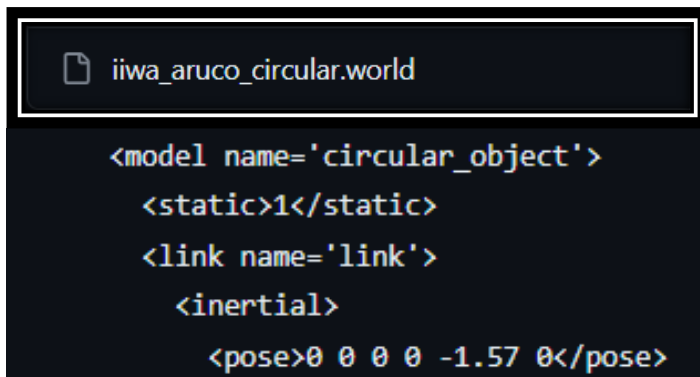


**Report – Homework 3**  
**Student: Matteo Langella**

# 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 in to the “*iiwa\_gazebo/worlds/*” folder.




```
<model name='circular_object'>
  <static>1</static>
  <link name='link'>
    <inertial>
      <pose>0 0 0 0 -1.57 0</pose>
```

I opened the object in gazebo with another world file provided by the “*iiwa\_stack*” package, and then I saved the context to create a world file.



```
<visual name="front_visual">
  <pose>0 0 0 0 -1.57 0</pose>
  <geometry>
    <cylinder>
      <radius>0.15</radius>
      <length>0.01</length>
    </cylinder>
  </geometry>
  <material>
    <ambient>255 0 0 1</ambient>
  </material>
</visual>
<collision name="collision">
  <pose>0 0 0 0 -1.57 0</pose>
  <geometry>
    <cylinder>
      <radius>0.15</radius>
      <length>0.01</length>
    </cylinder>
  </geometry>
</collision>
</link>
</model>
</sdf>
```

- 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.

 iiwa\_world\_circular.launch

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

  <!-- Loads the 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_aruco_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>
```

## iiwa\_gazebo\_circular\_object.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="$(find iiwa_gazebo)/launch/iiwa_world_circular.launch">
  <arg name="hardware_interface" value="$(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)">

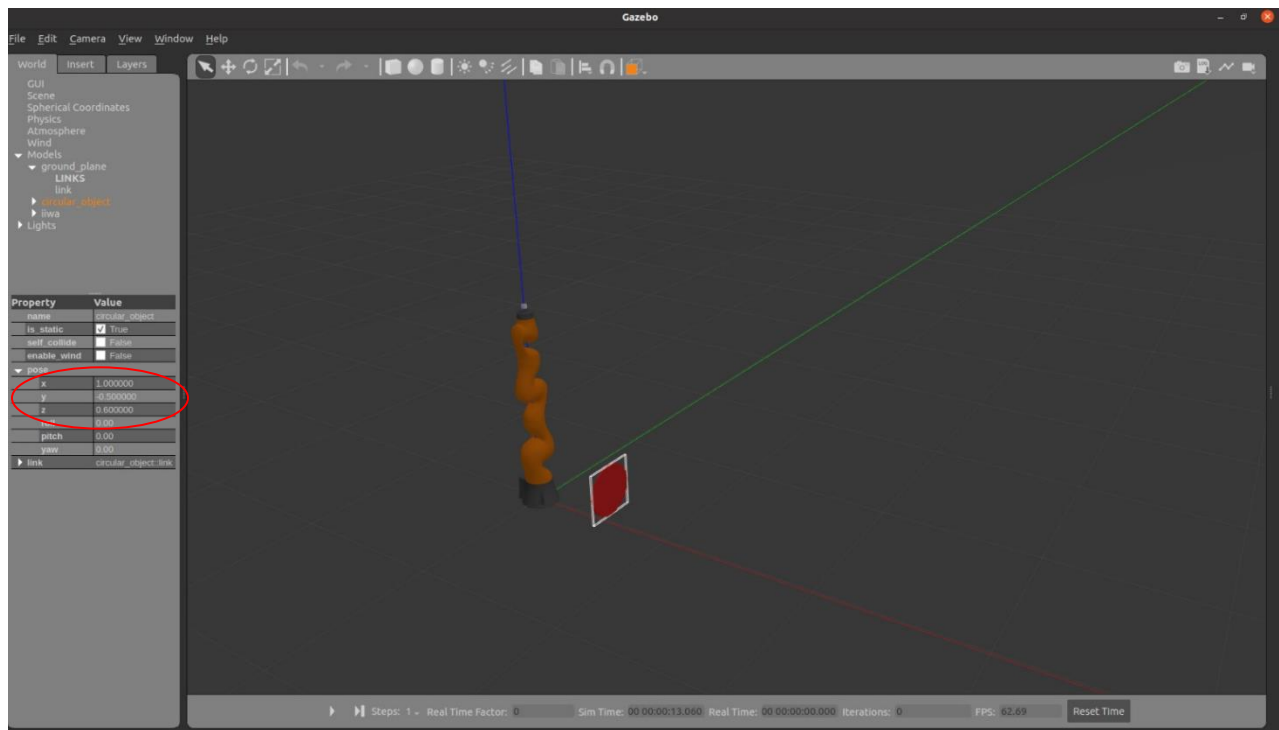
  <include file="$(find iiwa_control)/launch/iiwa_control.launch">
    <arg name="hardware_interface" value="$(arg hardware_interface)" />
    <arg name="controllers" value="joint_state_controller $(arg hardware_interface)_trajectory_controller" />
    <arg name="robot_name" value="$(arg robot_name)" />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

</group>

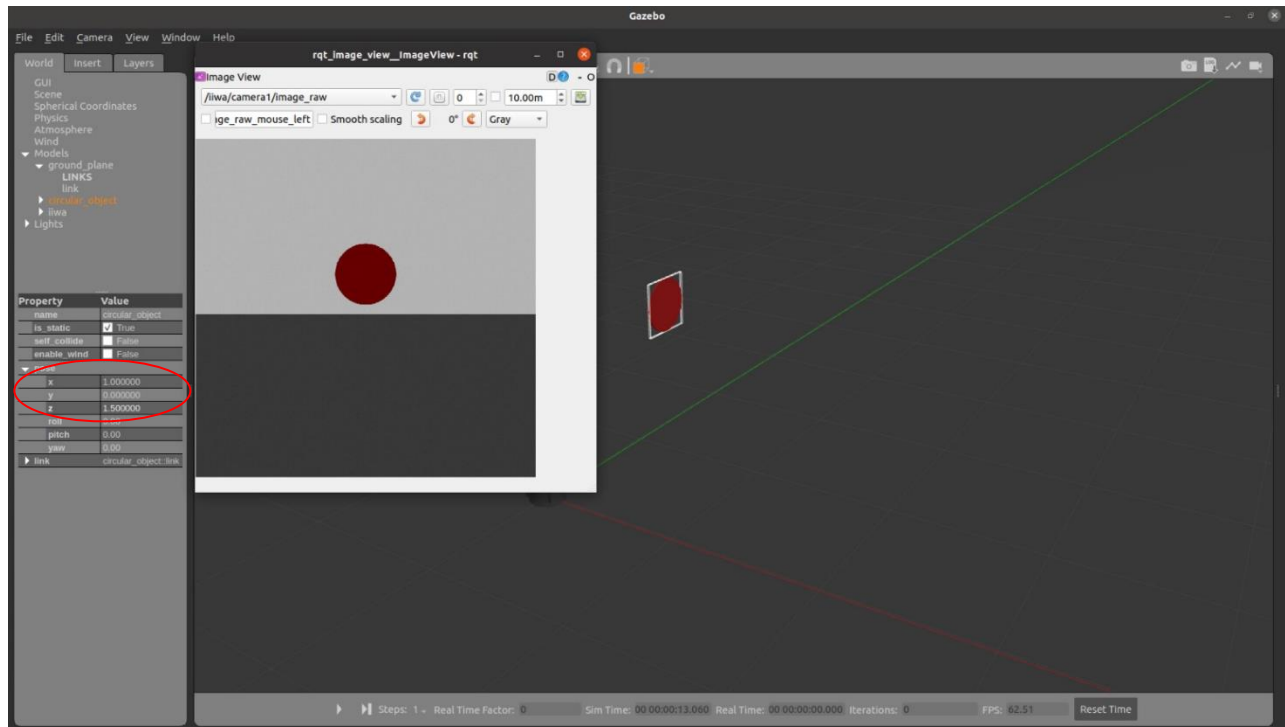
<!-- Spawn controllers - it uses an Effort Controller for each joint -->
<group ns="$(arg robot_name)" unless="$(arg trajectory)">

  <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>
```



Since that, in this configuration, the camera cannot see the circular object I changed its pose directly inside of Gazebo.



- c. Once the object is visible in the camera image, use the “*opencv\_ros/*” package to detect the circular object using “*openCV*” 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

```
// Setup SimpleBlobDetector parameters.
SimpleBlobDetector::Params params;

// Change thresholds
params.minThreshold = 0;
params.maxThreshold = 255;

//Filter by color
params.filterByColor=true;
params.blobColor=0;

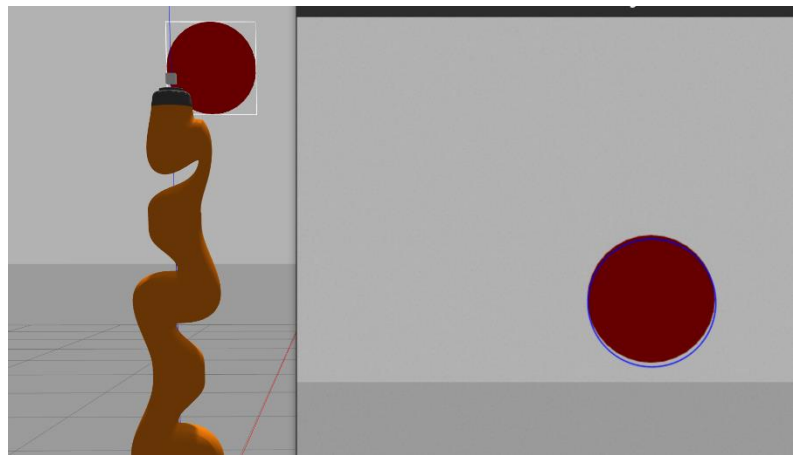
// Filter by Area.
params.filterByArea = false;
params.minArea = 0.1;
//params.maxArea=5000;

// Filter by Circularity
params.filterByCircularity = true;
params.minCircularity = 0.8;

// Filter by Convexity
params.filterByConvexity = true;
params.minConvexity = 0.9;

// Filter by Inertia
params.filterByInertia = false;
params.minInertiaRatio = 0.01;
```

To recognize the circular object, I used the “*SimpleBlobDetector*” function in “*OpenCV*”. This function allows me to set some parameters to recognize a “*blob*” type object and filter out the flickering of the camera.



As you can see, the function detects the circular object and draws a blue trace around it.

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

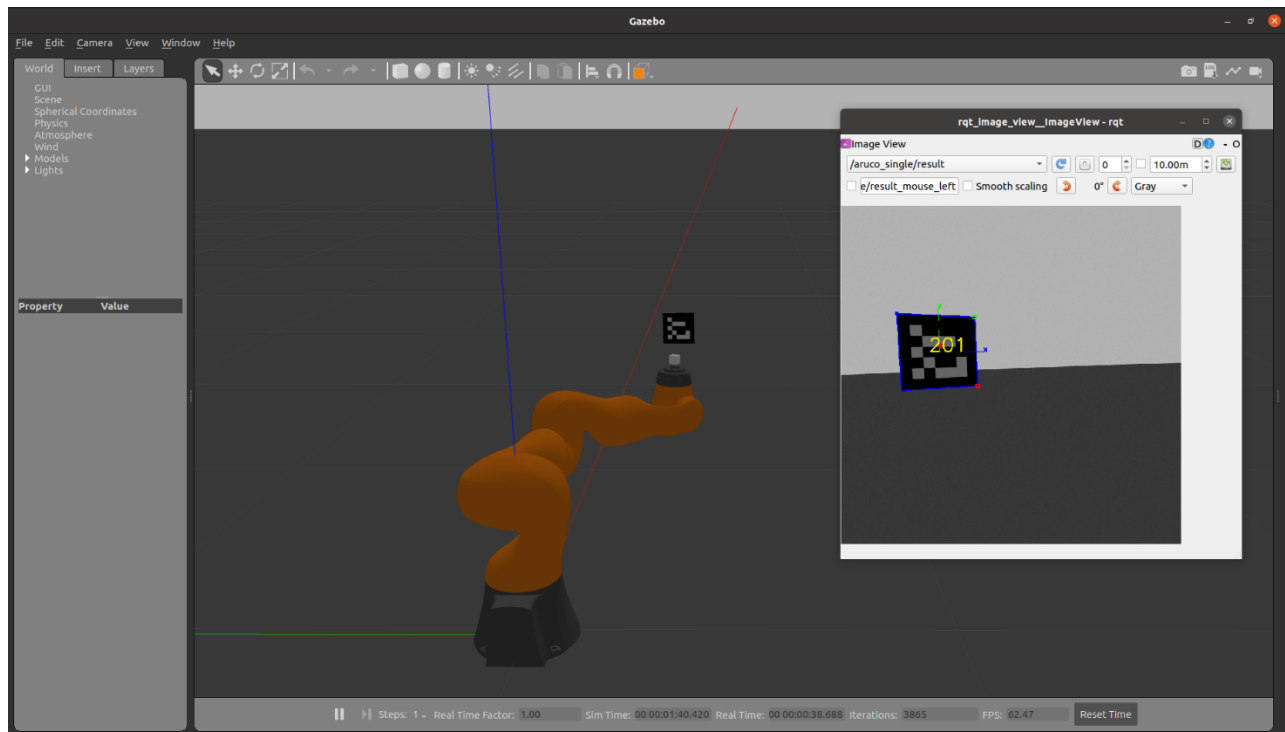
- a. The “*kdl\_robot*” package provides a “*kdl\_robot\_vision\_control*” node that implements a vision-based look-at-point control task with the simulated iiwa robot. It uses the “*VelocityJointInterface*” enabled by the “*iiwa\_gazebo\_aruco.launch*” and the “*usb\_cam\_aruco.launch*” launch files. Modify the “*kdl\_robot\_vision\_control*” node to implement a vision-based task that aligns the camera to the aruco marker with an appropriately 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.

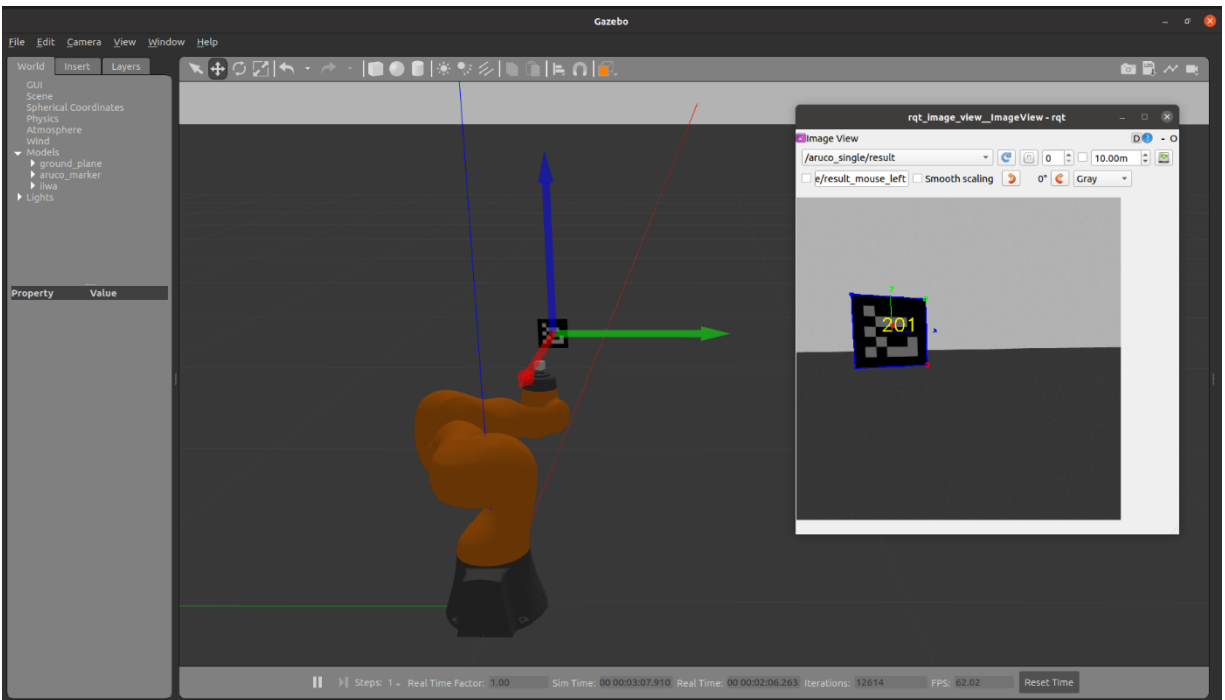
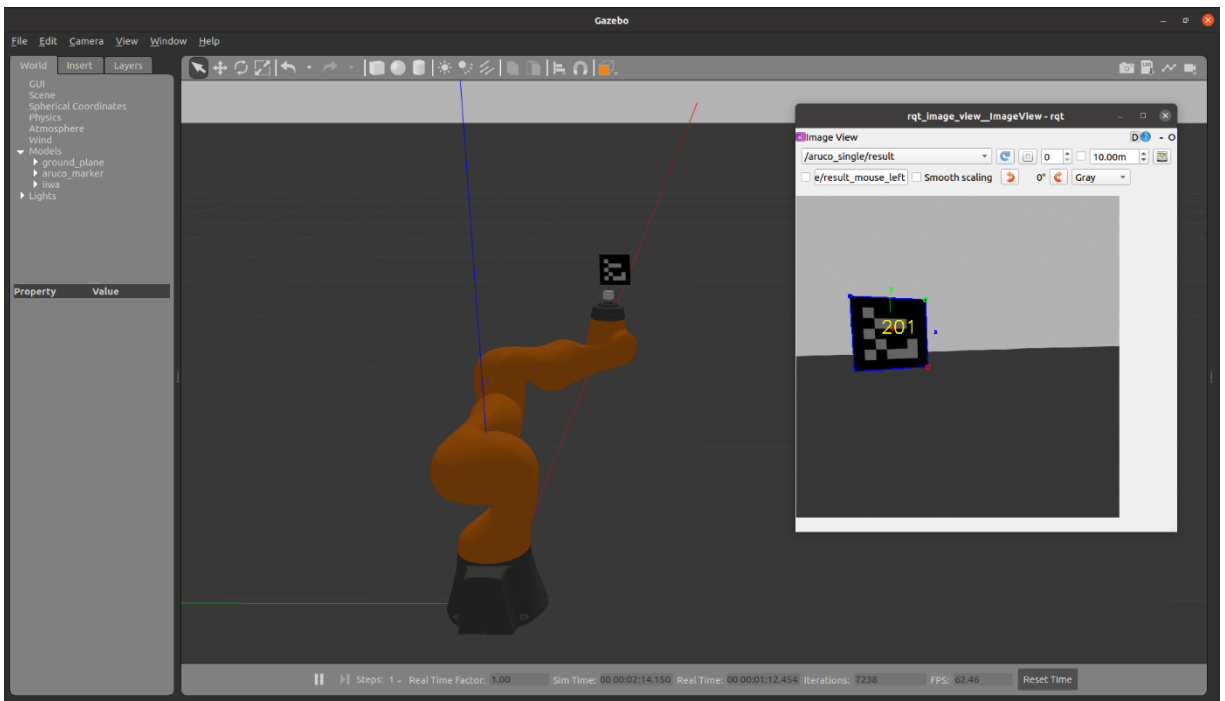
### kdl\_robot\_vision\_control.cpp

```
// From object to base frame with offset on rotation and position
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_pose[1], aruco_pose[2]));
KDL::Frame base_T_object = robot.getEEFrame() * cam_T_object;
double p_offset = -0.30; // Position offset
double R_offset = 0.314/2; // Orientation offset. Put at 0 to centre the aruco
base_T_object.p = base_T_object.p + KDL::Vector(p_offset, 0.0, 0.0);
base_T_object.M = base_T_object.M * KDL::Rotation::RotX(3.14)*KDL::Rotation::RotY(R_offset);

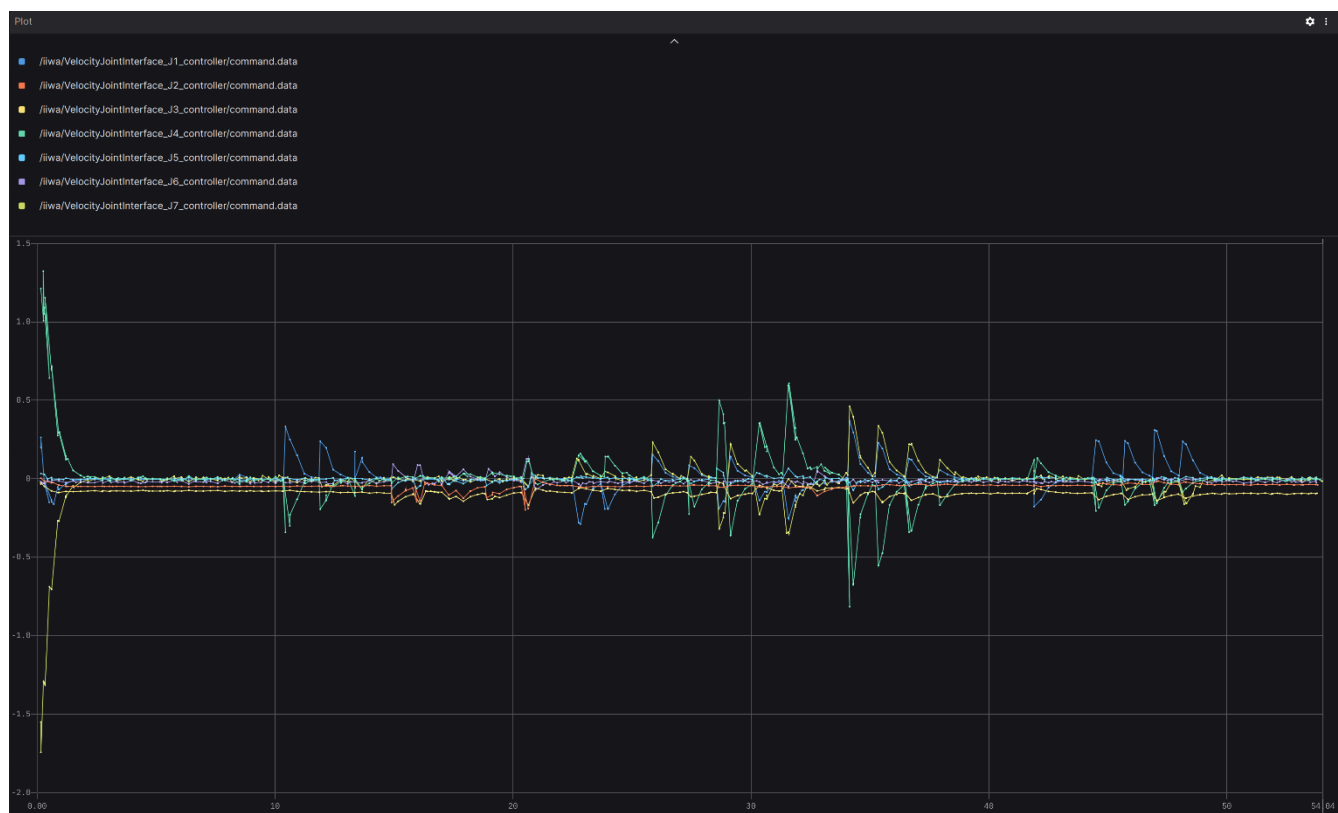
// compute error
// Eigen::Matrix<double,3,1> e_o = computeOrientationError(toEigen(base_T_object.M), toEigen(robot.getEEFrame().M));
// Eigen::Matrix<double,3,1> e_p2 = computeLinearError(toEigen(base_T_object.p), toEigen(robot.getEEFrame().p));
// Eigen::Matrix<double,6,1> x_tilde; x_tilde << e_p2, e_o[0], e_o[1], e_o[2];

// resolved velocity control law
// 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));
```









- b. An improved look-at-point algorithm can be devised by noticing that the task is belonging to  $S_2$ . Indeed, if we consider

$$s = \frac{cP_0}{\|cP_0\|} \in S_2$$

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

$$L(s) = \left[ -\frac{1}{\|cP_0\|} (I - ss^T) S(s) \right] R \in \mathbf{R}^{3 \times 6} \text{ with } R = \begin{bmatrix} R_c & 0 \\ 0 & R_c \end{bmatrix}$$

Where  $S(\cdot)$  is the skew-symmetric operator,  $R_c$  the current camera rotation matrix. Implement the following control law.

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

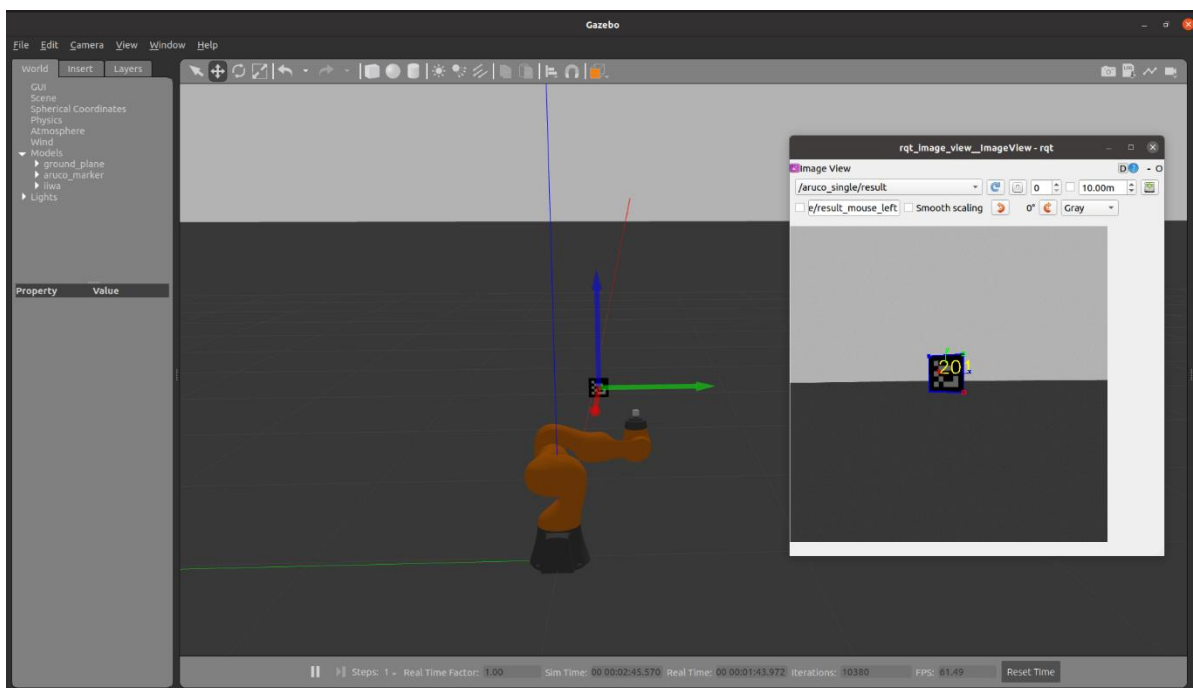
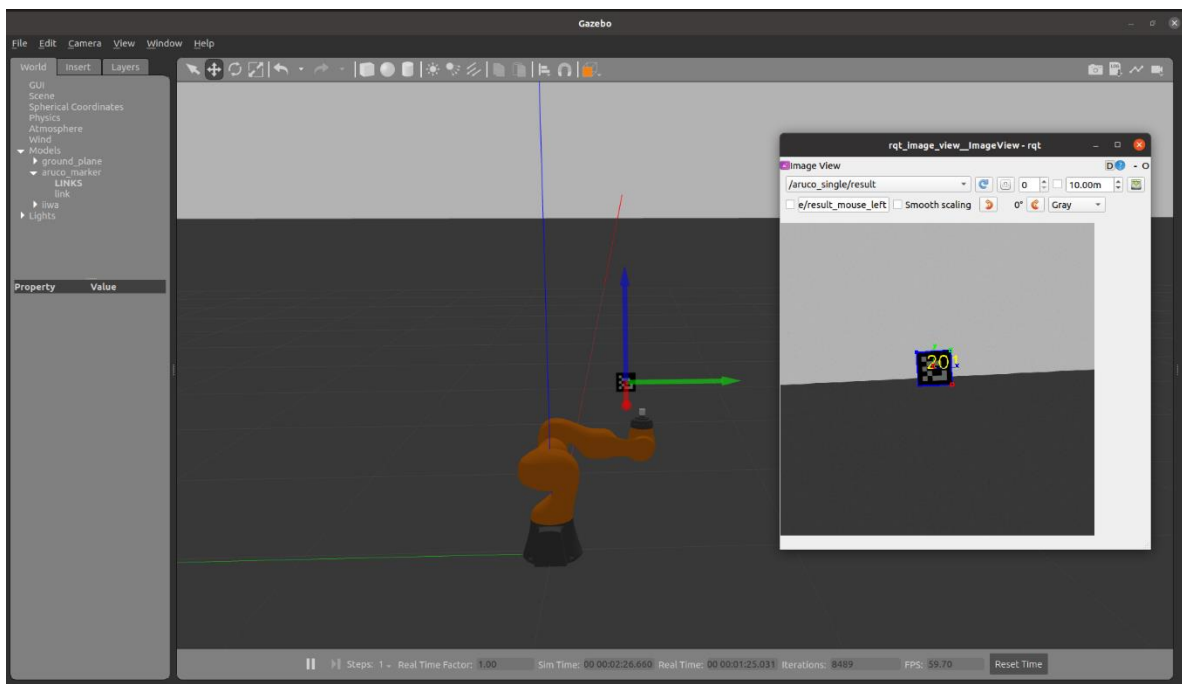
Where  $s_d$  is a desired value for  $s$ , e.g.  $s_d = [0,0,1]$ , and  $N = I - (LJ)^\dagger LJ$  being the matrix spanning the null space of the  $LJ$  matrix. Verify that for a chosen  $\dot{q}_0$  the  $s$  measure does not change by plotting joint velocities and the  $s$  components.

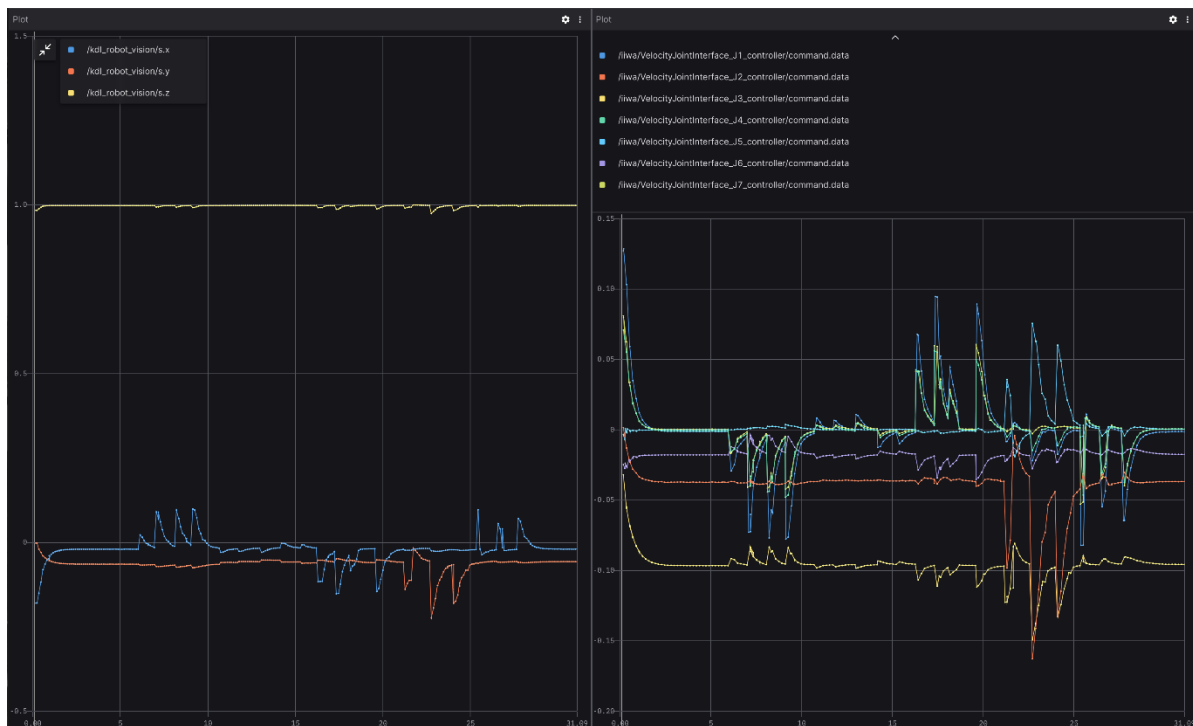
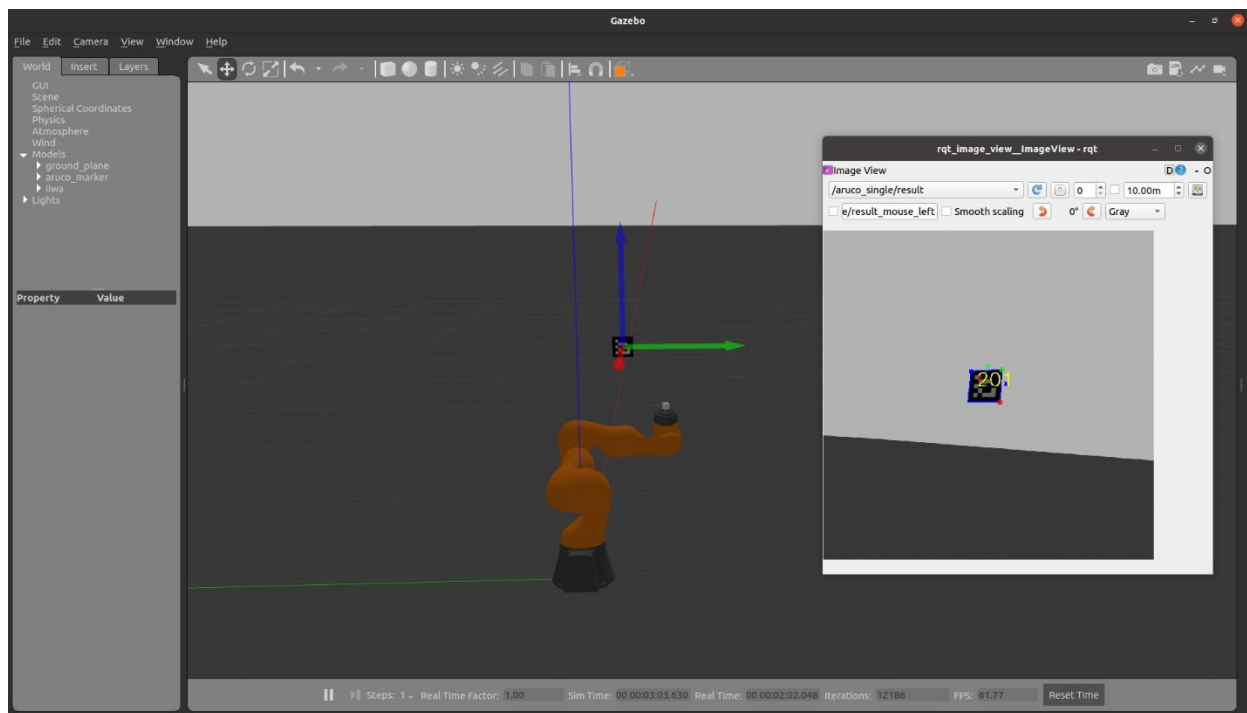
kdl\_robot\_vision\_control.cpp

```
//2.b
//calcolo matrici
Eigen::Matrix<double,3,1> c_Po = toEigen(cam_I_object.p);
Eigen::Matrix<double,3,1> s = c_Po/c_Po.norm();
Eigen::Matrix<double,3,3> Rc = 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> Rc_grande = Eigen::Matrix<double,6,6>::Zero();
Rc_grande.block(0,0,3,3) = Rc;
Rc_grande.block(3,3,3,3) = Rc;
L.block(0,0,3,3) = L_block;
L.block(0,3,3,3) = skew(s);
L=L*(Rc_grande.transpose());

//calcolo N
Eigen::MatrixX<double> LJ = L*toEigen(J_cam);
Eigen::MatrixX<double> LJ_pinv = LJ.completeOrthogonalDecomposition().pseudoInverse();
Eigen::MatrixX<double> N = Eigen::Matrix<double,7,7>::Identity()-(LJ_pinv*LJ);

//Control Law
dqddata=2*LJ_pinv*Eigen::Vector3d(0,0,1) + 2*N*(qddi - toEigen(jnt_pos));
```





With the current choice of  $\dot{q}_0$ , the  $s$  vector components stay around the desired value  $s = [0,0,1]$  with an acceptable tolerance degree.

- 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.

kdl\_robot\_test.cpp

```
// 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_pose[1], aruco_pose[2]));

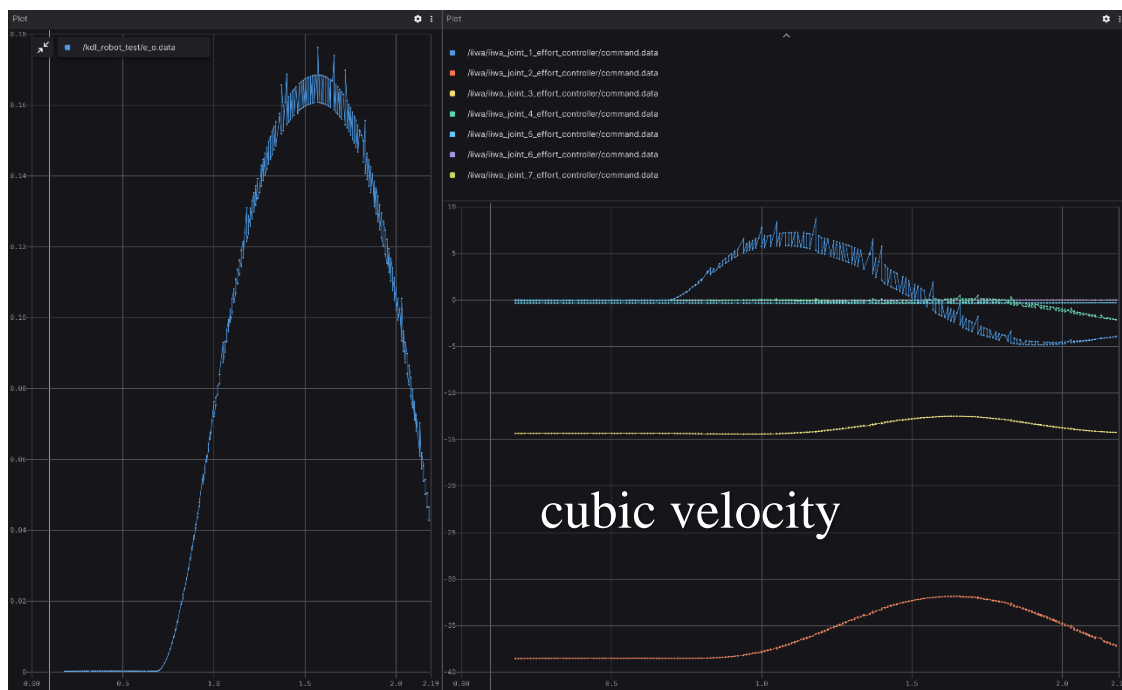
// look at point: compute rotation error from angle/axis
Eigen::Matrix<double,3,1> aruco_pos_n = toEigen(cam_T_object.p); //(aruco_pose[0],aruco_pose[1],aruco_pose[2]);
aruco_pos_n.normalize();
Eigen::Vector3d r_o = skew(Eigen::Vector3d(0,0,1))*aruco_pos_n;
double aruco_angle = std::acos(Eigen::Vector3d(0,0,1).dot(aruco_pos_n));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

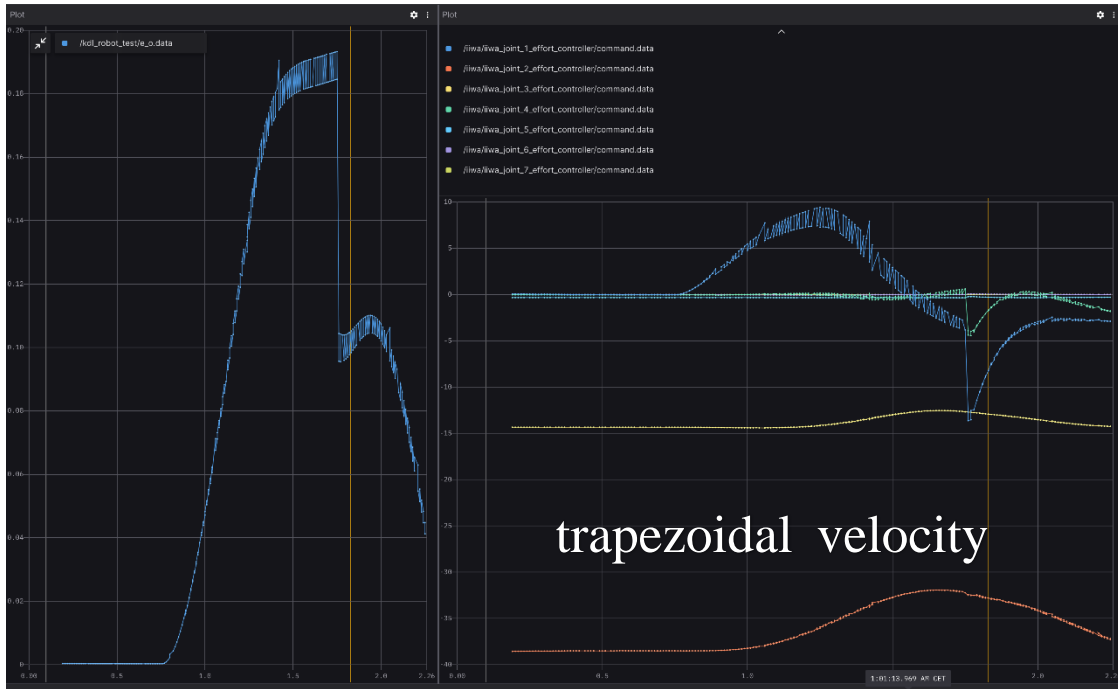
des_pose.M = robot.getEEFrame().M * Re;

// // debug
// std::cout << "jacobian: " << std::endl << robot.getEEJacobian().data << std::endl;
// std::cout << "jsim: " << std::endl << robot.getJsims() << std::endl;
// std::cout << "c: " << std::endl << robot.getCoriolis().transpose() << std::endl;
// std::cout << "g: " << std::endl << robot.getGravity().transpose() << std::endl;
// std::cout << "qd: " << std::endl << qd.data.transpose() << std::endl;
// std::cout << "q: " << std::endl << robot.getJntValues().transpose() << std::endl;
// std::cout << "tau: " << std::endl << tau.transpose() << std::endl;
// std::cout << "desired_pose: " << std::endl << des_pose << std::endl;
// std::cout << "current_pose: " << std::endl << robot.getEEFrame() << std::endl;

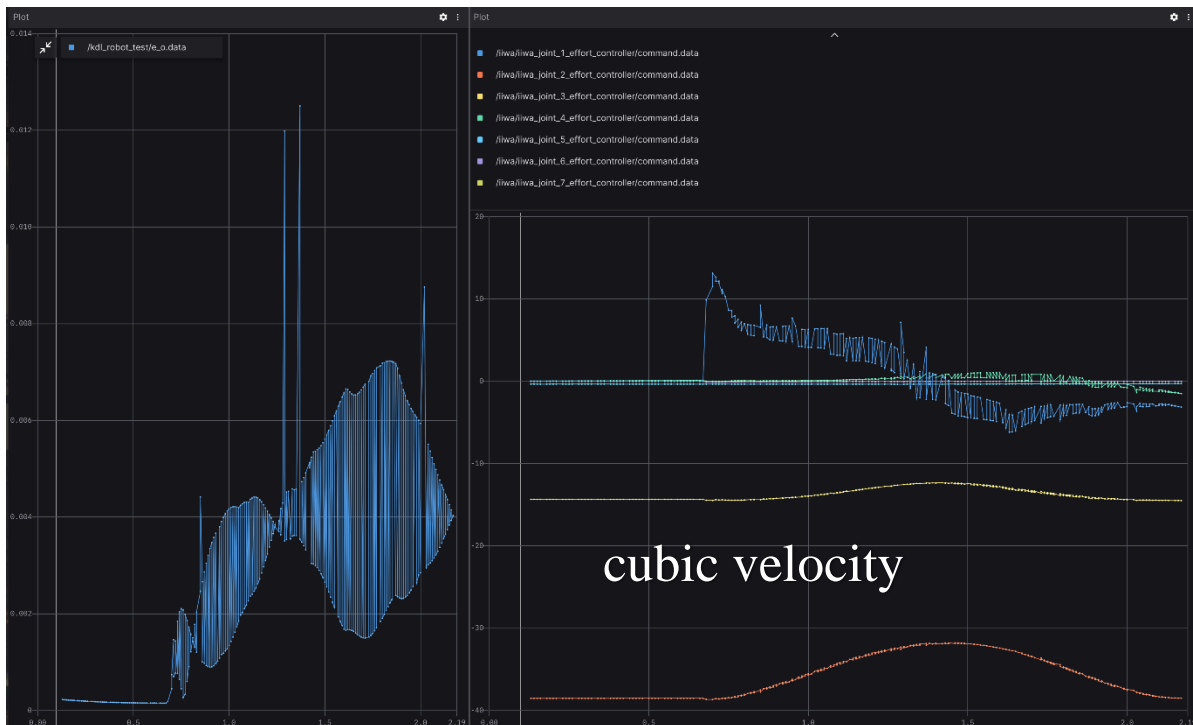
// inverse kinematics
qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3], jnt_pos[4], jnt_pos[5], jnt_pos[6];
qd = robot.getInvKin(qd, des_pose*robot.getFlangeEE().Inverse());
```

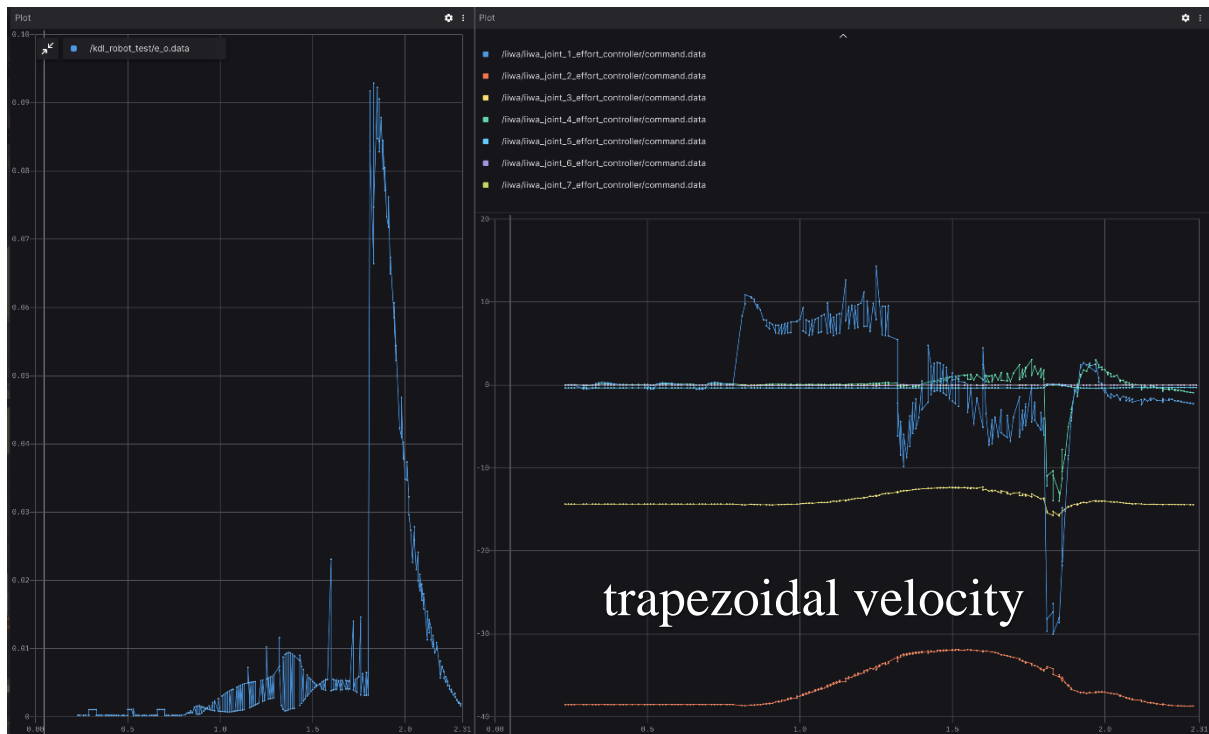
Plotting the results using the joint space inverse dynamics controller on the linear trajectories we get



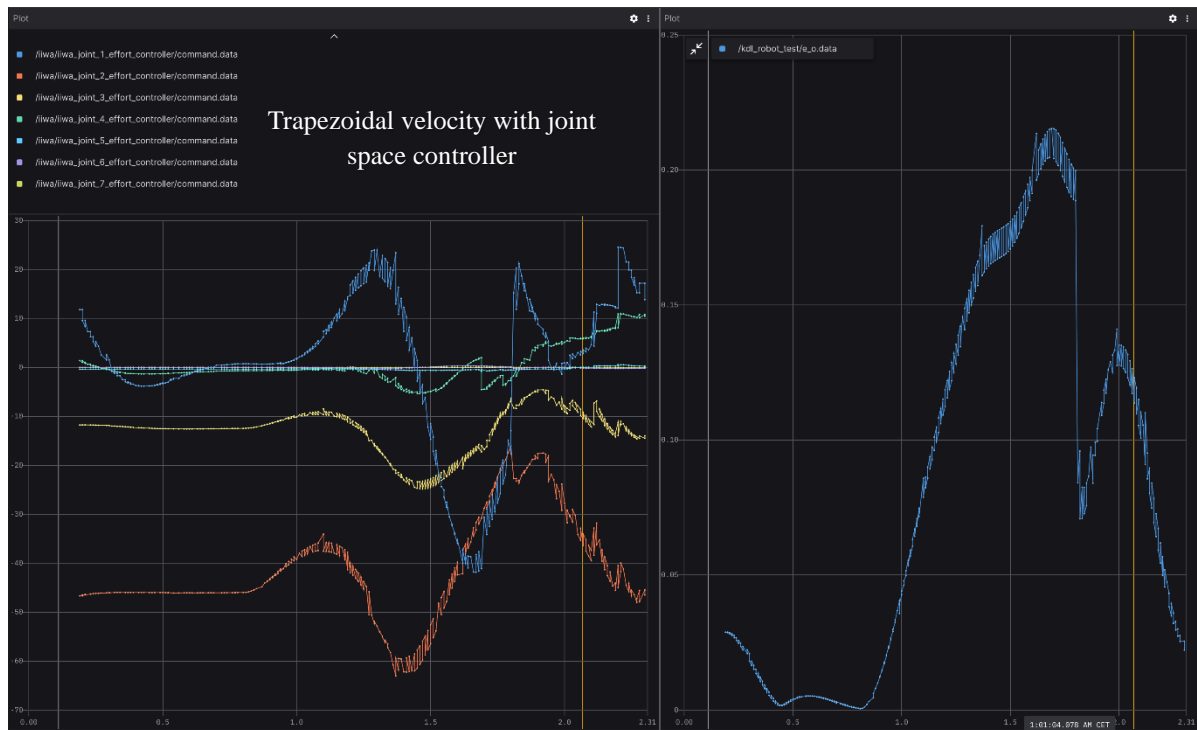


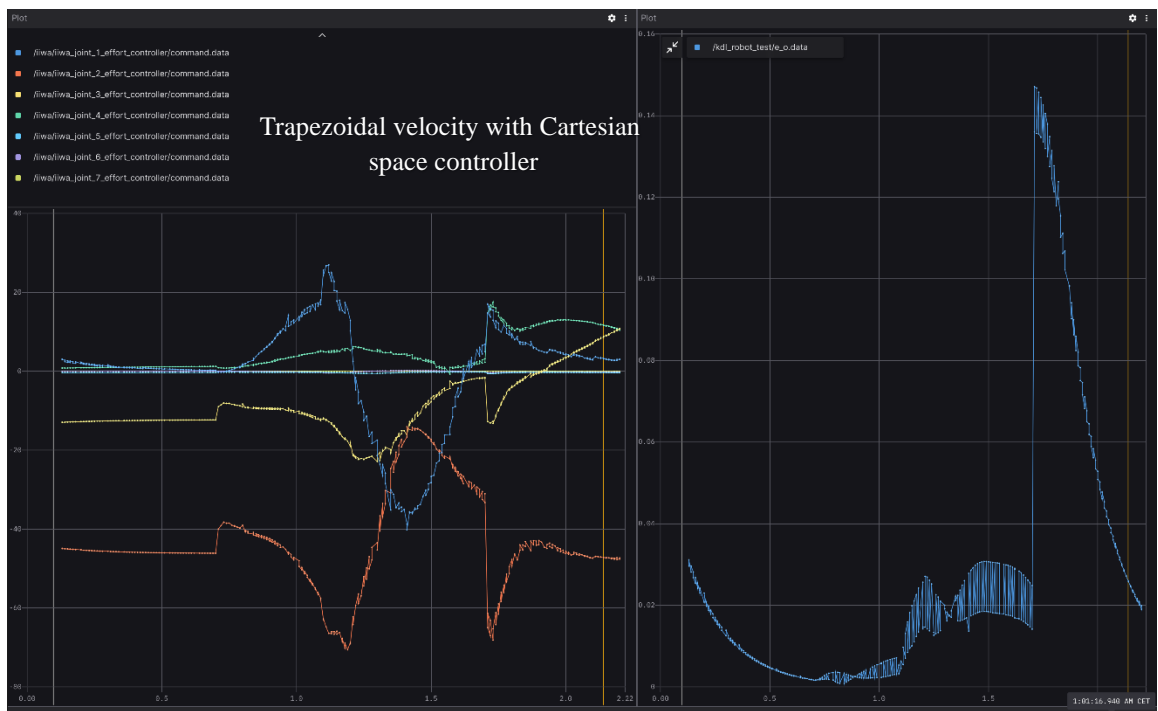
Instead, plotting the results using the Cartesian space inverse dynamics controller on the linear trajectories we get





Here I also present some plots of circular trajectories







Repository Github : <https://github.com/matteolangella/RL-23-24-HW3.git>