Note:

1. This project is from the course, ROS For Industrial Robots 101 on Robot Ignite Academy

: https://www.robotigniteacademy.com/en/course/ROS-for-Industrial-Robots-101/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 For Industrial Robots 101

July 09, 2020

1.0 Objectives

-Create a MoveIt package and Python script which allows UR5 to perform the given task (Please, refer to the link to see what the given task is). [1]

1.1 Abstract and Motivation

Nowadays, we have encountered many industrial robots such as small robotic arms have come to our normal life. Customers can even buy their own robotic arms online at relatively low cost. The one of reasons it happens is ROS MoveIt has reduced many tedious setup steps to develop a practical robotic arm. Therefore, we can also take advantage from it and build our own robotic arm that is capable of motion planning. Furthermore, it is crucial to explore the principle of kinematic solver so that we can have better understanding of how the motion planning works behind the scene.

1.2 Approach, Procedures, and Sample Solution.

In this project, our ultimate goal is to make the UR5 plan and execute the pre-defined motions in specific sequence. In order to achieve this, we can break down the whole process to the 4 major parts.

(1) Write up URDF (Universal Robot Description File) for UR5

(2) Generate MoveIt package with the URDF we write at part 1.

(3) Make connection between the MoveIt package and UR5 in Gazebo Simulation

(4) Write up a Python script that perform motion planning and execution of multiple pre-defined poses.

<1. URDF( or XACRO) for UR5>

Since we are given “ur5.urdf.xacro” file, we can make use of xacro meaning we can create a file that describe UR5 and it contains all joints, links, and controllers we need later on.

e.g. Usage of “ur5.urdf.xacro” file in our own URDF.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro"
       name="ur5" >

  <!-- common stuff -->
  <xacro:include filename="$(find ur_description)/urdf/common.gazebo.xacro" />

  <!-- ur5 -->
  <xacro:include filename="$(find ur_description)/urdf/ur5.urdf.xacro" />

  <!-- arm -->
  <xacro:ur5_robot prefix="" joint_limited="false"/>

  <link name="world" />
```

Note: as long as we have xacro file of a robot of interest, it is very simple to create a customized URDF (or XACRO). Please, take a look at the report (a link provided in reference [2]) if you want to know how to write up simple URDF file for your own robot.

<2. MoveIt Package for UR5>

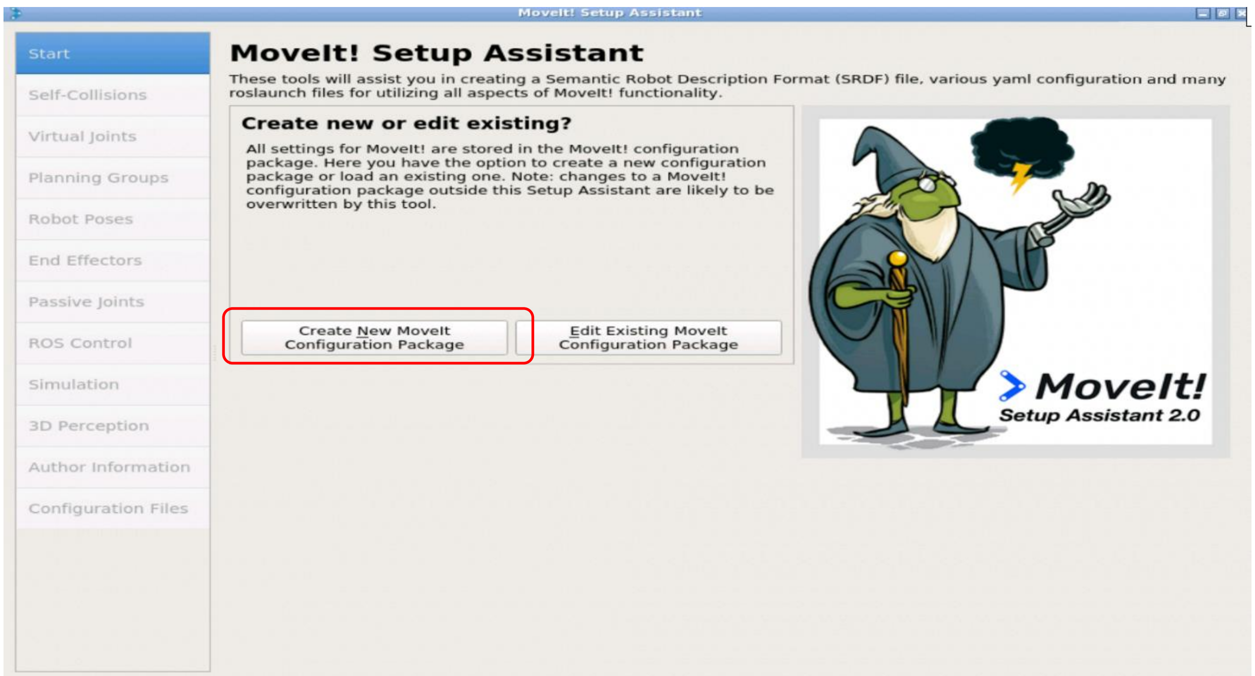
With help of MoveIt setup assistant, we can easily generate MoveIt Package that will be employed for motion planning and execution. This can be more easily explained with step by step procedure with images.

(1) Launch MoveIt Setup Assistant.

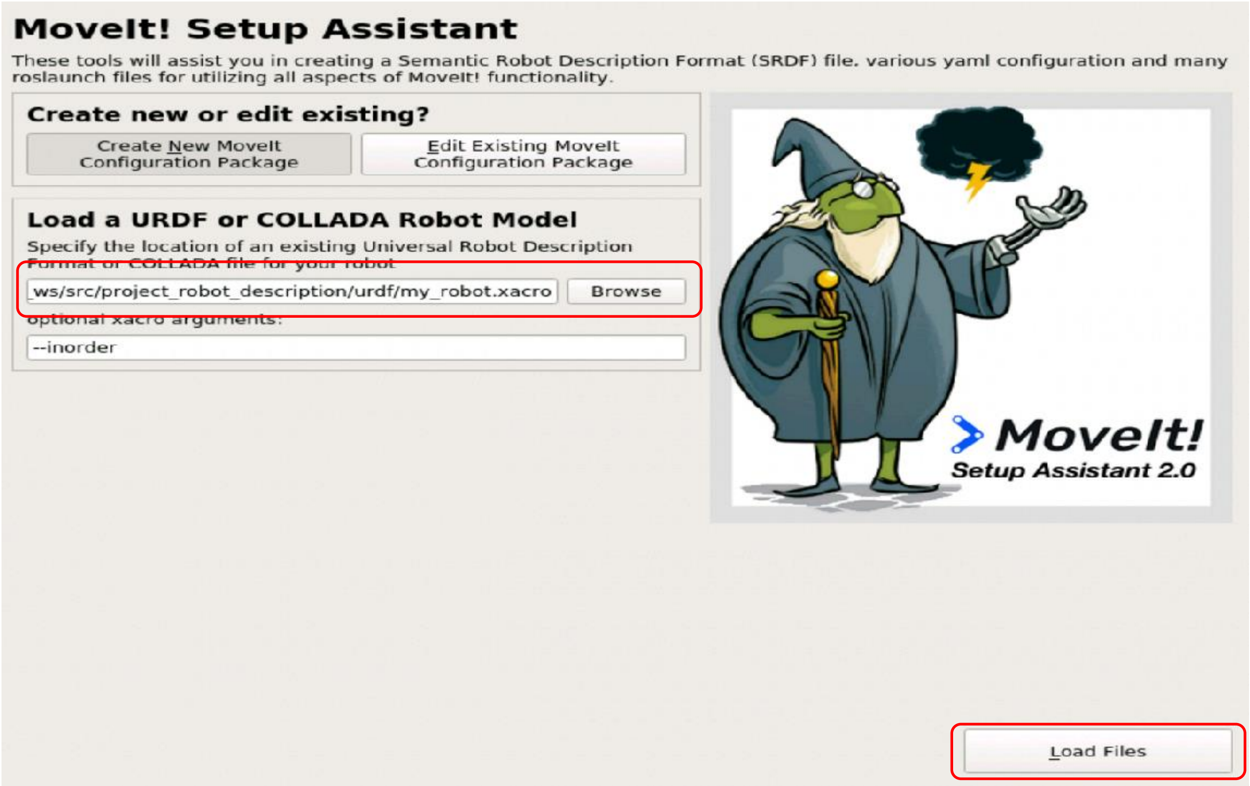
```
user:~$ roslaunch moveit_setup_assistant setup_assistant.launch
```

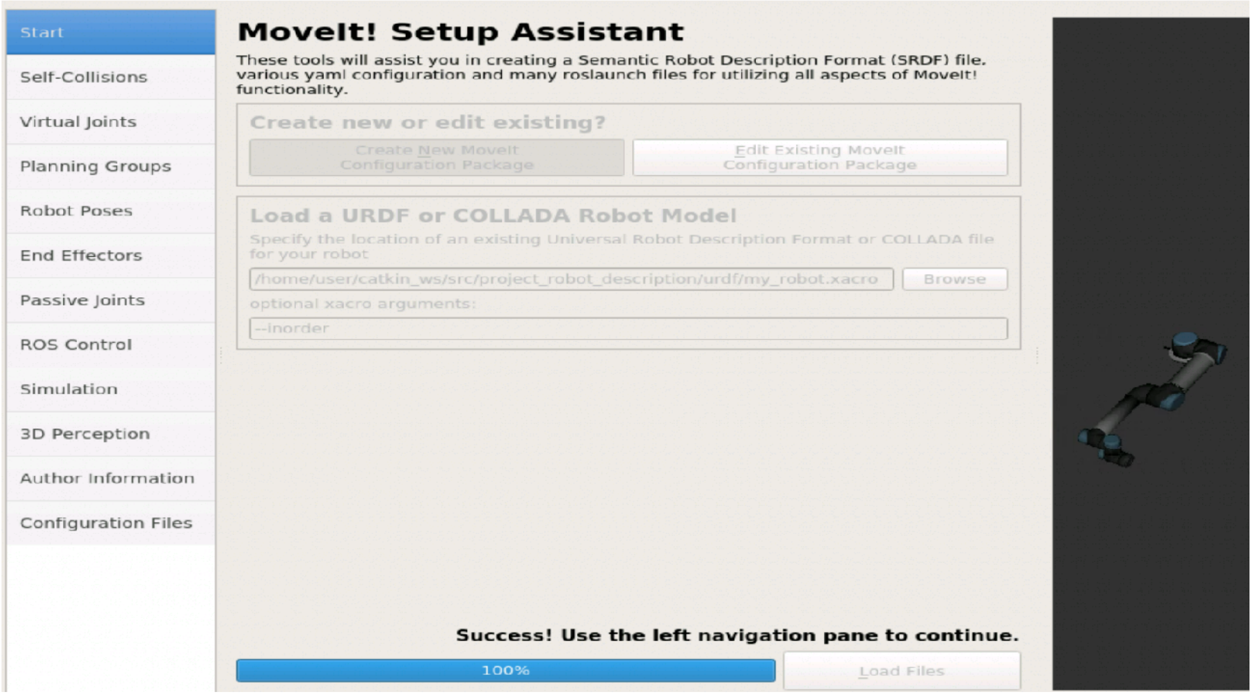
(2) Open “graphical interface”

(3) Click “Create New MoveIt Configuration Package”



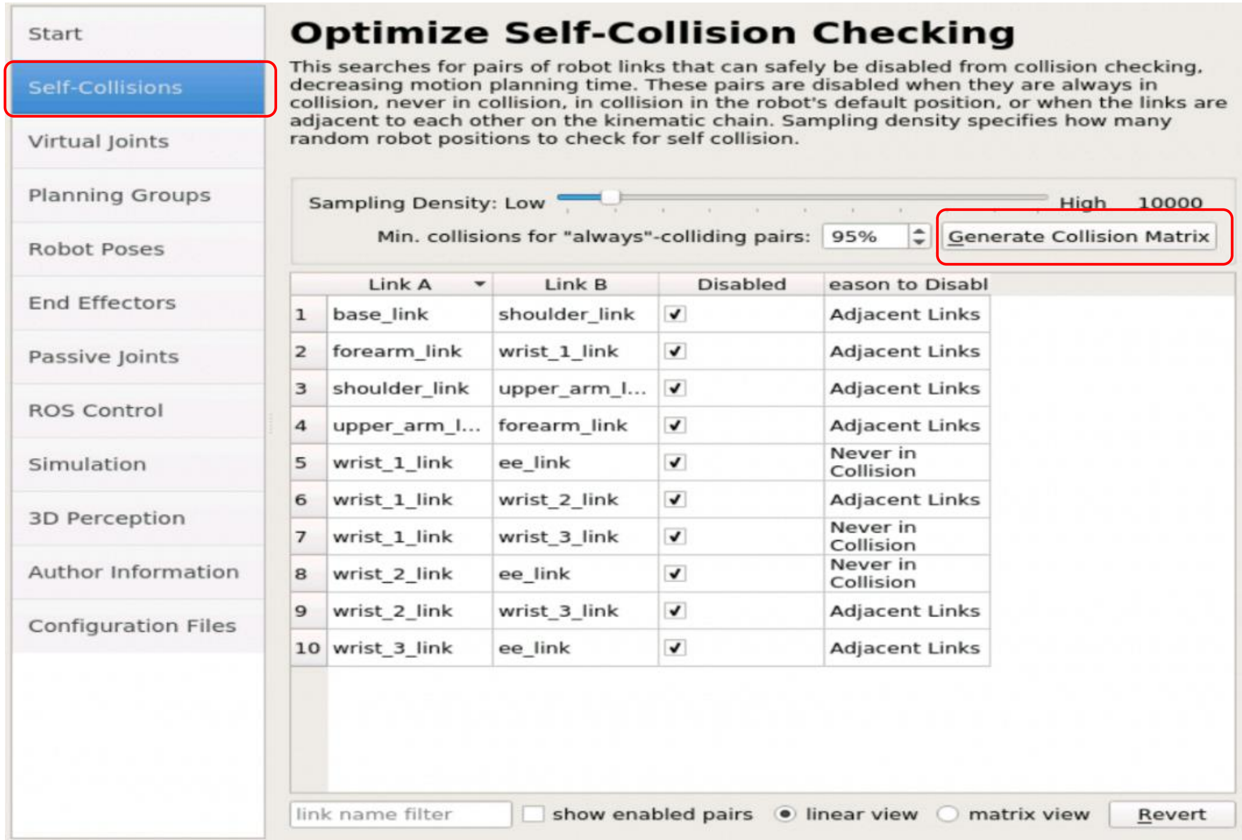

(4) Browse the XACRO File (my_robot.xacro) we create at the previous part and load the file.



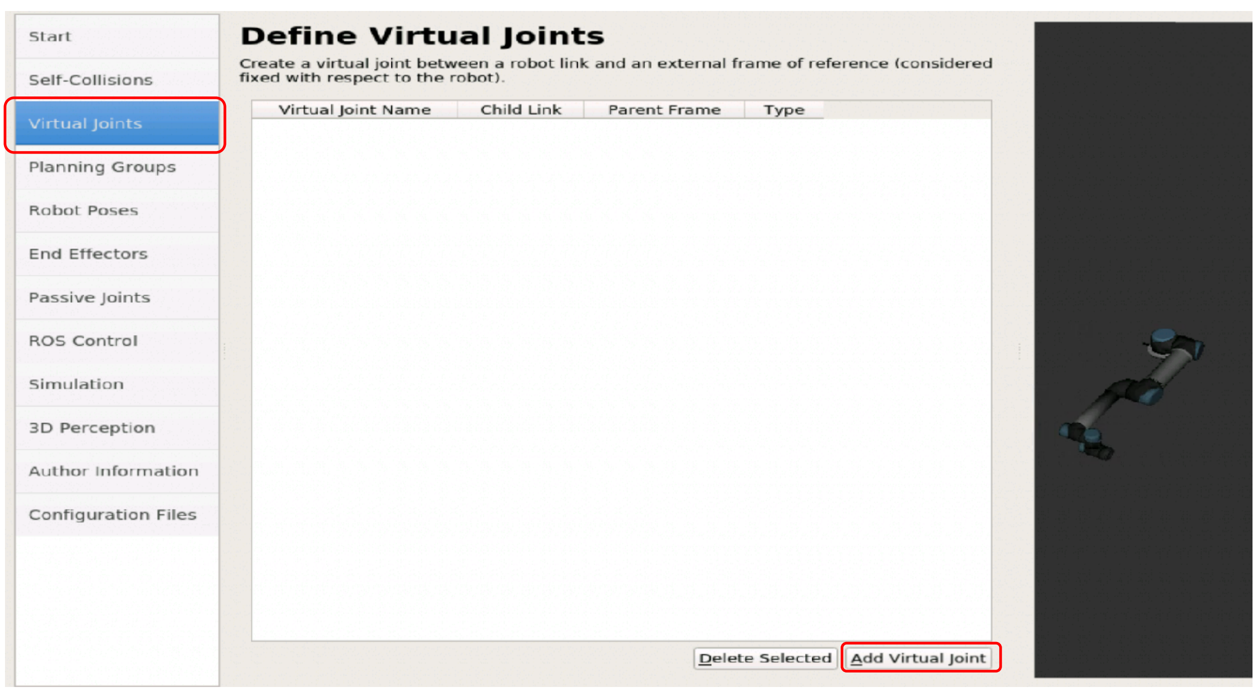



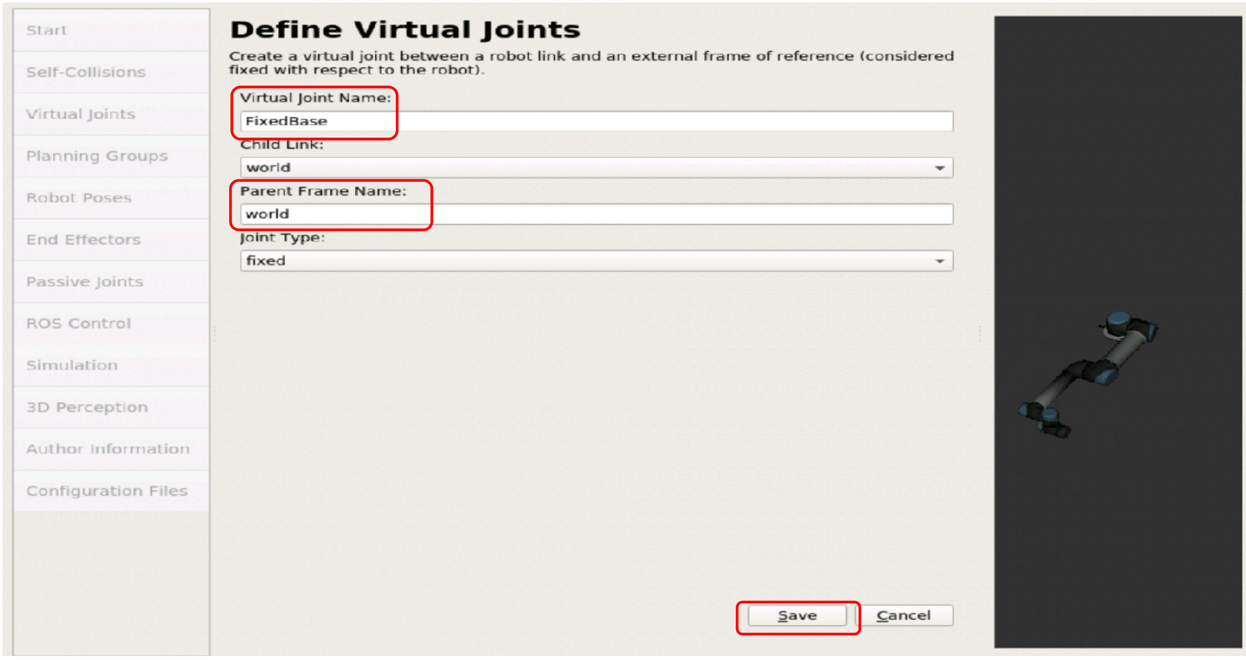

(5) Generate Collision Matrix in Self-Collisions option

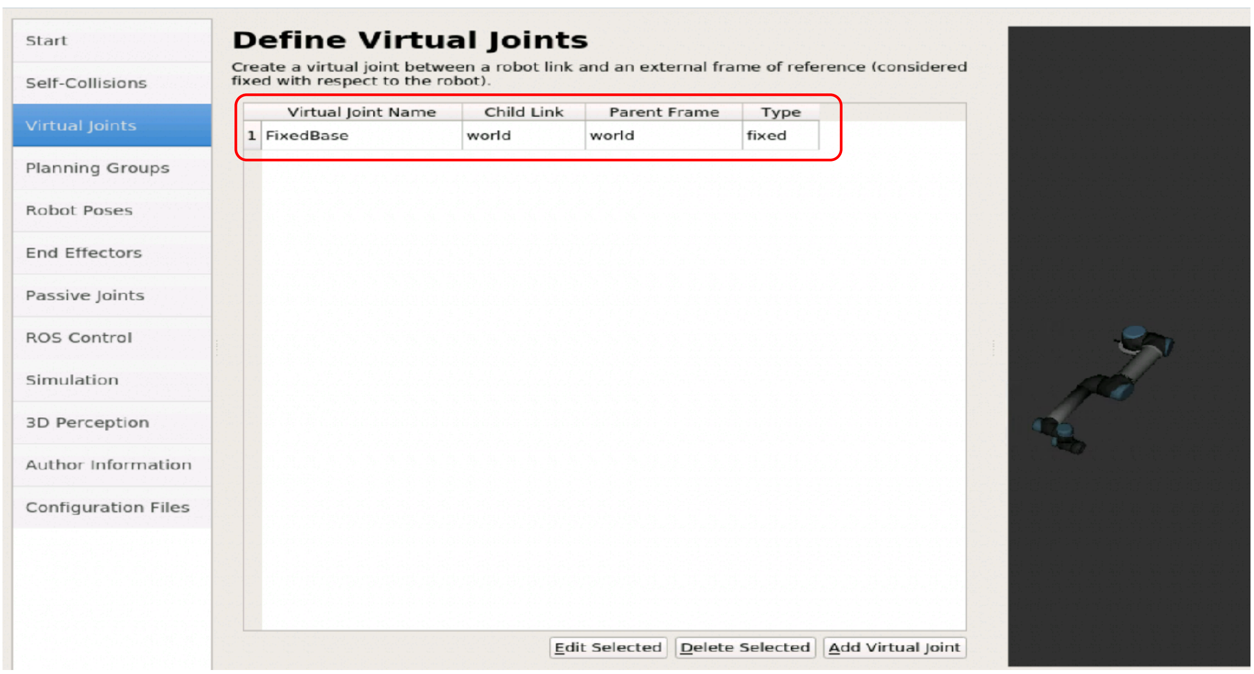
The setup assistant will automatically check self-collisions of all links and joints based on the XACRO file we loaded.




(6) Connect robot’s base with world by creating a virtual joint.

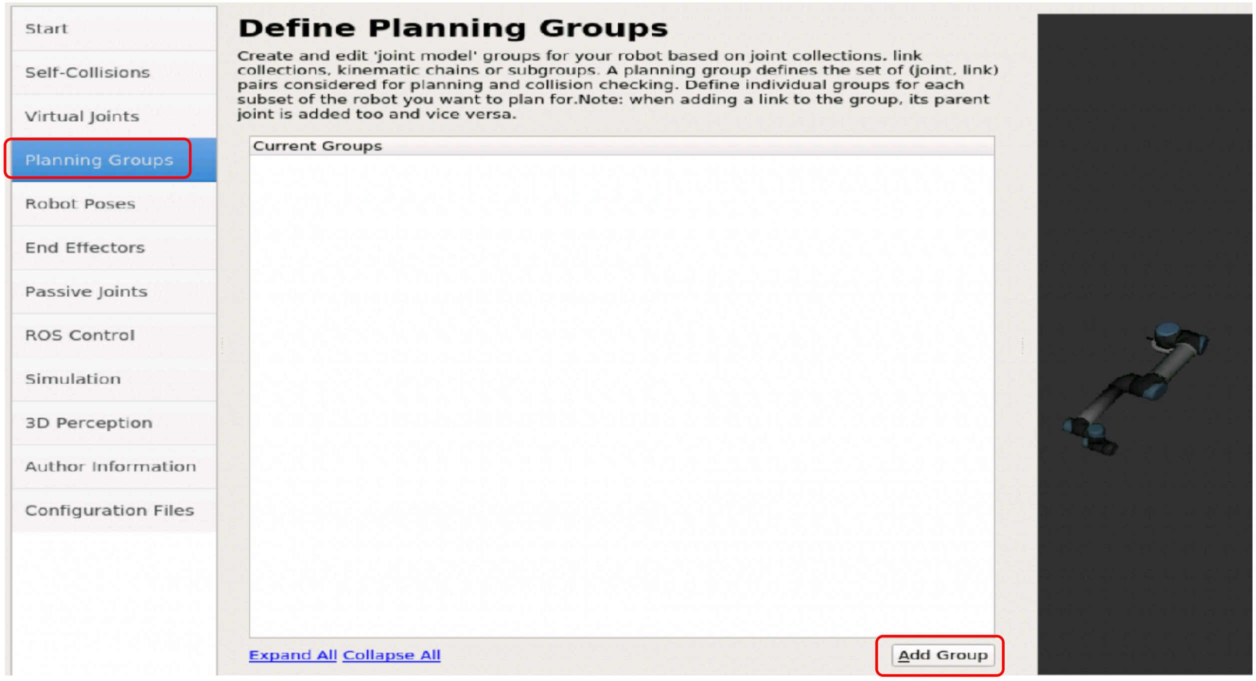




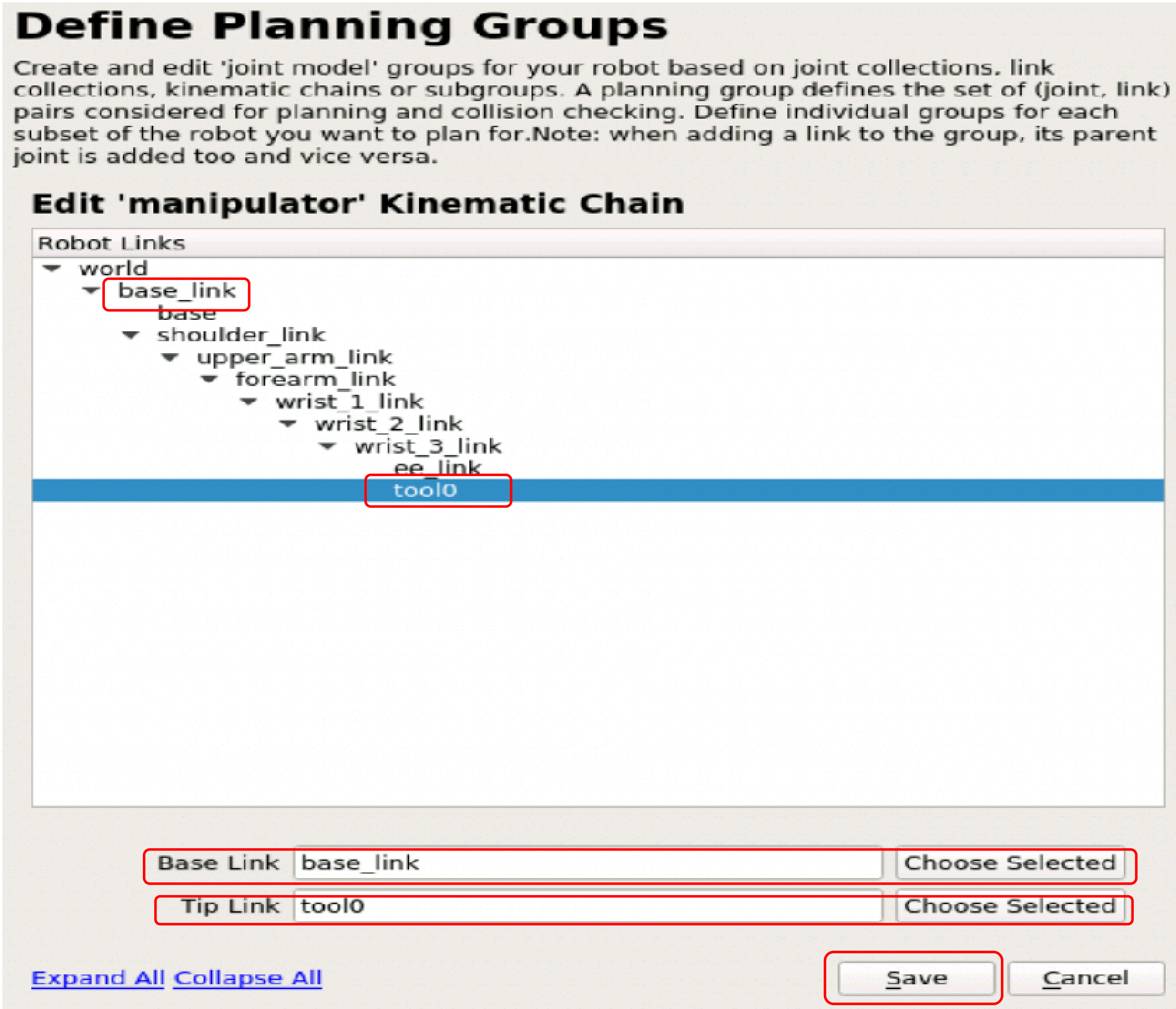



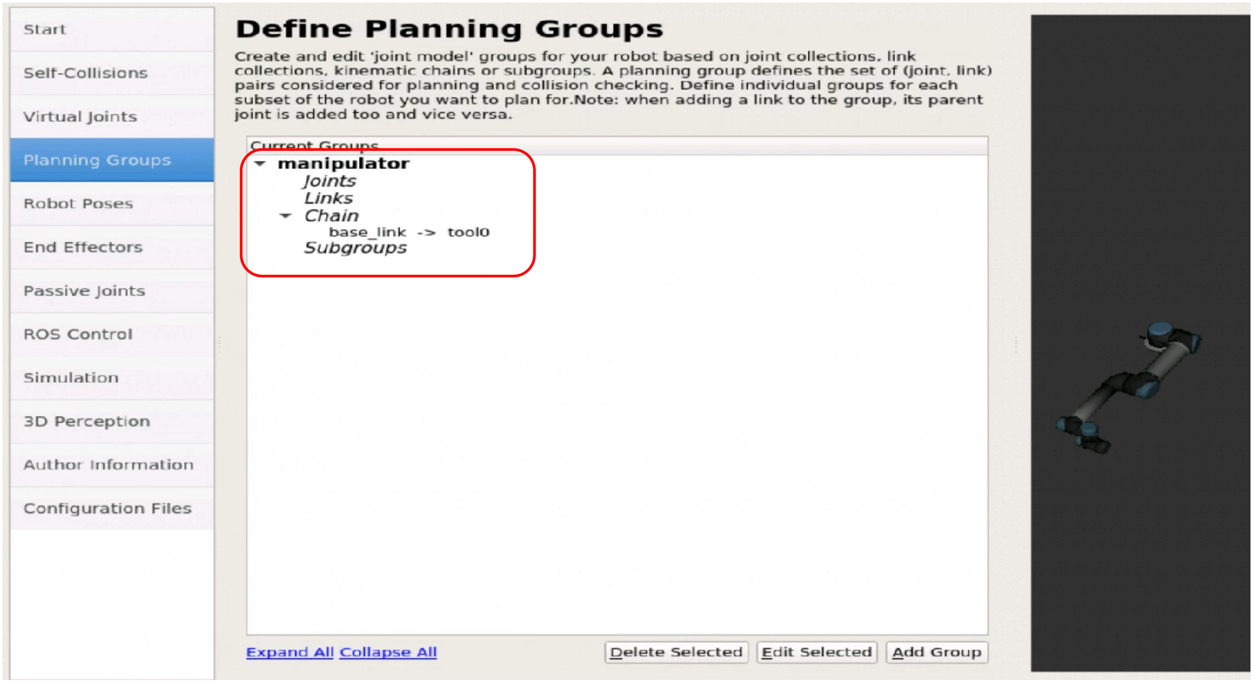

(7) Define Planning Groups.

We are defining a set (collection) of joints that will be used for motion planning. In our case, it will be all joints in UR5.


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.
Current Groups
Expand All Collapse All
Add Group

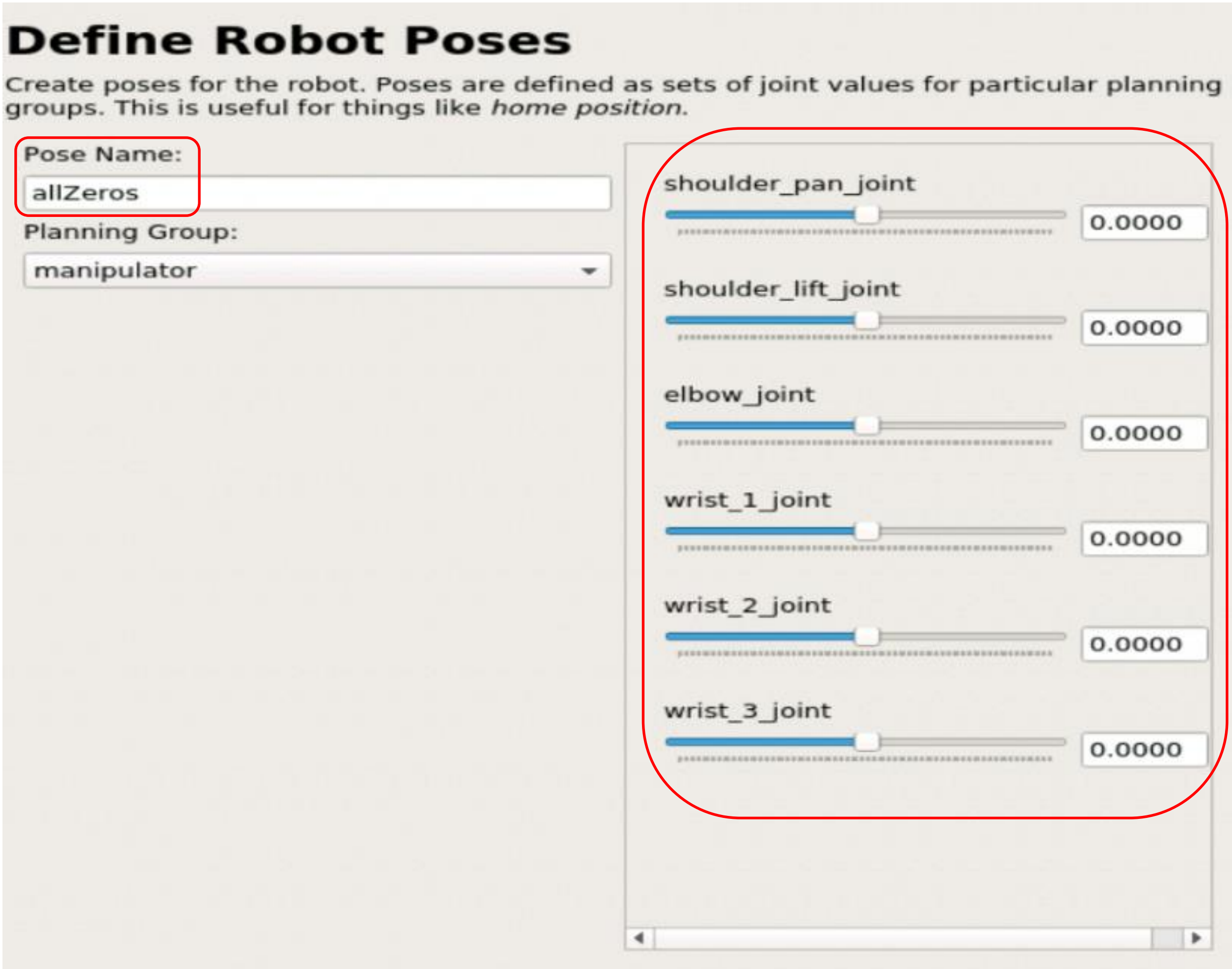

Define Planning Groups
Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the 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: manipulator
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

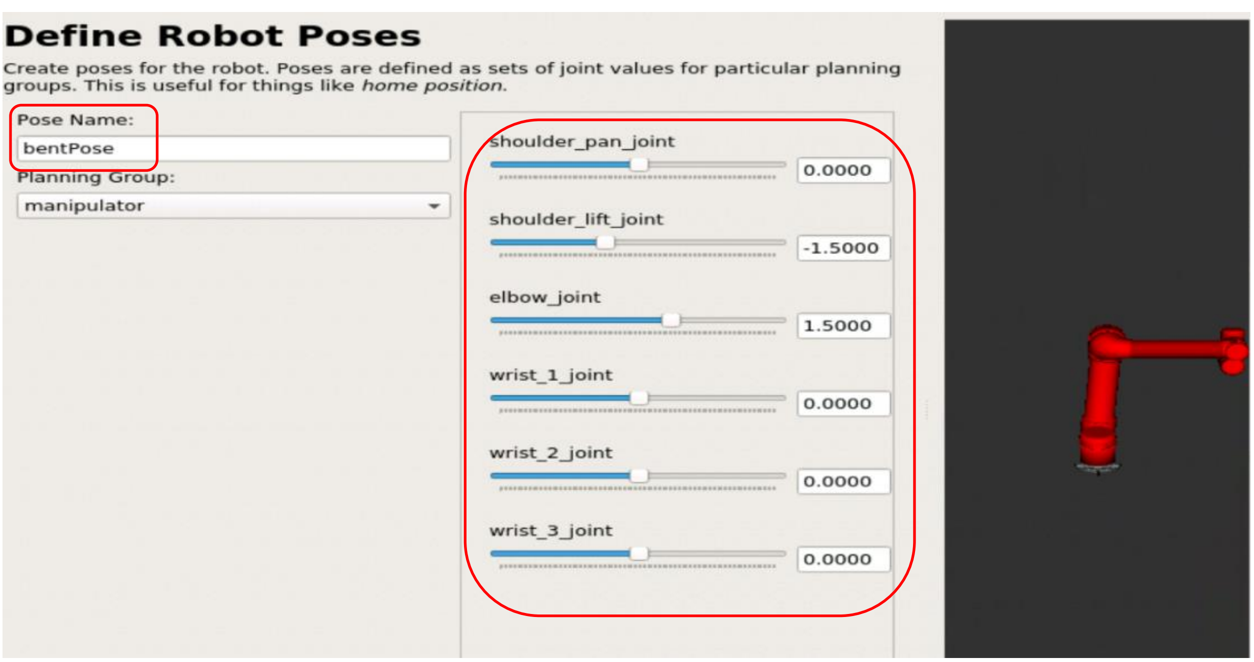

Define Planning Groups
Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the robot you want to plan for.Note: when adding a link to the group, its parent joint is added too and vice versa.
Edit 'manipulator' Kinematic Chain
Robot Links
world
base_link
base
shoulder_link
upper_arm_link
forearm_link
wrist_1_link
wrist_2_link
wrist_3_link
ee_link
tool0
Base Link
base_link
Choose Selected
Tip Link
tool0
Choose Selected
Expand All
Collapse All
Save
Cancel

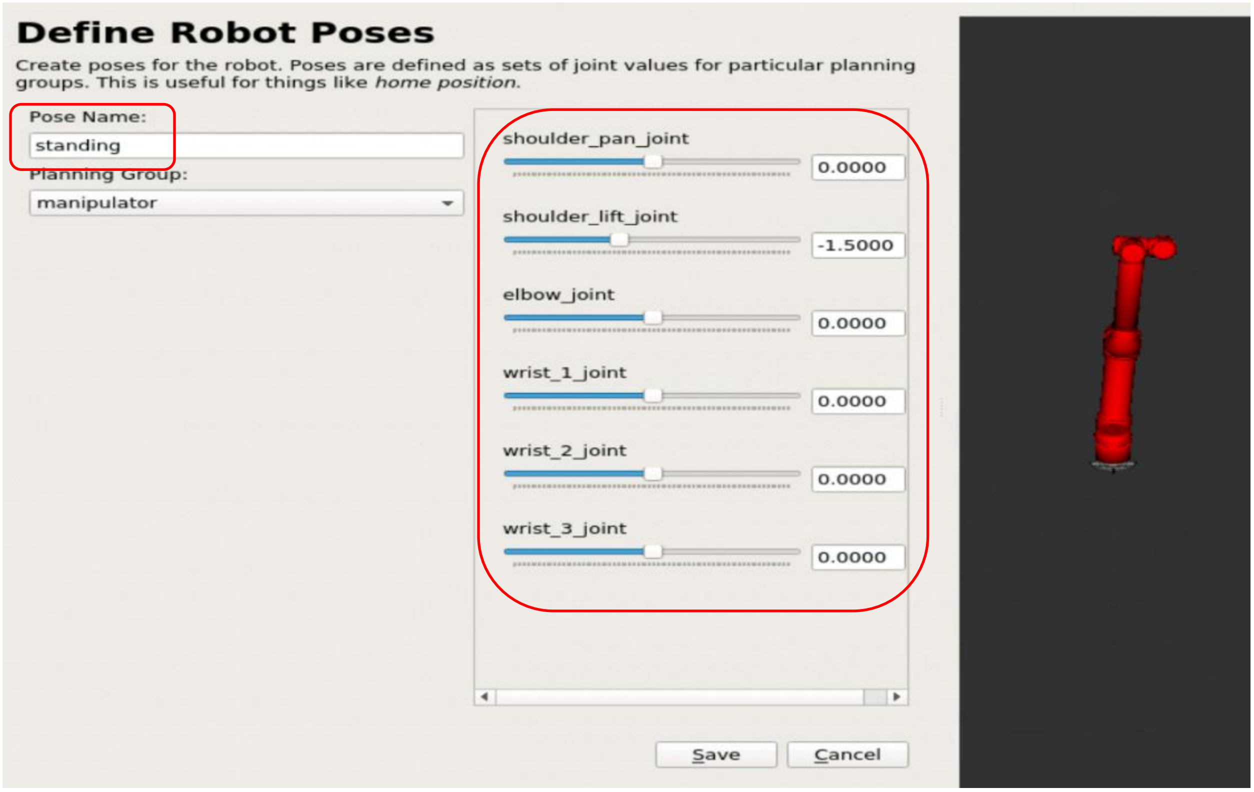



(8) Pre-define robot poses

MoveIt setup assistant tool provides a way we can define robot poses ahead by name. We will define 3 poses such as allZeros, bentPose, and standing

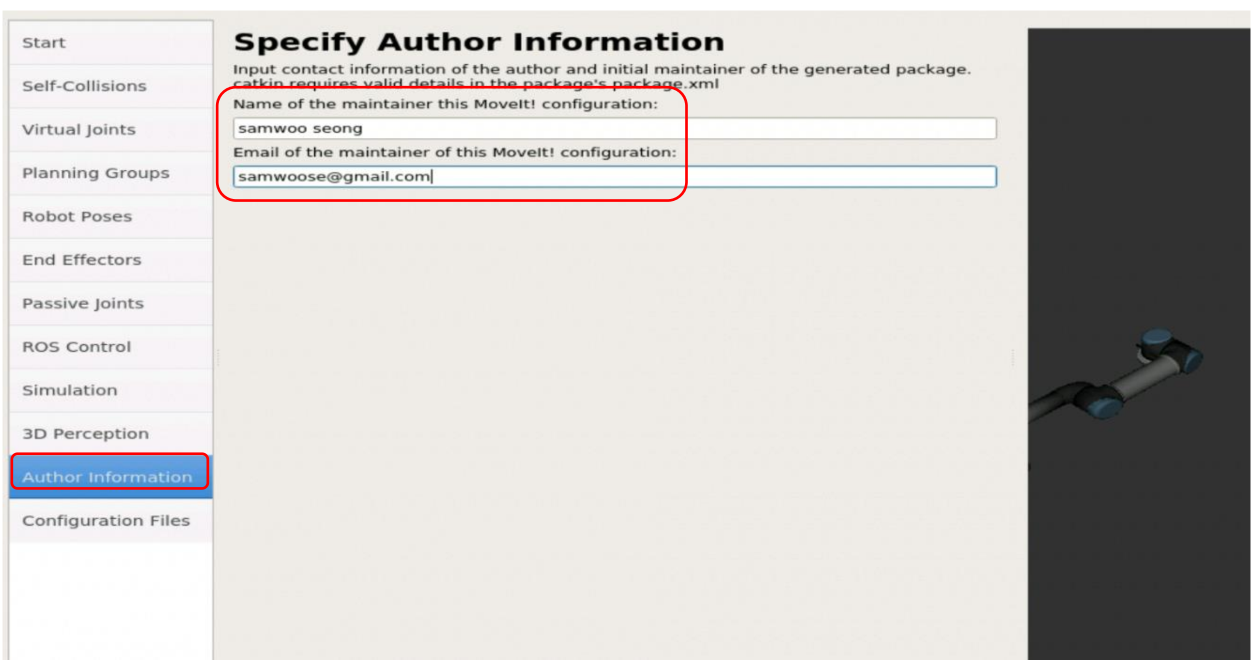

Define Robot Poses
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like home position.
Pose Name:
allZeros
Planning Group:
manipulator
shoulder_pan_joint
0.0000
shoulder_lift_joint
0.0000
elbow_joint
0.0000
wrist_1_joint
0.0000
wrist_2_joint
0.0000
wrist_3_joint
0.0000


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:
bentPose
Planning Group:
manipulator
shoulder_pan_joint
0.0000
shoulder_lift_joint
-1.5000
elbow_joint
1.5000
wrist_1_joint
0.0000
wrist_2_joint
0.0000
wrist_3_joint
0.0000

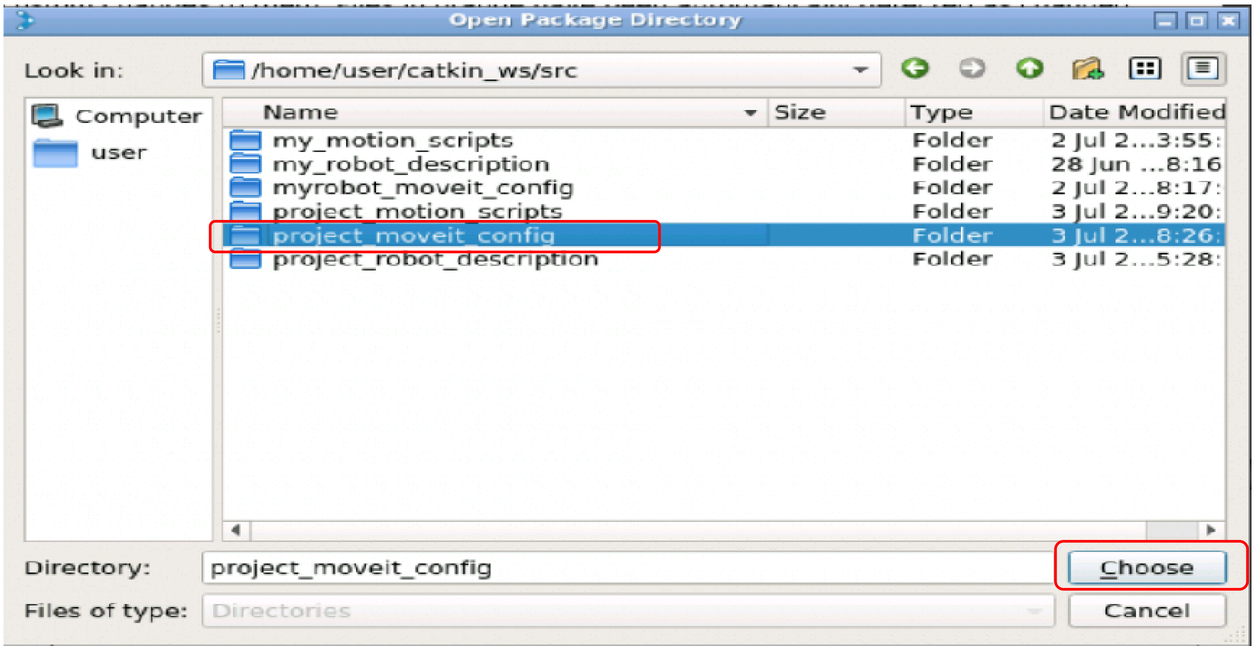


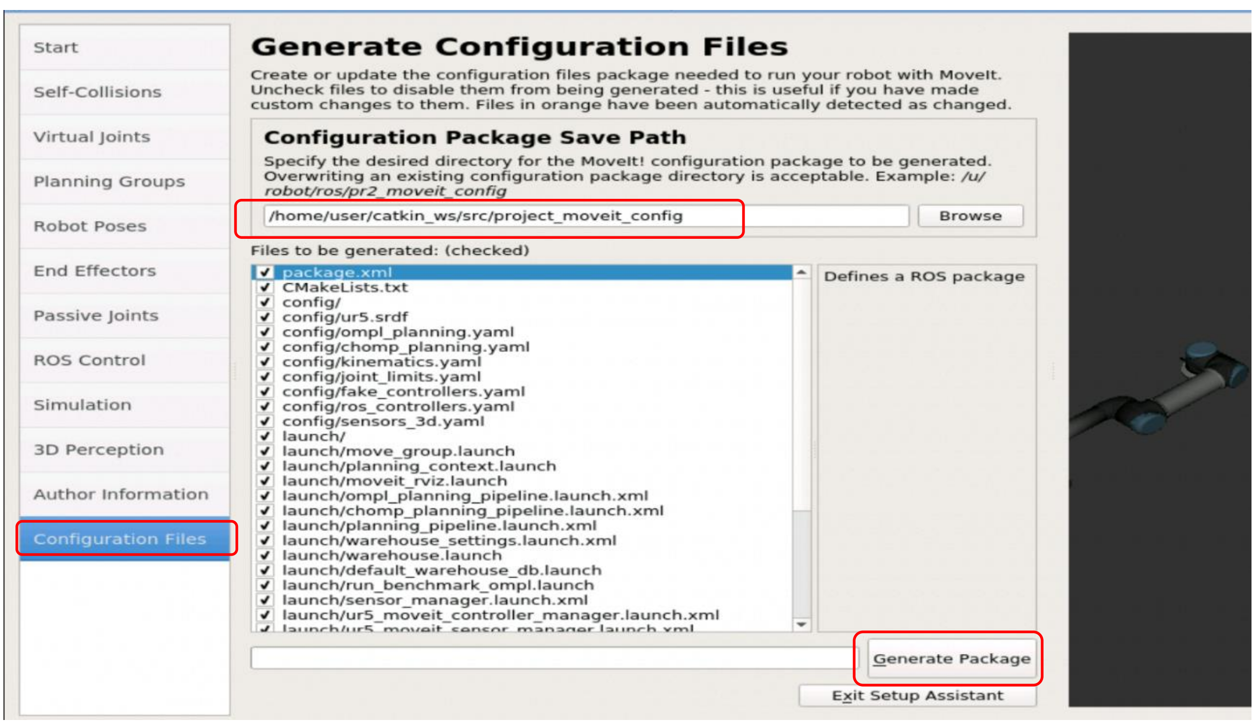

(9) Specify Author Information

e.g.



(10) Generate Configuration Files



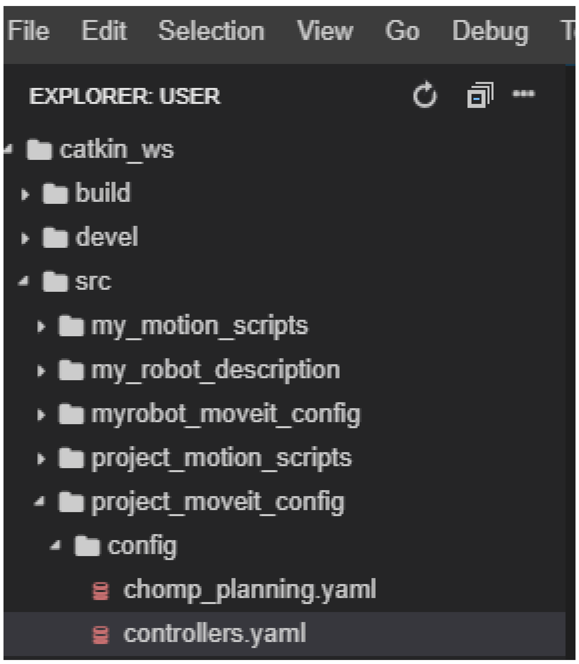
<3. Make connection between the MoveIt package and UR5 in Gazebo Simulation >

Now, it is time to connect the MoveIt package, “project_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.

(1) create “controllers.yaml” in “config” directory of “project_moveit_config”.

This contains information about joints of UR5 and controllers to be controllered.

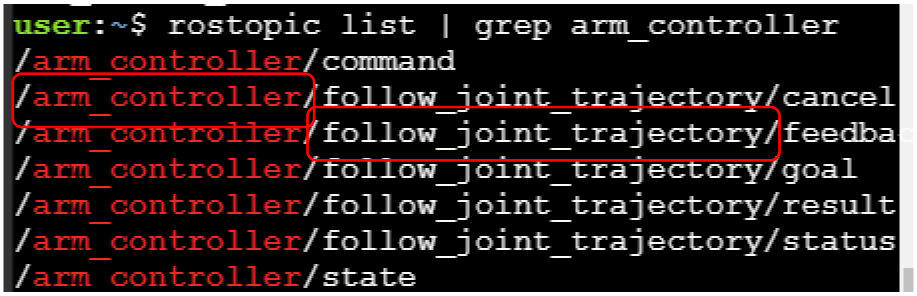
e.g. File directory



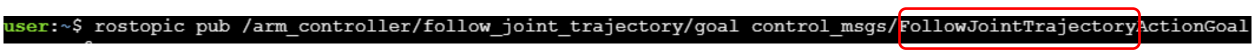

e.g. “controllers.yaml”

```
controller_list:
  - name: arm_controller
    action_ns: "follow_joint_trajectory"
    type: FollowJointTrajectory
    joints: [shoulder_pan_joint, shoulder_lift_joint, elbow_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint]
```

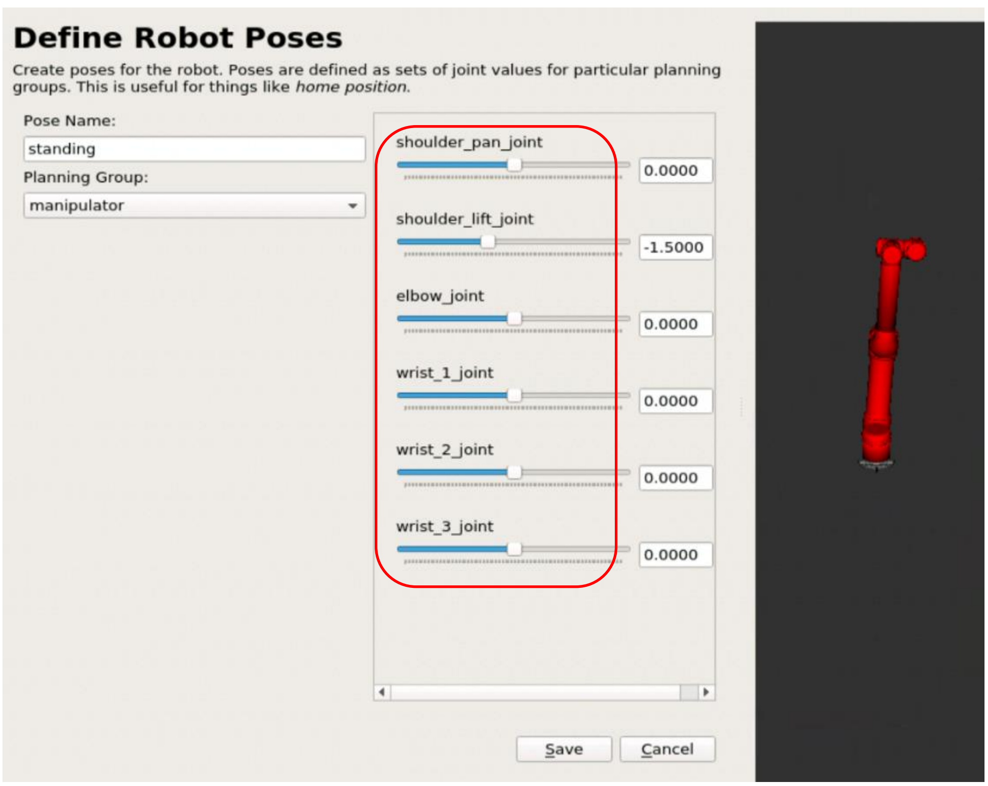
As we can see, name of controller is “arm_controller”, its Action Server is “follow_joint_trajectory”, its message type is “FollowJointTrajectory, and all names of joints are listed. This information can be found be as following e.g. controller and name of Action Server




e.g. message type that Action uses

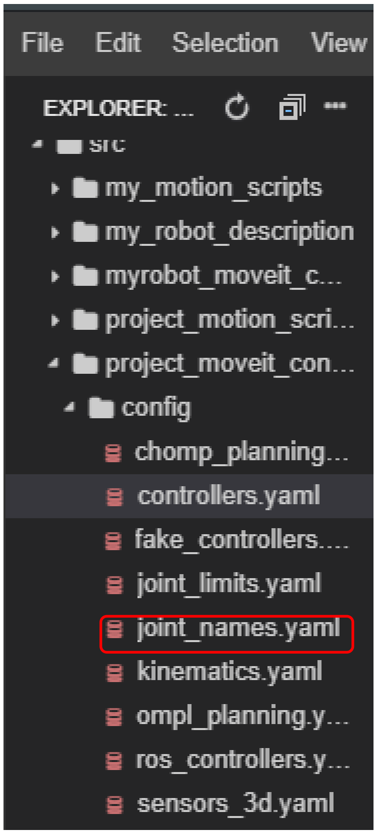



e.g. names of joints



(2) Next thing we need to do is to create file containing all the name of the joints of UR5.

e.g. directory of “joint_names.yaml”




e.g. “joint_names.yaml”

joint_names.yaml

```yaml
controller_joint_names: [shoulder_pan_joint, shoulder_lift_joint,
elbow_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint]
```

(3) Then, open the “ur5_moveit_controoler_manager.launch.xml” in “launch” directory and fill it like this.

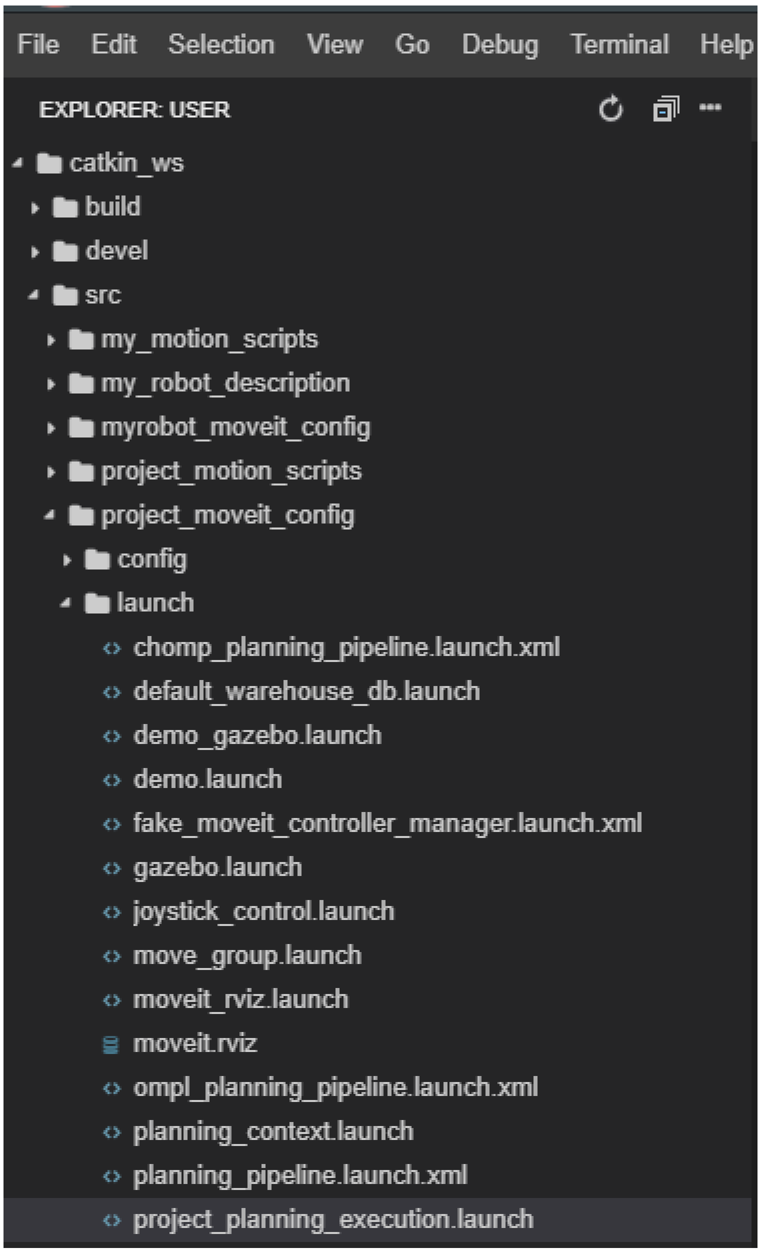
nes.yaml ur5_moveit_controller_manager.launch.xml

```xml
<launch>
  <rosparam file="$(find project_moveit_config)/config/controllers.yaml"/>
  <param name="use_controller_manager" value="false"/>
  <param name="trajectory_execution/execution_duration_monitoring" value="false"/>
  <param name="moveit_controller_manager"
  value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
</launch>
```

In this file, “controllers.yaml” and “MoveItSimpleControllerManager” plug are loaded. It will let us to send motion plan calculated by kinematic solver to UR5 in simulation.

(4) Create “project_planning_execution.launch” file in “launch” in “project_moveit_config” directory.

e.g. directory



e.g. “project_planning_execution.launch”

```xml
<launch>

  <rosparam command="load" file="$(find project_moveit_config)/config/joint_names.yaml"/>

  <include file="$(find project_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">[/joint_states]</rosparam>
  </node>

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

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

</launch>
```

Here, we are simply loading and launching files. The crucial part make MoveIt able to know where UR5 is at each moment is this part.

```xml
<rosparam param="/source_list">[/joint_states]</rosparam>
```

“/joint_states” can be found by “rostopic list” command

e.g.

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

<4. Write up a Python script that perform motion planning and execution with UR5>

Finally, it is time for programming. Thanks to MoveIt package, this can be done by creating a few objects of certain classes provided by MoveIt. The easiest way to explain is going through the Python script. The whole script, “planning_script.py” can be found in “catkin_ws” directory on GitHub.

-Part 1.

```python
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.

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

We are initializing moveit_commander module.

-Part 3.

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

We are initializing "move_group_python_interface_tutorial" node.

-Part 4.

```python
robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
group = moveit_commander.MoveGroupCommander("manipulator")
```

Here, three objects are created. "robot" object is for communicating with robot. "scene" object has something to do with the world around UR5. Then, most importantly, "group" object is in charge of setting up target pose, motion planning, and executing.

-Part 5.

```python
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 doing the job.

```
25  #Pose 2
26  group.set_named_target("bentPose")
27
28  plan1 = group.plan()
29  group.go(wait=True)
30
31  #Pose 3
32  group.set_named_target("standing")
33
34  plan2 = group.plan()
35  group.go(wait=True)
36
37  #Pose 2
38  group.set_named_target("bentPose")
39
40  plan3 = group.plan()
41  group.go(wait=True)
42
43  #Pose 1
44  group.set_named_target("allZeros")
45
46  plan2 = group.plan()
47  group.go(wait=True)
```

Since most lines of code work in a similar way except for the target pose, only from line 26 to line 29, we need to dig in. We can make use of one of pre-defined poses, “bentPose” as a parameter of the method, “set_named_target” with “group” object. Basically, it set a Pose that we want UR5 to get after execution. Then, with method “plan()”, the program will try to find a possible way to get the pose we are targeting. If there is a possible motion plan, we will execute that plan with “go(wait=True)” method. Finally, we repeat similar process to meet the sequence of motions given in the project description. Assuming UR5 starts with “allZeros” pose, it will have the following sequence of motions.

“allZeros” ➔ “bentPose” ➔ “standing”➔ “bentPose”➔ “allZeros”

-Part 7.

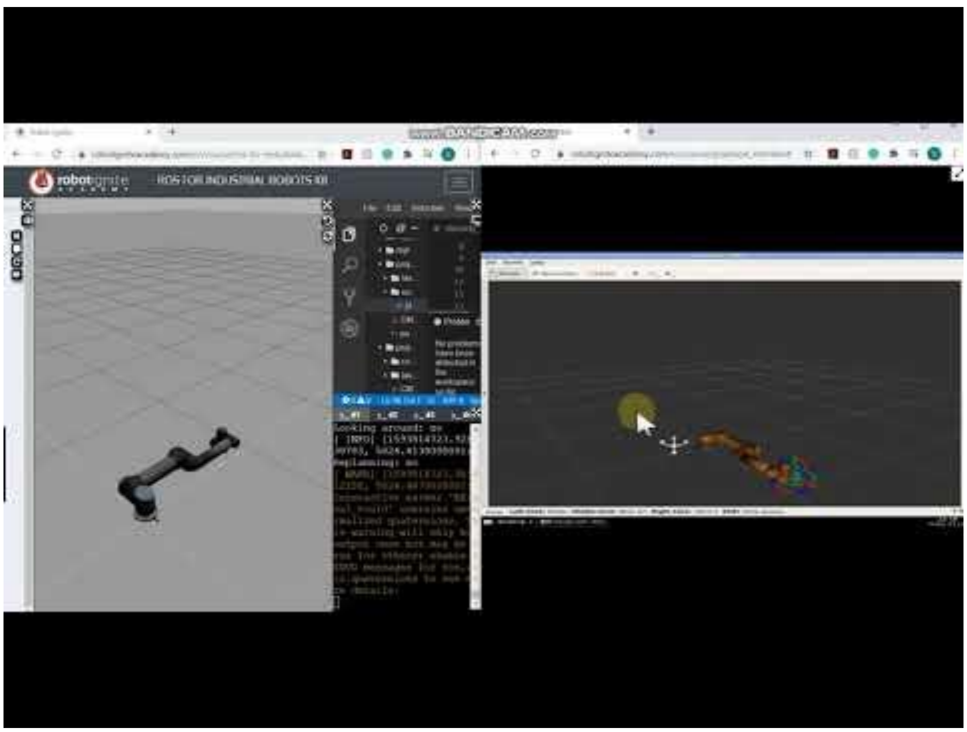
```
rospy.sleep(5)

moveit_commander.roscpp_shutdown()
```

Then, we call "sleep()" function (whenever you publish or subscribe something you need to call "sleep()" function) and shutdown the "moveit_commander" module.

Throughout the all 4 major steps, we are ready to perform the task we are aiming for. Please, take a look at ReadMe file to run this program.

## 1.3 Experimental Results



Video 1. Motion Planning and Executing

https://youtu.be/kb4uPHaihkc

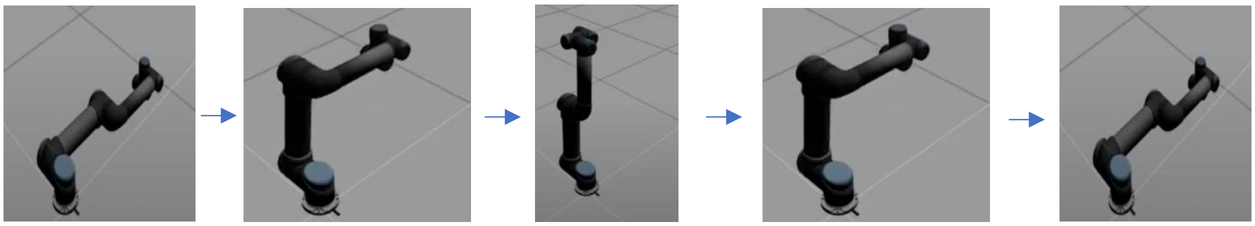


Figure 1. sequence of motions

1.4 Discussion

<1.4.1 Kinodynamic Motion Planning by Interior-Exterior Cell Exploration>

It is worth taking a look at more details about how motion planning works behind the scene. In our setting, we didn't particularly choose a motion planner. However, MoveIt! Package is able to select a proper planner for our program amongst multiple sample-based motion planners in OMPL (Open Motion Planning Library) by default. In our discussion, we are going to talk about one of the most famous planners, KPIECE (Kinodynamic Motion Planning by Interior-Exterior Cell Exploration) planner.

A goal of motion planning is apparently to find motion plans of something (in our case UR5) so that the something can reach to the targeting goal (in our case, a particular pose of UR5) with one of found motions. A motion, $\mu$, can be described mathematically.[3]

$$\mu = (s, u, t) \; where \; a \; state \; s \in Q, a \; control \; u \; \in U, and \; a \; duration \; t \; \in \; \mathbb{R}^{\geq 0}$$

$$(i.e., the \; motion \; \mu \; is \; produced \; by \; u \; being \; applied \; for \; u \; duration \; t \; from \; state \; s)$$

Therefore, basically we are trying to find a set(s) of $\mu$'s (it can be seen as a motion plan(s)) with a specific algorithm (i.e., a specific planner in our case KPIECE) here.

Many other popular planners are usually sampling single motion $\mu$ or single state $s$, iterating the process and ending up with possible sets of motions. Then, they check if the motion plans are valid. This process can be computationally expensive. On the other hand, KPIECE introduces concept of discretization which can be formed with multiple levels of grid and each grid consists of cells as below in Fig. 1. [3]

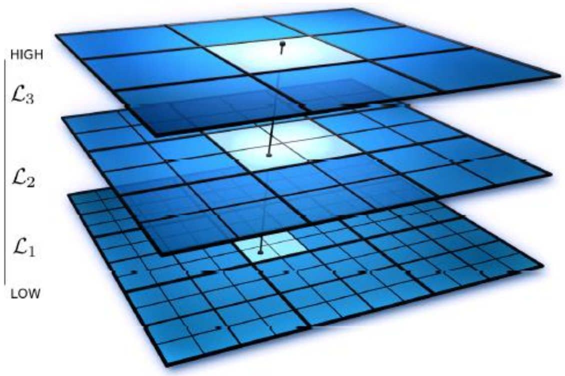



Fig. 1. An example discretization with three levels. The line intersecting the three levels defines a cell chain. [3]

Then, KPIECE samples cell chain instead of motion or state. From there, it selects motion $\mu$, state $s$, control $u$, and duration $t$ as the algorithm proceeds. There are two main algorithms in this procedure as follws.

**Algorithm 1** KPIECE($q_{start}$, $N_{iterations}$)

```
1: Let μ0 be the motion of duration 0 containing solely q_start
2: Create an empty Grid data-structure G
3: G.AddMotion(μ0)
4: for i ← 1...N_iterations do
5:    Select a cell chain c from G, with a bias on exterior cells (70% - 80%)
6:    Select μ from c according to a half normal distribution
7:    Select s along μ
8:    Sample random control u ∈ U and simulation time t ∈ ℝ+
9:    Check if any motion (s, u, t∘), t∘ ∈ (0, t] is valid (forward propagation)
10:   if a motion is found then
11:      Construct the valid motion μ∘ = (s, u, t∘) with t∘ maximal
12:      If μ∘ reaches the goal region, return path to μ∘
13:      G.AddMotion(μ∘)
14:   end if
15:   for every level L_j do
16:      P_j = α + β · (ratio of increase in coverage of L_j to simulated time)
17:      Multiply the score of cell p_j in c by P_j if and only if P_j < 1
18:   end for
19: end for
```

Fig. 2. Main algorithm in KPIECE [3].

First of all, we initialize motion $\mu_0$ with duration 0 (line1), create an empty Grid data structure (line2), G, and add the motion $\mu_0$ to the G (line3). Then, while the number of iterations, $N_{iterations}$ repeats sampling a cell chain, motion, state, control, and duration according to the previous samples (i.e., sampling motion is affected by a cell chain selected in the previous line) and certain randomness (e.g., half normal distribution) (line 5 ~ 8). Next, we check if the motion is valid. If so (i.e., motion is found), construct this motion with certain duration if this motion reaches the goal region and add this found motion to our data structure G. [3]

While adding the calculated motion, KPIECE also uses another algorithm. It is summarized below. For more details on Algorithm 2, please refer to the original paper

**Algorithm 2** ADDMOTION$(s, u, t)$

Split $(s, u, t)$ into motions $\mu_1, \ldots, \mu_k$ such that $\mu_i, i \in \{1, \ldots, k\}$ does not cross the boundary of any cell at the lowest level of discretization
**for** $\mu_\circ \in \{\mu_1, \ldots, \mu_k\}$ **do**
    Find the cell chain corresponding to $\mu_\circ$
    Instantiate cells in the chain, if needed
    Add $\mu_\circ$ to the cell at the lowest level in the chain
    Update coverage measures and lists of interior and exterior cells, if needed
**end for**

Fig. 3. Algorithm of ADDMOTION [3]

The advantage of sampling cell chain approaches is that it provides significant computational improvement because it doesn't sample single state or motion meaning it work faster than other previous approaches. For instance, it is much faster than other algorithms such as RRT, EST, and PDST in terms of runtime. Furthermore, it doesn't require distance metrics that are often used in other planners which sometimes it hard to define the distance metrics. However, it needs a projection of the state space and the specification of a discretization that forms the grid layers.

<1.4.2 MoveIt System Architecture>

MoveIt! Official website provides an intuitive and comprehensive image of its system architecture. [4]

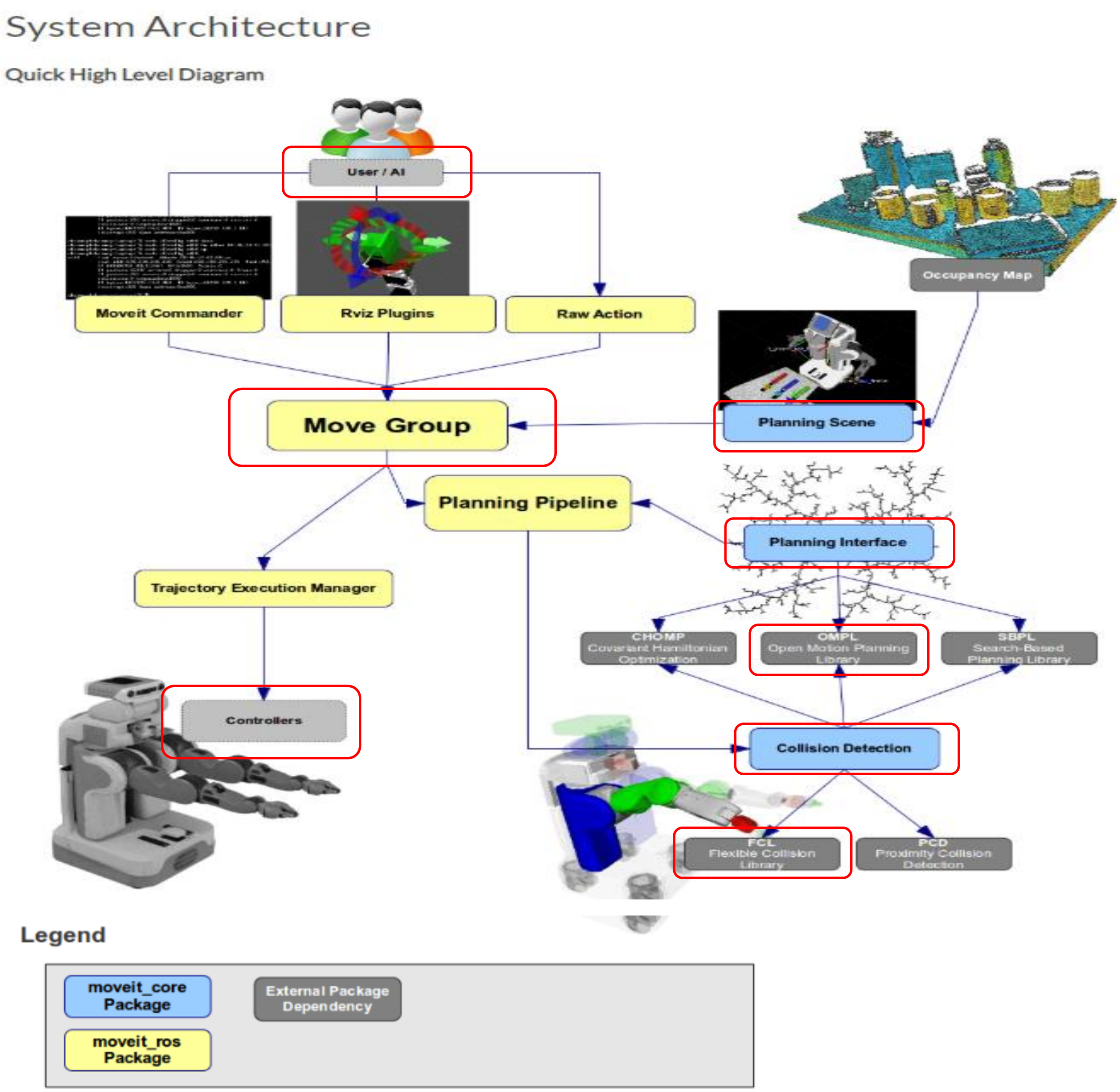



Fig. 4. MoveIt System Structure [4]

Through this system structure, we can get a general idea of how UR5 obtains a motion plan and execute the motion place. Basically, “Move Group” combines all important information about robot and its surrounding from user and Planning Scene. Then, it goes to Planning Pipeline which is responsible for motion planning and collision detection. For each task, they are communicating using different libraries. For instance, Our UR5 is using OMPL for planner and using FCL for collision detector. The robot calculates plans with one of methods in OMPL and check if this plan has any collision in it. If a possible motion plan has found, “Move Group” will let “trajectory Execution Manager” know so that “Controller” can properly control joints of the robot as we want.

For the future work, we are going to apply this motion planning technique into grasping task. Furthermore, we will try to investigate field of perception if time allows.

# References

[1] Project Contents Credit: Robot Ignite Academy

https://www.robotigniteacademy.com/en/course/ROS-for-Industrial-Robots-101/details/

[2] Report on URDF for robot modeling, Available Online:

https://github.com/Samwoose/URDF_RobotDesignWithROS/blob/master/Report_QUIZ_URDF%20FOR%20ROBOT%20MODELING.pdf

[3] Şucan, I. A., & Kavraki, L. E. (2009). Kinodynamic Motion Planning by Interior-Exterior Cell Exploration. *Springer Tracts in Advanced Robotics Algorithmic Foundation of Robotics VIII,* 449-464. doi:10.1007/978-3-642-00312-7_28, Available Online:

http://kavrakilab.org/publications/sucan-kavraki2009kinodynamic-motion-planning.pdf

[4] Concepts of MoveIt!, Available Online:

https://moveit.ros.org/documentation/concepts/