Report – Hands-on setup class

Students:

Nimrod Millenium Ndulue

Salvatore Del Peschio

Salvatore Oliviero

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.*

To create the circular object, it is necessary to establish a directory where an model.sdf file, representing the description of the circular object, will be placed. Additionally, other files essential for a detailed description of the object will be included in this directory.

The implementation of the .sdf file is as shown on the side:

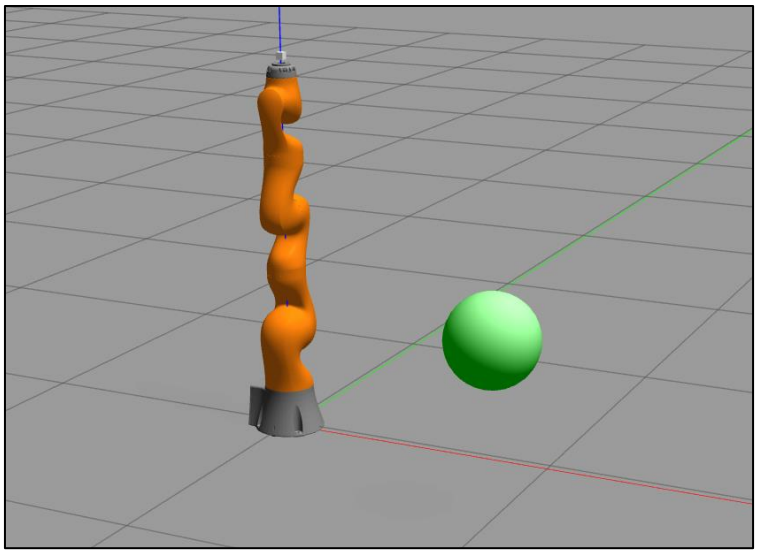
In this file it has been created a 15cm radius sphere colored in green, appropriately inserted in a collision box. Furthermore the inertia's values have been modified suitably.

It is important to notice, that for the object to maintain its configuration once the simulation has started (i.e., to prevent it from falling), it is necessary to specify that the object is static, that is done through the tag:

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

```
<?xml version="1.0"?>
<sdf version="1.6">
  <model name="cilinder_object">
    <link name="link">
      <inertial>
        <pose>0 0 0 0 0 0</pose>
        <mass>0.001</mass>
        <inertia>
          <ixx>1.8750008333333333e-06</ixx>
          <ixy>0.0</ixy>
          <ixz>0.0</ixz>
          <iyy>1.8750008333333333e-06</iyy>
          <iyz>0.0</iyz>
          <izz>1.8750008333333333e-06</izz>
        </inertia>
      </inertial>
      <visual name="front_visual">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <sphere>
            <radius>0.15</radius>
          </sphere>
        </geometry>
        <material>
          <ambient> 0 1 0 1 </ambient>
        </material>
      </visual>
      <!-- Hide the marker from the back -->
      <collision name="collision">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <box>
            <size>0.30 0.30 0.30</size>
          </box>
        </geometry>
      </collision>
    </link>
    <static> 1 </static>
  </model>
</sdf>
```

Once the object is created, to generate a world file containing it by default, it is necessary first to open Gazebo using the "iiwa_gazebo.launch" file where only the robot is present. Subsequently, add the path to the circular object, and then position it at the required location. The last step was to save the world in the proper directory of the iiwa_gazebo package.



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).*

The implementation of the task is performed by generating the necessary iiwa_gazebo_circular_object.launch file, wherein the path to the previously generated world file is incorporated including the "iiwa_world_circular_object.launch" file.

It was necessary to include the path of the models contained in iiwa_circular_object.world, this was done through the following instruction:

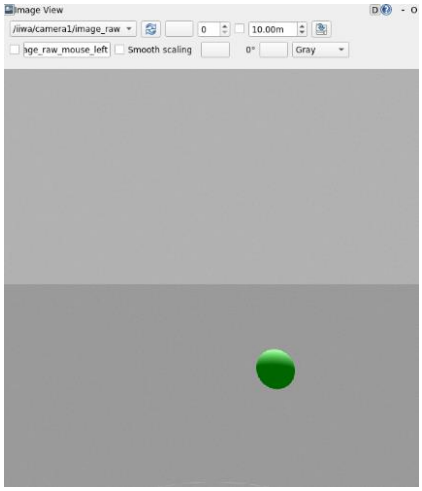
```
<env name=”GAZEBO_MODEL_PATH” value=”$(find iiwa_gazebo)/models:$(optenv GAZEBO_MODEL_PATH)” />
```

This sets the GAZEBO_MODEL_PATH environment variable to include the path to the models within the iiwa_gazebo package.

In the next page the code of iiwa_gazebo_circular_object.launch and iiwa_world_circular _object.launch are respectively reported:

```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 namespace 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="PositionJointInterface" />
    <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>
```

```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="false"/>
    <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="iiwa14"/>

    <!-- 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>

    <!-- 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>
```



After launching the file, to see the object through the camera its configuration was changed.

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.*

A blob detection algorithm in c++ was used to perform this task. A Blob is a group of connected pixels in an image that share some common property (E.g, grayscale value).

This code reads the image from cv_ptr->image and converts it from BGR to grayscale using cvtColor() function. The resulting grayscale image is stored in the im variable. Subsequently, we set the Blob Detector with defaults values.

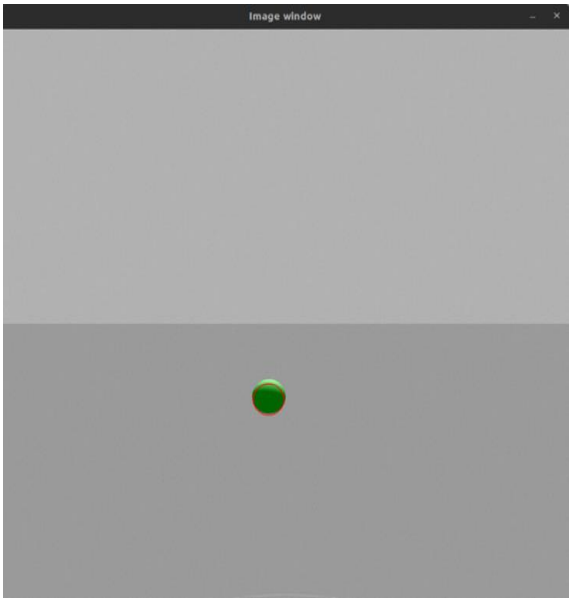
```
// Read image
cv::Mat image_grayscale;
cv::cvtColor(cv_ptr->image, image_grayscale, cv::COLOR_BGR2GRAY);

// Set up the detector with default parameters.
cv::SimpleBlobDetector::Params params;
cv::Ptr<cv::SimpleBlobDetector> detector = cv::SimpleBlobDetector::create(params);

// Detect blobs.
std::vector<cv::KeyPoint> keypoints;
detector->detect(image_grayscale, keypoints );

// Draw detected blobs as red circles.
// DrawMatchesFlags::DRAW_RICH_KEYPOINTS flag ensures the size of the circle corresponds to the size of blob
cv::Mat image_with_keypoints;
cv::drawKeypoints(cv_ptr->image, keypoints, image_with_keypoints, cv::Scalar(0,0,255), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

// Update GUI Window
cv::imshow(OPENCV_WINDOW, image_with_keypoints);
cv::waitKey(3);

// Output modified video stream
image_pub_.publish(cv_ptr->toImageMsg());
```

Through the following instruction we were able to detect blobs in the grayscale image using the instance detector of the SimpleBlobDetector class:

```
Detector->detect(im,keypoints);
```

The detected Keypoints are stored in a vector of keypoints. We then draw red circles at the positions of the detected blobs on the original image and then show it in a window.

On the side it is shown the result of the algorithm.



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 vision-based 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.*

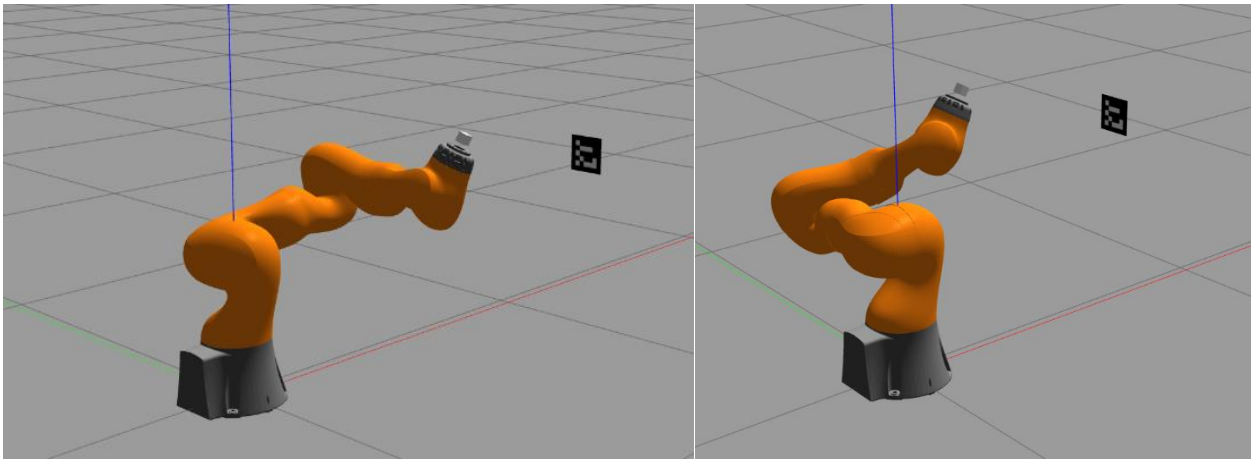
The look-at-point control was developed as follows:

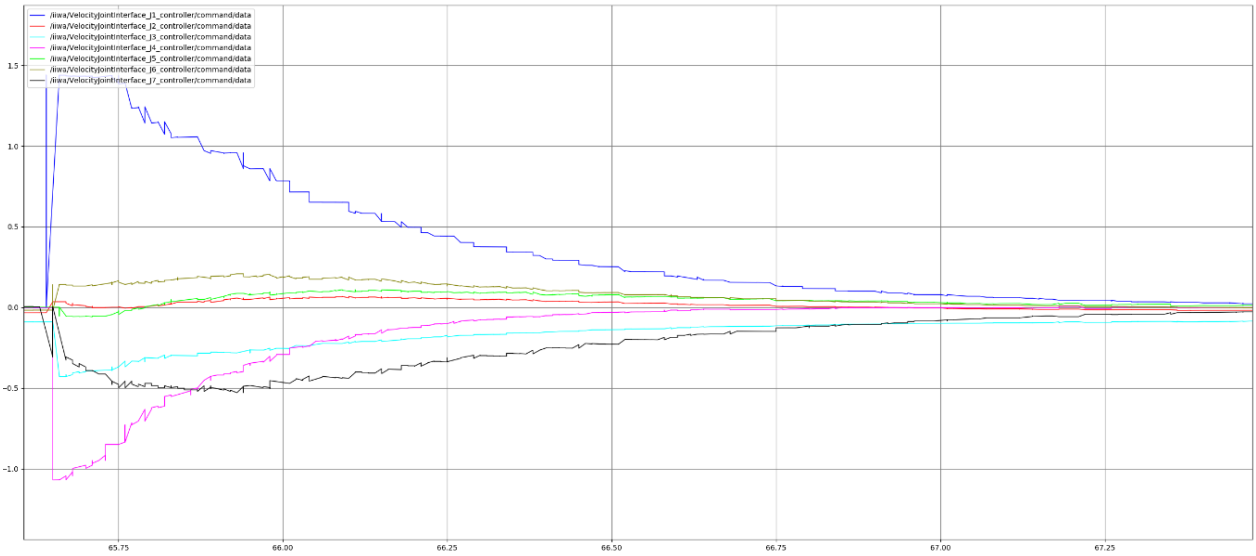
```
////////////////////////////////////////////////////////////////
// HW 2.a
// Desired orientation
Eigen::Matrix<double,3,3> R_d_ZYX = toEigen(KDL::Rotation::EulerZYX(-90.0*toRadians, 15.0*toRadians, -105.0*toRadians));
Eigen::Matrix<double,3,1> e_o_offset = computeOrientationError(R_d_ZYX, toEigen(robot.getEEFrame().M));
// Desired position
Eigen::Matrix<double,3,1> p_offset(0.0, 0.0, 0.60);
Eigen::Matrix<double,3,1> p_cart_cam_to_object = toEigen(robot.getEEFrame().M)*toEigen(cam_T_object.p);
Eigen::Matrix<double,3,1> p_cart_object = toEigen(robot.getEEFrame().p) + p_cart_cam_to_object;
Eigen::Vector3d p_cart_offset = p_cart_object - R_d_ZYX*p_offset;

Eigen::Matrix<double,3,1> e_p = computeLinearError(p_cart_offset, toEigen(robot.getEEFrame().p));
Eigen::Matrix<double,6,1> x_tilde = Eigen::Matrix<double,6,1>::Zero();

x_tilde << e_p,  e_o_offset[0], e_o_offset[1], e_o_offset[2];

// resolved velocity control low
Eigen::MatrixXd J_pinv = J_cam.data.completeOrthogonalDecomposition().pseudoInverse();

////////////////////////////////////////////////////////////////
```

A rotation matrix is specified using ZYX Euler angles, which is used to calculate the orientation error. An offset vector with a 60cm component along the Z-axis is created and transformed into the desired frame through pre-multiplication.

This operation allows us to obtain the positional offset between our camera and the Aruco marker. With this value, it is possible to calculate the desired position of the end-effector and, consequently, the linear error. Once both errors are obtained, the same control law from the original file is used to calculate the torques of the motors.

In the next page are reported the initial and final positions of the robot in a simulation of the requested algorithm, along with the plot of the joint velocities.





b) *An improved look-at-point algorithm can be devised by noticing that the task is belonging to S2.*

$$s = \frac{{}^{c}P_o}{||{}^{c}P_o||} \in \mathbb{S}^2,$$

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

$$L(s) = \left[-\frac{1}{||{}^{c}P_o||} \left(I - ss^T \right) \quad S(s) \right] R \in \mathbb{R}^{3\times 6} \quad \text{with} \quad R = \begin{bmatrix} R_c & 0 \\ 0 & R_c \end{bmatrix}$$

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.

```
////////////////////////////////////////////////////////////////
// HW 2.b

// Compute L
Eigen::Matrix<double,3,1> c_Po = toEigen(cam_T_object.p);
s = c_Po/c_Po.norm();
Eigen::Matrix<double,3,3> R_c = toEigen(robot.getEEFrame().M);
Eigen::Matrix<double,3,3> L_block = (-1/c_Po.norm())*(Eigen::Matrix<double,3,3>::Identity() - s*s.transpose());
Eigen::Matrix<double,3,6> L = Eigen::Matrix<double,3,6>::Zero();
Eigen::Matrix<double,6,6> R_c_big = Eigen::Matrix<double,6,6>::Zero();
R_c_big.block(0,0,3,3) = R_c;
R_c_big.block(3,3,3,3) = R_c;
L.block(0,0,3,3) = L_block;
L.block(0,3,3,3) = skew(s);
L = L*(R_c_big.transpose());

// Calcolo 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);

// Manipulability measure
Eigen::Matrix<double,7,1> prev_q = Eigen::Matrix<double,7,1>::Zero();
Eigen::Matrix<double,7,1> q_0_dot = Eigen::Matrix<double,7,1>::Zero();
manip_measure = sqrt((toEigen(J_cam)*toEigen(J_cam).transpose()).determinant());
double manip_measure_diff = manip_measure - prev_manip_measure;
prev_manip_measure = manip_measure;
for (int i = 0; i < robot.getNrJnts(); i++){
    if (prev_q.norm() != 0 && robot.getJntValues()[i]-prev_q[i])
            q_0_dot[i] = manip_measure_diff / (robot.getJntValues()[i]-prev_q[i]);
    prev_q[i] = robot.getJntValues()[i];
}

// // Calcolo q_dot
// dqd.data = 10*LJ_pinv*Eigen::Vector3d(0,0,1) + 5*N*(qdi - toEigen(jnt_pos)); // original
dqd.data = 10*LJ_pinv*Eigen::Vector3d(0,0,1) + 5*N*(q_0_dot + (qdi - toEigen(jnt_pos))); // using manipulability measure
////////////////////////////////////////////////////////////////
```

The provided code computes the control law as requested; all the elements necessary for the formula were obtained:

- The variable c_{P_o} represent the position of the Aruco Marker relative to the camera frame.
- The matrix L_{block} is used to construct the matrix L.
- The matrix N is computed, representing the projector in the null space of LJ.
- The vector $\dot{q}$ is computed using the control law. This vector represents the desired joint velocities of the robot.

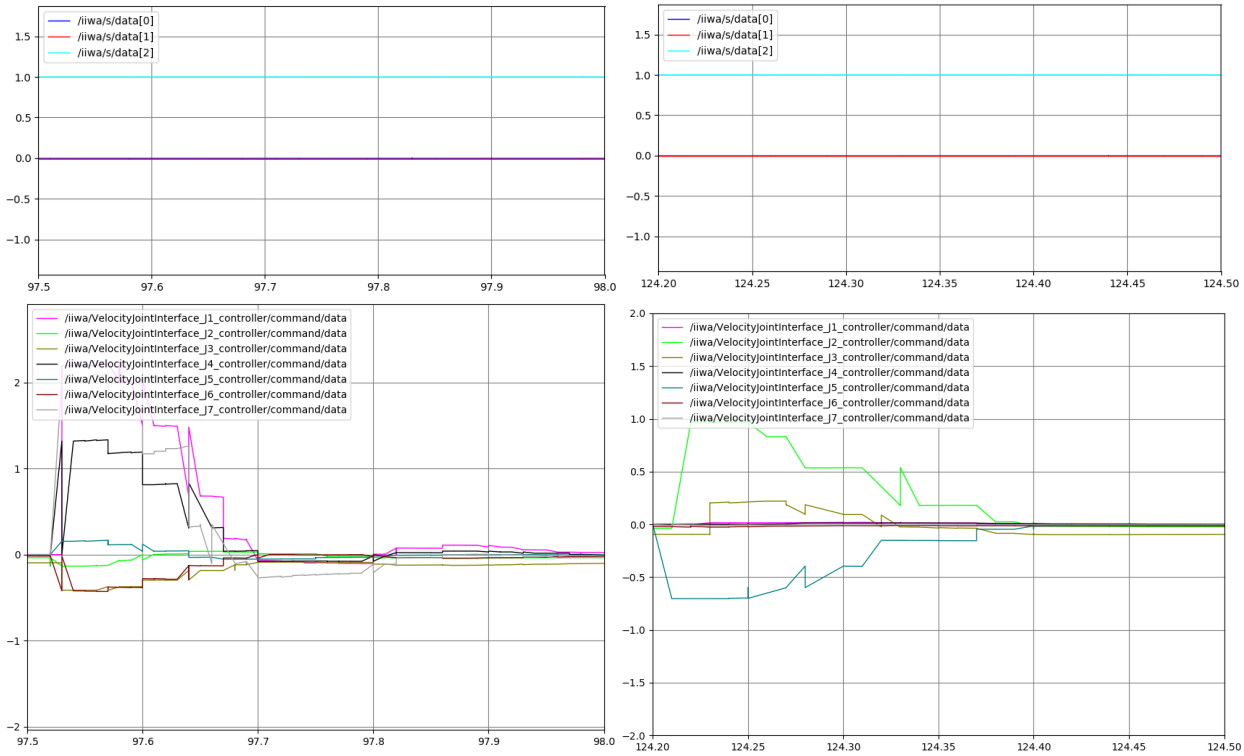
This code guides the robot to orient the camera towards the Aruco marker. To meet the requirements of the task, it was necessary to consider two values of $\dot{q}_0$, one of which had already been provided, while the other was chosen to be calculated using the manipulability measure, as reported below:

$$\dot{q}_0 = k_0 \left(\frac{\partial w(q)}{\partial q} \right)^T \quad with \quad w(q) = \sqrt{J(q)J(q)^T}$$

In the code, the formula has been implemented, based on the need for discretization, transitioning from a differential equation to a difference equation.

Subsequently, we have verified that, following the aruco marker, for the two chosen values of $\dot{q}_0$ the measure of s does not change. Moreover, the joint velocity for the $\dot{q}_0$ defined utilizing the distance from the initial position and the one containing also the manipulability measure are respectively plotted.

Original $\dot{q}_0$ *Manipolability measure* $\dot{q}_0$



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 eo 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.*

To address this task, the first step was to setup the simulation environment. We have created a new launch file iiwa_gazebo_aruco_effort.launch where effort controllers for each joint was created. Subsequently, we imported the controllers, the inverse kinematics function and modified the kdl_robot_test from the previous homework, that was used to setup the initial position of the robot and sent torques commands to it, merging it with the

vision-based controller. Two different algorithms have been implemented: one for the cartesian space and one for the joint space.

The obtained file kdl_robot_vision_control_effort.cpp is able to perform the tracking of the desired trajectory while pointing the camera towards the marker. We worked on the computation of the desired pose of the end-effector to make sure that at each time step of the trajectory the camera will be correctly oriented. Following is reported the operations applied to the orientation matrix that represents the desired pose in both the operational and joint spaces.

```
// look at point: compute rotation error from angle/axis
Eigen::Matrix<double,3,1> aruco_pos_n = toEigen(cam_T_object.p);
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);
qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3], jnt_pos[4], jnt_pos[5], jnt_pos[6];

// Joint space inverse dynamics control inverse kinematics
des_pose.M=robot.getEEFrame().M*Re*ee_T_cam.M.Inverse();
robot.getInverseKinematics(des_pose, des_cart_vel, des_cart_acc, qd, dqd, ddqd);
double Kp = 50;
double Kd = 2*sqrt(Kp);
tau = controller_.idCntr(qd, dqd, ddqd, Kp, Kd);

// Cartesian space inverse dynamics control
des_pose.M = robot.getEEFrame().M*Re;
robot.getInverseKinematics(des_pose, des_cart_vel, des_cart_acc, qd, dqd, ddqd);
double Kp = 100;
double Ko = 100;
tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc, Kp, Ko, 2*sqrt(Kp), 2*sqrt(Ko));

// Compute cartesian error
error = ((robot.getEEFrame().p - ee_T_cam.p) - des_pose.p).Norm(); // joint space controller
error = ((robot.getEEFrame().p) - des_pose.p).Norm();              // cartesian space controller
```

In the joint space, a post-multiplication operation with the inverse of the rotation matrix of the camera in the end-effector frame was required to align the camera's frame with the end effector's frame, this was necessary to obtain the end-effector orientation in order to have the camera pointing towards the marker. This was mandatory as we realized that the control algorithm in the joints space was acting on the end effector rather than directly on the camera.

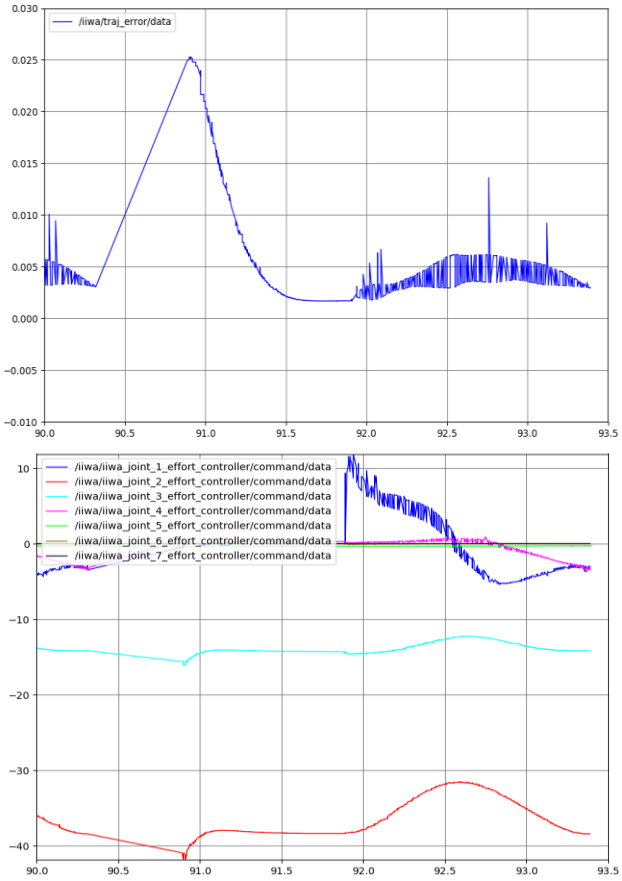
As for the the algorithm in the cartesian space, the following command was used:

```
des_pose.M = robot.getEEFrame().M * Re;
```

Post-multiplying the current orientation of the camera in the world frame with the orientation error, we align the camera's z-axis with the vector representing the pose of the aruco marker with respect to the camera.

Furthermore, we modified the way the error was computed to be able to plot the error in the cartesian space; note that in the computation of the error with the joint space controller it is mandatory to consider the translation of the camera from the end effector. Below are reported the plots of the error and torques:

Joint space controller:



Cartesian space controller: