Motoman manual

Author:
Thibault Barbié

Version:
0.0.1

March 15, 2017

Contents

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.

2 Motoman Project Utilisation

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 program 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
```

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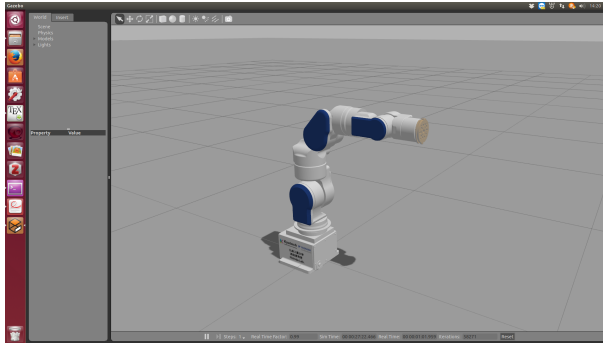
```
cd workspace
source devel/setup.bash
```

Figure 1: Yo

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 1.

2.1 Motion planning

From a script we can control the robot and ask it to move. But first we need to launch a launch file that will initialize everything. To begin everything we need to create a ros package. You can name it anything you want, in this manual we will call it *motion_planning*. We then create a launch folder inside the package.

```
cd workspace/src
catkin_create_pkg motion_planning roscpp
cd motion_planning
mkdir 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.

```xml
<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"/>
```

```xml
<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
```

```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;
}
```

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.

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. You can just copy and paste the following files.

```cmake
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})
```

```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>
<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 has been done to use the script you wrote. First compile everything. Everytime 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.

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 3.
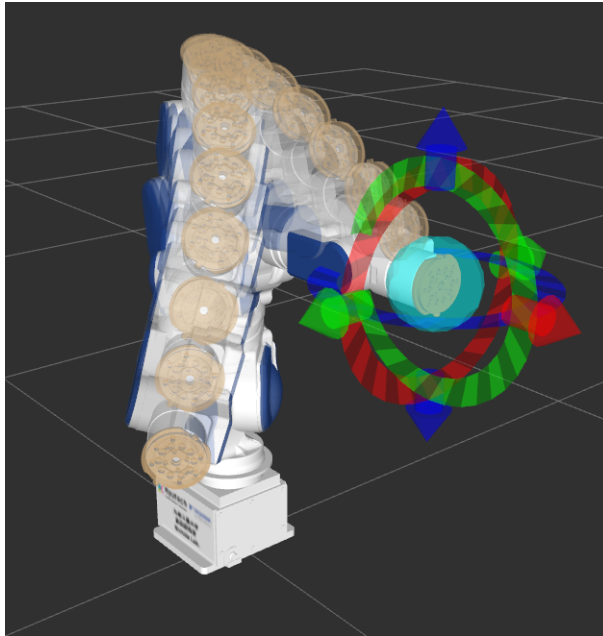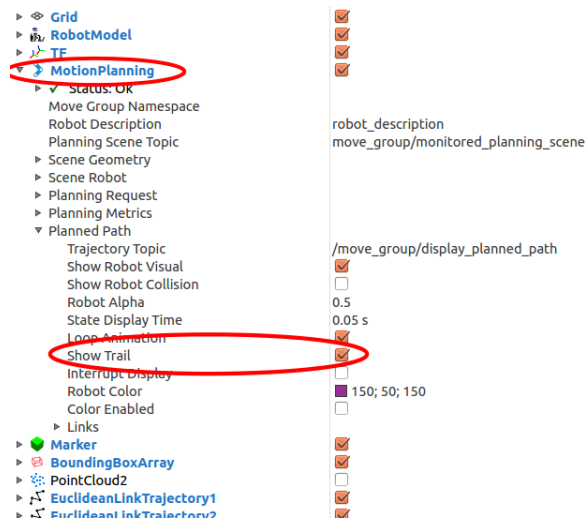
Figure 2: Yo

Figure 3: Yo

3 Kinect

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