Lecture 5 (28/09) - Mini-Project 1: Motion Planning

Starts 28 September, 2020 1:00 PM

Mini-Project 1: Motion Planning

We meet on Zoom at 13:00. Link: https://dtudk.zoom.us/j/63569471741?pwd=emNGMEQxNitXdGFhc3hSZXhqdVFNQT09

Reading Material

1. Read the MoveIt Documentation (Specifically the link "Move Group Python Interface")
link: Link

Time Plan

Day 1: 28/09

- 13:00 Demo, Local workspace and Mini-Project 1 Description
- 14:00 Let the work begin

Day 2: 05/10

- 13:00 More work and we're here to help

Description

Day 1: 28/09

Today we will give you the setup you'll be working on during the mini project 1.
We will demo the main functionality.
Then we will describe the project.
Finally, you'll start working on it. We will be here to help (help not provide the solution). Remember! This is group work (Do you have a plan? Who's doing what? How do you track progress?, etc...)

Setup your local workspace

First we'll need to setup your workspace. Download the following zip file and extract it to the src folder in your catkin workspace (/home/student/catkin_ws/src).

/content/enforced/27051-OFFERING-602991/lecture_5_new.zip

Next we need to make sure all the necessities have been installed. Run the following command in a terminal. Notice it's all just one long command.

```
sudo apt-get install ros-melodic-gazebo-ros ros-melodic-eigen-conversions r
ros-melodic-roslint libgazebo9* ros-melodic-moveit-core ros-melodic-moveit
ompl ros-melodic-moveit-fake-controller-manager ros-melodic-moveit-simpl
gazebo-ros-control
```

Now we'll do a clean build of the catkin workspace by running the following three commands:

```
cd /home/student/catkin_ws
catkin clean
catkin build
```

Let's test our new robot arm.

In three different terminals (or terminal tabs) run the following commands:

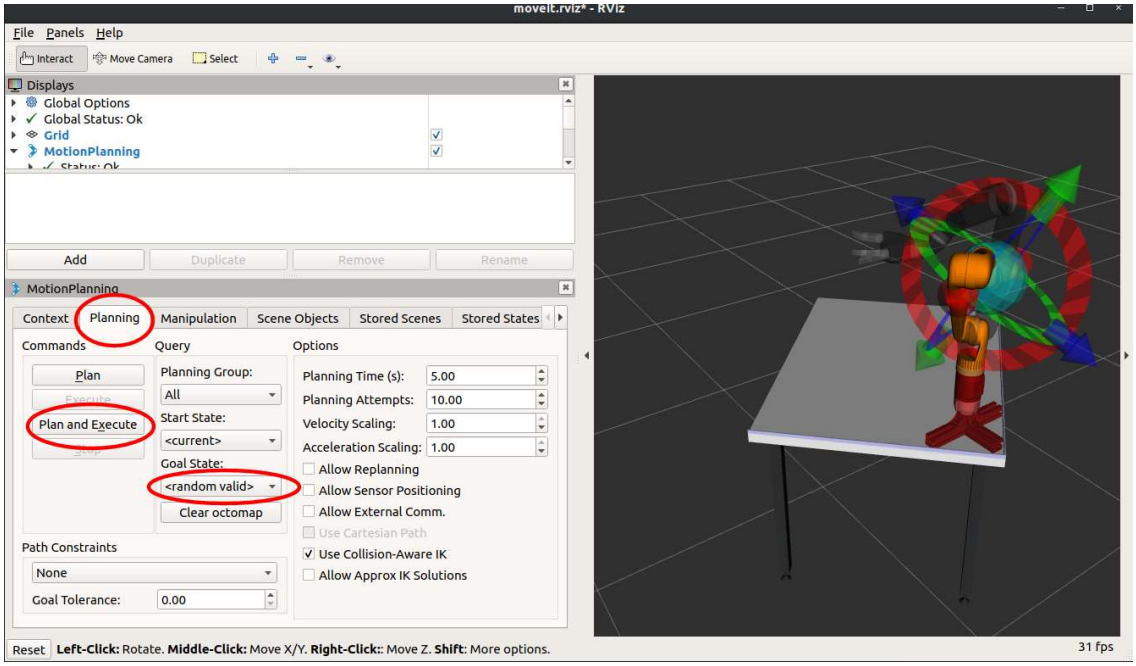
```
roslaunch jaco_on_table jaco_on_table_gazebo_controlled.launch load_grasp_
```

```
roslaunch jaco_on_table_moveit jaco_on_table_moveit.launch
```

```
roslaunch jaco_on_table_moveit jaco_on_table_rviz.launch
```

You should now have both Gazebo and Rviz open and a robot arm showing in both of them.

In Rviz go to the "planning" tab and set the "goal state" to "random valid" (see screenshot). Now press the "plan and execute" button and see the robot move both in Rviz and Gazebo.



Test if Everything Works

The idea here is that you can see the robot both in gazebo and in Rviz.
You should be able to control the robot using MoveIt in RVIZ

Then you should be able to look behind the scene with *moveit_commander_cmdline.py*

```
rosrun moveit_commander moveit_commander_cmdline.py
```

Let's use the correct moveit group

```
use Arm
```

Let's see the current state of the robot:

```
current
```

You can record the current position for further use:

```
record start
```

You can create a small movement by adjusting the current position

```
goal = start
goal[0] = 0.2
go goal
```

You can also move the end-effector linearly:

```
go up 0.1
go down 0.1
go left 0.1
go right 0.1
go backward 0.1
go forward 0.1
```

While you do all this observe both Gazebo and Rviz!

Demo 1: All the things you know

So now, let's use the same code we wrote in last lecture to control the robot joints, specifically the **moveit.py**

We'll show you this working on our machines but not give you the code.

The point of this part is that you can convert the code you had for the 4 joint arm to this 6 joint arm.

What are the changes you need to make?

Demo 2: Pose Command

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

So far we've commanded the robot with joint angles, so not even inverse kinematics are involved. Didn't we learn about how commanding the robot with [X,Y,Z,R,P,Y]?

Ok. Lets create a file named **lecture_5_2_pose_commands.py** and save it to **~/catkin_ws/src/hello_ros/scripts/lecture_5_2_pose_commands.py:**

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('hello_ros')

import sys
import copy
import rospy
import tf_conversions
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import shape_msgs.msg as shape_msgs
import math

from std_msgs.msg import String

def move_group_python_interface_tutorial():
  ## BEGIN_TUTORIAL
  ## First initialize moveit_commander and rospy.
  print "============ Starting tutorial setup"
  moveit_commander.roscpp_initialize(sys.argv)
  rospy.init_node('move_group_python_interface_tutorial',
                  anonymous=True)
```

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

## trajectories for RVIZ to visualize.
display_trajectory_publisher = rospy.Publisher(
                                '/move_group/display_planned_path',
                                moveit_msgs.msg.DisplayTrajectory)

print "============ Starting tutorial "
## We can get the name of the reference frame for this robot
print "============ Reference frame: %s" % group.get_planning_frame()
## We can also print the name of the end-effector link for this group
print "============ End effector frame: %s" % group.get_end_effector_lin
## We can get a list of all the groups in the robot
print "============ Robot Groups:"
print robot.get_group_names()
## Sometimes for debugging it is useful to print the entire state of the
## robot.
print "============ Printing robot state"
print robot.get_current_state()
print "============"

## Let's setup the planner
#group.set_planning_time(0.0)
group.set_goal_orientation_tolerance(0.01)
group.set_goal_tolerance(0.01)
group.set_goal_joint_tolerance(0.01)
group.set_num_planning_attempts(100)

## Planning to a Pose goal
print "============ Generating plan 1"

pose_goal = group.get_current_pose().pose
pose_goal.orientation = geometry_msgs.msg.Quaternion(*tf_conversions.trar
pose_goal.position.x =0.40
```

```
pose_goal.position.y =-0.10
pose_goal.position.z =1.35
print pose_goal
group.set_pose_target(pose_goal)



## Now, we call the planner to compute the plan
plan1 = group.plan()

print "============ Waiting while RVIZ displays plan1..."
rospy.sleep(0.5)



## You can ask RVIZ to visualize a plan (aka trajectory) for you.
print "============ Visualizing plan1"
display_trajectory = moveit_msgs.msg.DisplayTrajectory()
display_trajectory.trajectory_start = robot.get_current_state()
display_trajectory.trajectory.append(plan1)
display_trajectory_publisher.publish(display_trajectory);
print "============ Waiting while plan1 is visualized (again)..."
rospy.sleep(2.)

#If we're coming from another script we might want to remove the objects
if "table" in scene.get_known_object_names():
  scene.remove_world_object("table")
if "table2" in scene.get_known_object_names():
  scene.remove_world_object("table2")
if "groundplane" in scene.get_known_object_names():
  scene.remove_world_object("groundplane")

## Moving to a pose goal
group.go(wait=True)

## second movement
pose_goal = group.get_current_pose().pose
pose_goal.position.x =0.40
pose_goal.position.y =-0.10
```

```
pose_goal.position.z =1.35
pose_goal.orientation = geometry_msgs.msg.Quaternion(*tf_conversions.trar
group.set_pose_target(pose_goal)

plan1 = group.plan()
rospy.sleep(2.)

## You can ask RVIZ to visualize a plan (aka trajectory) for you.
display_trajectory = moveit_msgs.msg.DisplayTrajectory()
display_trajectory.trajectory_start = robot.get_current_state()
display_trajectory.trajectory.append(plan1)
display_trajectory_publisher.publish(display_trajectory);

rospy.sleep(2)

group.go(wait=True)
rospy.sleep(2.)

## When finished shut down moveit_commander.
moveit_commander.roscpp_shutdown()

## END_TUTORIAL
print "============ STOPPING"
R = rospy.Rate(10)
while not rospy.is_shutdown():
  R.sleep()
if __name__=='__main__':
try:
  move_group_python_interface_tutorial()
except rospy.ROSInterruptException:
  pass
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x lecture_5_2_pose_commands.py
```

Now we have to launch the files and play with MOVEIT:

rosrun hello_ros **lecture_5_2_pose_commands.py**

--- Explanation ---

There are 2 main point to notice here.

First is the following. Notice that we work on the *pose.position.* First we get all the structure from the current pose *group.get_current_pose().pose*. Then we populate the pose.position.x .y and .z:

```
pose_goal = group.get_current_pose().pose
pose_goal.position.x =0.40
pose_goal.position.y =-0.10
pose_goal.position.z =1.35
group.set_pose_target(pose_goal)
```

Second: Notice that we work on the *pose.orientation.* We provide euler angles but we convert them to quaternions before we assign them:

```
pose_goal = group.get_current_pose().pose
pose_goal.orientation = geometry_msgs.msg.Quaternion(*tf_conversions.tran
group.set_pose_target(pose_goal)
```

Demo 3: Cartesian Command

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

So we've managed to command the robot with Cartesian coordinates. But still the robot moves weirdly (un-predictably). How can we fix this? With Cartesian motion planning.

Ok. Lets create a file named **lecture_5_3_cartesian.py** and save it to **~/catkin_ws/src/hello_ros/scripts/lecture_5_3_cartesian.py:**

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('hello_ros')

import sys
import copy
import rospy
import tf_conversions
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import shape_msgs.msg as shape_msgs
import math

from std_msgs.msg import String

def move_group_python_interface_tutorial():
  ## BEGIN_TUTORIAL
  ## First initialize moveit_commander and rospy.
  print "============ Starting tutorial setup"
  moveit_commander.roscpp_initialize(sys.argv)
  rospy.init_node('move_group_python_interface_tutorial',
                  anonymous=True)

  robot = moveit_commander.RobotCommander()
  scene = moveit_commander.PlanningSceneInterface()
  group = moveit_commander.MoveGroupCommander("Arm")

  ## trajectories for RVIZ to visualize.
  display_trajectory_publisher = rospy.Publisher(
                                 '/move_group/display_planned_path',
                                 moveit_msgs.msg.DisplayTrajectory)

  print "============ Starting tutorial "
  ## We can get the name of the reference frame for this robot
  print "============ Reference frame: %s" % group.get_planning_frame()
  ## We can also print the name of the end-effector link for this group
  print "============ End effector frame: %s" % group.get_end_effector_lin
  ## We can get a list of all the groups in the robot
  print "============ Robot Groups:"
```

```
print robot.get_group_names()
## Sometimes for debugging it is useful to print the entire state of the
## robot.
print "============ Printing robot state"
print robot.get_current_state()
print "============"

## Let's setup the planner
#group.set_planning_time(0.0)
group.set_goal_orientation_tolerance(0.01)
group.set_goal_tolerance(0.01)
group.set_goal_joint_tolerance(0.01)
group.set_num_planning_attempts(100)

## Planning to a Pose goal
print "============ Generating plan 1"

pose_goal = group.get_current_pose().pose
waypoints = []

#pose_goal.orientation =
geometry_msgs.msg.Quaternion(*tf_conversions.transformations.quaternion_
0.  , 0.))
waypoints.append(pose_goal)
pose_goal.position.x =0.40
pose_goal.position.y =-0.10
pose_goal.position.z =1.55
print pose_goal

#Create waypoints
waypoints.append(pose_goal)

#createcartesian  plan
(plan1, fraction) = group.compute_cartesian_path(
                            waypoints,   # waypoints to follow
                            0.01,        # eef_step
                            0.0)         # jump_threshold
#plan1 = group.retime_trajectory(robot.get_current_state(), plan1, 1.0)


print "============ Waiting while RVIZ displays plan1..."
rospy.sleep(0.5)
```

```
## You can ask RVIZ to visualize a plan (aka trajectory) for you.
print "============ Visualizing plan1"
display_trajectory = moveit_msgs.msg.DisplayTrajectory()
display_trajectory.trajectory_start = robot.get_current_state()
display_trajectory.trajectory.append(plan1)
display_trajectory_publisher.publish(display_trajectory);
print "============ Waiting while plan1 is visualized (again)..."
rospy.sleep(2.)

#If we're coming from another script we might want to remove the objects
if "table" in scene.get_known_object_names():
  scene.remove_world_object("table")
if "table2" in scene.get_known_object_names():
  scene.remove_world_object("table2")
if "groundplane" in scene.get_known_object_names():
  scene.remove_world_object("groundplane")

## Moving to a pose goal
group.execute(plan1,wait=True)
rospy.sleep(4.)
## second movement
pose_goal2 = group.get_current_pose().pose
waypoints2 = []
#pose_goal.orientation =
geometry_msgs.msg.Quaternion(*tf_conversions.transformations.quaternion_
0.  , 0.))
waypoints.append(pose_goal)
pose_goal2.position.x =0.40
pose_goal2.position.y =-0.10
pose_goal2.position.z =1.2
print pose_goal2

#Create waypoints
waypoints2.append(copy.deepcopy(pose_goal2))

#createcartesian  plan
(plan2, fraction) = group.compute_cartesian_path(
                            waypoints2,   # waypoints to follow
                            0.01,        # eef_step
                            0.0)         # jump_threshold
#plan1 = group.retime_trajectory(robot.get_current_state(), plan1, 1.0)
rospy.sleep(2.)
```

```
## You can ask RVIZ to visualize a plan (aka trajectory) for you.
display_trajectory = moveit_msgs.msg.DisplayTrajectory()
display_trajectory.trajectory_start = robot.get_current_state()
display_trajectory.trajectory.append(plan2)
display_trajectory_publisher.publish(display_trajectory);

rospy.sleep(2)

group.execute(plan2,wait=True)
rospy.sleep(2.)

## When finished shut down moveit_commander.
moveit_commander.roscpp_shutdown()

## END_TUTORIAL
print "============ STOPPING"
R = rospy.Rate(10)
while not rospy.is_shutdown():
  R.sleep()
if __name__=='__main__':
try:
  move_group_python_interface_tutorial()
except rospy.ROSInterruptException:
  pass
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x lecture_5_3_cartesian.py
```

Now we have to launch the files and play with MOVEIT:

```
rosrun hello_ros lecture_5_3_cartesian.py
```

--- Explanation ---

There are 2 main point to notice here.

First is the following. Notice that we create a waypoints list. First we get the current pose *group.get_current_pose().pose* and we give set it as the first waypoint. Then we set it ti:

```
pose_goal = group.get_current_pose().pose
waypoints = []
```

```
waypoints.append(pose_goal)
pose_goal.position.x =0.40
pose_goal.position.y =-0.10
pose_goal.position.z =1.55

#Create waypoints
waypoints.append(pose_goal)

#createcartesian  plan
(plan1, fraction) = group.compute_cartesian_path(
                            waypoints,   # waypoints to follow
                            0.01,        # eef_step
                            0.0)         # jump_threshold

## Moving to a pose goal
group.execute(plan1,wait=True)
```

Second we HAVE to wait for an undefined amount of time (here its 4 seconds) to make sure that the moveit will take the correct pose and not some intermediate.

```
...
## Moving to a pose goal
group.execute(plan1,wait=True)
rospy.sleep(4.)
## second movement
pose_goal2 = group.get_current_pose().pose
...
```

Demo 4: Grasping

NOTE:

We define notes with a red vertical line,

terminal commands with a black one, and code with a blue one

Now, let's learn how to control the gripper!

Ok. Lets create a file named **lecture_5_4_close_gripper.py** and save it to **~/catkin_ws/src/hello_ros/scripts/lecture_5_4_close_gripper.py:**

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('hello_ros')

import sys
import copy
import rospy
import tf_conversions
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import shape_msgs.msg as shape_msgs
from sensor_msgs.msg import JointState
from numpy import zeros, array, linspace
from math import ceil

currentJointState = JointState()
def jointStatesCallback(msg):
  global currentJointState
  currentJointState = msg
if __name__ == '__main__':
  rospy.init_node('test_publish')

  # Setup subscriber
  #rospy.Subscriber("/joint_states", JointState, jointStatesCallback)

  pub = rospy.Publisher("/jaco/joint_control", JointState, queue_size=1)

  currentJointState = rospy.wait_for_message("/joint_states",JointState)
  print 'Received!'
  currentJointState.header.stamp = rospy.get_rostime()
  tmp = 0.7
  #tmp_tuple=tuple([tmp] + list(currentJointState.position[1:]))
  currentJointState.position = tuple(list(currentJointState.position[:6]) + [tmp]
+ [tmp]+ [tmp])
```

```
rate = rospy.Rate(10) # 10hz
for i in range(3):
  pub.publish(currentJointState)
  print 'Published!'
  rate.sleep()

print 'end!'
```

Next Let's create a file named **lecture_5_4_open_gripper.py** and save it to **~/catkin_ws/src/hello_ros/scripts/lecture_5_4_open_gripper .py:**

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('hello_ros')

import sys
import copy
import rospy
import tf_conversions
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import shape_msgs.msg as shape_msgs
from sensor_msgs.msg import JointState
from numpy import zeros, array, linspace
from math import ceil

currentJointState = JointState()
def jointStatesCallback(msg):
  global currentJointState
  currentJointState = msg
if __name__ == '__main__':
  rospy.init_node('test_publish')

  # Setup subscriber
  #rospy.Subscriber("/joint_states", JointState, jointStatesCallback)

  pub = rospy.Publisher("/jaco/joint_control", JointState, queue_size=1)

  currentJointState = rospy.wait_for_message("/joint_states",JointState)
  print 'Received!'
  currentJointState.header.stamp = rospy.get_rostime()
  tmp = 0.005
  #tmp_tuple=tuple([tmp] + list(currentJointState.position[1:]))
```

```
 currentJointState.position = tuple(list(currentJointState.position[:6]) + [tmp]
+ [tmp]+ [tmp])
 rate = rospy.Rate(10) # 10hz
 for i in range(3):
   pub.publish(currentJointState)
   print 'Published!'
   rate.sleep()

 print 'end!'
```

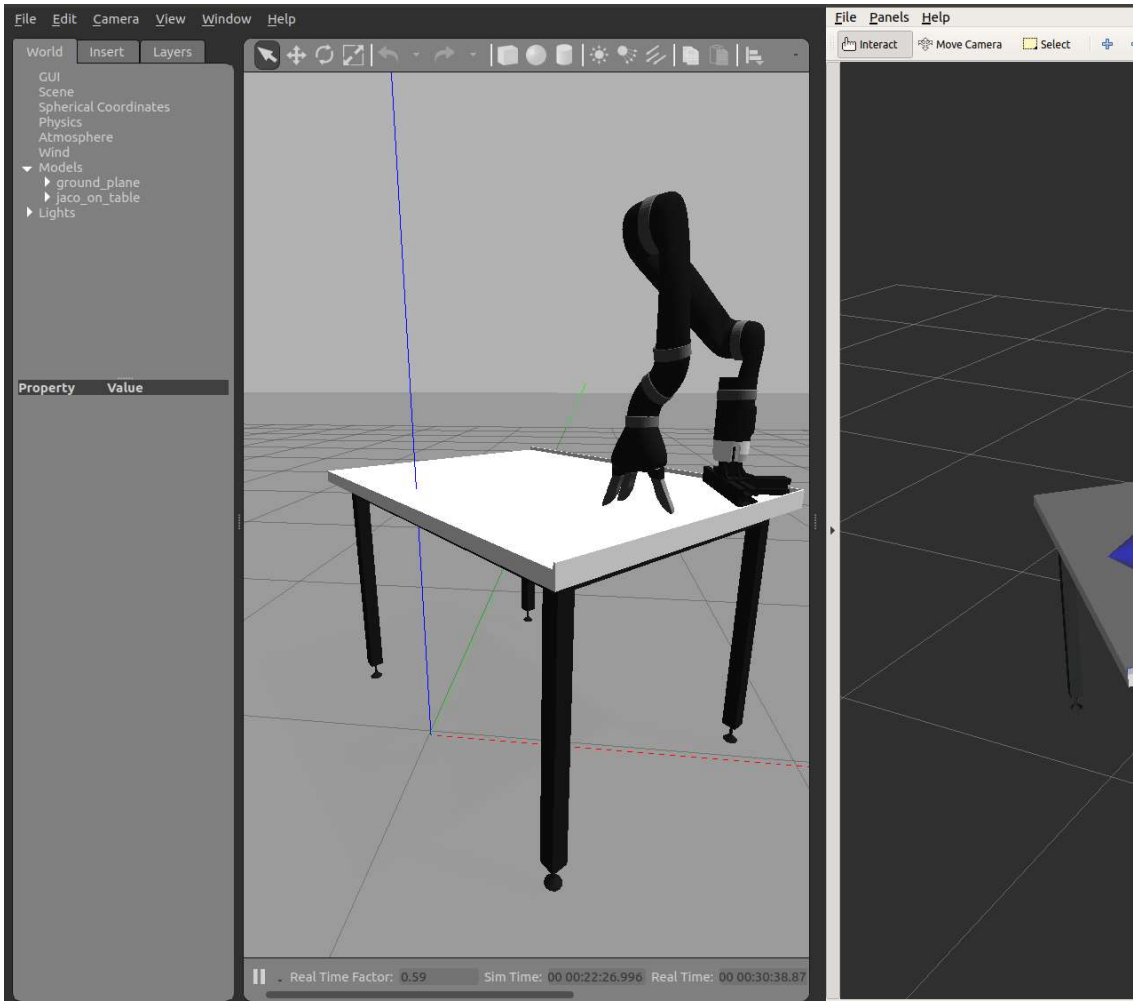
Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x lecture_5_4_open_gripper.py
chmod +x lecture_5_4_close_gripper.py
```

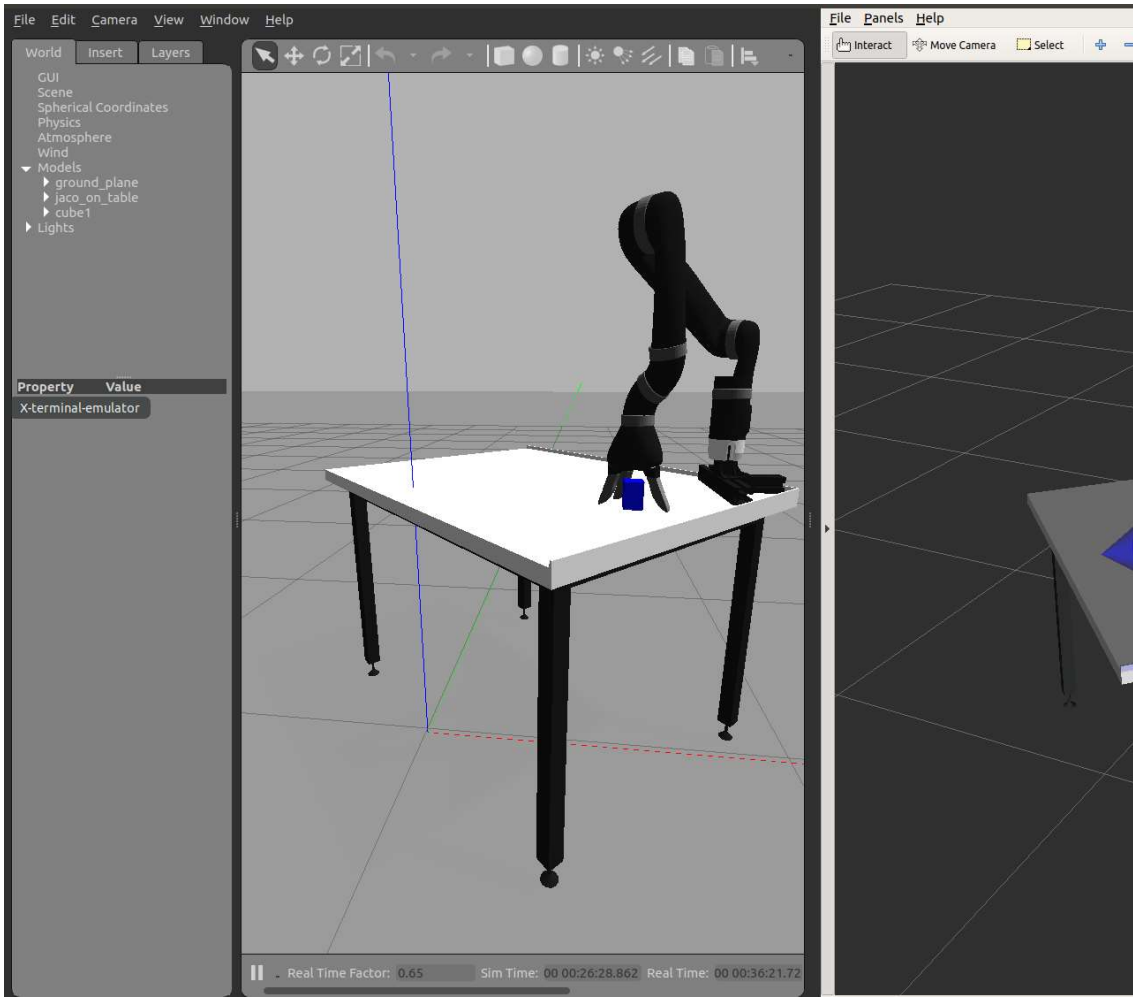
First lets send our robot to a know position with *moveit_commander_cmdline.py*

```
rosrun moveit_commander moveit_commander_cmdline.py
use Arm
a= [-1.18889039706 -0.212605149066 -0.63710924148 -2.83891025031
go a
```



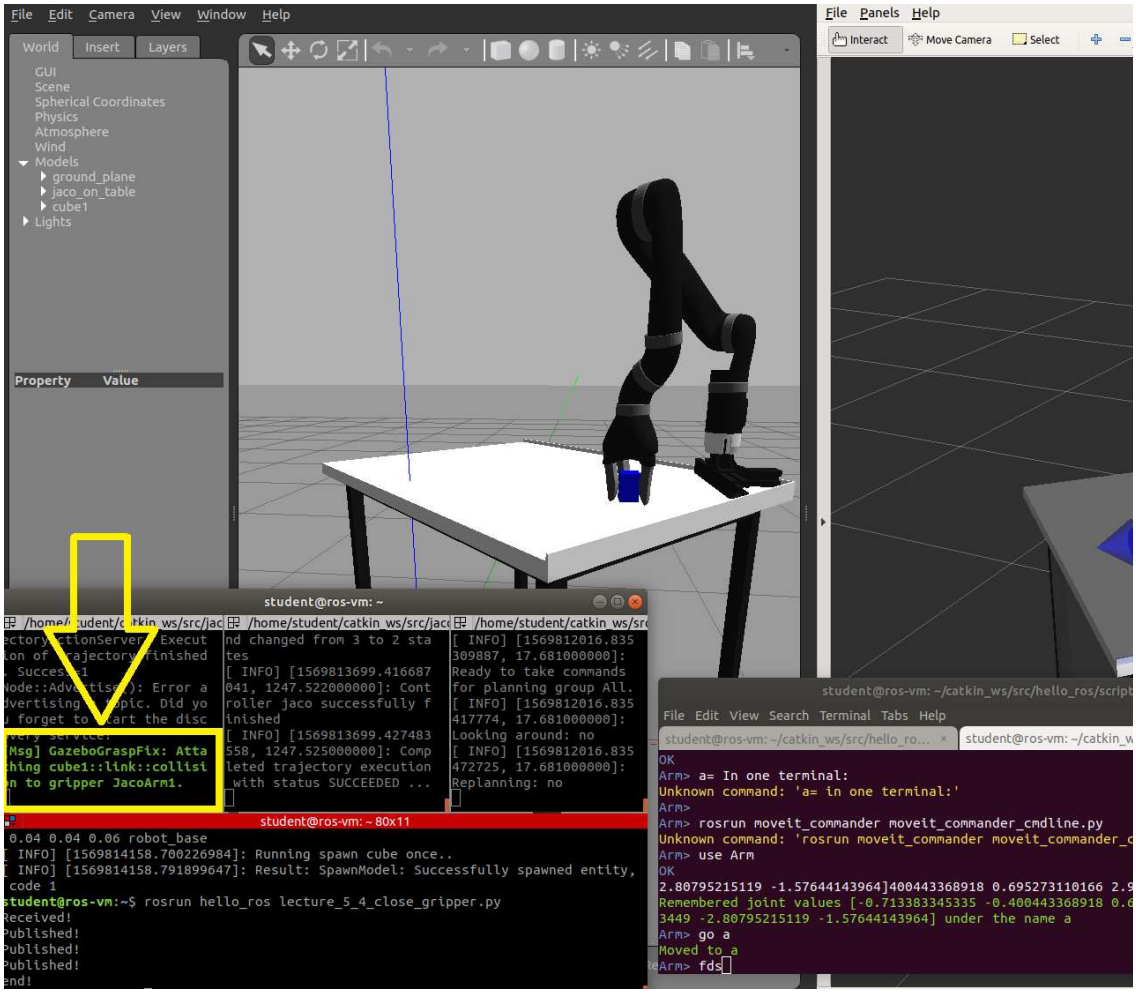
Let's input a target. In a new terminal:

```
rosrun gazebo_test_tools cube_spawner cube1 0.182 -0.135 0.76 0.04 0.04 0.06 robot_base
```



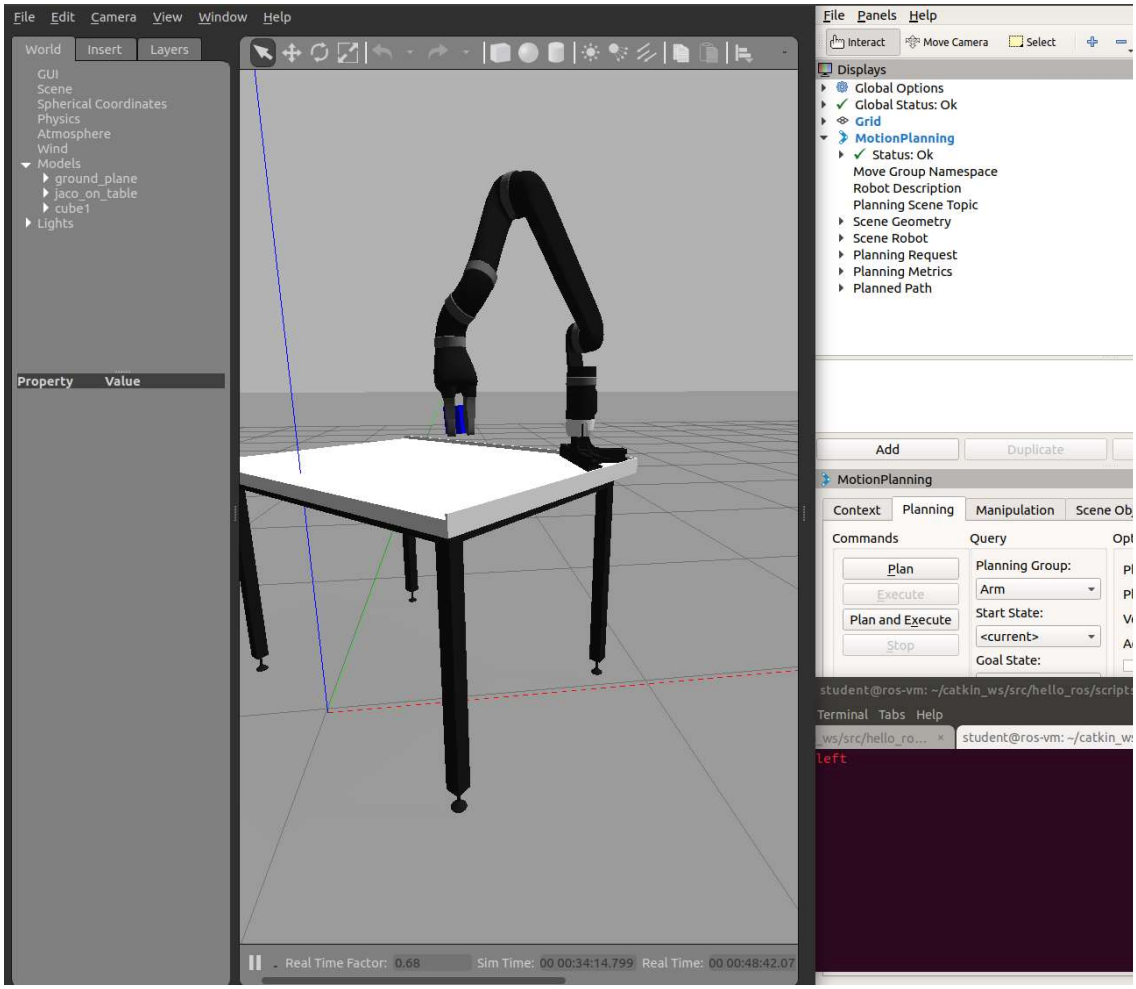
Let's input a target:

```
rosrun hello_ros lecture_5_4_close_gripper.py
```



No let's move up. n the commander terminal:

> go up 0.2



Finally, lets drop it:

```
rosrun hello_ros lecture_5_4_open_gripper.py
```