Homework 1

Valentina Giannotti

November 2, 2023

Github link: https://github.com/ValentinaGiannotti/arm.git

1 Create the Description of the Robot and Visualize it in Rviz

1.1 Download the `arm_description` Package

Download the `arm_description` package from the repo https://github.com/RoboticsLab2023/`arm_description.git` into your `catkin_ws` using git commands

```
$ git clone https://github.com/RoboticsLab2023/arm_description.git
```

1.2 Launch File and Rviz

Within the package create a launch folder containing a launch file named display.launch that loads the URDF as a `robot_description` ROS param and starts the `robot_state_publisher` node, the `joint_state_publisher` node, and the rviz node. Launch the file using roslaunch. Note: To visualize your robot in rviz you have to changhe the Fixed Frame in the lateral bar and add the RobotModel plugin interface. Optional: save a .rviz configuration file, thad auto- matically loads the RobotModel plugin by default, and give it as an argument to your node in the display.launch file

To visualize the robot's description in Rviz, I started by navigating to the '`arm_description`' package.Then I created a launch file Folder and inside it I added display.launch.

```
$ roscd arm_description  # Navigate to the package's directory
$ mkdir launch
$ touch display.launch
```

This launch file (XML) is used to set up and initiate various components required for visualizing a robot in the RViz visualization software.

```
<?xml version="1.0"?>

<launch>
  <!-- Load the URDF as a robot_description parameter-->
  <param name="robot_description" textfile="$(find arm_description)/urdf
  /arm_description.urdf"/>

  <!-- Start robot_state_publisher -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="
      robot_state_publisher" />

  <!-- Start joint_state_publisher -->
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="
      joint_state_publisher" />

  <!-- Start RViz with your custom configuration file -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find arm_description)/
      arm_config.rviz" />

</launch>
```

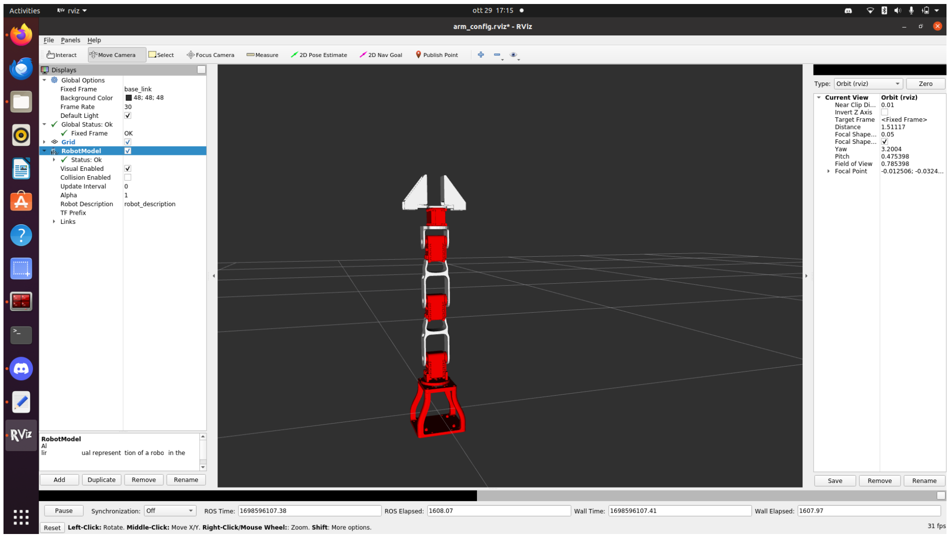

Figure 1:

This launch file loads the robot's configuration from a URDF file, initializes nodes to broadcast the robot's joint states for control and visualization, and then launches RViz with a tailored setup to visualize the robot. All of these actions are driven by the provided URDF model and configuration settings.

Afterward, I execute the 'display.launch' file and initiate Rviz by running the following command from the terminal:

```
$ roslaunch arm_description display.launch
```

To visualize my robot in rviz I change the Fixed Frame in the lateral bar and add the RobotModel plugin interface.

1.3 Edit URDF for Collision Shapes

Substitute the collision meshes of your URDF with primitive shapes. Use ¡box¿ geometries of reasonabe size approximating the links. Hint: Enable collision visualization in rviz (go to the lateral bar ¿ Robot model ¿ Collision Enabled) to adjust the collision meshes size

```
<link name="base_link">
    <visual>
      <geometry>
        <mesh filename="package://arm_description/meshes/base_link.stl" scale=
            "0.001 0.001 0.001"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.09 0.09 0.09"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="1.06682889e+08" ixy="0.0" ixz="0.0" iyy="9.92165844e+07"
         iyz="0.0" izz="1.26939175e+08"/>
    </inertial>
  </link>
```

The size of the primitive shape is chosen according to the approximate dimensions of the link it represents.

With the URDF file updated to include these simplified collision shapes, we can visualize the collision in Rviz as shown in figure 2.

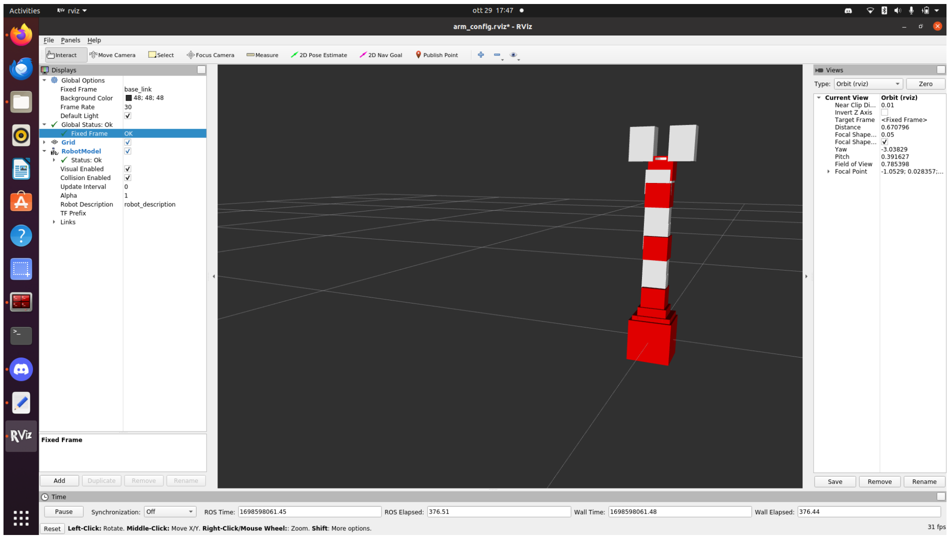

Figure 2:

1.4 Creating arm.gazebo.xacro

Create a file named arm.gazebo.xacro within your package, define a xacro:macro inside your file containing all the ¡gazebo¿ tags you find within your arm.urdf and import it in your URDF using xacro:include. Remember to rename your URDF file to arm.urdf.xacro, add the string xmlns:xacro="http://www.ros.org/wiki/xacro" within the ¡robot¿ tag, and load the URDF in your launch file using the xacro routine

To create the 'arm.gazebo.xacro' file, I follow these steps. First, I navigate to the package's directory. Once there, I create a new file within the package and name it 'arm.gazebo.xacro.' Now, within the 'arm.gazebo.xacro' file, I define a 'xacro:macro' that encapsulates all the '¡gazebo¿' tags.

```xml
<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="arm_gazebo">

    <gazebo reference="f4">
      <material>Gazebo/Red</material>
    </gazebo>

    <gazebo reference="f5">
      <material>Gazebo/Red</material>
    </gazebo>

    ...


  </xacro:macro>

</robot>
```

First, I rename the URDF file, changing it from ”arm.urdf” to ”arm.urdf.xacro.” Then, I incorporate the xacro:include directive to embed the content of ”arm.gazebo.xacro” within the ”arm.urdf.xacro” file. Subsequently, I remove all the Gazebo references that had previously existed in the ”arm.gazebo.xacro” file.

```
<?xml version="1.0"?>

<robot name="arm" xmlns:xacro="http://www.ros.org/wiki/xacro">

<xacro:include filename="$(find arm_description)/urdf/arm.gazebo.xacro"/>

  <!-- robot's links and joints definitions go here -->
    ....

<!-- arm.urdf.xacro -->
<xacro:arm_gazebo/>

</robot>
```

Then I modify the launch file like this:

```
<launch>
 <param name="robot_description" command="$(find xacro)/xacro '$(find
     arm_description)/urdf/arm.urdf.xacro'"/>

 ...

</launch>
```

2 Add transmission and controllers to the robot and spawn it in Gazebo

2.1 Create a arm_gazebo Package

```
$ cd /.../catkin_ws/src
$ catkin_create_pkg arm_gazebo roscpp rospy std_msgs gazebo_ros
$ catkin_build
```

2.2 Create a Launch folder and file

Within this package create a launch folder containing a `arm_world.launch` file

```
$ cd /catkin_ws/src/arm_gazebo
$ mkdir launch
$ touch launch/arm_world.launch
```

2.3 Edit the Launch File for Gazebo

Fill this launch file with commands that load the URDF into the ROS Parameter Server and spawn your robot using the `spawn_model` node.

```
<?xml version="1.0"?>
<launch>

    <!-- Loads the arm.world environment in Gazebo. -->

    <arg name="paused" default="false"/>
```

```
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="hardware_interface" default="PositionJointInterface"/>
    <arg name="debug" default="false"/>
    <arg name="robot_name" default="arm" />
    <arg name="model" default="arm" />

    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <!--arg name="world_name" value="$(find arm_gazebo)/worlds/arm.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>

    <!-- Load the URDF with the given hardware interface into the ROS
       Parameter Server -->
    <include file="$(find arm_description)/launch/$(arg model)_upload.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)"/>
        <arg name="robot_name" value="$(arg robot_name)" />
    </include>

    <!-- Run a python script to send a service call to gazebo_ros to spawn a
       URDF robot -->
    <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="
       false" output="screen"
          args="-urdf -model arm -param robot_description"/>
</launch>
```

Than I also created the `arm_upload.launch` in the `arm_description/launch` folder.

```
<?xml version="1.0"?>
<launch>

<param name="robot_description" command="$(find xacro)/xacro '$(find
    arm_description)/urdf/arm.urdf.xacro'"/>

</launch>
```

We can visualize the robot in Gazebo, shown in figure 3, by launching the `arm_world.launch` file:

```
$ catkin build
$ source devel/setup.bash
$ roslaunch arm_gazebo arm_world.launch
```

2.4 Add a PositionJointInterface as Hardware Interface

Now add a PositionJointInterface as hardware interface to your robot: create a `arm.transmission.xacro` file into your `arm_description/urdf` folder containing a xacro:macro with the hardware interface and load it into your arm.urdf.xacro file using xacro:include.

To add a PositionJointInterface hardware interface to your robot, create a file named `arm.transmission.xacro` in the 'urdf' folder of the `arm_description` repository. I'm going to show the code below.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:macro name="arm_transmission">

<xacro:arg name="robot_name" default="arm"/>
<xacro:arg name="hardware_interface" default="PositionJointInterface"/>
```

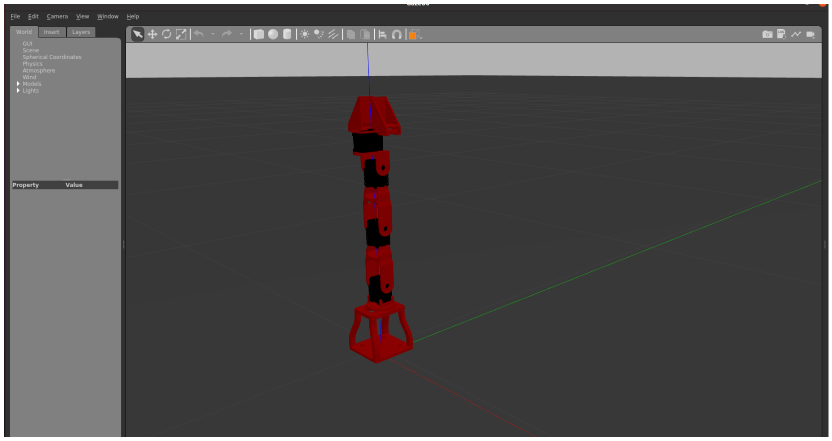

Figure 3:

```
    <transmission name="$(arg robot_name)_tran_0">
      <robotNamespace>/$(arg robot_name)</robotNamespace>
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="j0">
        <hardwareInterface>hardware_interface/$(arg hardware_interface)</
            hardwareInterface>
      </joint>
      <actuator name="$(arg robot_name)_motor_0">
        <hardwareInterface>hardware_interface/$(arg hardware_interface)</
            hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>

    ...

  </xacro:macro>
</robot>
```

To integrate this hardware interface into our robot model, the `<xacro:include>` tag was employed within the `arm.urdf.xacro` file, at the end of the file this macro is used. Below are the lines of code added to the arm.urdf.xacro file.

```
<robot name="arm" xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:include filename="$(find arm_description)/urdf/arm.transmission.xacro"/
    >
...
<xacro:arm_transmission />
</robot>
```

2.5 Add joint position controllers to your robot

Add joint position controllers to your robot: create a `arm_control` package with a `arm_control.launch` file inside its launch folder and a `arm_control.yalm` file within its config folder

```
$ cd/catkin_ws/src/arm_control
$ catkin_create_pkg arm_control roscpp controller_manager
    controller_manager_msgs
```

2.6 Add joint position controllers to your robot

Fill the `arm_control.launch` file with commands that load the joint controller configurations from the `arm_control.yalm` file to the parameter server and spawn the controllers using the `controller_manager` package.

```
<?xml version="1.0"?>
<launch>

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

    <!-- Loads the controllers -->
    <node name="controller_spawner" pkg="controller_manager" type="spawner"
       respawn="false"
          output="screen" ns="arm" args="joint_state_controller
              PositionJointInterface_J0_controller
              PositionJointInterface_J1_controller
              PositionJointInterface_J2_controller
              PositionJointInterface_J3_controller" />

    <!-- Converts joint states to TF transforms for rviz, etc -->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="
       robot_state_publisher"
          respawn="false" output="screen">
        <remap from="/joint_states" to="/arm/joint_states" />
       </node>

</launch>
```

2.7 Adding a `joint_state_controller` and a JointPositionController

Fill the `arm_control.yalm` adding a `joint_state_controller` and a JointPositionController to all the joints

```
    arm:
  # Publish all joint states -----------------------------------
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  # Controllers for singular joint ------------------------------------
  #
  # Effort Position Controllers ---------------------------------------

    # Forward Position Controllers ---------------------------------------
  PositionJointInterface_J0_controller:
    type: position_controllers/JointPositionController
    joint: j0
    pid: {p: 800.0, i: 100, d: 80.0}

  PositionJointInterface_J1_controller:
    type: position_controllers/JointPositionController
    joint: j1
    pid: {p: 800.0, i: 100, d: 80.0}

  PositionJointInterface_J2_controller:
    type: position_controllers/JointPositionController
    joint: j2
    pid: {p: 800.0, i: 100, d: 80.0}

  PositionJointInterface_J3_controller:
```

```
    type: position_controllers/JointPositionController
    joint: j3
    pid: {p: 800.0, i: 100, d: 80.0}
```

2.8 Create an arm_gazebo.launch file

Create an `arm_gazebo.launch` file into the launch folder of the `arm_gazebo` package loading the Gazebo world with `arm_world.launch` and spawning the controllers within `arm_control.launch` Go to the `arm_description` package and add the `gazebo_ros_control` plugin to your main URDF into the `arm.gazebo.xacro` file. Launch the simulation and check if your controllers are correctly loaded.

```
    <?xml version="1.0"?>
<launch>

    <include file="$(find arm_gazebo)/launch/arm_world.launch" />
    <include file="$(find arm_control)/launch/arm_control.launch"/>

</launch>
```

Then I add the `gazebo_ros_control` plugin to URDF into the `arm.gazebo.xacro` file. I modify `arm.gazebo.xacro` adding this code line:

```
   <gazebo>

   ...
      <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
        <robotNamespace>/arm</robotNamespace>
      </plugin>
    ...

    </gazebo>
```

3 Add a camera sensor to your robot

3.1 Add a camera_link and a fixed camera_joint

Go into your `arm.urdf.xacro` file and add a `camera_link` and a fixed `camera_joint` with `base_link` as a parent link. Size and position the camera link opportunely.

I modify `arm.urdf.xacro` file and add those code lines:

```
<?xml version="1.0"?>

<robot name="arm" xmlns:xacro="http://www.ros.org/wiki/xacro">


   ...
    <link name="camera_link">
         <visual>
      <geometry>
        <box size="0.05 0.05 0.05"/>
      </geometry>
    </visual>
        </link>

    <joint name="camera_joint" type="fixed">
        <parent link="base_link"/>
        <child link="camera_link"/>
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </joint>
    ...
```

```
</robot>
```

3.2 Add a camera_link and a fixed camera_joint

In the arm.gazebo.xacro add the gazebo sensor reference tags and the libgazebo_ros_camera plugin to your xacro.

```
    <?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="arm_gazebo">

  ...

    <!-- Now, add the camera link and Gazebo sensor -->
    <gazebo reference="camera_link">
      <sensor type="camera" name="camera1">
        <update_rate>30.0</update_rate>
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>800</width> <height>800</height> <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near> <far>300</far>
        </clip>
        <noise>
          <type>gaussian</type> <mean>0.0</mean> <stddev>0.007</stddev>
        </noise>
      </camera>
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>camera</cameraName>
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera_link_optical</frameName>
        <hackBaseline>0.0</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
        <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
        <CxPrime>0</CxPrime>
        <Cx>0.0</Cx>
        <Cy>0.0</Cy>
        <focalLength>0.0</focalLength>
      </plugin>
      </sensor>
    </gazebo>

  </xacro:macro>

</robot>
```

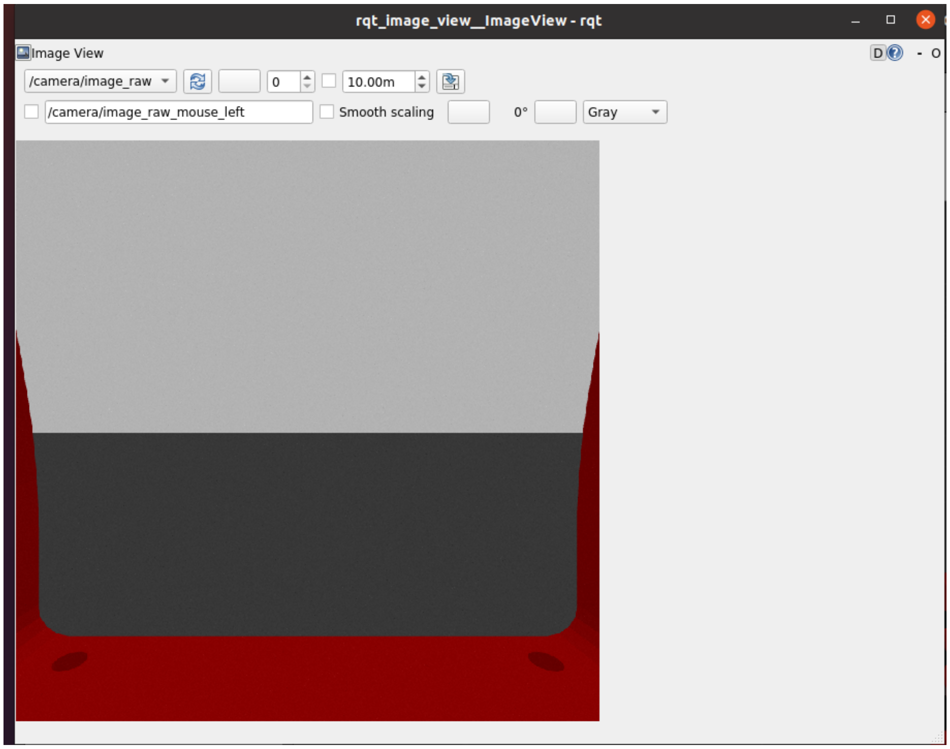

Figure 4:

3.3 Check the camera

Launch the Gazebo simulation with using `arm_gazebo.launch` and check if the image topic is correctly published using `rqt_image_view`

I launch the Gazebo simulation with `arm_gazebo.launch` in the terminal:

```
$roslaunch arm_gazebo arm_gazebo.launch
```

Then I Open a new terminal and run `rqt_image_view` to visualize the camera image topic.

```
$rosrun rqt_image_view rqt_image_view
```

There is nothing visible because there is nothing in the surrounding space, and the camera is positioned on the base.

3.4 Optionally: You can create a camera.xacro file

I will show now the camera.xacro file:

```
<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="arm_camera">

    <gazebo reference="camera_link">
      <sensor type="camera" name="camera1">
        <update_rate>30.0</update_rate>
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
```

```
        <width>800</width> <height>800</height> <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near> <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type> <mean>0.0</mean> <stddev>0.007</stddev>
      </noise>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>camera</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>camera_link_optical</frameName>
      <hackBaseline>0.0</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
      <CxPrime>0</CxPrime>
      <Cx>0.0</Cx>
      <Cy>0.0</Cy>
      <focalLength>0.0</focalLength>
    </plugin>
    </sensor>
  </gazebo>

 </xacro:macro>

</robot>
```

Then I modify arm.xacro.file adding this code's lines:

```
<?xml version="1.0"?>

<robot name="arm" xmlns:xacro="http://www.ros.org/wiki/xacro">

<xacro:include filename="$(find arm_description)/urdf/arm.gazebo.xacro"/>
<xacro:include filename="$(find arm_description)/urdf/arm.transmission.xacro"/
   >
<xacro:include filename="$(find arm_description)/urdf/camera.xacro"/>

...

 <xacro:arm_camera/>

</robot>
```

4 Reads the joint state and sends joint position commands

4.1 Create an arm_controller package

Create an `arm_controller` package with a ROS C++ node named `arm_controller_node`. The dependencies are roscpp, `sensor_msgs` and `std_msgs`. Modify opportunely the `CMakeLists.txt` file to compile your node. Hint: uncomment `add_executable` and `target_link_libraries` lines.

```
$cd /path/to/your/catkin_ws/src
$catkin_create_pkg arm_controller roscpp sensor_msgs std_msgs
```

4.2 Create a subscriber to the topic `joint_states` and a callback function

Create a subscriber to the topic `joint_states` and a callback function that prints the current joint positions. Note: the topic contains a `sensor_msgs/JointState`

```
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>

void jointStateCallback(const sensor_msgs::JointState::ConstPtr& msg) {
    // Your control logic goes here
       ROS_INFO("\nReceived joint positions:");
    for (size_t i = 0; i < msg->position.size(); i++) {
        ROS_INFO("%f", msg->position[i]);
    }
}
int main(int argc, char** argv){

    ros::init(argc, argv, "arm_controller_node");
    ros::NodeHandle nh;

    // Create a subscriber to the joint state topic
    ros::Subscriber joint_state_sub = nh.subscribe("/arm/joint_states", 10,
        jointStateCallback);
...

  return 0;
}
```

Subsequently, the CMakeLists.txt file was modified to compile your node by uncommenting the `add_executable` and `target_link_libraries` lines as follows:

```
...

add_executable(arm_controller_node src/arm_controller_node.cpp)

...

## Specify libraries to link a library or executable target against
target_link_libraries(arm_controller_node
   ${catkin_LIBRARIES}
 )
```

4.3 Create publishers

Create publishers that write commands into the controllers' /command topics. Note: the command is a `std_msgs/Float64`

```
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <std_msgs/Float64.h>

void jointStateCallback(const sensor_msgs::JointState::ConstPtr& msg) {

...
```

```
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "arm_controller_node");
    ros::NodeHandle nh;
    ros::Rate loop_rate(10);

    // Create a subscriber to the joint state topic
    ros::Subscriber joint_state_sub = nh.subscribe("/arm/joint_states", 10,
        jointStateCallback);

  ros::Publisher joint0_pub = nh.advertise<std_msgs::Float64>("/arm/
      PositionJointInterface_J0_controller/command", 1);
  ros::Publisher joint1_pub = nh.advertise<std_msgs::Float64>("/arm/
      PositionJointInterface_J1_controller/command", 1);
  ros::Publisher joint2_pub = nh.advertise<std_msgs::Float64>("/arm/
      PositionJointInterface_J2_controller/command", 1);
  ros::Publisher joint3_pub = nh.advertise<std_msgs::Float64>("/arm/
      PositionJointInterface_J3_controller/command", 1);

  while (ros::ok())
  {

    std_msgs::Float64 joint0_command;
    joint0_command.data =1.6;
    joint0_pub.publish(joint0_command);

    std_msgs::Float64 joint1_command;
    joint1_command.data = 0.5;
    joint1_pub.publish(joint1_command);

    std_msgs::Float64 joint2_command;
    joint2_command.data = -1.2;
    joint2_pub.publish(joint2_command);

    std_msgs::Float64 joint3_command;
    joint3_command.data = -1;
    joint3_pub.publish(joint3_command);

    ros::spinOnce();
    loop_rate.sleep();
  }

  return 0;
}
```

To start the controller, after launching `arm_gazebo.launch`, I opened another terminal and executed the following command:

```
$rosrun arm_controller arm_controller_node
```

The assumed position is as follows:

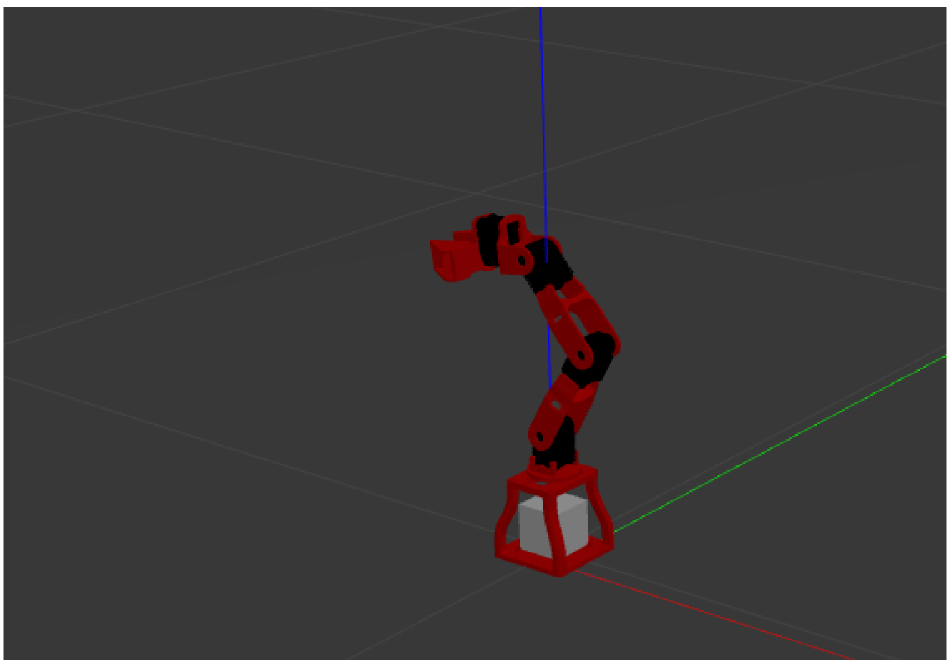

Figure 5: