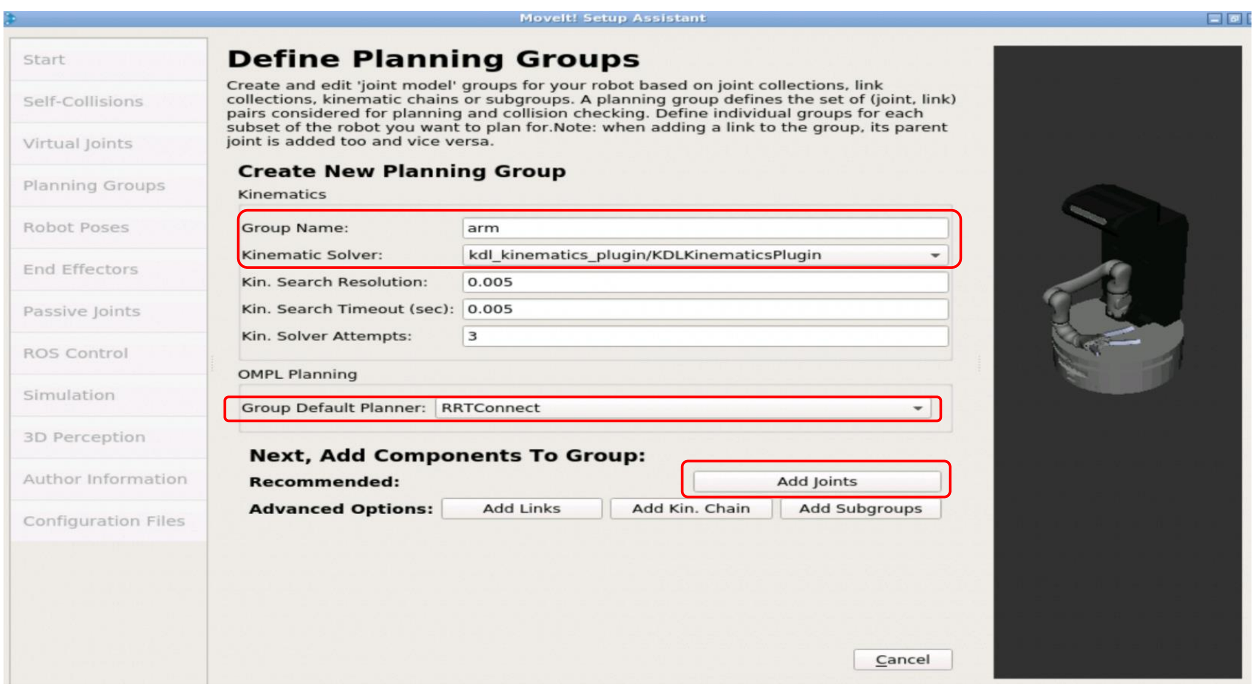
(5) Generate Collision Matrix in Self-Collisions option

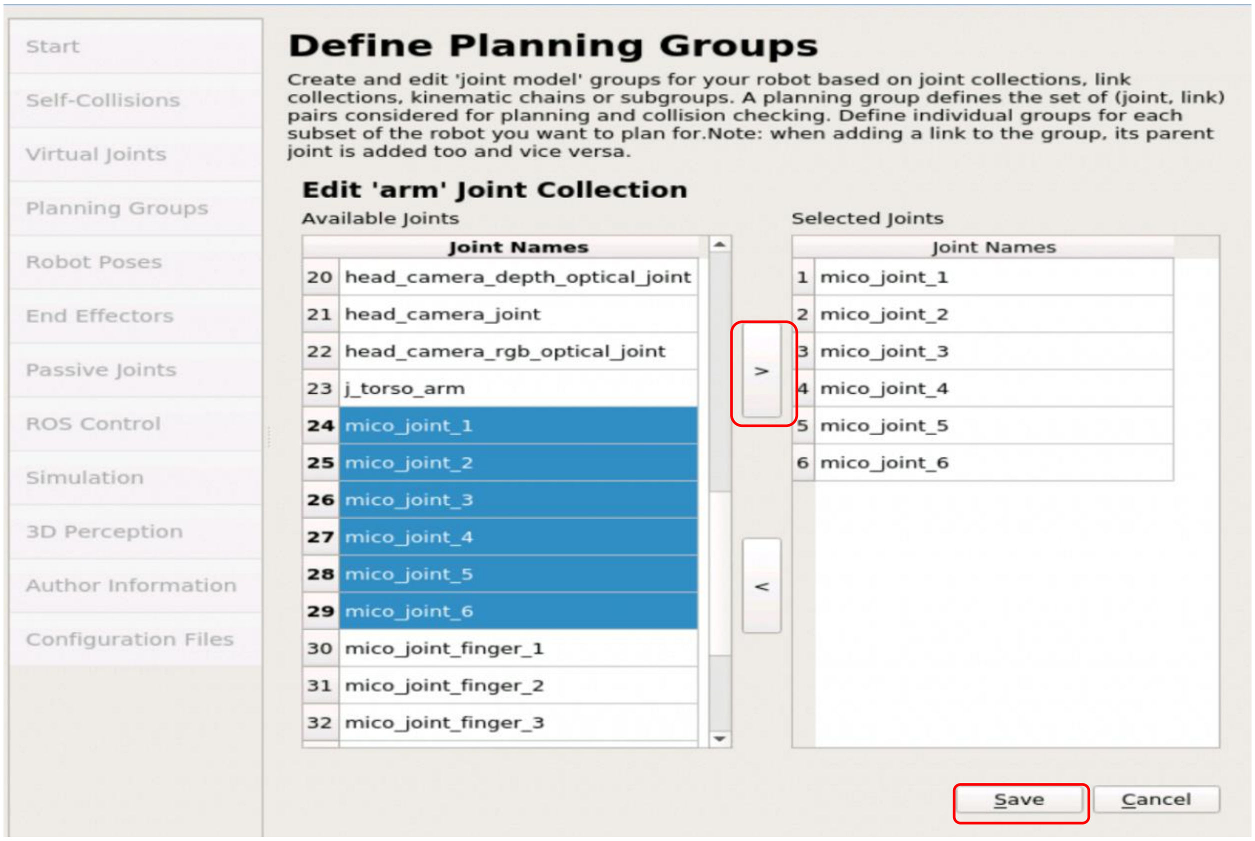
The setup assistant will automatically check self-collisions of all links and joints based on the XACRO file we loaded.

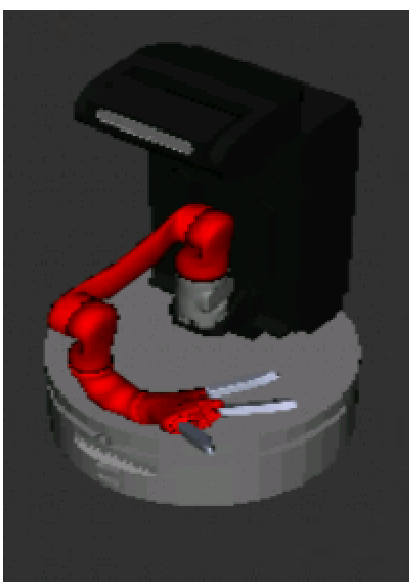


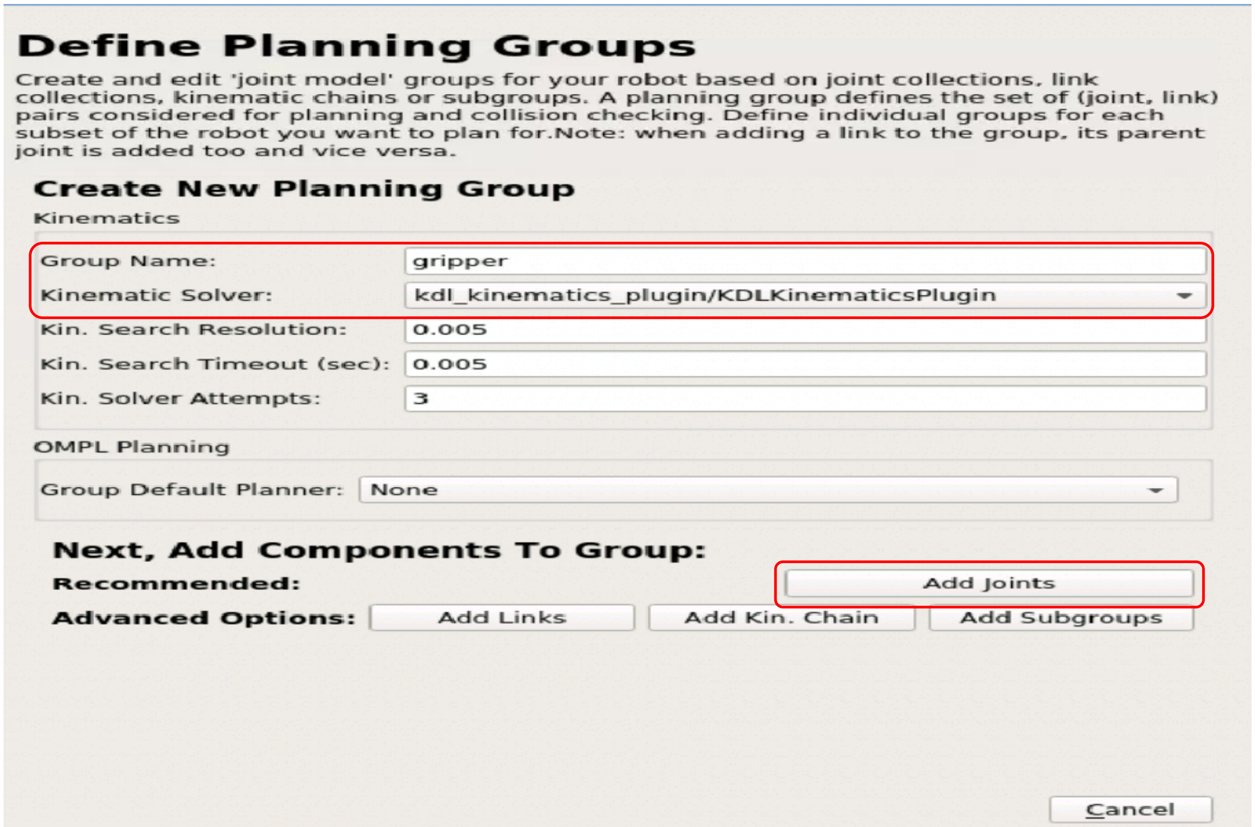
(6) Define Planning Groups.

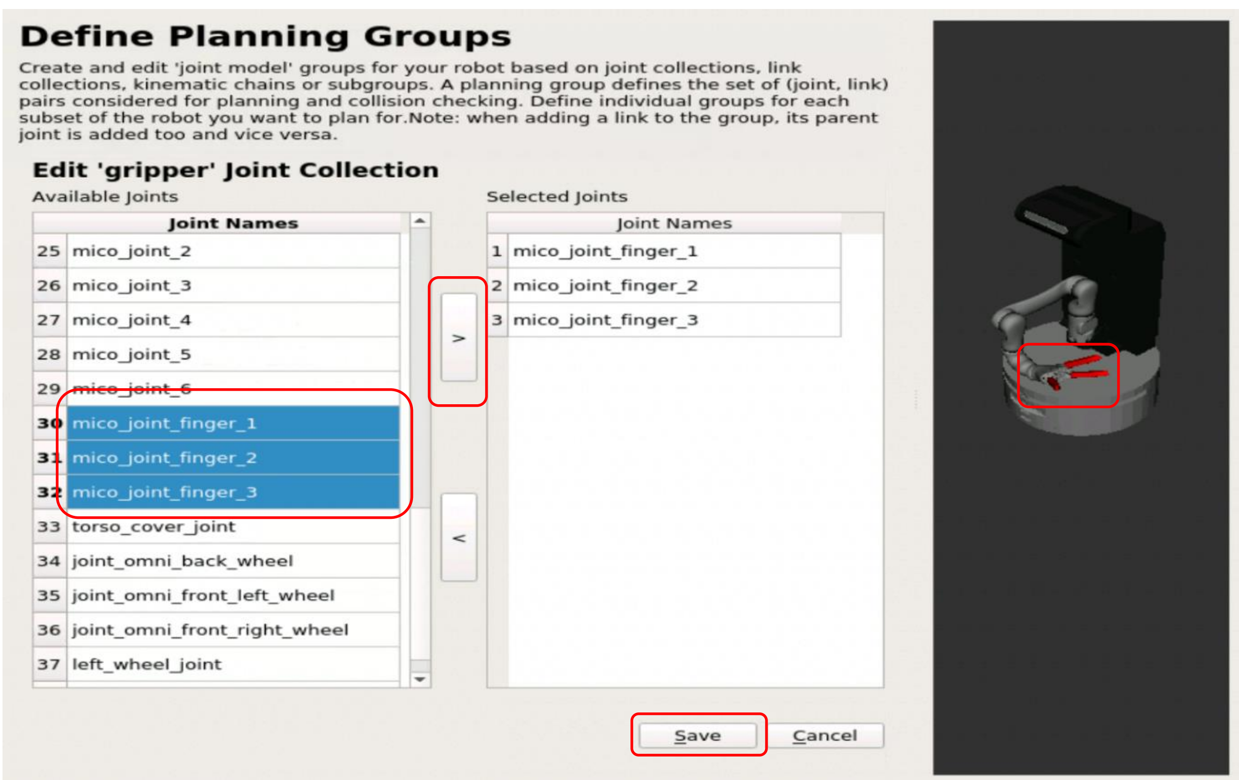
We are defining a set (collection) of joints that will be used for motion planning. In our case, it will be all joints in robotic arm on RB1. You can choose one of planners from OMPL Planning, but this is optional. In the image, RRTConnect planner is chosen.

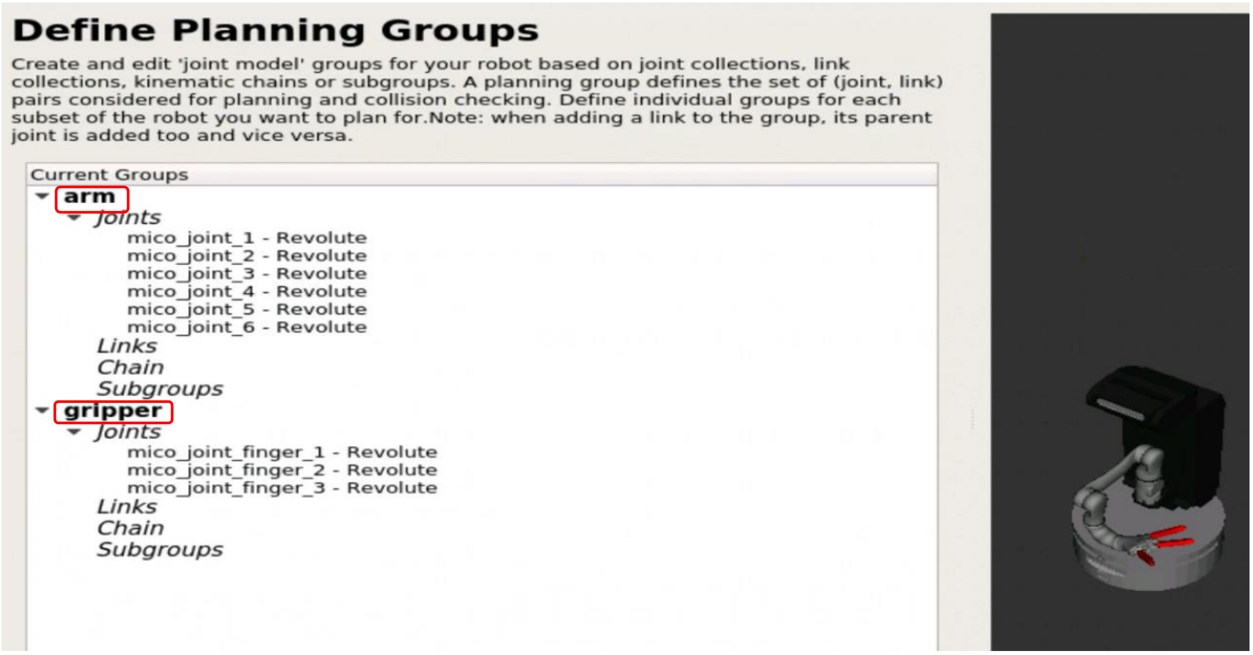



Start
Self-Collisions
Virtual Joints
Planning Groups
Robot Poses
End Effectors
Passive Joints
ROS Control
Simulation
3D Perception
Author Information
Configuration Files
Define Planning Groups
Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the robot you want to plan for.Note: when adding a link to the group, its parent joint is added too and vice versa.
Edit 'arm' Joint Collection
Available Joints
Joint Names
20 head_camera_depth_optical_joint
21 head_camera_joint
22 head_camera_rgb_optical_joint
23 j_torso_arm
24 mico_joint_1
25 mico_joint_2
26 mico_joint_3
27 mico_joint_4
28 mico_joint_5
29 mico_joint_6
30 mico_joint_finger_1
31 mico_joint_finger_2
32 mico_joint_finger_3
Selected Joints
Joint Names
1 mico_joint_1
2 mico_joint_2
3 mico_joint_3
4 mico_joint_4
5 mico_joint_5
6 mico_joint_6
Save
Cancel



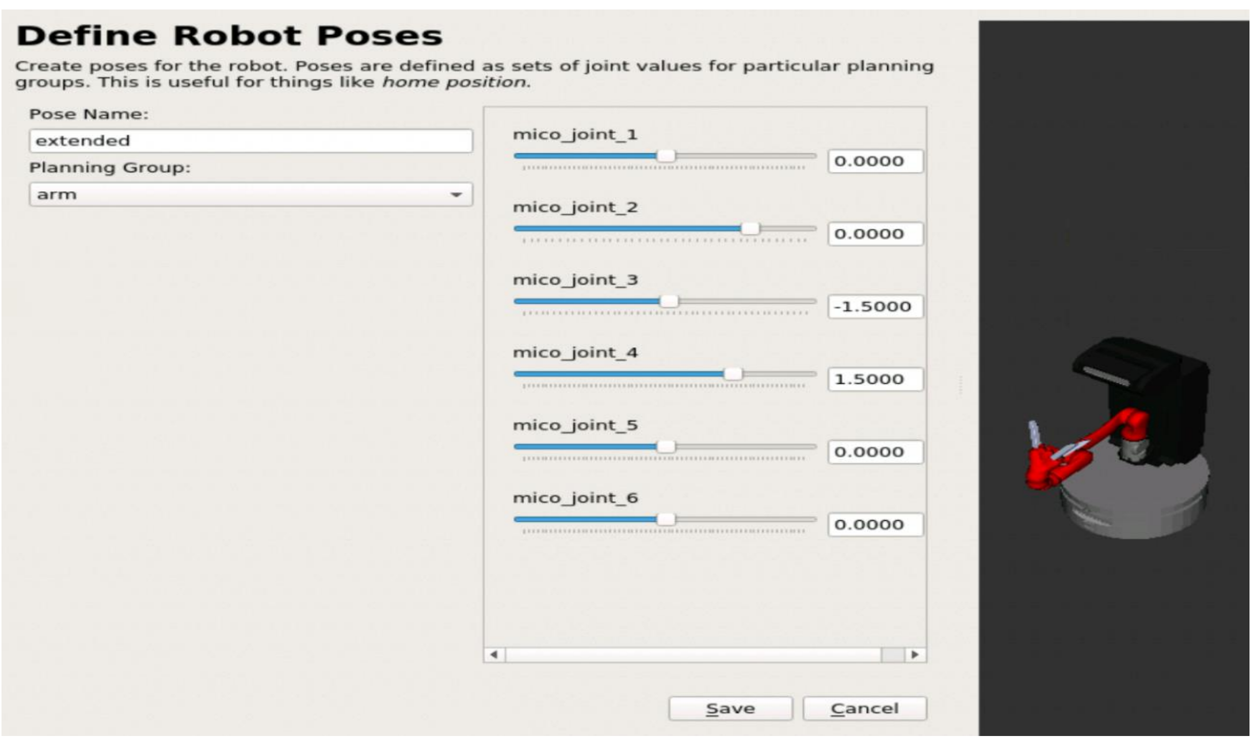

Define Planning Groups
Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the robot you want to plan for.Note: when adding a link to the group, its parent joint is added too and vice versa.
Create New Planning Group
Kinematics
Group Name: gripper
Kinematic Solver: kdl_kinematics_plugin/KDLKinematicsPlugin
Kin. Search Resolution: 0.005
Kin. Search Timeout (sec): 0.005
Kin. Solver Attempts: 3
OMPL Planning
Group Default Planner: None
Next, Add Components To Group:
Recommended: Add Joints
Advanced Options: Add Links Add Kin. Chain Add Subgroups
Cancel

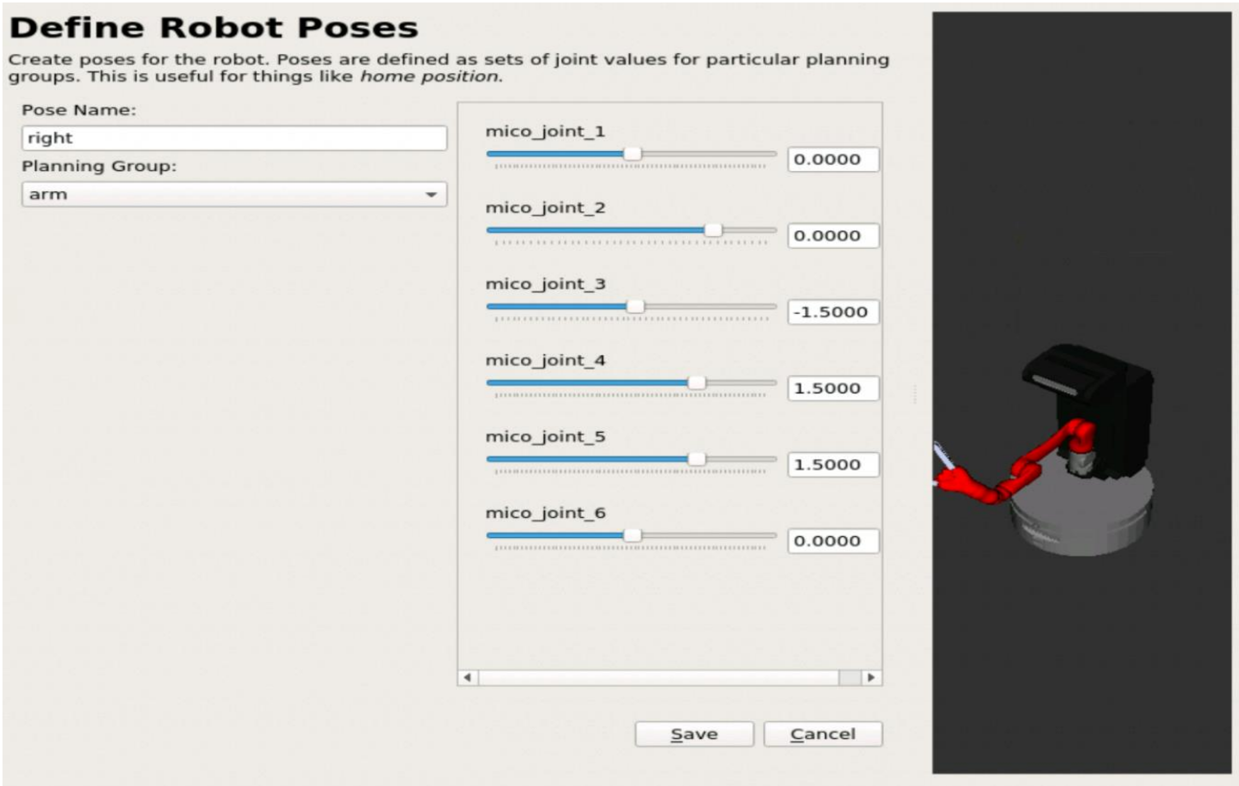

Define Planning Groups
Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the robot you want to plan for.Note: when adding a link to the group, its parent joint is added too and vice versa.
Edit 'gripper' Joint Collection
Available Joints
Joint Names
25 mico_joint_2
26 mico_joint_3
27 mico_joint_4
28 mico_joint_5
29 mico_joint_6
30 mico_joint_finger_1
31 mico_joint_finger_2
32 mico_joint_finger_3
33 torso_cover_joint
34 joint_omni_back_wheel
35 joint_omni_front_left_wheel
36 joint_omni_front_right_wheel
37 left_wheel_joint
Selected Joints
Joint Names
1 mico_joint_finger_1
2 mico_joint_finger_2
3 mico_joint_finger_3
Save
Cancel

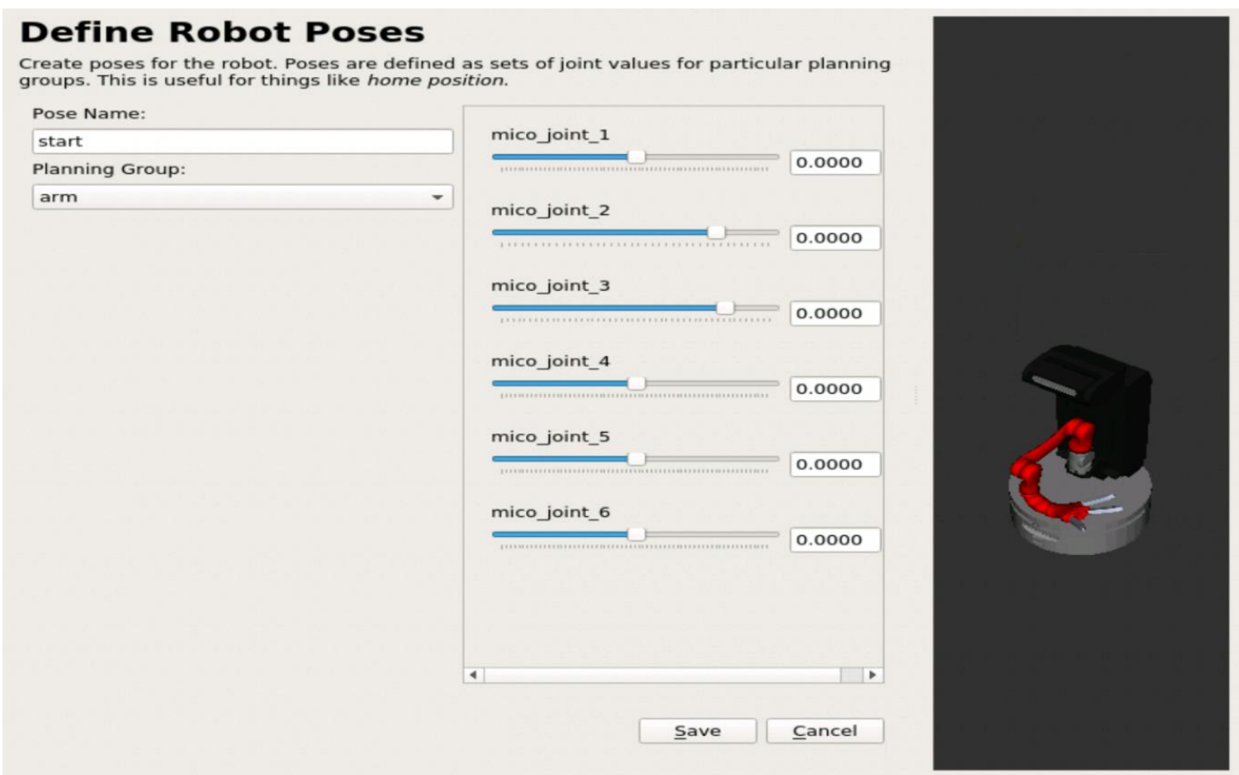


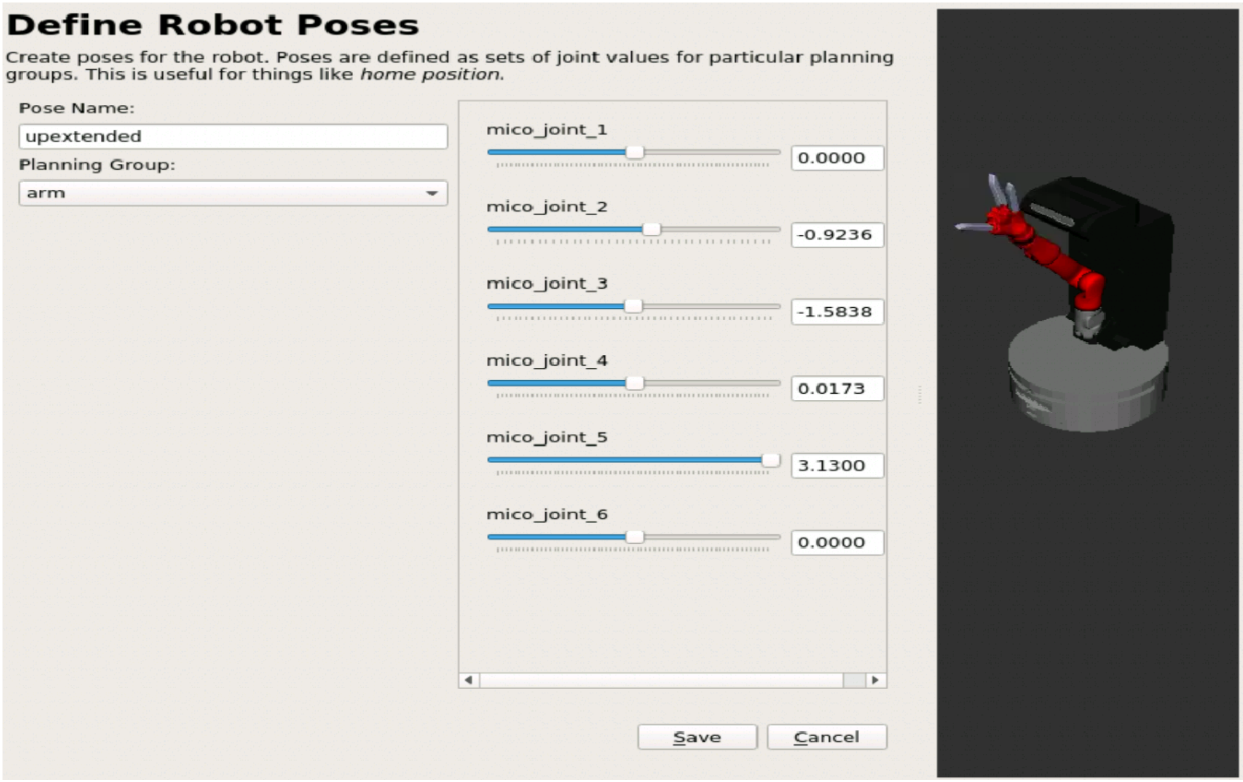
(7) Pre-define robot poses

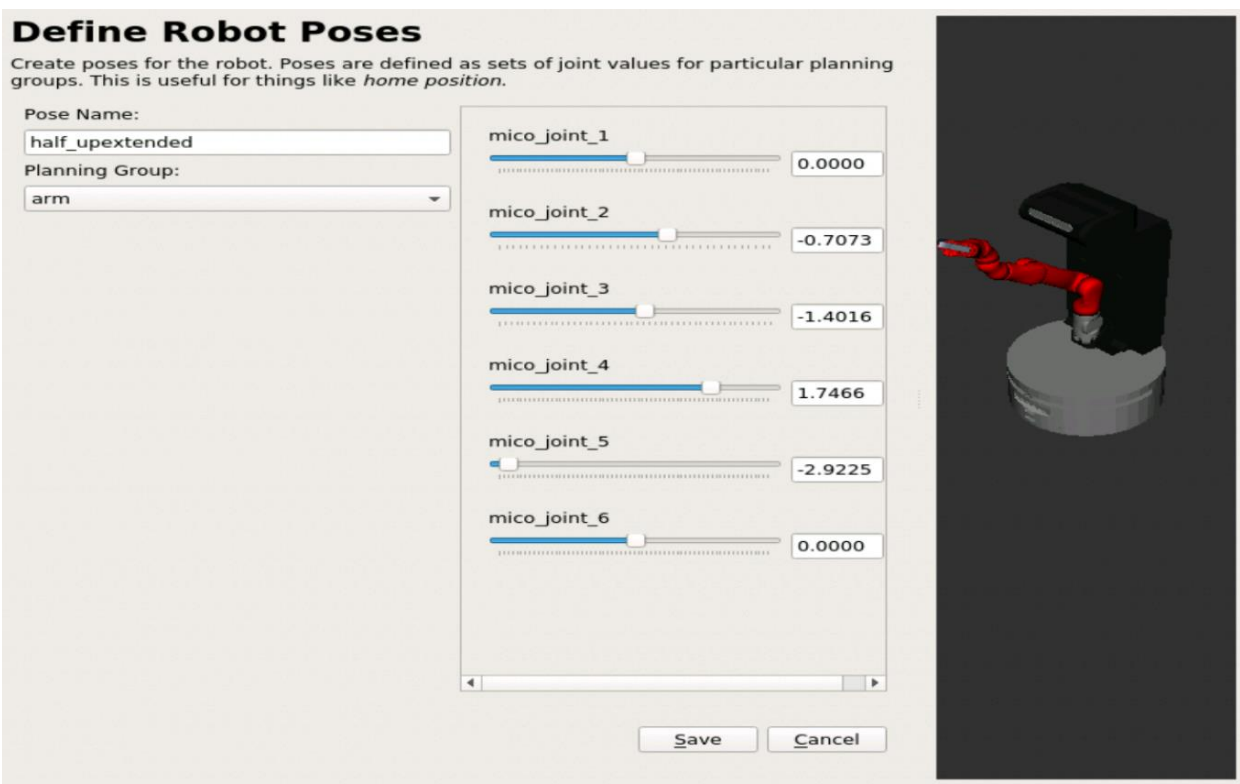
MoveIt setup assistant tool provides a way we can define robot poses ahead by name. We will define 5 poses such as extended, right, start, half_upextended and upextended for the arm and open, and grip for the gripper.

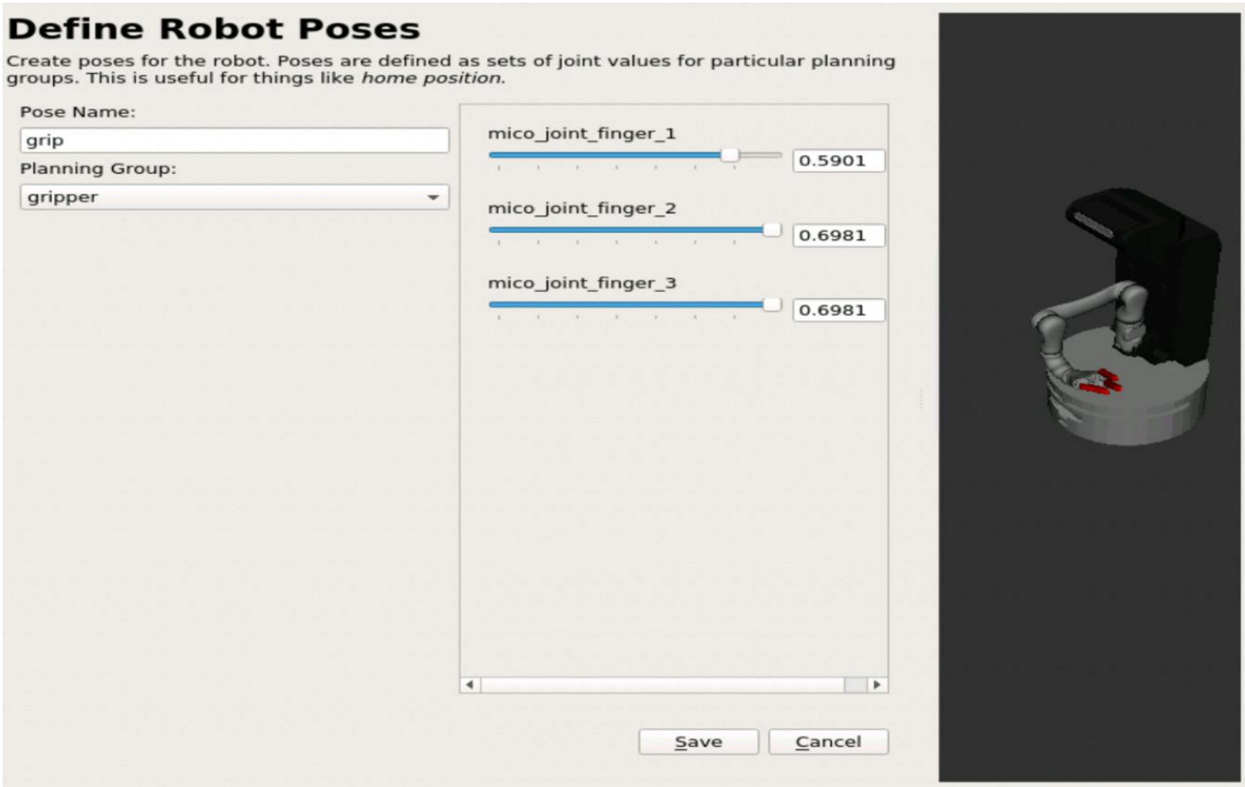


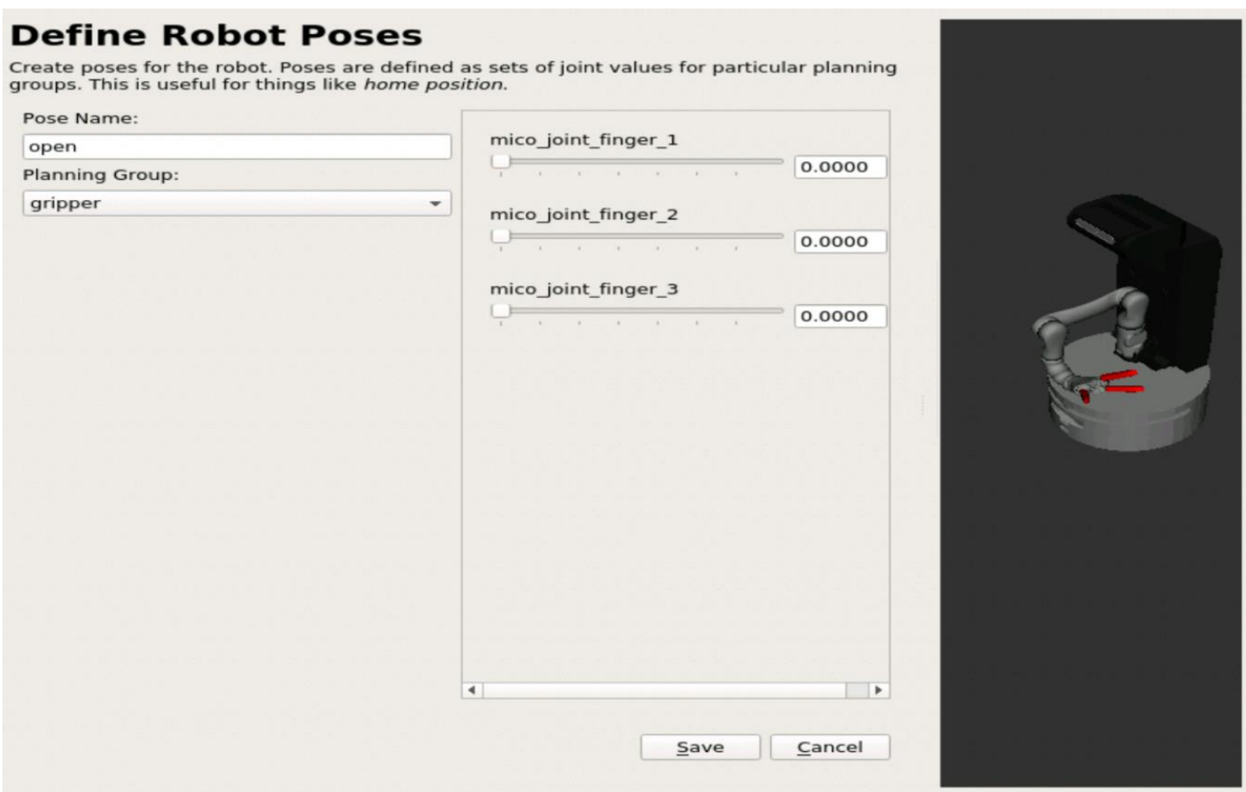

Define Robot Poses
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like home position.
Pose Name:
right
Planning Group:
arm
mico_joint_1
0.0000
mico_joint_2
0.0000
mico_joint_3
-1.5000
mico_joint_4
1.5000
mico_joint_5
1.5000
mico_joint_6
0.0000
Save
Cancel

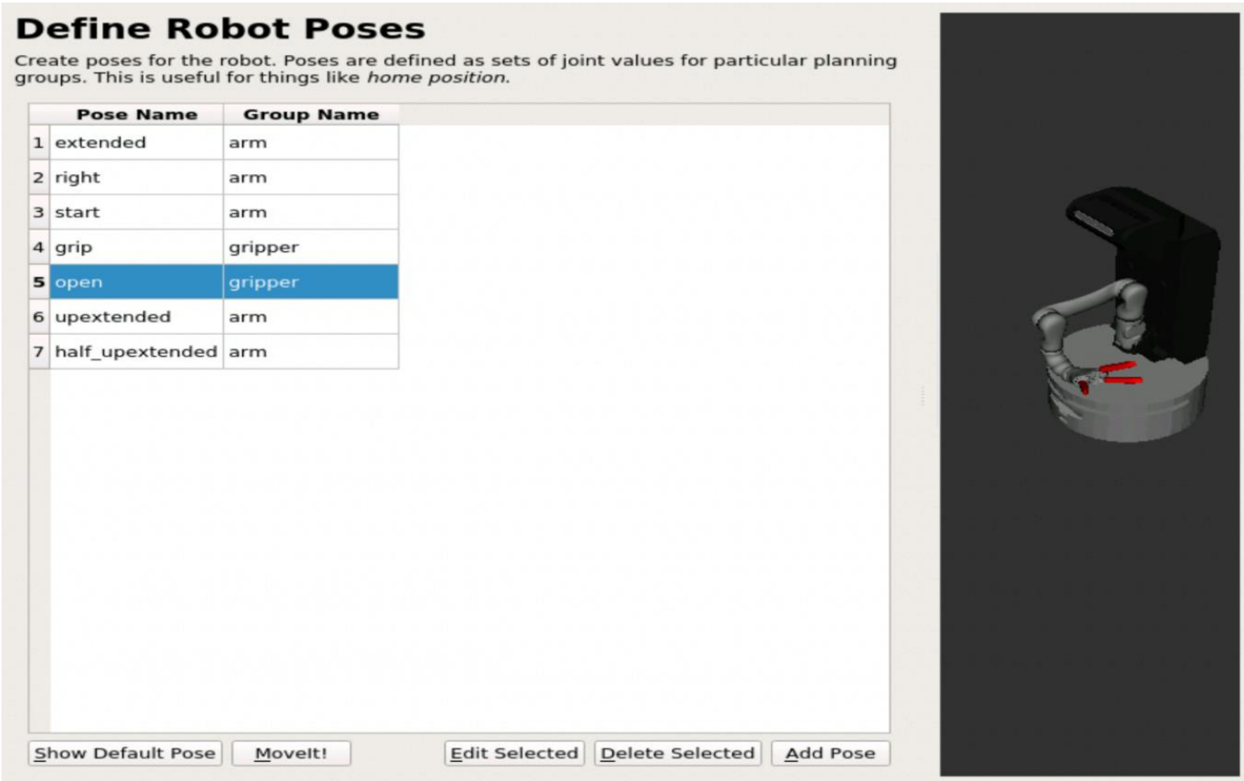

Define Robot Poses
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like home position.
Pose Name:
start
Planning Group:
arm
mico_joint_1
0.0000
mico_joint_2
0.0000
mico_joint_3
0.0000
mico_joint_4
0.0000
mico_joint_5
0.0000
mico_joint_6
0.0000
Save
Cancel


Define Robot Poses
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like *home position.*
Pose Name:
upextended
Planning Group:
arm
mico_joint_1
0.0000
mico_joint_2
-0.9236
mico_joint_3
-1.5838
mico_joint_4
0.0173
mico_joint_5
3.1300
mico_joint_6
0.0000
Save
Cancel

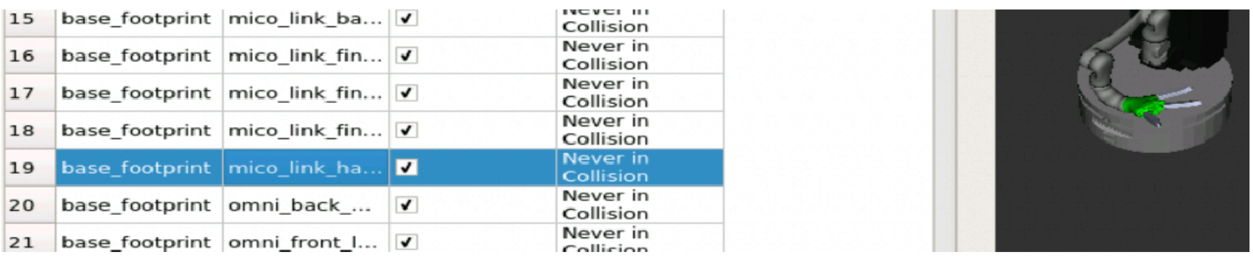

Define Robot Poses
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like *home position.*
Pose Name:
half_upextended
Planning Group:
arm
mico_joint_1
0.0000
mico_joint_2
-0.7073
mico_joint_3
-1.4016
mico_joint_4
1.7466
mico_joint_5
-2.9225
mico_joint_6
0.0000
Save
Cancel

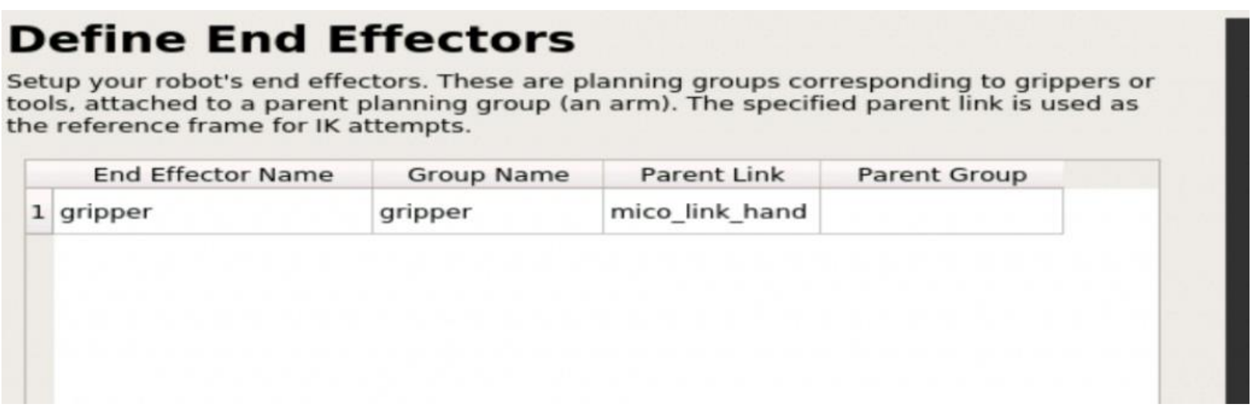

Define Robot Poses
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like home position.
Pose Name:
grip
Planning Group:
gripper
mico_joint_finger_1
0.5901
mico_joint_finger_2
0.6981
mico_joint_finger_3
0.6981
Save
Cancel


Define Robot Poses
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like home position.
Pose Name:
open
Planning Group:
gripper
mico_joint_finger_1
0.0000
mico_joint_finger_2
0.0000
mico_joint_finger_3
0.0000
Save
Cancel

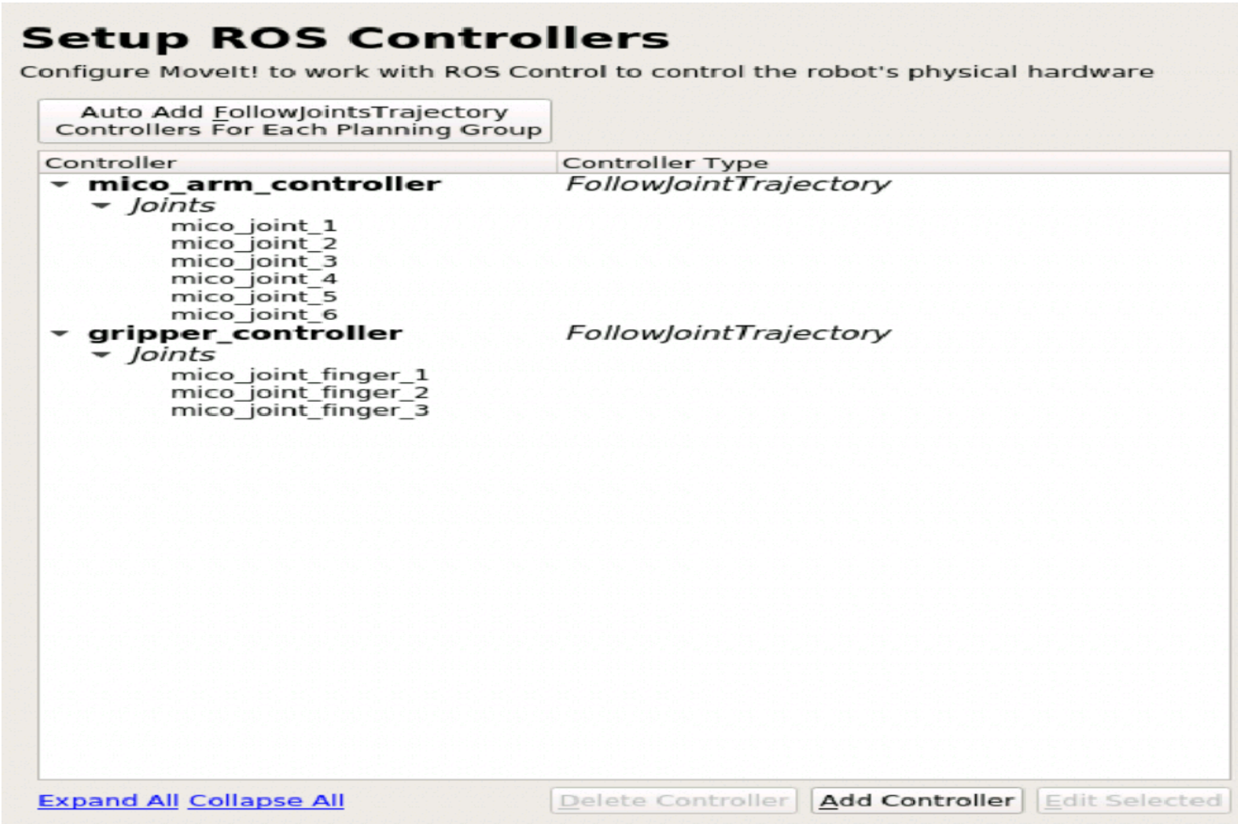


(8) Define end effectors





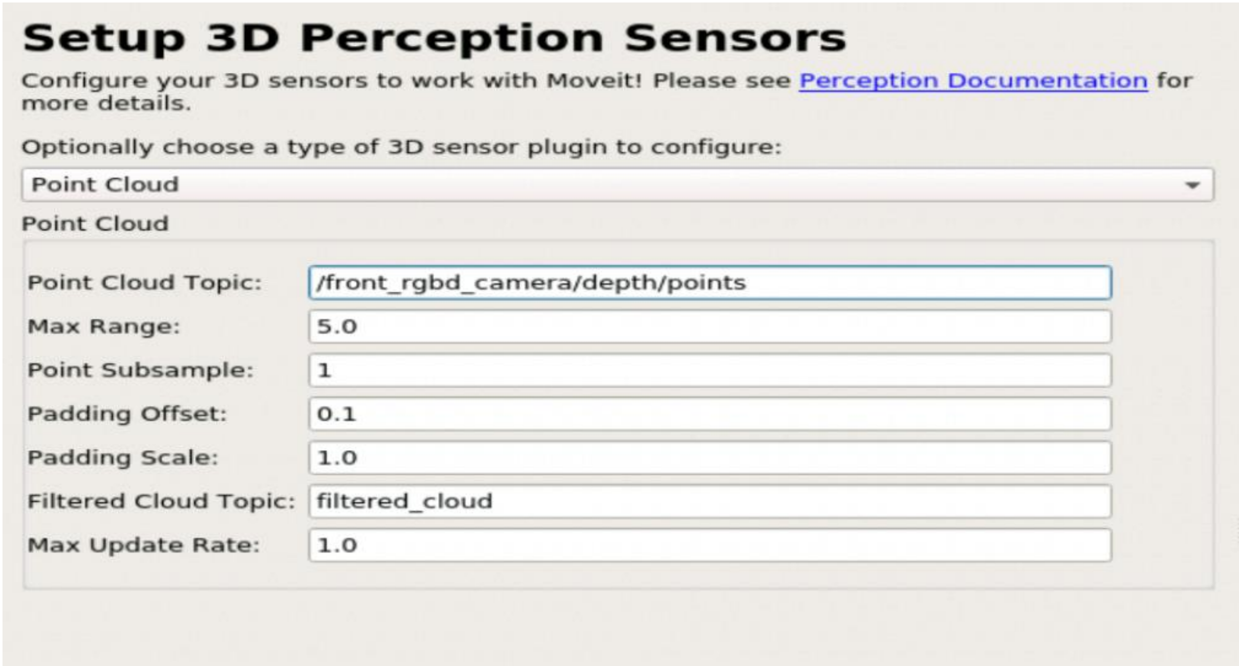
(9) Setup ROS Controllers



(10) Setup 3D Perception Sensors

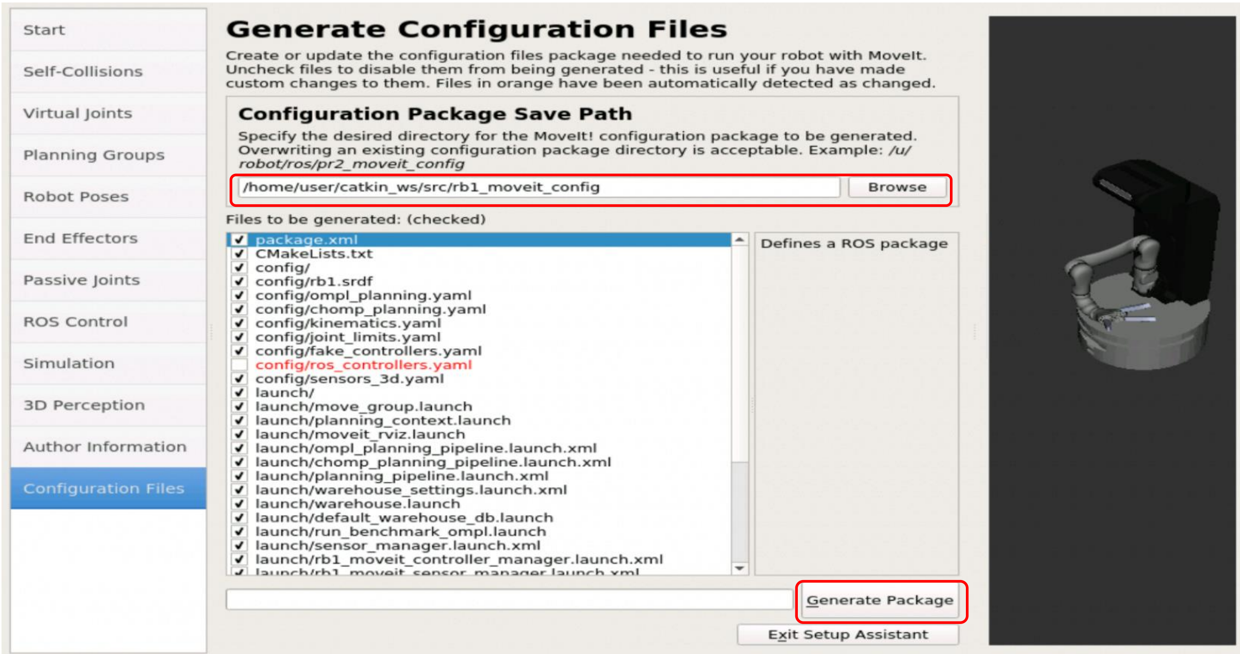
Note: this RGB-D camera will not be employed in RB1 case, but Fetch robot will make use of this sensor.

```
/e_stop
/front_rgbd_camera/depth/camera_info
/front_rgbd_camera/depth/image_raw
/front_rgbd_camera/depth/points
/front_rgbd_camera/parameter_descriptions
/front_rgbd_camera/parameter_updates
/front_rgbd_camera/rgb/camera_info
/front_rgbd_camera/rgb/image_raw
/front_rgbd_camera/rgb/image_raw/compressed
/front_rgbd_camera/rgb/image_raw/compressed/parameter_descriptions
/front_rgbd_camera/rgb/image_raw/compressed/parameter_updates
/front_rgbd_camera/rgb/image_raw/compressedDepth
/front_rgbd_camera/rgb/image_raw/compressedDepth/parameter_descriptions
/front_rgbd_camera/rgb/image_raw/compressedDepth/parameter_updates
```



(11) Specify author information

(12) Generate Configuration Files



<2. Connect the MoveIt! to Gazebo Simulation>

Now, it is time to connect the MoveIt package, "rb1_movetit_config" so that motions we will make with ROS and Python script will appear in simulation (or a real robot). This part is also a step by step procedure. This procedure is almost completed because we already added controllers with MoveIt! Setup Assistant. However, there is one more thing we have to do. MoveIt! Setup Assistant does not automatically generate the right controller name in ros_controllers.yaml file. Therefore, it is our duty to correct this part properly.

```
/rb1/mico_arm_controller/command
/rb1/mico_arm_controller/follow_joint_trajectory/ca
ncel
/rb1/mico_arm_controller/follow_joint_trajectory/fe
edback
/rb1/mico_arm_controller/follow_joint_trajectory/go
al
/rb1/mico_arm_controller/follow_joint_trajectory/re
sult
/rb1/mico_arm_controller/follow_joint_trajectory/st
atus
```

```
/rb1/gripper_controller/follow_joint_trajectory/st
tus
/rb1/gripper_controller/gains/mico_joint_finger_1/
arameter_descriptions
/rb1/gripper_controller/gains/mico_joint_finger_1/
arameter_updates
/rb1/gripper_controller/gains/mico_joint_finger_2/
arameter_descriptions
/rb1/gripper_controller/gains/mico_joint_finger_2/
arameter_updates
```

When you check the names of topics you will see “rb1/” is there. So, we need to ad “rb1/” before “arm_controller”

e.g. add “rb1” to the name tags for arm_controller and gripper_controller

```
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50
controller_list:
  - name: arm_controller
    action_ns: follow_joint_trajectory
    default: True
    type: FollowJointTrajectory
    joints:
      - mico_joint_1
      - mico_joint_2
      - mico_joint_3
      - mico_joint_4
      - mico_joint_5
      - mico_joint_6
  - name: gripper_controller
    action_ns: follow_joint_trajectory
    default: True
    type: FollowJointTrajectory
    joints:
      - mico_joint_finger_1
      - mico_joint_finger_2
      - mico_joint_finger_3
```

```
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50
controller_list:
  - name: rb1/arm_controller
    action_ns: follow_joint_trajectory
    default: True
    type: FollowJointTrajectory
    joints:
      - mico_joint_1
      - mico_joint_2
      - mico_joint_3
      - mico_joint_4
      - mico_joint_5
      - mico_joint_6
  - name: rb1/gripper_controller
    action_ns: follow_joint_trajectory
    default: True
    type: FollowJointTrajectory
    joints:
      - mico_joint_finger_1
      - mico_joint_finger_2
      - mico_joint_finger_3
```

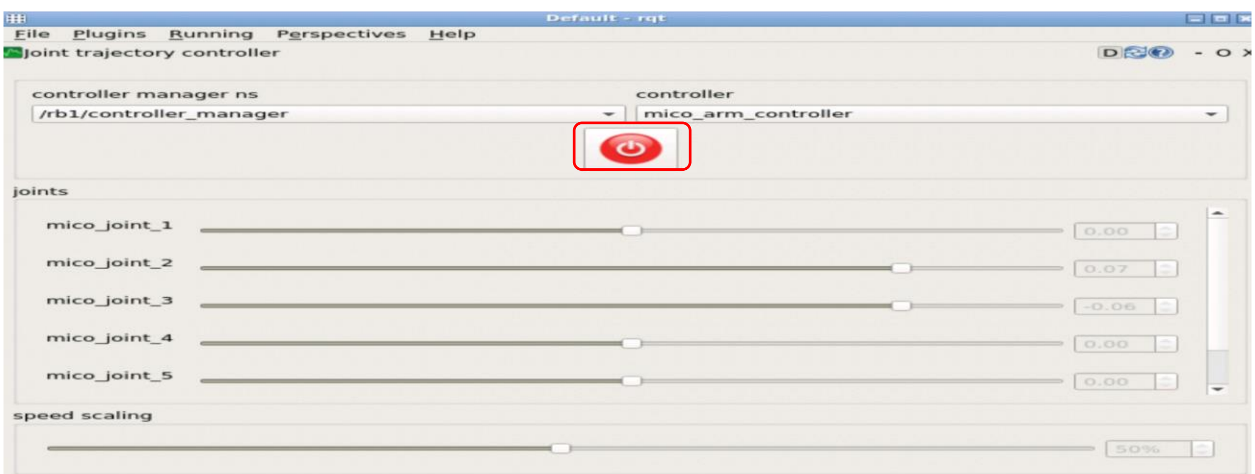
Furthermore, we need do some changes in “rb1_planning_execution.launch” file so that joint state publisher can publish joint states into a proper topic.

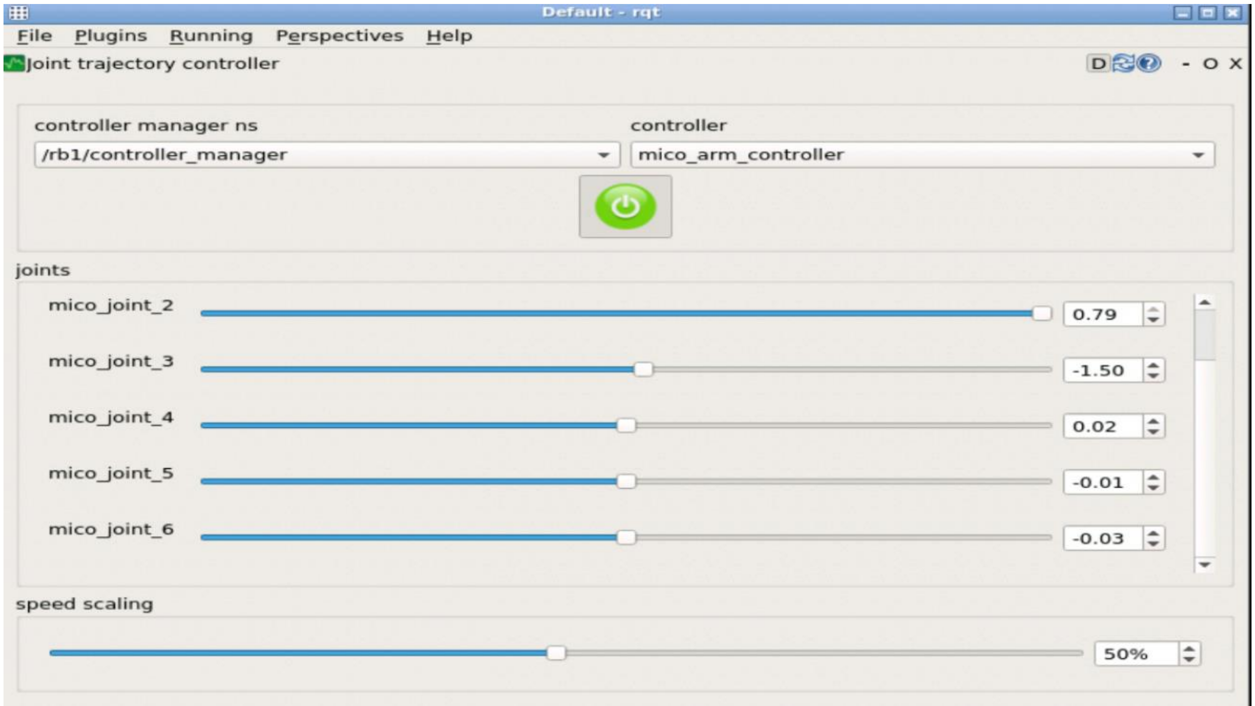
```
user:~$ rostopic list | grep joint_states
/rb1/joint_states
```

```
trollers.yaml    rb1_planning_execution.launch ×
<launch>

  <include file="$(find rb1_moveit_config)/launch/planning_context.launch" >
    <arg name="load_robot_description" value="true" />
  </include>

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
    <param name="/use_gui" value="false"/>
    <rosparam param="/source_list">[/rb1/joint_states]</rosparam>
  </node>

  <include file="$(find rb1_moveit_config)/launch/move_group.launch">
    <arg name="publish_monitored_planning_scene" value="true" />
  </include>

  <include file="$(find rb1_moveit_config)/launch/moveit_rviz.launch">
    <arg name="config" value="true"/>
  </include>

</launch>
```

Now, we need to run rqt then follow the process below.

“Plugins” -> “Robot Tools” -> “Jointrajectory controller” -> click “Enable/disable sending commands to the controller” button.





Now, we are ready to write up some Python script for the given task.

<3. Write up a Python script>

Given task is the following.

-Grab a cube on a table.

-Move around the cube over the table

-Put it back onto the table

Note: Since RGB-D camera is not functioning properly in the simulation on Robot Ignite Academy, performing grasping with perception is somewhat difficult. Therefore, in this project, we will set our task to forward Kinematic problem meaning we assume the location of target of interest is known beforehand. However, it is still worthwhile to recall simple pipeline of grasping task with perception.

-Obtain the location of the cube using point cloud data processing from RGB-D camera.

-Calculate proper position and orientation of end effector accordingly.

-Perform motion planning.

-Execute the calculated motion plan.

-Grab the object

-Do given tasks

In this project, the target's location is given in advance. Therefore, we can approach the task with combination of pre-defined poses and "set_pose_target" given the position of end effector. This position of the end effector is hard coded. Let's go through the Python Script.

-Part 1.

```
import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
```

We are importing necessary modules to perform the given motion planning task. The most important module is moveit_commander. The most of objects for the task will be created from this module.

-Part 2.

```
moveit_commander.roscpp_initialize(sys.argv)
```

We are initializing moveit_commander module.

-Part 3.

```
rospy.init_node('move_group_python_interface_tutorial', anonymous=True)
```

We are initializing "move_group_python_interface_tutorial" node.

-Part 4.

```
robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
group = moveit_commander.MoveGroupCommander("arm")
group2 = moveit_commander.MoveGroupCommander("gripper")
```

Here, three objects are created. “robot” object is for communicating with robot. “scene” object has something to do with the world around RB1. Then, most importantly, “group” and “group2” objects (one for arm and one for gripper respectively) are in charge of setting up target pose, motion planning, and executing and gripping task respectively.

-Part 5.

```
display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path', moveit_msgs.msg.DisplayTrajectory)
```

We are publishing a message (type: “DisplayTrajectory”) to the topic “move_group/display_planned_path” so that we can observe the calculated motion plan in RVIZ.

-Part 6.

From part 1 to part 5, it is basic setup for performing the task. Here, our script is actually performing the grasping task.

Get ready to approach the cube from the top.

```
#Forward Kinematic Grasping Solution.
group.set_named_target("start")
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)

group2.set_named_target("open")
group2.go(wait=True)
rospy.sleep(1)

group.set_named_target("upextended")
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)
```

Approach the cube.

```
pose_target = geometry_msgs.msg.Pose()
#pose_target.orientation.w = 1.0
pose_target.position.x = 0.46
pose_target.position.y = -0.04
pose_target.position.z = 0.58
#group.set_named_target("start")
group.set_pose_target(pose_target)
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)
```

Grab the cube and take different poses such as “upextended”, and “start”.

```
group2.set_named_target("grip")
group2.go(wait=True)
rospy.sleep(1)

group.set_named_target("upextended")
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)


group.set_named_target("start")
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)

group.set_named_target("upextended")
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)
```

Go back to where the cube belonged to and locate it into the original place.

```
pose_target = geometry_msgs.msg.Pose()
#pose_target.orientation.w = 1.0
pose_target.position.x = 0.5
pose_target.position.y = -0.04
pose_target.position.z = 0.59
#group.set_named_target("start")
group.set_pose_target(pose_target)
plan2 = group.plan()
group.go(wait=True)
rospy.sleep(1)

group2.set_named_target("open")
group2.go(wait=True)
rospy.sleep(1)
```

Set the robot arm to the very beginning position.

```
group.set_named_target("upextended")
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)


group.set_named_target("start")
plan1 = group.plan()
group.go(wait=True)
rospy.sleep(1)
```

Shut down the moveit_commander.

```
moveit_commander.roscpp_shutdown()
```

Note: please, refer to ReadMe file to run this program.

<Bonus objective: grasping with perception>

In this practice, most steps are quite similar to what we have done with RB1. However, since RGB-D camera works properly, we can get some help from it for our grasping task when it comes to object detection and obtaining its location. However, perception part will be revisited in other perception projects. Package that is responsible for obtaining the location of the cube and adjusting gripper is developed by Mike Ferguson. Then, more changes have been done by Robot Ignite Academy. We only modified a few lines of codes to fix robot's behavior while performing the grasping task.

To do so, we need to add gripper_controller and its type properly in "ros_controllers.yaml" in "config" directory in "fetch_moveit_config" package.

e.g.)

```yaml
      - forearm_roll_joint
      - wrist_flex_joint
      - wrist_roll_joint
      - l_gripper_finger_joint
      - r_gripper_finger_joint
    sim_control_mode: 1  # 0: position, 1: velocity
  # Publish all joint states
  # Creates the /joint_states topic necessary in ROS
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50
  controller_list:
    - name: arm_controller
      action_ns: follow_joint_trajectory
      default: True
      type: FollowJointTrajectory
      joints:
        - shoulder_pan_joint
        - shoulder_lift_joint
        - upperarm_roll_joint
        - elbow_flex_joint
        - forearm_roll_joint
        - wrist_flex_joint
        - wrist_roll_joint
    - name: gripper_controller
      action_ns: gripper_action
      type: GripperCommand
      default: true
      joints:
        - l_gripper_finger_joint
        - r_gripper_finger_joint
```

It has another issue with gripper's behavior that width of its fingers became too narrow right before it attempted to grasp the detected object (i.e., a cube)

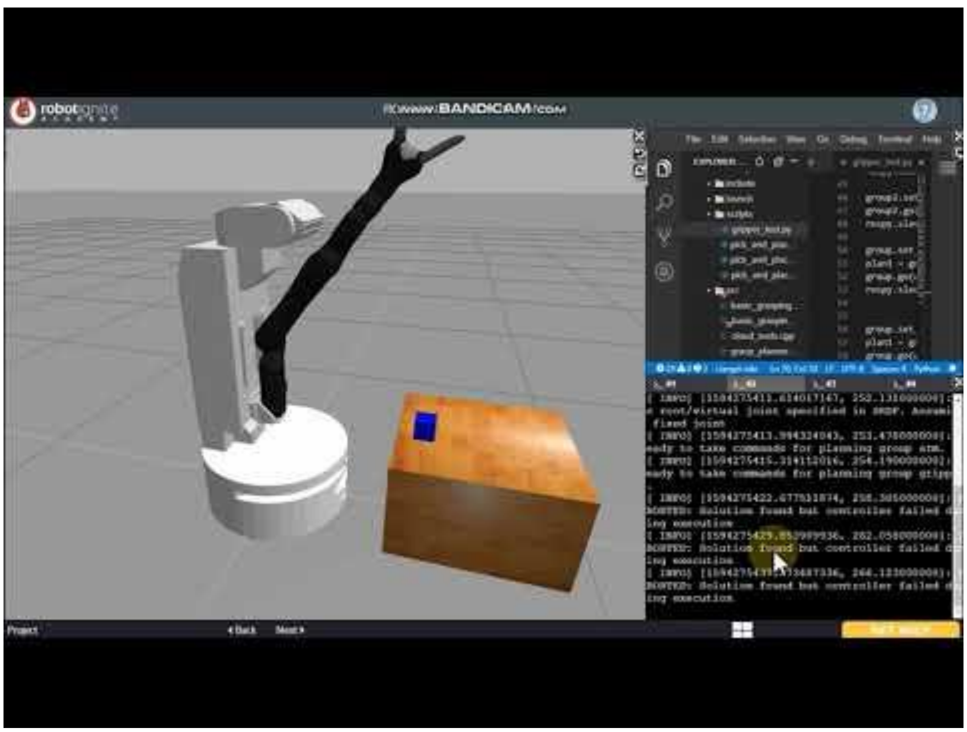
Therefore, I had to investigate and change the code in order to prevent this behavior.

We modified a line of code in "shape_grasp_planner.cpp" in "simple_grasping" package.

e.g.

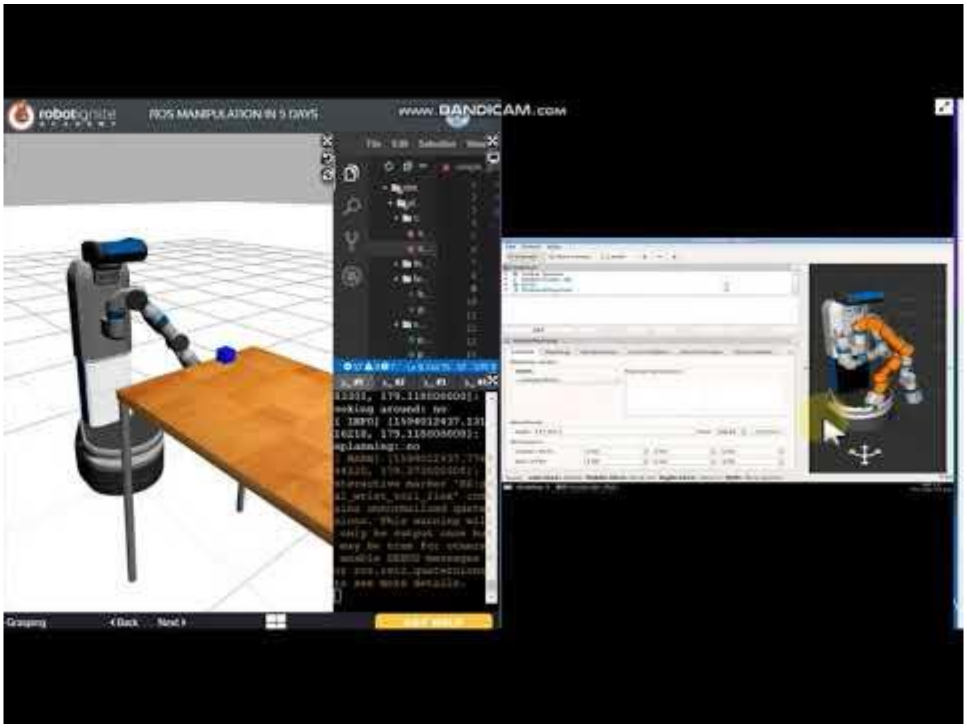
```
//double open = std::min(width + gripper_tolerance_, max_opening_);
double hardcodedNumber = 0.11;
double open = std::max(width + hardcodedNumber, max_opening_);
```

1.3 Experimental Results



Video 1. Grasping with RB1

https://youtu.be/Y9DIHNxFbbI



Video 2. Objection Detecting and Grasping with Fetch Robot (Bonus)

https://youtu.be/y1fcRCPzPnA

1.4 Discussion

Even though, we can perform manipulation tasks without knowing how planners work, it is worthwhile to understand basic concepts of planners. In this report, we will take a look at RRT-Connection which is one of commonly used planners in a grasping task.

Since RRT-Connection planner is developed based on RRT (Rapidly-Exploring Random Tree) planner, we can start with RRT planner. As it is illustrated in Figure 1.4.1, this planner has two main algorithms called BUILD_RRT and EXTEND respectively. In BUILD_RRT algorithm,

it generates position and orientation (a.k.a q) at random. Then, it performs EXTEND algorithm repeatedly until certain points K. Here, $T \ and \ q_{rand}$ indicate a current growing tree and randomly generated position and orientation at step k. In EXTEND algorithm, it generates nearest position & orientation based on $q_{rand}$and if the $q_{near}, q_{new} \ and \ q$ have a new configuration the q_{near} & q_{new} will be added properly to the tree as vertex and edge. If the q_{new}reaches our goal, we stop growing the tree. Otherwise, we continue to explore given workspace. [2]

```
BUILD_RRT(q_init)
  T.init(q_init);
  for k = 1 to K do
    q_rand ← RANDOM_CONFIG();
    EXTEND(T, q_rand);
  Return T
```

```
EXTEND(T, q)
  q_near ← NEAREST_NEIGHBOR(q, T);
  if NEW_CONFIG(q, q_near, q_new) then
    T.add_vertex(q_new);
    T.add_edge(q_near, q_new);
    if q_new = q then
      Return Reached;
    else
      Return Advanced;
  Return Trapped
```

Figure 1.4.1: RRT Construction algorithm [2]

On the top of RRT algorithm, RRT-Connection path planner algorithm can be developed. Instead of extending tree from only one side (i.e., start point) The RRT-Connection algorithm grows tree's branches from both sides, start point and target point. To be more specific, let's take a look at RRT-Connect algorithm in figure 1.4.2. In CONNECT part, extension is repeated until S is not advanced status anymore. In RRT_CONNECT_PLANNER, two trees T_a and T_b keep growing through K steps unless one of them gets trapped. Then, the algorithm connects the two trees when one reaches the other's branch. This is where CONNECT algorithm comes to play. An intuitive example of growing two trees with RRT-Connection path planner is shown in figure 1.4.3.

```
CONNECT(T, q)
  repeat
    S ← EXTEND(T, q);
  Until not (S = Advanced)
  Return S;
```

```
RRT_CONNECT_PLANNER(q_init, q_goal)
  T_a.init(q_init); T_b.init(q_goal)
  for k = 1 to K do
    q_rand ← RANDOM_CONFIG();
    if not (EXTEND(T_a, q_rand) = Trapped) then
      if (CONNECT(T_b, q_new) = Reached) then
        Return PATH(T_a, T_b);
    SWAP(T_a, T_b);
  Return Failure
```

Figure 1.4.2: RRT-Connect algorithm.

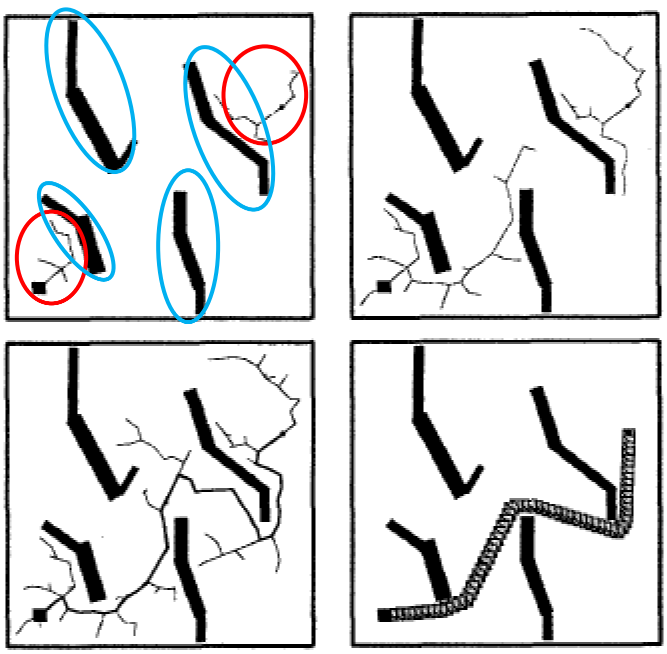


Figure 1.4.3: Growing two trees towards each other. Red circles: two trees, Blue circles: Obstacles. [2]

As we discuss, the RRT-Connect is a very intuitive, but one of powerful planners that is employed not only in Robotics, but also in many other fields such as pharmaceutical drug design and computer animation.

References

[1] Project Contents Credit: Robot Ignite Academy

https://www.robotigniteacademy.com/en/course/ros-manipulation-in-5-days/details/

[2] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, San Francisco, CA, USA, 2000, pp. 995-1001 vol.2, doi: 10.1109/ROBOT.2000.844730. Available Online:

https://ieeexplore.ieee.org/document/844730