DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Automation and Robotics Engineering

# ROBOTICS LAB

# HOMEWORK 3

## Implement a vision-based task

Instructor:
Mario Selvaggio

Student:
Francesco Grasso
P38000046

Accademic Year 2023/2024

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

**1.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.**

- A new folder named **circular_object** was created with the same files of the **aruco_marker** folder.

- The **model.sdf** file was modified modifing the name and the geometry of the **front_visual**:

```xml
<visual name="front_visual">
  <pose>0.00005 0 0 0 0 0</pose>
  <geometry>
    <cylinder>
      <radius>0.15</radius>
      <length>0.01</length>
    </cylinder>
  </geometry>
  <material>
      <script>
      <uri>model://circular_object/materials/scripts</uri>
      <uri>model://circular_object/materials/textures</uri>
      <name>Circle</name>
    </script>
  </material>
</visual>
```

**Figura 1:** model.sdf of circular_object

- The **name** and **description** parameters have been changed to **"circular_object"** in the **model.config** file.
- The **circle.material** file in the **circular_object/materials/scripts** folder has been modified based on the name of the **circle_green.png** texture present in **circular_object/materials /textures**.
- The model path has been added in the **insert** section of the new gazebo world.

The properties and the pose of the object have been changed as shown in the figure below.

- The world was saved in the folder /**iiwa_gazebo/worlds** with the name **iiwa_aruco_HW.world**.

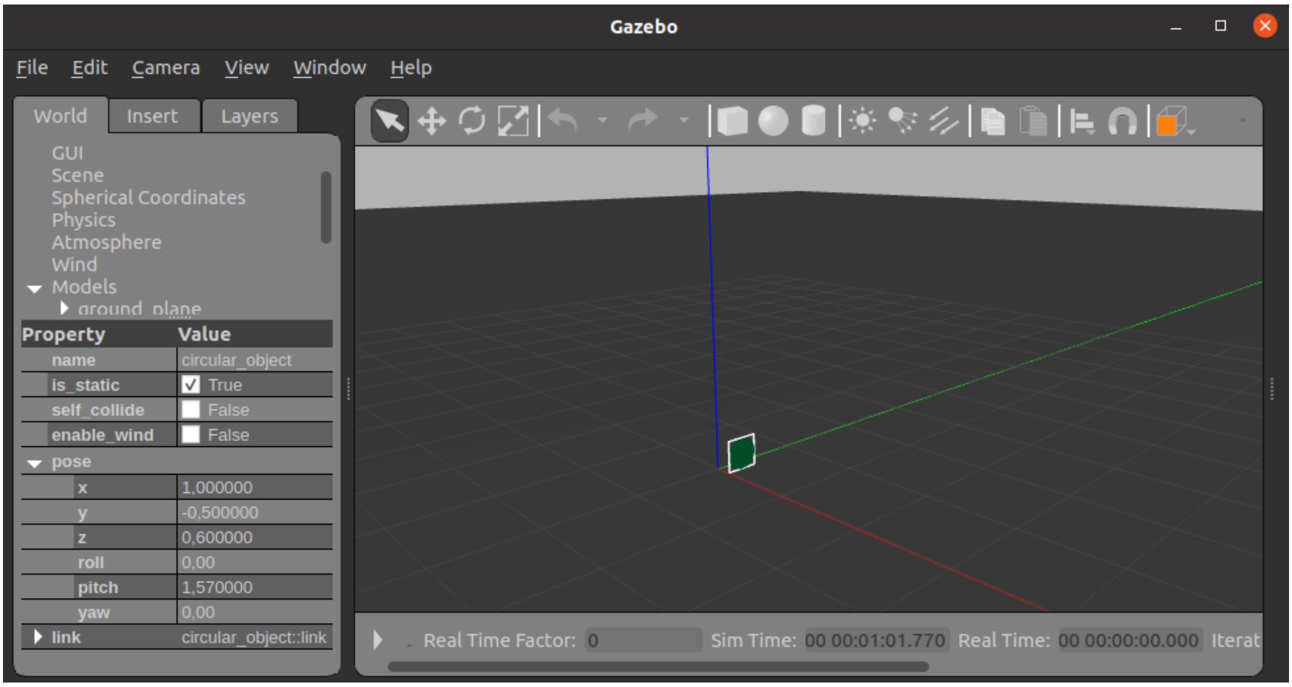


**Figura 2:** iiwa_aruco_HW.world

1.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).

```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>

    <!-- 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>
```

**Figura 3:** iiwa_gazebo_circular_object.launch

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

    <!-- Loads thee iiwa_aruco_HW.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_aruco_HW.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>
```

**Figura 4:** iiwa_world_circular_object.launch

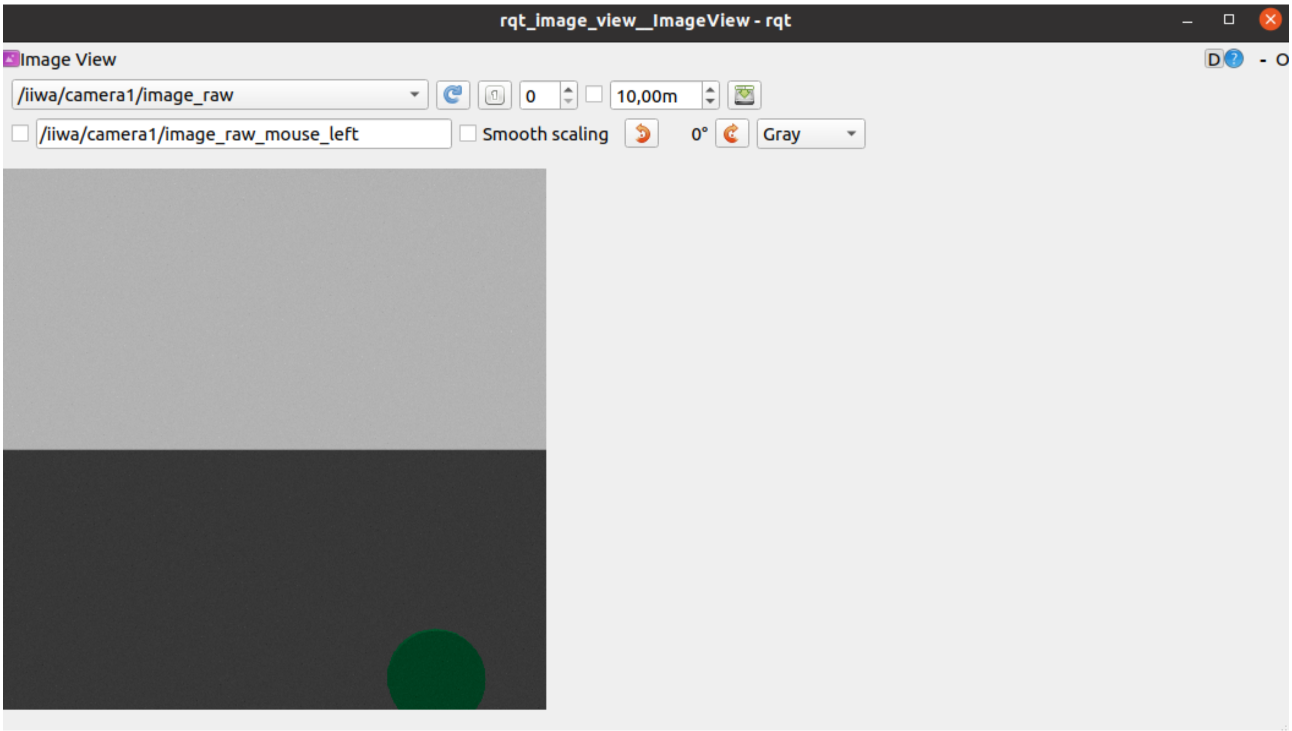


**Figura 5:** Image seen from the camera using **rqt_image_view**

The object position was changed to [1, 0, 1.5] to be fully visible by the camera:

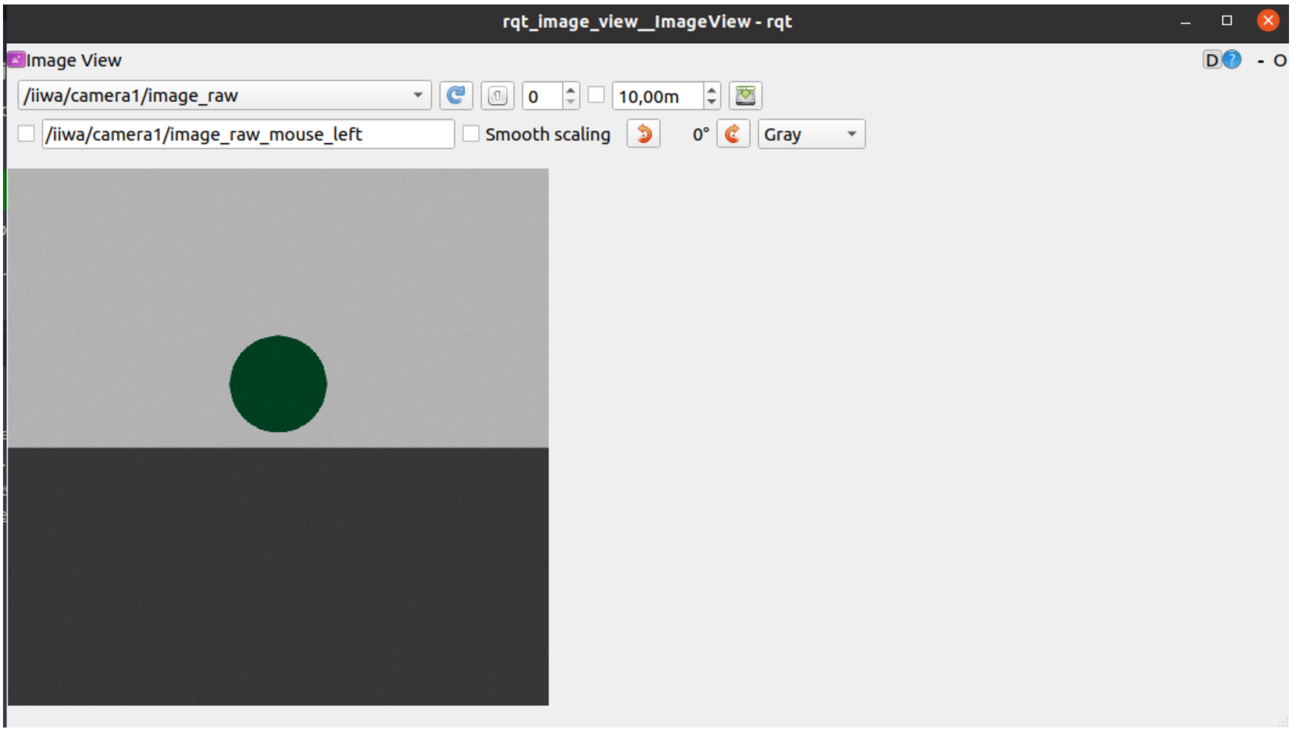


**Figura 6:** Image seen from the camera using **rqt_image_view** (new configuration)

1.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**

- **opencv_ros_node.cpp** was modified replacing the topic **"/usb_cam/image_raw"** with **"/usb_cam/camera1/image_raw"**.

- To detect the object via openCV functions has been implemented **The Hough gradient method** which is made up of two main stages.

The first stage involves edge detection and finding the possible circle centers and the second stage finds the best radius for each candidate center.

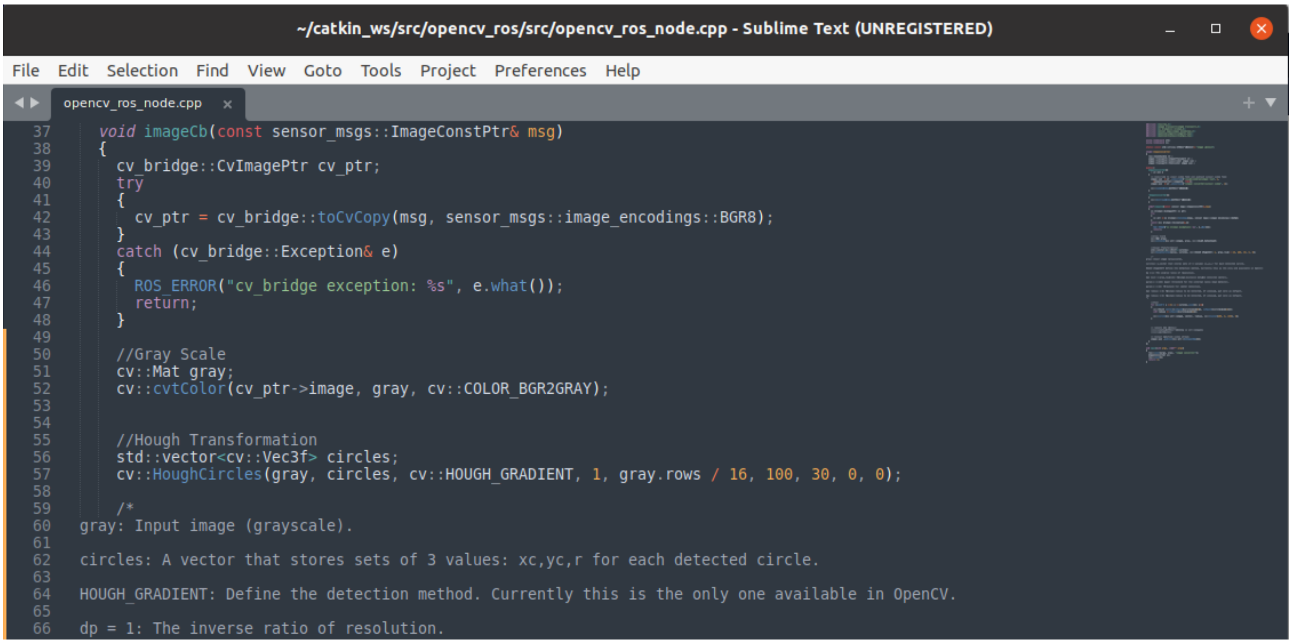


**Figura 7:** Hough Transformation


**Figura 8:** Plot function on the original frame

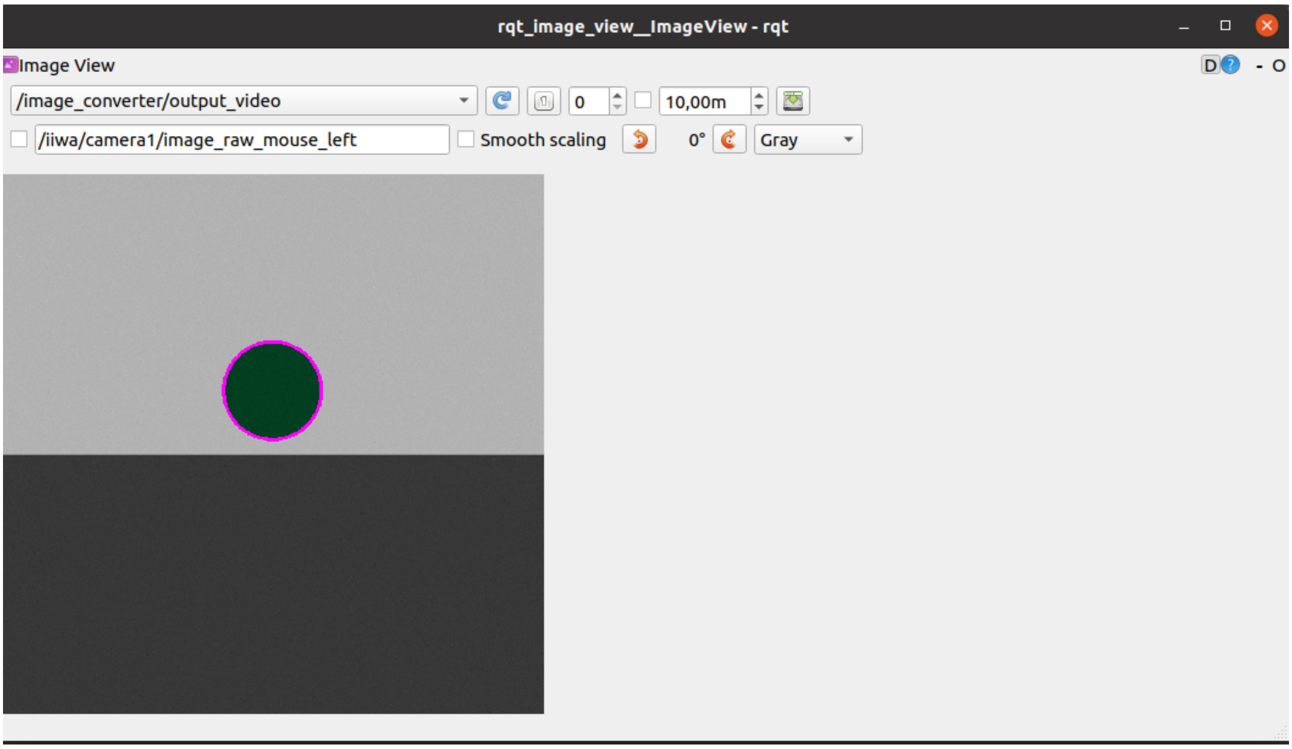


**Figura 9:** Image processed on **rqt_image_view**

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

2.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.

```cpp
/////////////////////////////// 2.a /////////////////////////////


KDL::Frame off_frame;

off_frame.p = cam_T_object.p - KDL::Vector(0,0,0.4);
off_frame.M = cam_T_object.M*KDL::Rotation::RotX(-3.14);

KDL::Frame des_frame=robot.getEEFrame()*off_frame;

Eigen::Matrix<double, 3, 1> e_o = computeOrientationError(toEigen(des_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(des_frame.p), toEigen(robot.getEEFrame().p));
Eigen::Matrix<double, 6, 1> x_tilde;

// e_o_w used to transform a 2D task in a 3D task, to don't rotate. In this way, we try to stay near the initial orientation.
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));

}
```

**Figura 10:** Alignment with position and orientation offset.

- As shown in the figure above, the file **kdl_robot_vision_control.cpp** has been changed so that the robot follows the marker with a constant offset.

- A new frame named **"off_frame"** has been defined putting it equal to the **"cam_T_object"** frame with a downward translation along the z-axis and a rotation around the x-axis.

- A desired frame was defined by multiplying the current end-effector frame and the offset frame.

- The errors between the desired pose and the current one have been calculated. A desired velocity vector named **"dqd.data"** was calculated using the concept of resolved velocity control.

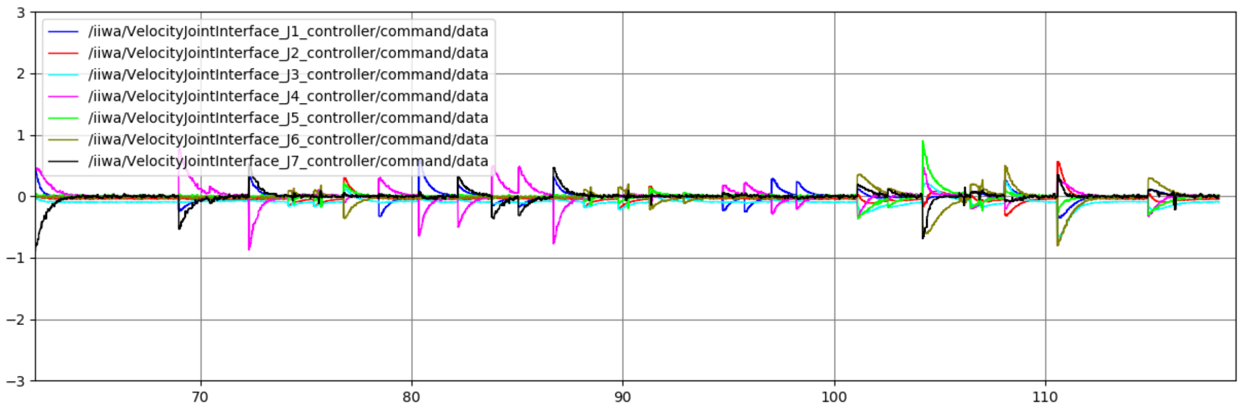


**Figura 11:** Velocity commands sent to the robot

The simulation was performed with the following terminal commands:

- **roslaunch aruco_ros usb_cam_aruco.launch camera:=/iiwa/camera1**
- **roslaunch iiwa_gazebo iiwa_gazebo_aruco.launch**
- **rosrun kdl_ros_control kdl_robot_vision_control**

**src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf**

- **rqt_plot**

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

$$s = \frac{P_o^c}{\|P_o^c\|} \in S^2 \tag{1}$$

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

$$L(s) = [-\frac{1}{\|P_o^c\|}(I - ss^T) \quad S(s)]R \in R^{3x6} \tag{2}$$

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

$$\dot{q} = k(LJ)^{\dagger} s_d + N\dot{q}_0, \tag{3}$$

**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 for a chosen $\dot{q}_0$ the s measure does not change by plotting joint velocities and the s components.**

To implement the control law specified above, the following lines of code have been added to **kdl_robot_vision_control.cpp**:

```cpp
///////////////////////////////// 2.b //////////////////////////////

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();


// R matrix

Eigen::Matrix<double,3,3> R_c   = toEigen(robot.getEEFrame().M);

Eigen::Matrix<double,6,6> R;
R = Eigen::MatrixXd::Zero(6,6);
R.block(0,0,3,3) = R_c;
R.block(3,3,3,3) = R_c;


// L matrix
Eigen::Matrix<double,3,3> matr;
matr = (-1/P_c_o.norm())*(Eigen::Matrix<double,3,3>::Identity() - s*s.transpose());

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


//N matrix

Eigen::MatrixXd LJ = L*toEigen(J_cam);
Eigen::MatrixXd LJ_pseudoinv  = LJ.completeOrthogonalDecomposition().pseudoInverse();
Eigen::MatrixXd N   = Eigen::Matrix<double,7,7>::Identity() - LJ_pseudoinv*LJ;

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

**Figura 12**

```
time: 92.19
s_x: -0.00612628, s_y -0.0198408, s_z 0.999784
norm_s:1
time: 92.19
s_x: -0.00612628, s_y -0.0198408, s_z 0.999784
norm_s:1
time: 92.2
s_x: -0.00612628, s_y -0.0198408, s_z 0.999784
norm_s:1
time: 92.2
s_x: -0.00615006, s_y -0.0198485, s_z 0.999784
norm_s:1
time: 92.21
s_x: -0.00615006, s_y -0.0198485, s_z 0.999784
norm_s:1
time: 92.21
s_x: -0.00615006, s_y -0.0198485, s_z 0.999784
```

**Figura 13:** s vector norm and components

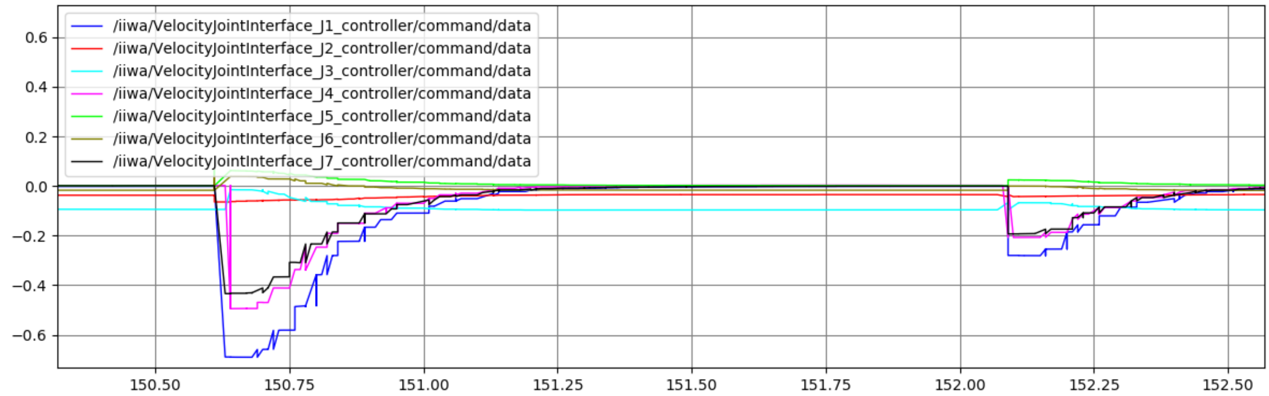


**Figura 14:** Velocity commands sent to the robot

**2.c) Develop a dynamic version of the vision-based controller. 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.**

Starting from **kdl_robot_test.cpp**, the following steps were followed:

- The **arucoPoseCallback** function with the associated subscriber was defined to obtain the pose of the aruco marker, as seen in

**kdl_robot_vision_control.cpp**.

• A new frame (camera frame) was defined for the End-Effector as seen in **kdl_robot_vision_control.cpp**.

• For the purposes of this simulation, **a linear trajectory with a cubic profile** have been chosen as example.

• As shown in the figure below, the following lines of code have been added considering a different desired pose.

```cpp
if(aruco_pose_available) {

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], aruc

 // 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.p = KDL::Vector(p.pos[0],p.pos[1],p.pos[2]);
des_pose.M=robot.getEEFrame().M*Re;

}
```

**Figura 15:** look at point with different desired pose

• A new publisher (**cart_norm_err**) was created in order to publish on the **iiwa/cart_norm_error** topic, the Cartesian error norm along the performed trajectory.

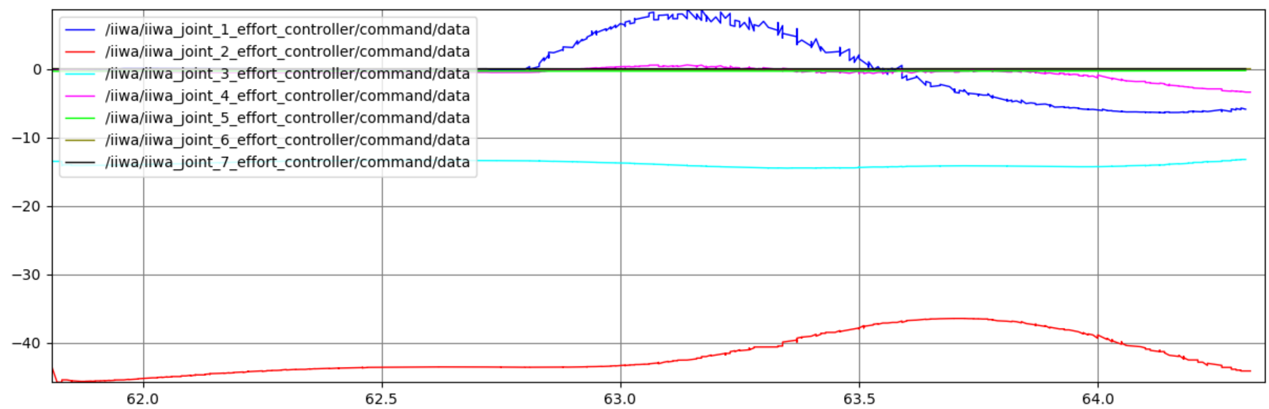


**Figura 16:** Commanded joint torques

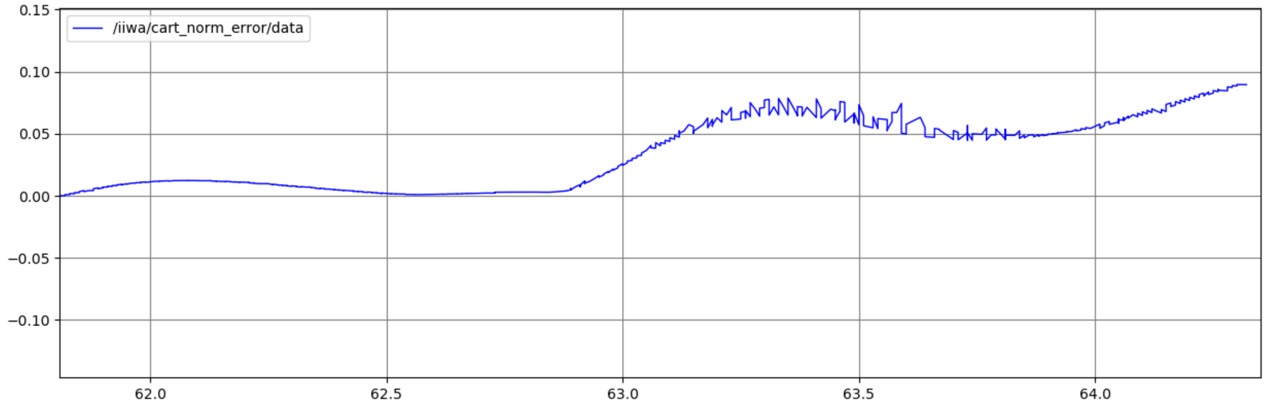


**Figura 17:** Cartesian error norm