Lecture 4 (21/09) - Robot Kinematics, Motion Planning and Execution

Starts 21 September, 2020 1:00 PM

Robot Kinematics, Motion Planning and Execution

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

Reading Material

1. Robot Vision and Control, Peter Corke
Chapter 2 and 3.3 (pages 17-78)
link: DTU find Link

2. Planning Algortihms, by Steven M. LaValle,
Mainly Chapters 2.2 (A*) and 5.4 (Potential Fields), 5.5(RRT)
link1, link2

Time Plan

- 13:00 Introduction to Forward and Inverse Kinematics
- 13:30 Introduction to trajectory planning
- 14:15 Introduction Path Planning
- 15:15 Tutorial on Moveit!
- 16:00 The Exercise with support on Discord (use link: https://discord.gg/AEsFTp9)

0 % 0 of 2 topics complete

31391-sffas-4-trajectory_planning_seperate_part1
PDF document

31391-sffas-4-trajectory_planning_seperate_part2
PDF document

Exercise 1: Create a Moveit Enabled Robot

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

Ok, then we can send our robot to a specific joint position.
But what about if there is an obstacle in the scene, or what about the path specifics (speed accelaration, etc)?

In ROS, we are lucky enough to have a tool for all of these: MoveIT!

Let's see what we can do with it!

We need a biger robot to see the full capabilities of the tool!

First install the following package, that we'll need later.

```
sudo apt install ros-melodic-joint-trajectory-controller
```

Lets create a file named **hello_gazebo_robot_3.urdf** and save it to **~/catkin_ws/src/hello_ros/urdf/hello_gazebo_robot_3.urdf:**

```
<?xml version="1.0"?>
<robot name="hello_ros_robot">
  <link name="world"/>
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.1"/>
      </geometry>
      <material name="silver">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </visual>
```

```
  <collision>
    <geometry>
      <cylinder length="0.05" radius="0.1"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0.025"/>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <origin rpy="0 0 0" xyz="0 0 0.025"/>
    <inertia ixx="0.0027" iyy="0.0027" izz="0.005" ixy="0" ixz="0" iyz="0"/>
  </inertial>
</link>
<joint name="fixed" type="fixed">
  <parent link="world"/>
  <child link="base_link"/>
</joint>
<link name="torso">
  <visual>
    <geometry>
      <cylinder length="0.5" radius="0.05"/>
    </geometry>
    <material name="silver">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.5" radius="0.05"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
    <inertia ixx="0.02146" iyy="0.02146" izz="0.00125" ixy="0" ixz="0" iyz="
  </inertial>
</link>
<joint name="hip" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="base_link"/>
  <child link="torso"/>
  <origin rpy="0 0 0" xyz="0.0 0.0 0.05"/>
```

```
</joint>
<link name="upper_arm">
 <visual>
  <geometry>
   <cylinder length="0.4" radius="0.05"/>
  </geometry>
  <material name="silver"/>
  <origin rpy="0 0 0" xyz="0 0 0.2"/>
 </visual>
 <collision>
  <geometry>
   <cylinder length="0.4" radius="0.05"/>
  </geometry>
  <origin rpy="0 0 0" xyz="0 0 0.2"/>
 </collision>
 <inertial>
  <mass value="1.0"/>
  <origin rpy="0 0 0" xyz="0 0 0.2"/>
  <inertia ixx="0.01396" iyy="0.01396" izz="0.00125" ixy="0" ixz="0" iyz="
 </inertial>
</link>
<joint name="shoulder" type="continuous">
 <axis xyz="0 1 0"/>
 <parent link="torso"/>
 <child link="upper_arm"/>
 <origin rpy="0 1.5708 0" xyz="0.0 -0.1 0.45"/>
</joint>
 <link name="lower_arm">
 <visual>
  <geometry>
   <cylinder length="0.4" radius="0.05"/>
  </geometry>
  <material name="silver"/>
  <origin rpy="0 0 0" xyz="0 0 0.2"/>
 </visual>
 <collision>
  <geometry>
   <cylinder length="0.4" radius="0.05"/>
  </geometry>
  <origin rpy="0 0 0" xyz="0 0 0.2"/>
 </collision>
 <inertial>
  <mass value="1.0"/>
```

```
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
    <inertia ixx="0.01396" iyy="0.01396" izz="0.00125" ixy="0" ixz="0" iyz="
  </inertial>
 </link>
 <joint name="elbow" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="upper_arm"/>
  <child link="lower_arm"/>
  <origin rpy="0 0 0" xyz="0.0 0.1 0.35"/>
 </joint>
 <link name="hand">
  <visual>
   <geometry>
    <box size="0.05 0.05 0.05"/>
   </geometry>
   <material name="silver"/>
  </visual>
  <collision>
   <geometry>
    <box size="0.05 0.05 0.05"/>
   </geometry>
  </collision>
  <inertial>
   <mass value="1.0"/>
   <inertia ixx="0.00042" iyy="0.00042" izz="0.00042" ixy="0" ixz="0" iyz="
  </inertial>
 </link>
 <joint name="wrist" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="lower_arm"/>
  <child link="hand"/>
  <origin rpy="0 0 0" xyz="0.0 0.0 0.425"/>
 </joint>

 <transmission name="tran0">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="hip">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
  </joint>
  <actuator name="motor0">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
```

```
    <mechanicalReduction>1</mechanicalReduction>
   </actuator>
  </transmission>
  <transmission name="tran1">
   <type>transmission_interface/SimpleTransmission</type>
   <joint name="shoulder">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
   </joint>
   <actuator name="motor1">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
    <mechanicalReduction>1</mechanicalReduction>
   </actuator>
  </transmission>
  <transmission name="tran2">
   <type>transmission_interface/SimpleTransmission</type>
   <joint name="elbow">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
   </joint>
   <actuator name="motor2">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
    <mechanicalReduction>1</mechanicalReduction>
   </actuator>
  </transmission>
  <transmission name="tran3">
   <type>transmission_interface/SimpleTransmission</type>
   <joint name="wrist">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
   </joint>
   <actuator name="motor3">

<hardwareInterface>hardware_interface/PositionJointInterface</hardwareIr
    <mechanicalReduction>1</mechanicalReduction>
   </actuator>
  </transmission>

  <gazebo>
   <plugin name="control" filename="libgazebo_ros_control.so">
   </plugin>
```

```
  </gazebo>
  <gazebo>
    <plugin name="joint_state_publisher"
filename="libgazebo_ros_joint_state_publisher.so">
      <jointName>hip, shoulder, elbow, wrist</jointName>
    </plugin>
  </gazebo>
</robot>
```

Lets make sure that it has no errors:

```
roscd hello_ros/urdf/
check_urdf hello_gazebo_robot_3.urdf
```

In order to use the controller we will need to create small configuration file like this:

Lets create a file named **controllers.yaml** and save it to **~/catkin_ws/src/hello_ros/urdf/controllers.yaml:**

```
arm_controller:
  type: "position_controllers/JointTrajectoryController"
  joints:
    - hip
    - shoulder
    - elbow
    - wrist
```

Let's create the ros launch file: **spawn_gazebo_3.launch** to run this:

```
<launch>
    <!-- Load the hello_ros URDF model into the parameter server -->
    <param name="robot_description" textfile="$(find
hello_ros)/urdf/hello_gazebo_robot_3.urdf" />
    <!-- Start Gazebo with an empty world -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
    <!-- Spawn a helo_ros_robot in Gazebo, taking the description from the
    parameter server -->
    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
        args="-param robot_description -urdf -model hello_ros_robot" />

    <rosparam file="$(find hello_ros)/urdf/controllers.yaml"
command="load"/>
```

```
    <node name="controller_spawner" pkg="controller_manager"
type="spawner"
      args="arm_controller"/>
    <!-- Convert /joint_states messages published by Gazebo to /tf
messages,
     e.g., for rviz-->
    <node name="robot_state_publisher" pkg="robot_state_publisher"
     type="robot_state_publisher"/>
</launch>
```

Now let's run the simulation again to see the difference!

```
roslaunch hello_ros spawn_gazebo_3.launch
```

Kill it now! (CTRL+C)

Before moving on, we have to create install moveit!

```
sudo apt-get install ros-melodic-moveit*
sudo apt-get install ros-melodic-geometric-shapes
sudo apt-get install ros-melodic-moveit ros-melodic-moveit-msgs
sudo apt-get install ros-melodic-rviz-visual-tools
sudo apt-get install ros-melodic-controller*
```

Now let's setup the moveit configuration. Happily, there is a tool for this:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

Follow my guide now:

1. Press "Create new MoveIt Configuration Package"
2. Browse and select the hello_gazebo_robot_3.urdf
3. Press "Load Files"

4. Select the "Self-Collisions"
5. Press "Generate Collision Matrix"

6. Select "Virtual Joints" --> "Add Virtual Joint"
7.1 NAME: base_link
7.2 Child link: base_link
7.3 Parent Frame Name: world

7.4 Type: fixed
9. Press "Save"

10. Select "Planning Groups"
11. Press "Add group"
12. Input group name: "arm"
13. Select "kdl_kinematics_plugin/KDLKinematicsPlugin"
14. Press "Add Joints"
15. Add all and Save

16. Select "End Effectors"
17. Name: Hand
18. Parent Link: hand
19. Press "Save"

20. In "Author information" just put some info

21. In "Configuration Files" provide a new folder called "~/catkin_ws/src/hello_ros_moveit_config"
22. Press Generete Package.
23. Press Exit.

Before building the new package we have to add two files.

~/catkin_ws/src/hello_ros_moveit_config/config/controllers.yaml:

```yaml
# Hand-made
controller_manager_ns: /
controller_list:
  - name: arm_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    joints:
      - hip
      - shoulder
      - elbow
      - wrist
```

Now you have to point to the manually created controllers by replacing the launch file

~/catkin_ws/src/hello_ros_moveit_config/launch/hello_ros_robot_moveit_controller_manager.launch.xml with:

```
<launch>
  <param name="moveit_controller_manager"

value="moveit_simple_controller_manager/MoveItSimpleControllerManager'
  <param name="controller_manager_name" value="/" />
  <param name="use_controller_manager" value="true" />
  <rosparam file="$(find hello_ros_moveit_config)/config/controllers.yaml"/>
</launch>
```

Lets built our new package:

```
catkin build
```

If you like autocomplete

to work you might want to type this command in the terminal:

```
rospack profile
```

Now we have to launch the files.

```
roslaunch hello_ros spawn_gazebo_3.launch
roslaunch hello_ros_moveit_config move_group.launch
roslaunch hello_ros_moveit_config moveit_rviz.launch config:=true
```

In Rviz select the world-frame as Fixed Frame and add the "MotionPlanning"-component. You can now drag the robots endeffector to a goal position, plan and execute the trajectory. Explore the interface further!

Exercise 2: MoveIt Commander

MoveIt! offers a commandline based commander which is similar to the ones on industrial robots and allows us to easily send commands to the robot and read its state

We will explore some basic commands here

First, Launch the gazebo and the Move group

```
roslaunch hello_ros spawn_gazebo_3.launch
roslaunch hello_ros_moveit_config moveit_rviz.launch config:=true
roslaunch hello_ros_moveit_config move_group.launch
```

Then start the commander

```
rosrun moveit_commander moveit_commander_cmdline.py
```

Let's select the move group we have created

```
use arm
```

Next, the current state of the group is given by the following command:

```
current
```

We can record the current position for further use

```
record start
```

We can create a small movement by adjusting the current position

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

We can also check whether a movement is feasible, before executing, like this:

```
plan start
execute
```

You can type help to see the rest of the available commands

```
help
```

Move the robot to an unfeasible position!

Exercise 3: Plan and execute with the robot programatically

NOTE:

We define notes with a red vertical line,

terminal commands with a black one,

and code with a blue one

Ok, Let's try to move the robot programmatically..

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

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

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

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)
  scene = moveit_commander.PlanningSceneInterface()
  robot = moveit_commander.RobotCommander()

  ## Instantiate a MoveGroupCommander object.  This object is an interface
  ## to one group of joints.  In this case the group is the joints in the left
  ## arm.  This interface can be used to plan and execute motions on the left
  ## arm.
```

```
group = moveit_commander.MoveGroupCommander("arm")


## We create this DisplayTrajectory publisher which is used below to
publish
## trajectories for RVIZ to visualize.
display_trajectory_publisher = rospy.Publisher(
                                '/move_group/display_planned_path',
                                moveit_msgs.msg.DisplayTrajectory)

## Wait for RVIZ to initialize. This sleep is ONLY to allow Rviz to come up.
#print "============ Waiting for RVIZ..."
#rospy.sleep(2)
#print "============ Starting tutorial "

## Getting Basic Information
## ^^^^^^^^^^^^^^^^^^^^^^^^
##
## 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 "============ Reference frame: %s" %
group.get_end_effector_link()

## 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 "============"
#rospy.sleep(2)

print "============ Generating plan "
group.set_joint_value_target([-0.5,-0.5,0.0,0.0])

## Let's setup the planner
group.set_planning_time(2.0)
group.set_goal_orientation_tolerance(1)
group.set_goal_tolerance(1)
```

```
group.set_goal_joint_tolerance(0.1)
group.set_num_planning_attempts(20)

## Now, we call the planner to compute the plan
## and visualize it if successful
## Note that we are just planning, not asking move_group
## to actually move the robot
#group.set_goal_position_tolerance(1.5)
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.  But the
## group.plan() method does this automatically so this is not that useful
## here (it just displays the same trajectory again).
#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(0.5)


## Moving to a pose goal
## ^^^^^^^^^^^^^^^^^^^^^

group.go(wait=True)

group.set_joint_value_target([0.0,0.0,0.0,0.0])
plan1 = group.plan()
group.go(wait=True)

group.set_joint_value_target([-0.5,-0.5,0.0,0.0])
plan1 = group.plan()
group.go(wait=True)


group.set_joint_value_target([0.0,0.0,0.0,0.0])
```

```
  plan1 = group.plan()
  group.go(wait=True)
  ## 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
```

Remember to make the python file to be executable.

If you like autocomplete to work you might want to type this command in the terminal:

```
rospack profile
```

Launch the setup (Gazebo, Moveit, Rviz) and run the script. In Gazebo you can see the robot moving. In Rviz you will see the planning + execution phase.

```
rosrun hello_ros moveit.py
```

Exercise 4: Plan with collision avoidance

NOTE:

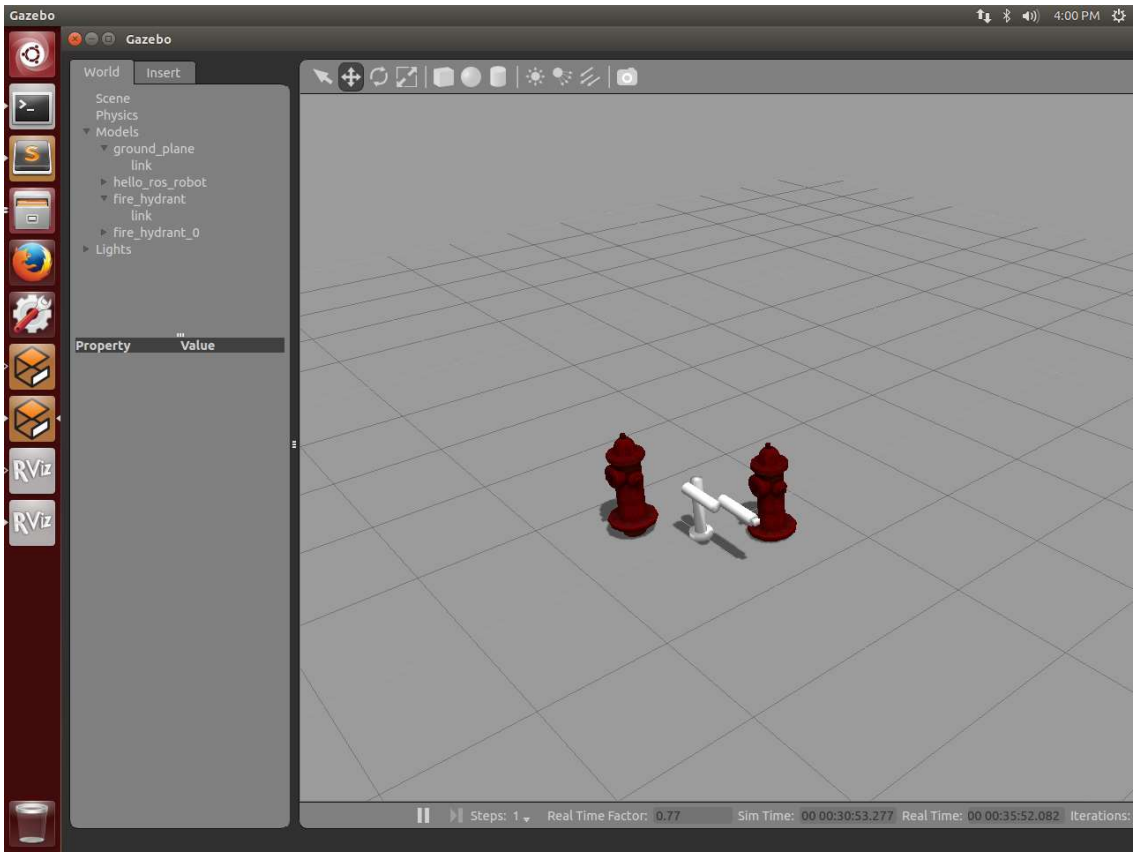
We define notes with a red vertical line,

terminal commands with a black one,

and code with a blue one

Ok let's play a little with gazebo now.

Add two obstacles in the workspace of the robot, similarly to the setup in the given image. (Insert-->http://gazebosim.prg/models -->fire_hydrant).



Now let's command our robot to a position:

```
rostopic pub /arm_controller/command trajectory_msgs/JointTrajectory '{joint_names: ["hip", "shoulder", "elbow", "wrist"], points: [{positions: [1.57, 0.0, 0.0, 0.0], time_from_start:[1.0,0.0]}]}'
```

That didn't work nicely enough... Let's undo it...:

```
rostopic pub /arm_controller/command trajectory_msgs/JointTrajectory '{joint_names: ["hip", "shoulder", "elbow", "wrist"], points: [{positions: [0.0,
```

```
0.0, 0.0, 0.0], time_from_start:[1.0,0.0]}]}'
```

So let's see how we can use move it to plan nicely and avoid collisions..

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

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

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

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)

  ## Instantiate a RobotCommander object.  This object is an interface to
  ## the robot as a whole.
  robot = moveit_commander.RobotCommander()

  ## Instantiate a PlanningSceneInterface object.  This object is an interface
  ## to the world surrounding the robot.
  scene = moveit_commander.PlanningSceneInterface()

  ## Instantiate a MoveGroupCommander object.  This object is an interface
  ## to one group of joints.  In this case the group is the joints in the left
  ## arm.  This interface can be used to plan and execute motions on the left
  ## arm.
  group = moveit_commander.MoveGroupCommander("arm")
```

```
  ## We create this DisplayTrajectory publisher which is used below to
publish
  ## trajectories for RVIZ to visualize.
  display_trajectory_publisher = rospy.Publisher(
                                 '/move_group/display_planned_path',
                                 moveit_msgs.msg.DisplayTrajectory)

  ## Wait for RVIZ to initialize. This sleep is ONLY to allow Rviz to come up.
  print "============ Waiting for RVIZ..."
  rospy.sleep(2)
  print "============ Starting tutorial "

  ## Getting Basic Information
  ## ^^^^^^^^^^^^^^^^^^^^^^^^^
  ##
  ## 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 "============ Reference frame: %s" %
group.get_end_effector_link()

  ## 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 "============"

  scene = moveit_commander.PlanningSceneInterface()
  robot = moveit_commander.RobotCommander()

  rospy.sleep(2)
  '''
  scene.remove_world_object('groundplane')
  scene.remove_world_object('table')
  '''
  p = geometry_msgs.msg.PoseStamped()
  p.header.frame_id = robot.get_planning_frame()
```

```
p.pose.position.x = 0.5
p.pose.position.y = 0.5
p.pose.position.z = 0.7
scene.add_box("table", p, (0.5, 0.5, 1.4))
p.pose.position.x = -0.5
p.pose.position.y = -0.5
p.pose.position.z = 0.7
scene.add_box("table2", p, (0.5, 0.5, 1.4))
p.pose.position.x = 0.
p.pose.position.y = 0.
p.pose.position.z = -0.01
scene.add_box("groundplane", p, (2, 2, 0.009))

## Planning to a Pose goal
## ^^^^^^^^^^^^^^^^^^^^^^^^
## We can plan a motion for this group to a desired pose for the
## end-effector
print "============ Generating plan 1"

group.set_joint_value_target([1.57,0.,0.,0.])



## Let's setup the planner
group.set_planning_time(2.0)
group.set_goal_orientation_tolerance(1)
group.set_goal_tolerance(1)
group.set_goal_joint_tolerance(0.1)
group.set_num_planning_attempts(20)

## Now, we call the planner to compute the plan
## and visualize it if successful
## Note that we are just planning, not asking move_group
## to actually move the robot
#group.set_goal_position_tolerance(1.5)
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.  But the
## group.plan() method does this automatically so this is not that useful
## here (it just displays the same trajectory again).
```

```
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(0.5)


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

## second movement
## ^^^^^^^^^^^^^^^^^^^^^


group.set_joint_value_target([0.,0.,0.,0.])

plan1 = group.plan()

rospy.sleep(0.5)

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(0.5)

group.go(wait=True)

## 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 moveit_fake_objects.py
```

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

```
rosrun hello_ros
moveit_fake_objects.py
```

Exercise 5: The EXERCISE

Part A

In **moveit_fake_objects.py**, the position of the fire_hydrant is hard-coded. Please modify the script to the exact position of the fire hydrants. Find the pose in gazebo.

Part B

Hard-coding is not cool! Now we want to modify the script to automatically add the fire_hydrants to the moveit-scene. Here are some hints:

1. Subscribe to the appropriate topic, that contains information about the models in gazebo (explore the relevant topics with the *rostopic* commands you have learned).

2. Filter out the fire_hydrants and their pose.

3. The poses can be used to add the objects to the moveit-scene.