Robotics Lab: Homework 3

Students: Anzalone Claudio, Maisto Paolo, Manzoni Antonio

Here is the link to my public repo on github:
https://github.com/PaoloMaisto/HW3_RL_Maisto_Paolo.git

We want to specify that all the participants of the group worked at each stage of the development of the project. In order to simplify the drafting of the report (as recommended by the professor) we have fairly divided the writing of the development of the various points.

Implement a vision-based task

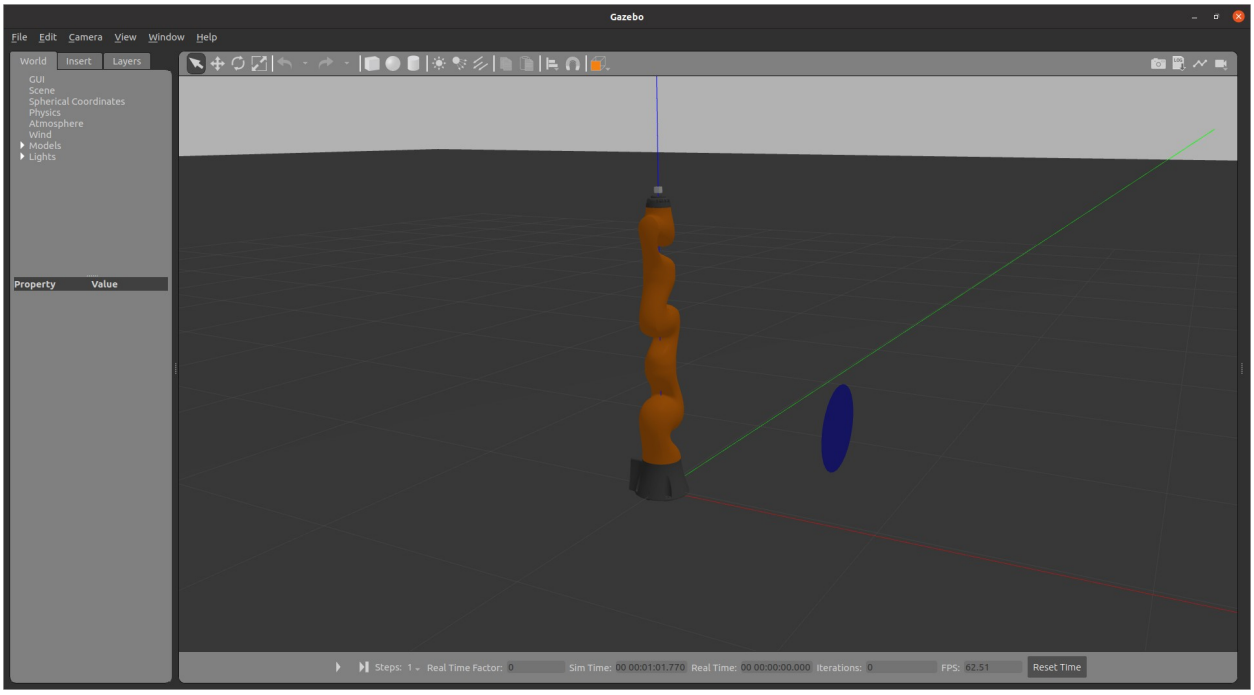
1. **Construct a gazebo world inserting a circular object and detect it via the opencv_ros package**

   (a) Go into the iiwa_gazebo package of the iiwa_stack. There you will find a folder models containing the aruco marker model for gazebo. Taking inspiration from this, 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.

Taking inspiration from the aruco marker model for gazebo, the circular object has been created in the path "*iiwa_gazebo/models*", within which the circular object is defined in the "*model.sdf*" file as a cylinder with a thickness of 0.0001 and a blue color with the function "*<ambient>*". It is declared with the name "Circular object" in the "*model.config*" file. Then to display it in our world the "*iiwa_circular_object.world*" file is created within which to the circular object is given the position given by the track and the following orientation: roll 0, pitch -1,57 and yaw 0 to accomplish the next point.

Used commands

```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_circular_object.launch
```

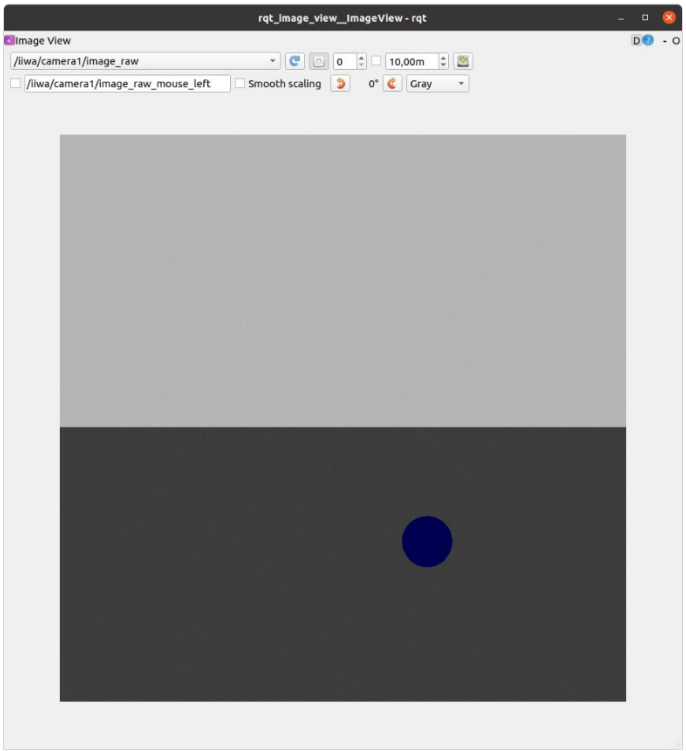


(b) Create a new launch file named launch/iiwa_gazebo_circular_object.launch that loads the iiwa robot with PositionJointInterface equipped with the camera into the new world via a launch/iiwa_world_circular_object.launch file. Make sure the robot sees the imported object with the camera, otherwise modify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

In this step two files ".*launch*" have been created, that is "*iiwa_gazebo_circular_object.launch*" and "*iiwa_world_circular_object.launch*" in the following path "*iiwa_gazebo/launch*". The first loads the iiwa robot with PositionJointInterface equipped with the camera and the "*iiwa_world_circular_object.launch*" file; while the second loads the world created in the previous step, so the "*iiwa_circular_object.world*" file. But the camera does not see the inserted object at all, so the pose and the orientation of the circular object are modified in order to see the object as shown below (only the x coordinate has been changed and it is x=2).
To verify that the robot sees the imported circular object with the camera are used the following commands:

Used commands

```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_circular_object.launch
$ rqt_image_view
```

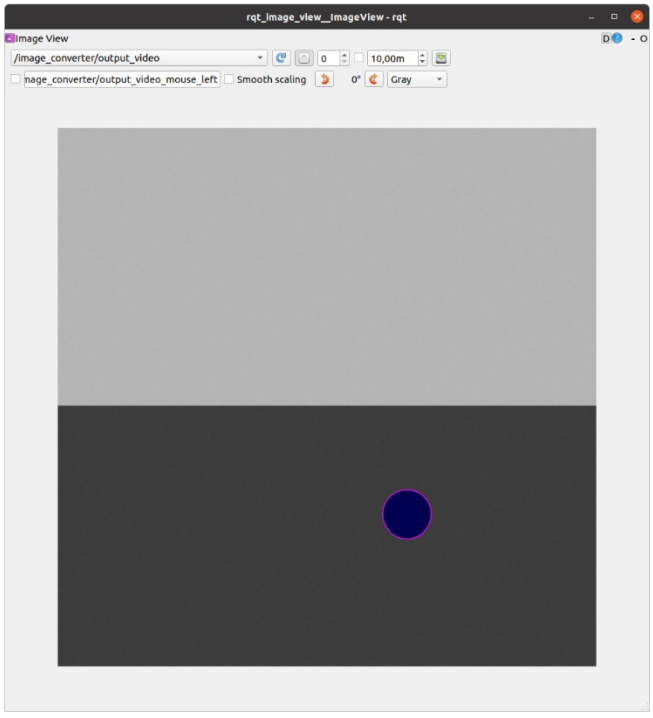


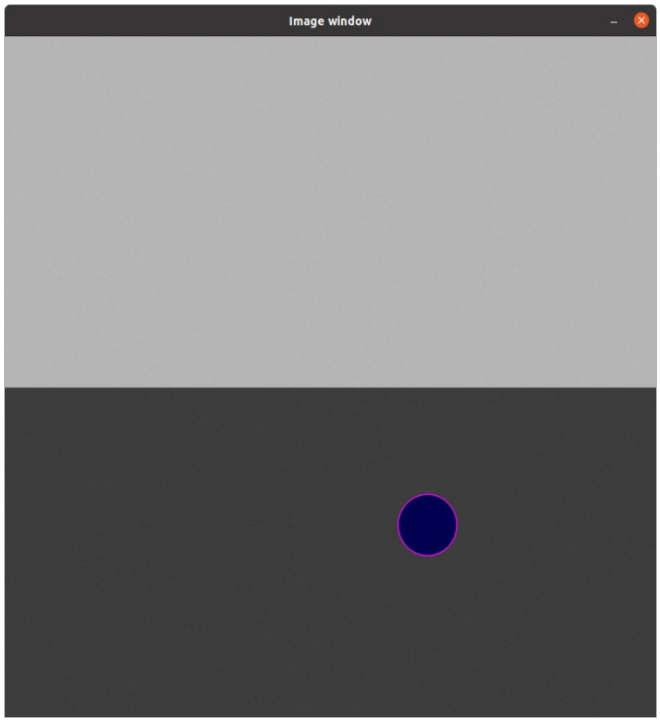
(c) Once the object is visible in the camera image, use the opencv_ros/ package to detect the circular object using open CV functions. Modify the opencv_ros_node.cpp to subscribe to the simulated image, detect the object via openCV functions, and republish the processed image.

At this point following the guide of "blob detection" the *opencv_ros/* package was used to detect the circular object using open CV functions, so the "*opencv_ros_node.cpp*" file has been modified. In this file is first taken the image seen from ROS (which is a coloured image), which is converted into an image at the gray level, on which the detection of keypoints in order to recognize the object. At the end on the color image are represented the keypoints with the fuchsia color and then the processed image is republished.

Used commands

```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_circular_object.launch
$ rosrun opencv_ros opencv_ros_node
$ rqt_image_view
```





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

(a) The kdl_robot package provides a kdl_robot_vision_control node that implements a visionbased look-at-point control task with the simulated iiwa robot. It uses the VelocityJointInterface enabled by the iiwa_gazebo_aruco.launch and the usb_cam_aruco.launch launch files. 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.

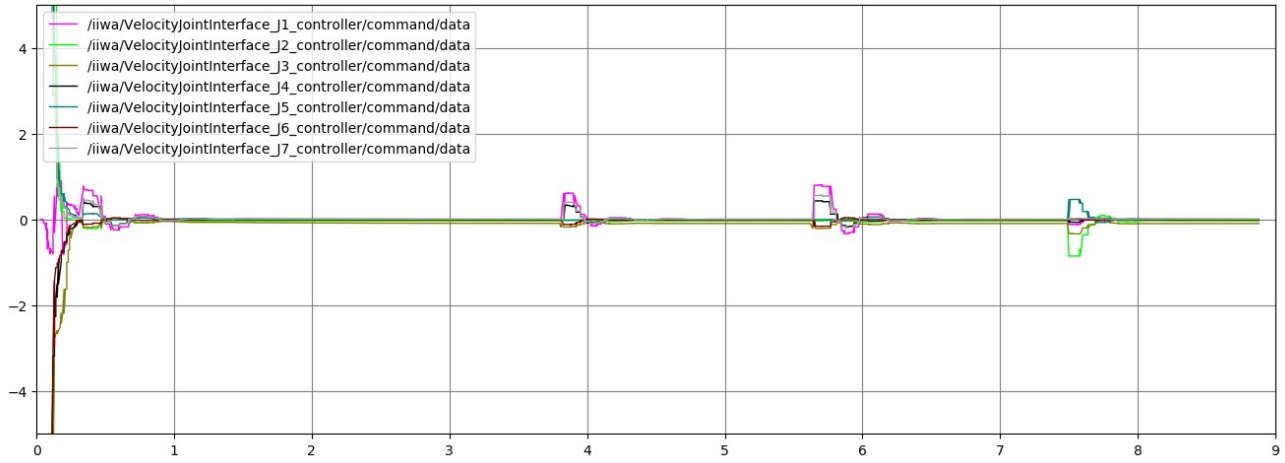
In order to implement the required vision-based task we created a frame offset that is rotated from the frame "*cam_T_object*" of 180° around x axis and that is placed by 0.5 along z axis. So, we computed the transformation of this frame to the base. Finally, we computed the orientation and position errors between orientation and position of the frame created before and the orientation and position of the actual frame of end effector.

```
237         // Create frame offset rotated by pi around x axis and placed by 0.5 along z axis
238         KDL::Frame frame_offset = cam_T_object;
239         frame_offset.M = cam_T_object.M * KDL::Rotation::RotX(-M_PI);
240         frame_offset.p = cam_T_object.p - KDL::Vector(0, 0, 0.5);
241         KDL::Frame base_T_frame_offset = robot.getEEFrame()*frame_offset;
242
243         // Compute errors
244         Eigen::Matrix<double,3,1> e_o = computeOrientationError(toEigen(base_T_frame_offset.M), toEigen(robot.getEEFrame().M));
245         Eigen::Matrix<double,3,1> e_o_w = computeOrientationError(toEigen(Fi.M), toEigen(robot.getEEFrame().M));
246         Eigen::Matrix<double,3,1> e_p = computeLinearError(toEigen(base_T_frame_offset.p),toEigen(robot.getEEFrame().p));
247         Eigen::Matrix<double,6,1> x_tilde; x_tilde << e_p,  e_o[0], e_o[1], e_o[2];
248
```

Used commands

```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_aruco.launch
$ roslaunch aruco_ros usb_cam_aruco.launch camera:=/iiwa/camera1/
$ rosrun kdl_ros_control kdl_robot_vision_control ./src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf
$ rqt_image_view
$ rqt_plot
```

Now, the tracking capability by moving the aruco marker via the gazebo interface is shown and the velocity commands sent to the robot are plotted as you can see in the following figures.



In the folder Simulations within github repository there are loaded the video "*Vision-based task.webm*" of the performed test in this point.

(b) An improved look-at-point algorithm can be devised by noticing that the task is belonging to S^2. Indeed, if we consider

$$s = \frac{{}^cP_o}{||{}^cP_o||} \in \mathbb{S}^2, \tag{1}$$

this is a unit-norm axis. The following matrix maps linear/angular velocities of the camera to changes in *s*

$$L(s) = R_c \begin{bmatrix} -\frac{1}{||{}^cP_o||}(I - ss^T) & S(s) \end{bmatrix} \in \mathbb{R}^{3\times 6}, \tag{2}$$

where *S(·)* is the skew-simmetric operator, R_c the current camera rotation matrix. Implement the following control law

$$\dot{q} = k(LJ)^{\dagger} s_d + N\dot{q}_0 \quad \text{with} \quad R = \begin{bmatrix} R_c & 0 \\ 0 & R_c \end{bmatrix}, \tag{3}$$

where s_d is a desired value for *s*, e.g. s_d = [0, 0, 1], and $N = (I - (LJ)^{\dagger}LJ)$ being the matrix spanning the null space of the *LJ* matrix. Verify that the for a chosen q_0 the s measure does not change by plotting joint velocities and the s components.

At this point we implemented within the code the equations (1), (2) and (3) as shown in the next figure.

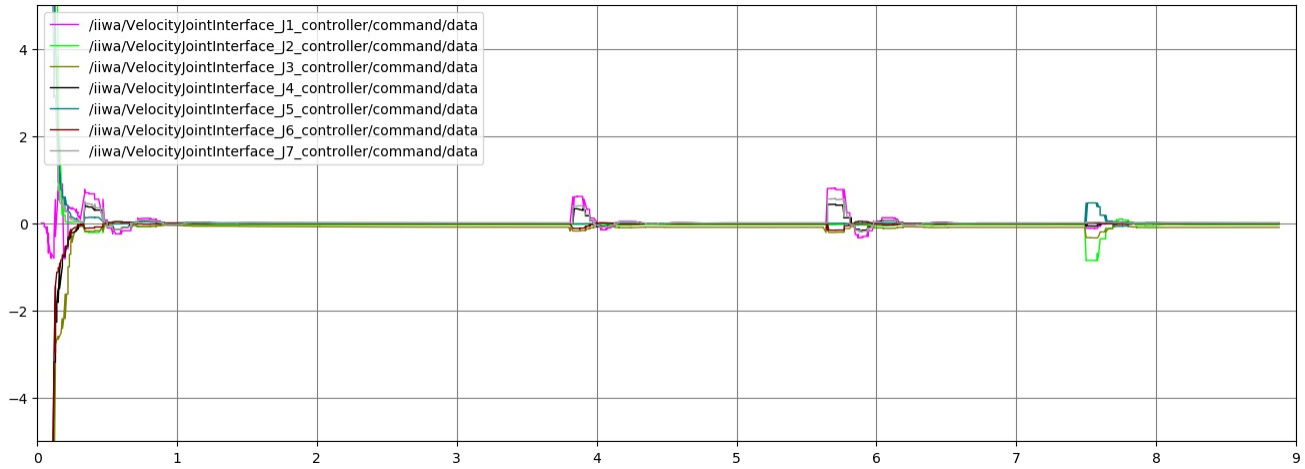
```
254         // Computing s (eq.1)
255         Eigen::Matrix<double,3,1> cP_o = toEigen(cam_T_object.p);
256         Eigen::Matrix<double,3,1> s = cP_o/cP_o.norm();
257
258         // R_c
259         Eigen::Matrix<double,3,3> R_c = toEigen(robot.getEEFrame().M);
260         // R (eq.2)
261         Eigen::Matrix<double,6,6> R = Eigen::MatrixXd::Zero(6,6);
262         R.block(0,0,3,3) = R_c;
263         R.block(3,3,3,3) = R_c;
264
265         // Computing L (eq.2)
266         Eigen::Matrix<double,3,6> L = Eigen::MatrixXd::Zero(3,6);
267         Eigen::Matrix<double,3,3> L_first = (-1/cP_o.norm())*(Eigen::MatrixXd::Identity(3,3)-s*s.transpose());
268         L.block(0,0,3,3) = L_first;
269         L.block(0,3,3,3) = skew(s);
270         L = L*R.transpose();
271
272         // Computing sd
273         Eigen::Matrix<double,3,1> sd = Eigen::Vector3d(0,0,1);
274         // Computing N
275         Eigen::MatrixXd LJ = L*toEigen(J_cam);
276         Eigen::MatrixXd LJ_pinv = LJ.completeOrthogonalDecomposition().pseudoInverse();;
277         Eigen::MatrixXd N = (Eigen::Matrix<double,7,7>::Identity())-(LJ_pinv*LJ);
278
279         double k = 10;
280         Eigen::Matrix<double,7,1> dq0 = qdi - toEigen(jnt_pos);
281         dqd.data = k*LJ_pinv*sd + N*dq0;
```

Used commands

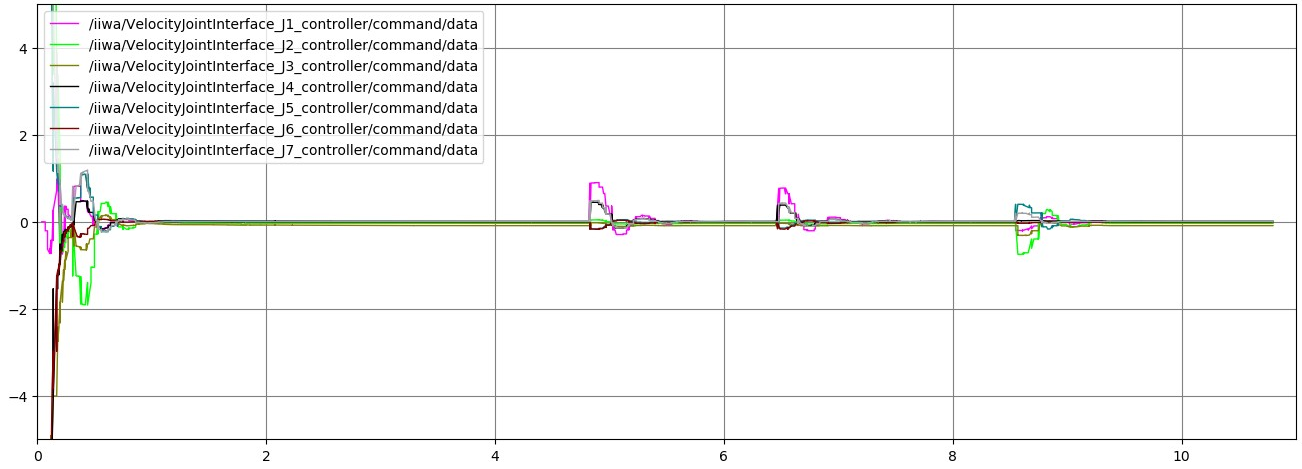
```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_aruco.launch
$ roslaunch aruco_ros usb_cam_aruco.launch camera:=/iiwa/camera1/
$ rosrun kdl_ros_control kdl_robot_vision_control ./src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf
$ rqt_plot
```

As follow, we have shown the plots of the joint velocities and the s components in two cases, in particular we have seen the plots for two chosen $\dot{q}_0$. Since this last one depends on "*qdi*", we made the changes on it and they are shown in the figure below the plots.
In order to plot the s components, we created three topics called "*/iiwa/sx*", "*/iiwa/sy*", "*/iiwa/sz*" which data are plotted through "*rqt_plot*" command.

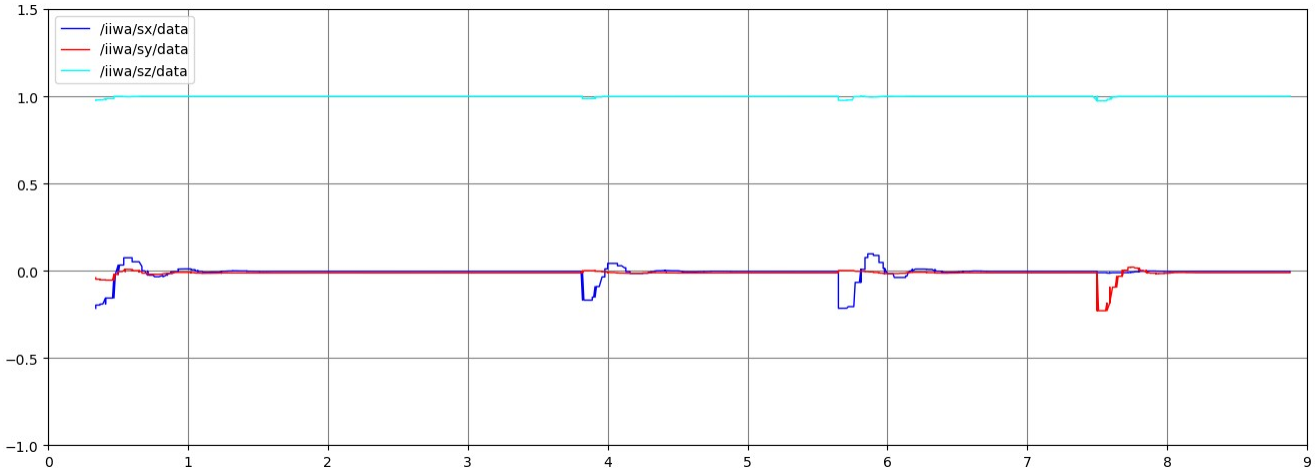
Joint velocities plot with $\dot{q}_0$ without modifications:



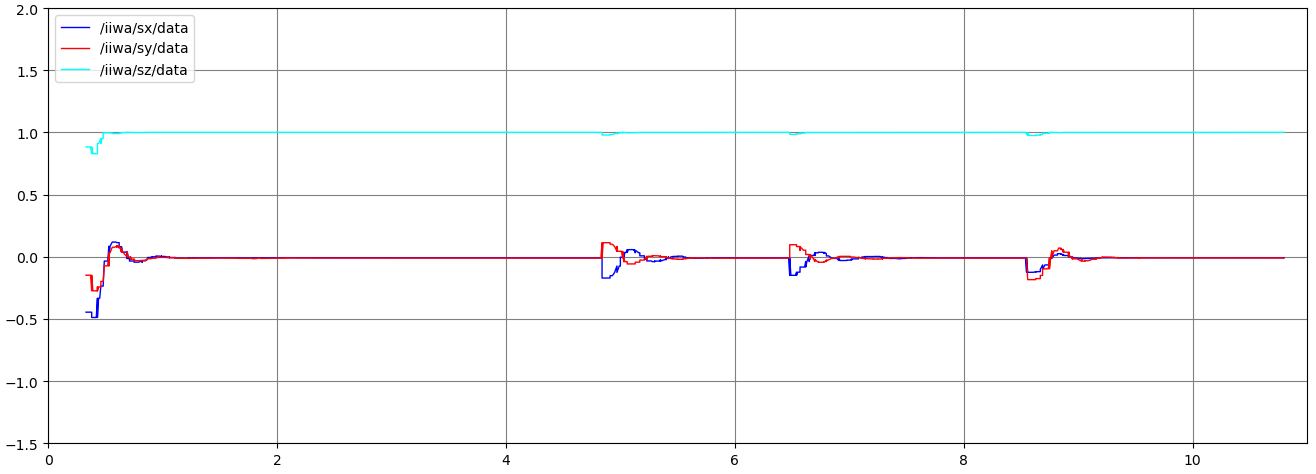
Joint velocities plot with $\dot{q}_0$ with modifications:



S components plot with $\dot{q}_0$ without modifications:



S components plot with $\dot{q}_0$ with modifications:



The modifications in the code:

```
106    // Initial desired robot state  // The comments near the lines are used to improve last request of point 2b
107    init_jnt_pos[0] = 0.0;
108    init_jnt_pos[1] = 1.57;
109    init_jnt_pos[2] = -1.57;  //-1.30
110    init_jnt_pos[3] = -1.57;
111    init_jnt_pos[4] = 1.57;   //1.30
112    init_jnt_pos[5] = -1.57;  //-1.20
113    init_jnt_pos[6] = 1.57;
114    Eigen::VectorXd qdi = toEigen(init_jnt_pos);
115
```

(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.
Hint: Replace the orientation error e_o with respect to a fixed reference (used in the previous homework), with the one generated by the vision-based controller. Plot the results in terms of commanded joint torques and Cartesian error norm along the performed trajectory.

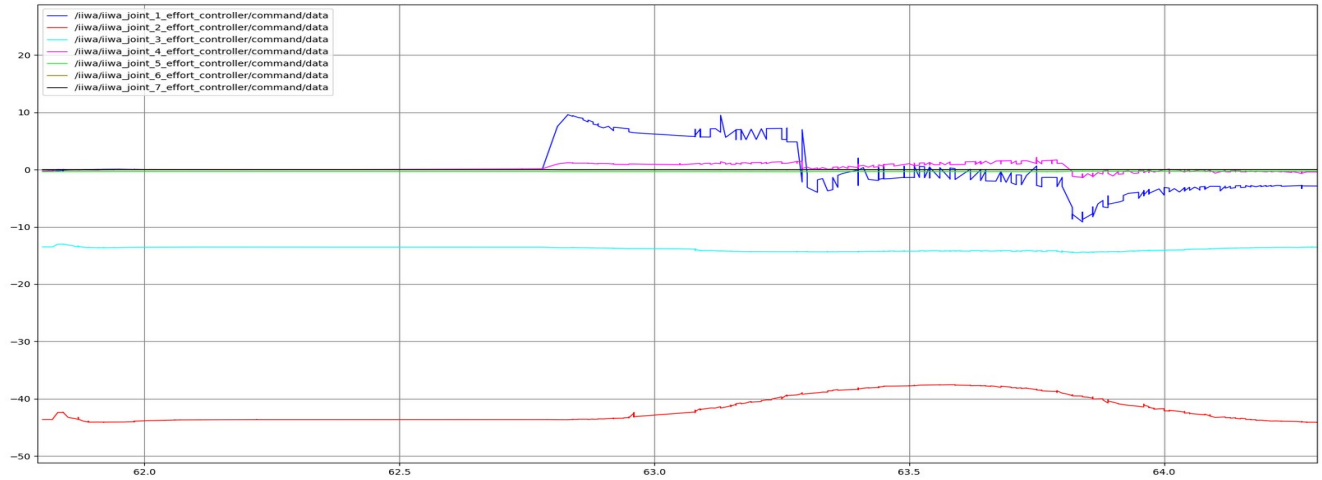
At the first, we created another launch file "*iiwa_gazebo_effort_aruco.launch*" in "*arm_gazebo*" folder in "*arm_stack*" folder where effort controllers "*iiwa_joint_X_effort_controller*" are loaded.
The two controllers are merged and are enabled the 4 trajectories of the previous homework with the vision-based task. Also here the "*compute_trajectory*" function is resumed in order to implement one of the 4 trajectories (always chosen from terminal).
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.
At the end, in order to plot the Cartesian norm error along the performed trajectories, we created two topics called "*/iiwa/Cartesian_error_norm_position*", "*/iiwa/Cartesian_error_norm_orientation*" which data are plotted through "*rqt_plot*" command. The data of the topics of the Cartesian norm position and orientation errors are computed as following figure and then they are published.

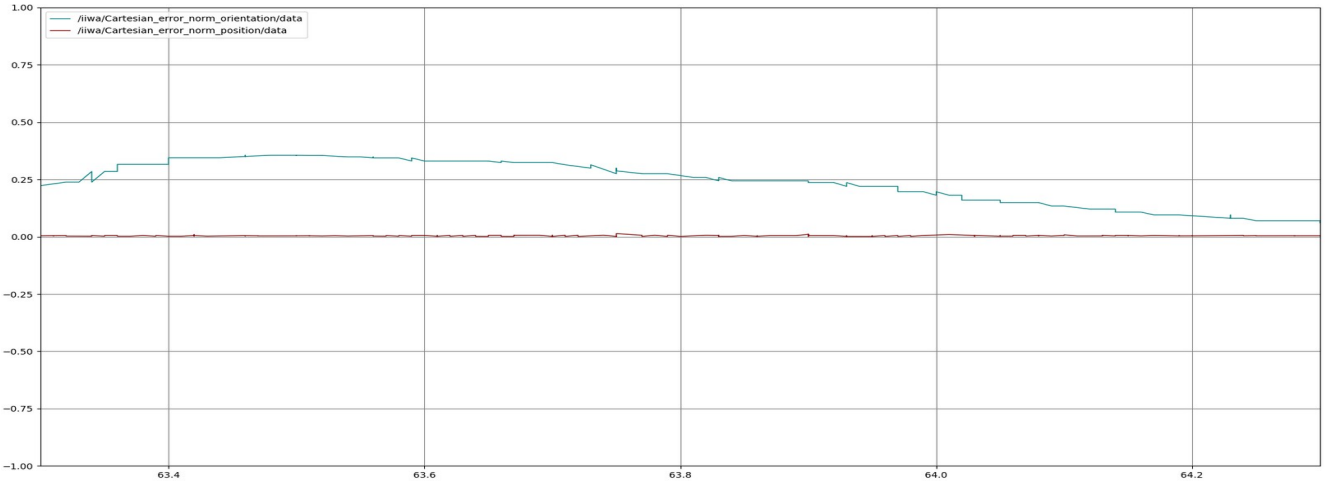
```
339         Eigen::Vector3d cart_error_pos = toEigen(des_pose.p) - toEigen(robot.getEEFrame().p);
340         double cart_error_pos_norm = cart_error_pos.norm();
341         Eigen::Vector3d or_des;
342         des_pose.M.GetRPY(or_des[0], or_des[1], or_des[2]);
343         Eigen::Vector3d or_e;
344         robot.getEEFrame().M.GetRPY(or_e[0], or_e[1], or_e[2]);
345         Eigen::Vector3d cart_error_or = or_des - or_e;
346         double cart_error_or_norm = cart_error_or.norm();
347
348         // Create Cartesian error norm msg
349         cart_err_norm_pos_msg.data = cart_error_pos_norm;
350         cart_err_norm_or_msg.data = cart_error_or_norm;
```

Used commands
```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_effort_aruco.launch
$ roslaunch aruco_ros usb_cam_aruco.launch camera:=/iiwa/camera1/
$ rosrun kdl_ros_control kdl_robot_test ./src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf
$ rqt_image_view
$ rqt_plot
```

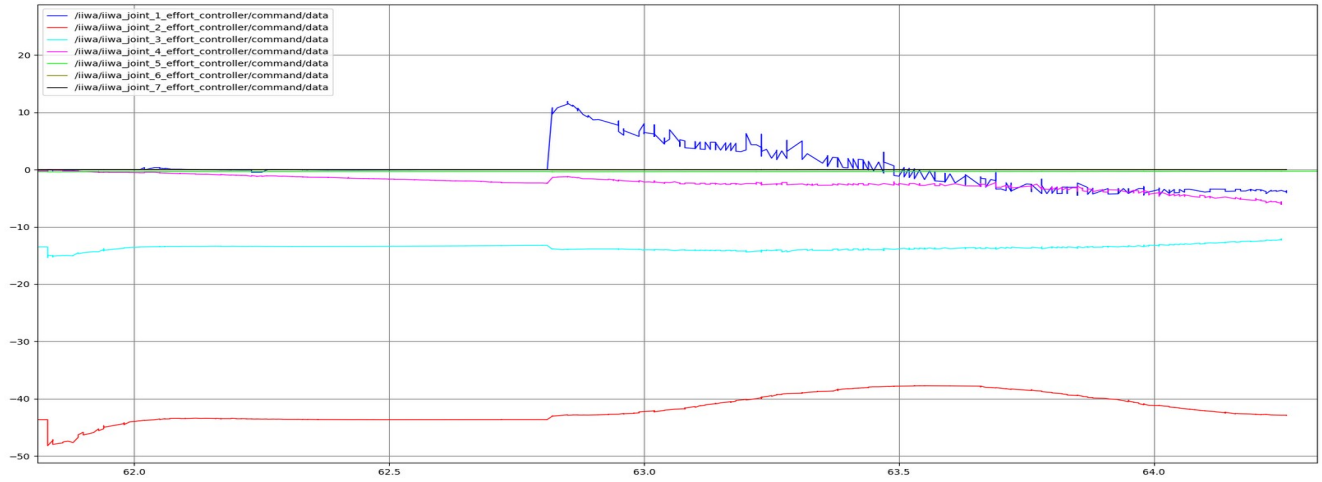
As follow the plots of the results are shown in terms of commanded joint torques and Cartesian error norm along the performed trajectory with Cartesian space inverse dynamics control and joint space inverse dynamics control:

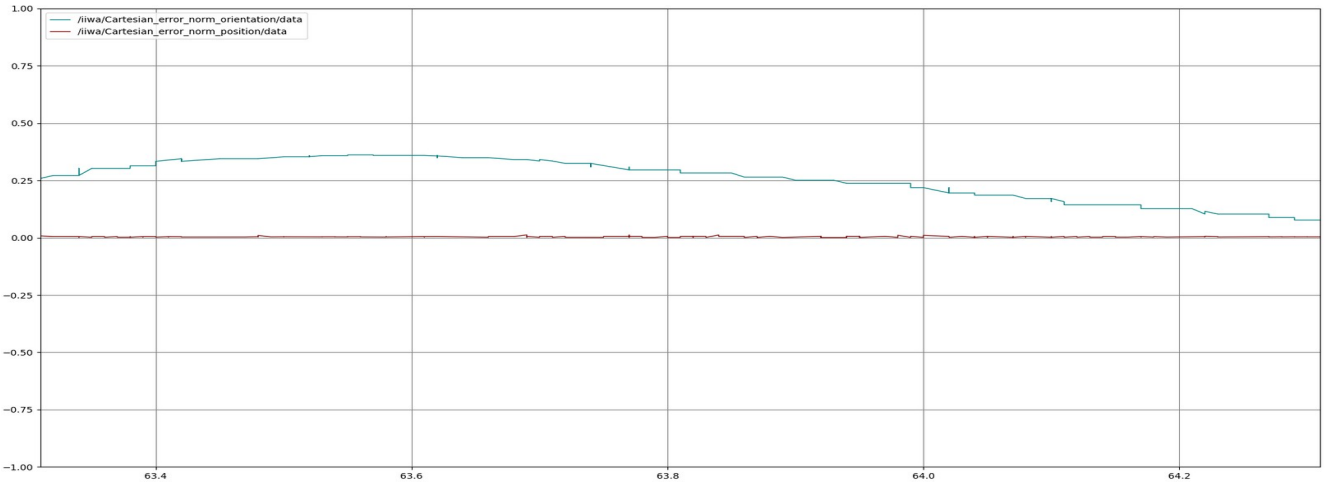
1 → Linear trajectory with trapezoidal velocity profile with operational space controller.



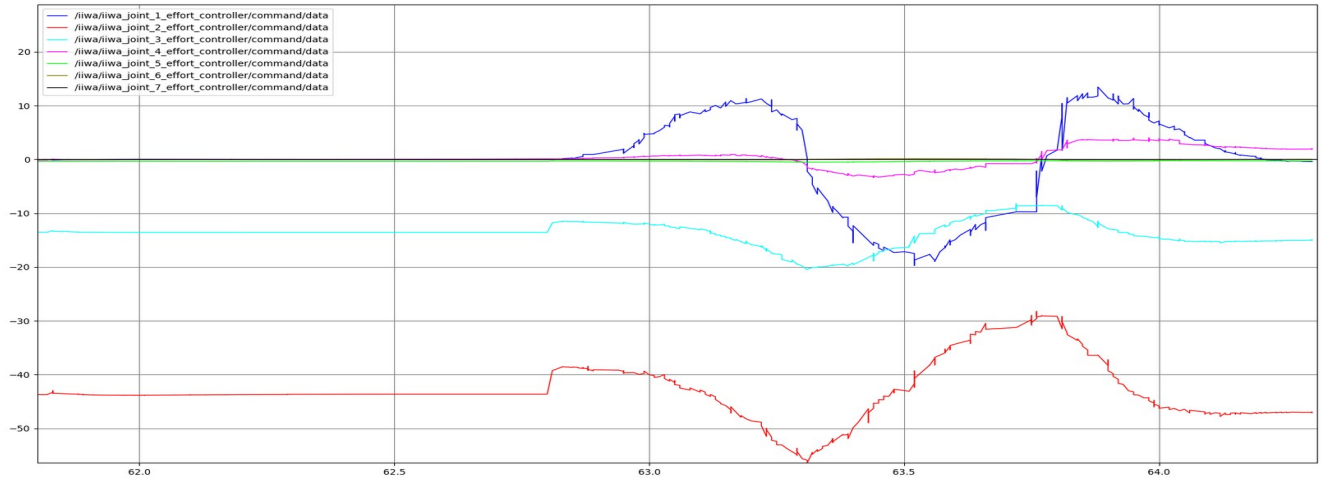


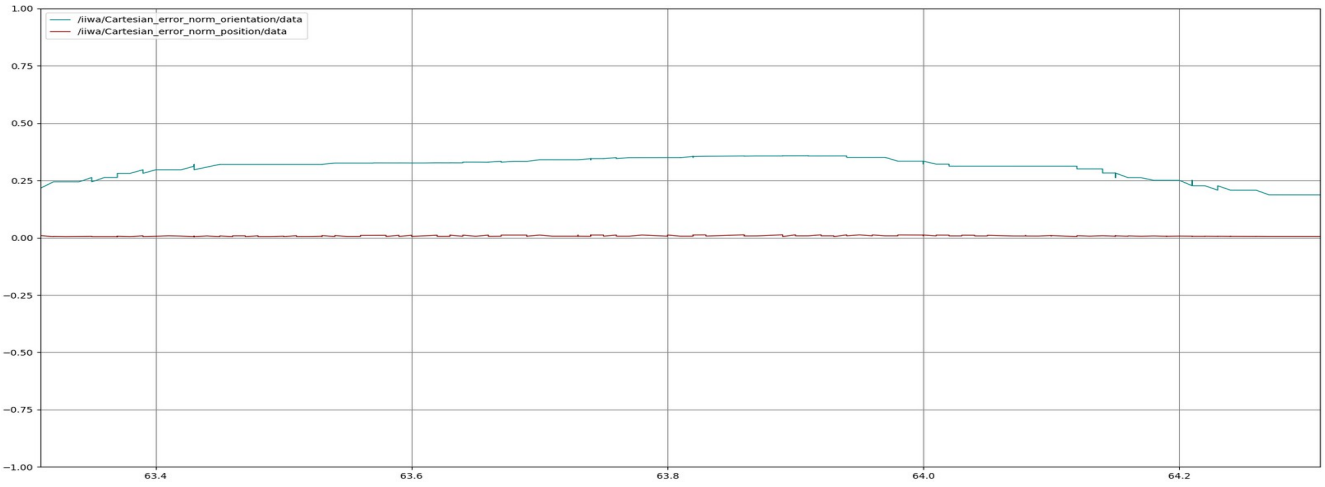
2 → Linear trajectory with cubic polynomial velocity profile with operational space controller.



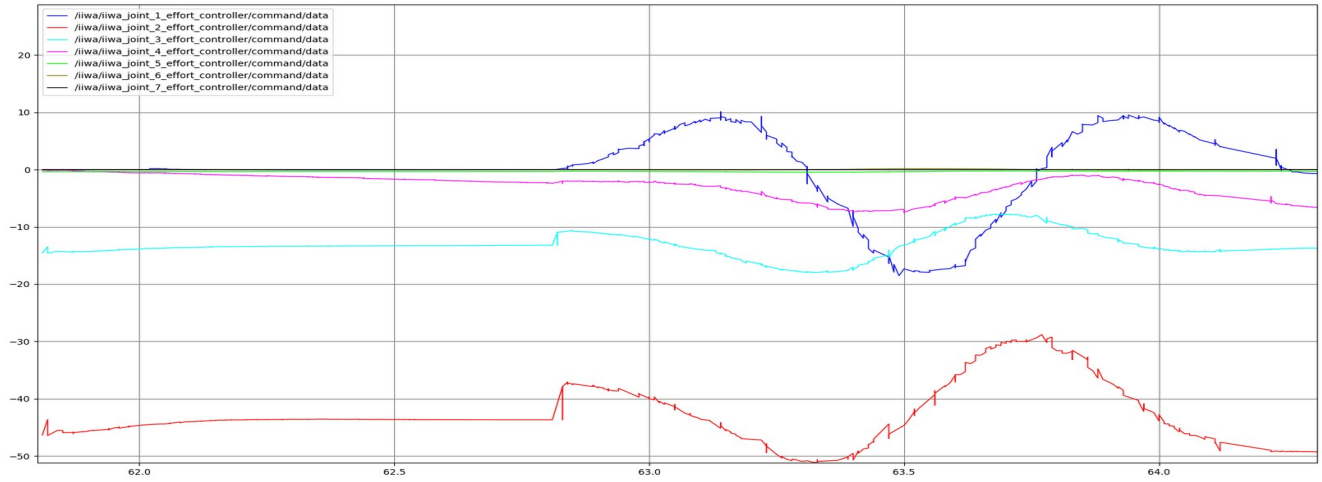


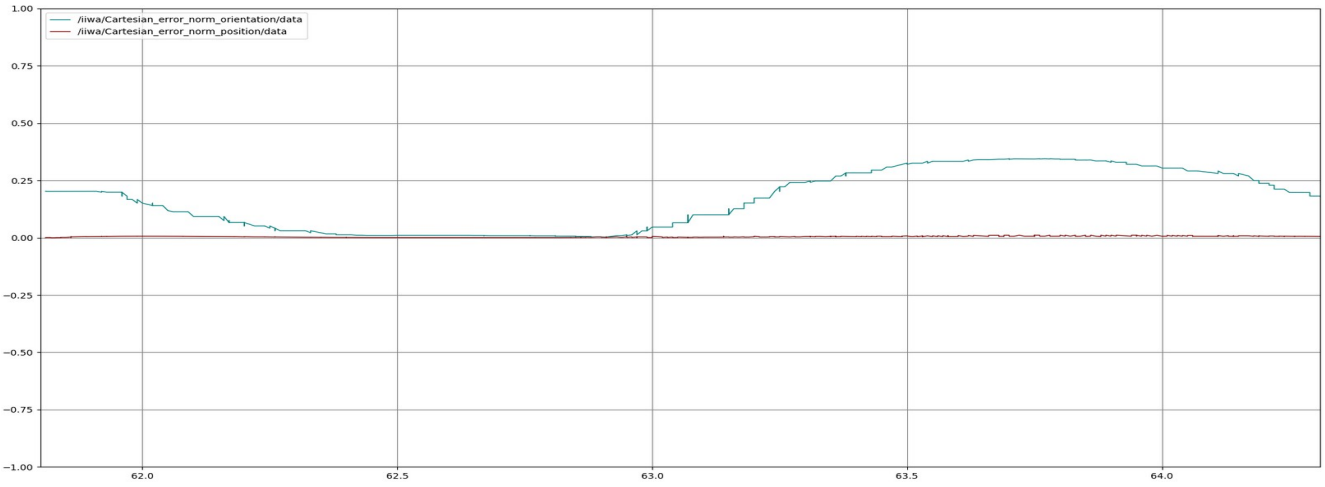
3 → Circular trajectory with trapezoidal velocity profile with operational space controller.



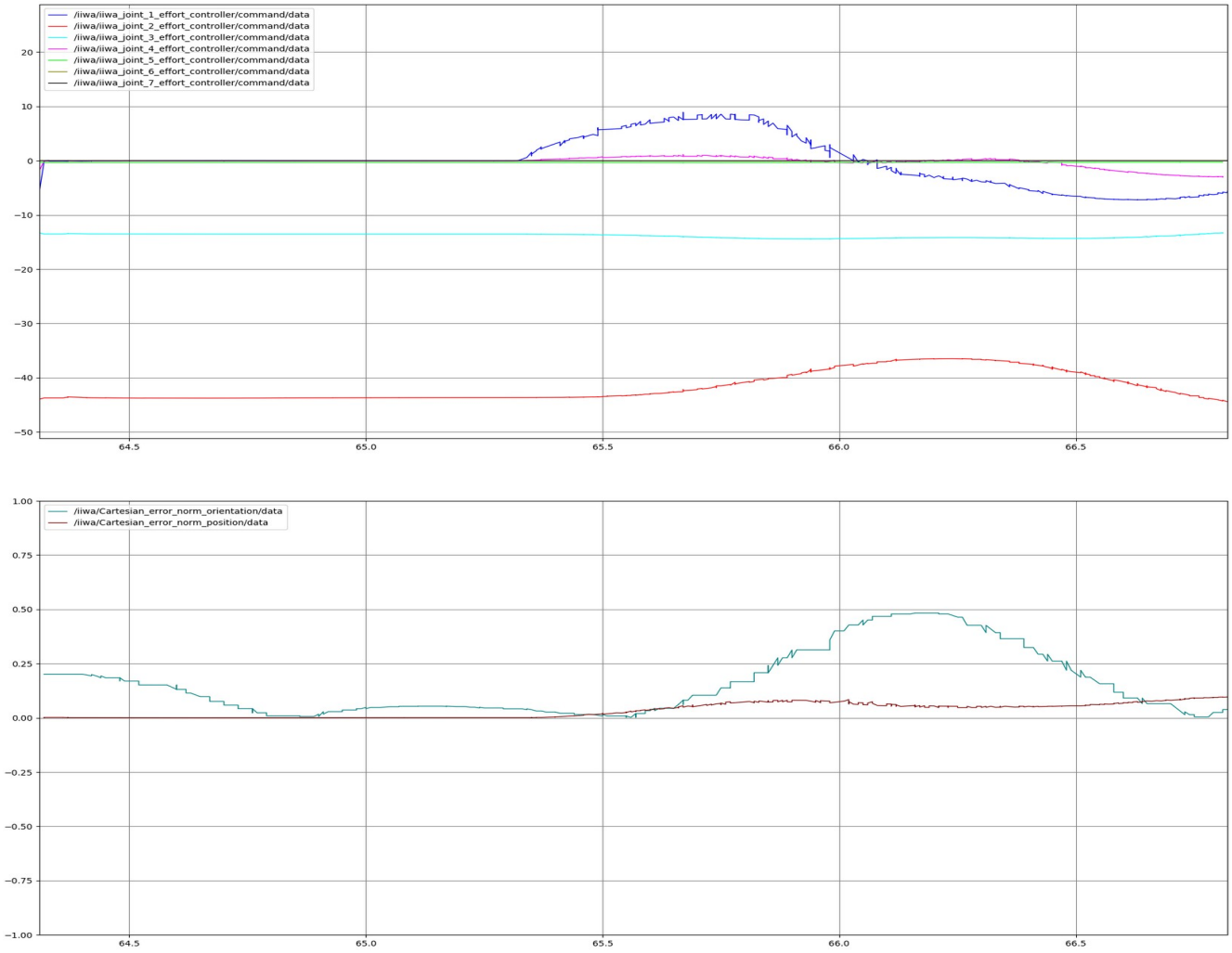


4 → Circular trajectory with cubic polynomial velocity profile with operational space controller.

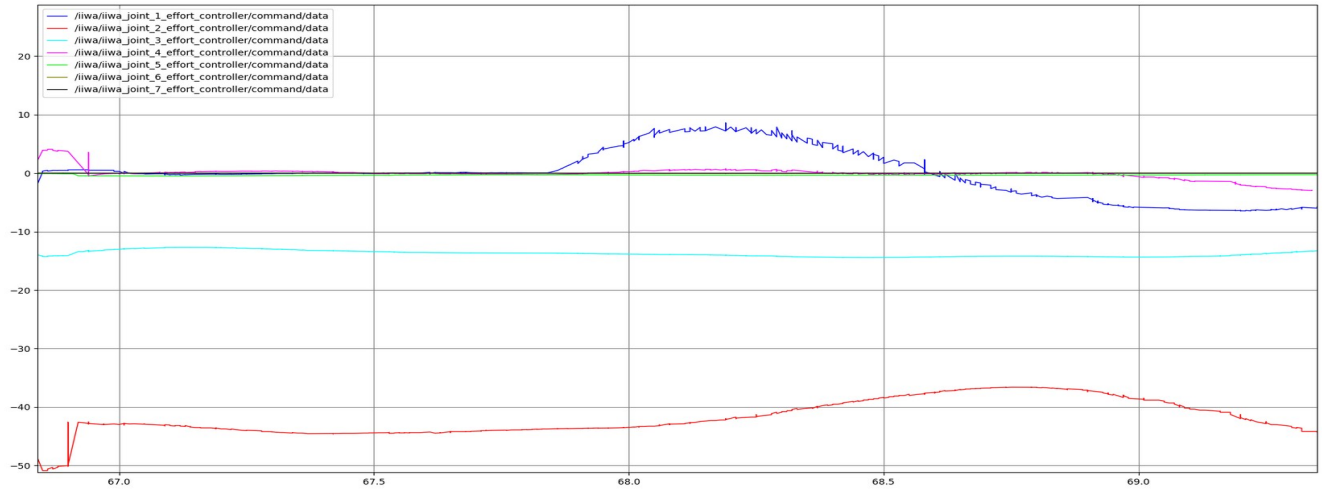


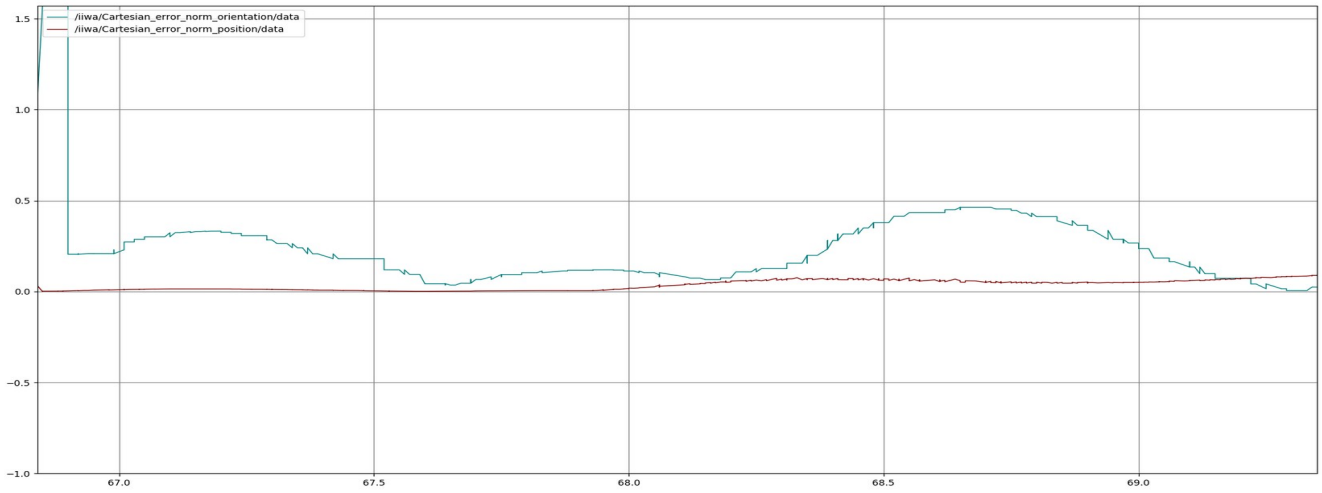


1 → Linear trajectory with trapezoidal velocity profile with joint space controller.

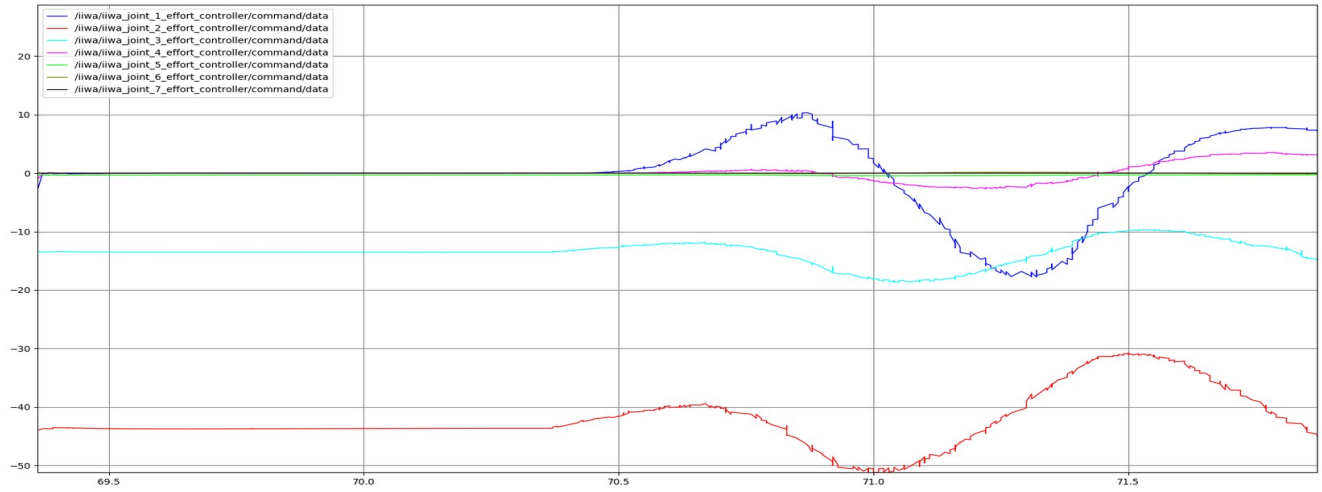


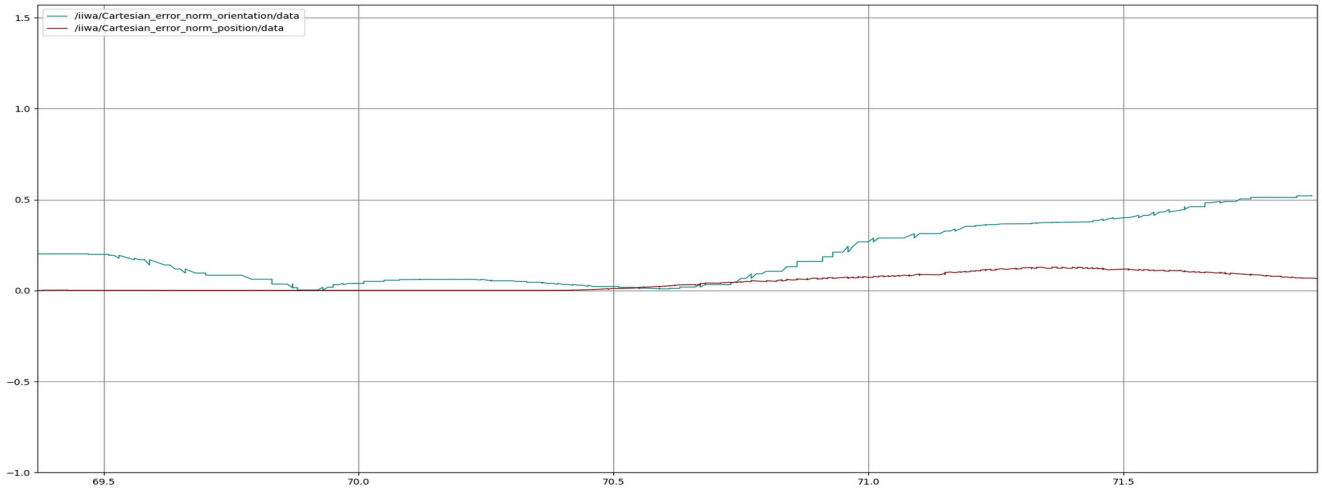
2 → Linear trajectory with cubic polynomial velocity profile with joint space controller.



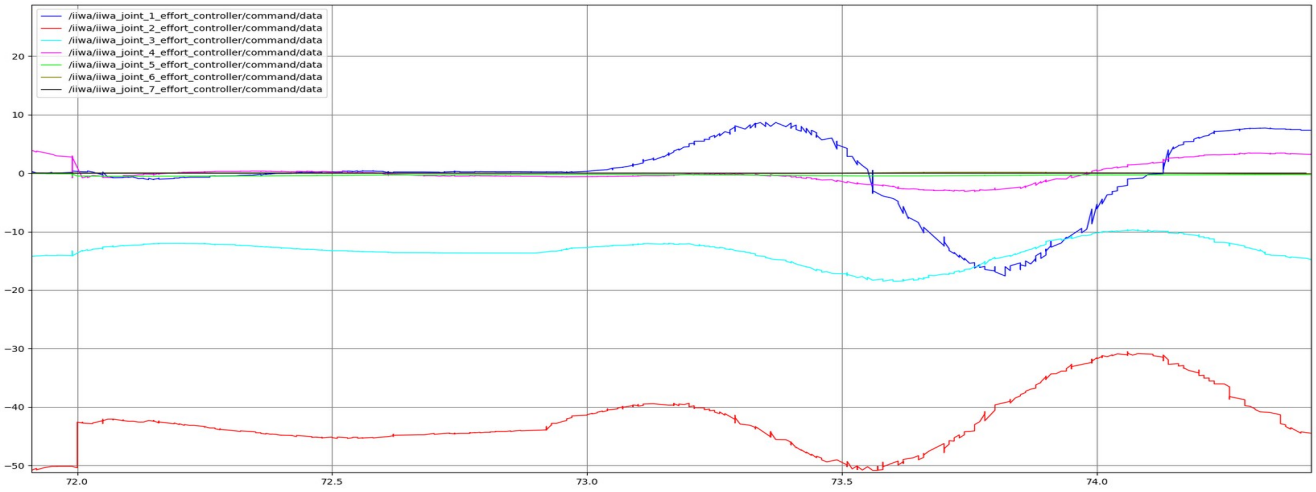


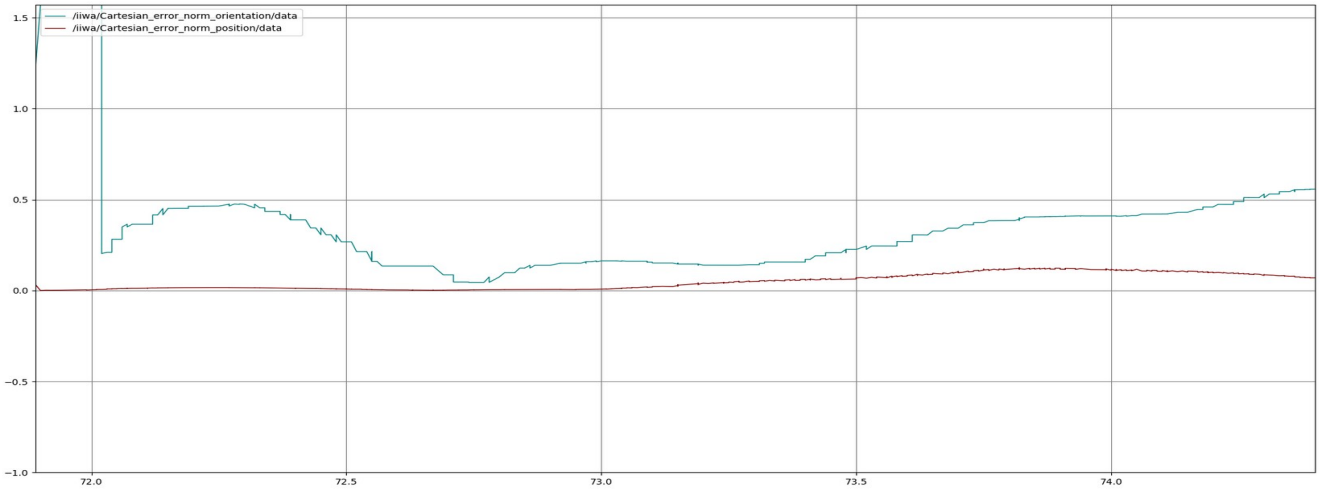
3 → Circular trajectory with trapezoidal velocity profile with joint space controller.





4 → Circular trajectory with cubic polynomial velocity profile with joint space controller.





In the folder Simulations within github repository there are loaded the videos of the tests of four trajectories that have been implemented.