The Construct
Learn and develop for robots

- *"The best among all the ROS resources."*
Nishanth
Project Manager

- *"Excellent for learning ROS and robotics in general."*
Jorge Gomes
Ph.D. Student

- *"Very practical, I have learned a lot and opened new fields for me to continue learning Robotics and Artificial Intelligence."*
José María
Professor

- *"The ROS IN 5 DAYS is incredibly informative. I would highly recommend any future books by The Construct team!"*
Miguel Duarte
Researcher of Robotics and A.I.

www.theconstructsim.com

The Construct ROS MANIPULATION IN 5 DAYS Téllez | Ezquerro | Rodríguez

The Construct

Ricardo Téllez, Ph.D.
Alberto Ezquerro
Miguel Ángel Rodríguez

2nd Version

ROS MANIPULATION IN 5 DAYS

ENTIRELY PRACTICAL ROBOT OPERATING SYSTEM TRAINING

ROS MANIPULATION IN 5 DAYS

Ricardo Téllez

Alberto Ezquerro

Miguel Angel Rodríguez

The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,

+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Written by Miguel Angel Rodríguez, Alberto Ezquerro and Ricardo Téllez
Edited by Yuhong Lin and Ricardo Téllez
Cover design by Lorena Guevara
Learning platform implementation by Ruben Alves

Version 3.0

The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,

+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Index

Introduction

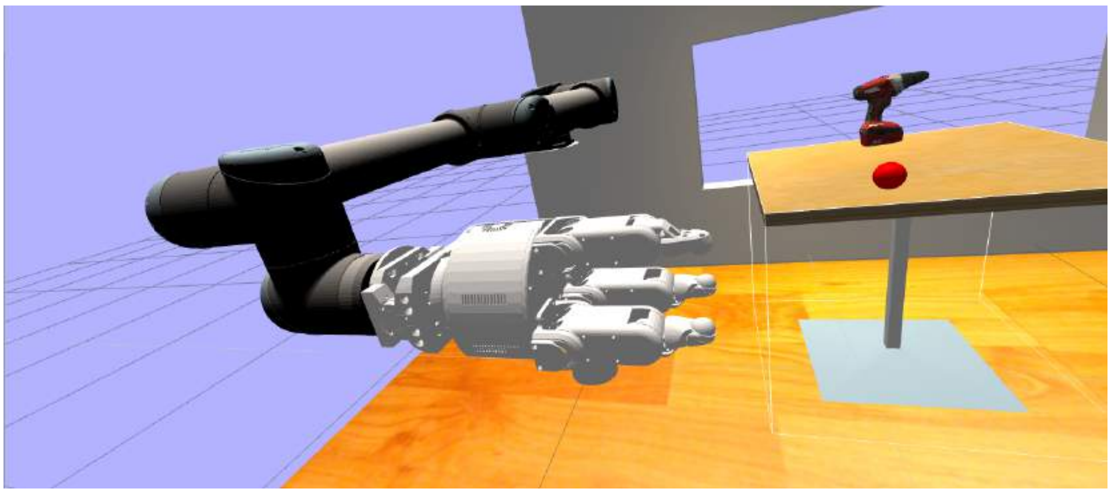
Introduction

Welcome to the fourth volume of the In 5 Days series of books for learning ROS.

If the first volume was dedicated to learning the basics of ROS (ROS Basics in 5 days), the second volume was dedicated to teach you how to use ROS for making a robot navigate in an environment (ROS Navigation in 5 days), the third volume was dedicated to teach how to make your robot perceive the world using ROS packages (ROS Perception in 5 Days), this fourth volume is dedicated to make your robot able to grasp objects. At present, making a robot interact with the environment is the final frontier in autonomy of robots. This is one of the most interesting skills for a robot because it allows it to really do tasks in human environment. If we achieve to make a robot grasp objects, then it will be able to clean rooms, bring stuff to the owner or go shopping for us.

In this book you are going to learn how to use grasping algorithms on a ROS based robot. You will learn how to make the robot plan an arm trajectory to a given location by using MoveIt! and also programmatically. You will learn to add perception to the grasping system to avoid obstacles while planning an arm movement. You will also learn how to do a proper grasp of an object.

In order to have a good grasping system, robots need to have three things: the proper sensors to perceive the object to grasp and the obstacles around, a robotic arm equipped with a gripper and the good algorithms to make sense of the sensed :-). This book is about how to program a robot grasp objects on a robot equipped with a robotic arm with gripper, a camera and point cloud device. That, we found, is the most common and basic setup for grasping robots.



Robotic Hand

Even if we only concentrate in teaching about ROS based manipulation, the task is not simple and it will require you 5 days of full work to understand and make it work for your robot.

- On day one, you will learn basics concepts about manipulation and grasping, as well as how to create a MoveIt! package for a robotic arm.
- On day two, you will learn to perform motion planning of the arm programmatically.
- Day three wil be dedicated to add perception to the grasping pipeline, so the robot can avoid obstacles in the arm tajectory.
- Finally days four and five will be for doing grasping with the gripper.

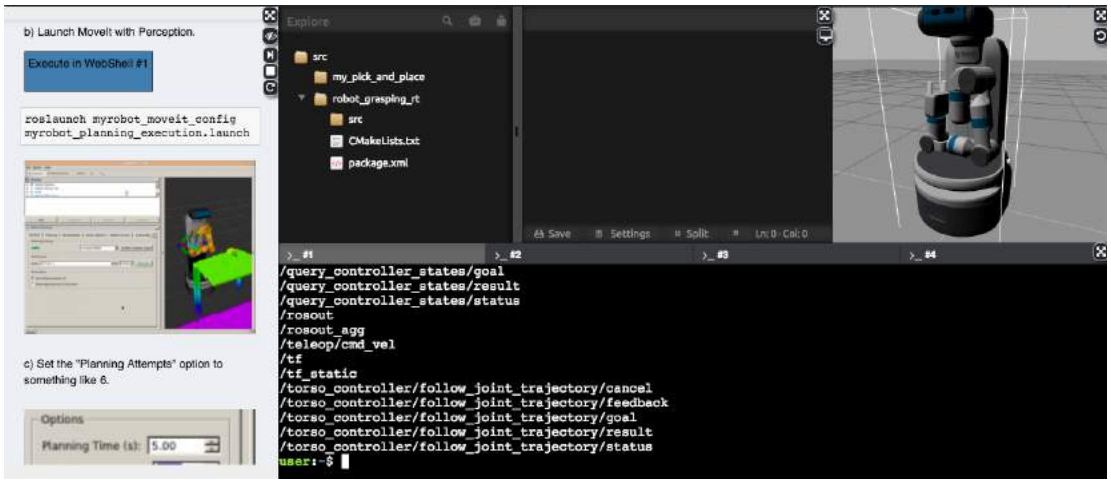
As you will see in Chapter 0, the whole book is structured on different chapters with a suggested time of completion. Follow the suggestions of that chapter if you want to maximize your learning experience. But do not stress too much if you cannot follow that rhythm. After all is just a suggestion.

Simulations

You are going to learn all the material using robot simulations. Robot simulations allow you to practice and see the results of your programming in real time, while you are learning and doing your tests. Simulations are also very convenient because those allow us to work with any robot of the world.

So for each chapter of the book you are going to need to launch a simulation and do your exercises with it. For that purpose we have prepared an online platform called the ROS Development Studio (ROSDS): http://rosds.online
Using the ROSDS, you can develop all the exercises of the book and launch the simulations without having to install anything in your computer, being the only requirement to have a web browser. Since you will be developing online with the browser, you can use any type of computer to program ROS and launch simulations.

So go now to the ROSDS and create a free account.



b) Launch MoveIt with Perception.

```
roslaunch myrobot_moveit_config myrobot_planning_execution.launch
```

c) Set the "Planning Attempts" option to something like 6.

```
/query_controller_states/goal
/query_controller_states/result
/query_controller_states/status
/rosout
/rosout_agg
/teleop/cmd_vel
/tf
/tf_static
/torso_controller/follow_joint_trajectory/cancel
/torso_controller/follow_joint_trajectory/feedback
/torso_controller/follow_joint_trajectory/goal
/torso_controller/follow_joint_trajectory/result
/torso_controller/follow_joint_trajectory/status
user:~$
```

ROSDS Environment

Once you have an account on ROSDS, you will have to open the **ROSjects** that we will provide to you for each Chapter. In the next Chapter, you have a complete guide about **ROSjects** and how to launch them.

We recommend that you perform all the exercises in the ROSDS using the free account, so you can see the results in the simulated robot with minimum hasle.

Regarding the debate about simulations versus real robots, our suggestion is that, once you understand a perception concept, get a real robot and try to apply to it (if you have the chance). Simulations are very good for learning and testing, but nothing replaces the experience of using a real robot.

Now is your turn to start learning. Go for it. Remember that we are here to answer your questions and doubts (contact us at feedback@theconstructsim.com). And remember that you can become a ROS master by studying the rest of subjects that can be found in our other books:

- ROS Basics in 5 days
- ROS Navigation in 5 days
- ROS Perception in 5 days

Keep pushing your ROS learning!

ROSjects

ROSjects

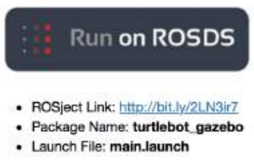
Throughout the whole book, you're going to **find a ROSject at the beginning of each Chapter**. With these ROSjects, you are going to be able to easily have access to all the material you'll need for each Chapter.

What is a ROSject?

A ROSject is, basically, a ROS project in the ROS Development Studio (ROSDS). ROSjects can easily be shared using a link. By clicking on the link, or copying it to the URL of your web browser, you will have a copy of the specific ROSject in your ROSDS workspace. This means you will have instant access to the ROSject. Also, you will be able to modify it as you wish.

How to open a ROSject?

At the beginning of each Chapter, you will see a section like this one:



- ROSject Link: http://bit.ly/2LN3ir7
- Package Name: **turtlebot_gazebo**
- Launch File: **main.launch**

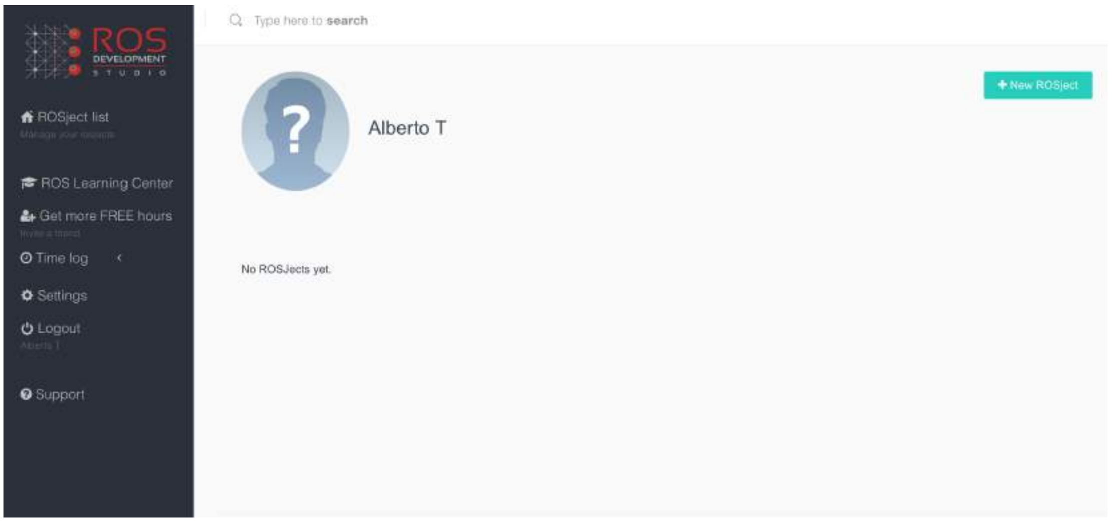
ROSject section

As you can see, it contains 3 things:

- **ROSject Link:** Link to get the ROSject
- **Package Name:** Name of the ROS package that contains the launch file to start the Gazebo simulation.
- **Launch File:** Name of the launch file that will start the Gazebo simulation related to the ROSject.

Step 1

Log into the ROSDS platform at http://rosds.online . If you don't have an account, you can create one for free. Once you log in, you will see an screen like the below one.

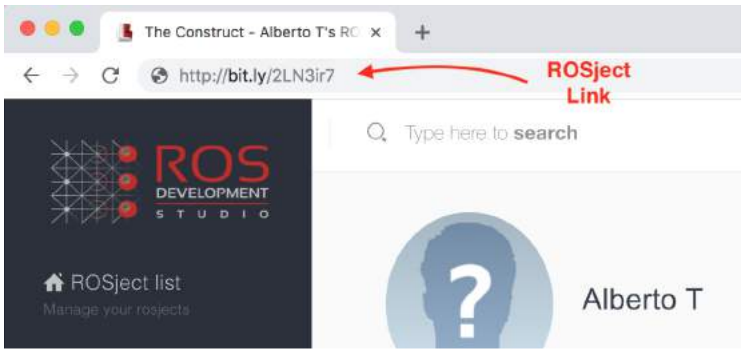


ROSjects Empty List

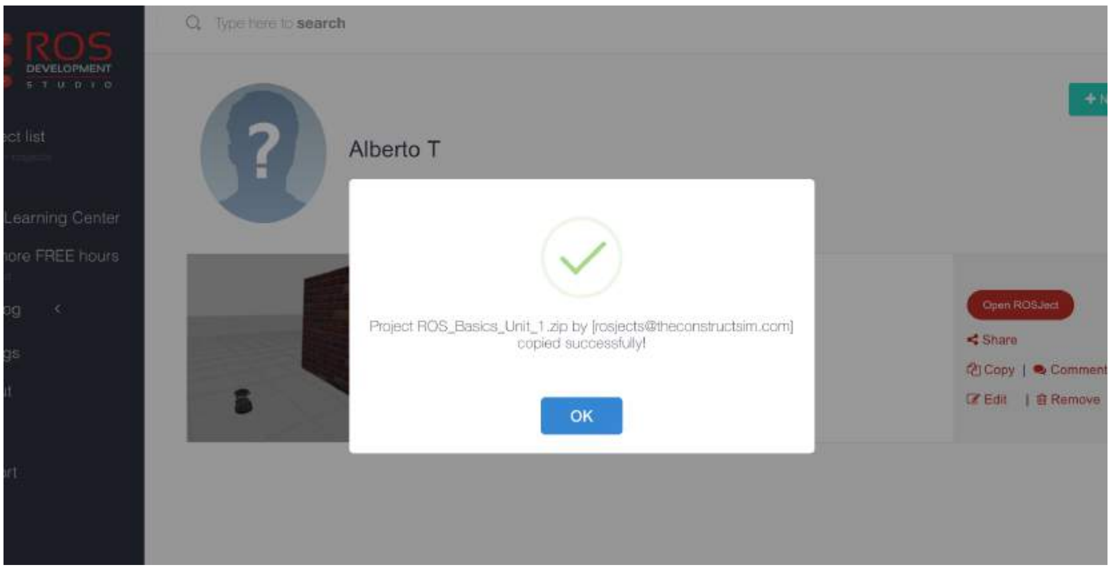
At this point you don't have any ROSject yet. So let's get one!

Step 2

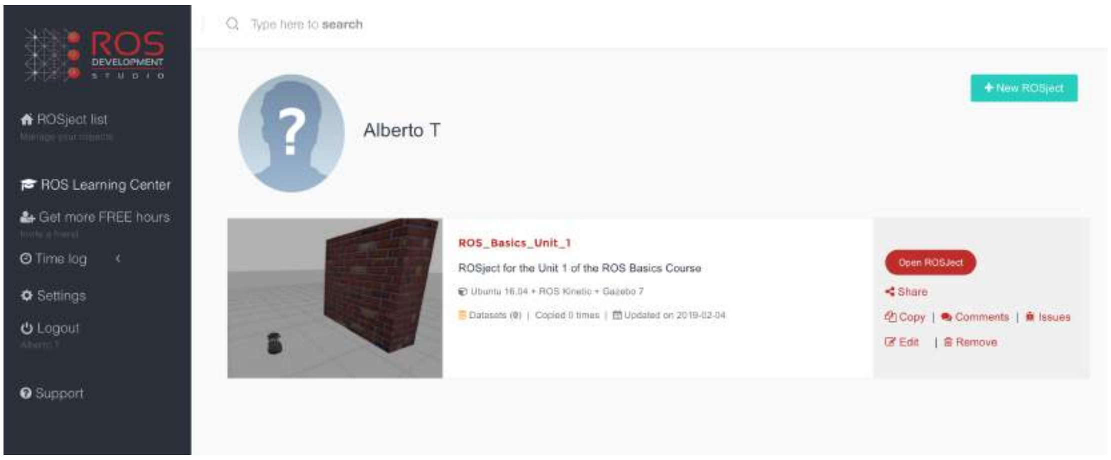
Copy the ROSject Link to your web browser. Once you have copied the URL to your web browser, you will automatically have that ROSject available in your workspace.



ROSject Link



Project ROS_Basics_Unit_1.zip by [rosjects@theconstructsim.com] copied successfully!

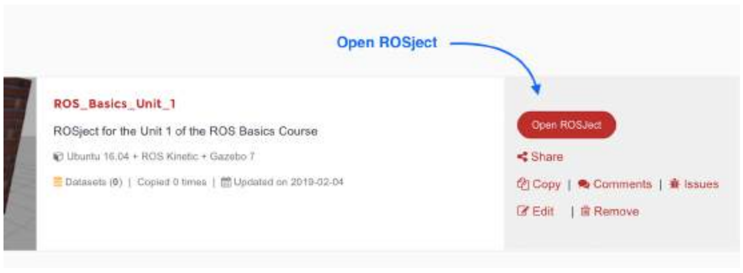


ROSject for the Unit 1 of the ROS Basics Course

ROSject in ROSDS workspace

Step 3

Open the ROSject. You can open the ROSject by clicking on the **Open ROSject** button.

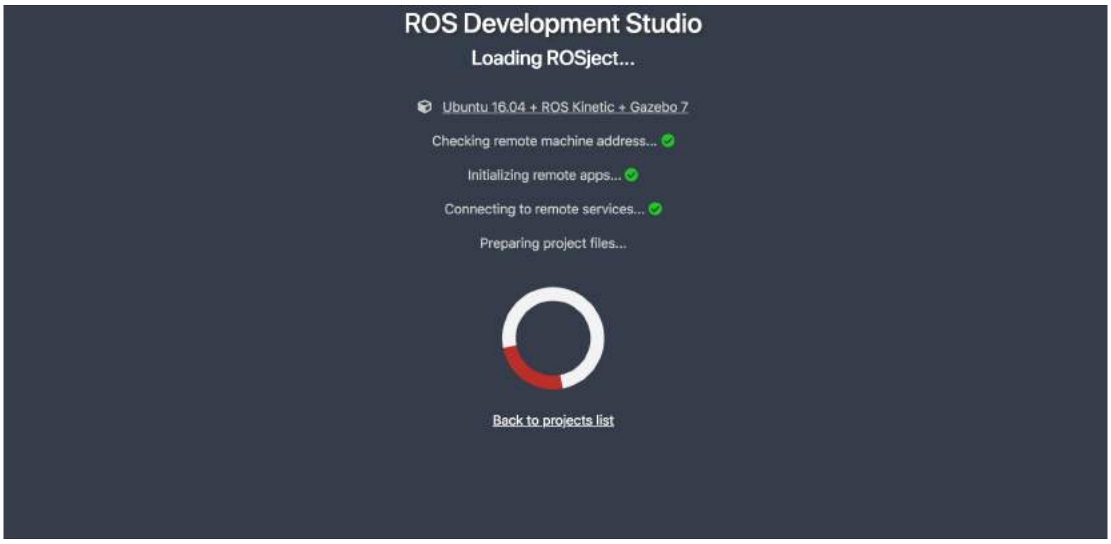


ROSject for the Unit 1 of the ROS Basics Course

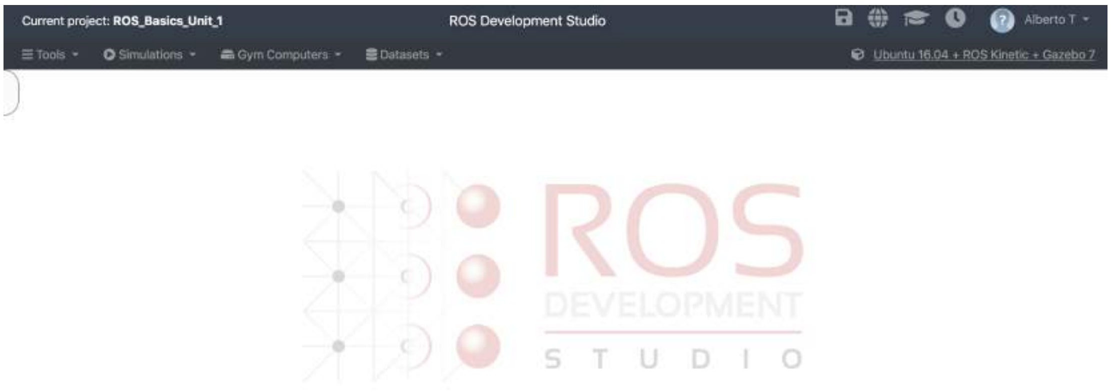
Open ROSject
ROS_Basics_Unit_1
Ubuntu 16.04 + ROS Kinetic + Gazebo 7
Datasets (0) | Copied 0 times | Updated on 2019-02-04
Open ROSJect
Share
Copy | Comments | Issues
Edit | Remove

Open ROSject

You will then go to a loading screen like the below one.


ROS Development Studio
Loading ROSject...
Ubuntu 16.04 + ROS Kinetic + Gazebo 7
Checking remote machine address...
Initializing remote apps...
Connecting to remote services...
Preparing project files...
Back to projects list

After a few seconds, you will get an environment like the below one.



ROSDS Environment

Contents of the ROSjects

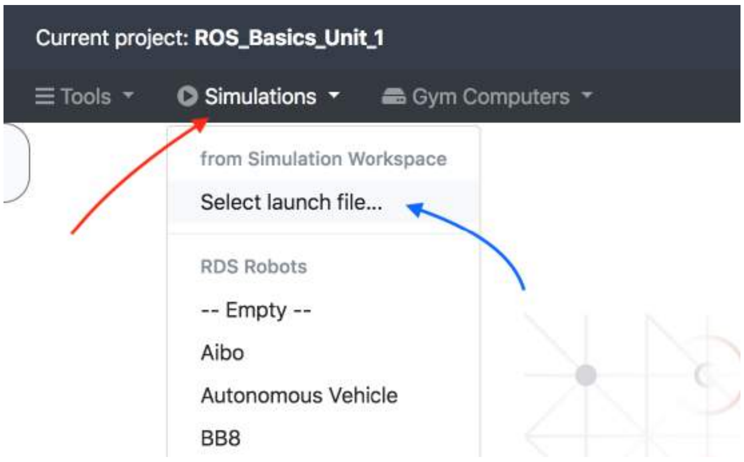
The ROSjects that are available for each chapter of the book will basically contain 2 things:

- The Gazebo simulation used for the Chapter
- All the scripts and files used for the Chapter

In order to open the simulation, follow the next steps:

Step 1

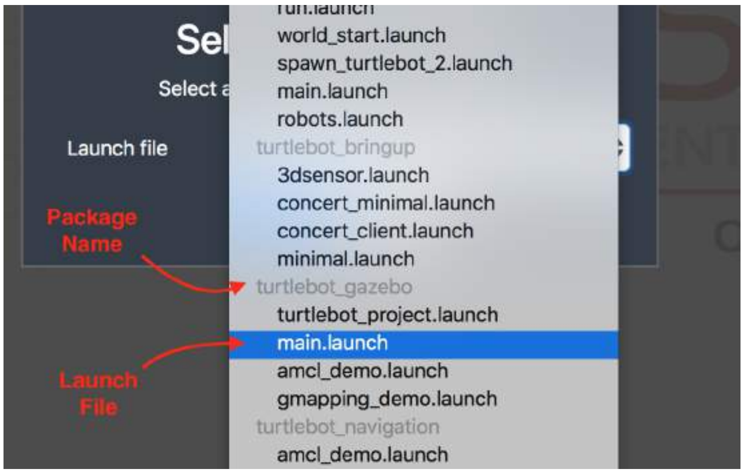
In the **Simulations** menu, click on the **Select launch file...** option. A list with all the packages and launch files available will appear.



Simulations menu

Step 2

Within the list, select the **package name** and **launch file** specified at the ROSject section of the chapter.

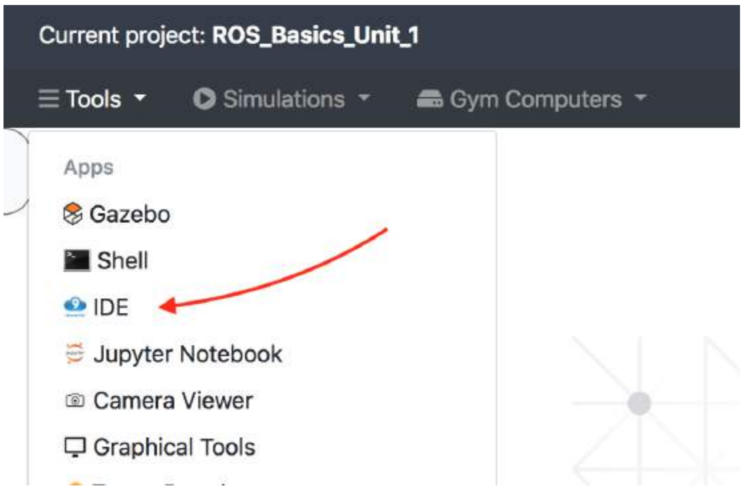


Launch Files list

In order to see all the files related to the chapter, follow the next steps:

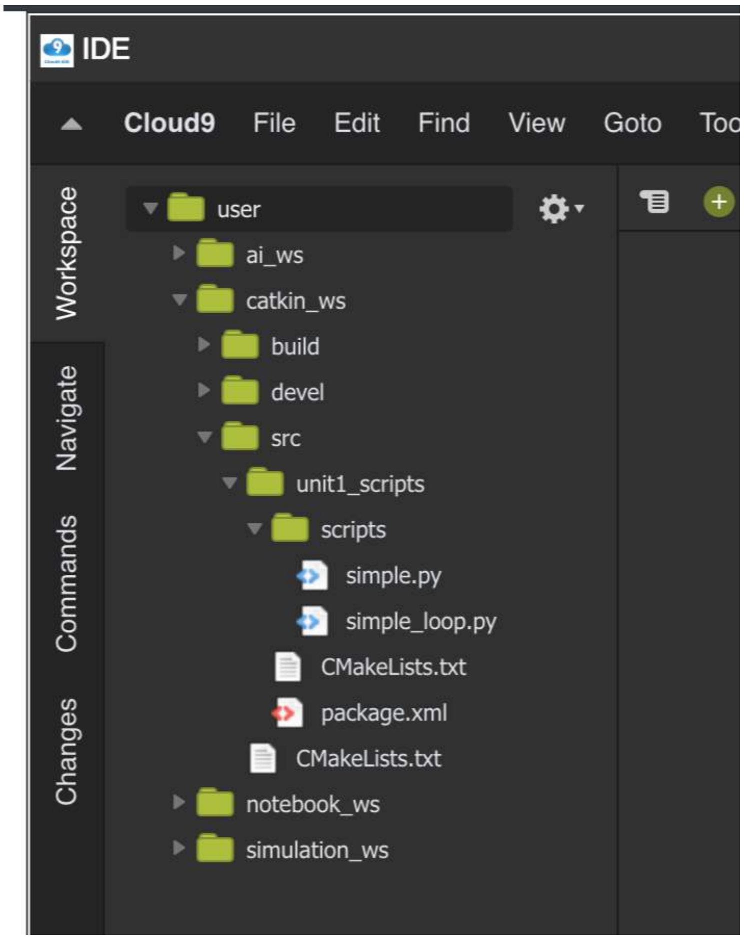
Step 1

In the **Tools** menu, click on the **IDE** option. An IDE will appear in your workspace.



Tools menu

2- You will find all the files inside the **catkin_ws** workspace. There, you have a ROS package containing all the files related to the chapter. This package will always follow the following name convention: **<unitX>_scripts**



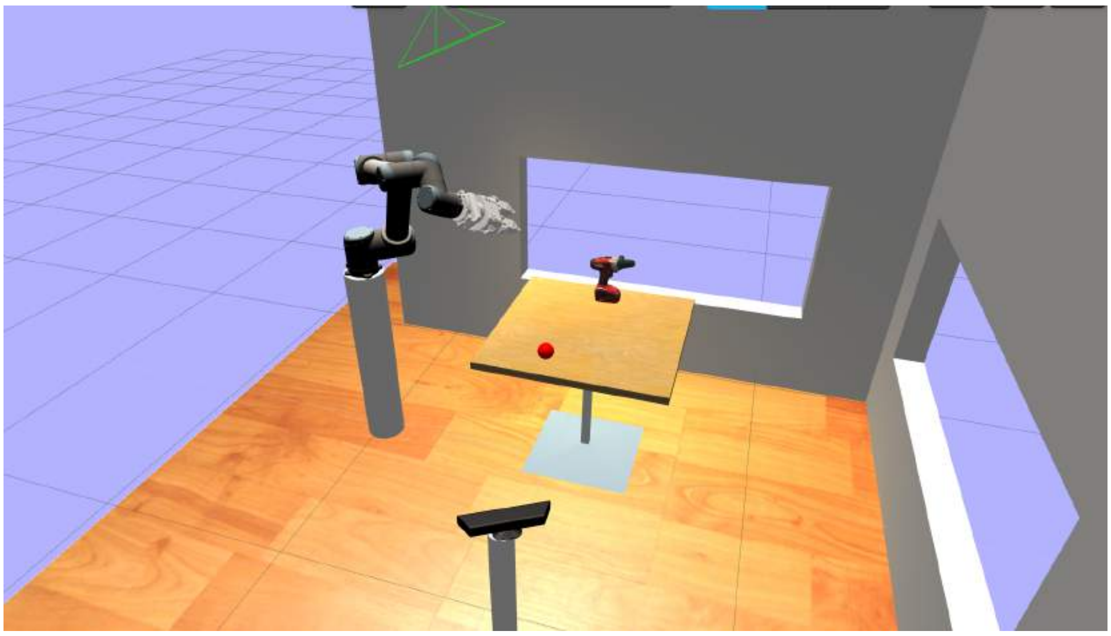
IDE workspace tree

And that’s it! Now just enjoy ROSDS and push your ROS learning.

Also, you can find more information and videotutorials about ROSDS here: https://www.youtube.com/watch?v=ELfRmuqgxns&list=PLK0b4e05LnzYGvX6EJN1gOQEl6aa3uyKS
If you have any doubt regarding ROSDS, don’t hesitate in contacting us to: **info@theconstructsim.com**

Unit 0. Introduction to the Course

ROS Manipulation in 5 Days



Manipulation Course Image

Unit 0: Introduction to the Course

- ROSject Link: https://bit.ly/2WHYHIv
- Package Name: **shadow_gazebo**
- Launch File: **main.launch**

SUMMARY
Estimated time of completion: 10 min This Unit is an introduction to the ROS Manipulation Course. You'll have a quick preview of the contents that you are going to cover during the Course, and you will also view a practical demo.

What is ROS Manipulation?

ROS Manipulation is the term used to refer to any robot that manipulates something in its environment. And what does this mean? Well, it means that it physically alters something in the world, for instance, changing it from its initial position.

The main goal of this course is to teach you the basic tools you need to know in order to be able to understand how ROS Manipulation works, and for you to learn how to implement it for any manipulator robot.

Do you want to have a taste?

With the proper introductions made, it is time to actually start. And... as we always do in the Robot Ignite Academy, let's start with practice! In the following demo, you will be testing apredefined script in a simulation, so you can have a practical look at what you are going to learn by completing this course. So... let's go!

Demo 1.1

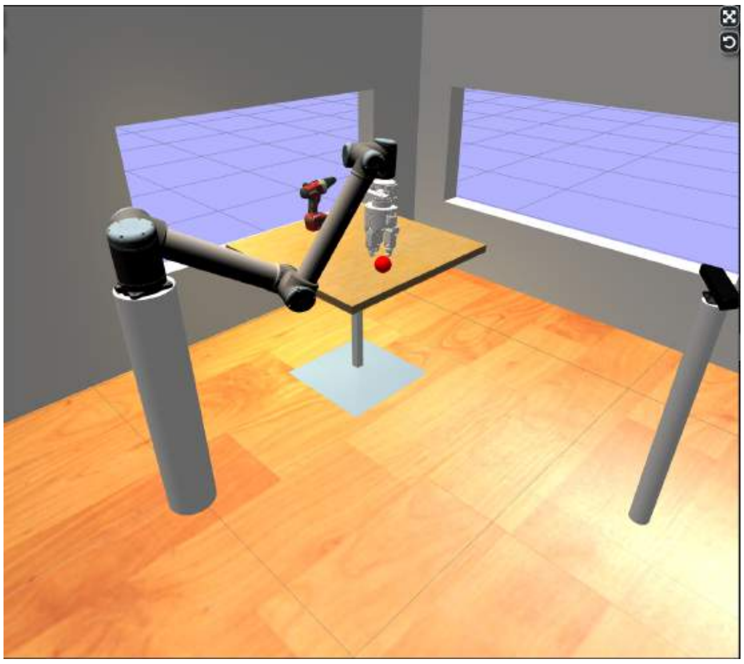
a) Execute the following piece of code in order to start the control system.

Note: You can execute a piece of Python code like this by clicking on it and then clicking on the play button on the top right-hand corner of the IPython notebook.

You can also press [CTRL]+[Enter] to execute it.

```
[ ]: from smart_grasping_sandbox.smart_grasper import SmartGrasper
     from tf.transformations import quaternion_from_euler
     from math import pi
     import time

     sgs = SmartGrasper()
```

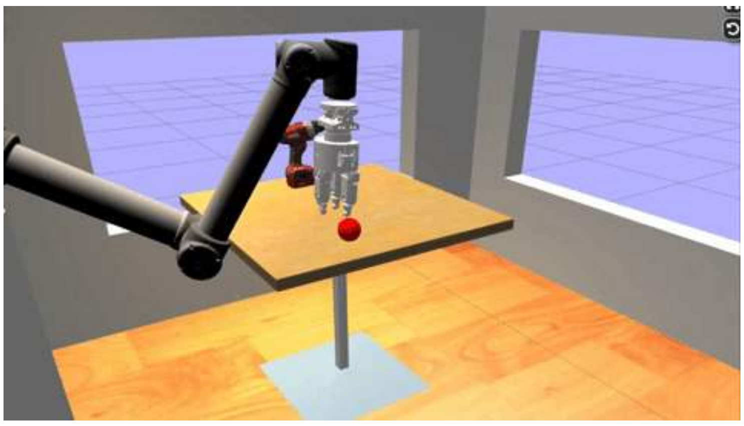
After executing the previous code and waiting for a few seconds, the robot should look like this:


Robot Arm Initialized

b) When the controllers are loaded, you can execute the following line of code in order to start the demo.

```
[ ]: sgs.pick()
```

Expected Result for Demo 1.1


Grasping Process

IMPORTANT NOTE
Once you are done with the demo, execute the following line of code in order to reset the world.

```
[ ]: sgs.reset_world()
```

IMPORTANT NOTE
Cool, right? In the previous Demo, what you've done is combine Motion Planning with Grasping and Perception, in order to be able to pick up the red ball that is lying on the table. By completing this course, you will learn how to do this, and how you can apply it to any Manipulator robot.

What will you learn with this Course?

Basically, during this course, you will address the following topics:

- Basics of ROS Manipulation.
- How to create and configure a MoveIt! package for a Manipulator robot.
- How to perform Motion Planning.
- How to perform Grasping.

Why you should learn it?

There are many reasons why you should learn to perform manipulation. Manipulation and manipulator robots are already being used in many environments, providing a solution to tasks that are really hard to develop for us humans, for many reasons. For instance:

Dangerous workspaces

This refers to environments that are dangerous for humans to be in. For instance:

- Space exploration
- Foundries
- Underwater Environments
- Factories

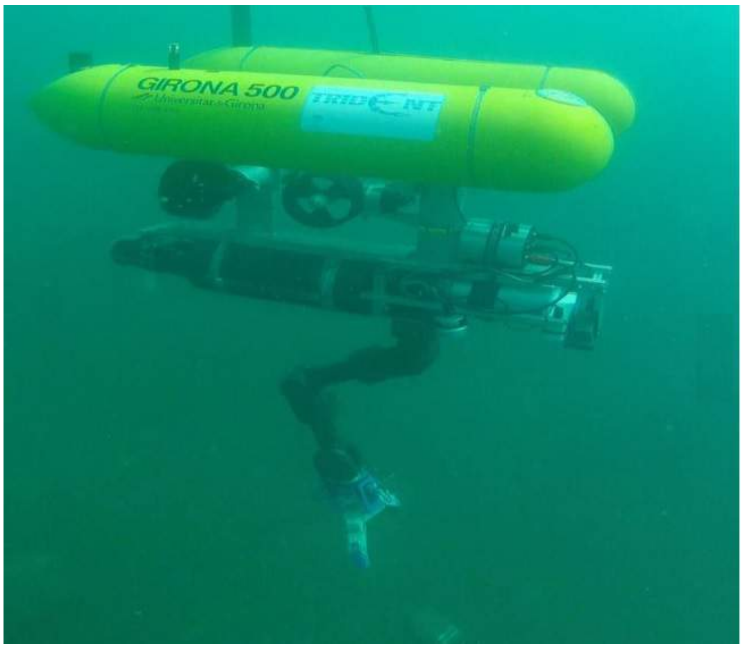


Image taken from http://srv.uib.es/trident/

Repetitive or unpleasant work

Basically, this refers to industrial environments, where robots have to do repetitive tasks for prolonged periods of time.

Image from http://www.mhi.org

Human intractable workspaces

This refers to workspaces that are very hard to be managed by humans. For instance:

- Too small workspaces
- Too big workspaces
- Workspaces where too much precision is needed

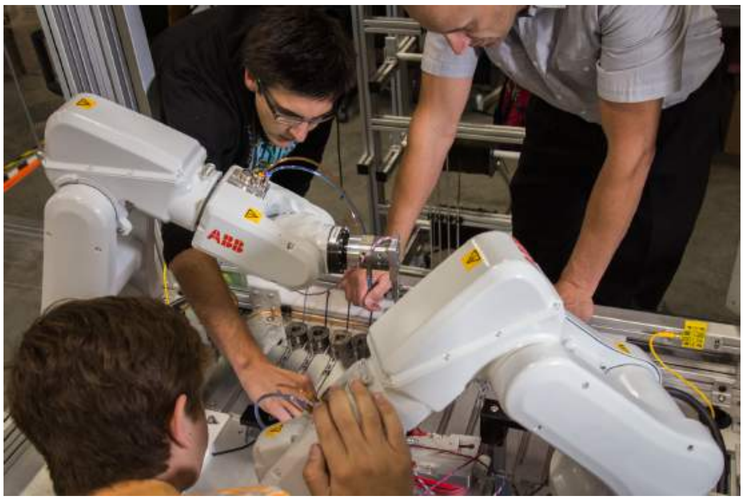


Image from http://www.controldesign.com

But, there are also many other fields that are beginning to use Manipulator robots.

Surgery

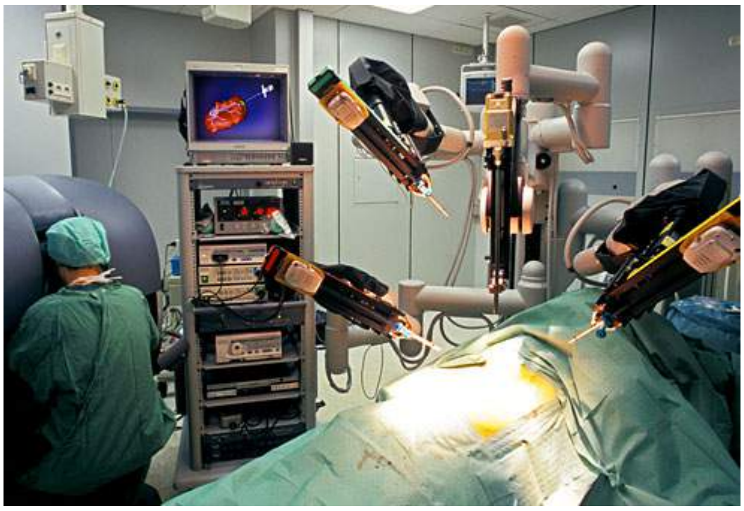


Image from www.avensonline.org

Patient care

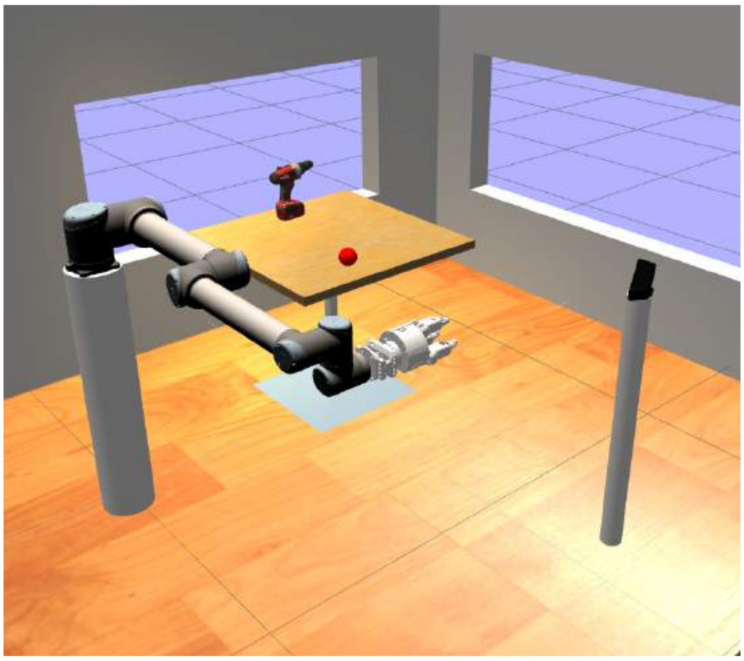
Image from http://www.huffingtonpost.com

Delivery Tasks

Anyway, in the near future, when current limitations are overcome, manipulation could be used in many other situations and environments. And you can be the first one to develop such technology! Why not?

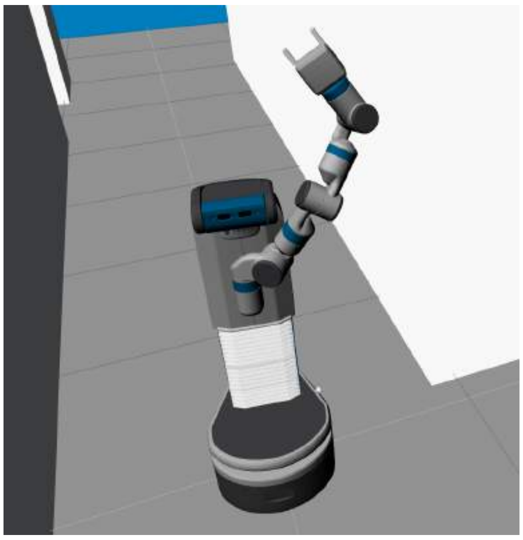
How will you learn it?

You will learn through hands-on experience from day one! This means that you will be working and testing your knowledge with various simulations. In the first section of the course, you'll work a Shadow robot simulation.



Shadow Robot

In the third and fourth chapters, you'll work with a Fetch robot.



Fetch Robot

And you will also have a Final Project based on a RB-1 simulation, like this one:

RB-1 Robot

Minimum requirements for the Course

In order to be able to fully understand the contents of this course, it is highly recommended to have the following knowledge:

- Basic ROS. You can get this knowledge by following the ROS in 5 Days Course.
- Basic Unix shell knowledge.
- Basic Joints knowledge. You can get this knowledge by following the TF ROS 101 Course.
- Basic URDF knowledge. You can get this knowledge by following the URDF 101 Course.
- Python Knowledge.

Special Thanks

- The Shadow Robot Company (especially Ugo Cupcic).

Shadow Robot Company

- Fetch Robotics

Fetch Robotics

- Robotnik

Robotnik

- Dave Coleman, creator and maintainer of the moveit_simples_grasps package.

- And, of course, we have to thank the whole ROS community for all the work done related to Robotics Manipulation.

ROS.org

Website: http://www.ros.org

Unit 1. Basic Concepts

ROS Manipulation in 5 Days

Unit 1: Basic Concepts

Run on ROSDS

- ROSject Link: http://bit.ly/2n7hnRq
- Package Name: **shadow_gazebo**
- Launch File: **main.launch**

SUMMARY
Estimated time of completion: 1h This unit is an introduction to some basic concepts regarding ROS Manipulation that you need to know in order to be able to fully understand the contents of the course.

Let's imagine the next scenario. You're chilling in your lab, listening to your favourite 80s music remix while carrying out a detailed investigation into the latest kitten videos that have been uploaded to Youtube... when suddenly, your boss comes into the room and says the magic words: ¡Hey (Insert Your Name Here)! I have a new project for you. I need you to make this robot grasp objects with ROS, and I need it yesterday. You'll probably start to sweat, while asking yourself questions like: Grasping? Yesterday? With ROS? What the hell do I need? Where do I start? How does BB-8 climb stairs? Don't panic! Keep calm... Robot Ignite Academy to the rescue!!

What do you need to perform ROS Manipulation?

Basically, you'll need to go through the following 4 main topics:

- MoveIt
- Motion Planning
- Perception
- Grasping

MoveIt

So, as we've pointed out before, the first thing you'll need to know about is MoveIt.

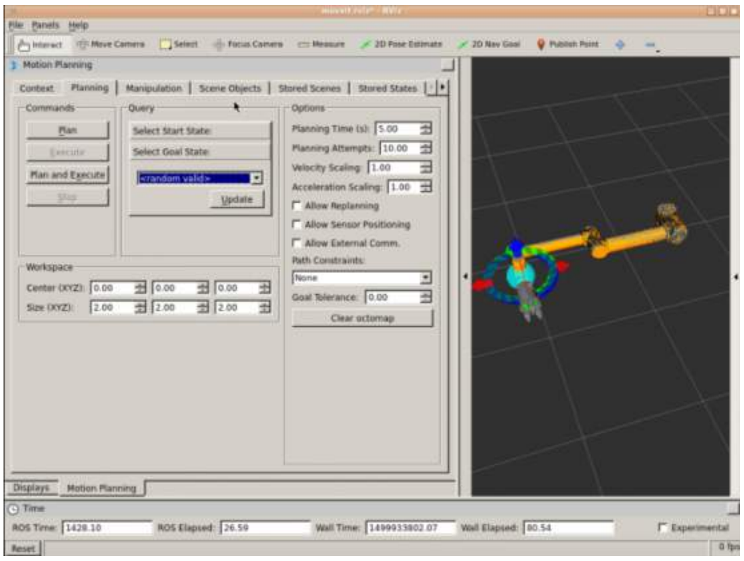
MoveIt is a set of packages and tools that allow you to perform manipulation with ROS. MoveIt provides software and tools in order to do Motion Planning, Manipulation, Perception, Kinematics, Collision Checking, Control, and Navigation. Yes! It is a huge and very useful tool, and it is basic knowledge if you want to learn about ROS Manipulation. You can learn more about it by checking all of its documentation on the official website: http://moveit.ros.org
In the following, you're going to have a first look at this tool. So, let's stop the talking, and get to the action!

Demo 1.1
a) Launch the MoveIt RViz environment. To do that, you can just execute the following command:

Execute in WebShell #1

```
[ ]: roslaunch smart_grasp_mod_moveit_config demo.launch
```

Now, if you open the Graphic Tools by hitting this icon



<random valid>

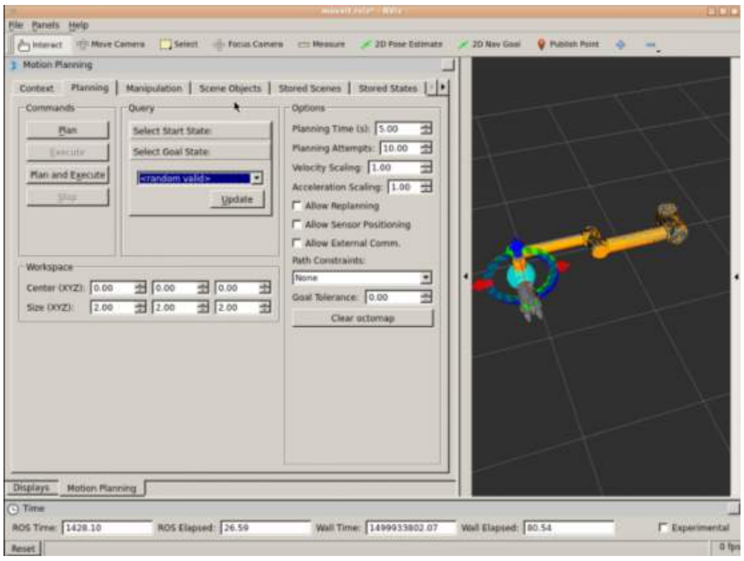
Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Allow Replanning
Allow Sensor Positioning
Allow External Comm.
Path Constraints: None
Goal Tolerance: 0.00

Workspace
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00

ROS Time: 1428.10 ROS Elapsed: 26.59 Wall Time: 1499933802.07 Wall Elapsed: 80.54 Experimental

MoveIt RViz

you will see something like this:



Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Allow Replanning
Allow Sensor Positioning
Allow External Comm.
Path Constraints: None
Goal Tolerance: 0.00
<random valid>
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00
ROS Time: 1428.10 ROS Elapsed: 26.59 Wall Time: 1499933802.07 Wall Elapsed: 80.54

MoveIt RViz environment

b) Move around the different options that the toll provides, and try to see if you can figure anything out!

IMPORTANT NOTE
Once you are done with this exercise, kill the MoveIt RViz environment by hitting Ctrl + C in the webshell where you executed the command.

IMPORTANT NOTE
Awesome, right? You are probably a little bit confused right now, since MoveIt is a huge tool that provides lots of options. But, don't get stressed! In the next unit (Chapter 2), you will learn how to deal with MoveIt and you'll see how this tool can help you to perform Manipulation in ROS.

Motion Planning

Great! So.... what else do you need in order to learn ROS Manipulation? Well, you'll need to know how to perform Motion Planning! And... what does Motion Planning mean? Well, it bascially means to plan a movement (motion) from point A to point B, without colliding with anything.

In other words, you will need to be able to control the different joints and links of your robot, avoiding collisions between them or with other elements in the environment.

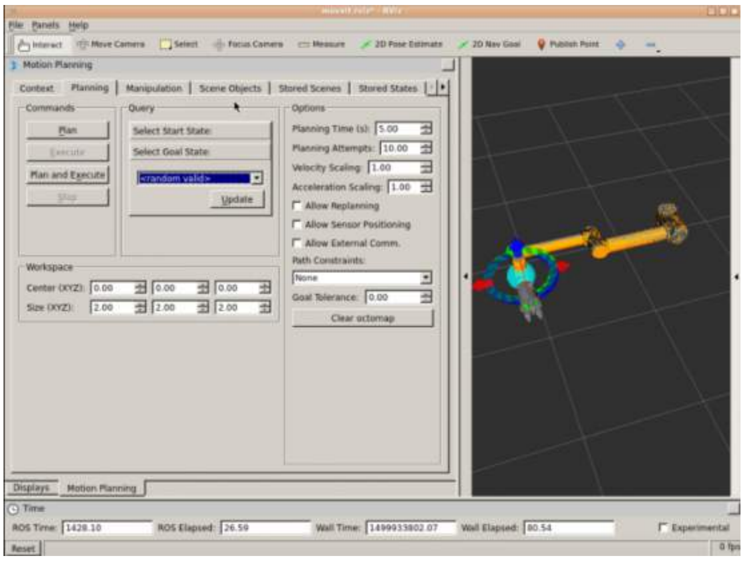
Let's see a better practical example of what this is about! Don't you agree?

Demo 1.2
a) Launch the MoveIt Rviz environment again. To do that, you can just execute the following commands:

Execute in WebShell #1

```
[ ]: roslaunch smart_grasp_mod_moveit_config
     smart_grasping_planning_execution.launch
```

If everything goes well, you should see something like this:



Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Goal Tolerance: 0.00
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00
ROS Time: 1428.10
ROS Elapsed: 26.59
Wall Time: 1499933802.07
Wall Elapsed: 80.54

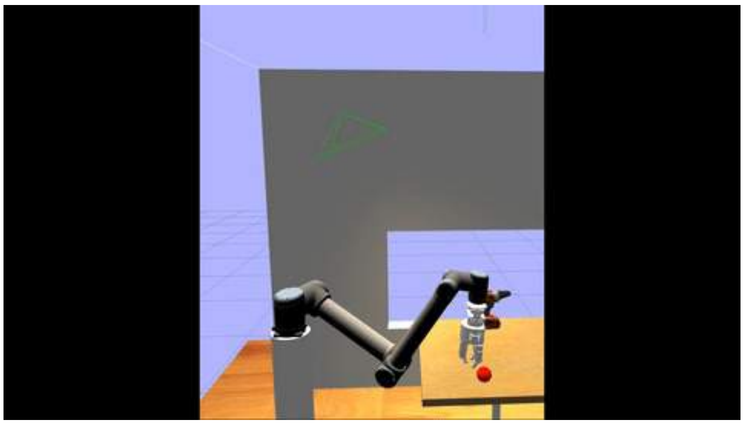
MoveIt RViz environment

b) Now, execute the following piece of code in order to control the joints of the robot.

```
[ ]: #! /usr/bin/env python

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

     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")
     display_trajectory_publisher =
     rospy.Publisher('/move_group/display_planned_path',
     moveit_msgs.msg.DisplayTrajectory, queue_size=1)
```

```
group_variable_values = group.get_current_joint_values()

group_variable_values[1] = -1.5
group.set_joint_value_target(group_variable_values)

plan2 = group.plan()

rospy.sleep(5)
group.go(wait=True)
rospy.sleep(5)

group_variable_values[2] = 1.5
group.set_joint_value_target(group_variable_values)

plan2 = group.plan()

rospy.sleep(5)
group.go(wait=True)
rospy.sleep(5)

moveit_commander.roscpp_shutdown()
```

You should see something like this (a little bit slower):



Motion Demo

IMPORTANT NOTE
Once you are done with this exercise, kill the MoveIt RViz environment by hitting Ctrl + C in the webshell where you executed the command.

IMPORTANT NOTE
Interesting, right? But, how does this work? Well. . . be patient! You'll learn how to perform Motion Planning in the third unit (Chapter 3).

Perception

You'll also need to know about Perception, of course! In order to be able to interact with any object in the environment, you first need to be able to visualize it. You need to know where it is, how it is, etc. . . and that's what Perception is for!

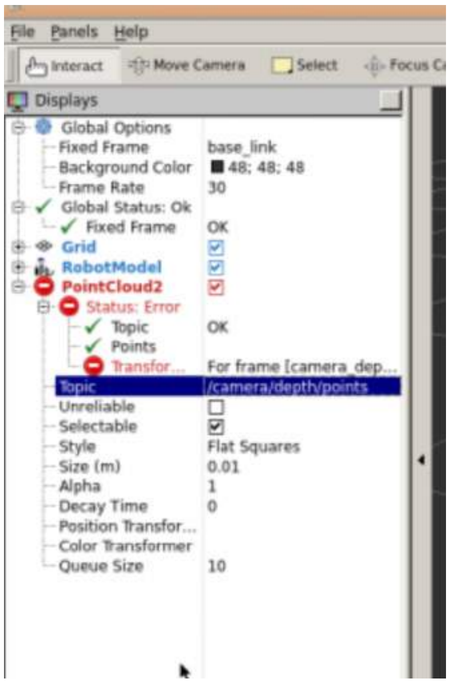
Perception is usually done using RGBD cameras, like a Kinect. In the following Demo, you'll have a look at the data that the Kinect camera placed in the simulation is capturing.

Demo 1.3
a) Launch RViz in order to visualize the RGBD data captured with the Kinect camera placed in the simulation.

Execute in WebShell #1

```
[ ]: rosrun rviz rviz
```

b) Add to RViz a PointCloud2 element in order to visualize the PointCloud captured by the Kinect Camera. After a few seconds, you should get an error, like this one:



```
Global Options
  Fixed Frame            base_link
  Background Color       ■ 48; 48; 48
  Frame Rate             30
Global Status: Ok
  Fixed Frame            OK
Grid                     ✓
RobotModel               ✓
PointCloud2              ✓
  Status: Error
    Topic                OK
    Points
    Transfor...          For frame [camera_dep...
  Topic                  /camera/depth/points
  Unreliable             ☐
  Selectable             ✓
  Style                  Flat Squares
  Size (m)               0.01
  Alpha                  1
  Decay Time             0
  Position Transfor...
  Color Transformer
  Queue Size             10
```

RViz Error

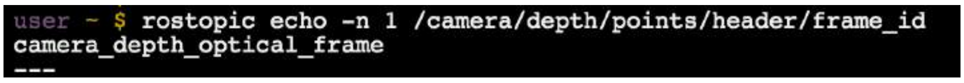
The PointCloud2 element in RViz allows you to visualize PointCloud data captured by RGBD cameras.

Note: Keep in mind that PointClouds consume lots of resources, since they carry lots of data. So, this process may slow your computer.

c) Execute the following command in order to get the frame related to the RGBD camera.

   Execute in WebShell #1

```
[ ]: rostopic echo -n1 /camera/depth/points/header/frame_id
```

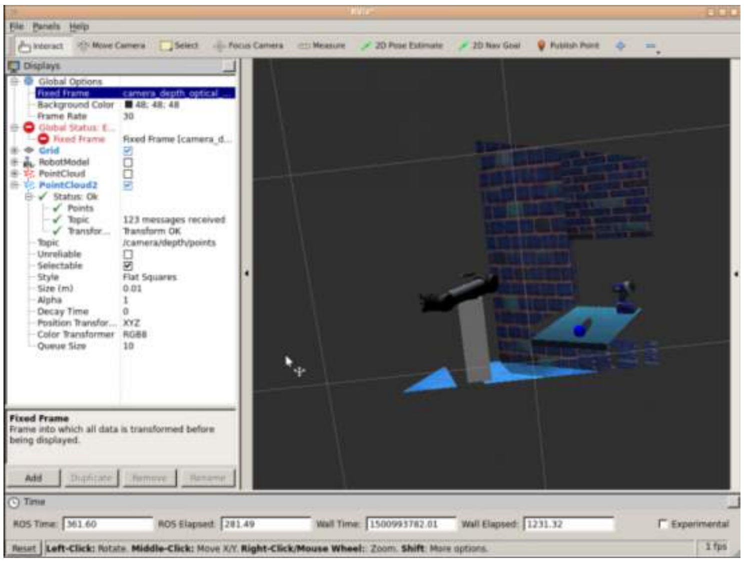


```
user ~ $ rostopic echo -n 1 /camera/depth/points/header/frame_id
camera_depth_optical_frame
---
```

Depth Frame

d) Modify the Fixed Frame in RViz for the frame that was obtained with the previous command. It won't appear in the available frames, so you will have to write it manually.

If everything goes okay, you should end up with something like this:



```
Global Options
  Fixed Frame          camera_depth_optical_...
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: E...
  Fixed Frame          Fixed Frame [camera_d...
Grid                   ✓
RobotModel             ☐
PointCloud             ☐
PointCloud2            ✓
  Status: Ok
    Points
    Topic              123 messages received
    Transfor...        Transform OK
  Topic                /camera/depth/points
  Unreliable           ☐
  Selectable           ✓
  Style                Flat Squares
  Size (m)             0.01
  Alpha                1
  Decay Time           0
  Position Transfor... XYZ
  Color Transformer    RGB8
  Queue Size           10

Fixed Frame
Frame into which all data is transformed before being displayed.

ROS Time: 361.60   ROS Elapsed: 281.49   Wall Time: 1500993782.01   Wall Elapsed: 1231.32
```

Depth Cloud in RViz

IMPORTANT NOTE
Once you are done with this exercise, kill the RViz environment by hitting Ctrl + C in the webshell where you executed the command.

IMPORTANT NOTE
So, you have now visualized data that the Kinect camera in the simulation is capturing right now. Obviously, we can use this data in order to improve manipulation tasks, such as detecting new objects that spawn into the simulation. Anyway, you will get deeper into Perception in Chapter 4, so just be patient!

Grasping

Finally, you'll need to know about Grasping. And what is Grasping? Well, the word Grasping refers to the action of catching an object from the environment in order to do an action with it; for instance, change its position. Inside the Grasping process, there are other variables that take place, such as the Perception of the environment. In the previous chapter, Introduction to the Course, you already saw a Grasping demo, so you are not going to repeat it this time.

Even though Grasping may look like a very easy and simple task, it is not. Not at all! There are lots of variables that need to be taken into account, and there are lots of things that can go wrong!

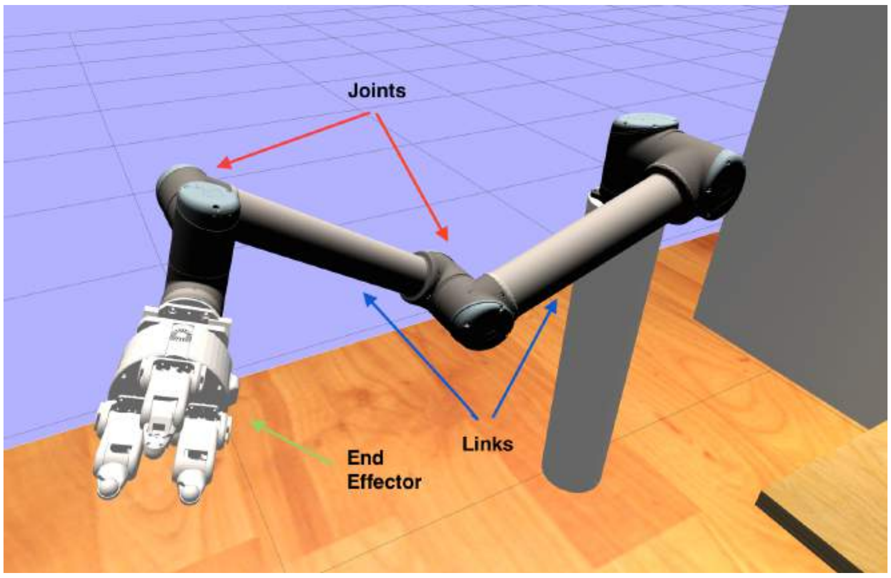
You will learn the basics about Grasping in Chapter 5.

And of course... you'll need a Manipulator robot!

And what is a Manipulator robot? Well, we know that a Maniplator Robot is any kind of robot that is able to physically alter the environment it works in.

Structure of a Manipulator Robot A Manipulator robot is modeled as a chain of rigid links, which are connected by joints, and which end in what is known as the end-effector of the robot. So, basically, any Manipulator is composed of these 3 elemental parts:

- Links: Rigid pieces that connect the joints of the manipulator.
- Joints: Connectors between the links of the manipulator, and which provide either translational or rotational movement.
- End Effector
- Grippers / Tools
- Sensors



Manipulator Robot

Exercise 1.1
Have a look at the Manipulator robot that you are working with in this unit, and identify the names of all the links, joints, and end effectors that are part of it.

Basic Terminology

Finally, it would be good to be familiar with some basic terminology that is often used in Robotic Manipulation.

DoFs for Manipulation

DoFs is the word used to refer to the Degrees of Freedom of a robot. And what does this mean? Well, it basically means the number of ways in which a robotic arm can move. A system has n DoFs if exactly n parameters are required to completely specify the configuration.

- In the case of a manipulator robot:

- The configuration is determined by the number of joints that the robot has.
- So, the number of joints determines the number of DoFs of the manipulator.
- A rigid object in a 3D space has 6 parameters
- 3 for positioning (x, y, z) and 3 for the orientation (roll, pitch, and yaw angles)

If a manipulator has less than 6 DoFs, the arm cannot reach every point in the workspace with arbitrary orientation
If a manipulator has more than 6 DoFs, the robot is kinematically redundant. Also, the more DoFs a manipulator has, the harder it will be to control it.

Exercise 1.2
Determine the number of DoFs that the simulated robot has.

Grippers

A gripper is a device that enables the holding of an object to be manipulated. The easier way to describe a gripper is to think of the human hand. Just like a hand, a gripper enables holding, tightening, handling, and releasing of an object. A gripper can be attached to a robot or it can be part of a fixed automation system. Many styles and sizes of grippers exist so that the correct model can be selected for the application.

Image from https://academy.autodesk.com

Well, that's all for this first unit! I hope it has helped you to better structure your ideas regarding Manipulation. Anyway, the good stuff begins now. So. . . let's go!

Chapter 2. Motion Planning using Graphical Interfaces

ROS Manipulation in 5 Days

Unit 2: Motion Planning using Graphical Interfaces

Run on ROSDS

- ROSject Link: http://bit.ly/2n6ggkV
- Package Name: **shadow_gazebo**
- Launch File: **main.launch**

SUMMARY
Estimated time of completion: 2h This unit will show you how to create a Moveit Package for your industrial robot. By completing this unit, you will be able to create a package that allows your robot to perform motion planning.

What is MoveIt?

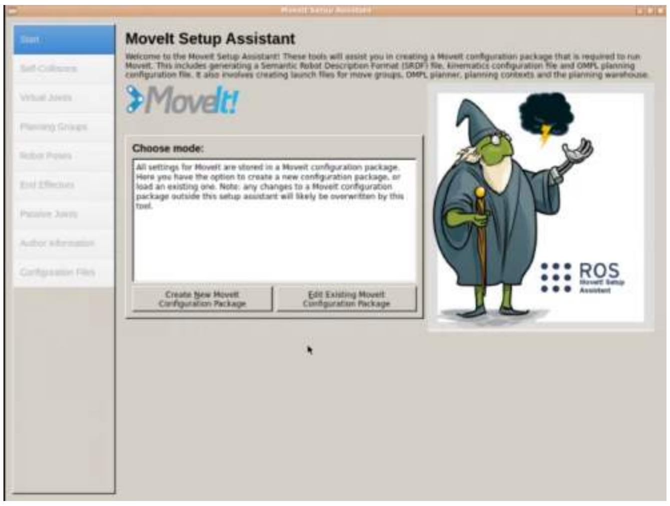
MoveIt is a ROS framework that allows you to perform motion planning with an specific robot. And... what does this mean? Well, it bascially means that it allows you to plan a movement(motion) from a point A to a point B, without colliding with anything.

MoveIt is a very complex and useful tool. So, within this MicroCourse, we are not going to dive into the details of how MoveIt works, or all the features it provides. If you are intereseted in learning more about MoveIt, you can have a look ath the official website here: http://moveit.ros.org/ Fortunately, MoveIt provides a very nice and easy-to-use GUI, which will help us to be able to interact with the robot in order to perform motion planning. However, before being able to actually use MoveIt, we need to build a package. This package will generate all the configuration and launch files required for using our defined robot (the one that is defined in the URDF file) with MoveIt. In order to generate this package, just follow all the steps described in the following exercise!

Generating MoveIt! configuration package using Setup Assistant tool

Exercise 2.1
a) First of all, you'll need to launch the MoveIt Setup Assistant. You can do that by typing the following command:

Execute in WebShell #1
```
roslaunch moveit_setup_assistant setup_assistant.launch
```
Now, if you open the Graphic Tools by hitting this icon you will see something like this:



MoveIt Setup Assistant

Welcome to the MoveIt Setup Assistant! These tools will assist you in creating a MoveIt configuration package that is required to run MoveIt. This includes generating a Semantic Robot Description Format (SRDF) file, kinematics configuration file and OMPL planning configuration file. It also involves creating launch files for move groups, OMPL planner, planning contexts and the planning warehouse.

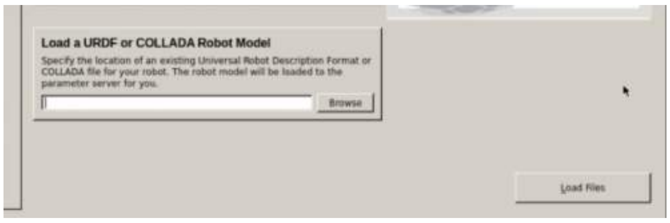
Choose mode:

All settings for MoveIt are stored in a MoveIt configuration package. Here you have the option to create a new configuration package, or load an existing one. Note: any changes to a MoveIt configuration package outside this setup assistant will likely be overwritten by this tool.

MoveIt Setup Assistant

Great! You now are at the MoveIt Setup Assistant. The next thing you'll need to is to load your robot file. So let's continue!

b) Click on the "Create New MoveIt Configuration Package" button. A new section like this will appear:

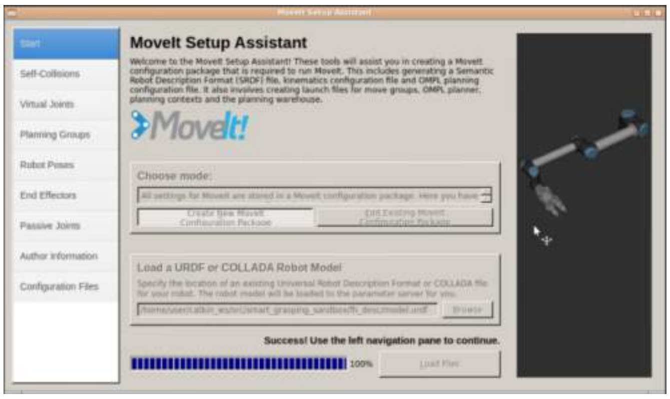


Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot. The robot model will be loaded to the parameter server for you.

Create New Package

Now, just click the "Browse" button, select the URDF file named **model.urdf** located in the **shadow_tc** package, and click on the "Load Files" button. You will probably need to copy this file into your workspace. You should now see something like this:



MoveIt Setup Assistant

Welcome to the MoveIt Setup Assistant! These tools will assist you in creating a MoveIt configuration package that is required to run MoveIt. This includes generating a Semantic Robot Description Format (SRDF) file, kinematics configuration file and OMPL planning configuration file. It also involves creating launch files for move groups, OMPL planner, planning contexts and the planning warehouse.

Choose mode:

All settings for MoveIt are stored in a MoveIt configuration package. Here you have

Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot. The robot model will be loaded to the parameter server for you.

/home/user/catkin_ws/src/smart_grasping_sandbox/fh_desc/model.urdf

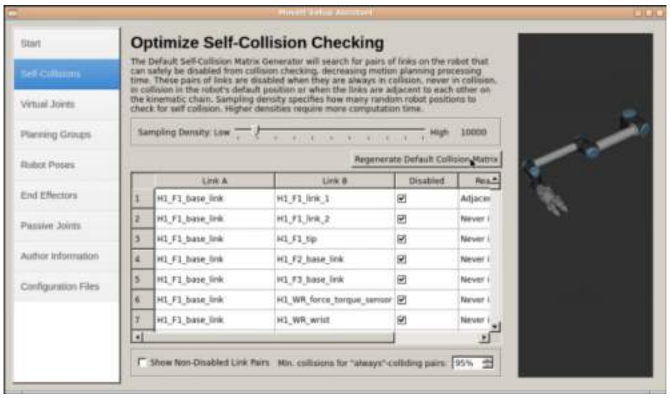
Success! Use the left navigation pane to continue.

100%

Load URDF

Great! So now, you've loaded the xacro file of your robot to the MoveIt Setup Assistant. Now, let's start configuring some things.

c) Go to the "Self-Collisions" tab, and click on the "Regenerate Default Collision Matrix" button. You will end with something like this:



Optimize Self-Collision Checking

The Default Self-Collision Matrix Generator will search for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time. These pairs of links are disabled when they are always in collision, never in collision, in collision in the robot's default position or when the links are adjacent to each other on the kinematic chain. Sampling density specifies how many random robot positions to check for self collision. Higher densities require more computation time.

Sampling Density: Low — High 10000

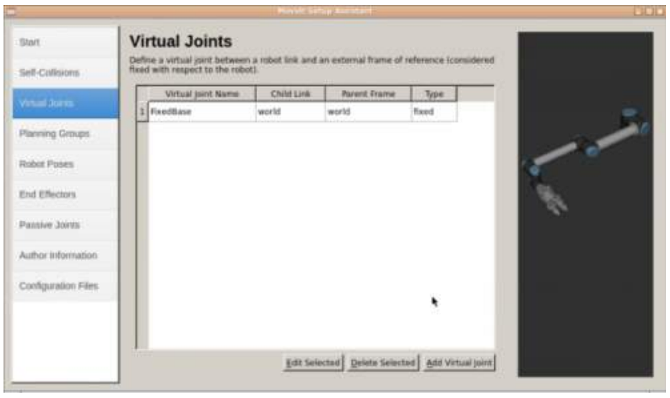
	Link A	Link B	Disabled	Rea
1	H1_F1_base_link	H1_F1_link_1	✓	Adjacen
2	H1_F1_base_link	H1_F1_link_2	✓	Never i
3	H1_F1_base_link	H1_F1_tip	✓	Never i
4	H1_F1_base_link	H1_F2_base_link	✓	Never i
5	H1_F1_base_link	H1_F3_base_link	✓	Never i
6	H1_F1_base_link	H1_WR_force_torque_sensor	✓	Never i
7	H1_F1_base_link	H1_WR_wrist	✓	Never i

☐ Show Non-Disabled Link Pairs Min. collisions for "always"-colliding pairs: 95%

Self-Collisions

Here, you are just defining some pairs of links that don't need to be considered when performing collision checking. For instance, because they are adjacent links, so they will always be in collision.

d) Next, move to the "Virtual Joints" tab. Here, you will define a virtual joint for the base of the robot. Click the "Add Virtual Joint" button, and set the name of this joint to FixedBase, and the parent to world. Just like this:



Virtual Joints

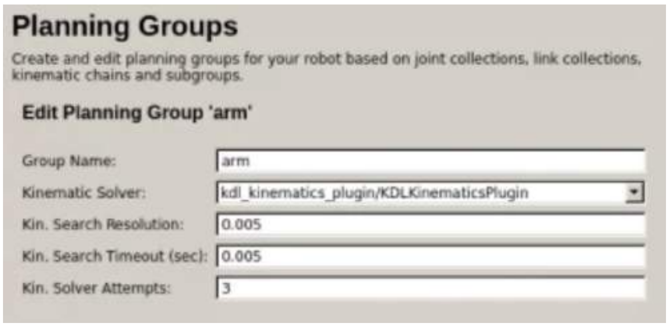
Define a virtual joint between a robot link and an external frame of reference (considered fixed with respect to the robot).

	Virtual Joint Name	Child Link	Parent Frame	Type
1	FixedBase	world	world	fixed

Virtual Joints

Finally, click the “Save” button. Basically, what you are doing here is to create an “imaginary” joint that will connect the base of your robot with the simulated world.

e) Now, open the “Planning Groups” tab and click the “Add Group” button. Now, you will create a new group called arm, which will use the KDLKinematicsPlugin. Just like this:



Planning Groups

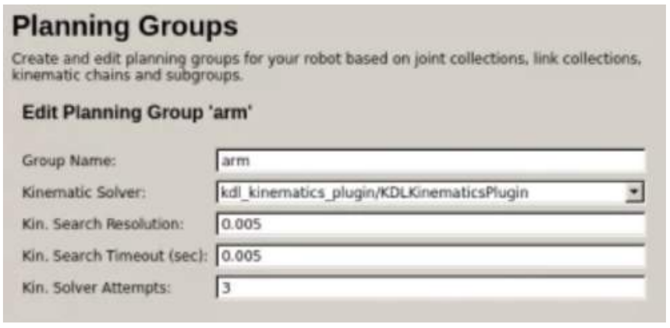
Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Edit Planning Group 'arm'

Group Name:	arm
Kinematic Solver:	kdl_kinematics_plugin/KDLKinematicsPlugin
Kin. Search Resolution:	0.005
Kin. Search Timeout (sec):	0.005
Kin. Solver Attempts:	3

Planning Group

Also, select one of the Default Planners for OMPL Planning. For instance, RRT or RRTConnect.



Planning Groups

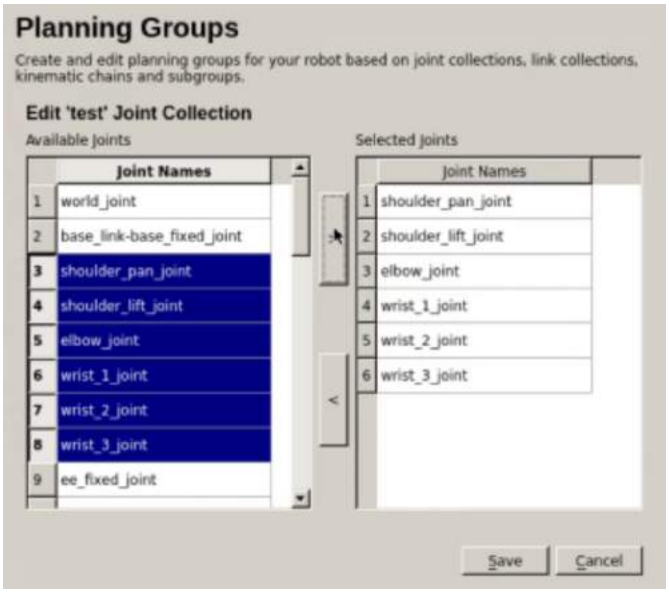
Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Edit Planning Group 'arm'

Group Name:	arm
Kinematic Solver:	kdl_kinematics_plugin/KDLKinematicsPlugin
Kin. Search Resolution:	0.005
Kin. Search Timeout (sec):	0.005
Kin. Solver Attempts:	3

Planning Group

Next, you will click on the "Add Joints" button, and you will select all the joints that form the arm of the robot, excluding the gripper. Just like this:



Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Edit 'test' Joint Collection

Available Joints

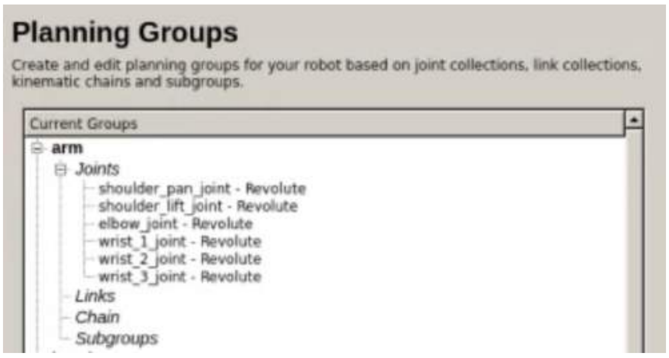
	Joint Names
1	world_joint
2	base_link-base_fixed_joint
3	shoulder_pan_joint
4	shoulder_lift_joint
5	elbow_joint
6	wrist_1_joint
7	wrist_2_joint
8	wrist_3_joint
9	ee_fixed_joint

Selected Joints

	Joint Names
1	shoulder_pan_joint
2	shoulder_lift_joint
3	elbow_joint
4	wrist_1_joint
5	wrist_2_joint
6	wrist_3_joint

Add Joints

Finally, click the "Save" button and you will end up with something like this:



Planning Groups

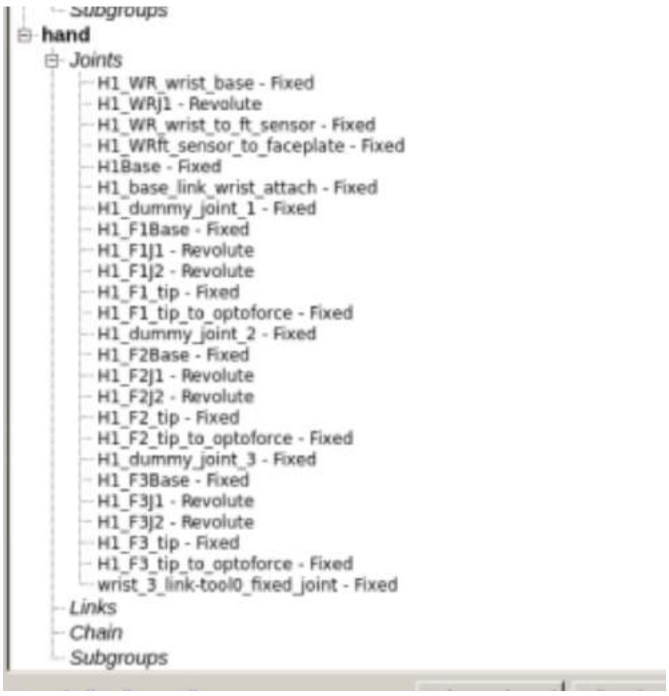
Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

```
Current Groups
⊟ arm
  ⊟ Joints
      shoulder_pan_joint - Revolute
      shoulder_lift_joint - Revolute
      elbow_joint - Revolute
      wrist_1_joint - Revolute
      wrist_2_joint - Revolute
      wrist_3_joint - Revolute
    Links
    Chain
    Subgroups
```

Planning Group 2

So now, you've defined a group of links for performing Motion Planning with, and you've defined the plugin you want to use to calculate those Plans.

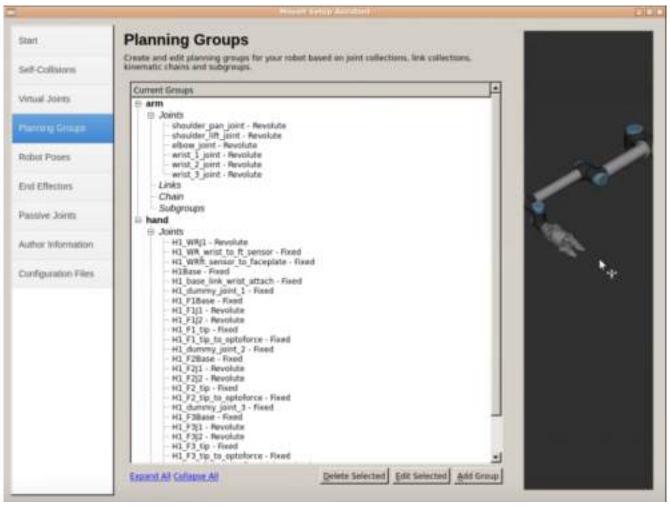
Now, repeat the same process, but this time for the gripper. In this case, you DO NOT have to define any Kinematics Solver. If you are not sure of what joints to add to the hand, you can have a look at the below image.



```
   Subgroups
⊟ hand
  ⊟ Joints
      H1_WR_wrist_base - Fixed
      H1_WRJ1 - Revolute
      H1_WR_wrist_to_ft_sensor - Fixed
      H1_WRft_sensor_to_faceplate - Fixed
      H1Base - Fixed
      H1_base_link_wrist_attach - Fixed
      H1_dummy_joint_1 - Fixed
      H1_F1Base - Fixed
      H1_F1J1 - Revolute
      H1_F1J2 - Revolute
      H1_F1_tip - Fixed
      H1_F1_tip_to_optoforce - Fixed
      H1_dummy_joint_2 - Fixed
      H1_F2Base - Fixed
      H1_F2J1 - Revolute
      H1_F2J2 - Revolute
      H1_F2_tip - Fixed
      H1_F2_tip_to_optoforce - Fixed
      H1_dummy_joint_3 - Fixed
      H1_F3Base - Fixed
      H1_F3J1 - Revolute
      H1_F3J2 - Revolute
      H1_F3_tip - Fixed
      H1_F3_tip_to_optoforce - Fixed
      wrist_3_link-tool0_fixed_joint - Fixed
    Links
    Chain
    Subgroups
```

Gripper Joints

At the end, you should end up with something similar to this:



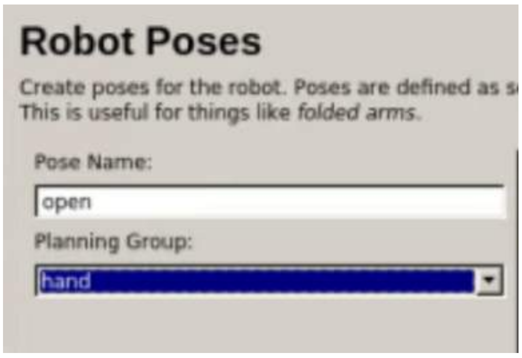
Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

```
Current Groups
⊟ arm
  ⊟ Joints
      shoulder_pan_joint - Revolute
      shoulder_lift_joint - Revolute
      elbow_joint - Revolute
      wrist_1_joint - Revolute
      wrist_2_joint - Revolute
      wrist_3_joint - Revolute
    Links
    Chain
    Subgroups
⊟ hand
  ⊟ Joints
      H1_WRJ1 - Revolute
      H1_WR_wrist_to_ft_sensor - Fixed
      H1_WRft_sensor_to_faceplate - Fixed
      H1Base - Fixed
      H1_base_link_wrist_attach - Fixed
      H1_dummy_joint_1 - Fixed
      H1_F1Base - Fixed
      H1_F1J1 - Revolute
      H1_F1J2 - Revolute
      H1_F1_tip - Fixed
      H1_F1_tip_to_optoforce - Fixed
      H1_dummy_joint_2 - Fixed
      H1_F2Base - Fixed
      H1_F2J1 - Revolute
      H1_F2J2 - Revolute
      H1_F2_tip - Fixed
      H1_F2_tip_to_optoforce - Fixed
      H1_dummy_joint_3 - Fixed
      H1_F3Base - Fixed
      H1_F3J1 - Revolute
      H1_F3J2 - Revolute
      H1_F3_tip - Fixed
      H1_F3_tip_to_optoforce - Fixed
```

Planning Groups 3

f) Now, you are going to create a couple of predefined poses for your robot. Go to the “Robot Poses” tab and click on the “Add Pose” button. In the left side of the screen, you will be able to define the name of the pose and the planning group it’s refered to. In this case, we will name the 1st Pose open, and it will be related, obviously, to the hand Group.



Robot Poses

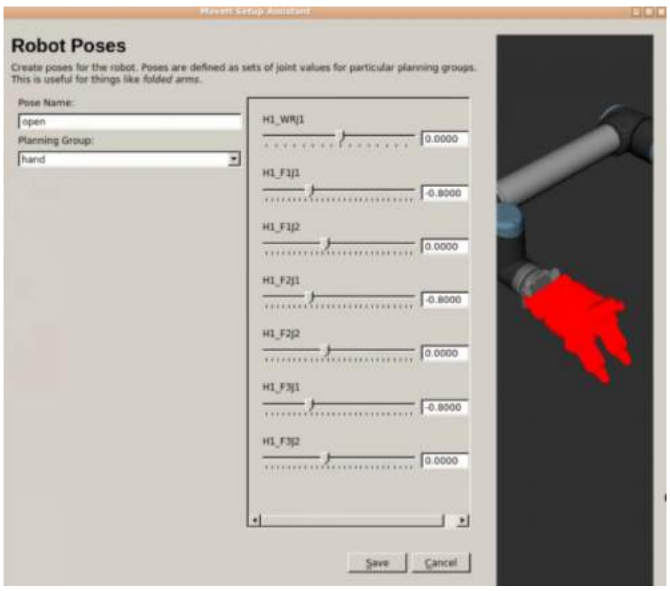
Create poses for the robot. Poses are defined as s
This is useful for things like *folded arms*.

Pose Name: open

Planning Group: hand

Hand Pose

Now, you will have to define the positions of the joints that will be related to this Pose. For this case, you can set them as in the image below:



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 *folded arms*.

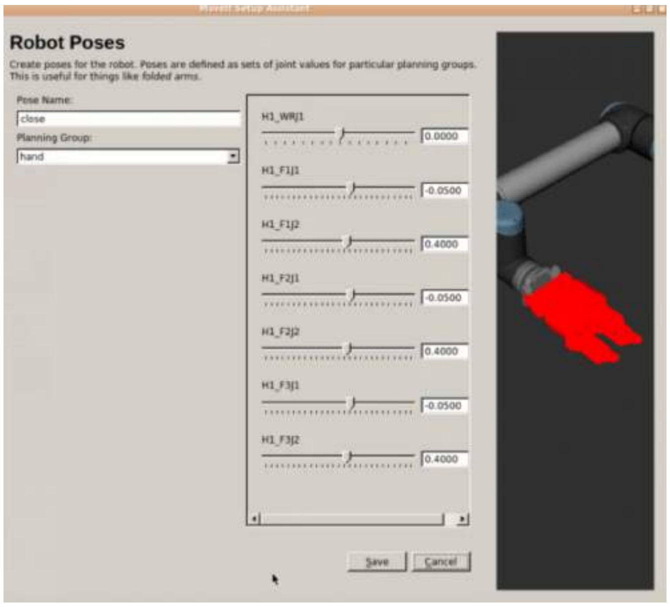
Pose Name: open

Planning Group: hand

H1_WRJ1	0.0000
H1_F1J1	-0.8000
H1_F1J2	0.0000
H1_F2J1	-0.8000
H1_F2J2	0.0000
H1_F3J1	-0.8000
H1_F3J2	0.0000

Hand Pose 1

Now, repeat the operation, but this time we will define the close Pose. For instance, it could be something like this:



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 *folded arms*.

Pose Name: close

Planning Group: hand

H1_WRJ1 0.0000

H1_F1J1 -0.0500

H1_F1J2 0.4000

H1_F2J1 -0.0500

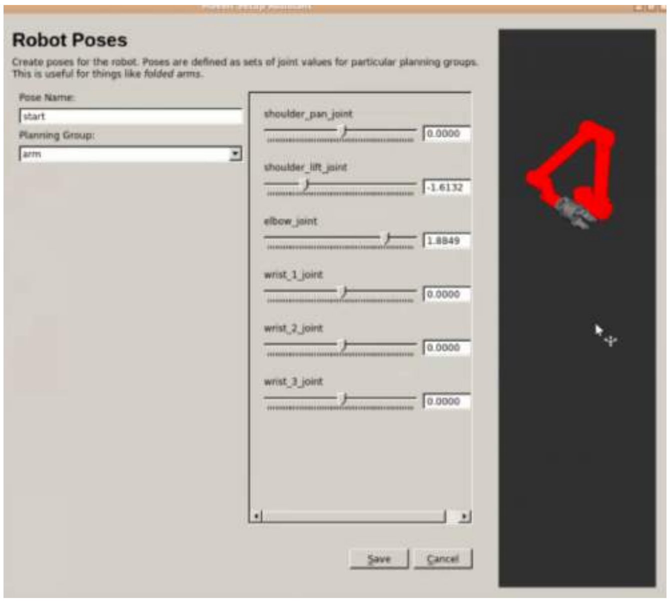
H1_F2J2 0.4000

H1_F3J1 -0.0500

H1_F3J2 0.4000

Hand Pose 2

Finally, let's create an start Pose for the arm Group. It could be something like this:



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 *folded arms*.

Pose Name: start

Planning Group: arm

shoulder_pan_joint 0.0000

shoulder_lift_joint -1.6132

elbow_joint 1.8849

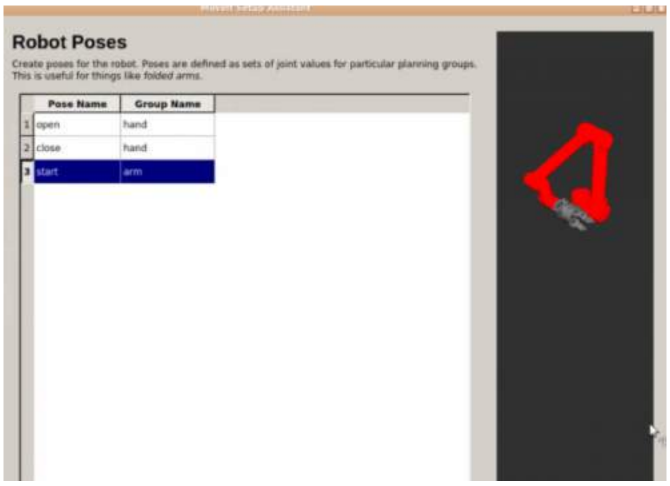
wrist_1_joint 0.0000

wrist_2_joint 0.0000

wrist_3_joint 0.0000

Arm Pose

At the end, you should have something similar to this:



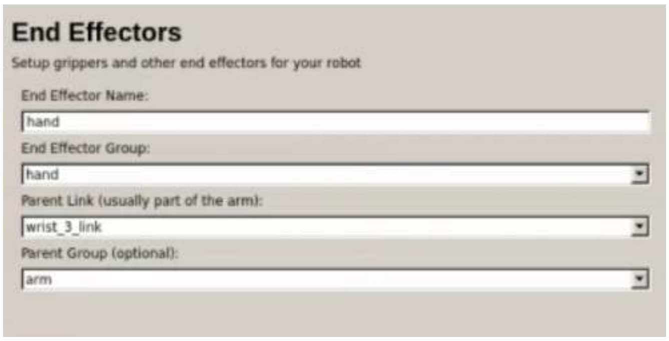
Robot Poses

Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like folded arms.

	Pose Name	Group Name
1	open	hand
2	close	hand
3	start	arm

Final Robot Poses

g) The next step will be to set up the End-Effector of the robot. For that, just go the End Effectors tab, and click on the "Add End Effector" button. We will name our End Effector hand.



End Effectors

Setup grippers and other end effectors for your robot

End Effector Name: hand

End Effector Group: hand

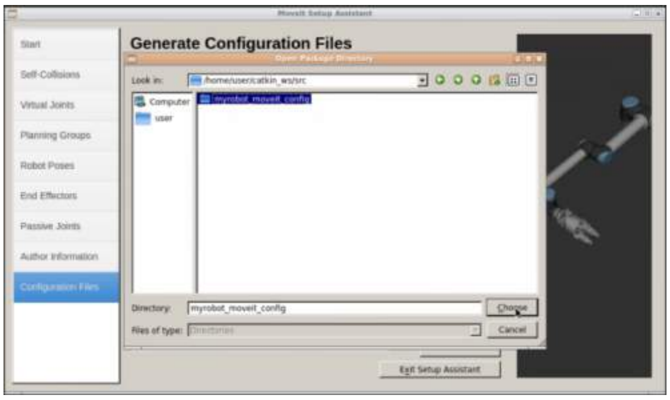
Parent Link (usually part of the arm): wrist_3_link

Parent Group (optional): arm

End-Effectors

h) Now, just enter your name and e-mail at the "Author Information" tab.

i) Finally, go to the "Configuration Files" tab and click the "Browse" button. Navigate to the catkin_ws/src directory, create a new directory, and name it myrobot_moveit_config. "Choose" the directory you've just created.



Generate Configuration Files

Look in: /home/user/catkin_ws/src

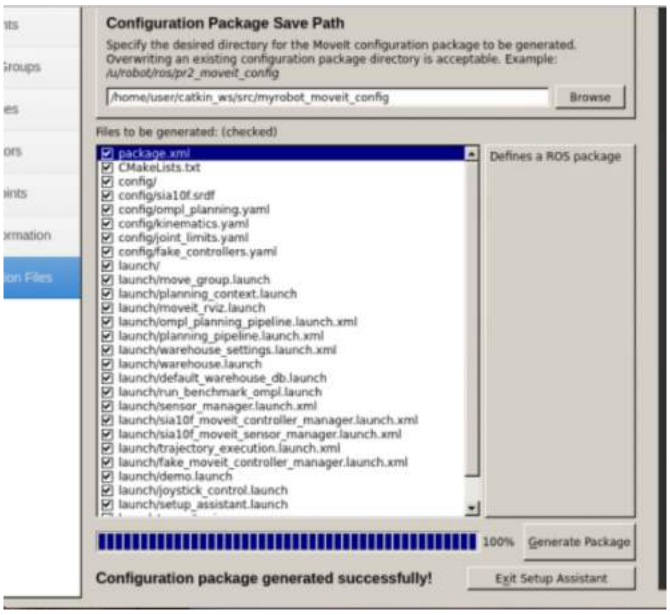
myrobot_moveit_config

Directory: myrobot_moveit_config

Files of type: Directories

Generate Package

Now, click the “Generate Package” button. If everyting goes well, you should see something like this:



Configuration Package Save Path

Specify the desired directory for the MoveIt configuration package to be generated. Overwriting an existing configuration package directory is acceptable. Example: */u/robot/ros/pr2_moveit_config*

```
/home/user/catkin_ws/src/myrobot_moveit_config
```

Files to be generated: (checked)

```
☑ package.xml
☑ CMakeLists.txt
☑ config/
☑ config/sia10f.srdf
☑ config/ompl_planning.yaml
☑ config/kinematics.yaml
☑ config/joint_limits.yaml
☑ config/fake_controllers.yaml
☑ launch/
☑ launch/move_group.launch
☑ launch/planning_context.launch
☑ launch/moveit_rviz.launch
☑ launch/ompl_planning_pipeline.launch.xml
☑ launch/planning_pipeline.launch.xml
☑ launch/warehouse_settings.launch.xml
☑ launch/warehouse.launch
☑ launch/default_warehouse_db.launch
☑ launch/run_benchmark_ompl.launch
☑ launch/sensor_manager.launch.xml
☑ launch/sia10f_moveit_controller_manager.launch.xml
☑ launch/sia10f_moveit_sensor_manager.launch.xml
☑ launch/trajectory_execution.launch.xml
☑ launch/fake_moveit_controller_manager.launch.xml
☑ launch/demo.launch
☑ launch/joystick_control.launch
☑ launch/setup_assistant.launch
```

Defines a ROS package

Configuration package generated successfully!

Package Generated

And that’s it! You have just created a MoveIt package for your articulated robot.

Data for Exercise 2.1

Check the following Notes in order to complete the Exercise: Note 1: If, for any reason, you need to edit your MoveIt package (for instances, in futures exercises you detect that you did an error), you can do that by selecting the Edit Existing Moveit Configuration Package option in the Setup Assistant, and then selecting your package. Note 2: If you modify your MoveIt package, you will need to restart the simulation in order to make this changes to have effect. You can do that by moving to the Next Chapter (Chapter 3), and then going back to this one (Chapter 2).
And that’s it! You’ve created your MoveIt package for your robot. But. . . now what?

Now that you’ve already created a MoveIt package, and you’ve worked a little bit with it, let’s take a deeper look at some key aspects of Moveit.

Let’s start with a quick look at MoveIt architecture. Understanding the architecture of MoveIt! helps to program and interface the robot to MoveIt. Here, you can have a look at a diagram showing MoveIt architecture.

The move_group node

We can say that **move group** is the heart of MoveIt, as this node acts as an integrator of the various components of the robot and delivers actions/services according to the user's needs.

The **move group** node collects robot information, such as the PointCloud, the joint state of the robot, and the transforms (TFs) of the robot in the form of topics and services.

From the parameter server, it collects the robot kinematics data, such as robot description (URDF), SRDF (Semantic Robot Description Format), and the configuration files. The SRDF file and the configuration files are generated while we generate a MoveIt! package for our robot. The configuration files contain the parameter file for setting joint limits, perception, kinematics, end effector, and so on. These are the files that have been created in the **config** folder of your package.

When MoveIt! gets all of this information about the robot and its configuration, we can say it is properly configured and we can start commanding the robot from the user interfaces. We can either use C++ or Python MoveIt! APIs to command the move group node to perform actions, such as pick/place, IK, or FK, among others. Using the RViz motion planning plugin, we can command the robot from the RViz GUI itself. And this is what you are going to do in the next section!

Basic Motion Planning

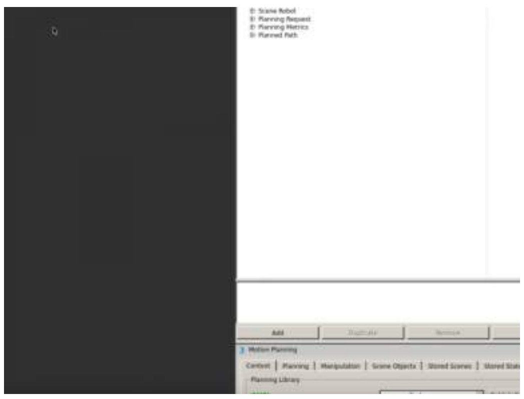
Well... to start, you can just launch the MoveIt Rviz environment and begin to do some tests about Motion Planning. So, follow the next exercise in order to do so!

Exercise 2.2
a) Execute the following command in order to start the MoveIt RViz demo environment.

Execute in WebShell #1

```
[ ]: roslaunch myrobot_moveit_config demo.launch
```

NOTE: It may happen that the Moveit Rviz window appears out of focus. Like this:

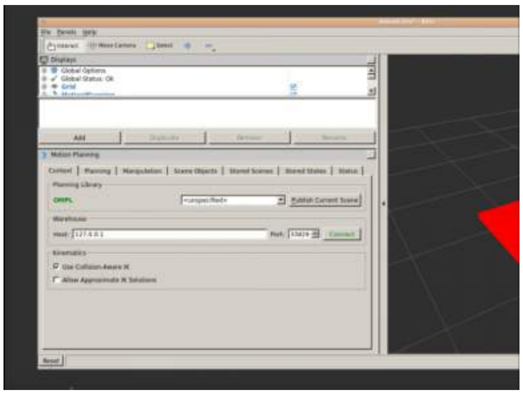


Bad Focus

If this is your case, just type the following command into the Web Shell:

```
[ ]: wmctrl -r :ACTIVE: -e 0,65,24,1500,550
```

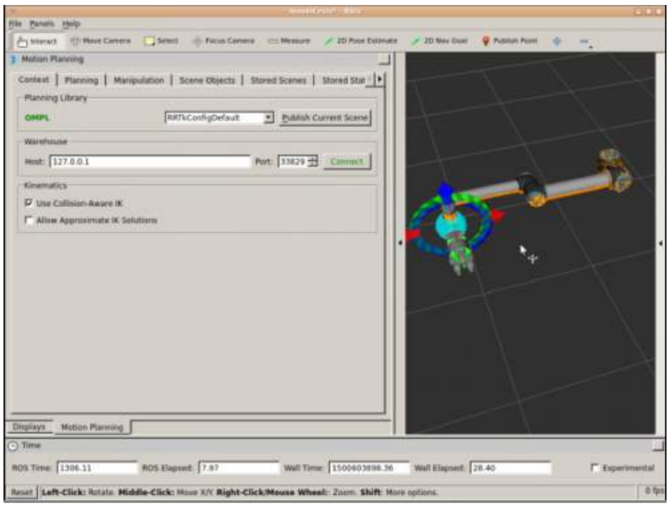
Now, your MoveIt Rviz window should appear like this:



Good Focus

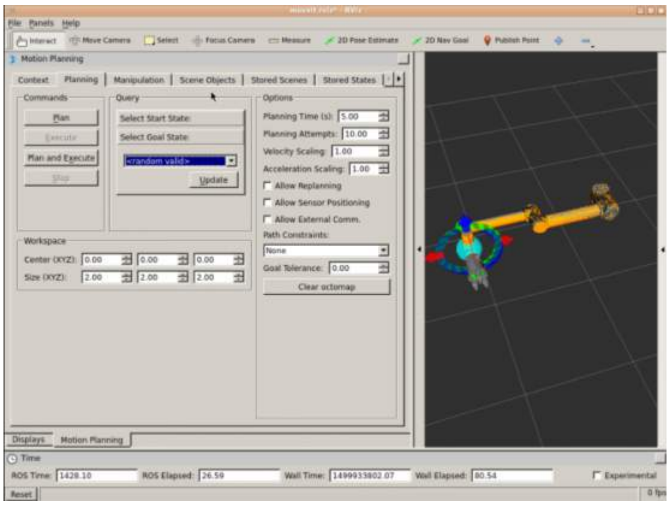
Now, you can just double-click on the top coloured part of the window in order to maximize.

If everything goes OK, you will see something like this:



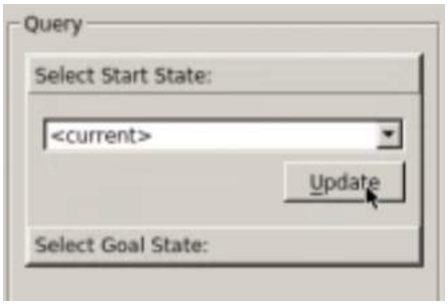
MoveIt RViz with UR5

b) Now, move to the Planning tab. Here:



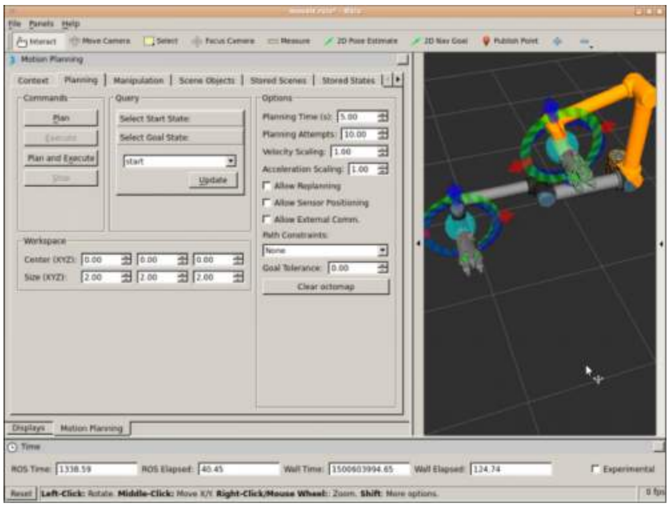
Planning Tab

c) Before start Planning anything, it is always a good practice to update the current Start State.



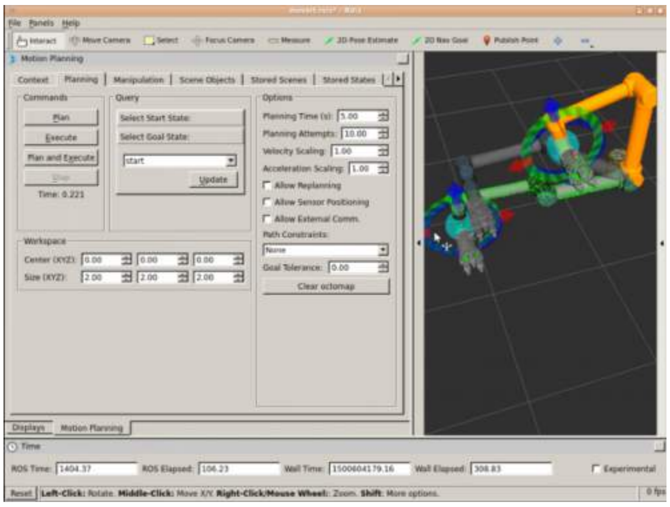
Query

d) At the query section, in the Goal State, you can choose the start option (which one of the Poses you defined in the Previous Exercise) and click on the “Update” button. Your robot scene will be updated with the new position that has been selected.



Updated Scene

e) Now, you can click on the “Plan” button at the “Commands” section. The robot will begin to plan a trajectory to reach that point.



Time: 0.221

start

Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Allow Replanning
Allow Sensor Positioning
Allow External Comm.
Path Constraints: None
Goal Tolerance: 0.00

Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00

ROS Time: 1404.37 ROS Elapsed: 106.23 Wall Time: 1500604179.16 Wall Elapsed: 308.83

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.

Robot Planning

f) Finally, if you click on the “Execute” button, the robot will execute that trajectory.

g) Now just play with the new tool! You can repeat this same process some more times. For instance, instead of moving the robot to the start position, you could set a random valid position as goal. You can also try to check and uncheck the different visualization options that appear in the upper “Displays” section.

You’ve now seen how to perform some basic Motion Planning through the MoveIt RViz GUI, and you’re a little more familiar with MoveIt. So... let’s discuss some interesting points!

MoveIt! planning scene

The term “planning scene” is used to represent the world around the robot and also store the state of the robot itself. The planning scene monitor inside of move_group maintains the planning scene representation. The move_group node consists of another section called the world geometry monitor, which builds the world geometry from the sensors of the robot and from the user input.

The planning scene monitor reads the joint_states topic from the robot, and the sensor information and world geometry from the world geometry monitor. The world scene monitor reads from the occupancy map monitor, which uses 3D perception to build a 3D representation of the environment, called Octomap. The Octomap can be generated from PointClouds, which are handled by a PointCloud occupancy map update plugin and depth images handled by a depth image occupancy map updater. You will see this part in the next chapter, when we introduce Perception.

MoveIt! kinematics handling

MoveIt! provides a great flexibility for switching the inverse kinematics algorithms using the robot plugins. Users can write their own IK solver as a MoveIt! plugin and switch from the default solver plugin whenever required. The default IK solver in MoveIt! is a numerical jacobian-based solver.

Compared to the analytic solvers, the numerical solver can take time to solve IK. The package called IKFast can be used to generate a C++ code for solving IK using analytical methods, which can be used for different kinds of robot manipulators and perform better if the DOF is less than 6. This C++ code can also be converted into the MoveIt! plugin by using some ROS tools.

Forward kinematics and finding jacobians are already integrated into the MoveIt! RobotState class, so we don't need to use plugins for solving FK.

MoveIt! collision checking

The CollisionWorld object inside MoveIt! is used to find collisions inside a planning scene, which are using the FCL (Flexible Collision Library) package as a backend. MoveIt! supports collision checking for different types of objects, such as meshes; primitive shapes, such as boxes, cylinders, cones, spheres, and such; and Octomap.

The collision checking is one of the computationally expensive tasks during motion planning. To reduce this computation, MoveIt! provides a matrix called ACM (Allowed Collision Matrix). It contains a binary value corresponding to the need to check for collision between two pairs of bodies. If the value of the matrix is 1, it means that the collision of the corresponding pair is not needed. We can set the value as 1 where the bodies are always so far that they would never collide with each other. Optimizing ACM can reduce the total computation needed for collision avoidance. This was done when you were creating the package, if you remember!

Moving the real robot

Until now, though, you've only moved the robot in the Moveit application. This is very useful because you can do many tests without worrying about any damage. Anyways, the final goal will be always to move the real robot, right?

The MoveIt package you've created is able to provide the necessary ROS services and actions in order to plan and execute trajectories, but it isn't able to pass this trajectories to the real robot. All the Kinematics you've been performing were executed in an internal simulator that MoveIt provides. In order to communicate with the real robot, it will be necessary to do a couple of modifications to the MoveIt package you've created at the beginning of the Chapter.

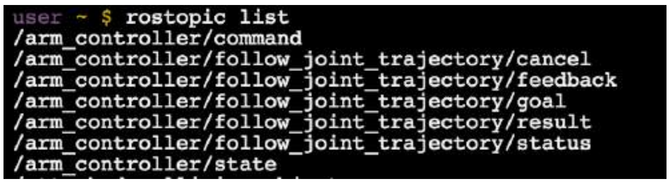
Obviously, in this Course you don't have a real robot to do this, so you will apply the same but for moving the simulated robot. In order to see what you need to change in your MoveIt package, just follow the next Exercise.

Exercise 2.3
a) First of all, you'll need to create a file to define how you will control the joints of your "real" robot. Inside the config folder of your moveit package, create a new file named controllers.yaml. Copy the following content inside it:

```
[ ]: 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
       - name: hand_controller
         action_ns: follow_joint_trajectory
         type: FollowJointTrajectory
         joints:
           - H1_F1J1
           - H1_F1J2
           - H1_F1J3
           - H1_F2J1
           - H1_F2J2
           - H1_F2J3
           - H1_F3J1
           - H1_F3J2
           - H1_F3J3
```

So basically, here you are defining the Action Servers that you will use for controlling the joints of your robot.

First, you are setting the name of the joint trajectory controller Action Server for controlling the arm of the robot. And how do you know that? Well, if you do a rostopic list in any Web Shell, you'll find between your topics the following structure:



```
user ~ $ rostopic list
/arm_controller/command
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/goal
/arm_controller/follow_joint_trajectory/result
/arm_controller/follow_joint_trajectory/status
/arm_controller/state
```

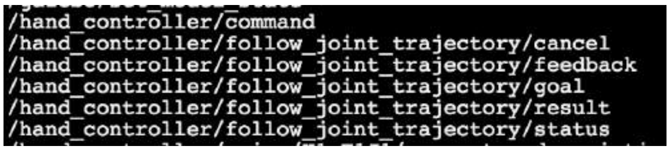
arm controller Action Server

So this way, you can know that your robot has a joint trajectory controller Action Server that is called /arm_controller/follow_joint_trajectory/.

Also, you can find out by checking the message that this Action uses, that it is of the type FollowJointTrajectory.

Finally, you already know the names of the joints that your robot uses. You saw them while you were creating the MoveIt package, and you can also find them in the model.urdf file, at the fh_desc package.

Then, it simply repeats the process described just now, but for the /hand_controller/follow_joint/trajectory action server.



```
/hand_controller/command
/hand_controller/follow_joint_trajectory/cancel
/hand_controller/follow_joint_trajectory/feedback
/hand_controller/follow_joint_trajectory/goal
/hand_controller/follow_joint_trajectory/result
/hand_controller/follow_joint_trajectory/status
```

hand controller Action Server

b) Next, you'll have to create a file to define the names of the joints of your robot. Again inside the config directory, create a new file called joint_names.yaml, and copy the following content in it:

```
[ ]: controller_joint_names: [shoulder_pan_joint, shoulder_lift_joint,
     elbow_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint, H1_F1J1,
     H1_F1J2, H1_F1J3, H1_F2J1, H1_F2J2, H1_F2J3, H1_F3J1, H1_F3J2,
     H1_F3J3]
```

c) Now, if you open the smart_grasping_sandbox_moveit_controller_manager.launch.xml, which is inside the launch directory, you'll see that it's empty. Put the next content inside it:

```
[ ]: <launch>
       <rosparam file="$(find
     myrobot_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_control
     ler_manager/MoveItSimpleControllerManager"/>
     </launch>
```

What you are doing here is basically load the controllers.yaml file you just created, and the MoveItSimpleControllerManager plugin, which will allow you to send the plans calculated in MoveIt to your "real" robot, in this case, the simulated robot.

d) Finally, you will have to create a new launch file that sets up all the system to control your robot. So, inside the launch directory, create a new launch file called my-robot_planning_execution.launch.

[ ]:
```
<launch>

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

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

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

</launch>
```

So finally here, we are loading the joint_names.yaml file, and launching some launch files we need in order to set up the MoveIt environment. You can check what those launch file do, if you want. But let's focus a moment on the joint_state_publisher node that is being launched.

If you do again a rostopic list, you will see that there is a topic called /joint_states. Into this topic is where the state of the joints of the simulated robot are published. So, we need to put this topic into the /source_list parameter, so MoveIt can know where the robot is at each moment.

f) Finally, you just have to launch this launch file you just created (my-robot_planning_execution.launch) and Plan a trajectory, just as you learnt to do in the previous exercise. Once the trajectory is planned, you can press the "Execute" button in order to execute the trajectory in the simulated robot.

Summarizing

Assume that we know the starting pose of the robot, a desired goal pose of the robot, the geometrical description of the robot, and geometrical description of the world. Then, motion planning is the technique to find an optimum path that moves the robot gradually from the start pose to the goal pose, while never touching any obstacles in the world, and without colliding with the robot links.

Here the geometrical description of the robot is our URDF file and the geometrical description of the world can also be included in URDF, and using the laser scanner/3D vision sensor, we can generate the world in 3D, which can help to avoid dynamic obstacles, rather than static objects defined using URDF.

In the case of the robotic arm, the motion planner should find a trajectory (consisting of joint spaces of each joint) in which the links of the robot should never collide with the environment, avoid self-collision (collision between two robot links), and also not violate the joint limits.

MoveIt! can talk to the motion planners through the plugin interface. We can use any motion planner by simply changing the plugin. This method is highly extensible, so we can try our own custom motion planners using this interface. The move_group node talks to the motion planner plugin via the ROS action/services. The default planner for the move_group node is OMPL (http://ompl.kavrakilab.org/).

To start motion planning, we should send a motion planning request to the motion planner, which specifies our planning requirements. The planning requirement may be setting a new goal pose of the end effector; for example, for a pick and place operation.

We can set additional kinematic constraints for the motion planners. Given next are some inbuilt constraints in MoveIt!:

- Position constraints: These restrict the position of a link
- Orientation constraints: These restrict the orientation of a link
- Visibility constraints: These restrict a point on the link to be visible in a particular area (view of a sensor)
- Joint constraints: These restrict a joint within its joint limits
- User-specified constraints: With these constraints, the user can define his own constraints using the callback functions

Using these constraints, we can send a motion planning request and the planner will generate a suitable trajectory, according to the request. The move_group node will generate the suitable trajectory from the motion planner, which obeys all of the constraints. This can be sent to the robot joint trajectory controllers.

Great! So now you already know how to interact with your robot using the MoveIt RViz application, which is a very interesting and useful tool. But, this is not the only way you can communicate with your robot!

In the next unit, you will see how you can perform Motion Planning with code. So... let's go!

##
Solutions
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions for the Unit 2: Unit 2 Solutions

```
[ ]:
```

Unit 3. Doing Motion Planning programatically

ROS Manipulation in 5 Days

Unit 3: Doing Motion Planning Programatically

- ROSject Link: http://bit.ly/2n6gh8t
- Package Name: **fetch_manipulation_gazebo**
- Launch File: **main.launch**

SUMMARY
Estimated time of completion: 2h This unit will show you how to perform motion planning with Python. By completing this unit, you will be able to create a Python program that performs motion planning on your robot.

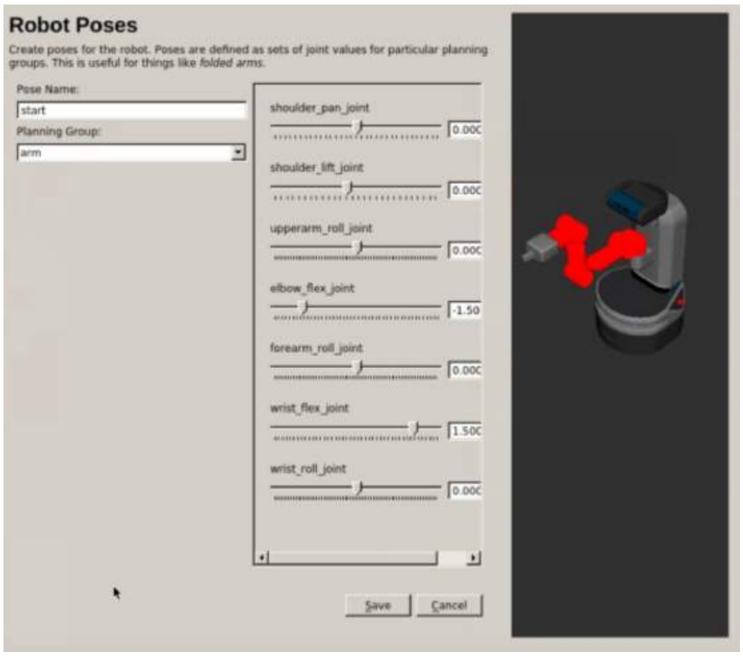
In the previous chapter, you saw that you can plan and execute trajectories for your robot using the MoveIt Rviz environment. But... this is not the common case. Usually, you will want to move your robot with your own scripts. And this is exactly what we are going to do in this chapter! For this course, we are going to use Python to control the robot, because it's easier and faster.

Anyways, let's go step by step. The 1st thing you will have to do is to create a MoveIt package for the Fetch robot, just as you learned to do in the Previous Chapter. So... let's go!

Exercise 3.1
a) Create a MoveIt package for the simulated robot as you saw in the previous chapter. You will have to add 2 Planning Groups to this package. One for the arm, and one for the end-effector. Note: You will find the URDF file needed in the fetch_description package, and it is named fetch.urdf.

You can create as many poses as you want, but these 2 are required:

Name: Start


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 *folded arms*.

Pose Name: start

Planning Group: arm

shoulder_pan_joint 0.000

shoulder_lift_joint 0.000

upperarm_roll_joint 0.000

elbow_flex_joint -1.50

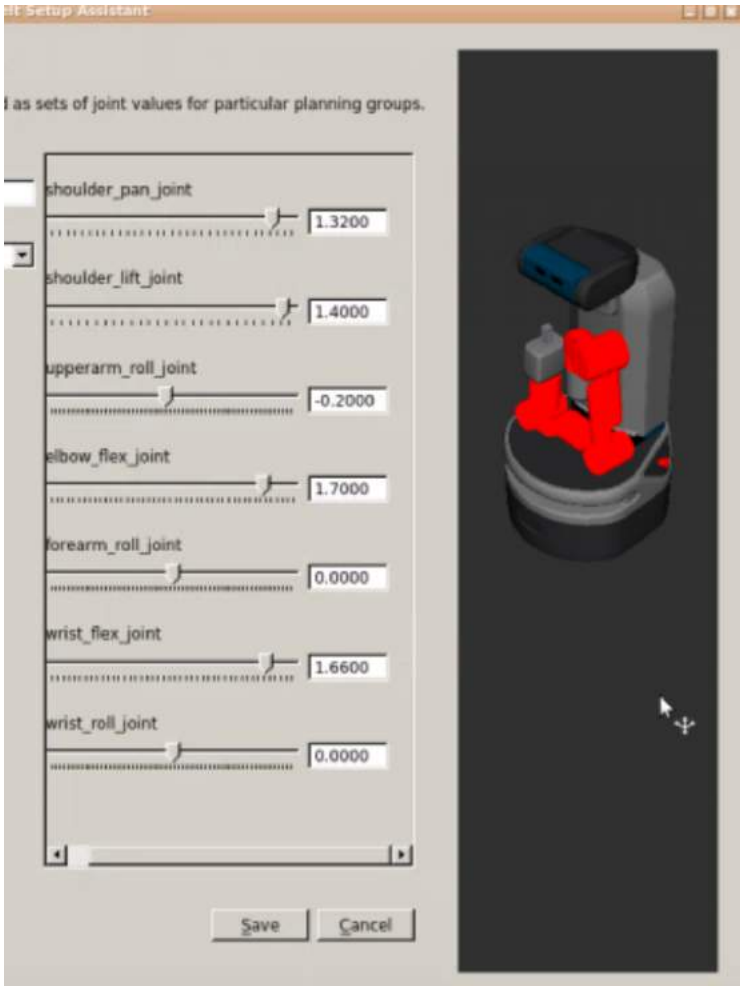
forearm_roll_joint 0.000

wrist_flex_joint 1.500

wrist_roll_joint 0.000

Fetch Pose 1

Name: home



as sets of joint values for particular planning groups.

shoulder_pan_joint 1.3200
shoulder_lift_joint 1.4000
upperarm_roll_joint -0.2000
elbow_flex_joint 1.7000
forearm_roll_joint 0.0000
wrist_flex_joint 1.6600
wrist_roll_joint 0.0000

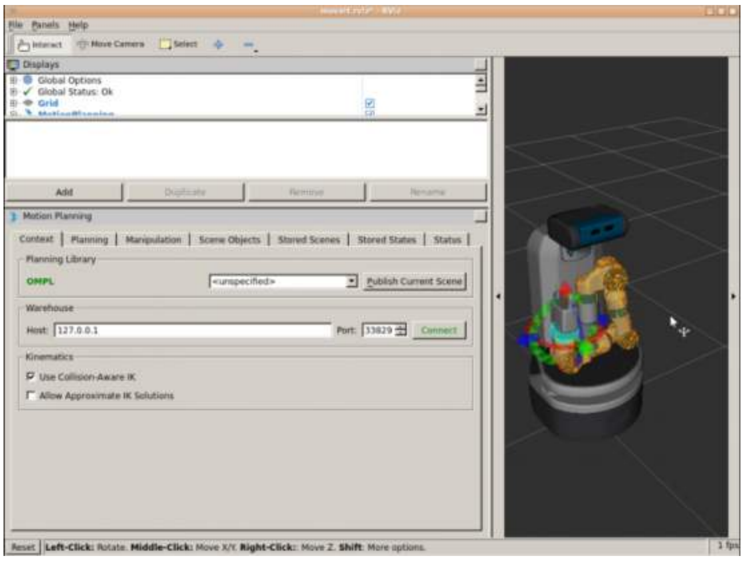
Fetch Pose 2

b) Connect this MoveIt package to the simulated robot, just as you learned to do in the previous chapter.

c) Test that you can plan and execute trajectories using MoveIt, and that these trajectories apply to the simulated robot.

Data for Exercise 3.1
Check the following notes in order to complete the Exercise: **Note 1**: It is possible that you could get confused when you try to add the Gripper to the **controllers.yaml** file. If you have it straight, go ahead and add it. If you don't, you can wait until Chapter 5 to add it, where it will be explained. It is not mandatory to add the Gripper to the controllers.yaml file now. It is mandatory, though, to add it into the Planning Groups and the End-Effector sections when you create the Moveit package. You already know how to do that! **Note 2**: In this case, it is not required to create a Virtual Joint, so you can leave this section empty.

Expected Result for Exercise 3.1



Fetch in MoveIt

```
[ ]: from

     HTML('<iframe width="560" height="315" src="https://www.youtube.com/em
     bed/rfcXZcKZd8A?rel=0&amp;controls=0&amp;showinfo=0" frameborder="0"
     allowfullscreen></iframe>')
```

```
[ ]: <iframe width="560" height="315"
     src="https://www.youtube.com/embed/rfcXZcKZd8A" frameborder="0"
     allow="autoplay; encrypted-media" allowfullscreen></iframe>
```

Solution for Exercise 3.1
Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

If you don't succeed in completing the Exercise, you can have a look at the this **Video**. You'll find here an step by step explanation about how to have it working.

Great! So now that you have created the MoveIt package, you're ready to begin with the main goal of this chapter! Well, almost ready...
Before actually starting with the contents of this chapter, let's execute the following command in order to raise the Fetch robot's torso. This will make it easier in order to plan and execute trajectories with the arm of the robot.

Execute in WebShell #1

```
[ ]: roslaunch fetch_gazebo_demo move_torso.launch
```

You should see how Fetch's torso rises. Something like this:

Fetch Raising Torso

IMPORTANT NOTE
Once the robot is set up, like in the image above, you must stop the command launched by pressing Ctrl+C in the webshell in which it was launched.

IMPORTANT NOTE

Planning a trajectory

As you've seen in the previous chapter, there is a difference between planning a trajectory, and executing it. In this first part of the chapter, we are going to see how to plan a trajectory with Python. For that, just follow the next Demo.

Demo 3.1
a) First of all, you'll need to launch the MoveIt RViz environment. Type the following command:

Execute in WebShell #1

```
[ ]: roslaunch fetch_moveit_config fetch_planning_execution.launch
```

NOTE 1: Note that this command may vary depending on how you've named your MoveIt package and your launch file. In this example command, it is assuming that they're named **fetch_moveit_config** and **fetch_planning_execution.launch**, respectively.

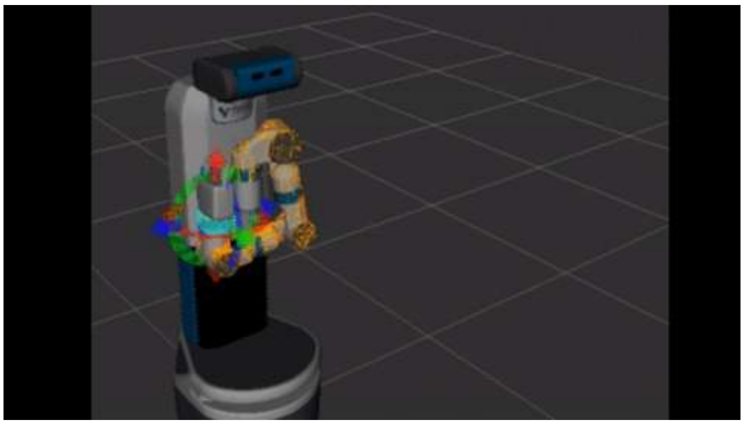
NOTE 2: If the window appears out of focus, just follow the steps described in the previous chapter.

b) Execute the following Python code. You can do this by clicking on the cell and pressing Ctr+Enter, or by clicking on the "Play" icon at the top-right corner of this notebook.

```
[ ]: #! /usr/bin/env python

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

     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")
     display_trajectory_publisher =
     rospy.Publisher('/move_group/display_planned_path',
     moveit_msgs.msg.DisplayTrajectory, queue_size=1)

     pose_target = geometry_msgs.msg.Pose()
     pose_target.orientation.w = 1.0
     pose_target.position.x = 0.96
     pose_target.position.y = 0
     pose_target.position.z = 1.18
     group.set_pose_target(pose_target)

     plan1 = group.plan()

     rospy.sleep(5)

     moveit_commander.roscpp_shutdown()
```

c) After a few seconds, you'll see in MoveIt RViz how the robot is planning the specified motion described in the code above.

Expected Result for Demo 3.1



Motion Planning Fetch

IMPORTANT NOTE
Once you have finished with the Demo, remember to RESTART the Kernel by hitting the 'RESTART KERNEL' icon at the top-right corner of the notebook.

IMPORTANT NOTE
That's great! But. . . how does this code work? What does each part mean? Let's break it down into smaller pieces.

```
[ ]: import sys
     import copy
     import rospy
     import moveit_commander
     import moveit_msgs.msg
     import geometry_msgs.msg
```

In this section of the code, we are just importing some modules and messages that we'll need for the program. The most important one here is the moveit_commander module, which will allow us to communicate with the MoveIt RViz interface.

```
[ ]: moveit_commander.roscpp_initialize(sys.argv)
```

Here, we are just initializing the moveit_commander module.

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

Here, we are just initializing a ROS node.

```
[ ]: robot = moveit_commander.RobotCommander()
```

Here, we are creating a RobotCommander object, which is, basically, an interface to our robot.

```
[ ]: scene = moveit_commander.PlanningSceneInterface()
```

Here, we are creating a PlanningSceneInterface object, which is, basically, an interface to the world that surrounds the robot.

```
[ ]: group = moveit_commander.MoveGroupCommander("arm")
```

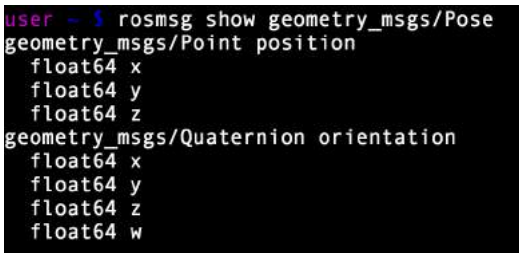
Here, we create a MoveGroupCommander object, which is an interface to the manipulator group of joints that we defined when we created the MoveIt package, back in the Chapter 1. This will allow us to interact with this set of joints, which, in this case, is the full arm.

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

Here, we are defining a Topic Publisher, which will publish into the /move_group/display_planned_path topic. By publishing into this topic, we will be able to visualize the planned motion through the MoveIt RViz interface.

```
[ ]: pose_target = geometry_msgs.msg.Pose()
     pose_target.orientation.w = 1.0
     pose_target.position.x = 0.7
     pose_target.position.y = -0.05
     pose_target.position.z = 1.1
```

Here, we are creating a Pose object, which is the type of message that we will send as a goal. Then, we just give values to the variables that will define the goal Pose. You can have a look at the full structure of the Pose message by issuing the following command on a WebShell: rosmsg show geometry_msgs/Pose



```
user ~ $ rosmsg show geometry_msgs/Pose
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

Pose Message

```
[ ]: plan1 = group.plan()
```

Finally, we are telling the “manipulator” group that we created previously to calculate the plan. If the plan is successfully computed, it will be displayed through MoveIt RViz.

```
[ ]: moveit_commander.roscpp_shutdown()
```

At the end, we just shut down the moveit_commander module.

Exercise 3.2
a) First of all, you’ll need to launch the MoveIt RViz environment. Type the following command:

Execute in WebShell #1

```
[ ]: roslaunch fetch_moveit_config fetch_planning_execution.launch
```

NOTE 1: Note that this command may vary depending on how you’ve named your MoveIt package and your launch file. In this example command, it is assuming that they’re named **fetch_moveit_config** and **fetch_planning_execution.launch**, respectively.

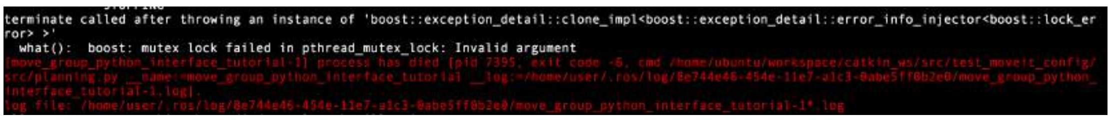
NOTE 2: If the window appears out of focus, just follow the steps described in the previous chapter.

b) Create a new package called my_motion_scripts. Inside this package, create a new directory called src, with a file named planning_script.py. Finally, copy the code you’ve just executed in the Demo above inside this file.

c) Inside the package, also create a launch directory that contains a launch file in order to launch the planning_script.py file.

d) Modify the values assigned to the pose_target variable. Then, launch your code and check if the new Pose was achieved successfully. Repeat the process and try it with different Poses.

Data for Exercise 3.1
Check the following notes in order to complete the exercise: Note 1: When you launch your Python script with your code, you may see the following error:



```
terminate called after throwing an instance of 'boost::exception_detail::clone_impl<boost::exception_detail::error_info_injector<boost::lock_er
ror> >'
  what():  boost: mutex lock failed in pthread_mutex_lock: Invalid argument
[move_group_python_interface_tutorial-1] process has died [pid 7395, exit code -6, cmd /home/ubuntu/workspace/catkin_ws/src/test_moveit_config/
src/planning.py __name:=move_group_python_interface_tutorial __log:=/home/user/.ros/log/8e744e46-454e-11e7-a1c3-0abe5ff0b2e0/move_group_python_
interface_tutorial-1.log].
log file: /home/user/.ros/log/8e744e46-454e-11e7-a1c3-0abe5ff0b2e0/move_group_python_interface_tutorial-1*.log
```

Python Error

This is a known issue, and you don’t need to worry about it. It is caused by certain launched processes that were not closed correctly, but it doesn’t has any effect on the execution of the code itself. This error should be fixed for the newest ROS distributions, like Kinetic.

Planning to a joint space goal

Sometimes, instead of just moving the end-effector towards a goal, we can be interested in setting a goal for a specific joint. Let's see how you can do this.

Demo 3.2
a) First of all, you'll need to launch the MoveIt RViz environment. Type the following command:

Execute in WebShell #1

```
[ ]: roslaunch fetch_moveit_config fetch_planning_execution.launch
```

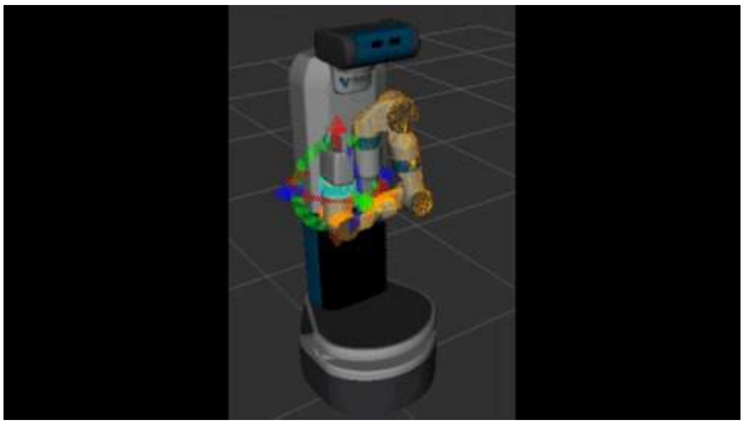
NOTE 1: Note that this command may vary depending on how you've named your MoveIt package and your launch file. In this example command, it is assuming that they're named fetch_moveit_config and fetch_planning_execution.launch, respectively.**
NOTE 2: If the window appears out of focus, just follow the steps described in the previous chapter.**

b) Execute the following Python code:

```
[ ]: #! /usr/bin/env python

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

     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")
     display_trajectory_publisher =
     rospy.Publisher('/move_group/display_planned_path',
     moveit_msgs.msg.DisplayTrajectory, queue_size=1)

     group_variable_values = group.get_current_joint_values()

     group_variable_values[5] = -1.5
     group.set_joint_value_target(group_variable_values)

     plan2 = group.plan()

     rospy.sleep(5)

     moveit_commander.roscpp_shutdown()
```

c) When the code finishes executing, you'll see in MoveIt RViz how the robot is planning the specified motion described in the code above.

Expected Result for Demo 3.2



Motion Planning Fetch 2

IMPORTANT NOTE
Once you have finished with the Demo, remember to RESTART the Kernel by hitting the 'RESTART KERNEL' icon at the top-right corner of the notebook.

IMPORTANT NOTE
That's great, right? But as we did before, let's analyze the new code we've introduced in order to understand what's going on.

```
[ ]: group_variable_values = group.get_current_joint_values()
```

Here, we are getting the current values of the joints.

```
[ ]: group_variable_values[5] = -1.5
     group.set_joint_value_target(group_variable_values)
```

Now, we modify the value of one of the joints, and set this new joint value as a target.

```
[ ]: plan2 = group.plan()
```

Finally, we just compute the plan for the new joint space goal.

Exercise 3.3

a) Inside the my_motion_scripts package, create a new file named joint_planning.py. Do some tests giving different values to different joints.

Getting some useful data

Through code, you can also get some valuable data that you may require for your code. Let's see some examples.

You can get the reference frame for a certain group by executing this line:

```
[ ]: print "Reference frame: %s" % group.get_planning_frame()
```

You can get the end-effector link for a certain group by executing this line:

```
[ ]: print "End effector: %s" % group.get_end_effector_link()
```

You can get a list with all of the groups of the robot, like this:

```
[ ]: print "Robot Groups:"
     print robot.get_group_names()
```

You can get the current values of the joints, like this:

```
[ ]: print "Current Joint Values:"
     print group.get_current_joint_values()
```

You can also get the current Pose of the end-effector of the robot, like this:

```
[ ]: print "Current Pose:"
     print group.get_current_pose()
```

Finally, you can check the general status of the robot, like this:

```
[ ]: print "Robot State:"
     print robot.get_current_state()
```

Exercise 3.4
a) First of all, you'll need to launch the MoveIt RViz environment. Type the following command:

Execute in WebShell #1

```
[ ]: roslaunch fetch_moveit_config fetch_planning_execution.launch
```

NOTE 1: Note that this command may vary depending on how you've named your MoveIt package and your launch file. In this example command, it is assuming that they're named **fetch_moveit_config** and **fetch_planning_execution.launch**, respectively.

NOTE 2: If the window appears out of focus, just follow the steps described in the previous chapter.

b) Create a new file inside the package called get_data.py. Add all the new code you've learnt just above and check what results you get.

Executing a trajectory

So, at this point, you've seen some methods that allow you to plan a trajectory with Python code. But. . . what about executing this trajectory with the "real" robot? In fact, it's very simple. In order to execute a trajectory, you just need to call the go() function from the planning group. Like this:

```
[ ]: group.go(wait=True)
```

By executing this line of code, you will be telling your robot to execute the last trajectory that has been set for the Planning Group. Let's try this in an exercise!

Exercise 3.5
a) First of all, you'll need to launch the MoveIt RViz environment. Type the following command:

Execute in WebShell #1

```
[ ]: roslaunch fetch_moveit_config fetch_planning_execution.launch
```

NOTE 1: Note that this command may vary depending on how you've named your MoveIt package and your launch file. In this example command, it is assuming that they're named **fetch_moveit_config** and **fetch_planning_execution.launch**, respectively.

NOTE 2: If the window appears out of focus, just follow the steps described in the previous chapter.

b) Create a new Python script called execute_trajectory.py, and copy the code shown in Demo 3.1 inside of it.

c) Add a line into your new script in order to execute that trajectory.

d) You can try this with any of the codes you have created for planning trajectories.

Expected Result for Demo 3.2

Motion Execution Fetch

Exercise 3.6
a) First of all, you'll need to launch the MoveIt RViz environment. Type the following command:

Execute in WebShell #1

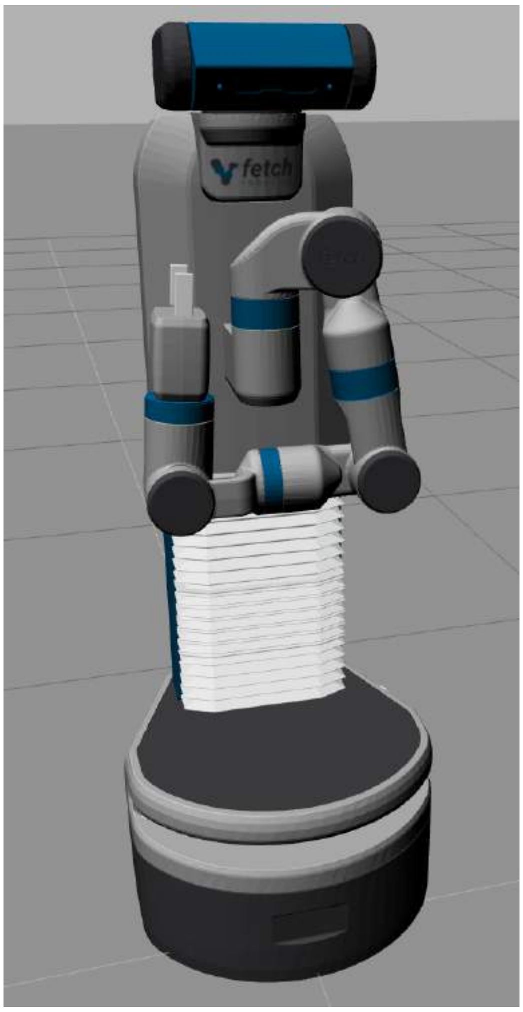
```
[ ]: roslaunch fetch_moveit_config fetch_planning_execution.launch
```

NOTE 1: Note that this command may vary depending on how you've named your MoveIt package and your launch file. In this example command, it is assuming that they're named **fetch_moveit_config** and **fetch_planning_execution.launch**, respectively.

NOTE 2: If the window appears out of focus, just follow the steps described in the previous chapter.

b) Modify the execute_trajectory.py script, so that it now concatenates at least 2 motions.

IMPORTANT NOTE
When you have finished this chapter, make sure to return the robot to the home position.


Fetch in Home Position

IMPORTANT NOTE
And that's it! You have finished this chapter! I really hope that you enjoyed it and, most of all, have learned a lot! Now, if you want to learn how you can add Perception to Motion Planning tasks, just go to the next chapter!

##
Solutions
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions for the Unit 3: Unit 3 Solutions

```
[ ]:
```

Unit 4. Adding Perception to Motion Planning

ROS Manipulation in 5 Days

Unit 4: Adding Perception to Motion Planning

- ROSject Link: http://bit.ly/2n6Zbrh
- Package Name: **fetch_manipulation_gazebo**
- Launch File: **main.launch**

SUMMARY
Estimated time of completion: 2h This unit will show you how to perform motion planning with Python. By completing this unit, you will be able to create a Python program that performs motion planning on your robot.

In the previous chapter, you saw that you can plan and execute trajectories for your robot using Python code. But... you weren't taking into account Perception, were you? Usually, you will want to take the data from a 3D vision sensor into account; for instance, a Kinect camera. This will give you real-time information about the environment, and will allow you to plan more realistic motions, introduing any change that the environment suffers. So, in this chapter, we are going to learn how you can add a 3D vision sensor to MoveIt in order to perform vision-assisted Motion Planning!

But, first of all, let's make some changes in the current simulation in order to be able to better work with Perception. For that, just follow the instructions described in the next exercise!

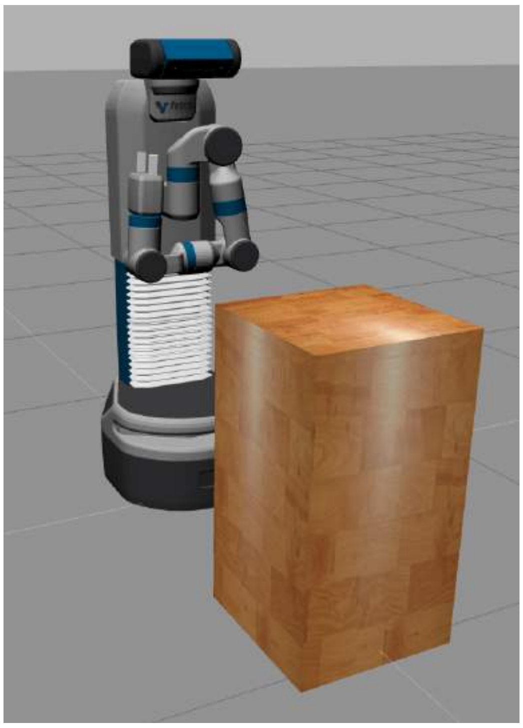
Exercise 5.1
a) Create a new file named table.urdf in your workspace. Paste the following code into that file.

```
[ ]: <robot name="simple_box">
       <link name="my_box">
         <inertial>
           <origin xyz="0 0 0.0145"/>
           <mass value="0.1" />
```

```
        <inertia  ixx="0.0001" ixy="0.0"  ixz="0.0"  iyy="0.0001"
iyz="0.0"  izz="0.0001" />
      </inertial>
      <visual>
        <origin xyz="-0.23 0 0.215"/>
        <geometry>
          <box size="0.47 0.46 1.3"/>
        </geometry>
      </visual>
      <collision>
        <origin xyz="-0.23 0 0.215"/>
        <geometry>
          <box size="0.47 0.46 1.3"/>
        </geometry>
      </collision>
    </link>
    <gazebo reference="my_box">
      <material>Gazebo/Wood</material>
    </gazebo>
    <gazebo>
      <static>true</static>
    </gazebo>
  </robot>
```

b) Execute the following command in order to spawn an object right in front of the Fetch robot.

```
[ ]: rosrun gazebo_ros spawn_model -file
     /home/user/catkin_ws/src/table.urdf -urdf -x 1 -model my_object
```



Object Spawned

c) Execute the following command in order to move the Fetch robot's head, so that it points to the new spawned object.

```
[ ]: roslaunch fetch_gazebo_demo move_head.launch
```

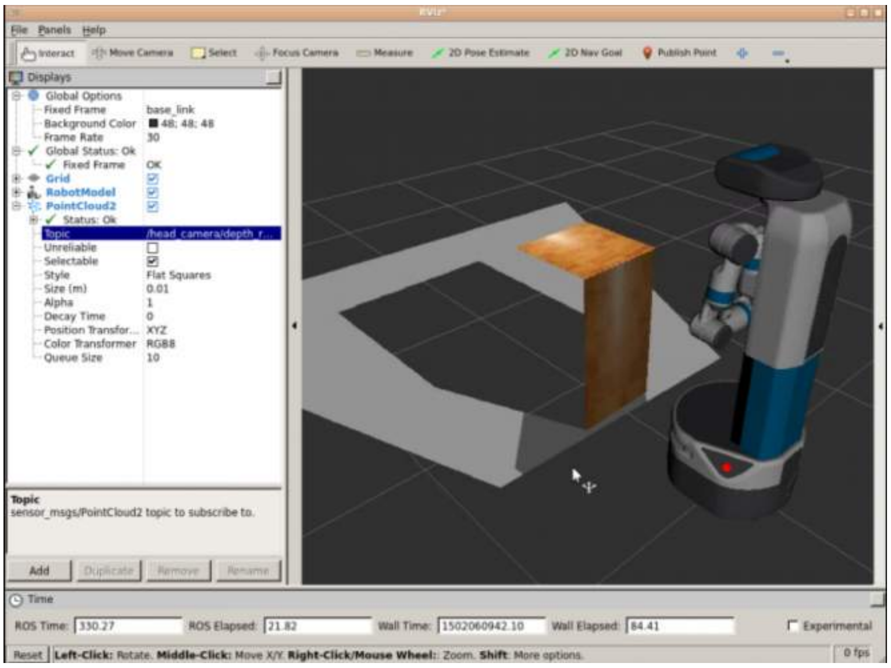
Fetch Looking Down

IMPORTANT NOTE
Once the robot is set up, like in the image above, you must stop the command launched by pressing Ctrl+C in the webshell from which it was launched.

IMPORTANT NOTE

d) Launch RViz and add the corresponding element in order to visualize the PointCloud of the camera.

Expected Result for Exercise 5.1



```
Global Options
  Fixed Frame          base_link
  Background Color     ■ 48; 48; 48
  Frame Rate           30
✓ Global Status: Ok
  ✓ Fixed Frame        OK
Grid                   ☑
RobotModel             ☑
PointCloud2            ☑
  ✓ Status: Ok
  Topic                /head_camera/depth_r...
  Unreliable           ☐
  Selectable           ☑
  Style                Flat Squares
  Size (m)             0.01
  Alpha                1
  Decay Time           0
  Position Transfor... XYZ
  Color Transformer    RGB8
  Queue Size           10

Topic
sensor_msgs/PointCloud2 topic to subscribe to.

ROS Time: 330.27   ROS Elapsed: 21.82   Wall Time: 1502060942.10   Wall Elapsed: 84.41   ☐ Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
```

Fetch and Object in RViz

Great! Now you have already created the appropriate environment in order to work with Perception. So... let's now see how we can add Perception to everything we've learned about Motion Planning in the previous chapter!

Adding Perception to MoveIt

In order to be able to add a sensor to the MoveIt package that you created in the previous chapter, you'll need to do some modifications inside the package, of course. In order to see exactly what you need to do, just follow the next exercise.

Exercise 5.2
a) First of all, you'll need to create a new file inside the config folder, named sensors_rgbd.yaml. Inside that file, copy the following contents:

```
[ ]: sensors:
         - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
           point_cloud_topic: /head_camera/depth_registered/points
           max_range: 5
           padding_offset: 0.01
           padding_scale: 1.0
           point_subsample: 1
           filtered_cloud_topic: output_cloud
```

Basically, what you are doing here is configuring the plugin that we'll use in order to interface the 3D sensor with MoveIt. The parameters that you are defining in the file are the following:

- sensor_plugin: This parameter specifies the name of the plugin we are using in the robot.
- point_cloud_topic: The plugin will listen to this topic for PointCloud data.
- max_range: This is the distance limit, in meters, in which any points above the range will not be used for processing.
- padding_offset: This value will be taken into account for robot links and attached objects when filtering clouds containing the robot links (self-filtering).
- padding_scale: This value will also be taken into account while self-filtering.
- point_subsample: If the update process is slow, points can be subsampled. If we make this value greater than 1, the points will be skipped instead of processed.
- filtered_cloud_topic: This is the final filtered cloud topic. We will get the processed PointCloud through this topic. It can be used mainly for debugging.

b) Next, you'll need to fill in the existing, but blank, fetch_moveit_sensor_manager.launch.xml file, which is located in the launch folder. You'll need to load the YAML file you've just created into this file.

```
[ ]: <launch>
         <rosparam command="load" file="$(find
     test_moveit_config)/config/sensors_rgbd.yaml" />
     </launch>
```

NOTE: Note that the content of the rosparam tag may vary depending on how you've named your MoveIt package. In this example command, it assumes that the package is named fetch_moveit_config.**

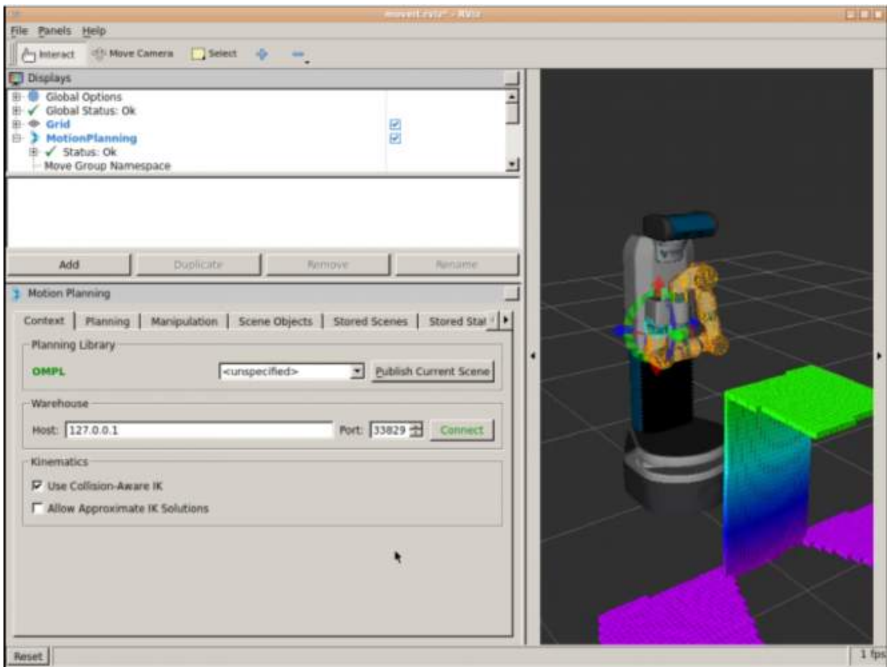
c) Finally, you'll have to have a look at the sensor_manager.launch.xml file. It should look something like this:

```
[ ]: <launch>

       <!-- This file makes it easy to include the settings for sensor
     managers -->

       <!-- Params for the octomap monitor -->
       <!--  <param name="octomap_frame" type="string" value="some frame in
     which the robot moves" /> -->
       <param name="octomap_resolution" type="double" value="0.025" />
       <param name="max_range" type="double" value="5.0" />

       <!-- Load the robot specific sensor manager; this sets the
     moveit_sensor_manager ROS parameter -->
       <arg name="moveit_sensor_manager" default="fetch" />
       <include file="$(find fetch_moveit_config)/launch/$(arg
     moveit_sensor_manager)_moveit_sensor_manager.launch.xml" />

     </launch>
```

NOTE: Note that the content of the include tag may vary depending on how you've named your MoveIt package. In this example command, it assumes that the package is named fetch_moveit_config.**

d) Now, you can launch the MoveIt RViz environment again, and you'll see a PointCloud in the scene, showing what the robot is visualizing.

Expected Result for Exercise 5.2



MoveIt RViz with Point Cloud

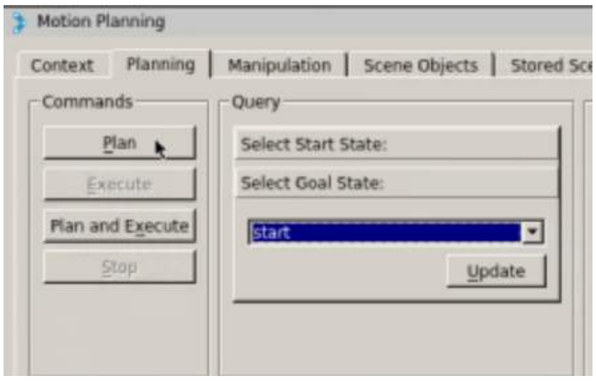
Interesting, right? But how does this work? What's going on internally? Let's explain it a little bit. Basically, you are using a plugin (PointCloudUpdater), that brings the simulated PointCloud obtained from the camera that is placed in Fetch's head, into the MoveIt planning scene. The robot environment is mapped as an octree representation, which can be built using a library called OctoMap. The OctoMap is incorporated as a plugin in MoveIt (called the Occupany Map Updator plugin), which can update octree from different kinds of sensor inputs, such as PointClouds and depth images from 3D vision sensors. Currently, there are the following plugins for handling 3D data in MoveIt:

- PointCloud Occupancy Map Updater: This plugin can take input in the form of PointClouds (sensor_msgs/PointCloud2). This is the one you are using in this chapter.
- Depth Image Occupancy Map Updater: This plugin can take input in the form of input depth images (sensor_msgs/Image).

So now you're getting real-time data from the robot's environment. Do you think this will affect the Motion Plans that are calculated in any way? Let's do the following exercise in order to check that!

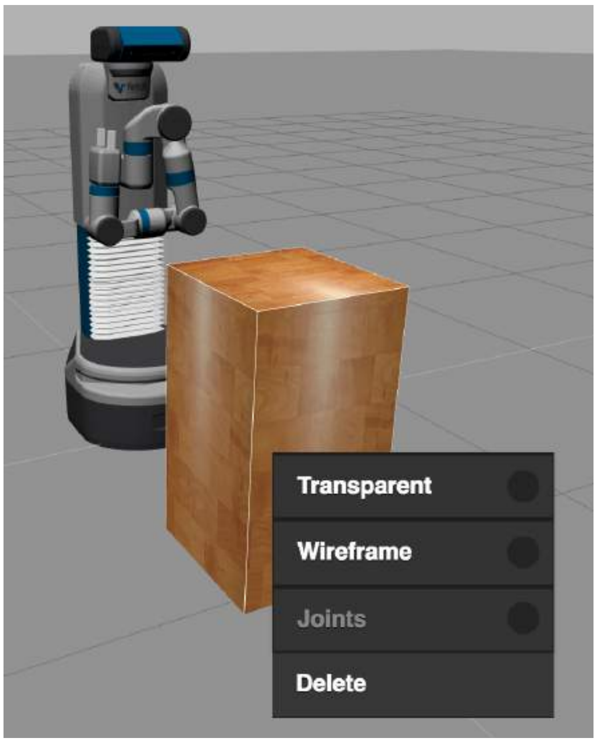
Exercise 5.3
a) Launch the MoveIt RViz environment with Perception, if you don't have it started.

b) In the motion screen, select the start Pose that you created in the previous chapter, and plan a trajectory to go there.



Select Goal State

c) Remove the object of the simulation by right-clicking on it, and selecting the Delete option.



Delete Object

d) Plan a trajectory to the start position again, and check if there is any difference from the previous trajectory.

That's right! The calculated Motion Plans will be different, depending on how the planning scene is. Makes sense, right?

Exercise 5.4

a) Modify the URDF file of the object that you are spawning a little bit, and try to spawn it in different positions.

b) Keep on performing Motion Planning with the object in different positions and with different shapes, and see how it affects the trajectories that are planned. You can also execute these trajectories, if you want.

Until now, you've been working using the PointCloud that is generated by the 3D sensor that is incorporated in the Fetch robot. But, we've previously seen that this is not the only way that we can add Perception to MoveIt, right? By doing the following exercise, let's see how we can also add Perception without using the PointCloud.

Exercise 5.5

a) Make sure that there is an object in front of the robot. If there isn't, execute the following command in order to spawn another object right in front of the Fetch robot.

```
[ ]: rosrun gazebo_ros spawn_model -file
     /home/user/catkin_ws/src/object.urdf -urdf -x 1 -model my_object
```

b) Modify the configuration files in order to use the DepthImageUpdater plugin, instead of the one you are currently using. As an example, you can have a look at the following configuration file:

```
[ ]: sensors:
          - sensor_plugin: occupancy_map_monitor/DepthImageOctomapUpdater
            image_topic: /head_camera/depth_registered/image_raw
            queue_size: 5
            near_clipping_plane_distance: 0.3
            far_clipping_plane_distance: 5.0
            skip_vertical_pixels: 1
            skip_horizontal_pixels: 1
            shadow_threshold: 0.2
            padding_scale: 4.0
            padding_offset: 0.03
            filtered_cloud_topic: output_cloud
```

c) Launch the whole environment again and plan a trajectory to check if it is detecting the environment correctly.

IMPORTANT NOTE
Once you are done with this chapter, make sure to DELETE the object you've added, and return the robot to the home position.

IMPORTANT NOTE
And that's it! You have finished this chapter! I really hope that you enjoyed it and, most of all, have learned a lot! Now, if you want to learn how you can perform Grasping, just go to the next chapter!

##
Solutions
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions for the Unit 4: Unit 4 Solutions

```
[ ]:
```

Unit 5. Grasping

ROS Manipulation in 5 Days

Unit 5: Grasping

- ROSject Link: http://bit.ly/2n9tttu
- Package Name: **fetch_manipulation_gazebo**
- Launch File: **main.launch**

SUMMARY
Estimated time of completion: 2h This unit will show you how to perform motion planning with Python. By completing this unit, you will be able to create a Python program that performs motion planning on your robot.

What is Grasping?

When working with manipulator robots, one of the main goals to reach is picking an object up from one position, and placing it in another one, which is commonly known as pick & place. We call Grasping the process of picking an object up by the robot end-effector (which can be a hand, a gripper, etc.). Although this may sound like a very simple task, it is not. In fact, it's a very complex process, because lots of variables need to be taken into account when picking the object. We humans handle our grasping using our intelligence, but in a robot, we have to create rules for it. One of the constraints in grasping is force; the gripper/end-effector should adjust the grasping force for picking up the object, but not make any deformation on the object while grasping it.

Grasping using MoveIt!

First of all, we are going to see how to perform Grasping in MoveIt. This means, without controlling the real robot.

And for that, we are going to be using the moveit_simple_grasps package. This package is very useful in order to generate grasp poses for simple objects, such as blocks or cylinders. And it's quite simple, which helps in the process of learning. This package takes the position of the object to be grasped as input, and generates the necessary grasping sequences in order to pick the object up.

This package already supports robots such as Baxter, REEM, and Clam arm, among others. But it is quite easy to interface any other custom manipulator robot to the package, without having to modify too many of the codes.

Exercise 6.1
a) First of all, you will create a new package called myrobot_grasping. Inside this package, create a new folder named launch, which will contain a launch file called grasp_generator_server.launch. Copy the following content inside this file.

```
[ ]: <launch>
       <arg name="robot" default="fetch"/>
       <arg name="group"        default="arm"/>
       <arg name="end_effector" default="gripper"/>

       <node pkg="moveit_simple_grasps" type="moveit_simple_grasps_server"
     name="moveit_simple_grasps_server">
         <param name="group"        value="$(arg group)"/>
         <param name="end_effector" value="$(arg end_effector)"/>
         <rosparam command="load" file="$(find
     myrobot_grasping)/config/$(arg robot)_grasp_data.yaml"/>
       </node>

     </launch>
```

Basically, what we are doing in this launch file is starting a grasp server that will provide grasp sequences to a grasp client node. We need to specify the following into the node:

- Planning Group of the arm
- Planning Group of end-effector

Also, we are loading the fetch_grasp_data.yaml file, which will contain detailed information about the gripper. We are going to create this file right now.

b) Inside this package, create a new folder named config, and create a new file named fetch_grasp_data.yaml inside this folder. Copy the following contents into this file.

```
[ ]: base_link: 'base_link'
     gripper:
       #The end effector name for grasping
```

```
end_effector_name: 'gripper'

# Gripper joints
joints: ['l_gripper_finger_joint', 'r_gripper_finger_joint']

#Posture of grippers before grasping
pregrasp_posture: [0.048, 0.048]
pregrasp_time_from_start: 4.0
grasp_posture: [0.016, 0.016]
grasp_time_from_start: 4.0
postplace_time_from_start: 4.0

# Desired pose from end effector to grasp [x, y, z] + [R, P, Y]
grasp_pose_to_eef: [-0.12, 0.0, 0.0]
grasp_pose_to_eef_rotation: [0.0, 0.0, 0.0]
end_effector_parent_link: 'wrist_roll_link'
```

So, in this file, we are basically providing detailed information about the gripper. These parameters can be tuned in order to improve the grasping process. The most important parameters that we are defining here are the following:

- end_effector_name: The name of the end-effector group, as stated in the MoveIt package.
- joints: The joints that form the gripper.
- pregrasp_posture: The position of the gripper before grasping (OPEN position).
- pregrasp_time_from_start: The time to wait before grasping.
- grasp_posture: The position of the gripper when grasping (CLOSED position).
- grasp_time_from_start: The time to wait after grasping.
- postplace_time_from_start: The name of the end-effector.
- grasp_pose_to_eef: The desired pose from end-effector to grasp - [x,y,z].
- grasp_pose_to_eef: The desired pose from end-effector to grasp - [roll, pitch, yaw].
- end_effector_parent_link: The name of the link that connects the gripper to the robot.

So, you have now created the basic structure to create a Grasping Server, which will allow you to send object positions to this server, and it will provide you with a sequence in order to grasp the specified object. But, for that, we need to communicate with that server, right? And that's exactly what you are going to do in the next section!

Creating a pick and place task

There are various ways in which we can perform pick and place tasks. For instance, we could send the robot a predefined sequence of joint values directly, which will cause the robot to always perform the same predefined motion. So, for this case, we must always place the object in the same position. This method is called forward kinematics because you need to previously know the sequence of joint values that are required in order to perform a certain trajectory. Another method would be by using inverse kinematics (IK), without any vision feedback. In this case, we provide the robot with the Pose (x, y, and z) where the object to picked is, and by doing some IK calculations, the robot would know which motions it needs to perform in order to reach the object's Pose. Finally, another method would be to use inverse kinematics, but with vision support or feedback. In this case, we would use a node that would identify the Pose of the object by reading the data of the sensors of the robot; for instance, a Kinect camera. Then, this vision node would provide the Pose of the object, and again, by using IK calculations, the robot would know the required motions to perform in order to reach the object. You are probably thinking that the most efficient and stylish method is the third one, and you're right. But, this method would require you to have some Object Recognition knowledge, which is not the subject of this course. So, for this course, we are going to use the second method, which in the end, is the same as the third one, without the object recognition part. If you already have some Perception knowledge, you can freely add an Object Recognition node that sends the Pose of the object, without having to set this Pose in the code itself.

So, to summarize, what you are going to do in this section of the course is create a node that will send an object Pose to the Grasping Server, in order to receive the appropiate joint values to reach that object. Well then, enough talking, let's get to work!

Robot Preparation

Before actually starting, let's first execute the following commandw in order to raise the Fetch robot's torso, and to properly orientate its head, as you did in the previous Units. This will make it easier in order to plan and execute trajectories with the arm of the robot.

Execute in WebShell #1

```
[ ]: roslaunch fetch_gazebo_demo move_torso.launch
```

```
[ ]: roslaunch fetch_gazebo_demo move_head.launch
```

At the end, you should have Fetch robot with its torso raised, and its head looking down. Like in the image below:

Exercise 6.2

a) Inside the package that you created in the previous exercise, create a new folder named scripts. Inside this folder, create a new file named pick_and_place.py. Copy the following code inside it.

```
[ ]: #!/usr/bin/env python

     import rospy

     from moveit_commander import RobotCommander, PlanningSceneInterface
     from moveit_commander import roscpp_initialize, roscpp_shutdown
     from actionlib import SimpleActionClient, GoalStatus

     from geometry_msgs.msg import Pose, PoseStamped, PoseArray, Quaternion
     from moveit_msgs.msg import PickupAction, PickupGoal
     from moveit_msgs.msg import PlaceAction, PlaceGoal
     from moveit_msgs.msg import PlaceLocation
     from moveit_msgs.msg import MoveItErrorCodes
     from moveit_simple_grasps.msg import GenerateGraspsAction,
     GenerateGraspsGoal, GraspGeneratorOptions

     from tf.transformations import quaternion_from_euler

     import sys
     import copy
     import numpy


     # Create dict with human readable MoveIt! error codes:
     moveit_error_dict = {}
     for name in MoveItErrorCodes.__dict__.keys():
         if not name[:1] == '_':
             code = MoveItErrorCodes.__dict__[name]
             moveit_error_dict[code] = name


     class Pick_Place:
         def __init__(self):
             # Retrieve params:
             self._table_object_name =
     rospy.get_param('~table_object_name', 'Grasp_Table')
             self._grasp_object_name =
     rospy.get_param('~grasp_object_name', 'Grasp_Object')

             self._grasp_object_width =
     rospy.get_param('~grasp_object_width', 0.01)

             self._arm_group     = rospy.get_param('~arm', 'arm')
             self._gripper_group = rospy.get_param('~gripper', 'gripper')

             self._approach_retreat_desired_dist =
     rospy.get_param('~approach_retreat_desired_dist', 0.2)
             self._approach_retreat_min_dist =
     rospy.get_param('~approach_retreat_min_dist', 0.1)

             # Create (debugging) publishers:
             self._grasps_pub = rospy.Publisher('grasps', PoseArray,
     queue_size=1, latch=True)
             self._places_pub = rospy.Publisher('places', PoseArray,
     queue_size=1, latch=True)

             # Create planning scene and robot commander:
             self._scene = PlanningSceneInterface()
             self._robot = RobotCommander()

             rospy.sleep(1.0)

             # Clean the scene:
             self._scene.remove_world_object(self._table_object_name)
             self._scene.remove_world_object(self._grasp_object_name)

             # Add table and Coke can objects to the planning scene:
             self._pose_table    = self._add_table(self._table_object_name)
             self._pose_coke_can =
     self._add_grasp_block_(self._grasp_object_name)

             rospy.sleep(1.0)

             # Define target place pose:
             self._pose_place = Pose()

             self._pose_place.position.x = self._pose_coke_can.position.x
             self._pose_place.position.y = self._pose_coke_can.position.y -
     0.10
             self._pose_place.position.z = self._pose_coke_can.position.z +
     0.08

             self._pose_place.orientation =
     Quaternion(*quaternion_from_euler(0.0, 0.0, 0.0))

             # Retrieve groups (arm and gripper):
             self._arm     = self._robot.get_group(self._arm_group)
             self._gripper = self._robot.get_group(self._gripper_group)

             # Create grasp generator 'generate' action client:
             self._grasps_ac =
     SimpleActionClient('/moveit_simple_grasps_server/generate',
     GenerateGraspsAction)
             if not self._grasps_ac.wait_for_server(rospy.Duration(5.0)):
                 rospy.logerr('Grasp generator action client not
     available!')
                 rospy.signal_shutdown('Grasp generator action client not
     available!')
                 return

             # Create move group 'pickup' action client:
             self._pickup_ac = SimpleActionClient('/pickup', PickupAction)
             if not self._pickup_ac.wait_for_server(rospy.Duration(5.0)):
                 rospy.logerr('Pick up action client not available!')
                 rospy.signal_shutdown('Pick up action client not
     available!')
                 return

             # Create move group 'place' action client:
             self._place_ac = SimpleActionClient('/place', PlaceAction)
             if not self._place_ac.wait_for_server(rospy.Duration(5.0)):
                 rospy.logerr('Place action client not available!')
                 rospy.signal_shutdown('Place action client not
     available!')
                 return

             # Pick Coke can object:
             while not self._pickup(self._arm_group,
     self._grasp_object_name, self._grasp_object_width):
                 rospy.logwarn('Pick up failed! Retrying ...')
                 rospy.sleep(1.0)

             rospy.loginfo('Pick up successfully')

             # Place Coke can object on another place on the support
     surface (table):
             while not self._place(self._arm_group,
     self._grasp_object_name, self._pose_place):
                 rospy.logwarn('Place failed! Retrying ...')
                 rospy.sleep(1.0)

             rospy.loginfo('Place successfully')

         def __del__(self):
             # Clean the scene:
             self._scene.remove_world_object(self._grasp_object_name)
             self._scene.remove_world_object(self._table_object_name)

         def _add_table(self, name):
             p = PoseStamped()
             p.header.frame_id = self._robot.get_planning_frame()
             p.header.stamp = rospy.Time.now()

             p.pose.position.x = 1.0
             p.pose.position.y = 0.08
             p.pose.position.z = 1.02

             q = quaternion_from_euler(0.0, 0.0, numpy.deg2rad(90.0))
             p.pose.orientation = Quaternion(*q)
             # Table size from ~/.gazebo/models/table/model.sdf, using the
     values
             # for the surface link.
             self._scene.add_box(name, p, (0.86, 0.86, 0.02))

             return p.pose

         def _add_grasp_block_(self, name):
             p = PoseStamped()
             p.header.frame_id = self._robot.get_planning_frame()
             p.header.stamp = rospy.Time.now()

             p.pose.position.x = 0.62
             p.pose.position.y = 0.21
             p.pose.position.z = 1.07

             q = quaternion_from_euler(0.0, 0.0, 0.0)
             p.pose.orientation = Quaternion(*q)

             # Coke can size from
     ~/.gazebo/models/coke_can/meshes/coke_can.dae,
             # using the measure tape tool from meshlab.
             # The box is the bounding box of the coke cylinder.
             # The values are taken from the cylinder base diameter and
     height.
             self._scene.add_box(name, p, (0.077, 0.077, 0.070))

             return p.pose

         def _generate_grasps(self, pose, width):
             """
             Generate grasps by using the grasp generator generate action;
     based on
             server_test.py example on moveit_simple_grasps pkg.
             """

             # Create goal:
             goal = GenerateGraspsGoal()

             goal.pose  = pose
             goal.width = width

             options = GraspGeneratorOptions()
             # simple_graps.cpp doesn't implement GRASP_AXIS_Z!
             #options.grasp_axis      = GraspGeneratorOptions.GRASP_AXIS_Z
             options.grasp_direction =
     GraspGeneratorOptions.GRASP_DIRECTION_UP
             options.grasp_rotation  =
     GraspGeneratorOptions.GRASP_ROTATION_FULL

             # @todo disabled because it works better with the default
     options
             #goal.options.append(options)

             # Send goal and wait for result:
             state = self._grasps_ac.send_goal_and_wait(goal)
             if state != GoalStatus.SUCCEEDED:
                 rospy.logerr('Grasp goal failed!: %s' %
     self._grasps_ac.get_goal_status_text())
                 return None

             grasps = self._grasps_ac.get_result().grasps

             # Publish grasps (for debugging/visualization purposes):
             self._publish_grasps(grasps)

             return grasps

         def _generate_places(self, target):
             """
             Generate places (place locations), based on
             https://github.com/davetcoleman/baxter_cpp/blob/hydro-devel/
             baxter_pick_place/src/block_pick_place.cpp
             """

             # Generate places:
             places = []
             now = rospy.Time.now()
             for angle in numpy.arange(0.0, numpy.deg2rad(360.0),
     numpy.deg2rad(1.0)):
                 # Create place location:
                 place = PlaceLocation()

                 place.place_pose.header.stamp = now
                 place.place_pose.header.frame_id =
     self._robot.get_planning_frame()

                 # Set target position:
                 place.place_pose.pose = copy.deepcopy(target)

                 # Generate orientation (wrt Z axis):
                 q = quaternion_from_euler(0.0, 0.0, angle )
                 place.place_pose.pose.orientation = Quaternion(*q)

                 # Generate pre place approach:
                 place.pre_place_approach.desired_distance =
     self._approach_retreat_desired_dist
                 place.pre_place_approach.min_distance =
     self._approach_retreat_min_dist

                 place.pre_place_approach.direction.header.stamp = now
                 place.pre_place_approach.direction.header.frame_id =
     self._robot.get_planning_frame()

                 place.pre_place_approach.direction.vector.x =  0
                 place.pre_place_approach.direction.vector.y =  0
                 place.pre_place_approach.direction.vector.z = 0.2

                 # Generate post place approach:
                 place.post_place_retreat.direction.header.stamp = now
                 place.post_place_retreat.direction.header.frame_id =
     self._robot.get_planning_frame()

                 place.post_place_retreat.desired_distance =
     self._approach_retreat_desired_dist
                 place.post_place_retreat.min_distance =
     self._approach_retreat_min_dist

                 place.post_place_retreat.direction.vector.x = 0
                 place.post_place_retreat.direction.vector.y = 0
                 place.post_place_retreat.direction.vector.z = 0.2

                 # Add place:
                 places.append(place)

             # Publish places (for debugging/visualization purposes):
             self._publish_places(places)

             return places

         def _create_pickup_goal(self, group, target, grasps):
             """
             Create a MoveIt! PickupGoal
             """

             # Create goal:
             goal = PickupGoal()

             goal.group_name  = group
             goal.target_name = target

             goal.possible_grasps.extend(grasps)

             goal.allowed_touch_objects.append(target)
             goal.support_surface_name = self._table_object_name

             # Configure goal planning options:
             goal.allowed_planning_time = 7.0

             goal.planning_options.planning_scene_diff.is_diff = True
             goal.planning_options.planning_scene_diff.robot_state.is_diff
     = True
             goal.planning_options.plan_only = False
             goal.planning_options.replan = True
             goal.planning_options.replan_attempts = 20

             return goal

         def _create_place_goal(self, group, target, places):
             """
             Create a MoveIt! PlaceGoal
             """

             # Create goal:
             goal = PlaceGoal()

             goal.group_name           = group
             goal.attached_object_name = target

             goal.place_locations.extend(places)

             # Configure goal planning options:
             goal.allowed_planning_time = 7.0

             goal.planning_options.planning_scene_diff.is_diff = True
             goal.planning_options.planning_scene_diff.robot_state.is_diff
     = True
             goal.planning_options.plan_only = False
             goal.planning_options.replan = True
             goal.planning_options.replan_attempts = 20

             return goal

         def _pickup(self, group, target, width):
             """
             Pick up a target using the planning group
             """

             # Obtain possible grasps from the grasp generator server:
             grasps = self._generate_grasps(self._pose_coke_can, width)
             # Create and send Pickup goal:
             goal = self._create_pickup_goal(group, target, grasps)

             state = self._pickup_ac.send_goal_and_wait(goal)
             if state != GoalStatus.SUCCEEDED:
                 rospy.logerr('Pick up goal failed!: %s' %
     self._pickup_ac.get_goal_status_text())
                 return None

             result = self._pickup_ac.get_result()

             # Check for error:
             err = result.error_code.val
             if err != MoveItErrorCodes.SUCCESS:
                 rospy.logwarn('Group %s cannot pick up target %s!: %s' %
     (group, target, str(moveit_error_dict[err])))

                 return False

             return True

         def _place(self, group, target, place):
             """
             Place a target using the planning group
             """

             # Obtain possible places:
             places = self._generate_places(place)

             # Create and send Place goal:
             goal = self._create_place_goal(group, target, places)

             state = self._place_ac.send_goal_and_wait(goal)
             if state != GoalStatus.SUCCEEDED:
                 rospy.logerr('Place goal failed!: %s' %
     self._place_ac.get_goal_status_text())
                 return None

             result = self._place_ac.get_result()

             # Check for error:
             err = result.error_code.val
             if err != MoveItErrorCodes.SUCCESS:
                 rospy.logwarn('Group %s cannot place target %s!: %s' %
     (group, target, str(moveit_error_dict[err])))

                 return False
             return True

         def _publish_grasps(self, grasps):
             """
             Publish grasps as poses, using a PoseArray message
             """

             if self._grasps_pub.get_num_connections() > 0:
                 msg = PoseArray()
                 msg.header.frame_id = self._robot.get_planning_frame()
                 msg.header.stamp = rospy.Time.now()

                 for grasp in grasps:
                     p = grasp.grasp_pose.pose

                     msg.poses.append(Pose(p.position, p.orientation))

                 self._grasps_pub.publish(msg)

         def _publish_places(self, places):
             """
             Publish places as poses, using a PoseArray message
             """

             if self._places_pub.get_num_connections() > 0:
                 msg = PoseArray()
                 msg.header.frame_id = self._robot.get_planning_frame()
                 msg.header.stamp = rospy.Time.now()

                 for place in places:
                     msg.poses.append(place.place_pose.pose)

                 self._places_pub.publish(msg)


     def main():
         p = Pick_Place()

         rospy.spin()

     if __name__ == '__main__':
         roscpp_initialize(sys.argv)
         rospy.init_node('pick_and_place')

         main()

         roscpp_shutdown()
```

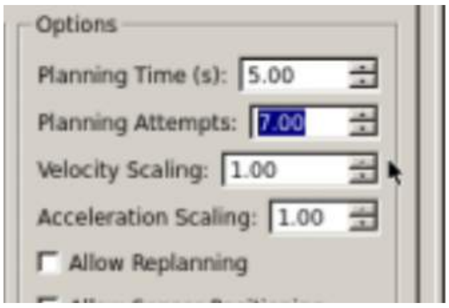
Yes, I know... this is a lot of code! And there's no explanation at all! But, don't worry if you don't understand much of the code at this point. After completing this exercise, we are going to go into detail about the main parts of the code. Trust me!

b) First of all, launch the demo.launch file generated in the MoveIt package in Chapter 3.

   Execute in WebShell #1

```
[ ]: roslaunch fetch_moveit_config demo.launch
```

c) Set the "Planning Attempts" option to something like 6.



Options
Planning Time (s): 5.00
Planning Attempts: 7.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Allow Replanning

Planning Attempts

d) Next, launch the grasping server you created in the previous exercise.

   Execute in WebShell #1

```
[ ]: roslaunch myrobot_grasping grasp_generator_server.launch
```

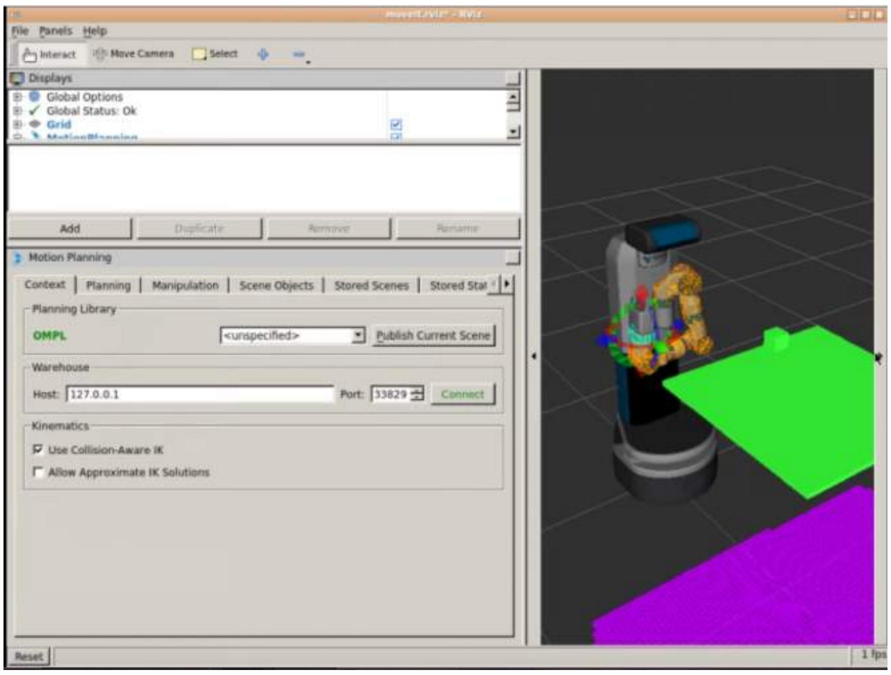
e) Finally, launch this Python script that you've just created, and move to the MoveIt RViz screen in order to visualize what's going on.

   Execute in WebShell #1

```
[ ]: rosrun myrobot_grasping pick_and_place.py
```

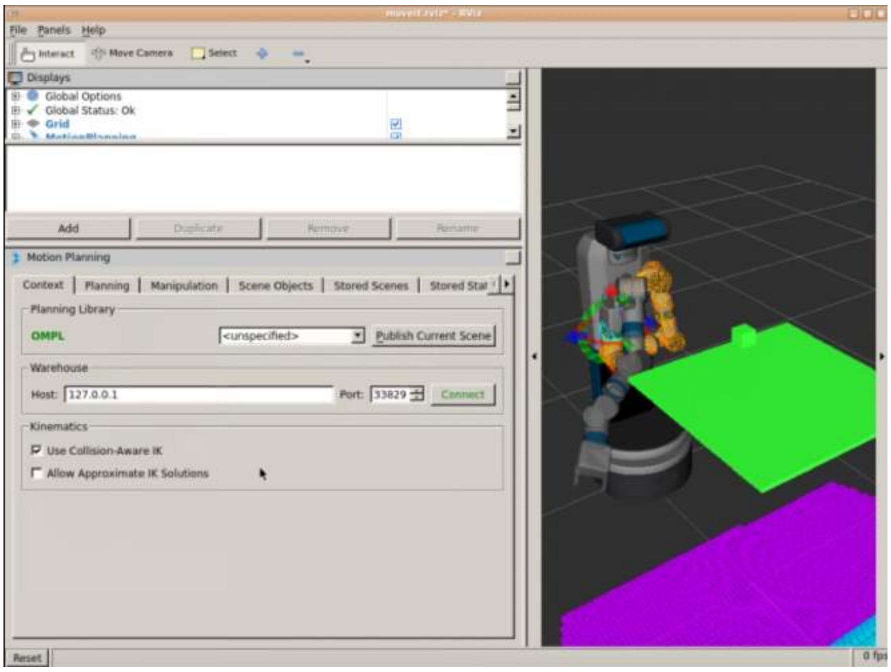
IMPORTANT READ: Since the update of the system to ROS Kinetic, the simulation of the pick and place action in MoveIt may fail. If you try to run it and it fails, just move one and continue following the Notebook. You will do this pick and place action in the next exercise, which is exactly the same as this one but executing it in the Gazebo simulated robot. That exercise should be working fine. Sorry for the inconveniences.

Expected Result for Exercise 6.2
Initial status:



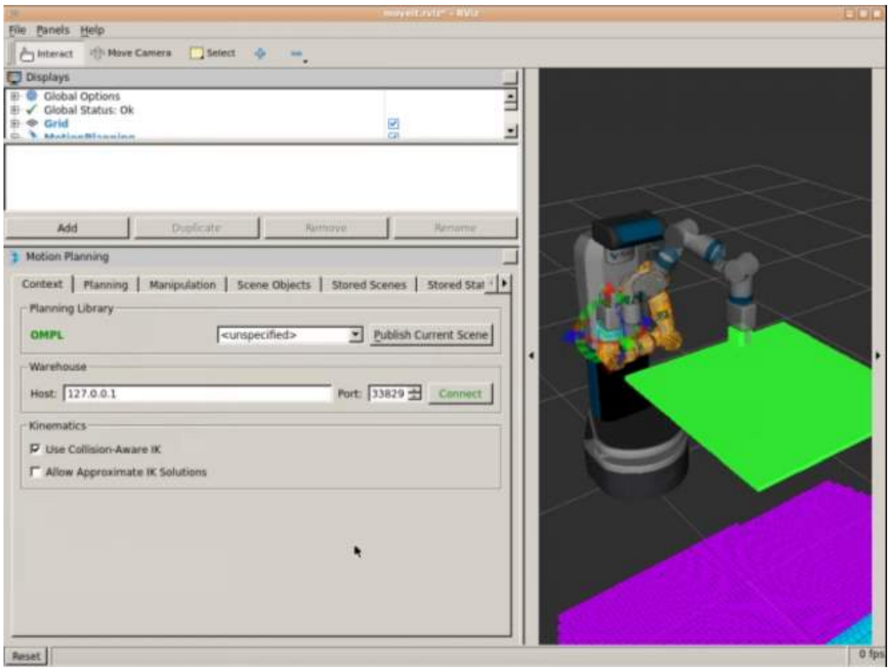
Grasp Object and Table in RViz

Going to grasp pose:



Planning Trajectory

Grasping object:



OMPL <unspecified>

Host: 127.0.0.1 Port: 33829

☑ Use Collision-Aware IK
☐ Allow Approximate IK Solutions

Grasping Object

Several things happened in the previous exercise, so let's go by parts!

- First of all, you saw a couple of green blocks appearing in the RViz screen, right? Can you guess what they are? Well, that were the table and the object to be grasped, of course! Inside the pick_and_place.py script, we have created these 2 objects and added them into the MoveIt Planning Scene. Whenever you add an object to the Planning Scene, it will appear in this way.
- Second, the Pick Action starts. After getting the grasp object position, our node sends this position to the grasp server, which will generate IK and check if there's any valid IK in order to pick the object up. If it finds any feasible IK solution, the arm will begin to execute the specified motions in order to pick the object up.
- After the object is picked by the gripper, the Place Action starts. Again, our node will send a Pose to the grasp server where the arm should place the object. Then, the server will check for a valid IK solution for that specified Pose. If it finds any valid solution, the gripper will move to that position and release the object.

You can have a look at the topics /grasp and /place in order to better see what's going on. Ok, so now you have a better idea of what happened in the previous exercise. Let's take a deeper look into the code in order to see how all of these work.

Explaining the code

First of all, let's check how we created and added both the table and the grasping object.

```
[ ]: def _add_table(self, name):
         p = PoseStamped()
         p.header.frame_id = self._robot.get_planning_frame()
         p.header.stamp = rospy.Time.now()
         #Table position
         p.pose.position.x = 0.45
         p.pose.position.y = 0.0
         p.pose.position.z = 0.22
         q = quaternion_from_euler(0.0, 0.0, numpy.deg2rad(90.0))
         p.pose.orientation = Quaternion(*q)
         # Table size
         self._scene.add_box(name, p, (0.5, 0.4, 0.02))
         return p.pose
```

In this section of the code, what we are doing is creating and adding a table to the Planning Scene. Quite self explanatory, right? We first create a PoseStamped message, and we fill it with the necessary information: the frame_id, the time stamp, and the position and orientation where the table will be placed. Then, we add the table to the scene using the add_box function.

The following is the creation of the grasp object.

```
[ ]: def _add_grasp_block_(self, name):
         p = PoseStamped()
         p.header.frame_id = self._robot.get_planning_frame()
         p.header.stamp = rospy.Time.now()
         p.pose.position.x = 0.62
         p.pose.position.y = 0.21
         p.pose.position.z = 1.07
         q = quaternion_from_euler(0.0, 0.0, 0.0)
         p.pose.orientation = Quaternion(*q)
         # Grasp Object can size
         self._scene.add_box(name, p, (0.03, 0.03, 0.09))
         return p.pose
```

In the above section of the code, what we are doing is the exact same thing as what we did with the table, but for the grasp object. Nothing new here!

After creating the grasp object and the grasp table, we will see how to set the pick position and the place position from the following code snippet. Here, the pose of the grasp object created in the planning scene is retrieved and fed into the place pose in which the Y axis of the place pose is subtracted by 0.06. So, when the pick and place happens, the grasp object will be placed 0.06 meters (6cm) away from the initial pose of the object in the direction of Y.

```
[ ]: # Add table and grap object to the planning scene:
     self._pose_table    = self._add_table(self._table_object_name)
```

```
    self._pose_grasp_obj = self._add_grasp_block_(self._grasp_object_name)
    rospy.sleep(1.0)
    # Define target place pose:
    self._pose_place = Pose()
    self._pose_place.position.x = self._pose_grasp_obj.position.x
    self._pose_place.position.y = self._pose_grasp_obj.position.y - 0.06
    self._pose_place.position.z = self._pose_grasp_obj.position.z
    self._pose_place.orientation = Quaternion(*quaternion_from_euler(0.0,
    0.0, 0.0))
```

The next step is to generate the grasp Pose Array data for visualization, and then send the grasp goal to the grasp server. If there is a grasp sequence, it will be published; otherwise, it will show as an error.

```
[ ]: def _generate_grasps(self, pose, width):
         # Create goal:
         goal = GenerateGraspsGoal()
         goal.pose  = pose
         goal.width = width
         ......................
         .......................
         state = self._grasps_ac.send_goal_and_wait(goal)
         if state != GoalStatus.SUCCEEDED:
             rospy.logerr('Grasp goal failed!: %s' %
     self._grasps_ac.get_goal_status_text())
             return None
         grasps = self._grasps_ac.get_result().grasps
         # Publish grasps (for debugging/visualization purposes):
         self._publish_grasps(grasps)
         return grasps
```

This function will create a Pose Array data for the pose of the place.

```
[ ]: def _generate_places(self, target):
         # Generate places:
         places = []
         now = rospy.Time.now()
         for angle in numpy.arange(0.0, numpy.deg2rad(360.0),
     numpy.deg2rad(1.0)):
             # Create place location:
             place = PlaceLocation()
             ............................................
             .......................................
             # Add place:
             places.append(place)
```

```
        # Publish places (for debugging/visualization purposes):
        self._publish_places(places)
```

The next function is **_create_pickup_goal()**, which will create a goal for picking up the grasping object. This goal has to be sent into MoveIt!.

```
[ ]: def _create_pickup_goal(self, group, target, grasps):
         """
         Create a MoveIt! PickupGoal
         """

         # Create goal:
         goal = PickupGoal()

         goal.group_name  = group
         goal.target_name = target

         goal.possible_grasps.extend(grasps)

         goal.allowed_touch_objects.append(target)

         goal.support_surface_name = self._table_object_name

         # Configure goal planning options:
         goal.allowed_planning_time = 7.0

         goal.planning_options.planning_scene_diff.is_diff = True
         goal.planning_options.planning_scene_diff.robot_state.is_diff =
     True
         goal.planning_options.plan_only = False
         goal.planning_options.replan = True
         goal.planning_options.replan_attempts = 20

         return goal
```

Also, there is the **_create_place_goal()** function, which creates a place goal for MoveIt.

```
[ ]: def _create_place_goal(self, group, target, places):
         """
         Create a MoveIt! PlaceGoal
         """

         # Create goal:
         goal = PlaceGoal()
```

```
        goal.group_name             = group
        goal.attached_object_name = target

        goal.place_locations.extend(places)

        # Configure goal planning options:
        goal.allowed_planning_time = 7.0

        goal.planning_options.planning_scene_diff.is_diff = True
        goal.planning_options.planning_scene_diff.robot_state.is_diff =
    True
        goal.planning_options.plan_only = False
        goal.planning_options.replan = True
        goal.planning_options.replan_attempts = 20

        return goal
```

The important functions that are performing picking and placing are given below. These functions will generate a pick and place sequence, which will be sent to MoveIt and print the results of the motion planning, whether it is successful or not:

```
[ ]: def _pickup(self, group, target, width):
        """
        Pick up a target using the planning group
        """

        # Obtain possible grasps from the grasp generator server:
        grasps = self._generate_grasps(self._pose_coke_can, width)

        # Create and send Pickup goal:
        goal = self._create_pickup_goal(group, target, grasps)

        state = self._pickup_ac.send_goal_and_wait(goal)
        if state != GoalStatus.SUCCEEDED:
            rospy.logerr('Pick up goal failed!: %s' %
    self._pickup_ac.get_goal_status_text())
            return None

        result = self._pickup_ac.get_result()

        # Check for error:
        err = result.error_code.val
        if err != MoveItErrorCodes.SUCCESS:
            rospy.logwarn('Group %s cannot pick up target %s!: %s' %
    (group, target, str(moveit_error_dict[err])))

            return False
        return True
```

```
[ ]: def _place(self, group, target, place):
         """
         Place a target using the planning group
         """

         # Obtain possible places:
         places = self._generate_places(place)

         # Create and send Place goal:
         goal = self._create_place_goal(group, target, places)

         state = self._place_ac.send_goal_and_wait(goal)
         if state != GoalStatus.SUCCEEDED:
             rospy.logerr('Place goal failed!: %s' %
     self._place_ac.get_goal_status_text())
             return None

         result = self._place_ac.get_result()

         # Check for error:
         err = result.error_code.val
         if err != MoveItErrorCodes.SUCCESS:
             rospy.logwarn('Group %s cannot place target %s!: %s' % (group,
     target, str(moveit_error_dict[err])))

             return False

         return True
```

Grasping in the Real Robot

So, now you've seen how to perform Grasping in MoveIt. But. . . what about applying this to the real robot? Well, actually, it's quite easy at this point.

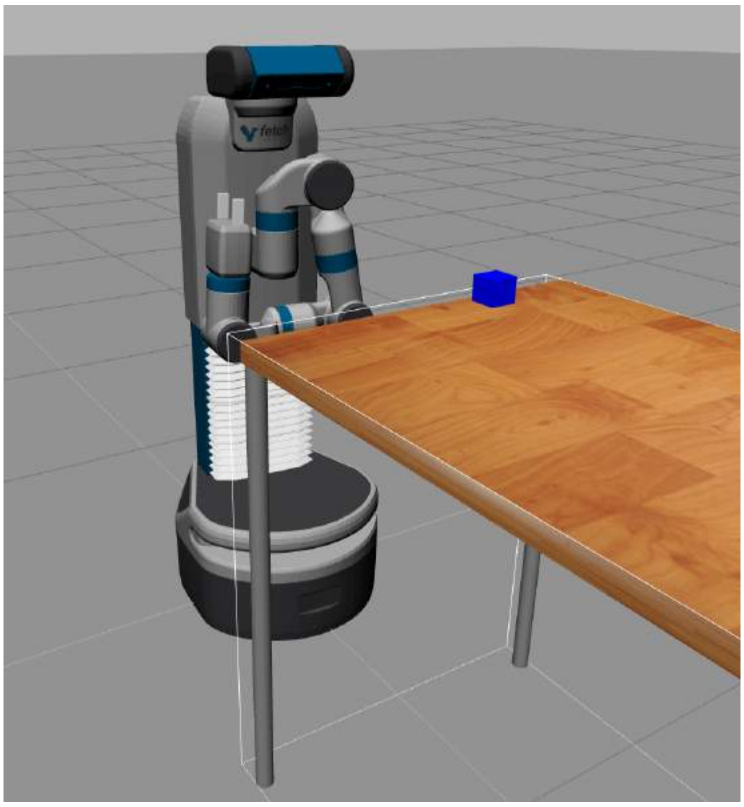
Exercise 6.3
a) First of all, execute the following commands in order to spawn a table and a Grasping object into the simulation.

Execute in WebShell #1

```
[ ]: rosrun gazebo_ros spawn_model -database table -gazebo -model table -x
     1.30 -y 0 -z 0
```

```
[ ]: rosrun gazebo_ros spawn_model -database demo_cube -gazebo -model
     grasp_cube -x 0.65 -y 0.15 -z 1.07
```

You should end up with something like this.

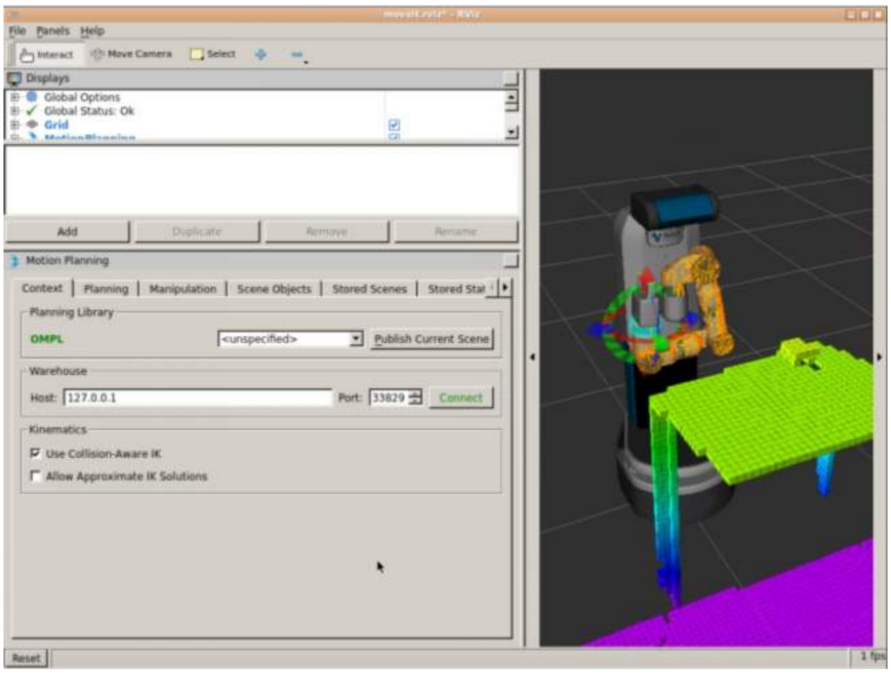


Grasping Object and Table in simulation

b) Launch MoveIt with Perception.

Execute in WebShell #1

```
[ ]: roslaunch fetch_moveit_config fetch_planning_execution.launch
```



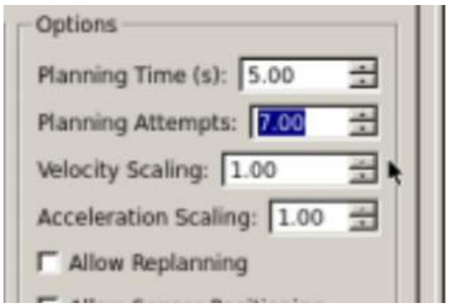
Planning Library OMPL <unspecified> Publish Current Scene

Warehouse Host: 127.0.0.1 Port: 33829 Connect

Kinematics ☑ Use Collision-Aware IK ☐ Allow Approximate IK Solutions

Perception in RViz

c) Set the “Planning Attempts” option to something like 6.



Options

Planning Time (s): 5.00

Planning Attempts: 7.00

Velocity Scaling: 1.00

Acceleration Scaling: 1.00

☐ Allow Replanning

Planning Attempts

d) Launch the grasping server you created in the previous exercise.

Execute in WebShell #2

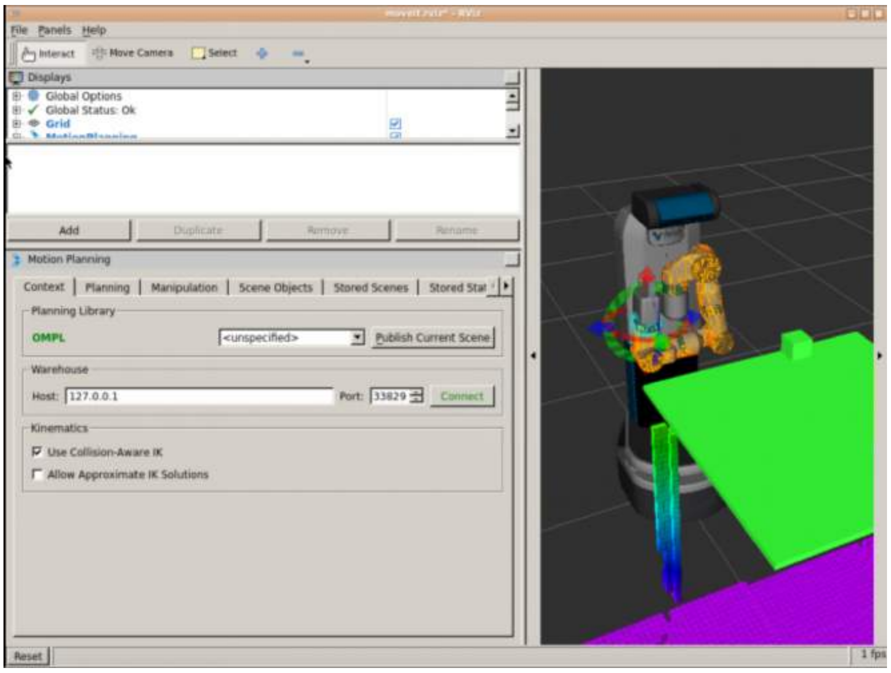
```
[ ]: roslaunch myrobot_grasping grasp_generator_server.launch
```

e) Launch this Python script.

Execute in WebShell #3

```
[ ]: rosrun myrobot_grasping pick_and_place.py
```

Expected Result for Exercise 6.3
RViz view:



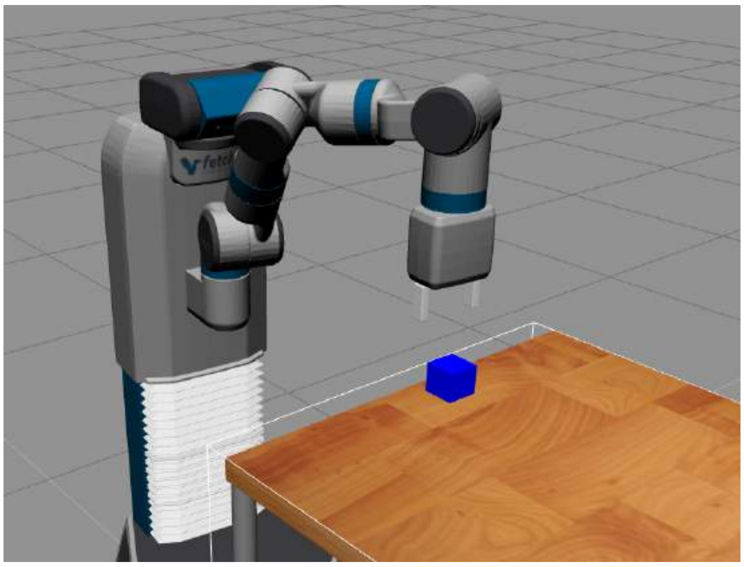
Planning Library: OMPL <unspecified>

Warehouse: Host: 127.0.0.1 Port: 33829

Kinematics:
- [✓] Use Collision-Aware IK
- [ ] Allow Approximate IK Solutions

Scene in Rviz

Simulation view:



Grasping in the simulation

And that's all! You've finished the course! Well. . . almost finished. If you go to the next unit, you'll be able to do a project where you will test everything you've learned during the course.

##
Solutions

Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions for the Unit 5: Unit 5 Solutions

Unit 6. Course Project

ROS Manipulation in 5 Days

Unit 6: Project

Run on ROSDS

- ROSject Link: http://bit.ly/2naklom
- Package Name: **rb1_tc**
- Launch File: **main.launch**

SUMMARY
Estimated time of completion: 2h
This last unit is meant to put together all of the knowledge that you've gained during this course. You will do this by doing a project, which will be based on the RB1 simulation. Using this simulation, you will have to practice everything you've learned during the course: build and configure a MoveIt Package, connect this package to the running simulation, perform Motion Planning using Python, and finally, perform Grasping.

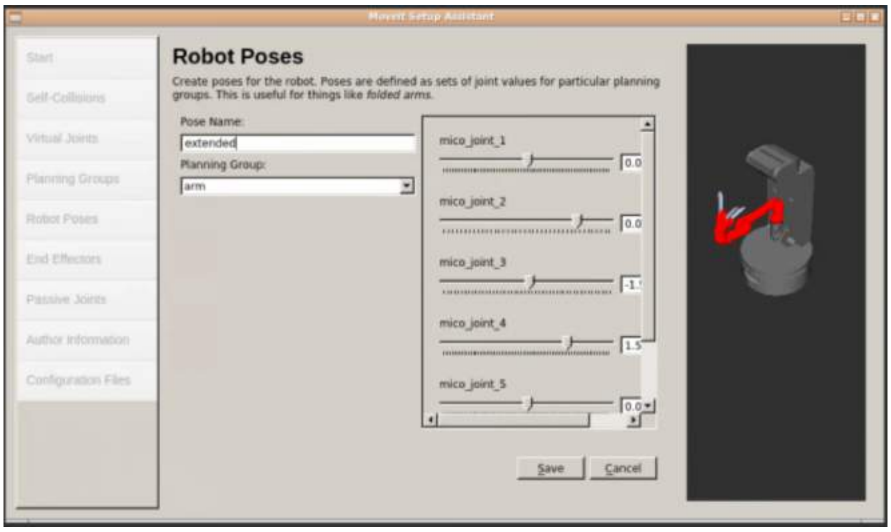
The Project will be divided in 4 parts. In order to complete it, just follow the steps tht will be described below. Are you ready? Then let's go!

1. Build the MoveIt package

First of all, you will need to create a MoveIt package for your manipulator. To do this, you can just follow the steps described in Chapter 2 of this course, bearing in mind the differences between both robots.

You should create 3 predefined Poses for the robot. For instance, they could be these 3:

Pose 1:



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 *folded arms*.

Pose Name: extended

Planning Group: arm

mico_joint_1 0.0

mico_joint_2 0.0

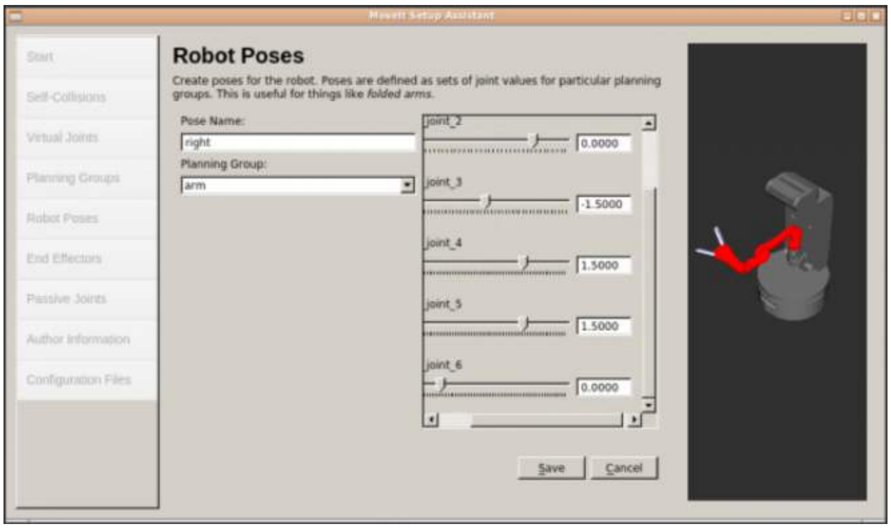
mico_joint_3 -1.

mico_joint_4 1.5

mico_joint_5 0.0

Pose 1

Pose 2:



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 *folded arms*.

Pose Name: right

Planning Group: arm

joint_2 0.0000

joint_3 -1.5000

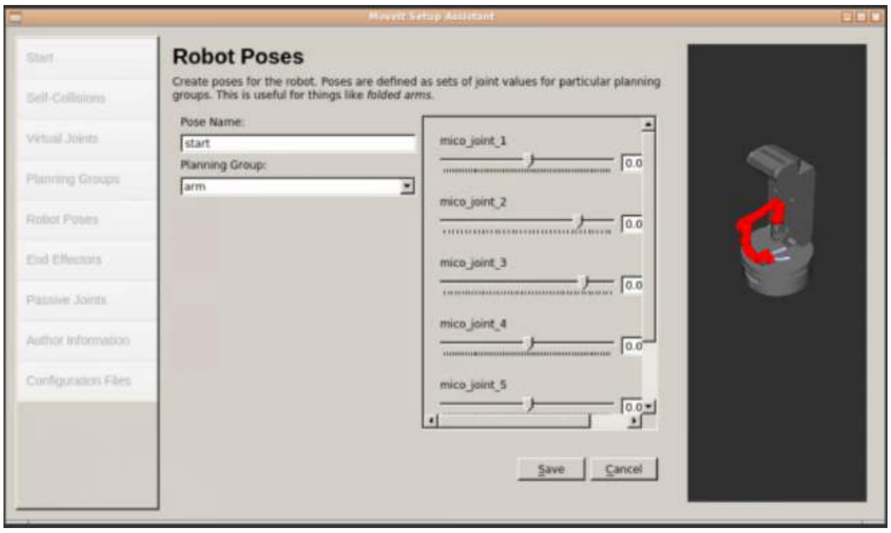
joint_4 1.5000

joint_5 1.5000

joint_6 0.0000

Pose 2

Pose 3:



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 *folded arms*.

Pose Name: start

Planning Group: arm

mico_joint_1 0.0

mico_joint_2 0.0

mico_joint_3 0.0

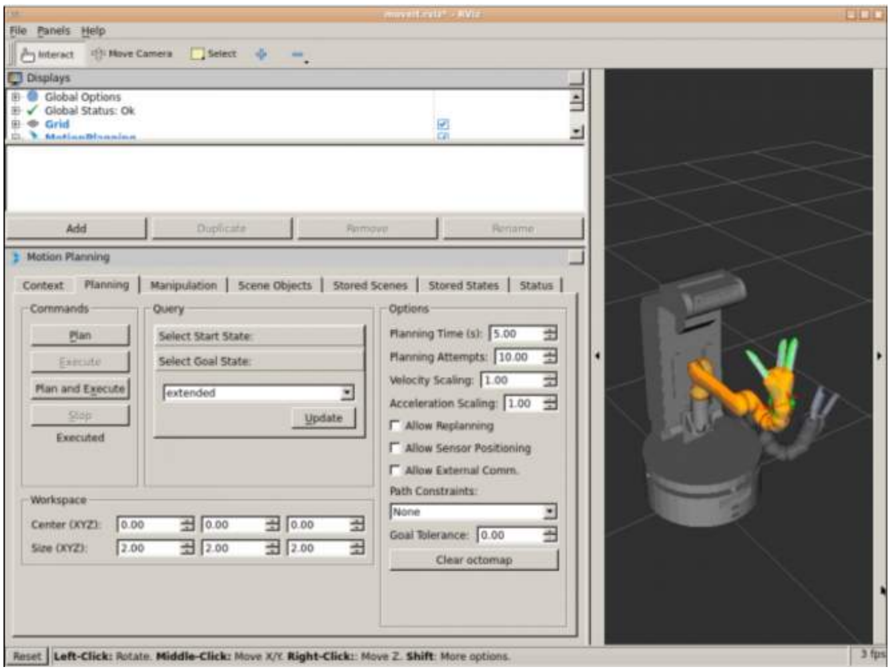
mico_joint_4 0.0

mico_joint_5 0.0

Pose 3

Once the package is created, test that it works properly by launching the demo.launch file, and Planning some trajectories.

At the end, you should be able to see something like this:



Select Goal State: extended

Planning Time (s): 5.00

Planning Attempts: 10.00

Velocity Scaling: 1.00

Acceleration Scaling: 1.00

Path Constraints: None

Goal Tolerance: 0.00

Center (XYZ): 0.00 0.00 0.00

Size (XYZ): 2.00 2.00 2.00

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.

RB1 Robot in MoveIt

2. Connect the MoveIt package with the simulation

Once you can successfully perform Motion Planning in MoveIt, it's time to connect it to the simulation in order to be able to move the "real" robot. You can follow the last part of the second chapter to complete this.

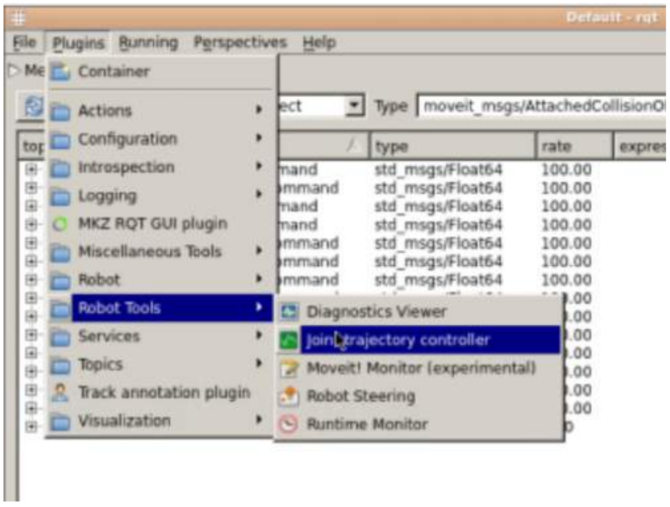
Before trying to execute any trajectory, though, you should follow the next steps:

a) First of all, execute the following command in a webshell:

```
[ ]: rqt
```

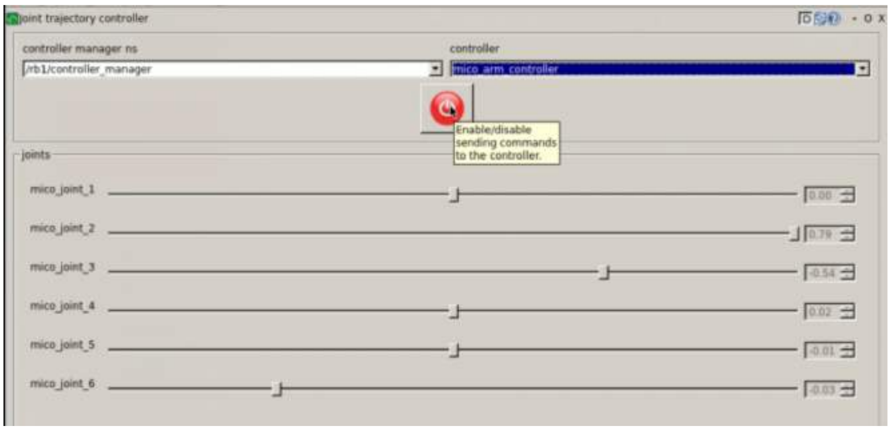
rqt GUI

b) Now, open the Plugins menu, select the Robot Tools option, and then the Joint Trajectory Controller option.



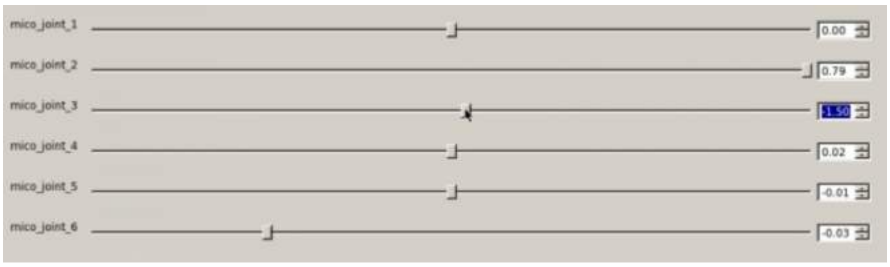
Joint Trajectory Controller

c) Now, let's connect the mico_arm_controller. In the controller manager ns menu, select /rb1/controller_manager. And in the controller menu, select the mico_arm_controller option. Finally, click on the Enable button.



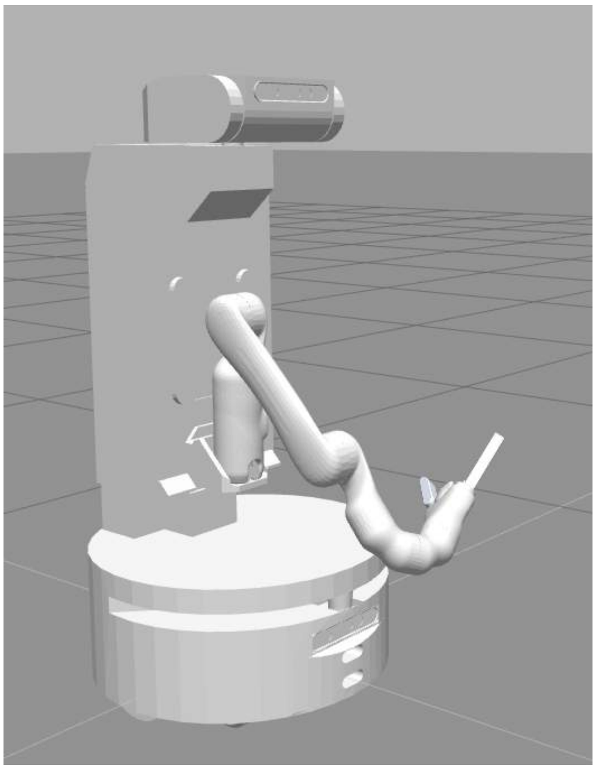
Enable Controller

d) Finally, set the mico_joint_3 to “-1.5.”



Move Joints

At the end, you should have the RB1 robot like this:



RB1 Moved

Now, you can begin executing your trajectories with the real robot!

3. Python Script

Once everything is working, and you are able to move the robot in the simulation through MoveIt, it's time to do it with code! In this last step of the project, you will have to create a Python Script that does the following:

- First, it moves the robot to one of the predefined Poses you set in Step 2.
- Then, it moves the robot to the second predefined Pose.
- After this, it goes back to the previous Pose.
- Finally, it returns to it's initial Pose.

4. Add Perception to the MoveIt package

Now, you will have to add Perception to your MoveIt package, just as you learned to do in Chapter 4. The steps you should follow are the following:

- Create an object to spawn into the simulation.

- Make the proper modification to your MoveIt package in order to add Perception.
- Launch the MoveIt package and check that you are able to visualize the environment.

Here is an example of what you should see.

5. Grasping

Finally, you will have to spawn into the simulation a table with a cube on its surface. Then, you will have to create the proper Python script to do the following:

- Approach the cube and grasps it.
- Pick up the cube and elevate it a little bit.
- Finally, place the cube in another position on the table.

Here you can see an example of what your script should do.

5. Congratulations! You completed the Course!

So, that's it! You have completed 100% of the course! But... what could you do now? If you want to know what you can do after finishing this course, have a look at the next unit!

##
Solutions
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions for the Project: Project Solutions

```
[ ]:
```

Final Recommendations

I'm finished, now what?

ROS Development Studio (ROSDS)

ROSDS logo

ROSDS is the The Construct web based tool to program ROS robots online. It requires no installation in your computer. Hence, you can use any type of computer to work on it (Windows, Linux or Mac). Additionally, free accounts are available. **Create a free ROSDS account here**: http://rosds.online
You can use any of the many ROSjects available in order to apply all the things you've learned during the Course. You just need to paste the ROSject link to your browser's URL, and you will automatically have the simulation prepared in your ROSDS workspace.

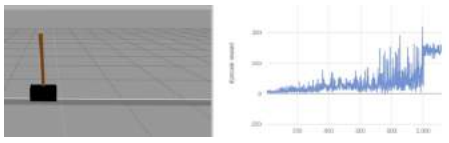
Down below you can check some examples of the Public Rosjects we provide:

ARIAC Competition

Run on ROSDS

- ROSject Link: https://bit.ly/2t2px0t

Cartpole Reinforcement Learning



Run on ROSDS

- ROSject Link: https://bit.ly/2t2uGWr

ARIAC Competition

Run on ROSDS

- ROSject Link: https://bit.ly/2Tt4lw8

Want to learn more?

robotignite ACADEMY

Robot Ignite Logo

Once you have finished the course, you can still learn a lot of interesting ROS subjects.

- Take more advanced courses that we offer at the Robot Ignite Academy, like Perception or Navigation. Access the Academy here: http://www.robotigniteacademy.com
- Or, you can go to the ROS Wiki and check the official documentation of ROS, now with new eyes.

Thank You and hope to see you soon!