Robotics Lab: Homework 1 report

Student: Ludovica Ruggiero

Contents

Introduction

The goal of this homework is to build ROS packages to simulate a 4-degrees-of-freedom robotic manipulator arm into the Gazebo environment. Some of the implemented points were a collaborative effort with my colleague Luigi Catello. Together, we worked on tasks such as creating collision meshes, inserting the camera sensor, and we generally discussed about some implementation features.

The code referenced in this report is accessible via the following URL: https://github.com/ludruggiero/RL.git.

1 Robot Description

1.a Setup: arm_description creation

Firstly, the ‘`arm_description`‘ package has been downloaded from the repository at https://github.com/RoboticsLab2023/arm_description.git, using the ‘`git clone`‘ command inside the catkin workspace.

1.b Setup: display.launch creation

Within this package, a ‘launch‘ folder was created, containing a launch file named ‘`display.launch`‘. This file contains commands to load the URDF file located in the ‘`urdf`‘ folder into a ROS parameter named ‘`robot_description`‘.

Next, two nodes were initialized: the ‘`joint_state_publisher`‘ node, responsible for publishing the state of the 4 joints, and the ‘`robot_state_publisher`‘ node which subscribes to the ‘`\joint_states`‘ topic and publishes two topics: ‘`tf`‘ and ‘`tf_static`‘. This setup is illustrated in Figure 1.1.

```
<launch>

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

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

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

    <!-- Start RViz -->
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
    arm_description)/config/config.rviz" />

</launch>
```

Code 1: display_launch.launch code

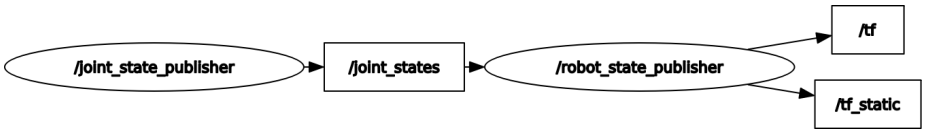

Figure 1.1: rqt graph obtained by launching display_launch.launch

The displayed rviz configuration is shown in Figure 1.2 and has been saved in the `config.rviz` file, saved in the `config` folder of the package.

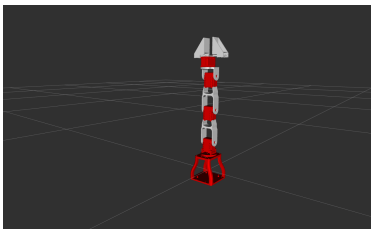

Figure 1.2: rviz configuration

1.c Collision meshes

To reduce the computational complexity, the collision meshes have been substituted with the `<box>` primitive shapes.

To do that, in the `arm.urdf` file, the `<collision>` tag of each link has been replaced as shown in Code 2, adjusting the collision mesh sizes and the collision frame origin positions. The measuring tool in RViz has been useful to measure the link for obtaining more accurate results.

```
<link name= 'dyn2'>
....
    <collision>
      <geometry>
          <box size="0.055 0.035 0.045"/>
       </geometry>
      <origin rpy="0 0 0" xyz="0.012 0 0"/>
    </collision>
...
</link>
```

Code 2: Collision of link "dyn2"

The RViz configuration obtained by enabling the collision visualization is shown in Figure 1.3.

1.d Gazebo macro

The `urdf` have been transfomed into a `xacro` file in order to create some `xacro::macro`. It had been sufficient to rename such file into `arm.urdf.xacro`,

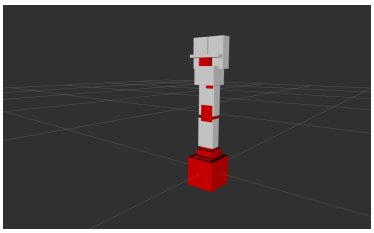

Figure 1.3: RViz configuration with collision meshes enabled

change the the `robot_description` parameter definitions in the launch file as shown in Code 3, and to add the string `xmlns:xacro="http://www.ros.org/wiki/xacro"` within the `<robot>` tag.

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

Code 3: new definition of the robot_description parameter

Then a `xacro:: macro` containing all the `gazebo` tags necessary to use the gazebo environment was created, included and used in the launch file.

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <xacro:macro name="gazebo_tags">
....
    <gazebo reference="f4">
      <material>Gazebo/Red</material>
    </gazebo>
    ....
    </xacro:macro>
</robot>
```

Code 4: arm_gazebo.xacro structure

2 Transmissions and Controllers

2.a Package creation

To control the robot's trajectory, it was necessary to incorporate transmissions and actuators into the robot's joints. To do this, a new package named `arm_gazebo` was created.

2.b arm_world.launch creation

The robot was spawn in the gazebo environment trough the `arm_world.launch` file in the launch file directory.

2.c Gazebo setup

Within this launch file, the Gazebo `empty_world` was initiated, configuring the `robot_description` parameter as described in Code 3. Additionally the urdf model was spawned into the Gazebo world, where its name have been set to `arm`.

```
<launch>
    <!-- Include Gazebo ROS package to set up Gazebo -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
    ...
    <!-- Spawn robot in Gazebo -->
    <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
    output="screen"
    respawn="false" args="-urdf -param
    robot_description -model arm" />

</launch>
```

Code 5: arm_world.launch file: empty_world and urdf_spawner being started

Once the file had been launched, the world shown in Figure 2.1 was be spawned.

After a few minutes the robot collapsed due to gravity, as can be seen in Figure 2.2: that's because no actuators were inserted in the model and nothing could balance the external forces.

2.d Position Joint Interface

The four revolute joint needed to be provided of some trasmission and actuators. This operation has been done using a `xacro::macro` that provided some hardware interface to each joint. In this file, the mechanical reduction has been explicitly set to 1, meaning that the motor is directly coupled to the joint.

```
<!-- Provides transmissions and actuators to a mobile joint -->
    <robot xmlns:xacro="http://www.ros.org/wiki/xacro">
```

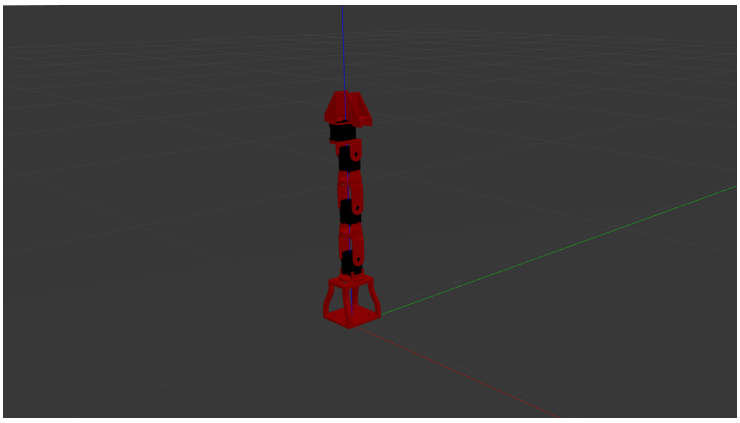

Figure 2.1: Urdf model spawned in Gazebo

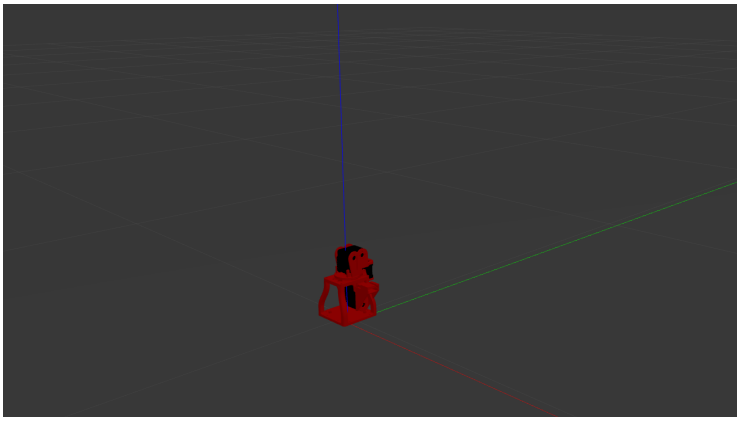

Figure 2.2: Urdf model spawned in Gazebo and collapsed due to gravity

```
    <xacro:macro name="joint_interface" params="joint_name">
        <transmission name="${joint_name}_joint_transmission">
            <type>transmission_interface/SimpleTransmission</type>
            <actuator name="${joint_name}_actuator">
                <mechanicalReduction>1</mechanicalReduction>
            </actuator>
            <joint name="${joint_name}">
                <hardwareInterface>hardware_interface/
                PositionJointInterface</hardwareInterface>
            </joint>
        </transmission>
    </xacro:macro>
</robot>
```

Code 6: arm_transmission.xacro file

The `xacro::macro` in Code 6 has been included in the urdf arm description and used for each mobile joint.

2.e Arm Control Package Creation

The `arm_control` package has been used to insert some controllers for the joint. The package included a launch folder and a configuration folder, used to define and launch the controllers.

2.f control_launch.launch file

The information provided in the configuration file has been loaded into the `control_launch.launch` file. This file also adds the controllers to the model through the `control_manager` package, which has been initialized as shown in Code 7.

```
...
<rosparam file="$(find arm_control)/config/arm_control.yaml" command="load"/>
...
<!-- Spawn joint controllers using controller_manager -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner"
  respawn="false" output="screen" ns="/$(arg robot_name)" args="$(arg controllers)">
  </node>
...
```

Code 7: arm_control.launch file: initializing the controller_spawner and loading the configuration file

2.g Controller configuration file

In the `arm_control.yaml` file located in the `config` folder of the respective package, the following controllers were included: `JointStateController`, which publishes the joint states at a rate of 50 times per second, and four instances of `JointPositionController`. This implies that our controllers do not consider the robot dynamics and exclusively operate on information based on the joint positions.

```
/arm:
# Publish all joint states-----------------------

joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

# Forward Position Controllers ------------------
joint0_position_controller:
    type: position_controllers/JointPositionController
    joint: j0
```

Code 8: arm_control.yaml file: definition of two joints

2.h Robot and controllers started in the Gazebo world

In the `arm\_gazebo` package, the `arm\_gazebo.launch` file had the aim to launch the robot and its controller.

To achive this, it was necessary to include the two launch files previously described and load them.

```
<launch>
....
  <!-- Load Gazebo world with arm_world.launch -->
  <include file="$(find arm_gazebo)/launch/arm_world.launch"/>
...
<!-- The namespace is necessary in order
to communicate with ROS -->
  <group ns="$(arg robot_name)">
  <!-- Spawn controllers - it uses a
  Position Controller for each joint -->
    <include file="$(find arm_control)/launch/arm_control.launch">
      <arg name= "hardware_interface"
      value="$(arg hardware_interface)" />
      <arg name= "controllers" value="joint_state_controller
      joint0_position_controller joint1_position_controller
      joint2_position_controller joint3_position_controller"/>
      <arg name="robot_name" value="$(arg robot_name)"/>
    </include>
  </group>
  ...
</launch>
```

Code 9: arm_gazebo.launch file

There was a need to update the `arm_description` package: the `gazebo_ros_control` plugin was added in the `arm_gazebo.xacro` macro:

```
....
 <!-- ros_control plugin -->
  <gazebo>
      <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
        <robotNamespace>/arm</robotNamespace>
      </plugin>
    </gazebo>
....
```

Code 10: arm_gazebo.xacro macro updated

After executing the launch file, the initial visualization was as shown in Figure 2.1, but the robot remained stationary without collapsing.

To verify if the controllers were loaded correctly, the `rqt_graph` node was launched, as seen in Figure 2.3. The controllers were successfully launched and Gazebo received the control signal, as indicated in the terminal output Code 11.

```
[urdf_spawner-4] process has finished cleanly
log file: /home/ludovica/.ros/log/a577be1a-7996-11ee-8893-d3774f4ac8cc/urdf_spawner-4*.log
[ INFO] [1698939844.863939598, 0.374000000]: Loaded gazebo_ros_control.
[INFO] [1698939845.128505, 0.637000]: Controller Spawner:
Waiting for service controller_manager/switch_controller
[INFO] [1698939845.130469, 0.639000]: Controller Spawner:
Waiting for service controller_manager/unload_controller
[INFO] [1698939845.131773, 0.641000]: Loading controller:
joint_state_controller
[INFO] [1698939845.148803, 0.658000]: Loading controller:
joint0_position_controller
[INFO] [1698939845.157114, 0.666000]: Loading controller:
joint1_position_controller
[INFO] [1698939845.164765, 0.674000]: Loading controller:
joint2_position_controller
[INFO] [1698939845.168564, 0.678000]: Loading controller:
joint3_position_controller
[INFO] [1698939845.172937, 0.682000]: Controller Spawner:
Loaded controllers: joint_state_controller,
joint0_position_controller,
joint1_position_controller, joint2_position_controller,
joint3_position_controller
[INFO] [1698939845.178611, 0.688000]: Started controllers:
joint_state_controller, joint0_position_controller,
joint1_position_controller, joint2_position_controller,
joint3_position_controller
```

Code 11: output of the terminal when calling the launch file

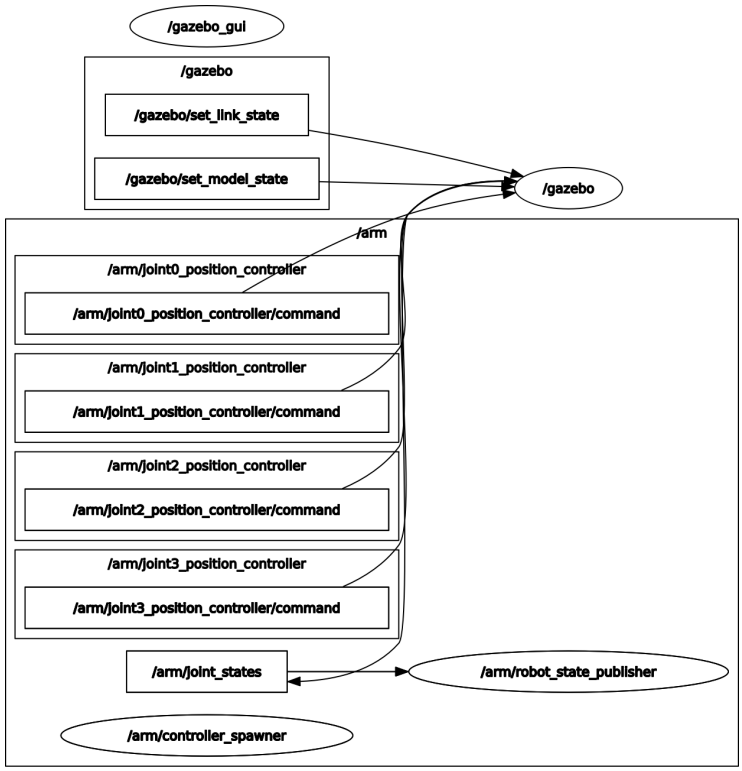

Figure 2.3: rqt_graph

3 Camera Sensor

A camera sensor was added to the robot, specifically to the base link.

3.a Link and joint definition

For this purpose, in the `urdf.xacro` file, the `camera\_joint` and the `camera\_link` were added. It was necessary to set the position and the dimension of the box used to represent the camera.

```
..
    <joint name="camera_sensor_joint" type="fixed">
              <axis xyz="0 0 1" />
              <origin xyz="${xyz}" rpy="${rpy}"/>
              <parent link="base_link"/>
              <child link="camera_link"/>
        </joint>
    <link name="camera_link">
           <collision>
              <origin xyz="0 0 0" rpy="0 0 0"/>
              <geometry>
                 <box size="0.02 0.08 0.05"/>
              </geometry>
           </collision>
           <visual>
              <origin xyz="0 0 0" rpy="0 0 0"/>
              <geometry>
                 <box size="0.02 0.08 0.05"/>
              </geometry>
           </visual>
           <inertial>
              <mass value="0.0001" />
              <origin xyz="0 0 0" rpy="0 0 ${pi}"/>
              <inertia ixx="0.0000001"
              ixy="0" ixz="0" iyy="0.0000001"
              iyz="0" izz="0.0000001" />
           </inertial>
        </link>
```

Code 12: camera_link and camera_joint definition

3.b Gazebo setup

To visualize the camera in Gazebo, the `urdf.xacro` file needed to include the gazebo tags and plugin. This was achieved as shown in the following code:

```
....
       <gazebo reference="camera_link">
          <sensor type="camera" name="camera">
             <update_rate>30.0</update_rate>
```

```
            <camera name="head">
                <horizontal_fov>1.3962634</horizontal_fov>
                <image>
                    <width>800</width>
                    <height>600</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>${robot_name}/camera</cameraName>
                <imageTopicName>image_raw</imageTopicName>
                <cameraInfoTopicName>camera_info</cameraInfoTopicName>
                <frameName>camera_link</frameName>
                <hackBaseline>0.07</hackBaseline>
                <distortionK1>0.0</distortionK1>
                <distortionK2>0.0</distortionK2>
                <distortionK3>0.0</distortionK3>
                <distortionT1>0.0</distortionT1>
                <distortionT2>0.0</distortionT2>
            </plugin>
        </sensor>
        <material>Gazebo/Green</material>
    </gazebo>
....
```

Code 13: camera_sensor definition and plugin added

The camera is represented by a green box positioned on the base link, as shown in Figure 3.1.

3.c Image view

The axis and position of the camera frame in the xacro file were adjusted in order to make it look forward. To obtain a correct visualization, the `rqt_image_view` was used, and some objects were added in the gazebo environment. The result is illustreted in Figure 3.2

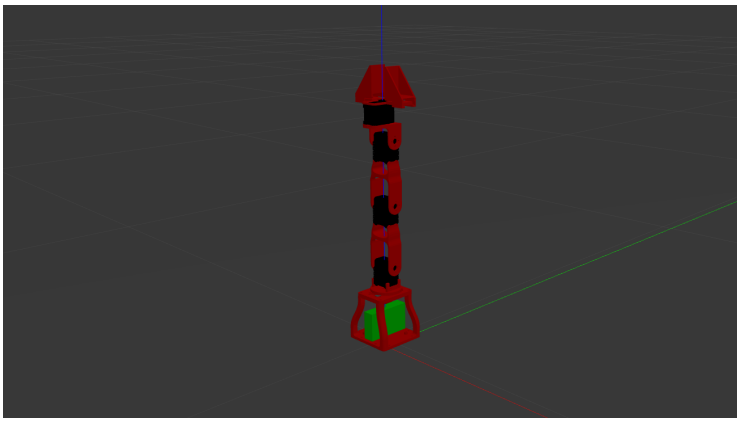

Figure 3.1: The camera is represented by a green box positioned on the base link

3.d Xacro file

The code above was been inserted in a `xacro::macro` named `camera_sensor`. This file was downloaded online and modified according to the robot's needs.

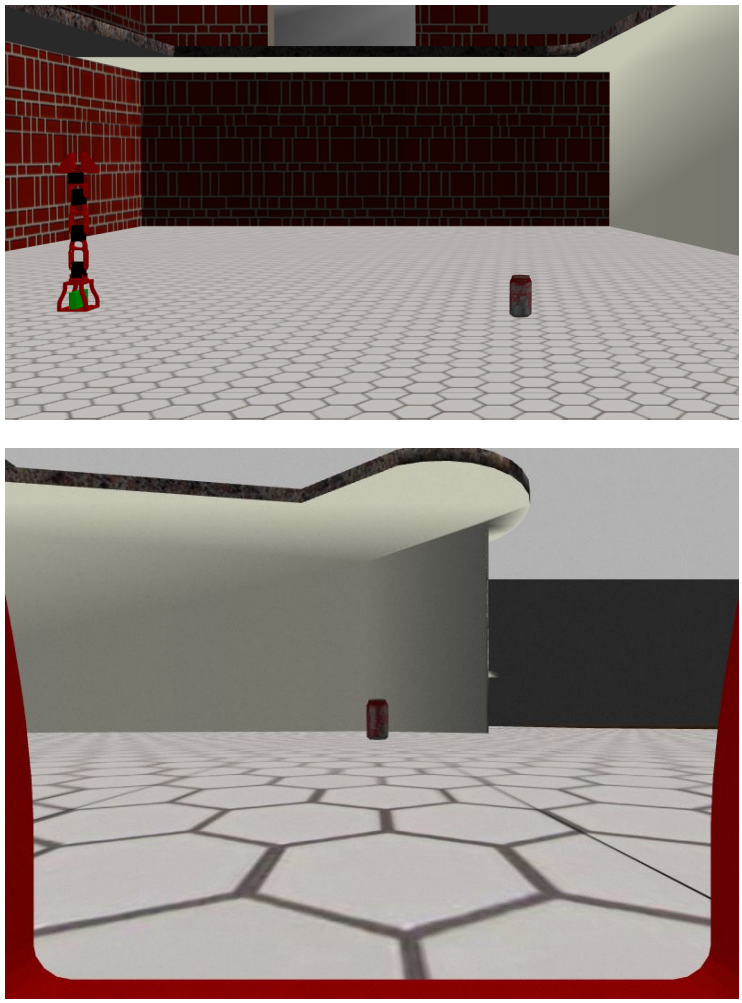

Figure 3.2: Image view: gazebo environment and camera sensor acquired image

4 Arm Controller Node

4.a arm_controller_package

A new ROS C++ package named `arm_controller` was created, whose dependencies are `roscpp`, `sensor_msgs` and `std_msgs`.

To properly compile this node, in the `CMakeLists.txt` the following line were to be uncommented:

```
....
add_executable(${PROJECT_NAME}_node
src/arm_controller_node.cpp src/Arm_Controller.cpp)
...
target_link_libraries(${PROJECT_NAME}_node
   ${catkin_LIBRARIES}
 )
 ...
```

Code 14: CMakeLists.text modified

The node was defined as an instance of a class named `Arm_Controller`. This class has a function named `start_controller` that receives the joint states from the `joint_state_controller` added in Section 2. It computes the control commands and sends them to the actuator.

The main of the C++ node is presented in Code 15

```
#include "ros/ros.h"
#include "Arm_Controller/Arm_Controller.h"
#include <sstream>
#include <cmath>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "arm_controller_node");
    ros::NodeHandle n;
    Arm_Controller arm_controller(n);
    arm_controller.start_controller();
    return 0;

}
```

Code 15: arm_controller_node.cpp

Here a node was initialized and assigned to the `arm_controller` object, that was then started.

4.b Subscriber

The class contains a subscriber to the topic `joint_states` in order to receive the information about the joints.

This message is a `sensor_msgs/JointState`, and contains information about the position, the velocity and effort. In the constructor of the class, the `subscribe`

function was called. This function provides the node the information about the name of the topic to subscribe to, the length of the message queue, the type of message to receive and the function to call when the message has been received.

```
    Arm_Controller::Arm_Controller(ros::NodeHandle& nodeHandle )
    : nodeHandle_(nodeHandle),latestJointState()
{
.....

  // the data sent to the callback function is the object of
 // the class. The private parameters can then be modified.
  subscriber_ = nodeHandle_.subscribe("/arm/joint_states", 1,
                                  &Arm_Controller::topicCallback, this);

.....
  ROS_INFO("Successfully launched node.");
}
    }
```

Code 16: Arm_Controller.cpp contructor

The callback function reads the message and copies its content in a variable named `latestJointState`, that is a parameter of the class.

```
    void Arm_Controller::topicCallback(const sensor_msgs::
                                        JointState& message)
{
    latestJointState = message;
}
```

Code 17: subscriber callback function

This variable will be used in the `start_controller` function.

4.c Publishers

The class also contains some publishers, which are used to command the actuators.

Those need to be initialized in the constructor of the class by calling the `NodeHandle.advertise` function. The topics need to match the already existing ones in order to send the messages correctly

```
....
publisher_joint_0 = nodeHandle_.advertise
<std_msgs::Float64>("/arm/joint0_position_controller/command",1);
....
```

Code 18: j_0 publisher initialization

The messages were sent to the robot in the `start_controller` function trough the `Publisher.publish` function

```
....
publisher_joint_0.publish(cmd_messages[0]);
....
```

Code 19: messages being published

4.d Control law and results

The implemented control law makes the joint positions oscillate with an amplitude of $\frac{\pi}{2}$:

```
void Arm_Controller::start_controller(){
    while (ros::ok()) {
        int i;
        std_msgs::Float64 cmd_messages[4];
        ros::Rate loopRate(10);
        ros::spinOnce(); // calling the callback function
        double frequency = 0.05;
        double amplitude = M_PI / 2.0; // +/- pi/2

        //cmd_messag
        for(i=0;i<4; ++i){
            double time = ros::Time::now().toSec();
            double angle = amplitude *
                sin(2 * M_PI * frequency * time);
            ROS_INFO("\n--Joint States--");
            ROS_INFO("Joint %s: Position
            %.4f", latestJointState.name[i].c_str(),
            latestJointState.position[i]);
            cmd_messages[i].data = angle;
        }
         // Sending commands to the actuators
        ....
        loopRate.sleep();
    }
}
```

Code 20: start_controller_function

To test the node, the `arm_gazebo.launch` file was launched with the controller node. The topics were sent and received correctly,as can be seen in Figure 4.1 and the robot in Gazebo moved as expected.

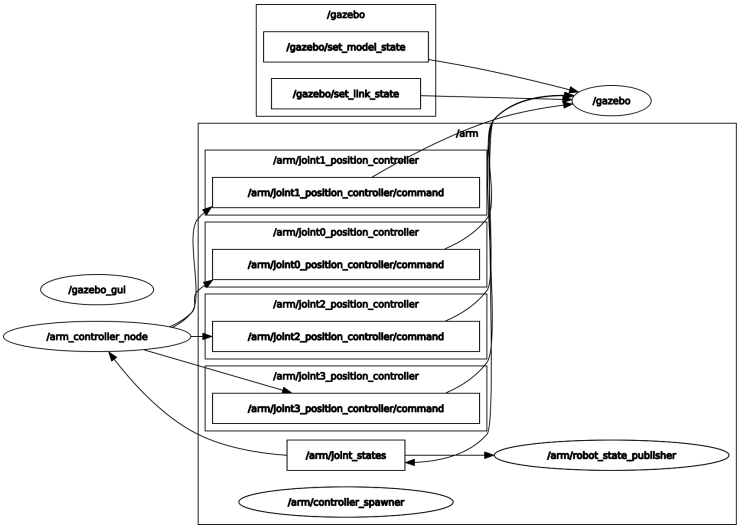

Figure 4.1: rosgraph obtained launching gazebo and the controller node

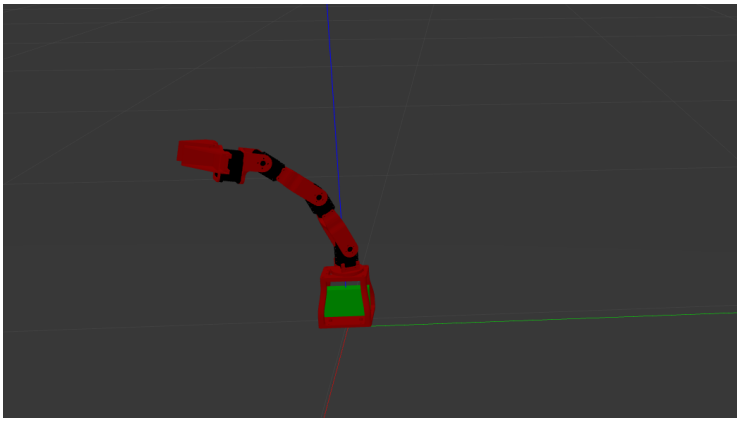

Figure 4.2: The robot moved accordingly to the control law specified in the function