# Robotics Lab: Homework 3

Implement a vision-based task

**Pietro Falco**
**P38 000208**

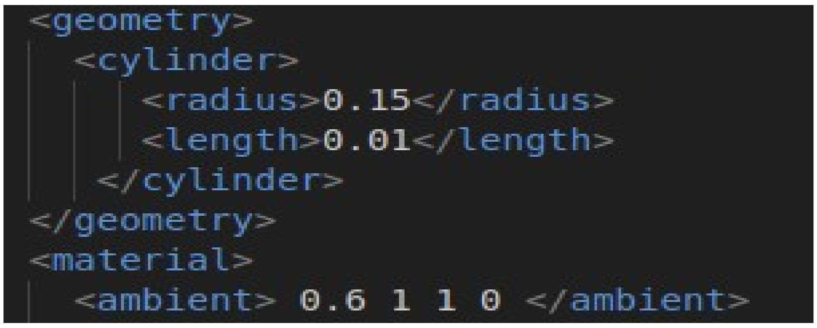
## 1 - Implement a vision-based task

***The goal of this homework is to implement a vision-based controller for a 7-degrees-of-freedom robotic manipulator arm into the Gazebo environment. The kdl_robot package (at the following link: [https: //github.com/mrslvg/kdl_robot)](https://github.com/mrslvg/kdl_robot) must be used as starting point. The student is requested to address the following points and provide a detailed report of the employed methods. In addition, a personal github repo with all the developed code must be shared with the instuctor. The report is due in one week from the homewerk release.***

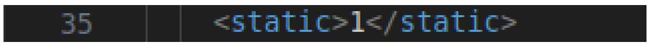
1. **Construct a gazebo world inserting a circular object and detect it via the opencv_ros package**

   **(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.***

We created a new folder named **circular_object** that is a copy of the folder named **aruco_marker**. Then we modified the file model.sdf adding the following lines:



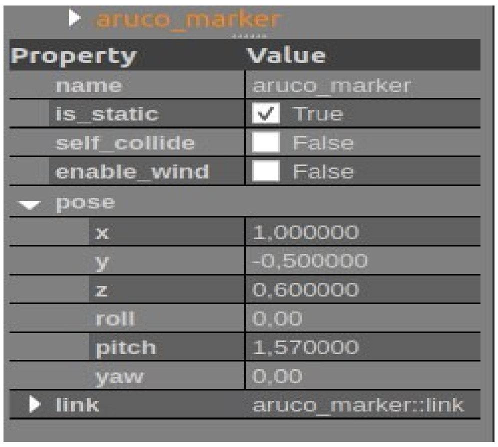
```
<geometry>
  <cylinder>
    <radius>0.15</radius>
    <length>0.01</length>
  </cylinder>
</geometry>
<material>
  <ambient> 0.6 1 1 0 </ambient>
```



```
35      <static>1</static>
```

Then we launched **iiwa_gazebo** with **roslaunch iiwa_gazebo iiwa_gazebo.launch**, and in the section "***insert***" we added the path concerning the **models** folder in **circular_object folder** and we visualized the desired **circular_object** on the **ground_plane**. Then we ticked the property "**is_static**" to make sure that the marker did not fall by the force of gravity and we modified the coordinates of its pose.

We saved the file as "**iiwa_circular_object.world**".



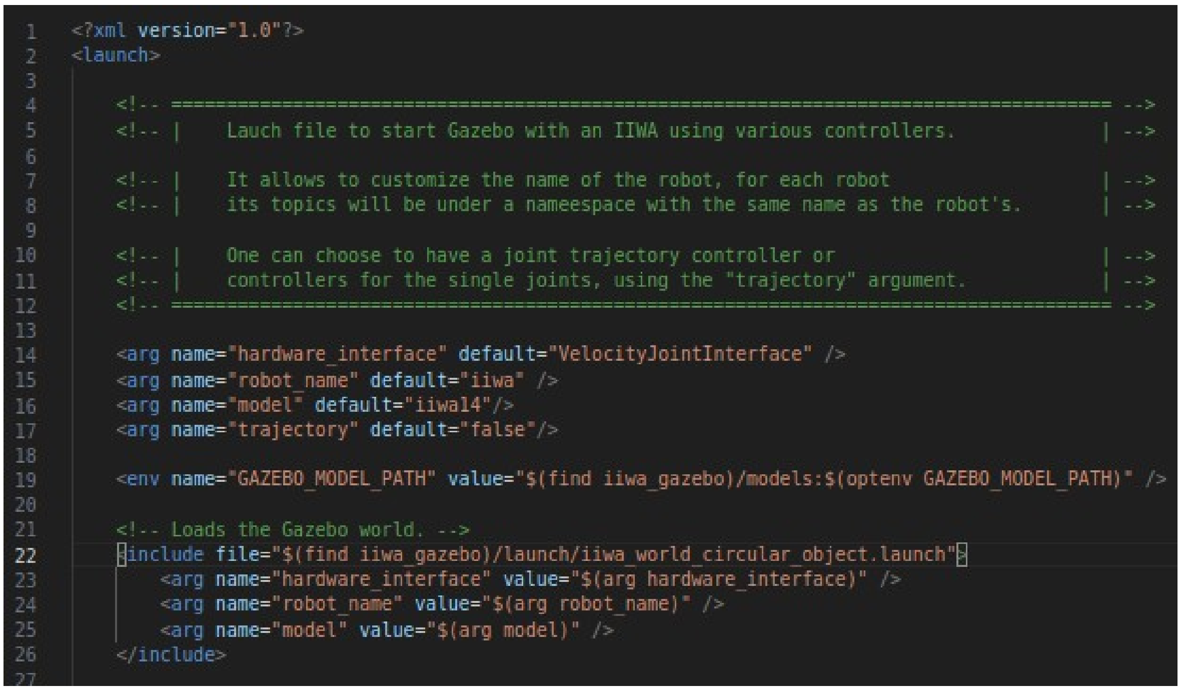
aruco_marker

| Property | Value |
|---|---|
| name | aruco_marker |
| is_static | ✓ True |
| self_collide | False |
| enable_wind | False |
| pose | |
| x | 1,000000 |
| y | -0,500000 |
| z | 0,600000 |
| roll | 0,00 |
| pitch | 1,570000 |
| yaw | 0,00 |
| link | aruco_marker::link |

**(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).***

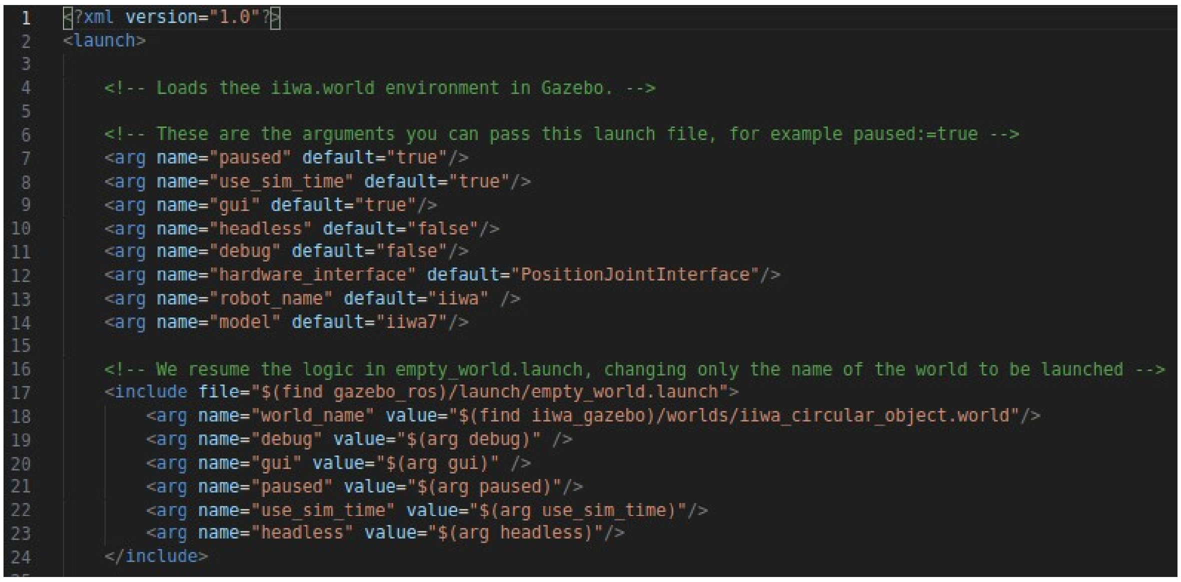
We created two new launch file named respectively **iiwa_gazebo_circular_object.launch** and **iiwa_world_circular_object.launch.** Those two files load the iiwa robot with **PositionJointInterface** and the **new world** previously created:

**iiwa_gazebo_circular_object.launch**:



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

    <!-- ======================================================================================= -->
    <!-- |    Lauch file to start Gazebo with an IIWA using various controllers.              | -->

    <!-- |    It allows to customize the name of the robot, for each robot                    | -->
    <!-- |    its topics will be under a nameespace with the same name as the robot's.        | -->

    <!-- |    One can choose to have a joint trajectory controller or                         | -->
    <!-- |    controllers for the single joints, using the "trajectory" argument.             | -->
    <!-- ======================================================================================= -->

    <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>
```

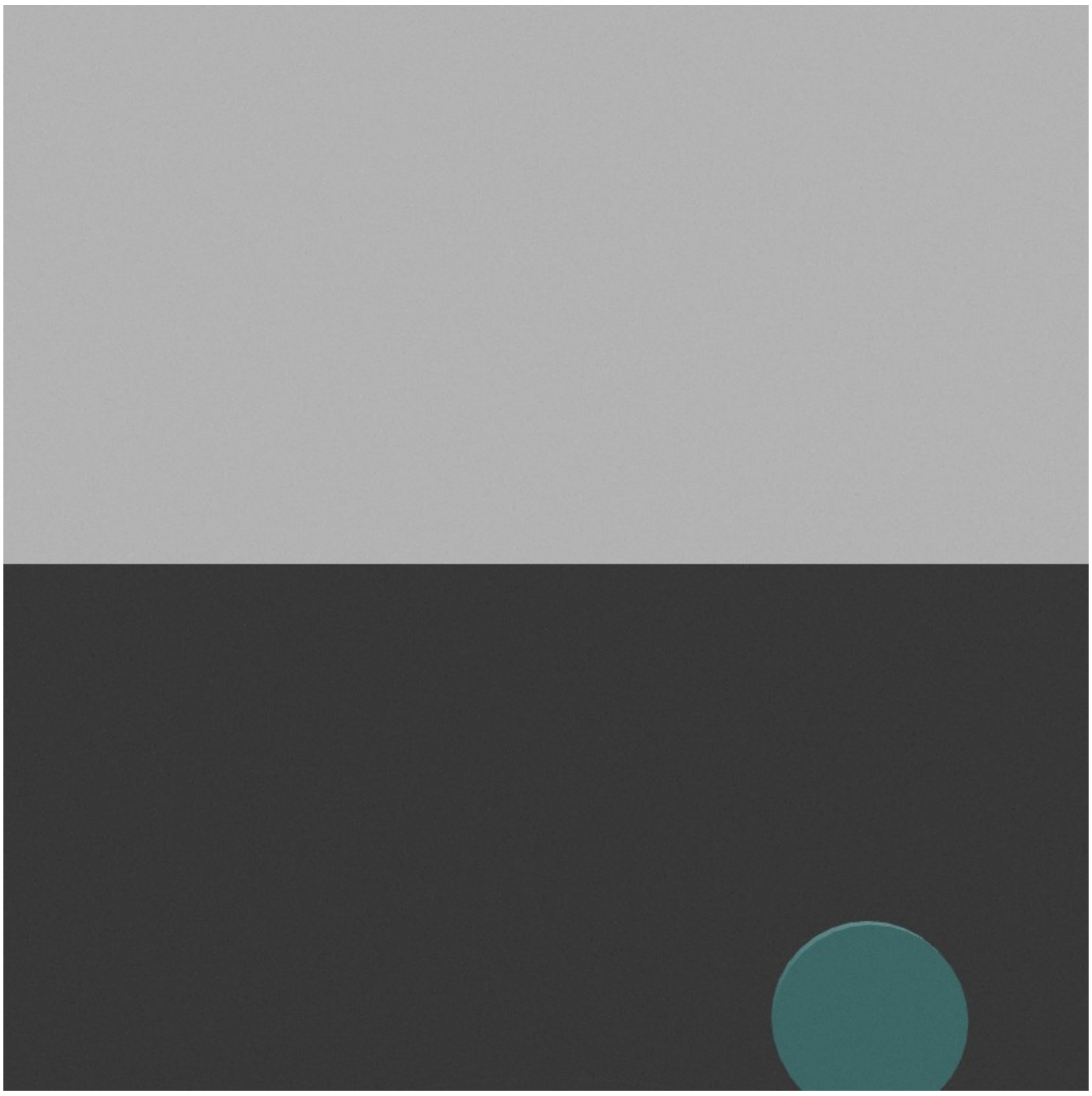
**iiwa_world_circular_object.launch**:



```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_object.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>
```

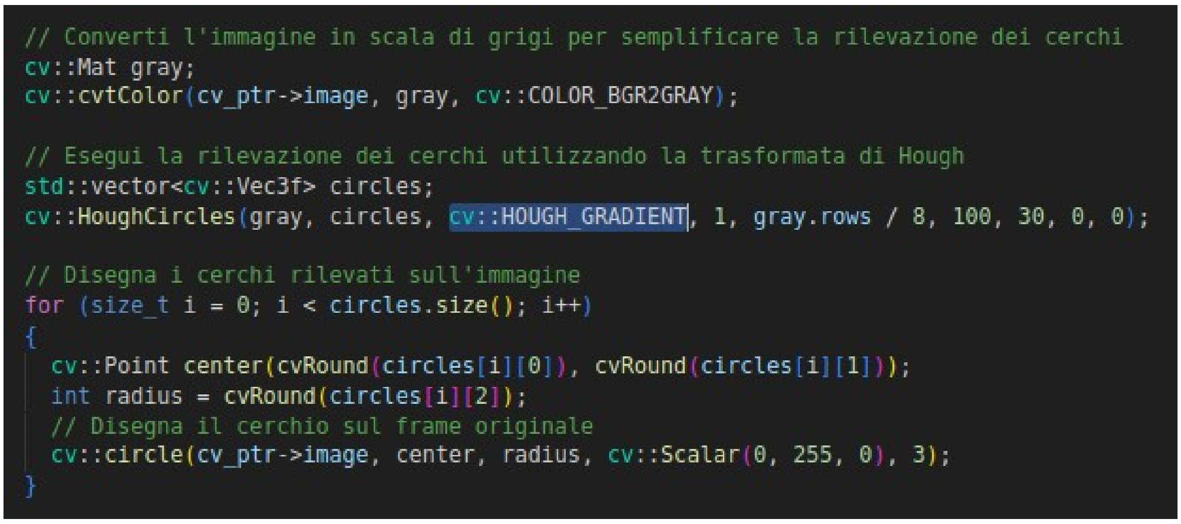
To check if everything was fine we used the **rqt_image_view** command.

This is the result:



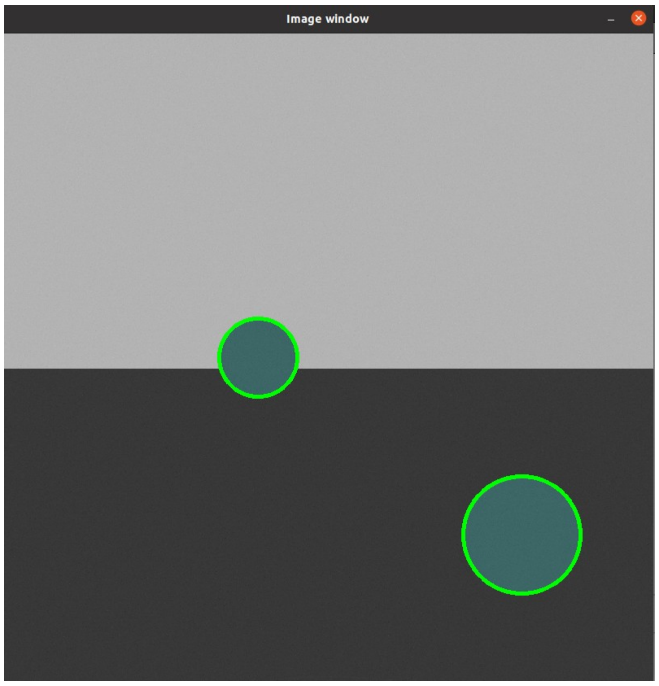
**(c)** ***Once the object is visible in the camera image, use the opencv_ros/ package to detect the circular object using open CV functions. Modify the opencv_ros_node.cpp to subscribe to the simulated image, detect the object via openCV functions, and republish the processed image***

We modified **opencv_ros_node.cpp** replacing the topic "***/usb_cam/image_raw"*** with "***/usb_cam/camera1/image_raw***". Then we added the following lines:



```
// Converti l'immagine in scala di grigi per semplificare la rilevazione dei cerchi
cv::Mat gray;
cv::cvtColor(cv_ptr->image, gray, cv::COLOR_BGR2GRAY);

// Esegui la rilevazione dei cerchi utilizzando la trasformata di Hough
std::vector<cv::Vec3f> circles;
cv::HoughCircles(gray, circles, cv::HOUGH_GRADIENT, 1, gray.rows / 8, 100, 30, 0, 0);

// Disegna i cerchi rilevati sull'immagine
for (size_t i = 0; i < circles.size(); i++)
{
  cv::Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
  int radius = cvRound(circles[i][2]);
  // Disegna il cerchio sul frame originale
  cv::circle(cv_ptr->image, center, radius, cv::Scalar(0, 255, 0), 3);
}
```

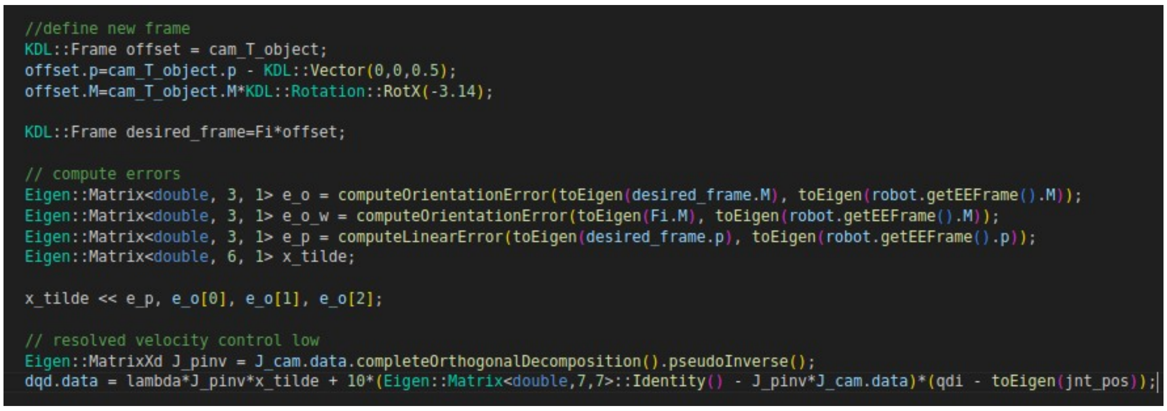
What we have done is converting the image from **RGB** to **greyscale** in order to simplify the detection.
Then we used the **HougCircles** which is a function from the OpenCV library. This function detects circles in an image using the circular Hough Transform. Once done that it was easy to draw the circles on the image.



## 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.***

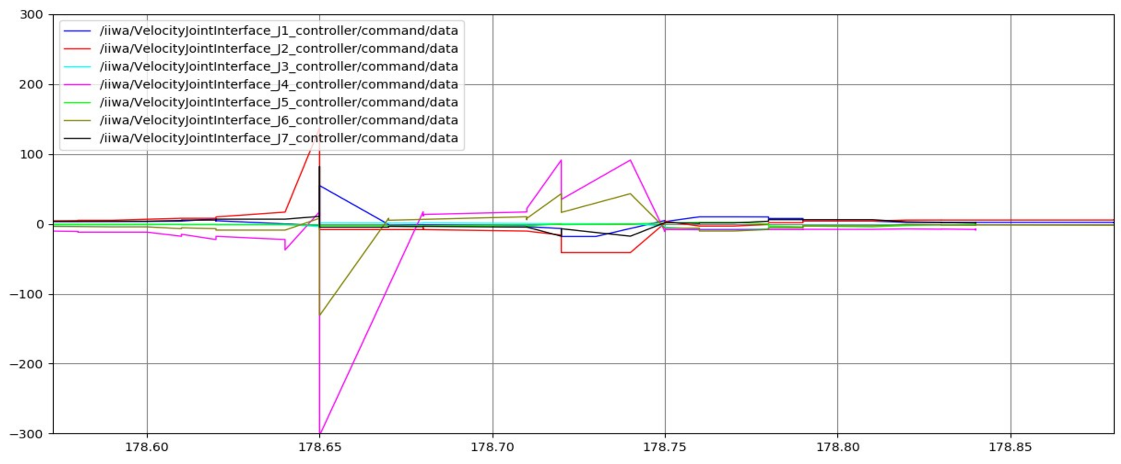
The main goal was to get the r**obot following an object** with a **constant offset**. In order to do that we used both **forward** and **inverse kinematics** to ensure that the **end-effector frame** reaches the desired configuration in the operational space. We modified the **kdl_robot_vision_control.cpp** file adding these lines:



```
//define new frame
KDL::Frame offset = cam_T_object;
offset.p=cam_T_object.p - KDL::Vector(0,0,0.5);
offset.M=cam_T_object.M*KDL::Rotation::RotX(-3.14);

KDL::Frame desired_frame=Fi*offset;

// compute errors
Eigen::Matrix<double, 3, 1> e_o = computeOrientationError(toEigen(desired_frame.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(desired_frame.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];

// resolved velocity control low
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));
```

We defined a new frame named ‘**offset**’ putting it equal to the ‘**cam_T_object**’ frame that undergoes a downward translation along the z-axis and a 180-degree rotation around the x-axis relative to the original frame. Then we defined a new frame **“desired_frame”** which is obtained by the product between **Fi** (the current end-effector frame) and the **offset** frame. Anyway we computed the **errors** between the **desired** and **current position/orientation** of the robot’s end-effector frame and we put them into a vector named “**x_tilde”**.
At the end we calculated a desired velocity vector named ‘**dqd.data**’ using the concept of resolved velocity control. This vector took into account position and orientation errors, ensuring that the manipulator moves appropriately towards the desired position and orientation.

Then we used “**rqt_plot**” to plot the velocity commands sent to the robot obtained moving the aruco marker in the gazebo world:



**(b)** ***An improved look-at-point algorithm can be devised by noticing that the task is belonging to S2. Indeed, if we consider***

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

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

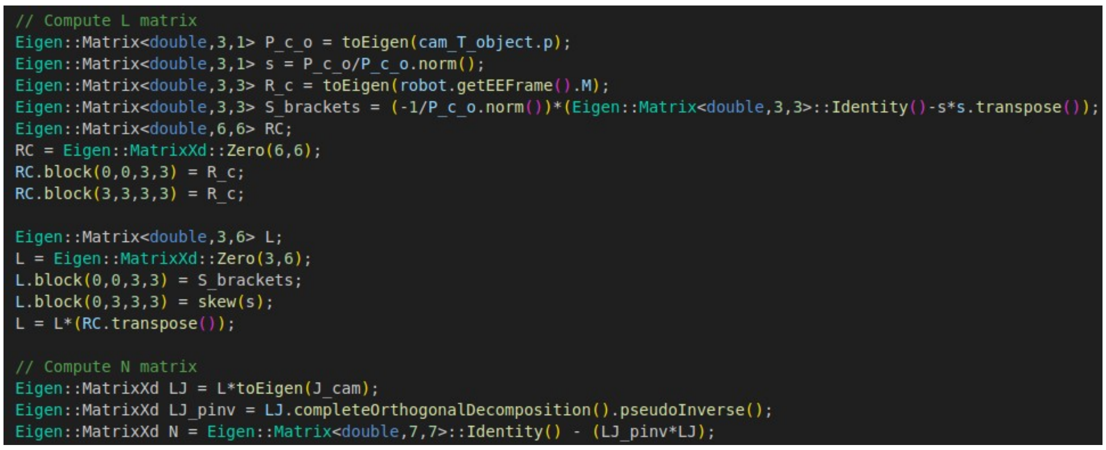
$$L(s) = R_c \begin{bmatrix} -\frac{1}{||{}^cP_o||}\left(I - ss^T\right) & S(s) \end{bmatrix} \in \mathbb{R}^{3\times 6}$$

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

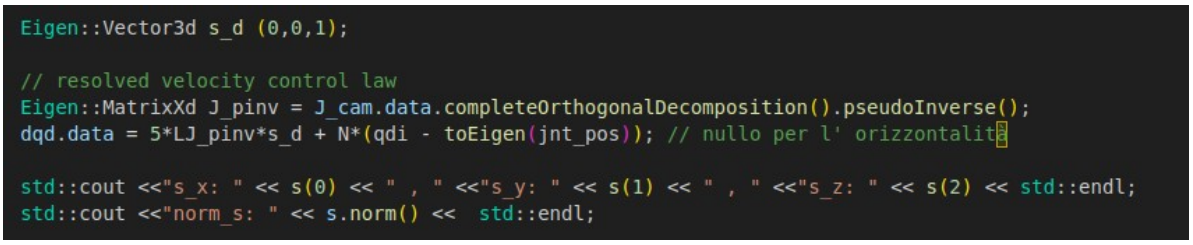
$$\dot{q} = k(LJ)^{\dagger}s_d + N\dot{q}_0$$

***where sd is a desired value for s, e.g. sd = [0, 0, 1], and N=(I – (LJ)† LJ) being the matrix spanning the null space of the LJ matrix. Verify that the for a chosen*** $\dot{q}_0$ ***the s measure does not change by plotting joint velocities and the s components.***

In order to implemet the above control law we added these lines in the **kdl_robot_vision_control.cpp**:

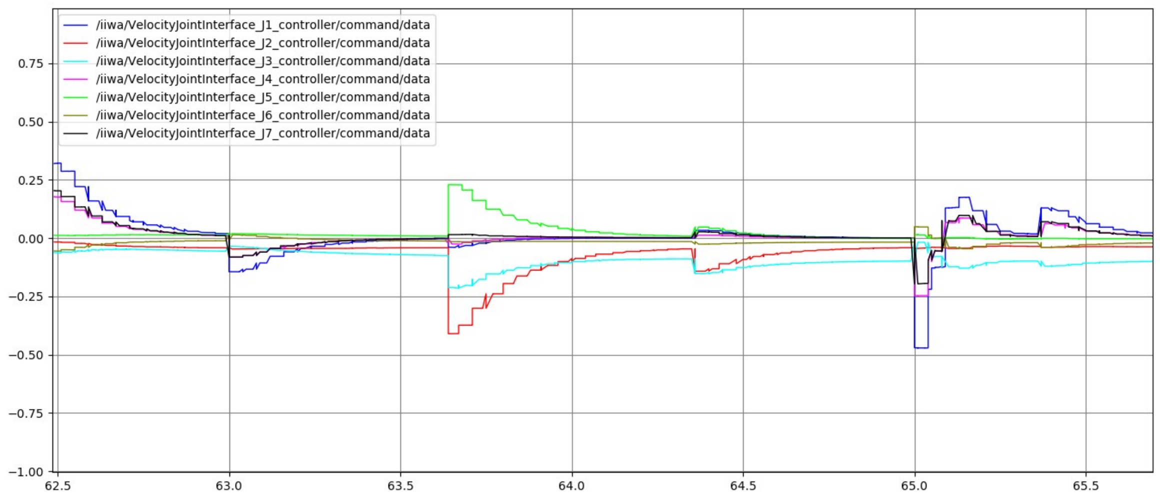


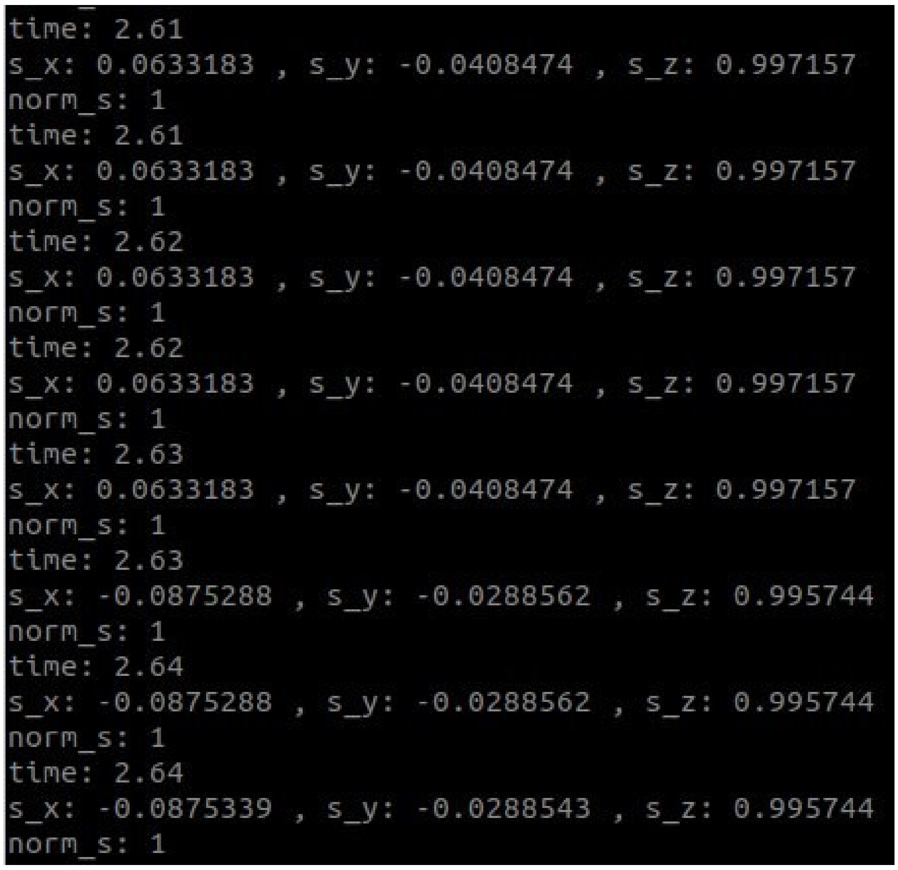
```cpp
// Compute L matrix
Eigen::Matrix<double,3,1> P_c_o = toEigen(cam_T_object.p);
Eigen::Matrix<double,3,1> s = P_c_o/P_c_o.norm();
Eigen::Matrix<double,3,3> R_c = toEigen(robot.getEEFrame().M);
Eigen::Matrix<double,3,3> S_brackets = (-1/P_c_o.norm())*(Eigen::Matrix<double,3,3>::Identity()-s*s.transpose());
Eigen::Matrix<double,6,6> RC;
RC = Eigen::MatrixXd::Zero(6,6);
RC.block(0,0,3,3) = R_c;
RC.block(3,3,3,3) = R_c;

Eigen::Matrix<double,3,6> L;
L = Eigen::MatrixXd::Zero(3,6);
L.block(0,0,3,3) = S_brackets;
L.block(0,3,3,3) = skew(s);
L = L*(RC.transpose());

// Compute N matrix
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);
```



```
Eigen::Vector3d s_d (0,0,1);

// resolved velocity control law
Eigen::MatrixXd J_pinv = J_cam.data.completeOrthogonalDecomposition().pseudoInverse();
dqd.data = 5*LJ_pinv*s_d + N*(qdi - toEigen(jnt_pos)); // nullo per l' orizzontalità

std::cout <<"s_x: " << s(0) << " , " <<"s_y: " << s(1) << " , " <<"s_z: " << s(2) << std::endl;
std::cout <<"norm_s: " << s.norm() <<  std::endl;
```

In this part of code we defined the vectors **s** and **c_P_o** and the matricies **L** and **RC** used in the formula. We also computed the **N** matrix and we chose appropriate gains.
At the end we printed the **s** components and the relative **norm**. Moreover we plotted the joint velocities with the **“rqt_plot”** command.





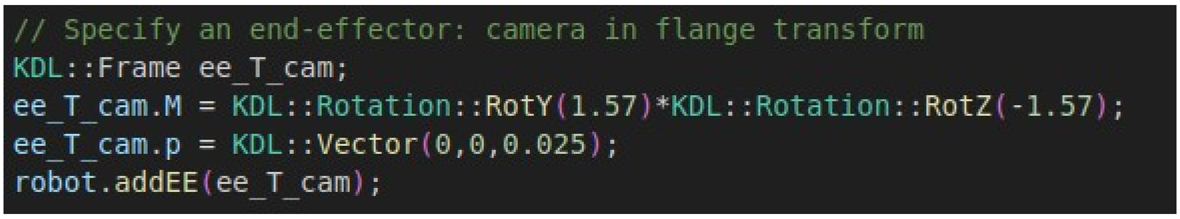
```
time: 2.61
s_x: 0.0633183 , s_y: -0.0408474 , s_z: 0.997157
norm_s: 1
time: 2.61
s_x: 0.0633183 , s_y: -0.0408474 , s_z: 0.997157
norm_s: 1
time: 2.62
s_x: 0.0633183 , s_y: -0.0408474 , s_z: 0.997157
norm_s: 1
time: 2.62
s_x: 0.0633183 , s_y: -0.0408474 , s_z: 0.997157
norm_s: 1
time: 2.63
s_x: 0.0633183 , s_y: -0.0408474 , s_z: 0.997157
norm_s: 1
time: 2.63
s_x: -0.0875288 , s_y: -0.0288562 , s_z: 0.995744
norm_s: 1
time: 2.64
s_x: -0.0875288 , s_y: -0.0288562 , s_z: 0.995744
norm_s: 1
time: 2.64
s_x: -0.0875339 , s_y: -0.0288543 , s_z: 0.995744
norm_s: 1
```

**(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.***

In order to do that we did a merge between the code used so far and the **kdl_robot_test.cpp** that we developed for the previous Homework.

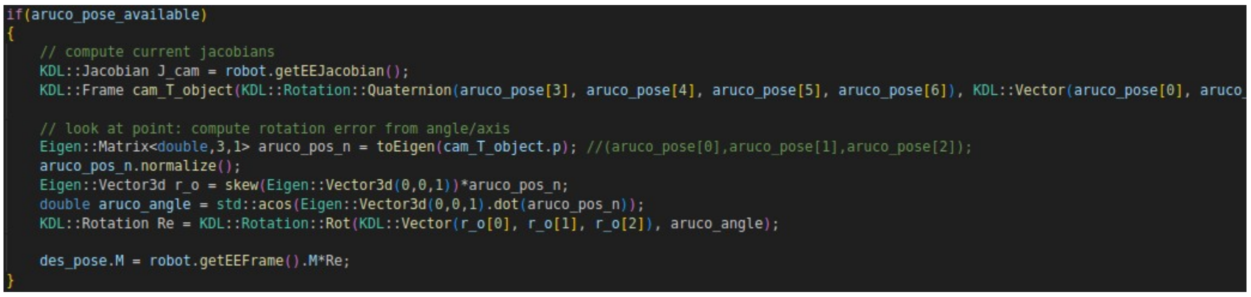
Starting from our **kdl_robot_test.cpp** we added to it the some lines to add the vision features.
First of all we defined the **aruco_pose** and **aruco_pose_available** variables and we added the **arucoPoseCallback** function that is used to put values in the **aruco_pose** variable.

Then we defined a **new frame** for the end-effector. This is different from the previous **Homework** since it was defined as an **Identity frame.**



```
// Specify an end-effector: camera in flange transform
KDL::Frame ee_T_cam;
ee_T_cam.M = KDL::Rotation::RotY(1.57)*KDL::Rotation::RotZ(-1.57);
ee_T_cam.p = KDL::Vector(0,0,0.025);
robot.addEE(ee_T_cam);
```

Then we added a new ***if*** inside the **while** in the **main** the .cpp file.



```
if(aruco_pose_available)
{
    // compute current jacobians
    KDL::Jacobian J_cam = robot.getEEJacobian();
    KDL::Frame cam_T_object(KDL::Rotation::Quaternion(aruco_pose[3], aruco_pose[4], aruco_pose[5], aruco_pose[6]), KDL::Vector(aruco_pose[0], aruco_

    // 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 added this ***if*** inside the ***if(robot_state_available)*** to make the script robust. Infact if the camera doesn't see the aruco, the robot still does the trajectory without stopping it.
What we do here is getting the **Jacobian** of the **End-Effector (camera)** and its frame.
Then we used the ***"look at point"*** developed yet in the **vision** file, but to change the desired orientation of the end effector ***(des_pose)*** we did the productbetween the **end effector frame** and **Re** which is the rotation matrix used to bring the **aruco_marker frame** with respect to the **camera frame** from its *detected position* to the *desired one* using the **angle/axis** representation. The angle is the **aruco_angle** variable and **r_o** is the axis.

As regard the plotting of the **cartesian_error_norm** we created a new *publisher* that publishes the ***cart_error_topic*** topic which is the error related to the position of the joints with respect to the trajectory that the robot follows.

```
ros::Publisher cart_error = n.advertise<std_msgs::Float64>("/iiwa/cart_error_topic", 1);
```

Then we declared a message for the norm of the cartesian error in addition to those related to the joints.

```
// Messages
std_msgs::Float64 tau1_msg, tau2_msg, tau3_msg, tau4_msg, tau5_msg, tau6_msg, tau7_msg, cart_err_norm_msg;
std_srvs::Empty pauseSrv;
```

Later we calculated joint error Matrix by the linear difference between the *desired position* **toEigen(des_pose.p)** and the *current position* of the robot's end effector **toEigen(robot.getEEFrame().p)**, and the result is assigned to the *errors matrix* ***"errors"***.

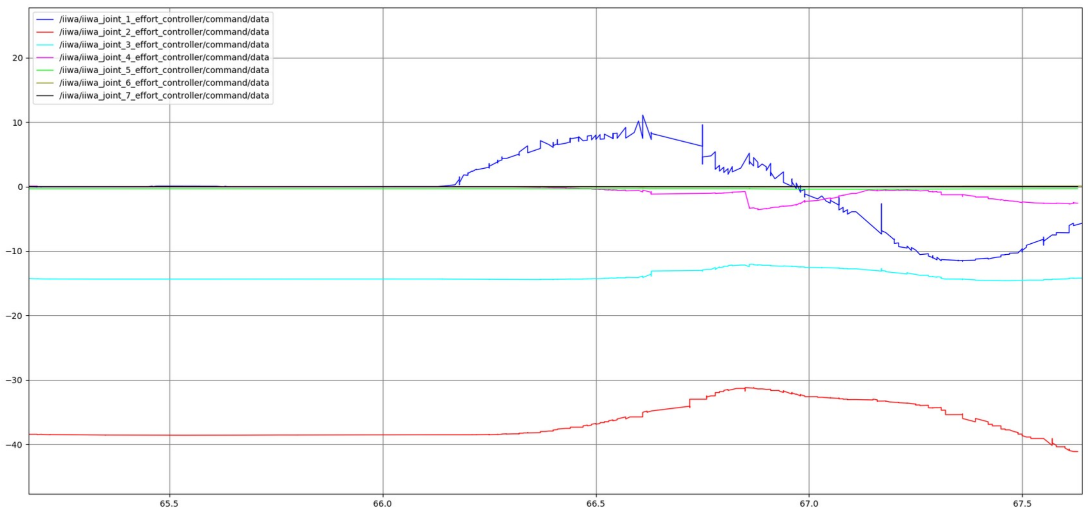
```
Eigen::MatrixXd errors = computeLinearError(toEigen(des_pose.p),toEigen(robot.getEEFrame().p));
```

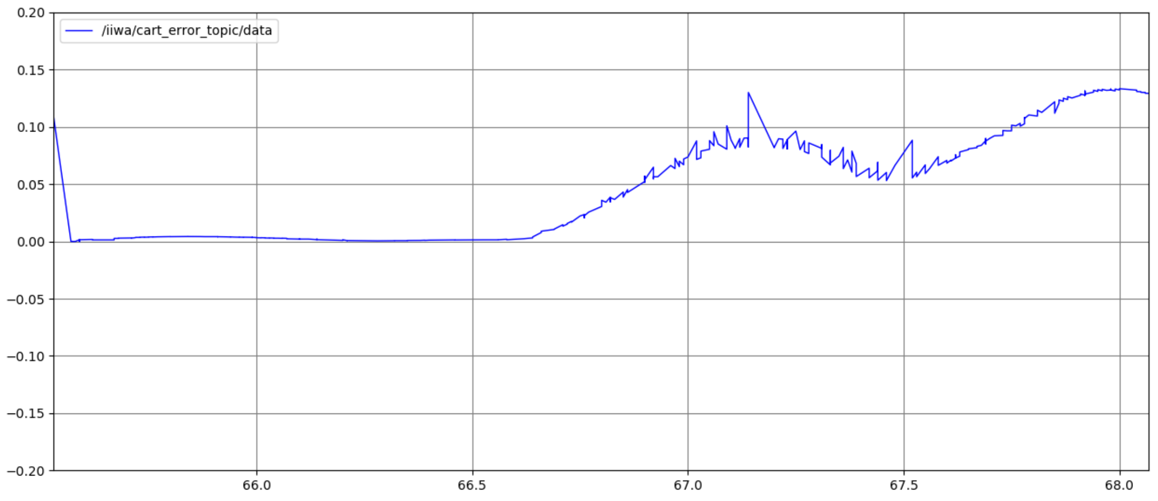
At the end we calculated the *norm of errors* and the result is assigned to the data field of the **cart_err_norm_msg** *message*. The **cart_err_norm_msg** *message* is assigned to the **"cart_error_topic"** which we can plot using the **rqt_plot** command.

```
cart_err_norm_msg.data = errors.norm();
cart_error.publish(cart_err_norm_msg);
```

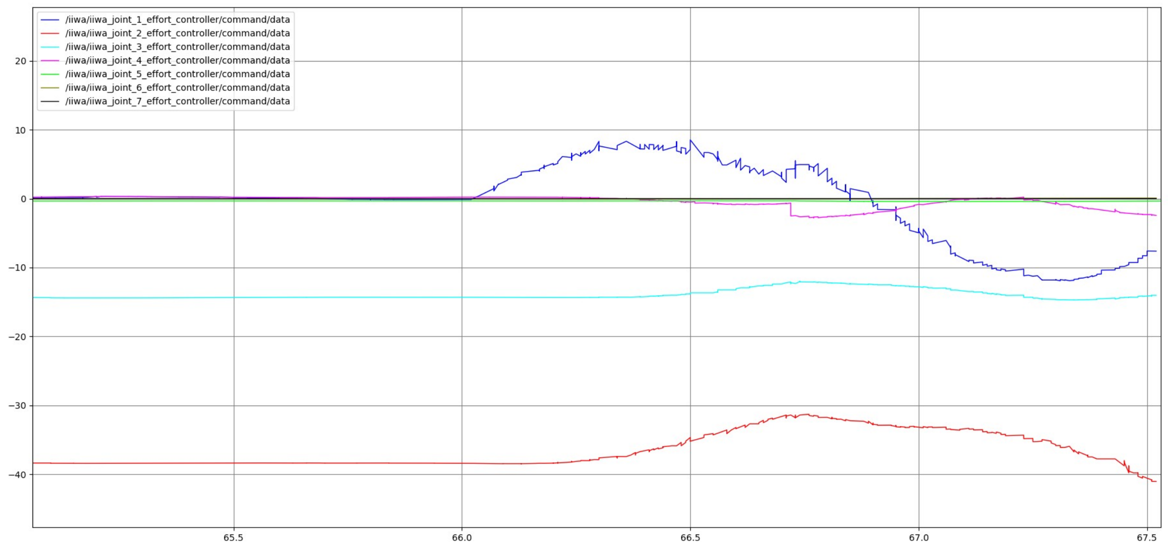
Plotting of *the results in terms of commanded joint torques and Cartesian error norm along the performed trajectory:*

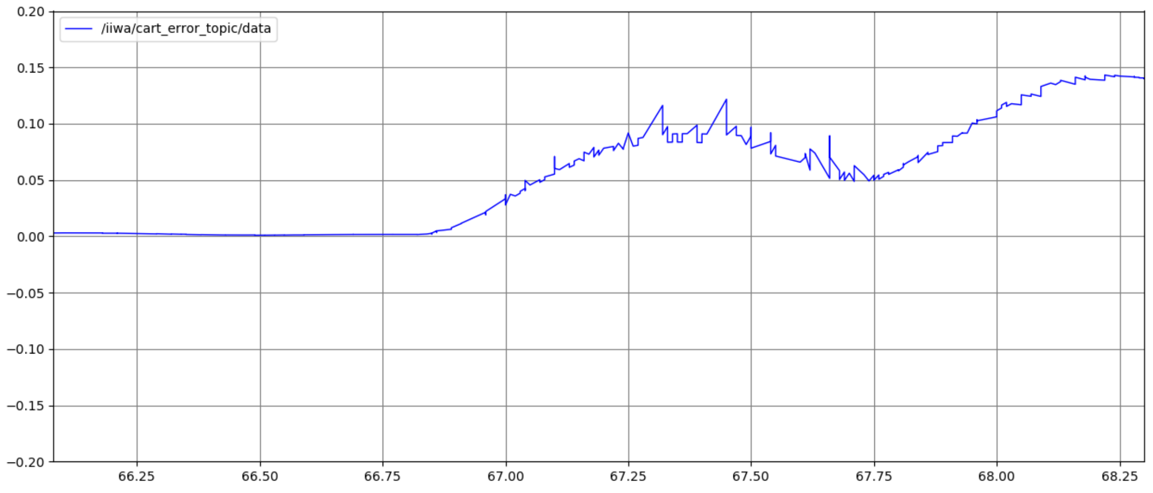
- Linear Trapezoidal trajectory with the joint space controller:



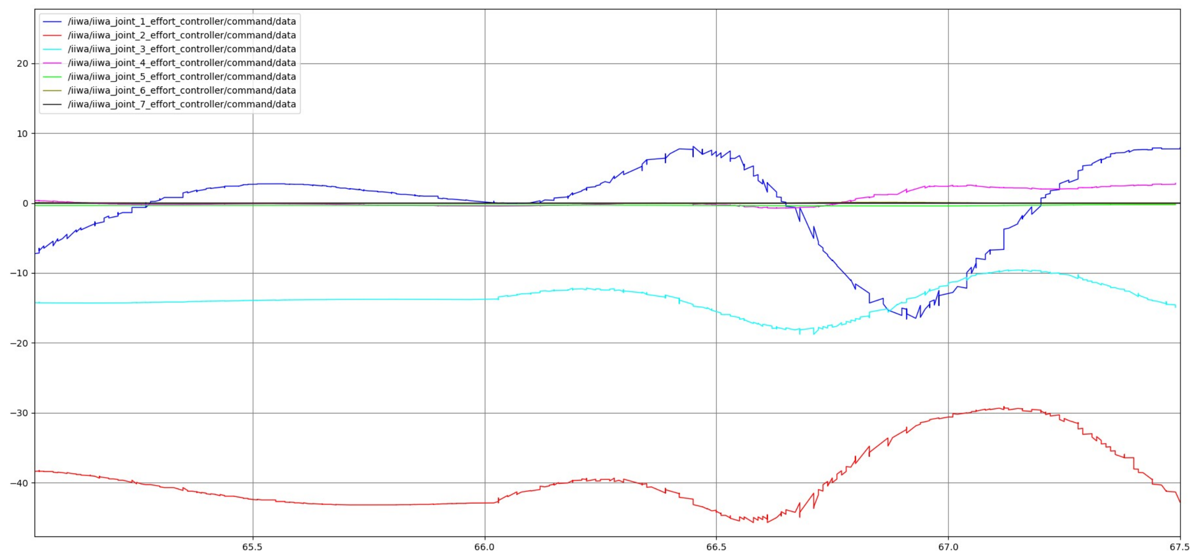


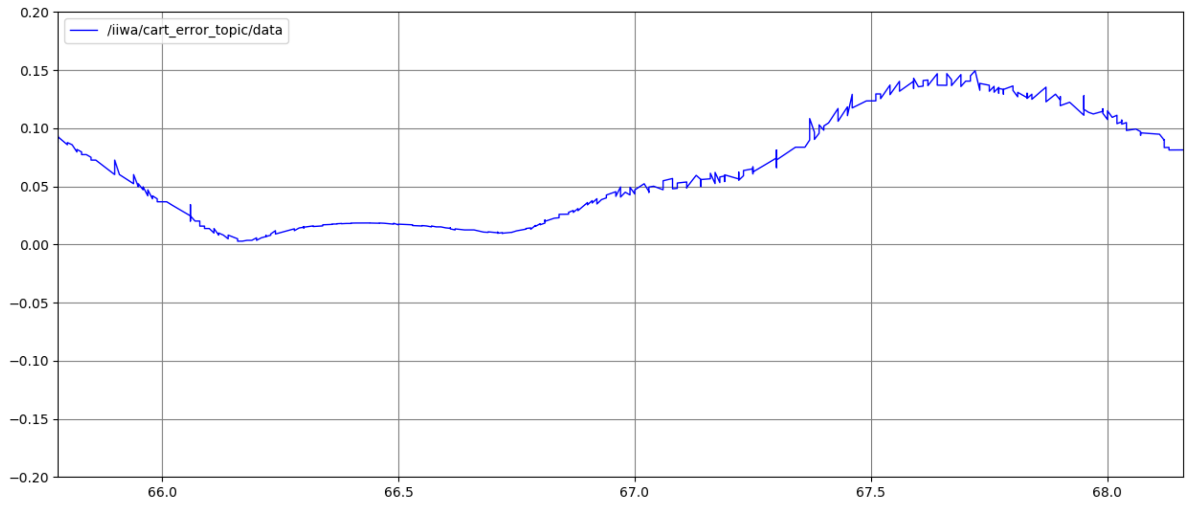
- Linear Cubic trajectory with the joint space controller:



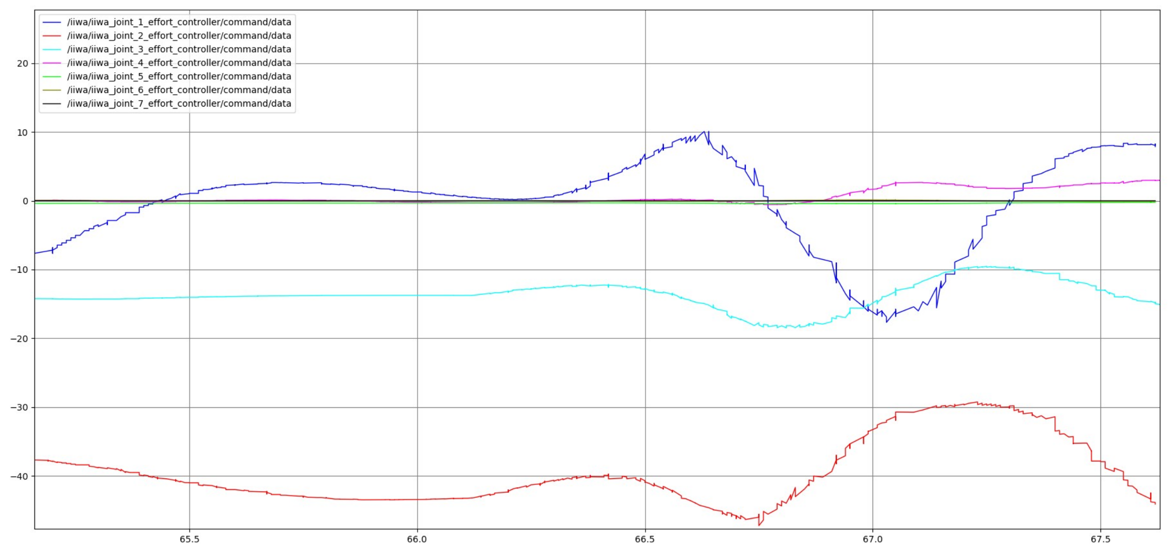


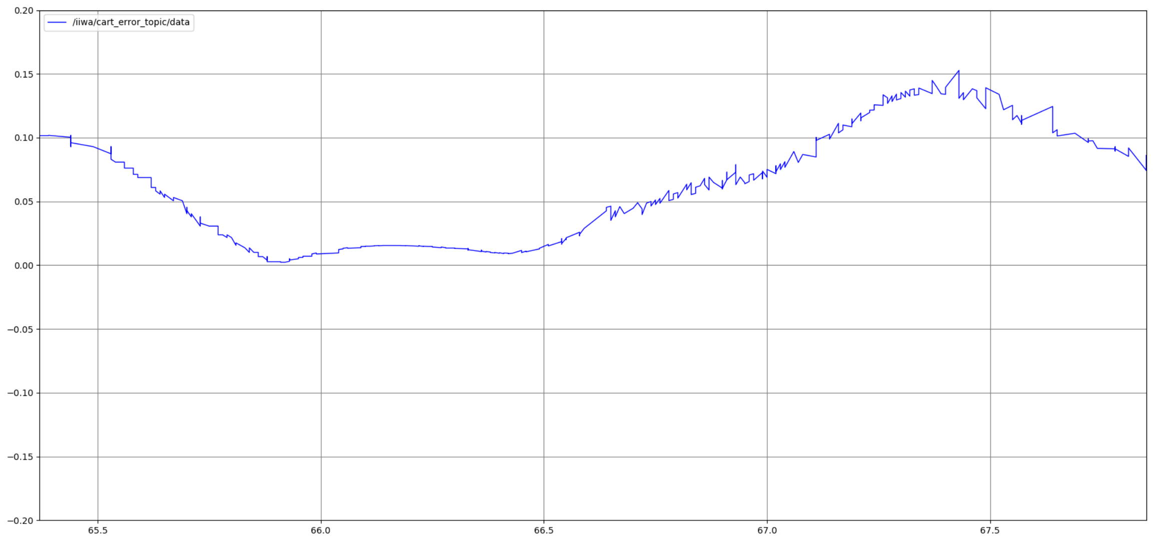
- Circular Cubic trajectory with the joint space controller:



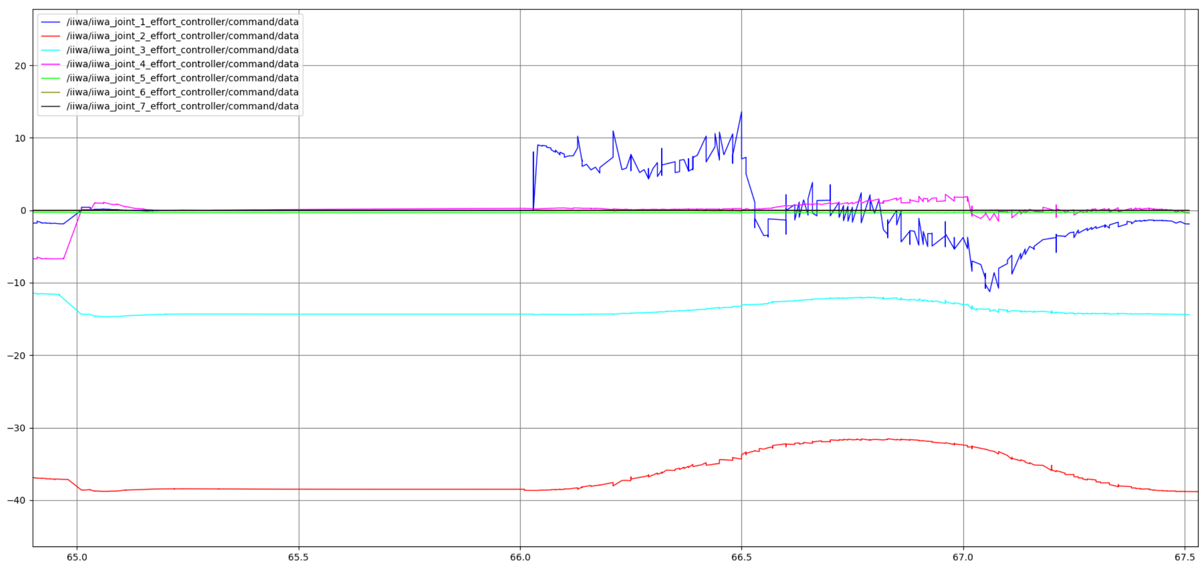


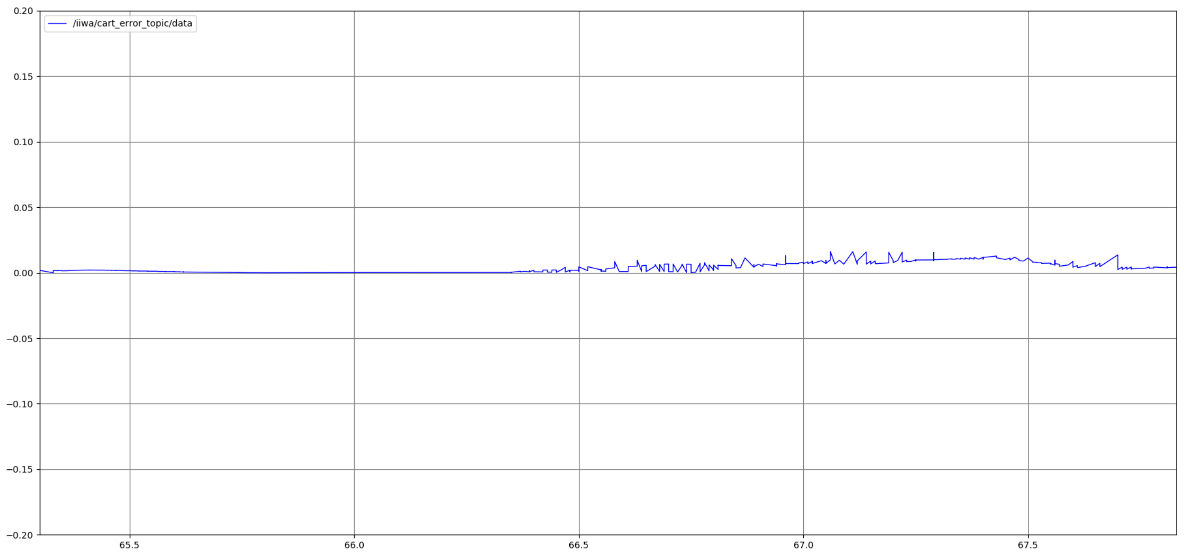
- Circular Trapezoidal trajectory with the joint space controller:



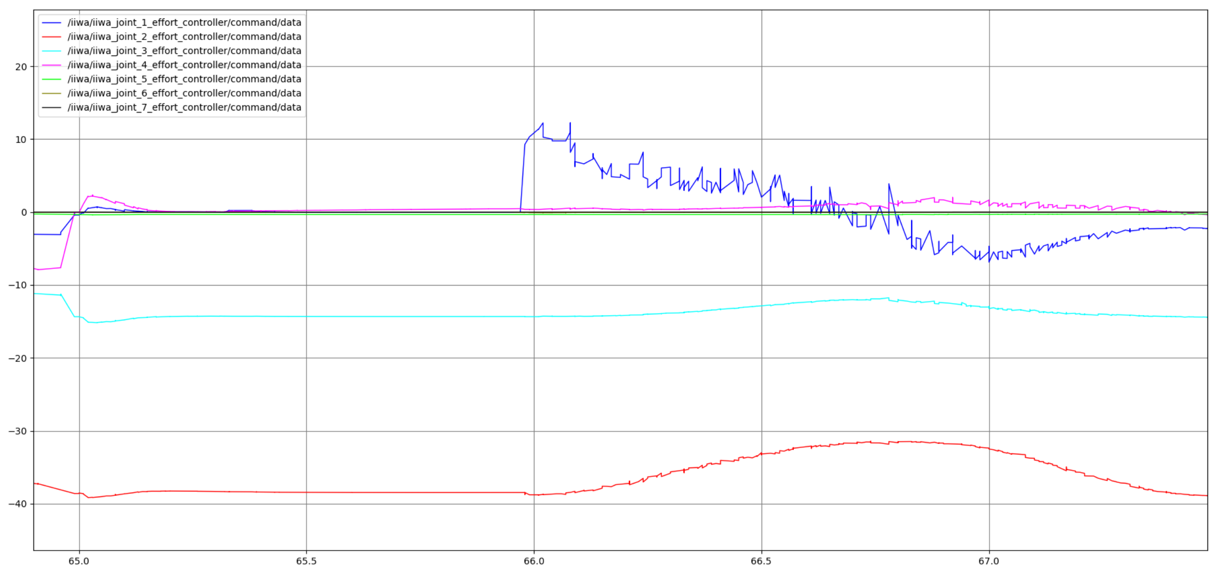


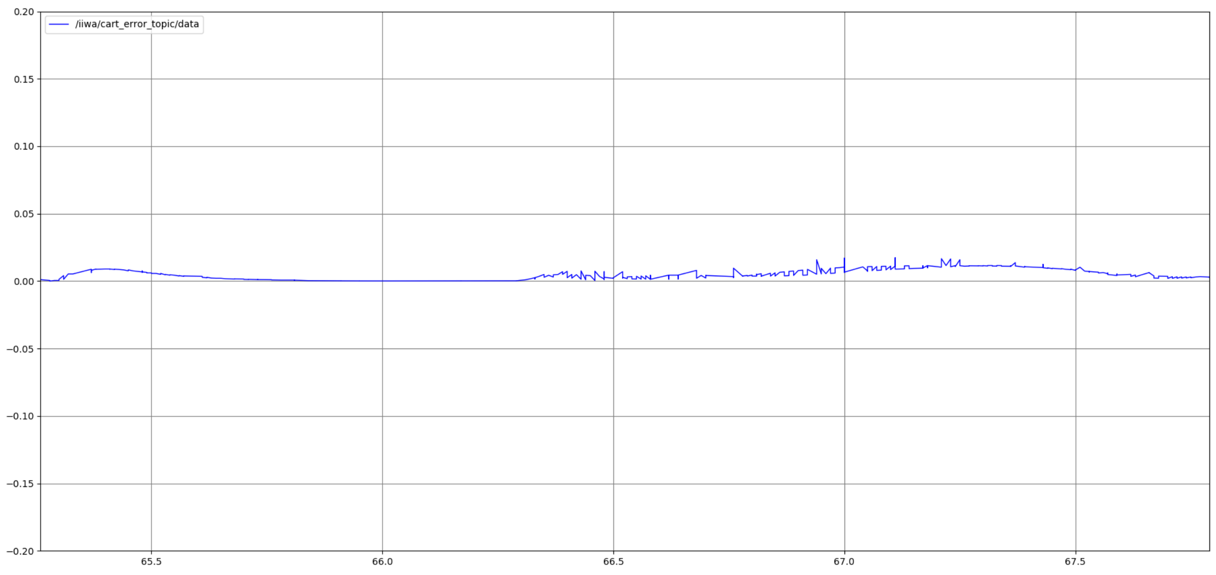
- Linear Trapezoidal trajectory with the operational space controller:



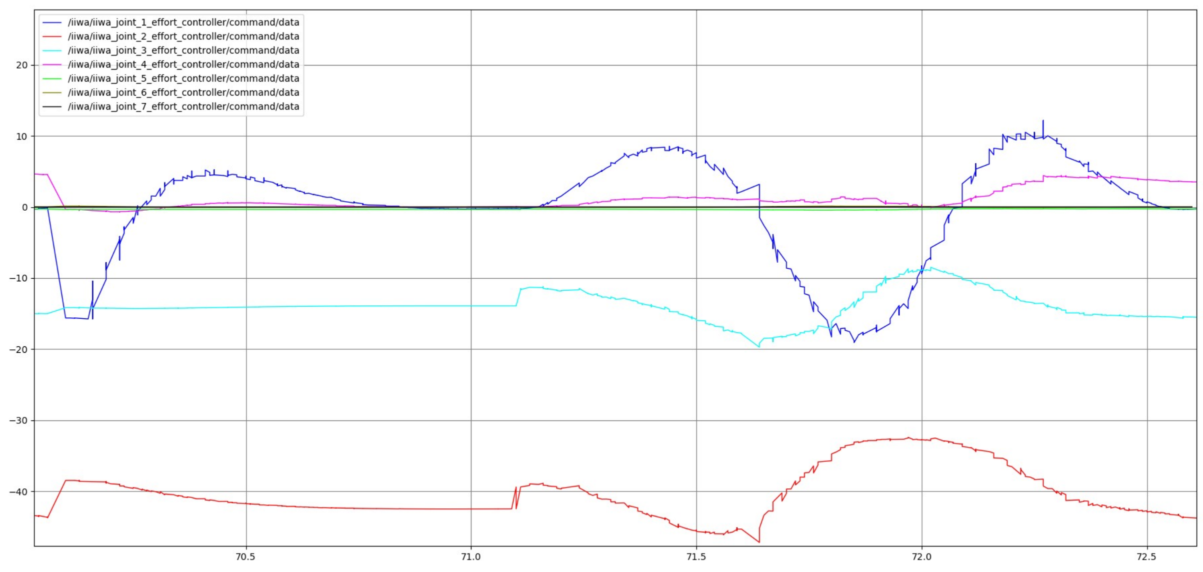


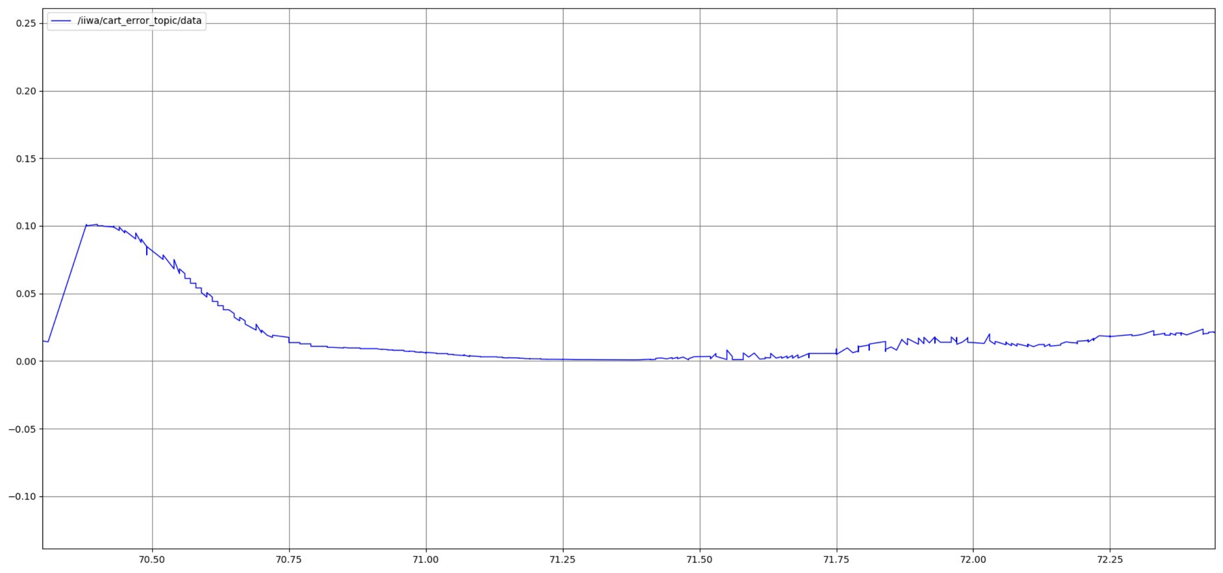
- Linear Cubic trajectory with the operational space controller:



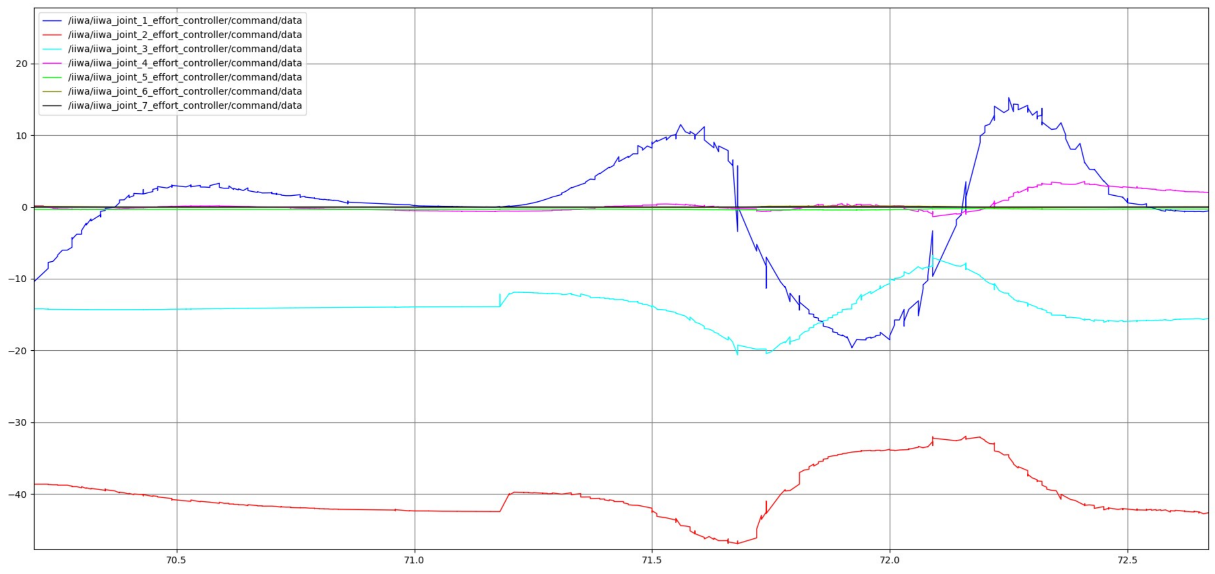


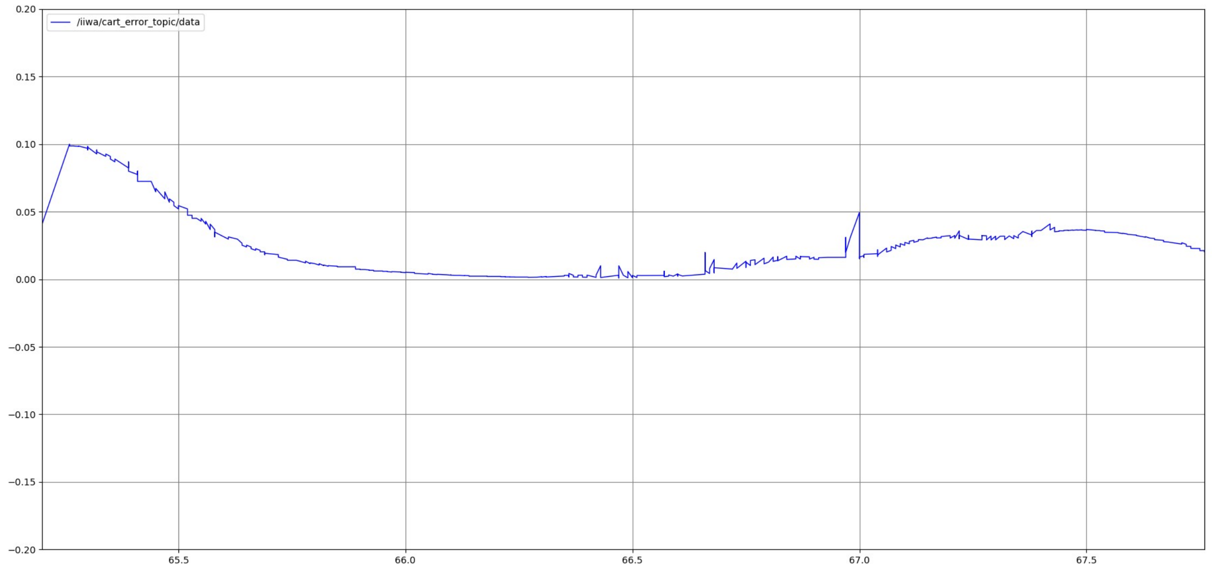
- Circular Cubic trajectory with the operational space controller:





- Circular Trapezoidal trajectory with the operational space controller:





You can find all the files at the following GitHub url: https://github.com/FalcoPietro/Homework3

Group members:
Davide Busco
Pietro Falco
Davide Rubinacci
Giuseppe Saggese