Report – Homework 3

Students:

Luca Marseglia

https://github.com/marseluca/homework3rl.git

Benito Vodola

https://github.com/BenitoVodola/Homework3.git

Domenico Tuccillo

https://github.com/DomenicoTuccillo/homework3.git

Debora Ippolito

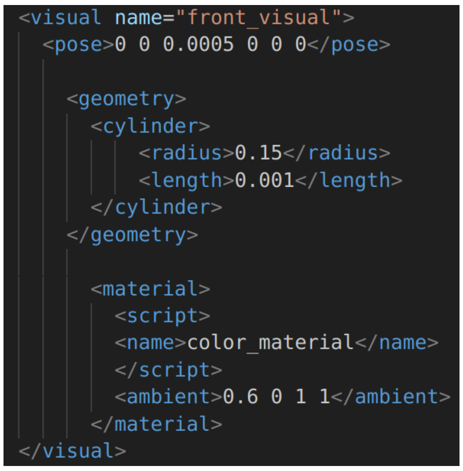
https://github.com/DeboraIppolito/homework3.git

1. Construct a gazebo world inserting a circular object and detect it via the opencv_ros package

(a) Create a new model named circular_object that represents a 15 cm radius colored circular object and import it into a new gazebo world as a static object at x=1, y=-0.5, z = 0.6 (orient it suitably to accomplish the next point). Save the new world into the /iiwa_gazebo/worlds/ folder.

We created a new .sdf file named circular_object.sdf in which we implemented the model of a 15cm radius, 0.1 cm length purple cylinder.

In file "iiwa_gazebo/models/circular_object/circular_object.sdf":

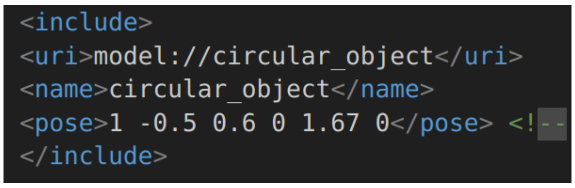


```
<visual name="front_visual">
  <pose>0 0 0.0005 0 0 0</pose>

    <geometry>
      <cylinder>
          <radius>0.15</radius>
          <length>0.001</length>
      </cylinder>
    </geometry>

      <material>
        <script>
        <name>color_material</name>
        </script>
        <ambient>0.6 0 1 1</ambient>
      </material>
</visual>
```

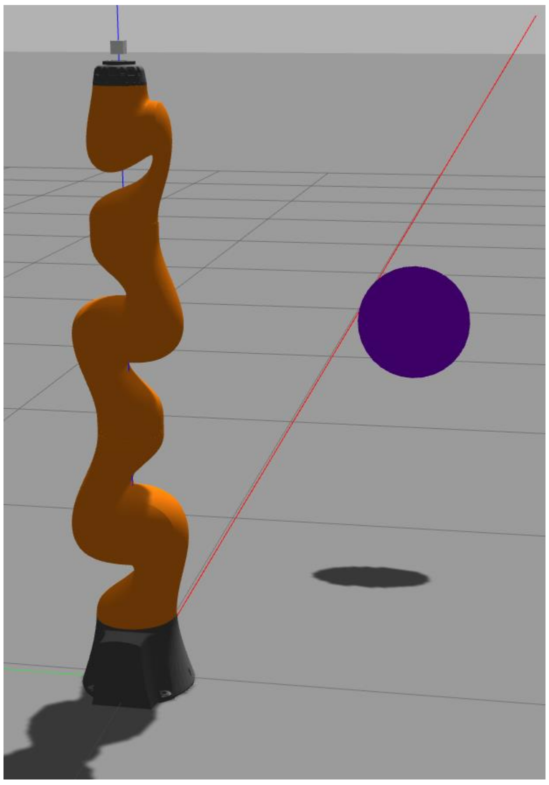
The same dimensions apply to "rear_visual" and "collision".

We then included it in a new world file named iiwa_circular.world in the desired position:

x=1, y=-0.5, z = 0.6;

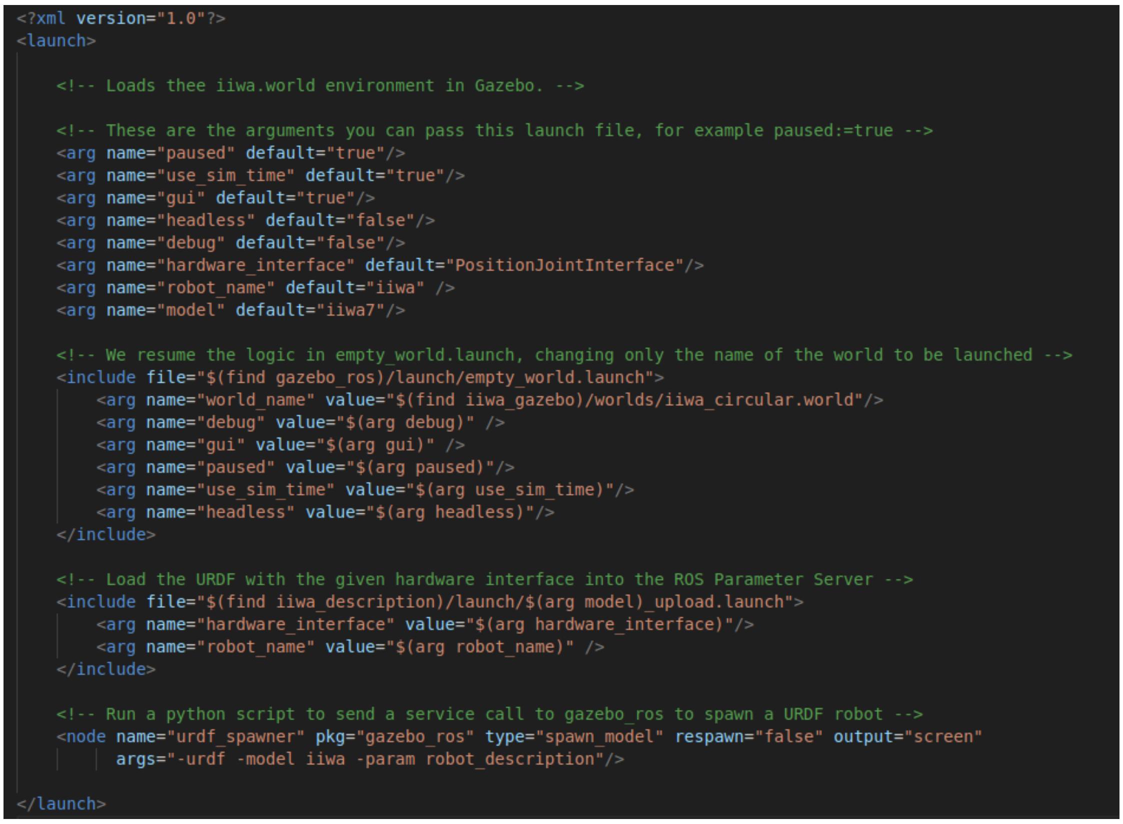


```
<include>
<uri>model://circular_object</uri>
<name>circular_object</name>
<pose>1 -0.5 0.6 0 1.67 0</pose> <!--
</include>
```



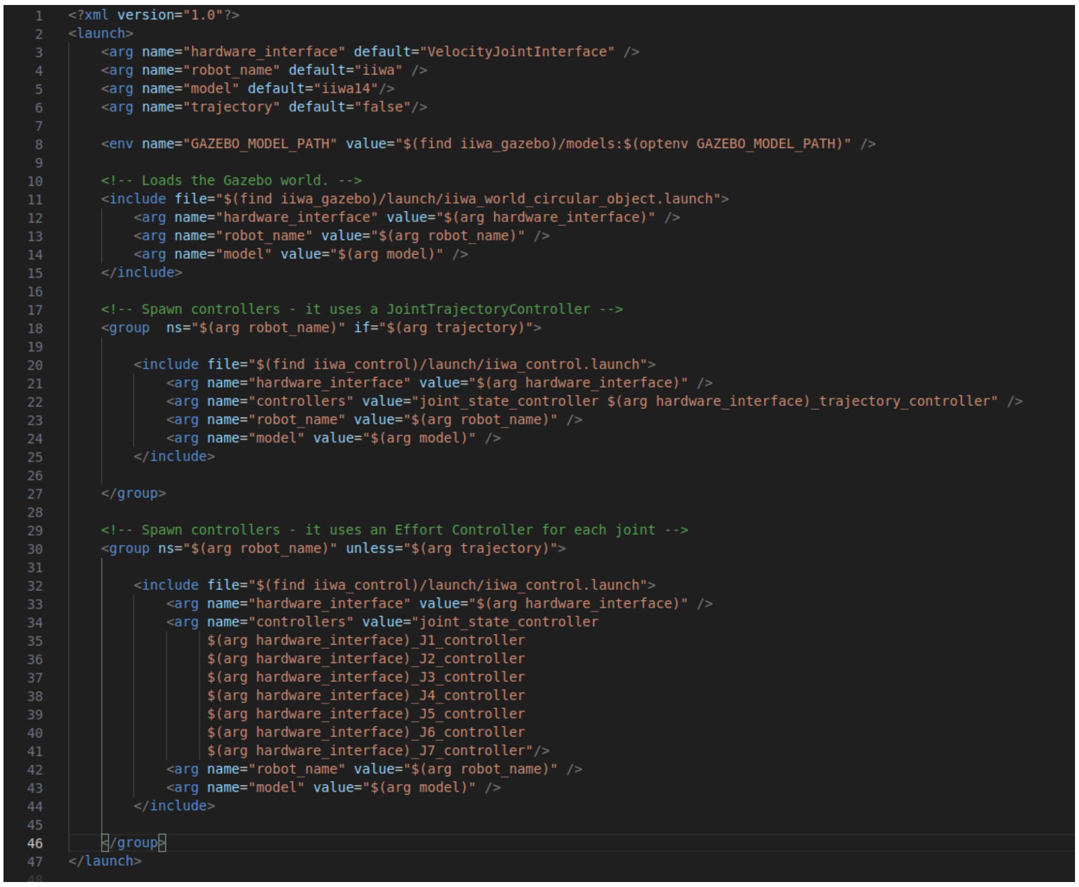
(b) Create a new launch file named launch/iiwa_gazebo_circular_object.launch that loads the iiwa robot with PositionJointInterface equipped with the camera into the new world via a launch/iiwa_world_circular_object.launch file. Make sure the robot sees the imported object with the camera, otherwise modify its configuration.

We first created the iiwa_world_circular_object.launch file to load the iiwa_circular.world in gazebo:



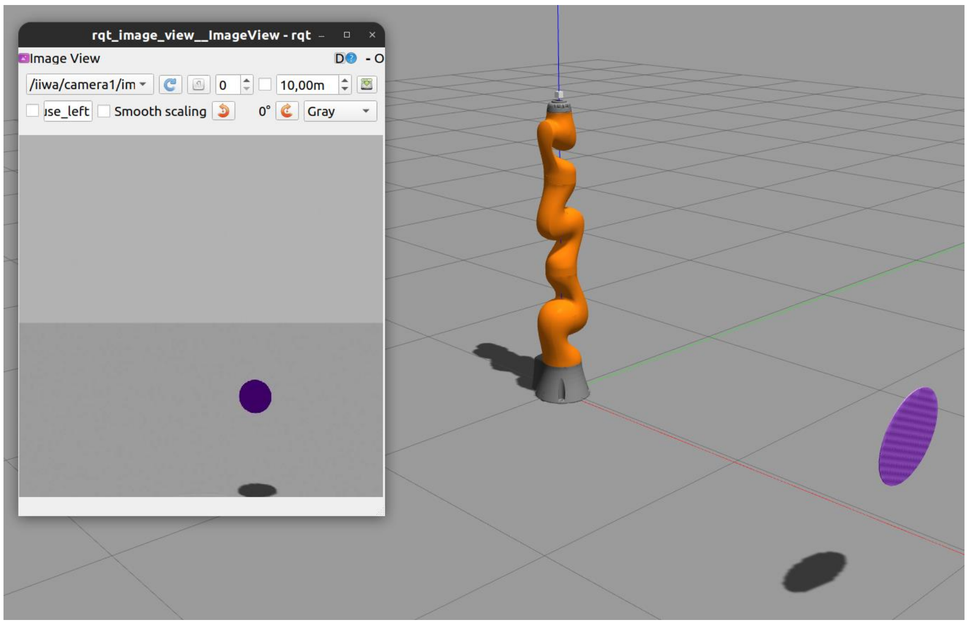
```xml
<?xml version="1.0"?>
<launch>

    <!-- Loads thee iiwa.world environment in Gazebo. -->

    <!-- These are the arguments you can pass this launch file, for example paused:=true -->
    <arg name="paused" default="true"/>
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>
    <arg name="hardware_interface" default="PositionJointInterface"/>
    <arg name="robot_name" default="iiwa" />
    <arg name="model" default="iiwa7"/>

    <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find iiwa_gazebo)/worlds/iiwa_circular.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="$(find iiwa_description)/launch/$(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 iiwa -param robot_description"/>

</launch>
```

We then imported it into the iiwa_gazebo_circular_object.launch file that loads the robot and the controllers:



```xml
<?xml version="1.0"?>
<launch>
    <arg name="hardware_interface" default="VelocityJointInterface" />
    <arg name="robot_name" default="iiwa" />
    <arg name="model" default="iiwa14"/>
    <arg name="trajectory" default="false"/>

    <env name="GAZEBO_MODEL_PATH" value="$(find iiwa_gazebo)/models:$(optenv GAZEBO_MODEL_PATH)" />

    <!-- Loads the Gazebo world. -->
    <include file="$(find iiwa_gazebo)/launch/iiwa_world_circular_object.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)" />
        <arg name="robot_name" value="$(arg robot_name)" />
        <arg name="model" value="$(arg model)" />
    </include>

    <!-- Spawn controllers - it uses a JointTrajectoryController -->
    <group  ns="$(arg robot_name)" if="$(arg trajectory)">

        <include file="$(find iiwa_control)/launch/iiwa_control.launch">
            <arg name="hardware_interface" value="$(arg hardware_interface)" />
            <arg name="controllers" value="joint_state_controller $(arg hardware_interface)_trajectory_controller" />
            <arg name="robot_name" value="$(arg robot_name)" />
            <arg name="model" value="$(arg model)" />
        </include>

    </group>

    <!-- Spawn controllers - it uses an Effort Controller for each joint -->
    <group ns="$(arg robot_name)" unless="$(arg trajectory)">

        <include file="$(find iiwa_control)/launch/iiwa_control.launch">
            <arg name="hardware_interface" value="$(arg hardware_interface)" />
            <arg name="controllers" value="joint_state_controller
                 $(arg hardware_interface)_J1_controller
                 $(arg hardware_interface)_J2_controller
                 $(arg hardware_interface)_J3_controller
                 $(arg hardware_interface)_J4_controller
                 $(arg hardware_interface)_J5_controller
                 $(arg hardware_interface)_J6_controller
                 $(arg hardware_interface)_J7_controller"/>
            <arg name="robot_name" value="$(arg robot_name)" />
            <arg name="model" value="$(arg model)" />
        </include>

    </group>
</launch>
```

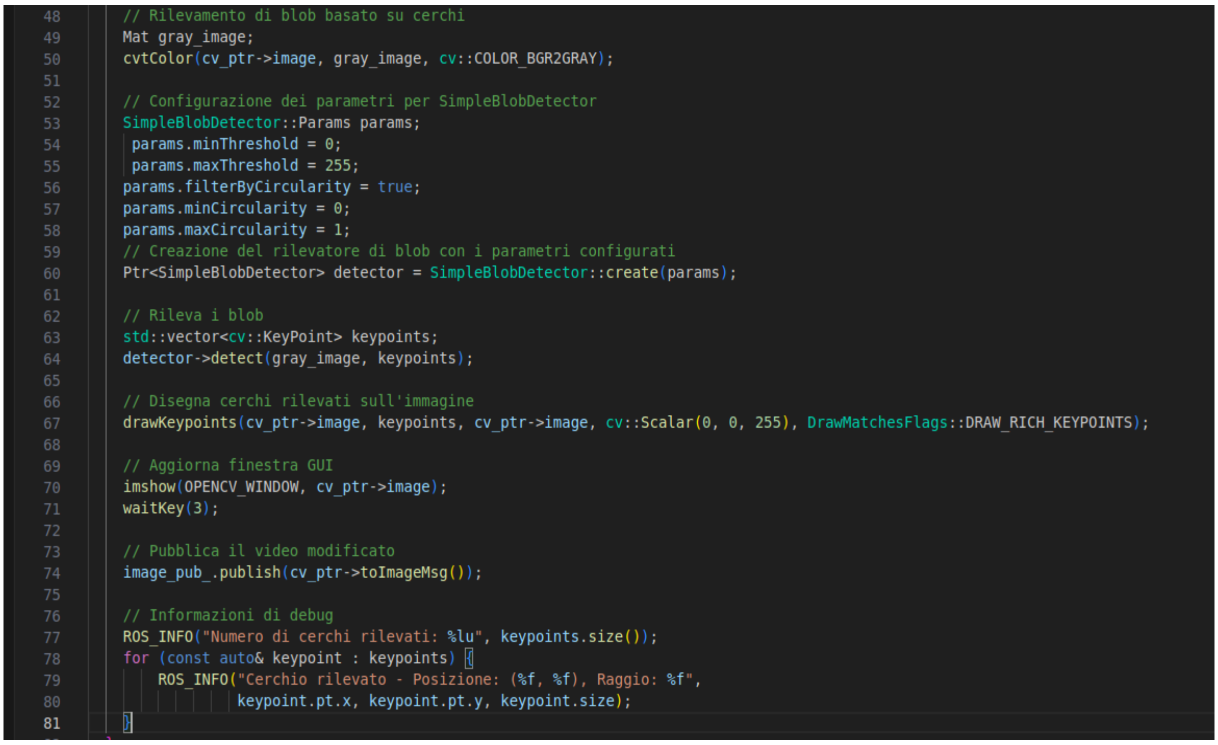
We set a different pose with *x=2* due to the distance detecting problem of camera: if the circular object is too close to the camera, it cannot detect the object.



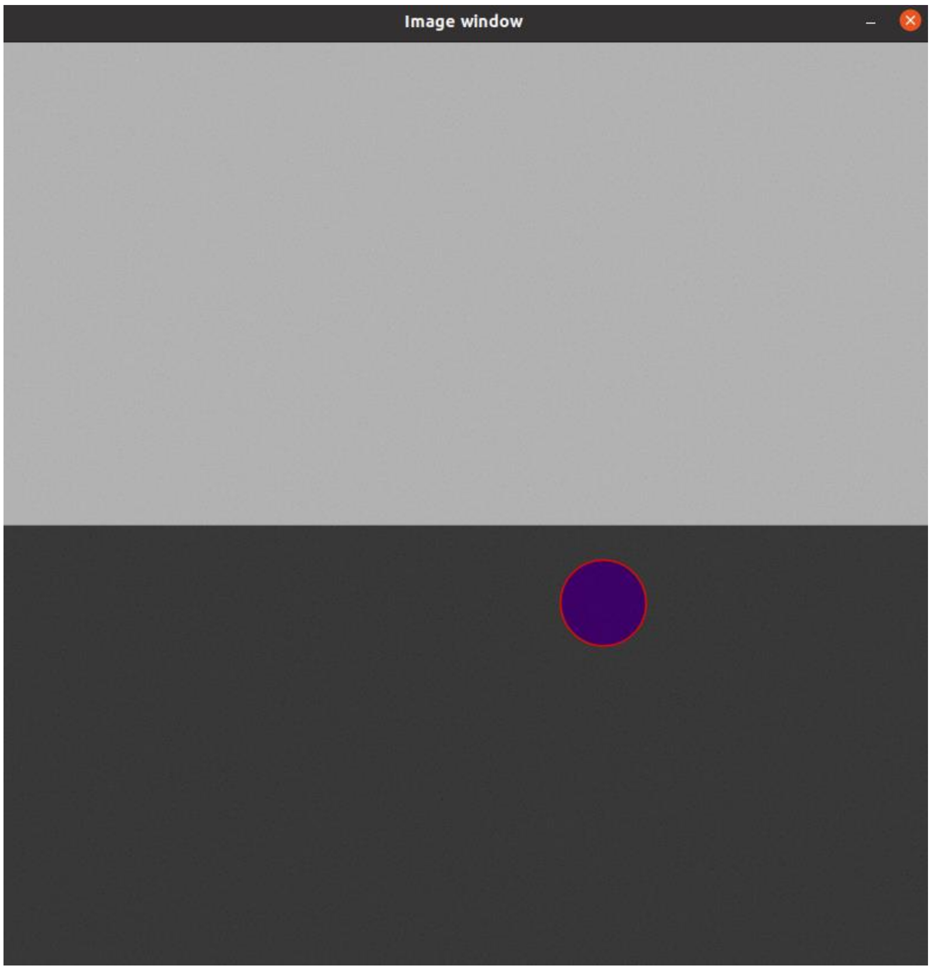
(c) Once the object is visible in the camera image, use the opencv_ros/ package to detect the circular object using open CV functions. Modify the opencv_ros_node.cpp to subscribe to the simulated image, detect the object via openCV functions 1, and republish the processed image.

We used the simpleBlobDetector function to detect the circular object. To draw a circle around the detected blob, we must:

1) Define a Mat type variable to convert the image to the openCV format;
2) Convert an image from one color space to another it is necessary to use the function cvtColor;
3) Define the parameters to pass to the creator of detector;
4) Define a vector of keypoints;
5) Update the window and publish the message.



```
// Rilevamento di blob basato su cerchi
Mat gray_image;
cvtColor(cv_ptr->image, gray_image, cv::COLOR_BGR2GRAY);

// Configurazione dei parametri per SimpleBlobDetector
SimpleBlobDetector::Params params;
 params.minThreshold = 0;
 params.maxThreshold = 255;
params.filterByCircularity = true;
params.minCircularity = 0;
params.maxCircularity = 1;
// Creazione del rilevatore di blob con i parametri configurati
Ptr<SimpleBlobDetector> detector = SimpleBlobDetector::create(params);

// Rileva i blob
std::vector<cv::KeyPoint> keypoints;
detector->detect(gray_image, keypoints);

// Disegna cerchi rilevati sull'immagine
drawKeypoints(cv_ptr->image, keypoints, cv_ptr->image, cv::Scalar(0, 0, 255), DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

// Aggiorna finestra GUI
imshow(OPENCV_WINDOW, cv_ptr->image);
waitKey(3);

// Pubblica il video modificato
image_pub_.publish(cv_ptr->toImageMsg());

// Informazioni di debug
ROS_INFO("Numero di cerchi rilevati: %lu", keypoints.size());
for (const auto& keypoint : keypoints) {
    ROS_INFO("Cerchio rilevato - Posizione: (%f, %f), Raggio: %f",
             keypoint.pt.x, keypoint.pt.y, keypoint.size);
}
```



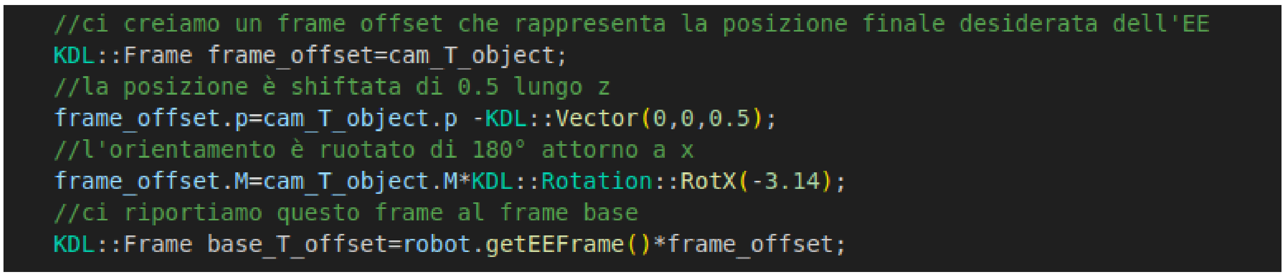
2. Modify the look-at-point vision-based control example

(a) Modify the kdl_robot_vision_control node to implement a vision-based task that aligns the camera to the aruco marker with an appropiately chosen position and orientation offsets. Show the tracking capability by moving the aruco marker via the interface and plotting the velocity commands sent to the robot.

The original version of the look-at-point controller tried to align the z-axis of the camera to the distance vector between the camera and the centre of the aruco marker. In this way the camera was always looking in the direction of the marker.

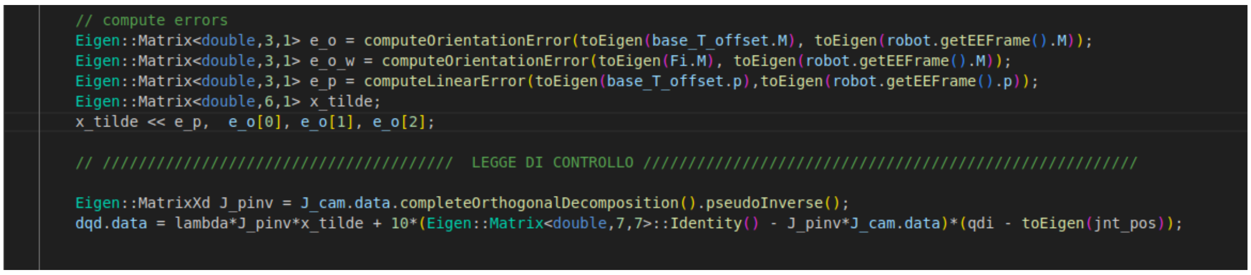
Now we implement a controller that makes the camera look at the marker always from the same point of view. Specifically, we always want to look at the marker from the front and from the same distance.

We first created a new frame called offset_frame representing the desired camera pose with respect to the marker. Starting from the aruco frame, the offset_frame is shifted in the z direction of 0.5 meters and rotated of 180° around the x-axis in clockwise. Then, we brought this frame back to the base frame.



```
//ci creiamo un frame offset che rappresenta la posizione finale desiderata dell'EE
KDL::Frame frame_offset=cam_T_object;
//la posizione è shiftata di 0.5 lungo z
frame_offset.p=cam_T_object.p -KDL::Vector(0,0,0.5);
//l'orientamento è ruotato di 180° attorno a x
frame_offset.M=cam_T_object.M*KDL::Rotation::RotX(-3.14);
//ci riportiamo questo frame al frame base
KDL::Frame base_T_offset=robot.getEEFrame()*frame_offset;
```

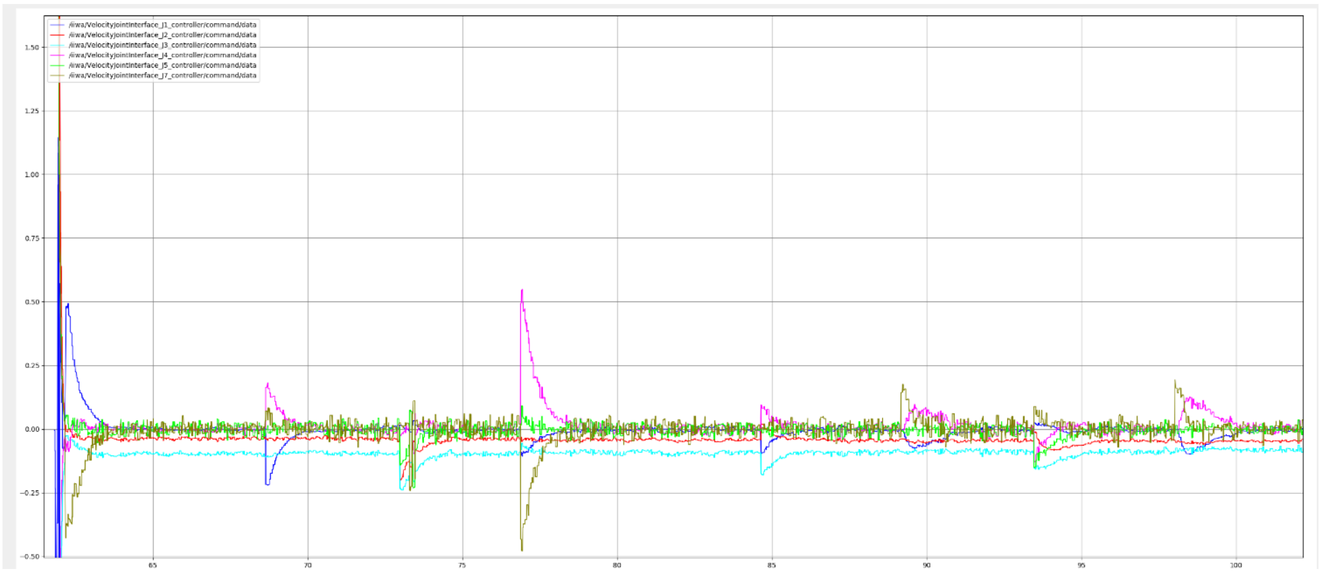
Then, we aligned the end-effector frame with the offset frame computing the orientation and position errors:



```
// compute errors
Eigen::Matrix<double,3,1> e_o = computeOrientationError(toEigen(base_T_offset.M), toEigen(robot.getEEFrame().M));
Eigen::Matrix<double,3,1> e_o_w = computeOrientationError(toEigen(Fi.M), toEigen(robot.getEEFrame().M));
Eigen::Matrix<double,3,1> e_p = computeLinearError(toEigen(base_T_offset.p),toEigen(robot.getEEFrame().p));
Eigen::Matrix<double,6,1> x_tilde;
x_tilde << e_p,  e_o[0], e_o[1], e_o[2];

// ////////////////////////////////////////  LEGGE DI CONTROLLO ////////////////////////////////////////////////////////

Eigen::MatrixXd J_pinv = J_cam.data.completeOrthogonalDecomposition().pseudoInverse();
dqd.data = lambda*J_pinv*x_tilde + 10*(Eigen::Matrix<double,7,7>::Identity() - J_pinv*J_cam.data)*(qdi - toEigen(jnt_pos));
```

In such way the desired pose is set equal to the one of the aruco marker with respect to the camera frame, but shifted toward the camera of 0.5 m and rotated to make the camera to point toward the marker, the other axes are also oriented in such way that the camera always look at the marker in the same orientation even when the latter is rotating. This is what we've looking for, if it's desired a different behaviour by keeping the camera always pointed upwards, one can simply substitute the first component of the orientation error with the first component of the e_o_w error when giving the error to the variable x_tilde.

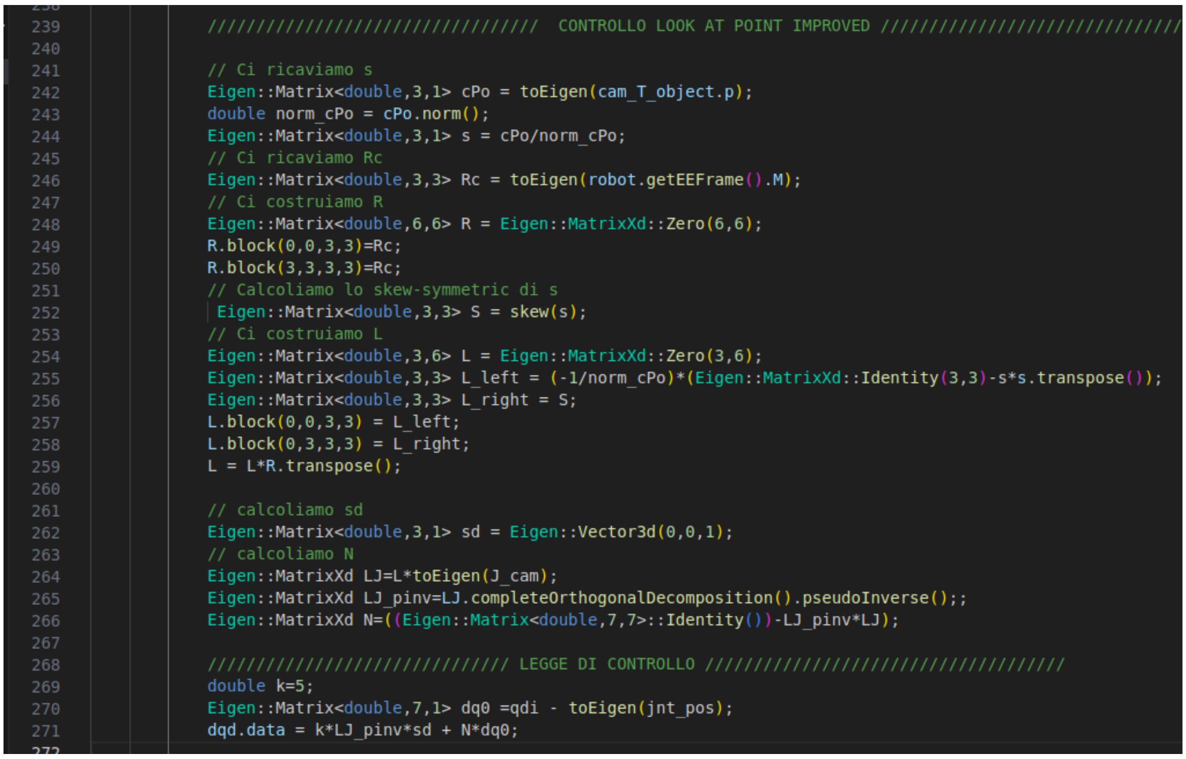
The resulting velocity commands sent to the robot are shown below:



Note that the spikes in the velocity are placed in time when the pose of the marker has been modified.
In the repos it is possible to appreciate two video of this simulation, the first one is “2.a_gazebo” and shows the tracking capability of the manipulator by moving and rotating in all possible ways the aruco marker, while in the video called “2.a_cam” it is possible to appreciate how the camera stays perfectly alligned with the aruco marker and always at the same selected distance.

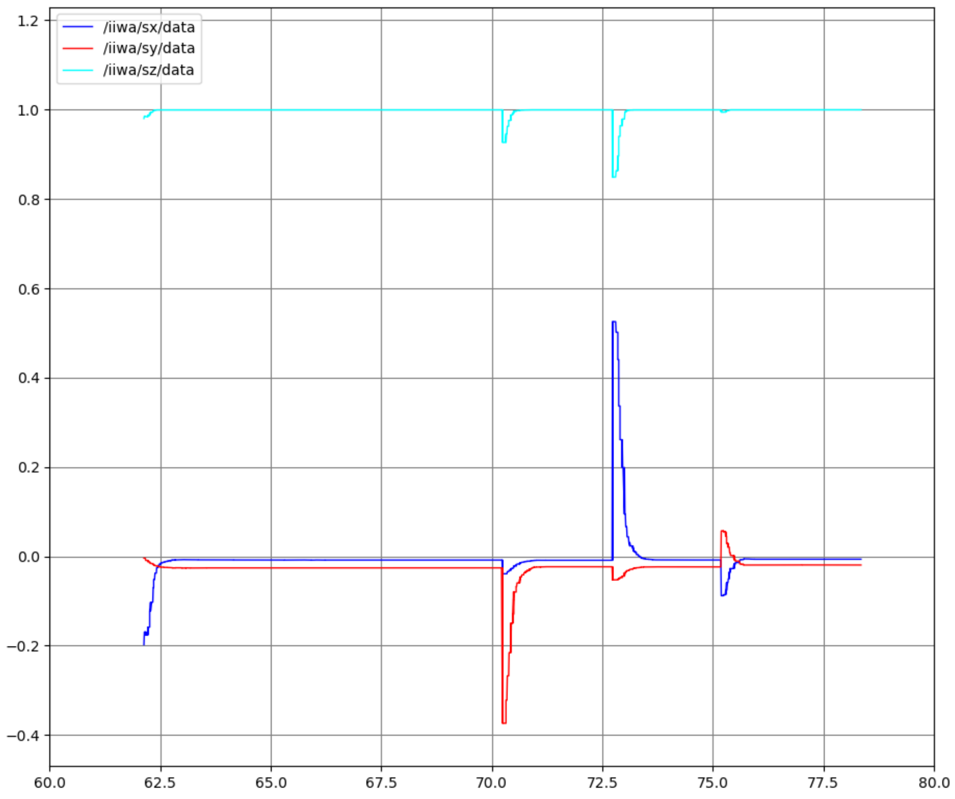
(b) An improved look-at-point algorithm can be devised by noticing that the task is belonging to $\mathbb{S}^2$.

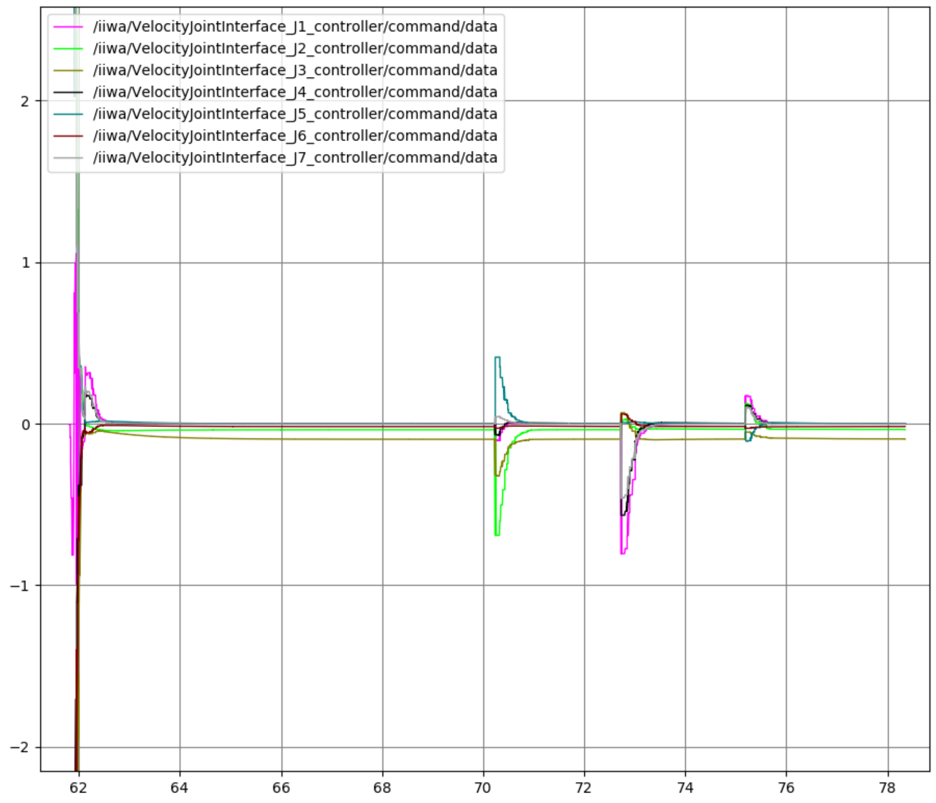
We simply implemented the given formula:



```
//////////////////////////////////  CONTROLLO LOOK AT POINT IMPROVED ////////////////////////////////

// Ci ricaviamo s
Eigen::Matrix<double,3,1> cPo = toEigen(cam_T_object.p);
double norm_cPo = cPo.norm();
Eigen::Matrix<double,3,1> s = cPo/norm_cPo;
// Ci ricaviamo Rc
Eigen::Matrix<double,3,3> Rc = toEigen(robot.getEEFrame().M);
// Ci costruiamo R
Eigen::Matrix<double,6,6> R = Eigen::MatrixXd::Zero(6,6);
R.block(0,0,3,3)=Rc;
R.block(3,3,3,3)=Rc;
// Calcoliamo lo skew-symmetric di s
 Eigen::Matrix<double,3,3> S = skew(s);
// Ci costruiamo L
Eigen::Matrix<double,3,6> L = Eigen::MatrixXd::Zero(3,6);
Eigen::Matrix<double,3,3> L_left = (-1/norm_cPo)*(Eigen::MatrixXd::Identity(3,3)-s*s.transpose());
Eigen::Matrix<double,3,3> L_right = S;
L.block(0,0,3,3) = L_left;
L.block(0,3,3,3) = L_right;
L = L*R.transpose();

// calcoliamo sd
Eigen::Matrix<double,3,1> sd = Eigen::Vector3d(0,0,1);
// calcoliamo N
Eigen::MatrixXd LJ=L*toEigen(J_cam);
Eigen::MatrixXd LJ_pinv=LJ.completeOrthogonalDecomposition().pseudoInverse();;
Eigen::MatrixXd N=((Eigen::Matrix<double,7,7>::Identity())-LJ_pinv*LJ);

////////////////////////////// LEGGE DI CONTROLLO ////////////////////////////////////
double k=5;
Eigen::Matrix<double,7,1> dq0 =qdi - toEigen(jnt_pos);
dqd.data = k*LJ_pinv*sd + N*dq0;
```

Here follow the graphs of joint velocity and s components:



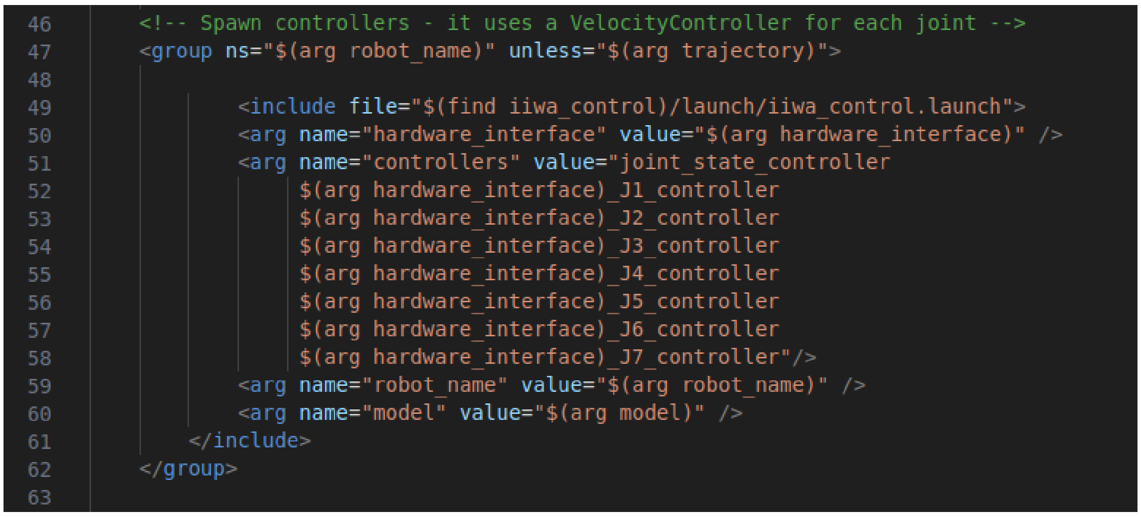


To appreciate the results of previous graphs, you will find a video on github showing how the camera follows the aruco marker moving. The s tracks almost everywhere the desired one s_d= [0,0,1], it is a little bit different from it only when we moved the marker, so it's necessary some recovery time to return to regime.

(c) Develop a dynamic version of the vision-based contoller. Track the reference velocities generated by the look-at-point vision-based control law with the joint space and the Cartesian space inverse dynamics controllers developed in the previous homework. To this end, you have to merge the two controllers and enable the joint tracking of a linear position trajectory and the vision-based task.

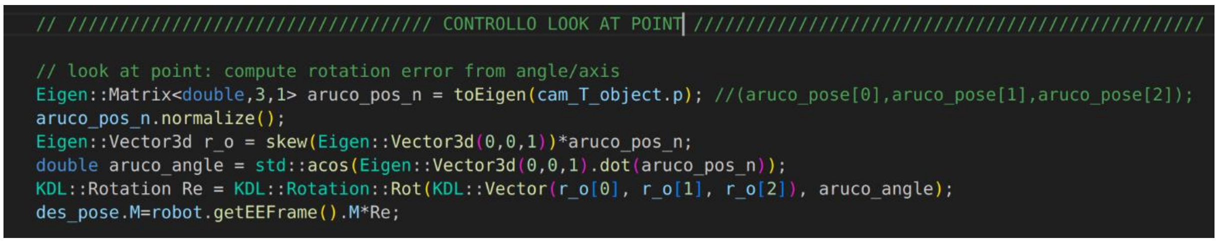
We modified the code from the previous homework adding the aruco topics, the camera frame and the look-at-point algorithm.

To be able to run the code we then created a new launch file called iiwa_gazebo_effort_aruco.launch which spawns the same gazebo world of the previous points but it uses the effort controllers.



```
<!-- Spawn controllers - it uses a VelocityController for each joint -->
<group ns="$(arg robot_name)" unless="$(arg trajectory)">

        <include file="$(find iiwa_control)/launch/iiwa_control.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)" />
        <arg name="controllers" value="joint_state_controller
             $(arg hardware_interface)_J1_controller
             $(arg hardware_interface)_J2_controller
             $(arg hardware_interface)_J3_controller
             $(arg hardware_interface)_J4_controller
             $(arg hardware_interface)_J5_controller
             $(arg hardware_interface)_J6_controller
             $(arg hardware_interface)_J7_controller"/>
        <arg name="robot_name" value="$(arg robot_name)" />
        <arg name="model" value="$(arg model)" />
    </include>
</group>
```

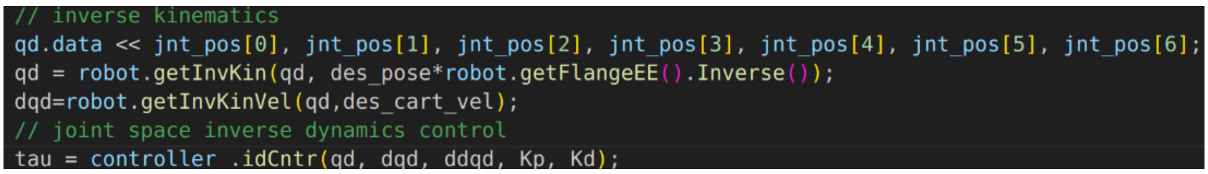
In the main we added the part of code which evaluates the desired orientation for the look at point task, while the desired position is one between the four trajectories created for the previous homework.



```
// ////////////////////////////////// CONTROLLO LOOK AT POINT ////////////////////////////////////////////////

// look at point: compute rotation error from angle/axis
Eigen::Matrix<double,3,1> aruco_pos_n = toEigen(cam_T_object.p); //(aruco_pose[0],aruco_pose[1],aruco_pose[2]);
aruco_pos_n.normalize();
Eigen::Vector3d r_o = skew(Eigen::Vector3d(0,0,1))*aruco_pos_n;
double aruco_angle = std::acos(Eigen::Vector3d(0,0,1).dot(aruco_pos_n));
KDL::Rotation Re = KDL::Rotation::Rot(KDL::Vector(r_o[0], r_o[1], r_o[2]), aruco_angle);
des_pose.M=robot.getEEFrame().M*Re;
```

We decided to control the manipulator with the joint space inverse dynamics for the tracking of a linear trajectory created with a cubic polynomial, while with the cartesian space inverse dynamics controller we chose to track a circular trajectory constructed with a trapezoidal profile of velocity.

In order to assign the tracking of the marker to the orientation of the camera we just assigned the resulting Re matrix to the M field of the structure des_pos, opportunely referred to the base frame.

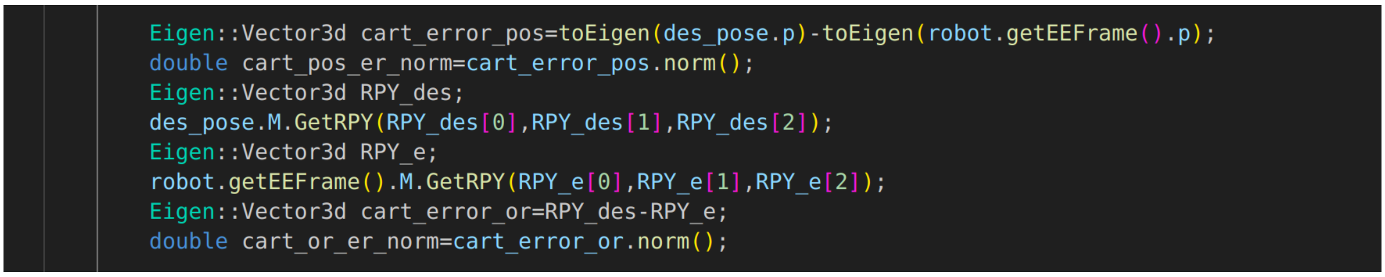


```
// inverse kinematics
qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3], jnt_pos[4], jnt_pos[5], jnt_pos[6];
qd = robot.getInvKin(qd, des_pose*robot.getFlangeEE().Inverse());
dqd=robot.getInvKinVel(qd,des_cart_vel);
// joint space inverse dynamics control
tau = controller .idCntr(qd, dqd, ddqd, Kp, Kd);
```

Since the joint space controller needs a reference into the joint space, it was necessary to use some inverse kinematics solver and functions, however we understood that the function CrtToJnt, employed by getInvKin does not take into account the added camera, so before passing des_pose to the joint reference we had to report it to the previous frame by multiplying it by the inverse of the homogenous transformation matrix between the end effector and the camera.

Clearly this was not necessary for the cartesian space controllers which take as input directly the desired pose. In order to achieve the same behaviour of the look-at-point task we also modified the controller implemented in the previous homework, simply by adding an orientation error component e_o_w to keep the camera pointed upwards like we previously discussed in the point 2.a.

Finally, it was necessary to plot the error of position and orientation, but since the latter is provided by rotation matrices it was necessary to convert it to something else and we chose to compute the corresponding RPY angles for them with a function of KDL, then we computed the norm of the difference.



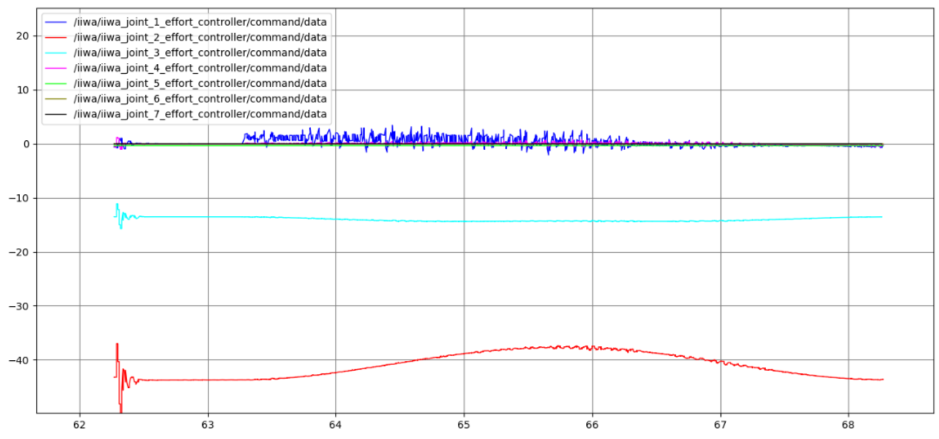
```
Eigen::Vector3d cart_error_pos=toEigen(des_pose.p)-toEigen(robot.getEEFrame().p);
double cart_pos_er_norm=cart_error_pos.norm();
Eigen::Vector3d RPY_des;
des_pose.M.GetRPY(RPY_des[0],RPY_des[1],RPY_des[2]);
Eigen::Vector3d RPY_e;
robot.getEEFrame().M.GetRPY(RPY_e[0],RPY_e[1],RPY_e[2]);
Eigen::Vector3d cart_error_or=RPY_des-RPY_e;
double cart_or_er_norm=cart_error_or.norm();
```

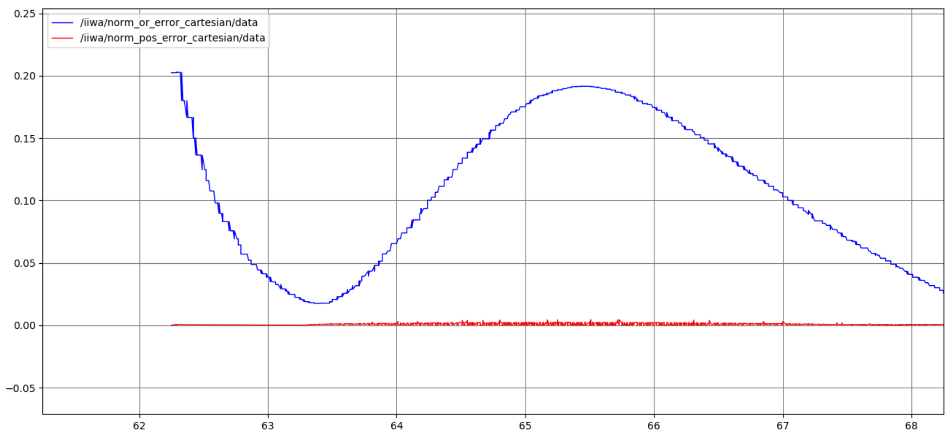
Then we plotted joint torques and the norm of the position and orientation error for:

- The cubic linear trajectory with the inverse dynamics controller in the joint space
- The trapezoidal circular trajectory with the inverse dynamics controller in the cartesian space

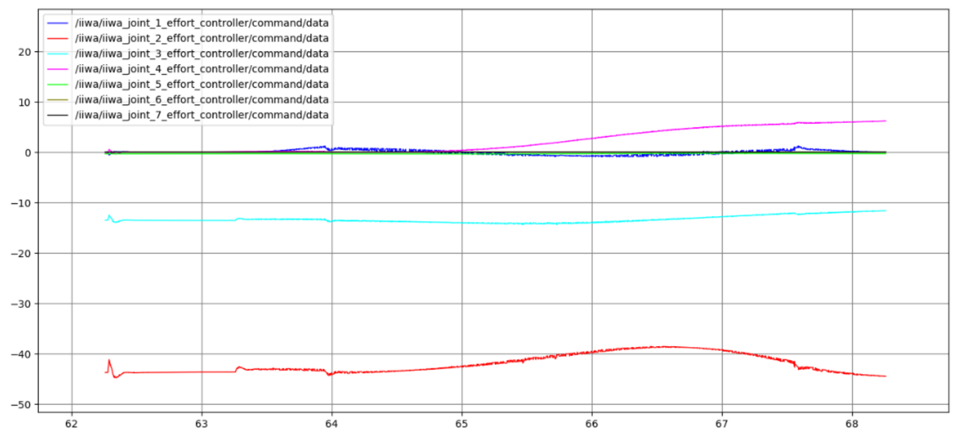
The error is very close to zero both for position and orientation, where it goes under 0.05 radians at the end of both trajectories.

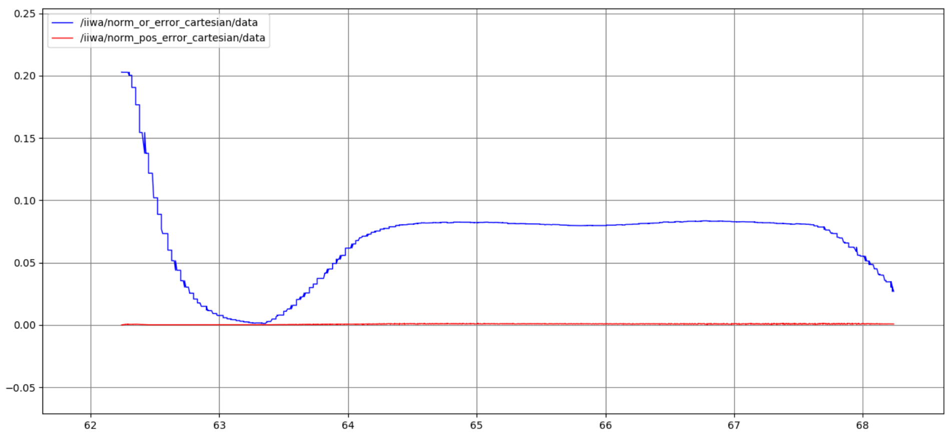
Cubic Linear trajectory with the joint space controller (gains: 150, 72)





Trapezoidal Circular with the controller in the Cartesian space (gains: 100, 40, 30)


/iiwa/iiwa_joint_1_effort_controller/command/data
/iiwa/iiwa_joint_2_effort_controller/command/data
/iiwa/iiwa_joint_3_effort_controller/command/data
/iiwa/iiwa_joint_4_effort_controller/command/data
/iiwa/iiwa_joint_5_effort_controller/command/data
/iiwa/iiwa_joint_6_effort_controller/command/data
/iiwa/iiwa_joint_7_effort_controller/command/data


/iiwa/norm_or_error_cartesian/data
/iiwa/norm_pos_error_cartesian/data