Motoman manual

Author:
Thibault Barbié

Version:
0.0.1

March 29, 2017

Contents

Chapter 1

Introduction

The motoman industrial robot is a complicated system that needs to be handled carefully. This manual was written to help to understand how to do basic operations such as connecting the robot to its rviz visualization, using motion planners and using the kinects.

The motoman robot is a very expensive system, if you are scared to do something wrong that could break it then please ask the more experienced lab members.

This is the original manual. A japanese version of this manual will be soon written but it will be less detailed. If you have any questions please check the english version (this one).

Chapter 2

Installation and first script

In this chapter we will explain how to install the motoman project on your computer and make you write a script that use a motion planner to find a plan for the robot. This chapter is essentially a chain of instructions, there are few details about the files you will manipulate. The next chapter will be more in depth.

2.1 Initialization

First thing to do is to create a ROS workspace where you will work. So go to where you want to create your workspace and then create the workspace with a *src* folder inside. Then go inside the *src* folder and initialize the catkin workspace.

```
cd /where/you/want/your/workspace
mkdir -p my_workspace/src
cd my_workspace/src
catkin_init_workspace
```

Then download the motoman project repository. To do it you need to have the git software in your computer. To install git just type the following command.

```
sudo apt-get install git
```

Then you need to download the repository. The *clone* command after the *git* command means you want to take the repository data and copy them into your current folder. So first go where your ros workspace and clone the github repository inside the *src* folder.

```
cd my_workspace/src
git clone https://github.com/Nishida-Lab/motoman_project.git
```

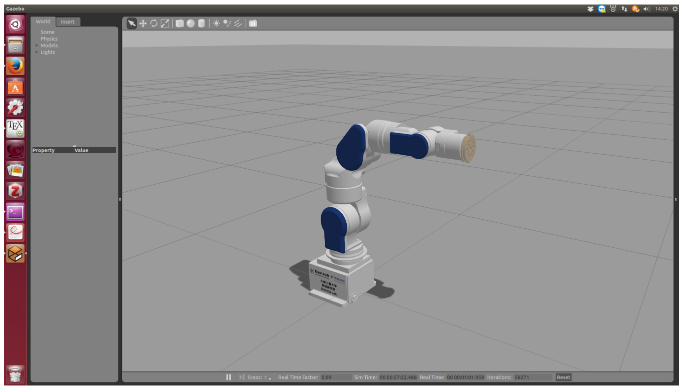


Figure 2.1: That is what you should see if the installation has been rightly completed.

Then normaly a *motoman_project* file would have been created in your *src* folder. The next step is to actually compile the project. To do this you need to use the *catkin_make* command in the root of your workspace. But before that you need to be sure to have all the dependencies.

```
cd my_workspace
wstool init src src/motoman_project/dependencies.rosinstall
sudo apt-get install ros-indigo-industrial-msgs
sudo apt-get install ros-indigo-industrial-robot-simulator
sudo apt-get install ros-indigo-industrial-robot-client
sudo apt-get install ros-indigo-ros-controllers
rosdep install -i --from-paths src
catkin_make
```

After compiling everything (it could take some time!) you will be able to use the project. A simple test is to launch one of the launch file of the project. First you need to source the workspace to be sure that you can use the ROS command associated to your project.

```
cd workspace
source devel/setup.bash
```

Then launch the empty environment with motoman inside by the following command.

```
cd my_workspace
source devel/setup.bash
roslaunch motoman_gazebo sia5_empty_world.launch
```

You should normally have the gazebo software begin to run and you can soon see the motoman robot sia5 inside your screen like in Figure 2.1.

2.2 Finding a motion plan

In this section we will create a script that connect to the robot and use a motion planner to find a plan between a start position and a goal position. There are many things to do before being able to do it but if you follow the steps it should not be difficult. This section focus on writing the script rather than understanding everything. The next chapters will give more details about it.

To begin everything we need to create a ros package where our script will be written. You normally have one metapackage in your *src* folder named *motoman_project*. It is really easy to create a new package in ros with the catkin command. You can name it anything you want, in this manual we will call it *motion_planning*. To make our script able to run we will need to initialize gazebo and rviz. For this reason you will need to launch gazebo and rviz every time you will want to use your script. As it is tiring it is easier to just create a launch file that will be launch before the script and that will automatically call the gazebo and rviz part by itself.

```
cd workspace/src
catkin_create_pkg motion_planning roscpp
cd motion_planning
mkdir launch
cd launch
touch initialization.launch
```

We use the *roscpp* argument because our package will need it to create a ros node. You can create the following launch file in the launch folder, we will describe it more later.

Code 2.1: initialization.launch

```
<launch>
  <arg name="model" default="$(find
      motoman_description)/robots/sia5/sia5.urdf.xacro"/>
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="false"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find
        motoman_gazebo)/worlds/sia5/sia5_empty.world"/>
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>
  <param name="robot_description" command="$(find xacro)/xacro.py '$(arg
      model)'" />

  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
      respawn="false" output="screen"
     args="-urdf -model sia5 -param robot_description"/>

  <include file="$(find
      motoman_control)/launch/sia5/sia5_sim_control.launch"/>


  <include file="$(find
      motoman_sia5_moveit_config)/launch/moveit_planning_execution.launch">
    <arg name="load_robot_description" value="true"/>
    <arg name="urdf_model" value="$(find
        motoman_description)/robots/sia5/sia5.urdf.xacro"/>
    <arg name="srdf_model" value="$(find
        motoman_moveit)/config/sia5/sia5.srdf"/>
    <arg name="joint_limits_config" value="$(find
        motoman_moveit)/config/sia5/joint_limits.yaml"/>
    <arg name="kinematics_config" value="$(find
        motoman_moveit)/config/sia5/kinematics.yaml"/>
    <arg name="controllers_config" value="$(find
        motoman_moveit)/config/sia5/controllers.yaml"/>
    <arg name="use_depth_sensor" value="false"/>
    <arg name="rviz_config" value="$(find
        motoman_moveit)/launch/rviz/moveit_sia5.rviz"/>
    <!-- Configuration planning library -->
    <arg name="ompl_config" default="$(find
        motoman_sia5_moveit_config)/config/ompl_planning.yaml"/>
    <!-- Choose planner [ompl|chomp|stomp] -->
    <arg name="planning_config" default="ompl"/>
    <!-- If you choose ompl, "use_ompl" is true. -->
    <arg name="use_ompl" default="true"/>
    <!-- If you choose stomp, "use_stomp" is true. -->
    <arg name="use_stomp" default="false"/>
  </include>
</launch>
```

In the *src* folder we can write our script to move the robot. First we create the file (we named it *moving.cpp* but every name is ok, just change the CMakeLists file accordingly).

```
cd workspace/src/motion_planning/src
touch moving.cpp
```

Then just write the following code inside the *moving.cpp* file. This script shows how to ask moveit to find a plan from the current position to any goal position you define. However it may be possible that no plan will be found. In this situation the workspace is empty so it should be quite easy to find a solution but when a lot of obstacles are populating the environment then it becomes a difficult task to find a collision free trajectory for the robot.

Code 2.2: moving.cpp

```
#include <moveit/move_group_interface/move_group.h>

int main(int argc, char** argv)
{
  // Initialization of the ROS node
  ros::init(argc, argv, "moving_the_robot");

  // Initialization of moveit
  moveit::planning_interface::MoveGroup group("arm");

  // Setting the start position
  group.setStartState(*group.getCurrentState());

  // Setting the goal position
  std::map<std::string, double> joints;

  joints["joint_s"] = -0.8;
  joints["joint_l"] = 0.2;
  joints["joint_e"] = 0.0;
  joints["joint_u"] = -0.4;
  joints["joint_r"] = 0.35;
  joints["joint_b"] = 0.6;
  joints["joint_t"] = 0.4;

  group.setJointValueTarget(joints);

  // Running the moveit planning
  moveit::planning_interface::MoveGroup::Plan result_plan;
  group.plan(result_plan);

  return 0;
}
```

After having wrote the script we need to compile it. To do it we need to modify two files : the CMakeLists.txt and package.xml. These two files have already been created when you created the package with the catkin command, so you just have to replace their contents with the following files.

Code 2.3: CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(motion_plannign)
set(CMAKE_CXX_FLAGS "-std=c++0x ${CMAKE_CXX_FLAGS}")
```

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  moveit_msgs
  moveit_commander
  moveit_core
  moveit_ros_planning
  moveit_ros_planning_interface
  pluginlib
  cmake_modules)
find_package(Boost REQUIRED system filesystem date_time thread)
find_package(Eigen REQUIRED)

include_directories(SYSTEM ${Boost_INCLUDE_DIR} ${EIGEN_INCLUDE_DIRS})
include_directories(include ${catkin_INCLUDE_DIRS})
link_directories(${catkin_LIBRARY_DIRS})

catkin_package(
  CATKIN_DEPENDS
    moveit_core
    moveit_ros_planning_interface
)

add_executable(motion_planning src/moving.cpp)
target_link_libraries(moving_robot ${catkin_LIBRARIES}
     ${Boost_LIBRARIES})
```

Code 2.4: package.xml

```
<?xml version="1.0"?>
<package>
  <name>motion_planning</name>
  <version>0.0.0</version>
  <description>A package to use motion planning</description>

  <maintainer email="your@mail.mail">your_name</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>pluginlib</build_depend>
  <build_depend>moveit_core</build_depend>
  <build_depend>moveit_ros_planning_interface</build_depend>
  <build_depend>moveit_ros_perception</build_depend>
  <build_depend>cmake_modules</build_depend>

  <run_depend>pluginlib</run_depend>
  <run_depend>moveit_core</run_depend>
```

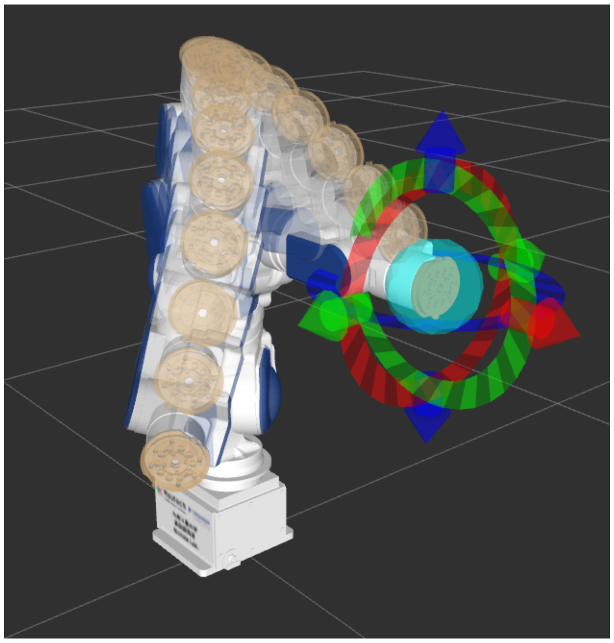


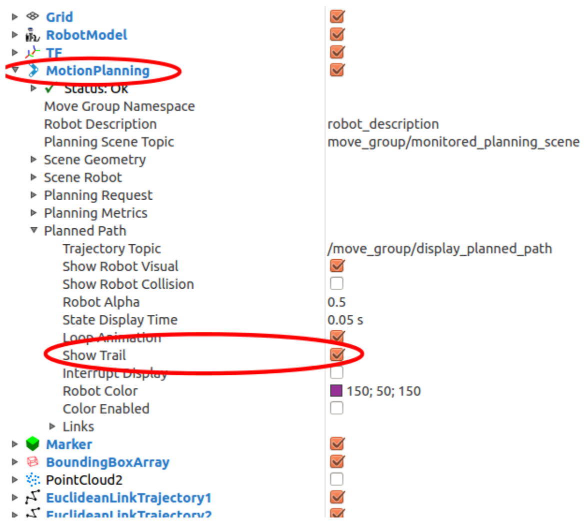
Figure 2.2: Yo

```
  <run_depend>moveit_fake_controller_manager</run_depend>
  <run_depend>moveit_ros_planning_interface</run_depend>
  <run_depend>moveit_ros_perception</run_depend>
  <run_depend>roscpp</run_depend>

</package>
```

So now everything have been done to use the script you wrote. First compile everything. Every time you will modify your C++ files you will need to compile again all your workspace. Once compile you will need to first launch the launch file that will initialize gazebo and rviz, it will also load the ompl library. When the launch file has been ran you can then run your script.

```
cd workspace
catkin_make
source devel/setup.bash
roslaunch motion_planning initialization.launch
rosrun motion_planning motion_planning
```

The result of this script could be seen in Figure 2.2. The figure shows a trail of the movement. You can activate it by clicking on the left panel to the *motion planning* line and then clicking on the *Show trail* box as it can be seen in the Figure 2.3.



Grid	✓
RobotModel	✓
TF	✓
MotionPlanning	✓
Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	move_group/monitored_planning_scene
Scene Geometry	
Scene Robot	
Planning Request	
Planning Metrics	
Planned Path	
Trajectory Topic	/move_group/display_planned_path
Show Robot Visual	✓
Show Robot Collision	☐
Robot Alpha	0.5
State Display Time	0.05 s
Loop Animation	✓
Show Trail	✓
Interrupt Display	☐
Robot Color	🟪 150; 50; 150
Color Enabled	☐
Links	
Marker	✓
BoundingBoxArray	✓
PointCloud2	☐
EuclideanLinkTrajectory1	✓
EuclideanLinkTrajectory2	✓

Figure 2.3: Yo

Chapter 3

Motion planning

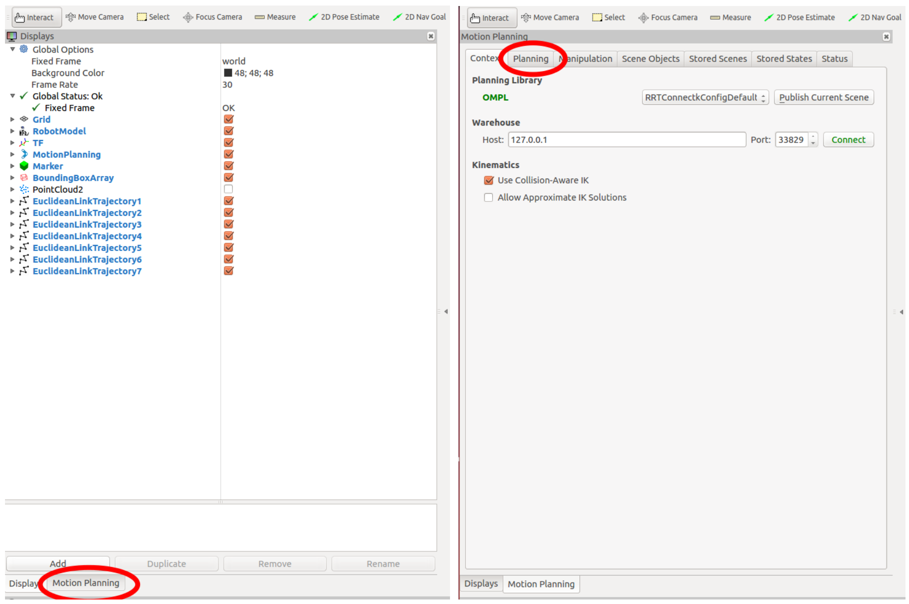
This chapter will focus on the motion planning part. You will learn how to select the motion planner you want (RRT, RRT*, STOMP, CHOMP...), how to create a planning problem and how to solve it.

3.1 GUI utilisation

We can directly use the GUI to solve some motion planning problems. Indeed, moveit has an interface in rviz. Using the GUI will be usefull when you want to quickly test your robot motion or a planning problem. It can also show how long it will last for the motion planner you selected to find a motion plan. However, when you will need to find precise plan (with exact coordinate for joints) or when you will want to generate a lot of trajectories it will be far easier to use scripts.

Using the GUI is really simple. First go to the *motion planning* tab and click on the *planning* tab after (figure 3.1). In this tab you can select your start state and your goal state. Initially the start state should be initialized with the current position of your robot which should be its home position. However, if it not the case you can initialize it yourself by clicking on the *select start state* tab and clicking on the *update* button (the current option should be selected) as shown in figure 3.2. You can see that you can choose other ways to initialize the start goal. For instance you can decide to select a random valid position or pre-determined position. The pre-determined position are generated with the moveit initialization procedure. If you use the motoman project it should have only two pre-determined position : the home position and the up position. As for the goal position, you can either do the same as the start position (selecting random position or pre-determined ones) or just use your mouse and move the robot through the light blue ball inside the gui (see figure 3.3).

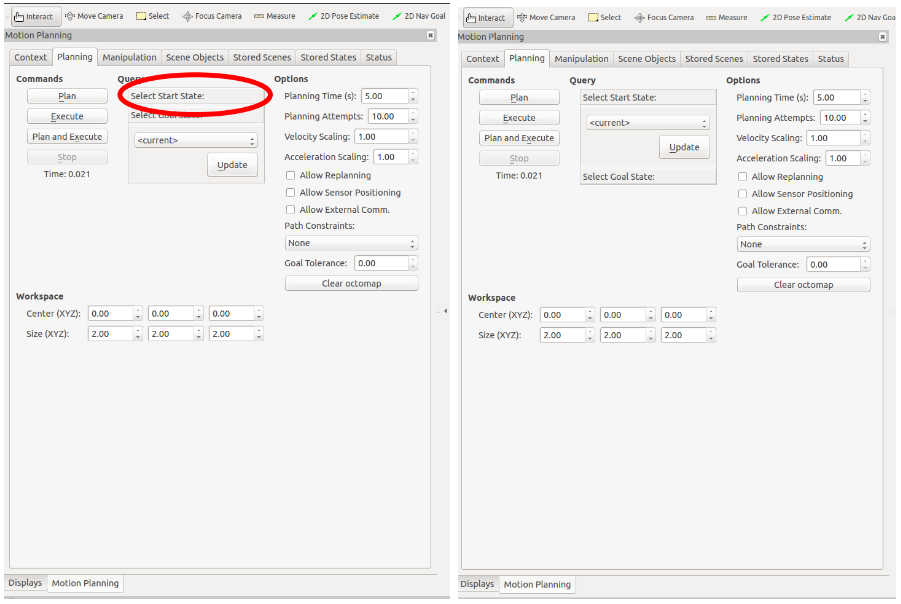
When you have chosen your start and goal position you just need to click on the *plan* button to try to find a motion plan. If you click on *plan and execute* you will see your robot moving after having successfully found a plan.



Global Options
Fixed Frame world
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Fixed Frame OK
Grid ✓
RobotModel ✓
TF ✓
MotionPlanning ✓
Marker ✓
BoundingBoxArray ✓
PointCloud2
EuclideanLinkTrajectory1 ✓
EuclideanLinkTrajectory2 ✓
EuclideanLinkTrajectory3 ✓
EuclideanLinkTrajectory4 ✓
EuclideanLinkTrajectory5 ✓
EuclideanLinkTrajectory6 ✓
EuclideanLinkTrajectory7 ✓

Planning Library
OMPL RRTConnectkConfigDefault
Warehouse
Host: 127.0.0.1 Port: 33829
Kinematics
✓ Use Collision-Aware IK
Allow Approximate IK Solutions

Figure 3.1: Clicking on *motion planning*, then *planning*.



Time: 0.021
Query
Select Start State: <current>
Select Goal State:
Options
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

Figure 3.2: Clicking on *select start state* and then the *update* button.

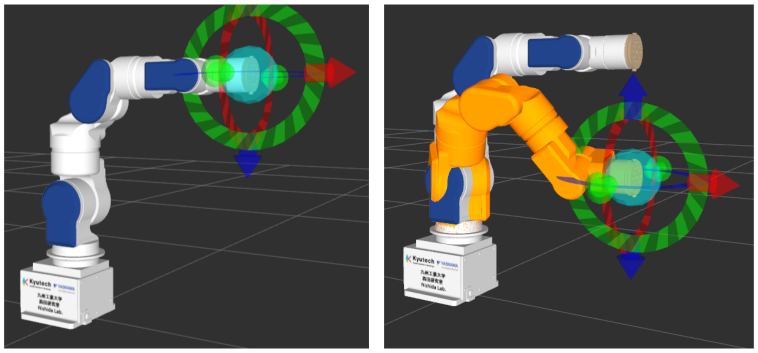


Figure 3.3: Changing the goal position by moving the end effector ball through the GUI.

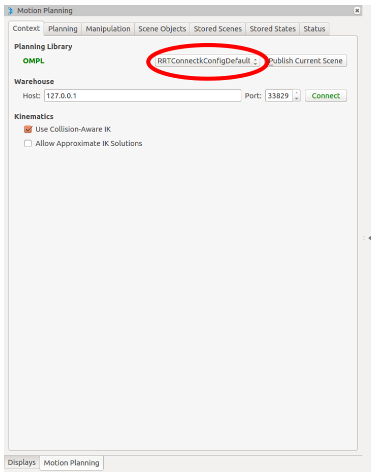
If you are using the OMPL library you may want to change the motion planner, just click on the tab in the *Planning Library* section (see figure 3.4).

3.2 Simple script

In this section we review the script we used during chapter 2 to plan a motion between the home position to a specific position. We will describe the different parts of the code and in the following section we will build on this simple script. The script we used is the following one.

Code 3.1: moving.cpp

```
#include <moveit/move_group_interface/move_group.h>

int main(int argc, char** argv)
{
  // Initialization of the ROS node
  ros::init(argc, argv, "moving_the_robot");

  // Initialization of moveit
  moveit::planning_interface::MoveGroup group("arm");

  // Setting the start position
  group.setStartState(*group.getCurrentState());

  // Setting the goal position
  std::map<std::string, double> joints;

  joints["joint_s"] = -0.8;
  joints["joint_l"] = 0.2;
  joints["joint_e"] = 0.0;
  joints["joint_u"] = -0.4;
```



Planning Library
OMPL RRTConnectkConfigDefault
Warehouse
Host: 127.0.0.1 Port: 33829
Kinematics
[✓] Use Collision-Aware IK
[ ] Allow Approximate IK Solutions

Figure 3.4: Changing the goal position by moving the end effector ball through the GUI.

```
  joints["joint_r"] = 0.35;
  joints["joint_b"] = 0.6;
  joints["joint_t"] = 0.4;

  group.setJointValueTarget(joints);

  // Running the moveit planning
  moveit::planning_interface::MoveGroup::Plan result_plan;
  group.plan(result_plan);

  return 0;
}
```

The first important line is the creation of the object *group*. Moveit will do the interface between the motion planners and the robot but it needs to be initialized to know what robot (or what part of the robot) to work on. In our case the whole parts of the robot will be used and it has been named *arm* during the moveit initialization procedure, that is why we put it as an argument. Then we need to define the start and goal position for our planning problem. In this simple script we define the start position as the current position of the robot. Hence we need to get the current position of the robot with the method *getCurrentState* of our object *group*. For the goal position, we decided to define by ourself the exact position of all the joints of the robot and input it as the goal. To do it we first define a map data structure which will associate for each string (the name of the joint) a double (the joint value). The robot is a 7 degrees of freedom robot so there are 7 joints to give values to. Each joint has a name, you can see in figure **??** their position with their name. The *setJointValueTarget* method of the object *group* is then the method to call to input the goal position from the previously created map. Eventually we need to call the *plan* method from the *group* object, but this method requires a *Plan* object as argument to store the solution (or nothing if no plan is found). It should be noted that the plan method returns a boolean indicating if a solution has been found or not.

So this script was a really simple one to show the basis of a planning with the moveit library. In the following sections we will show how to add some obstacles in the environment, to change the motion planner and how to do a lot of planning.

3.3 Moving to a pose

Instead of specifying all the joint values it can be useful to just ask the end-effector to be at a defined pose (position and orientation). It is simple to do with moveit but be careful as your specified pose may not be possible to reach by the robot end-effector. The following code is the entire code to plan a motion to a pose goal.

Code 3.2: moving.cpp

```cpp
#include <moveit/move_group_interface/move_group.h>

int main(int argc, char** argv)
{
  // Initialization of the ROS node
  ros::init(argc, argv, "moving_the_robot");

  // Initialization of moveit
  moveit::planning_interface::MoveGroup group("arm");

  // Setting the start position
  group.setStartState(*group.getCurrentState());

  // Setting the goal position
  geometry_msgs::Pose target;
  target.orientation.w = 1.0;
  target.position.x = 0.6;
  target.position.y = -0.1;
  target.position.z = 0.42;
  group.setPoseTarget(target);

  // Running the moveit planning
  moveit::planning_interface::MoveGroup::Plan result_plan;
  group.plan(result_plan);

  return 0;
}
```

You can see that the only part that changed was the goal state definition compared to the simple script. We first create a *Pose* object and then set its attributes. Finally we inform moveit by using the *setPoseTarget* method.

3.4 Changing motion planner

If you want to use an other motion planner than the default one you can do it quite easily ! If the motion planner you want to use is inside OMPL then you simply need to use the method *setPlannerId* with the name of the planner for attribute. For instance, if you want to use the TRRT motion planner just add the following line.

```
group.setPlannerId("TRRTkConfigDefault");
```

Here is the list of planner's name you can use :

- SBLkConfigDefault
- ESTkConfigDefault
- LBKPIECEkConfigDefault
- BKPIECEkConfigDefault
- KPIECEkConfigDefault
- RRTkConfigDefault
- RRTConnectkConfigDefault
- RRTstarkConfigDefault
- TRRTkConfigDefault
- PRMkConfigDefault
- PRMstarkConfigDefault

If you want to use the STOMP planner you cannot use the method *setPlannerId*, instead you will need to modify the launch file. You will need to put *stomp* inside the *planning_config* argument, *false* for the *use_ompl* argument and *true* for the *use_stomp* argument like the following.

```
<!-- Choose planner [ompl|chomp|stomp] -->
<arg name="planning_config" default="stomp"/>
<!-- If you choose ompl, "use_ompl" is true. -->
<arg name="use_ompl" default="true"/>
<!-- If you choose stomp, "use_stomp" is true. -->
<arg name="use_stomp" default="false"/>
```

3.5 Adding an obstacle

3.5.1 A simple box

The previous scripts we have shown are a bit boring without any obstacle. In this section we will explain how to add an obstacle which will be visible in Rviz. First you will need to include the *planning_scene_interface.h* file and create a *PlanningSceneInterface* object. This object will be used to add obstacle in the moveit environment. Then you will need to create a vector which will contain all the obstacles you want to add. An obstacle is a *CollisionObject* object, it will need a unique to be identified. In this simple example we choose to add a box as obstacle, you should then first define its dimensions. When the shape of the obstacle has been defined (in this case a box) you will need to define its pose in the environment. In this example we only choose its (x, y, z) coordinates but you can also choose its orientation. Once it has been done you can simply use the *push_back* method of the obstacle to add its primitive and pose information. Finally you should add the obstacle to the vector previously created and update the *planning_scene.* The following code illustrate all of this. Note that the goal position has been changed to show that the robot needs to avoid the newly add obstacle.

Code 3.3: moving.cpp

```cpp
#include <moveit/move_group_interface/move_group.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

int main(int argc, char** argv)
{
  // Initialization of the ROS node
  ros::init(argc, argv, "moving_the_robot");

  // Initialization of moveit
  moveit::planning_interface::MoveGroup group("arm");

  // Setting the start position
  group.setStartState(*group.getCurrentState());

  // Setting the goal position
  std::map<std::string, double> joints;

  joints["joint_s"] = 0.12;
  joints["joint_l"] = -0.11;
  joints["joint_e"] = -0.17;
  joints["joint_u"] = -1.49;
  joints["joint_r"] = 0.18;
  joints["joint_b"] = 1.35;
  joints["joint_t"] = -0.42;

  group.setJointValueTarget(joints);
  // Adding obstacle
  moveit::planning_interface::PlanningSceneInterface planning_scene;
  std::vector<moveit_msgs::CollisionObject> collision_objects;

  moveit_msgs::CollisionObject obstacle;
  obstacle.header.frame_id = group.getPlanningFrame();
  sleep(5.0);
  // The id of the object is used to identify it.
  obstacle.id = "obstacle";

  // Define a box to add to the world.
  shape_msgs::SolidPrimitive primitive;
  primitive.type = primitive.BOX;
  primitive.dimensions.resize(3);
  primitive.dimensions[0] = 0.4;
  primitive.dimensions[1] = 0.3;
  primitive.dimensions[2] = 0.05;

  // A pose for the box (specified relative to frame_id)
  geometry_msgs::Pose obstacle_pose;
  obstacle_pose.orientation.w = 1.0;
  obstacle_pose.position.x = 0.45;
  obstacle_pose.position.y = 0.0;
  obstacle_pose.position.z = 0.45;

  obstacle.primitives.push_back(primitive);
  obstacle.primitive_poses.push_back(obstacle_pose);
  obstacle.operation = obstacle.ADD;

  collision_objects.push_back(obstacle);
  planning_scene.addCollisionObjects(collision_objects);

  // Running the moveit planning
  moveit::planning_interface::MoveGroup::Plan result_plan;
  group.plan(result_plan);

  return 0;
}
```

3.5.2 Other primitives

Sometimes you will not want to have obstacle with the shape of a box, you can for example have a bottle near your robot that would be better described as a cylinder rather than a box. Moveit gives you in four basic shapes : box, sphere, cylinder and cone. However you can also add a more complicated primitive by using meshes. When you want to use another primitive than the box you will only need to change the primitive part of the code, there is no need to change the position and orientation part.

If you want to use a sphere you will need first to resize the dimension of your primitive to 1. Indeed, a sphere is entirely described by its radius, so there is only 1 parameter required. The following code shows how to change the box to a sphere with a $0.04m$ radius .

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.SPHERE;
primitive.dimensions.resize(1);
primitive.dimensions[0] = 0.04; // radius
```

If you prefer a cylinder, it is almost the same except that you will need two parameters : one for the height and one for the radius.

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.CYLINDER;
primitive.dimensions.resize(2);
primitive.dimensions[0] = 0.04; // height
primitive.dimensions[1] = 0.4; // radius
```

Finally, if you want a cone you will also need two parameters : one for the height and one for the radius of the base.

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.CONE;
primitive.dimensions.resize(2);
primitive.dimensions[0] = 0.04; // height
primitive.dimensions[1] = 0.4; // radius
```

Chapter 4

Real robot

In this chapter we will learn how to manipulate the real robot instead of only the simulation. Remember that using the robot could be dangerous for you and the other lab members ! Besides, you need to be careful and prevent it to make damage (to the cable or to its surroundings).

4.1 Connecting to the robot

4.2 Moving the robot

Chapter 5

Kinect

In this chapter we will learn how to use the Kinect and create script to gather the data they collect in real time.