Robotics Lab: Homework 1

Building your robot manipulator

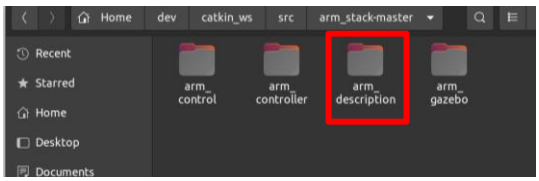
ANDREA MORGHEN P38000230
a.morghen@studenti.unina.it
https://github.com/Andremorgh/RL_HW_01.git

Esteemed Prof.
Mario Selvaggio

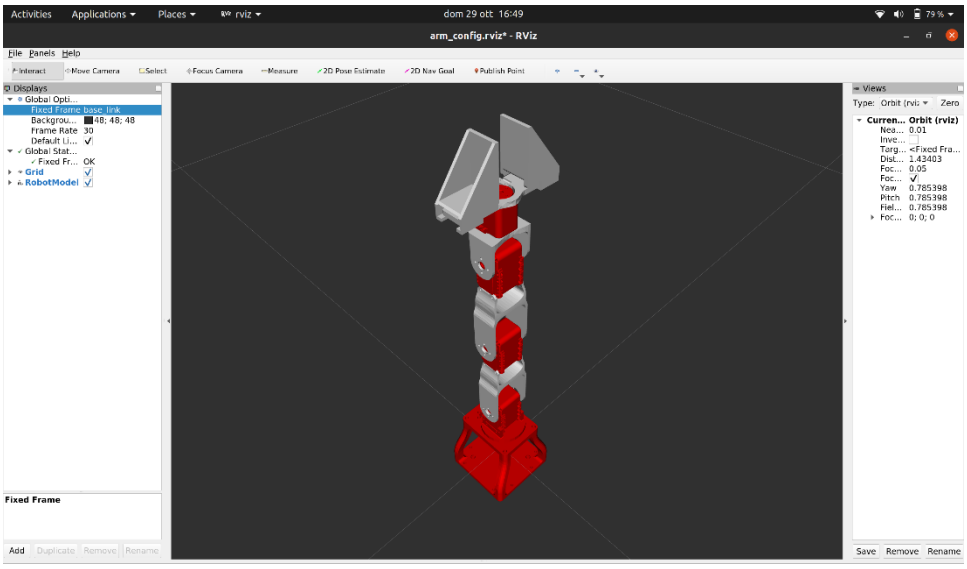
1. Create the description of your robot and visualize it in Rviz

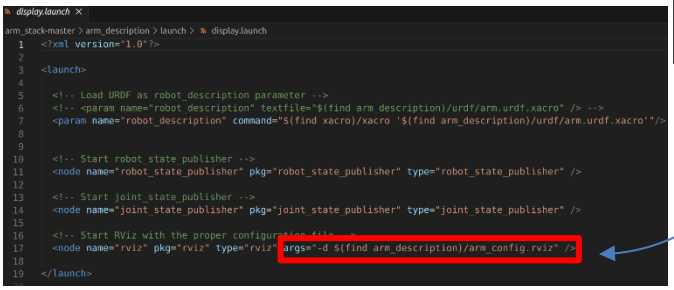
(a) Download the arm_description package from the repo https://github.com/RoboticsLab2023/arm_description.git into your catkin_ws using git commands

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



(b) 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 change the Fixed Frame in the lateral bar and add the RobotModel plugin interface. **Optional:** save a .rviz configuration file, that automatically loads the RobotModel plugin by default, and give it as an argument to your node in the display.launch file

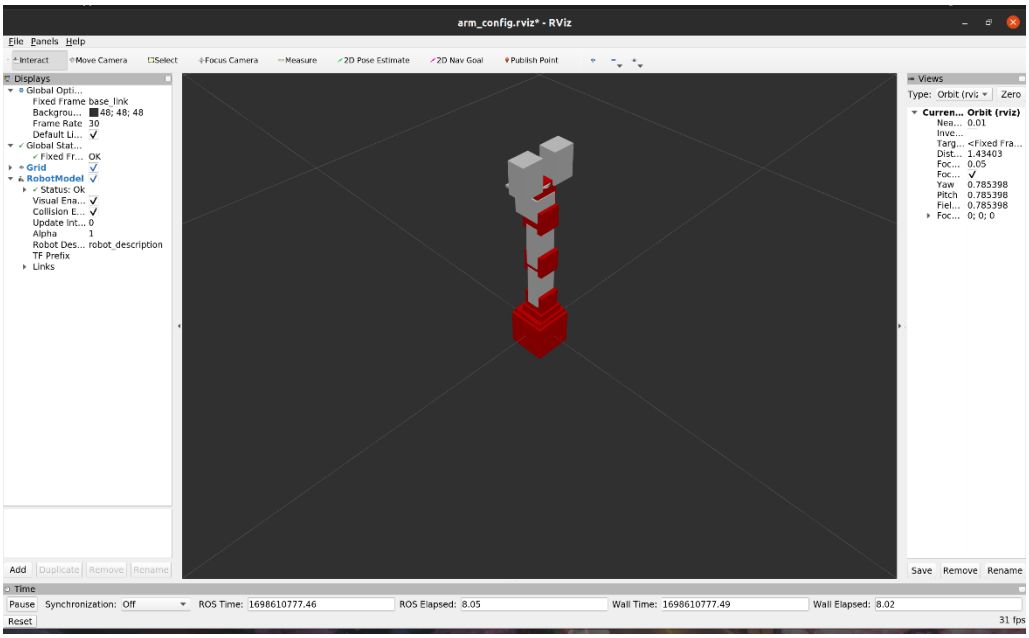




(c) Substitute the collision meshes of your URDF with primitive shapes. Use <box> geometries of 3easonable size approximating the links. **Hint:** Enable collision visualization in rviz (go to the lateral bar > Robot model > Collision Enabled) to adjust the collision meshes size

```
arm.urdf.xacro
arm_stack-master > arm_description > urdf > arm.urdf.xacro
<xacro:include filename="$(find arm_description)/urd
<!-- Import Transmissions -->
<xacro:include filename="$(find arm_description)/urd
<!-- Import Camera plugin -->
<xacro:include filename="$(find arm_description)/urd

<link name="base_link">
  <visual>
    <geometry>
      <mesh filename="package://arm_description/mesh
    </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
  </inertial>
</link>
```

This was done for all the links, calibrating the size of the boxes according to the size of the link.



(d) 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

arm.gazebo.xacro

```
arm_stack-master > arm_description > urdf > arm.gazebo.xacro
<?xml version="1.0"?>

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

  <xacro:macro name="arm_gazebo">

    <!-- Load Gazebo lib and set the robot namespace -->
    <gazebo>
      <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
        <robotNamespace>/arm</robotNamespace>
      </plugin>
    </gazebo>

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

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

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

    <gazebo reference="crawer_base">
```

arm.urdf.xacro

```
arm_stack-master > arm_description > urdf > arm.urdf.xacro
<?xml version="1.0"?>

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

  <!-- Include the Gazebo-specific tags and collision properties -->
  <xacro:include filename="$(find arm_description)/urdf/arm.gazebo.xacro" />
  <!-- Import Transmissions -->
  <xacro:include filename="$(find arm_description)/urdf/arm.transmission.xacro" />
  <!-- Import Camera plugin -->
  <xacro:include filename="$(find arm_description)/urdf/camera.urdf.xacro"/>

  <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>
```

display.launch

```
arm_stack-master > arm_description > launch > display.launch
<?xml version="1.0"?>

<launch>

  <!-- Load URDF as robot_description parameter -->
  <!-- <param name="robot_description" textfile="$(find arm_description)/urdf/arm.urdf.xacro" /> -->
  <param name="robot_description" command="$(find xacro)/xacro '$(find arm_description)/urdf/arm.urdf.xacro'"/>


  <!-- 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 the proper configuration file -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find arm_description)/arm_config.rviz" />

</launch>
```

```
  <xacro:arm_gazebo/>

  <xacro:arm_transmission robot_name="arm"/>

  <xacro:camera_sensor xyz="0 0 1" rpy="0 1.57 1.57" parent="base_link"/>

</robot>
```

2. Add transmission and controllers to your robot and spawn it in Gazebo

(a) Create a package named arm_gazebo

catkin_create_pkg arm_gazebo

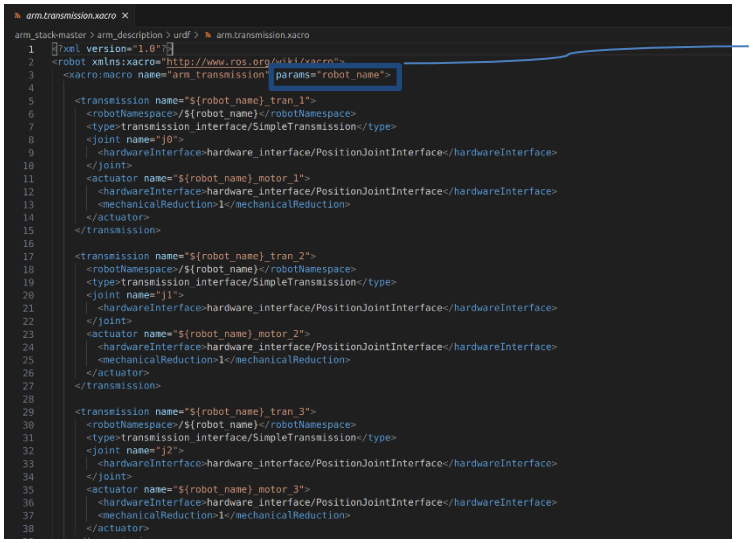
(b) Within this package create a launch folder containing a arm_world.launch file

mkdir launch

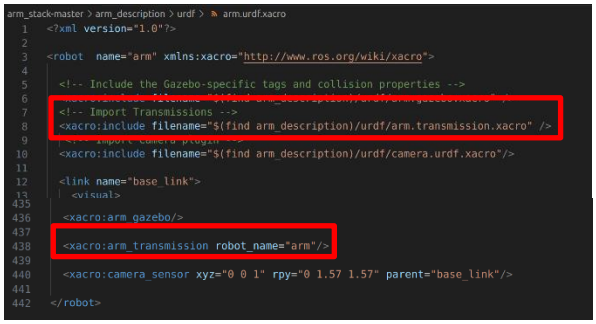
touch arm_world.launch

(c) Fill this launch file with commands that load the URDF into the ROS Parameter Server and spawn your robot using the spawn_model node. **Hint:** follow the iiwa_world.launch example from the package iiwa_stack: https://github.com/IFL-CAMP/iiwa_stack/tree/master. Launch the arm_world.launch file to visualize the robot in Gazebo

(d) 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.launch the file

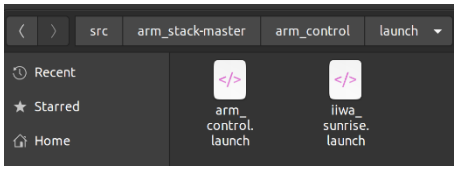


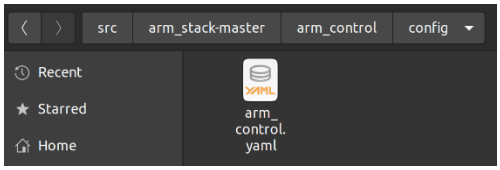
The parameter “robot_name” is not strictly necessary but has been inserted anyway.



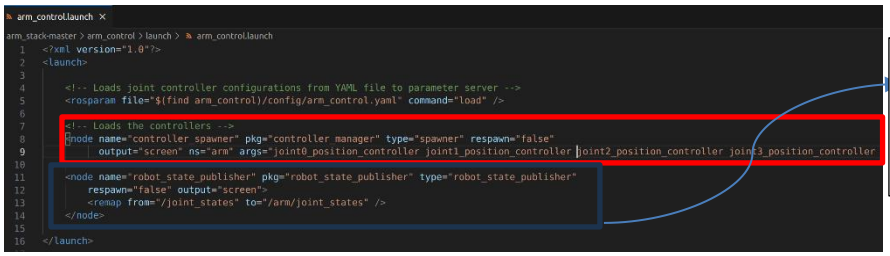
(e) Add joint position controllers to your robot: create a arm_control package with a arm_control.launch file inside its launch folder and a arm_control.yaml file within its config folder

catkin_create_pkg arm_control

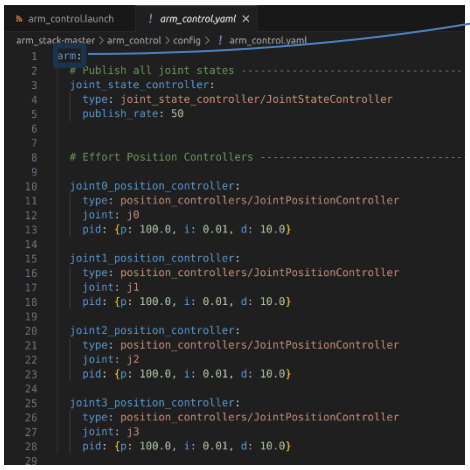




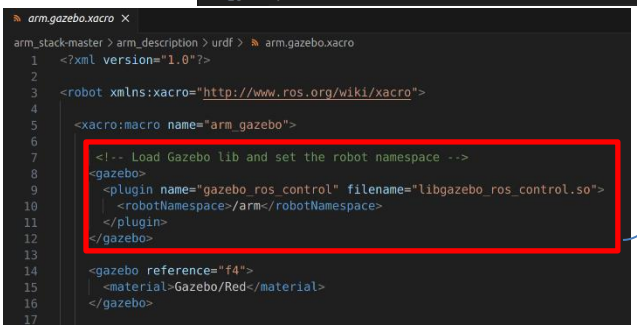
(f) Fill the arm_control.launch file with commands that load the joint controller configurations from the .yaml file to the parameter server and spawn the controllers using the controller_manager package. **Hint:** follow the iiwa_control.launch example from corresponding package



(g) Fill the arm arm_control.yaml adding a joint_state_controller and a JointPositionController to all the joints

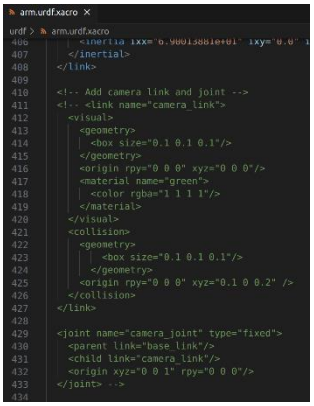


(h) Create an arm_gazebo.launch file into the launch folder of the arm_gazebo package loading the Gazebo world with arm_world.launch and spawning the controllers within arm_control.launch. Go to the arm_description package and add the gazebo_ros_control plugin to your main URDF into the arm.gazebo.xacro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



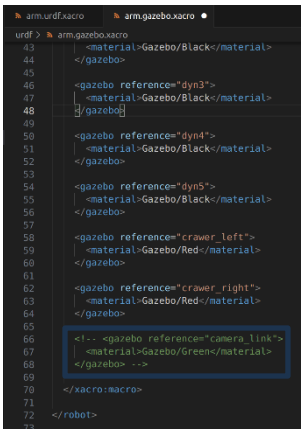
3. Add a camera sensor to your robot

(a) 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



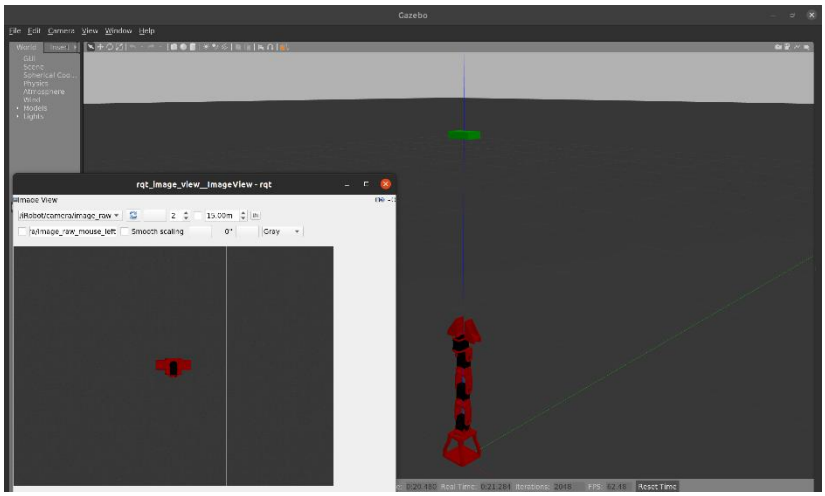
the code is commented out since in the final file I used camera.xacro as suggested in point (d)

(b) In the arm.gazebo.xacro add the gazebo sensor reference tags and the libgazebo_ros_camera plugin to your xacro (slide 74-75)

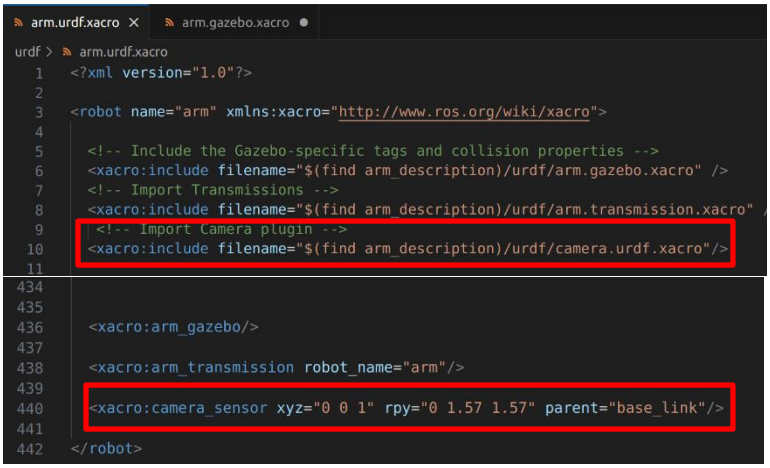


the code is commented out since in the final file I used camera.xacro as suggested in point (d)

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



(d) **Optionally:** You can create a camera.xacro file (or download one from https://github.com/CentroEPiaggio/irobotcreate2ros/blob/master/model/camera.urdf.xacro) and add it to your robot URDF using <xacro:include>



4. Create a ROS publisher node that reads the joint state and sends joint position commands to your robot

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

catkin_create_pkg arm_controller roscpp sensor_msgs std_msgs

touch src/arm_controller_node.cpp

dependencies

```
Activities  Applications  Places  Text Editor
Open
~/dev/catkin_ws/src
```

```
cmake_minimum_required(VERSION 3.0.2)
project(arm_controller)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  sensor_msgs
  std_msgs
)

## System dependencies are found with CMake's conventions
```

```
##################################
## catkin specific configuration ##
##################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES arm_controller
 CATKIN_DEPENDS roscpp sensor_msgs std_msgs
#  DEPENDS system_lib
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)
```

```
## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
add_executable(arm_controller_node src/arm_controller_node.cpp)
add_executable(arm_controller_publisher_node src/arm_controller_publisher_node.cpp)

## Rename C++ executable without prefix
## The above recommended prefix causes long target names, the following renames the
## target back to the shorter version for ease of user use
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_no
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catk

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

I preferred to create two distinct nodes, the first one showing the current values of the controllers and the second one publishing the values on the controllers

(b) Create a subscriber to the topic joint_states and a callback function that prints the current joint positions (see Slide 45). **Note:** the topic contains a sensor_msgs/JointState

arm_controller_node.cpp

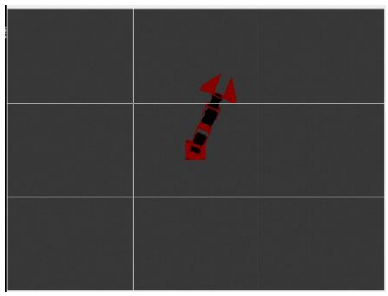
```cpp
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>

void jointStateCallback(const sensor_msgs::JointState::ConstPtr& jointState) {
    // Stampa le posizioni dei giunti
    for (size_t i = 0; i < jointState->position.size(); ++i) {
        ROS_INFO("Joint %zu: %f", i, jointState->position[i]);
    }
}

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

    // Crea un subscriber per il topic "joint_states"
    ros::Subscriber jointStateSub = nh.subscribe("/arm/joint_states", 100, jointStateCallback);

    ros::spin();  // Mantieni il nodo attivo

    return 0;
}
```

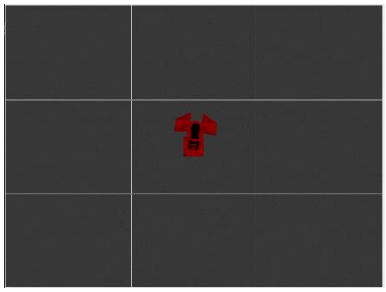
(c) Create publishers that write commands onto the controllers' /command topics (see Slide 46). **Note:** the command is a std_msgs/Float64

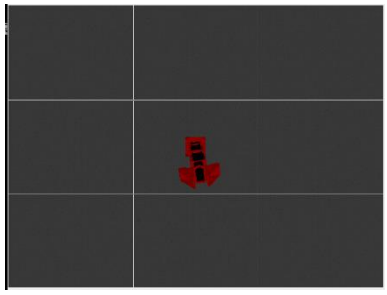
arm_controller_publisher_node.cpp

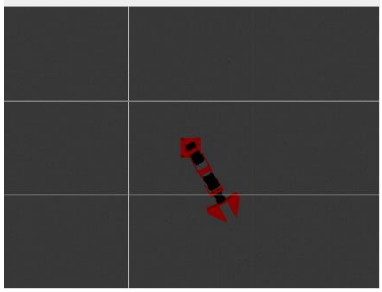
```cpp
#include <ros/ros.h>
#include "std_msgs/Float64.h"

int main(int argc, char* argv[]) {
    ros::init(argc, argv, "arm_controller_publisher_node");
    ros::NodeHandle nh;
    // Crea publisher per inviare comandi ai controller
    ros::Publisher joint0CommandPub = nh.advertise<std_msgs::Float64>("/arm/joint0_position_controller/command", 100);
    ros::Publisher joint1CommandPub = nh.advertise<std_msgs::Float64>("/arm/joint1_position_controller/command", 100);
    ros::Publisher joint2CommandPub = nh.advertise<std_msgs::Float64>("/arm/joint2_position_controller/command", 100);
    ros::Publisher joint3CommandPub = nh.advertise<std_msgs::Float64>("/arm/joint3_position_controller/command", 100);

    // Imposta la frequenza di aggiornamento (Hz) per la tua traiettoria
    double frequency = 10.0;  // Esempio: 1 Hz

    // Durata di un periodo della traiettoria (secondi)
    double period = 100.0 / frequency;

    // Valori di ampiezza massima e minima della traiettoria
    double max_amplitude = 10.0;
    double min_amplitude = -10.0;

    // Inizia dalla posizione minima
    double current_position = min_amplitude;
    bool increasing = true;

    ros::Rate loop_rate(frequency);

    while (ros::ok())
    {
        // Genera comandi di posizione a forma di onda triangolare
        std_msgs::Float64 position_command_j0;
        position_command_j0.data = current_position;
        // Pubblica il comando sul topic del controller del giunto
        joint0CommandPub.publish(position_command_j0);

        // Invio messaggi distinti anche se il contenuto è il medesimo per avere generalità
        std_msgs::Float64 position_command_j1;
        position_command_j1.data = current_position;
        joint1CommandPub.publish(position_command_j1);

        std_msgs::Float64 position_command_j2;
        position_command_j2.data = current_position;
        joint2CommandPub.publish(position_command_j2);

        std_msgs::Float64 position_command_j3;
        position_command_j3.data = current_position;
        joint3CommandPub.publish(position_command_j3);

        // Calcolo onda quadra
        if (increasing)
        {
            current_position += (2.0 * max_amplitude / period) * (1.0 / frequency);
            if (current_position >= max_amplitude)
            {
                current_position = max_amplitude;
                increasing = false;
            }
        }
        else
        {
            current_position -= (2.0 * max_amplitude / period) * (1.0 / frequency);
            if (current_position <= min_amplitude)
            {
                current_position = min_amplitude;
                increasing = true;
            }
        }

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

    return 0;
}
```

I chose an equal triangular wave trajectory for all joints, the result is shown below









```
[ INFO] [1698609018.744288193, 83.060000000]: Joint 3: -0.217000
[ INFO] [1698609018.764475164, 83.080000000]: Joint 0: -0.229000
[ INFO] [1698609018.764497860, 83.080000000]: Joint 1: -0.229000
[ INFO] [1698609018.764505745, 83.080000000]: Joint 2: -0.229000
[ INFO] [1698609018.764515998, 83.080000000]: Joint 3: -0.219000
[ INFO] [1698609018.784171624, 83.100000000]: Joint 0: -0.231000
[ INFO] [1698609018.784211170, 83.100000000]: Joint 1: -0.231000
[ INFO] [1698609018.784217394, 83.100000000]: Joint 2: -0.231000
[ INFO] [1698609018.784224302, 83.100000000]: Joint 3: -0.221000
[ INFO] [1698609018.804395286, 83.120000000]: Joint 0: -0.233000
[ INFO] [1698609018.804424130, 83.120000000]: Joint 1: -0.233000
[ INFO] [1698609018.804431804, 83.120000000]: Joint 2: -0.233000
[ INFO] [1698609018.804442010, 83.120000000]: Joint 3: -0.223000
[ INFO] [1698609018.824117969, 83.140000000]: Joint 0: -0.235000
[ INFO] [1698609018.824146270, 83.140000000]: Joint 1: -0.235000
[ INFO] [1698609018.824154577, 83.140000000]: Joint 2: -0.235000
[ INFO] [1698609018.824164386, 83.140000000]: Joint 3: -0.225000
[ INFO] [1698609018.844420466, 83.160000000]: Joint 0: -0.237000
[ INFO] [1698609018.844446001, 83.160000000]: Joint 1: -0.237000
[ INFO] [1698609018.844452717, 83.160000000]: Joint 2: -0.237000
[ INFO] [1698609018.844459299, 83.160000000]: Joint 3: -0.227000
```

```
    generate(dot_graph)
  File "/opt/ros/noetic/lib/tf/view_frames", line 89, in generate
    m = r.search(vstr)
TypeError: cannot use a string pattern on a bytes-like object
root@andrea-Surface-Pro-8:/home/dev/catkin_ws# rospack find tf
/opt/ros/noetic/share/tf
root@andrea-Surface-Pro-8:/home/dev/catkin_ws# rosrun tf tf_relay
[rosrun] Couldn't find executable named tf_relay below /opt/ros/noetic/share/t
f
root@andrea-Surface-Pro-8:/home/dev/catkin_ws# rqt_image_view
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
^Croot@andrea-Surface-Pro-8:/home/dev/catkin_ws# rosrun arm_
arm_control      arm_controller   arm_description  arm_gazebo
root@andrea-Surface-Pro-8:/home/dev/catkin_ws# rosrun arm_controller arm_contr
oller_
arm_controller_node           arm_controller_publisher_node
root@andrea-Surface-Pro-8:/home/dev/catkin_ws# rosrun arm_controller arm_contr
oller_
arm_controller_node           arm_controller_publisher_node
root@andrea-Surface-Pro-8:/home/dev/catkin_ws# rosrun arm_controller arm_contr
oller_publisher_node
```

arm_controller_node

arm_controller_publisher_node