UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria dell'Automazione e Robotica

Report

HOMEWORK 3

Academic Year 2024/25

Professor
Mario Selvaggio

Candidate
Daniele Fontana **P38000297**

Abstract

This report will show how I succeeded in completing each task of the Homework 3.

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 first task asks to create a blue colored spherical object model, insert it in a gazebo world and detect it using OpenCV functions.
The second one requires to spawn an aruco tag in a Gazebo world, then implement a vision-based controller for the simulated iiwa robot.
Firstly, the controller has to compute velocity commands in order to accomplish a positioning task and a look-at-point task. Then, we had to do the same thing as before, but with torques commands and to merge the look-at-point task with a linear trajectory.

Contents

Chapter 1

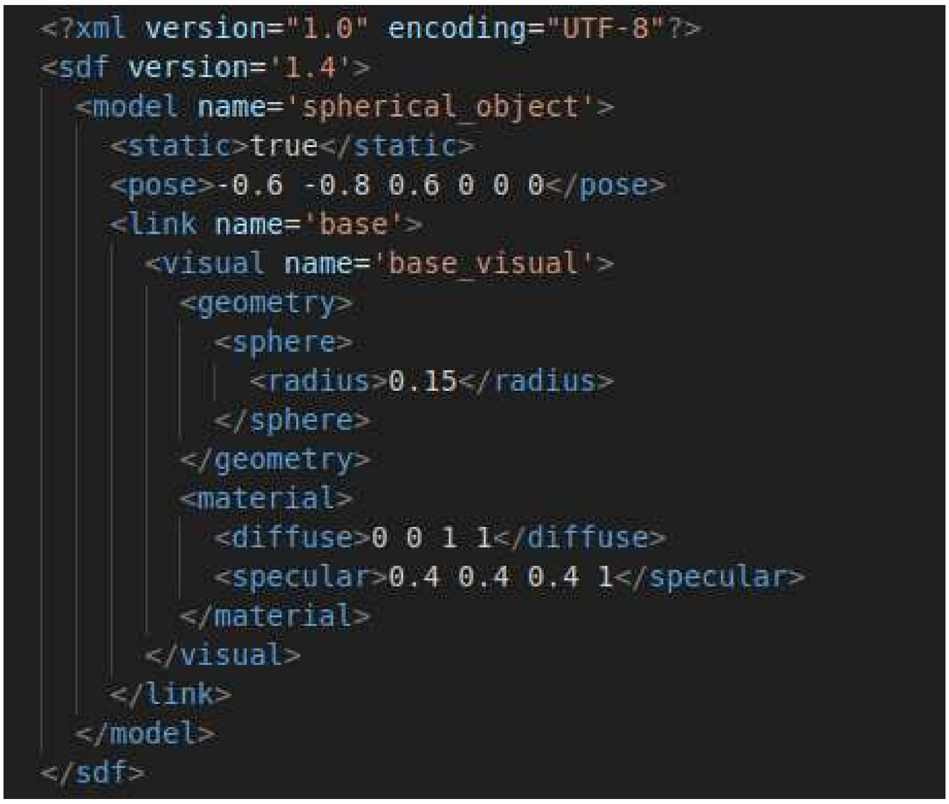
Circular Object Detection in Gazebo

Regarding the first task of the homework, first of all I cloned the repositories https://github.com/RoboticsLab2024/ros2_kdl_package, https://github.com/RoboticsLab2024/ros2_iiwa and https://github.com/RoboticsLab2024/ros2_vision since I used these packages as starting point.
The first step was to modify the *gazebo/model* folder, creating a new folder inside named spherical_object, that contains the representation of a blue sphere with a 15 cm radius. The center of the sphere has been positioned in

$$x = -0.6 \quad y = -0.8 \quad z = 0.6$$

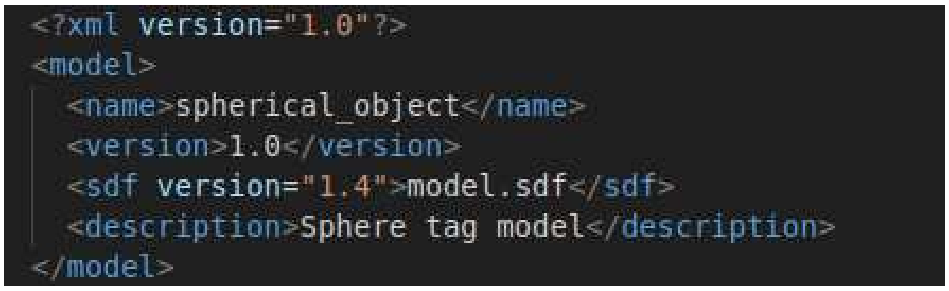
to make it visible by the camera, that we will attach on the end effector later on.

The codes used to create the object are the following:



```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.4'>
  <model name='spherical_object'>
    <static>true</static>
    <pose>-0.6 -0.8 0.6 0 0 0</pose>
    <link name='base'>
      <visual name='base_visual'>
        <geometry>
          <sphere>
            <radius>0.15</radius>
          </sphere>
        </geometry>
        <material>
          <diffuse>0 0 1 1</diffuse>
          <specular>0.4 0.4 0.4 1</specular>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

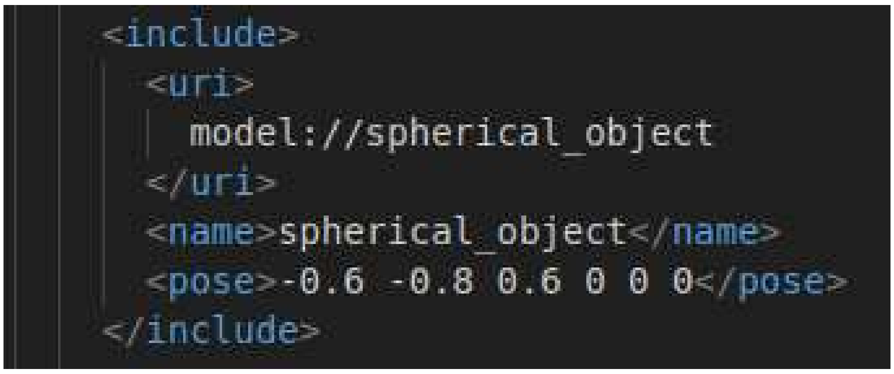
Figure 1.1: Code snippet of *model.sdf*



```
<?xml version="1.0"?>
<model>
  <name>spherical_object</name>
  <version>1.0</version>
  <sdf version="1.4">model.sdf</sdf>
  <description>Sphere tag model</description>
</model>
```

Figure 1.2: Code snippet of *model.config*

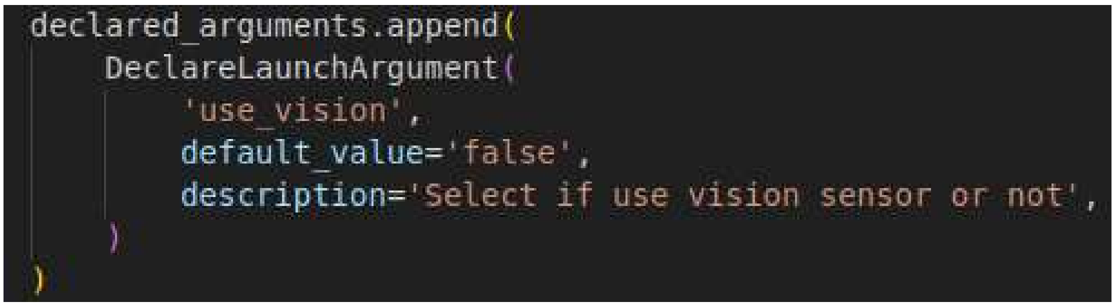
Then I spawned the spherical_object model into our world file by writing these lines in *empty.world* :



```
<include>
  <uri>
    model://spherical_object
  </uri>
  <name>spherical_object</name>
  <pose>-0.6 -0.8 0.6 0 0 0</pose>
</include>
```

Figure 1.3: Code snippet of *empty.world*

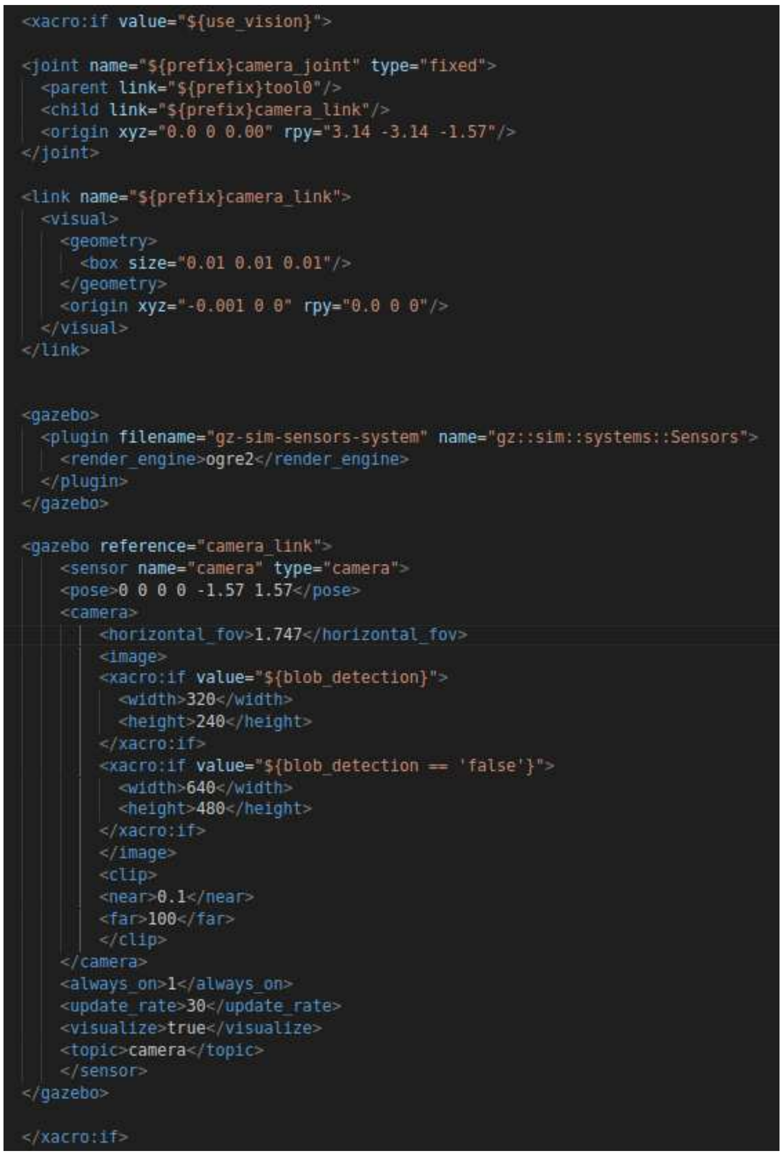
After that, I proceeded in equipping the end effector of the robot with the camera. First of all I set the *use_vision argument in the iiwa.launch.py* to false. In order to load the robot with the camera into the new world the argument *use_vision:="true"* has to be specified when lauching it. This was made possible by adding `<xacro:if value="${use_vision}">` statement in *iiwa.urdf.xacro*



```
declared_arguments.append(
    DeclareLaunchArgument(
        'use_vision',
        default_value='false',
        description='Select if use vision sensor or not',
    )
)
```

Figure 1.4: Modified *iiwa.launch.py*

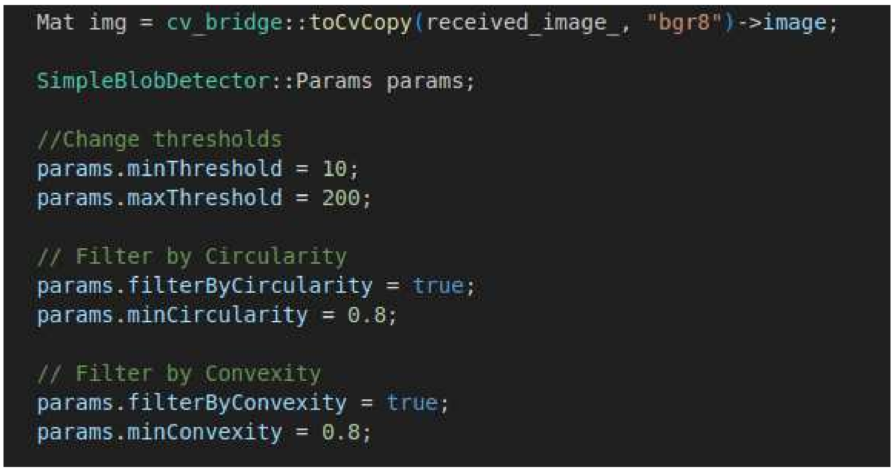
Then I modified the xacro file in order to spawn the robot with the camera attached to the end effector:



```
<xacro:if value="${use_vision}">

<joint name="${prefix}camera_joint" type="fixed">
  <parent link="${prefix}tool0"/>
  <child link="${prefix}camera_link"/>
  <origin xyz="0.0 0 0.00" rpy="3.14 -3.14 -1.57"/>
</joint>

<link name="${prefix}camera_link">
  <visual>
    <geometry>
      <box size="0.01 0.01 0.01"/>
    </geometry>
    <origin xyz="-0.001 0 0" rpy="0.0 0 0"/>
  </visual>
</link>


<gazebo>
  <plugin filename="gz-sim-sensors-system" name="gz::sim::systems::Sensors">
    <render_engine>ogre2</render_engine>
  </plugin>
</gazebo>

<gazebo reference="camera_link">
    <sensor name="camera" type="camera">
    <pose>0 0 0 0 -1.57 1.57</pose>
    <camera>
        <horizontal_fov>1.747</horizontal_fov>
        <image>
        <xacro:if value="${blob_detection}">
          <width>320</width>
          <height>240</height>
        </xacro:if>
        <xacro:if value="${blob_detection == 'false'}">
          <width>640</width>
          <height>480</height>
        </xacro:if>
        </image>
        <clip>
        <near>0.1</near>
        <far>100</far>
        </clip>
    </camera>
    <always_on>1</always_on>
    <update_rate>30</update_rate>
    <visualize>true</visualize>
    <topic>camera</topic>
    </sensor>
</gazebo>

</xacro:if>
```

Figure 1.5: Code snippet of *iiwa.urdf.xacro*

Once we have spawned the camera and the spherical object in our world, I used the *ros2_opencv_node.cpp* , contained in the *ros2_opencv* package, in order to subscribe to the topic */videocamera* . So, we are now receiving the simulated image from the camera. Then, I have used the OpenCV functions to detect our spherical object, using the following filters, based on color, convexity and circularity:



```
Mat img = cv_bridge::toCvCopy(received_image_, "bgr8")->image;

SimpleBlobDetector::Params params;

//Change thresholds
params.minThreshold = 10;
params.maxThreshold = 200;

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

// Filter by Convexity
params.filterByConvexity = true;
params.minConvexity = 0.8;
```

Figure 1.6: Code snippet of *ros2_ opencv_ node.cpp*

Then I created the topic /processed_image where the results of the detection will be published. All the objects detected by the node will appear with a pink circle around them. In our case, these are the detection results:

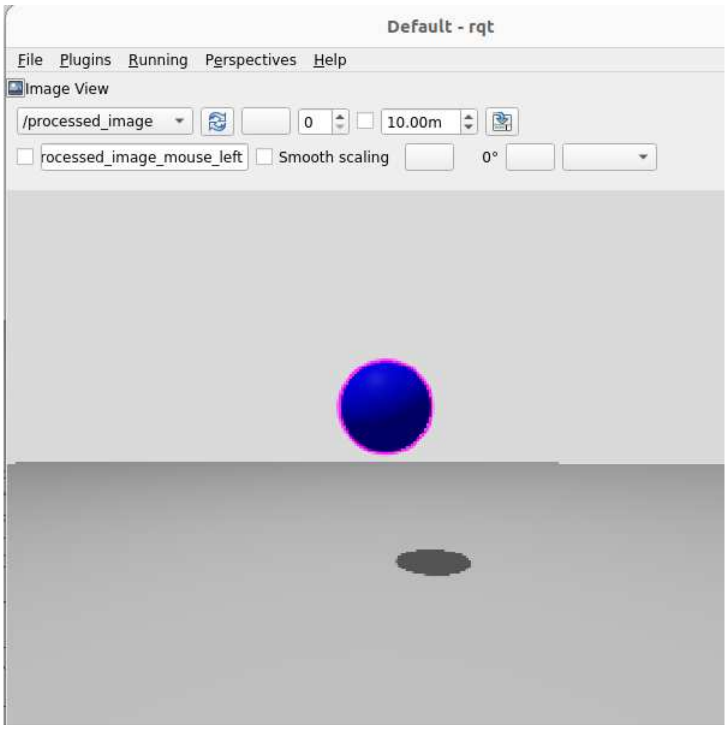

Default - rqt
File Plugins Running Perspectives Help
Image View
/processed_image
0
10.00m
rocessed_image_mouse_left
Smooth scaling
0°

Figure 1.7: Image detection results

Chapter 2

Vision-based controllers in ROS2

First of all I created a *ros2_kdl_vision_control.cpp* node in *ros2_kdl_package.*

2.1 Velocity controller for vision based task

2.1.1 Positioning

I started by implementing the **positioning** task: we wanted our manipulator's end-effector to reach a certain position and orientation with respect to the aruco marker.
We used the *simple_single.cpp* node, contained in *aruco_ros* package,

to detect the marker in the simulation and to publish the results in the /aruco_single/result topic. The marker id has been set to 201 and its size to 0.1.

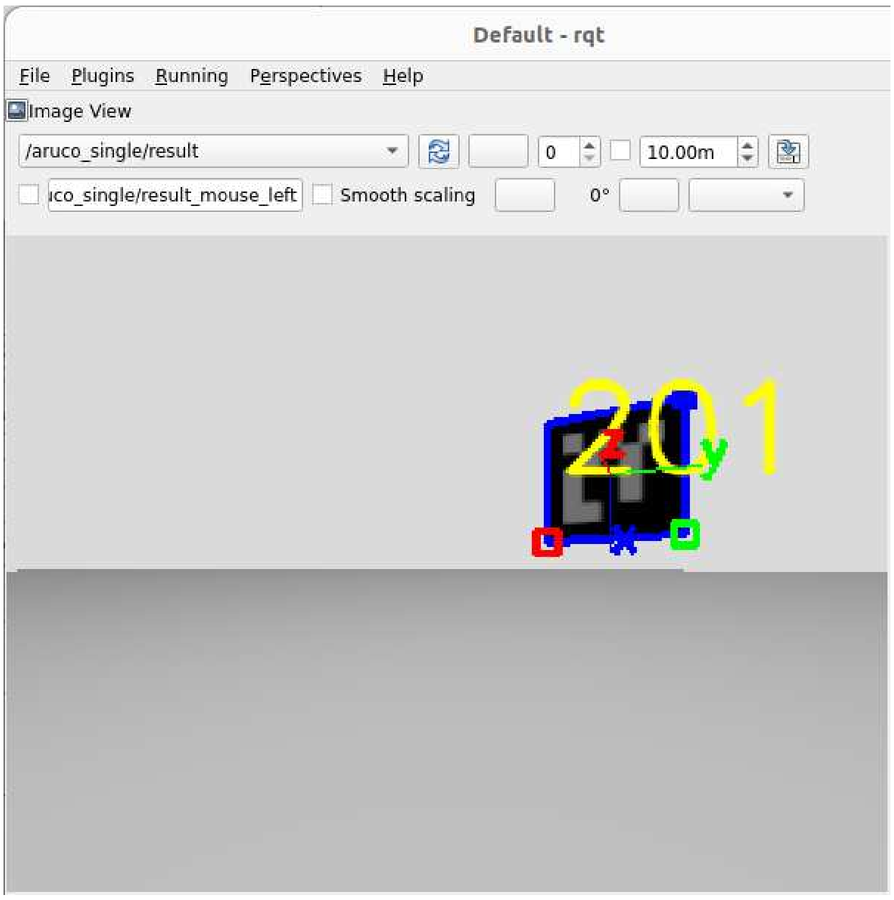
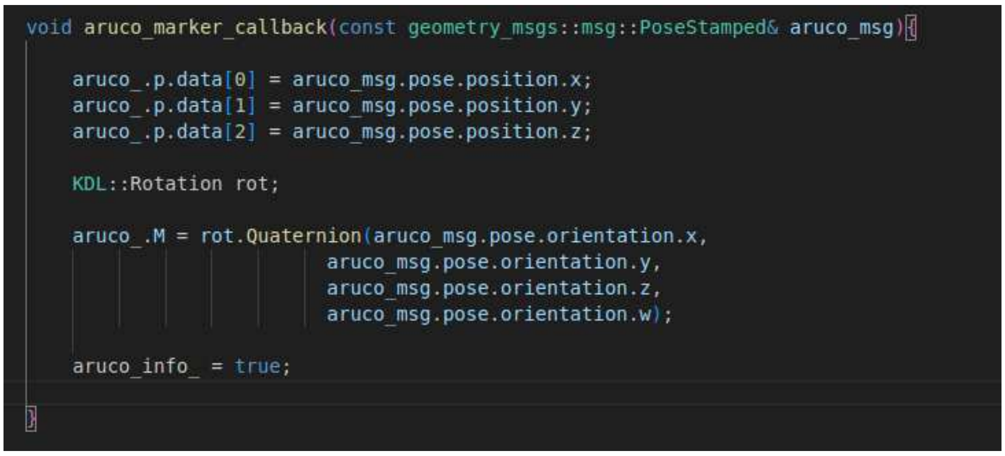


Figure 2.1: Aruco detection result

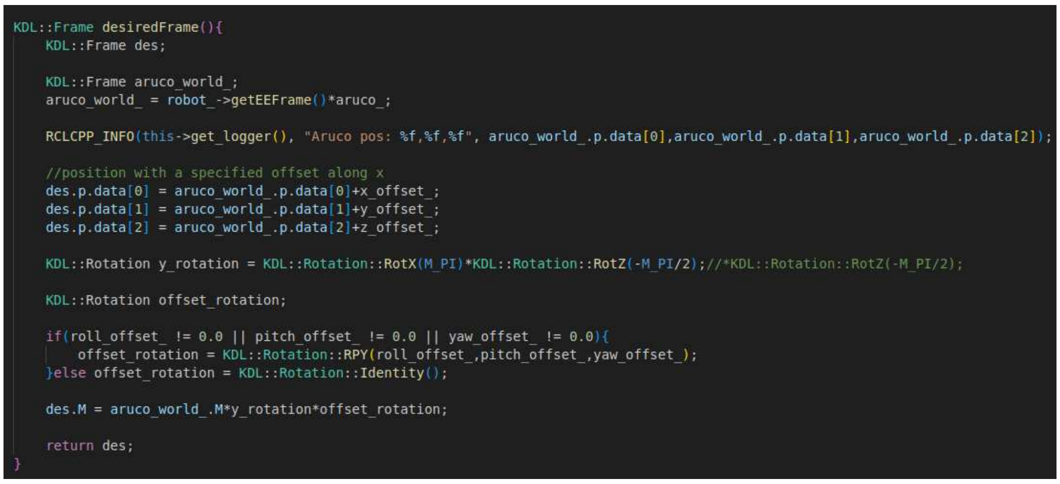
Starting from the results of the detection, we were able to retrieve position and orientation of the aruco marker.



```cpp
void aruco_marker_callback(const geometry_msgs::msg::PoseStamped& aruco_msg){

    aruco_.p.data[0] = aruco_msg.pose.position.x;
    aruco_.p.data[1] = aruco_msg.pose.position.y;
    aruco_.p.data[2] = aruco_msg.pose.position.z;

    KDL::Rotation rot;

    aruco_.M = rot.Quaternion(aruco_msg.pose.orientation.x,
                              aruco_msg.pose.orientation.y,
                              aruco_msg.pose.orientation.z,
                              aruco_msg.pose.orientation.w);

    aruco_info_ = true;

}
```

Figure 2.2: Code snippet of obtaining aruco marker frame

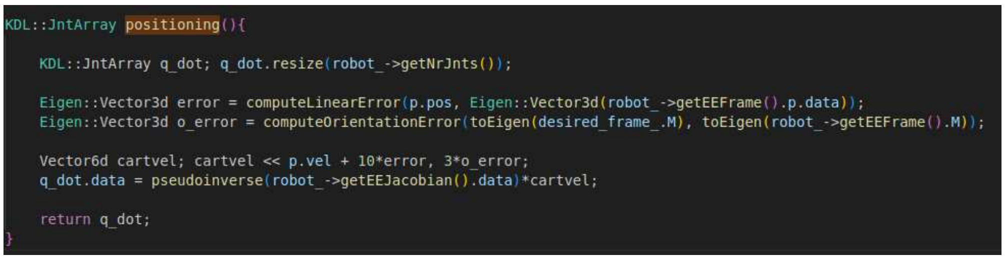
The aruco frame is computed with respect to the end-effector frame (i.e. camera frame), so we transformed it making it referred to the world frame. Then, I computed a desired frame for our manipulator, starting from the aruco one with settable offsets on orientation and position. In particular, the default are $x\_off{=}0.3$, $y\_off{=}0.1$, $z\_off{=}0.0$, $roll\_off{=}0.0$, $pitch\_off{=}0.0$, $yaw\_off{=}0.0$.



```cpp
KDL::Frame desiredFrame(){
    KDL::Frame des;

    KDL::Frame aruco_world_;
    aruco_world_ = robot_->getEEFrame()*aruco_;

    RCLCPP_INFO(this->get_logger(), "Aruco pos: %f,%f,%f", aruco_world_.p.data[0],aruco_world_.p.data[1],aruco_world_.p.data[2]);

    //position with a specified offset along x
    des.p.data[0] = aruco_world_.p.data[0]+x_offset_;
    des.p.data[1] = aruco_world_.p.data[1]+y_offset_;
    des.p.data[2] = aruco_world_.p.data[2]+z_offset_;

    KDL::Rotation y_rotation = KDL::Rotation::RotX(M_PI)*KDL::Rotation::RotZ(-M_PI/2);//*KDL::Rotation::RotZ(-M_PI/2);

    KDL::Rotation offset_rotation;

    if(roll_offset_ != 0.0 || pitch_offset_ != 0.0 || yaw_offset_ != 0.0){
        offset_rotation = KDL::Rotation::RPY(roll_offset_,pitch_offset_,yaw_offset_);
    }else offset_rotation = KDL::Rotation::Identity();

    des.M = aruco_world_.M*y_rotation*offset_rotation;

    return des;
}
```

Figure 2.3: Code snippet of desired frame computation

The desired frame is then used to compute the position and orientation error, which is in turn applied to calculate the desired joint velocities.



```
KDL::JntArray positioning(){

    KDL::JntArray q_dot; q_dot.resize(robot_->getNrJnts());

    Eigen::Vector3d error = computeLinearError(p.pos, Eigen::Vector3d(robot_->getEEFrame().p.data));
    Eigen::Vector3d o_error = computeOrientationError(toEigen(desired_frame_.M), toEigen(robot_->getEEFrame().M));

    Vector6d cartvel; cartvel << p.vel + 10*error, 3*o_error;
    q_dot.data = pseudoinverse(robot_->getEEJacobian().data)*cartvel;

    return q_dot;
}
```

Figure 2.4: Code snippet of positioning control

In our case, with the previous mentioned conditions, the result is the following:

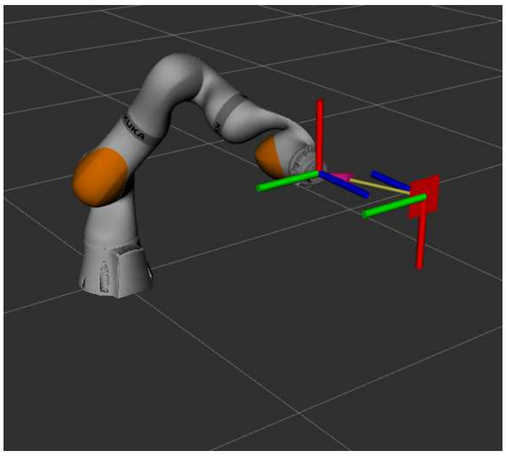


Figure 2.5: Positioning result in Rviz

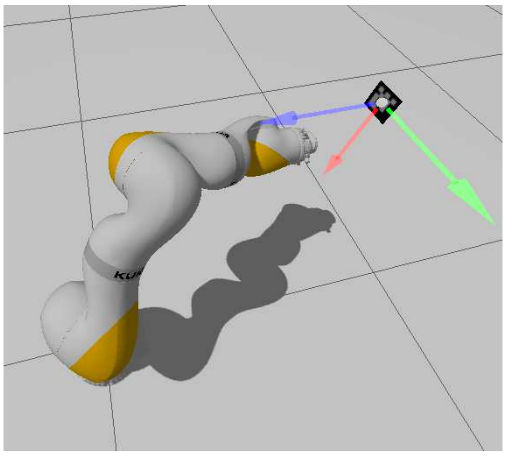


Figure 2.6: Positioning result in Gazebo

2.1.2 Look-at-point

For the **look-at-point** task, we needed to show the tracking capability of our robot of following the aruco marker in the environment. So I implemented the given control law to obtain proper joint velocities.

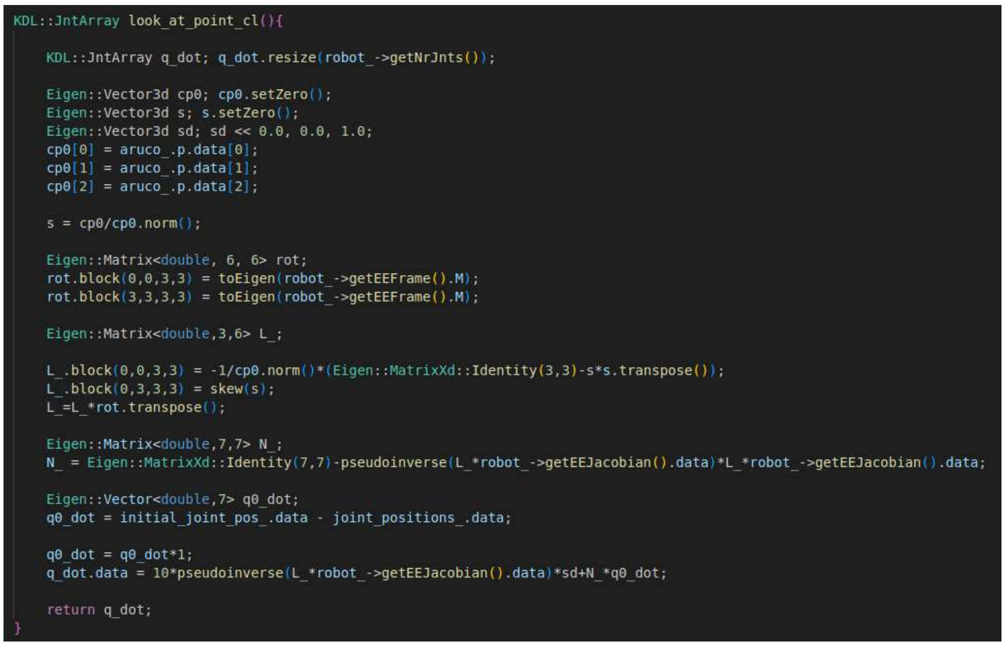
$$\dot{q} = k(LJ_c)^{\dagger} s_d + N\dot{q}_0$$

where

$$s = \frac{P_o^c}{\|P_o^c\|}$$

and

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



```
KDL::JntArray look_at_point_cl(){

    KDL::JntArray q_dot; q_dot.resize(robot_->getNrJnts());

    Eigen::Vector3d cp0; cp0.setZero();
    Eigen::Vector3d s; s.setZero();
    Eigen::Vector3d sd; sd << 0.0, 0.0, 1.0;
    cp0[0] = aruco_.p.data[0];
    cp0[1] = aruco_.p.data[1];
    cp0[2] = aruco_.p.data[2];

    s = cp0/cp0.norm();

    Eigen::Matrix<double, 6, 6> rot;
    rot.block(0,0,3,3) = toEigen(robot_->getEEFrame().M);
    rot.block(3,3,3,3) = toEigen(robot_->getEEFrame().M);

    Eigen::Matrix<double,3,6> L_;

    L_.block(0,0,3,3) = -1/cp0.norm()*(Eigen::MatrixXd::Identity(3,3)-s*s.transpose());
    L_.block(0,3,3,3) = skew(s);
    L_=L_*rot.transpose();

    Eigen::Matrix<double,7,7> N_;
    N_ = Eigen::MatrixXd::Identity(7,7)-pseudoinverse(L_*robot_->getEEJacobian().data)*L_*robot_->getEEJacobian().data;

    Eigen::Vector<double,7> q0_dot;
    q0_dot = initial_joint_pos_.data - joint_positions_.data;

    q0_dot = q0_dot*1;
    q_dot.data = 10*pseudoinverse(L_*robot_->getEEJacobian().data)*sd+N_*q0_dot;

    return q_dot;
}
```

Figure 2.7: Code snippet of look-at-point control

I carried out a look-at-point control test by moving the aruco marker in Gazebo. At first the robot aligned with the center of the marker, which is originally positioned at coordinates x=*0.0*, y=-*0.4*, z=*0.5*, then I set *y=-0.3* , waited for the robot to align, and finally I set x=-*0.3*.

These are the results of the plotted velocities commands with the look-at-point control law and the above specified conditions:

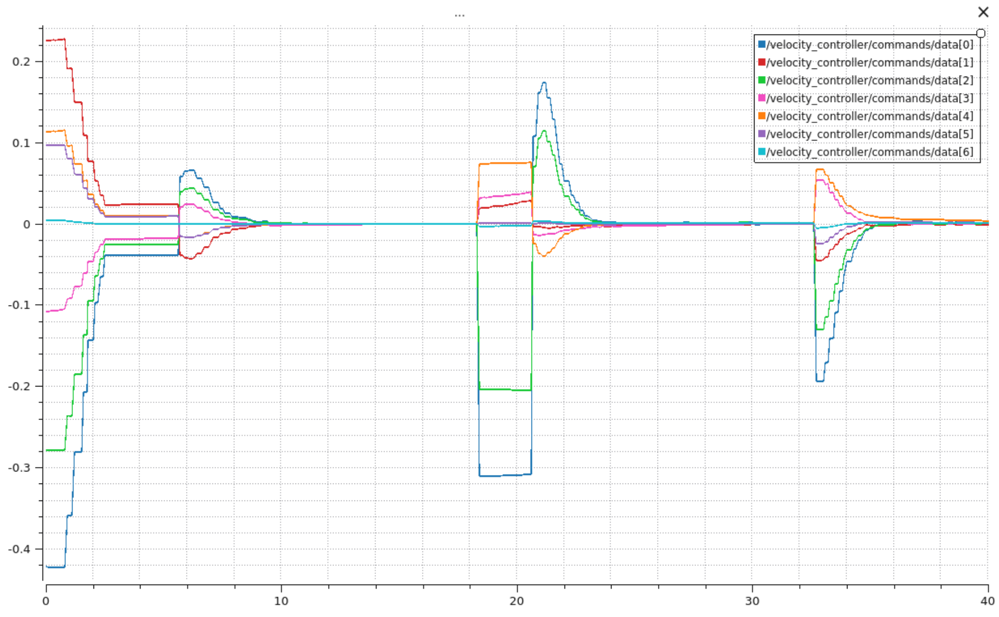


Figure 2.8: Plotted look-at-point velocities

The related video are in the repository.

2.2 Torque controller for vision based task

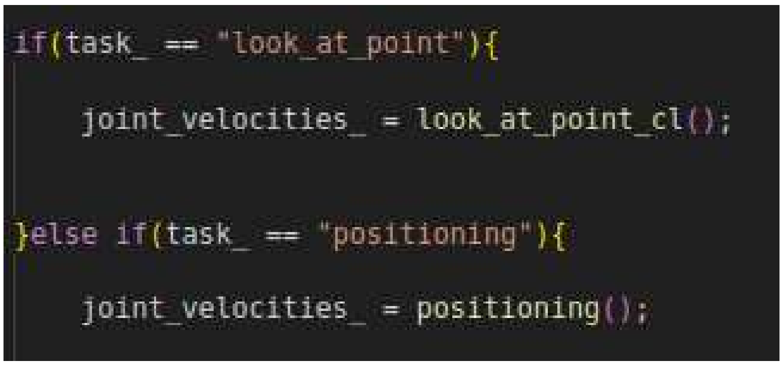
For what concerns the second part of this chapter, I implemented the look-at-point and positioning tasks using the joint space and the Cartesian space inverse dynamics controllers.

2.2.1 Joint space controller - Positioning

For the positioning task with the joint space controller, I simply computed the joint velocities with the function we saw in *Figure 2.4* , and then obtained the joint positions and accelerations by integrating and differentiating the velocities. Finally, I gave these velocities, positions and accelerations as input to the controller developed in the previous homework.

2.2.2 Joint space controller - Look-at-point

Moreover, I implemented the look-at-point task using the same function we saw above in *Figure 2.7* , and then proceeded in a similar way of the positioning task.

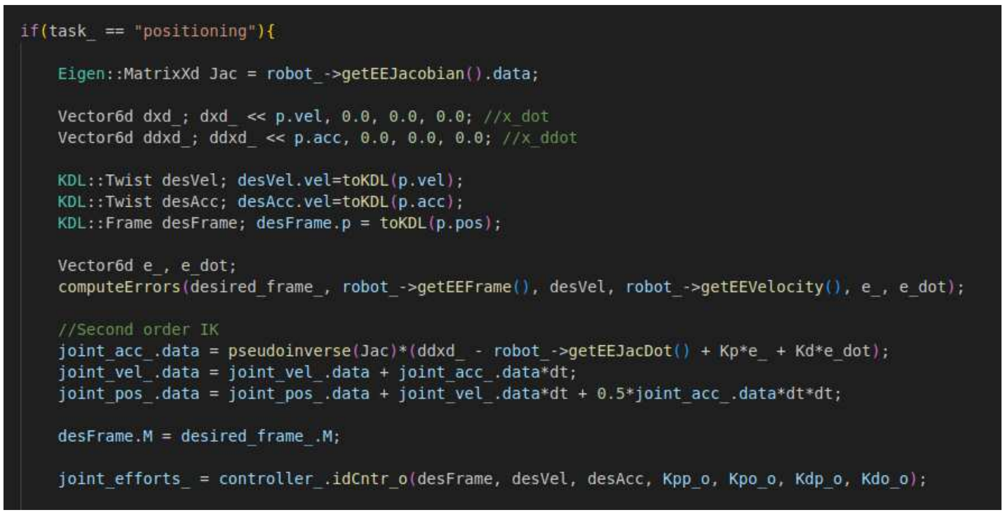


```
if(task_ == "look_at_point"){

    joint_velocities_ = look_at_point_cl();


}else if(task_ == "positioning"){

    joint_velocities_ = positioning();
```

Figure 2.9: Joint space positioning and look-at-point

2.2.3 Operational space controller - Positioning

For the positioning task with the operational space control, similarly to what we did in *Figure 2.3*, I computed a desired frame for our manipulator. I used this frame in order to plan a linear trajectory from the starting pose to the final one, and then I exploited these trajectory points to get twists and frames for our operational space control, and then used the desired frame rotation matrix to set the desired frame orientation.

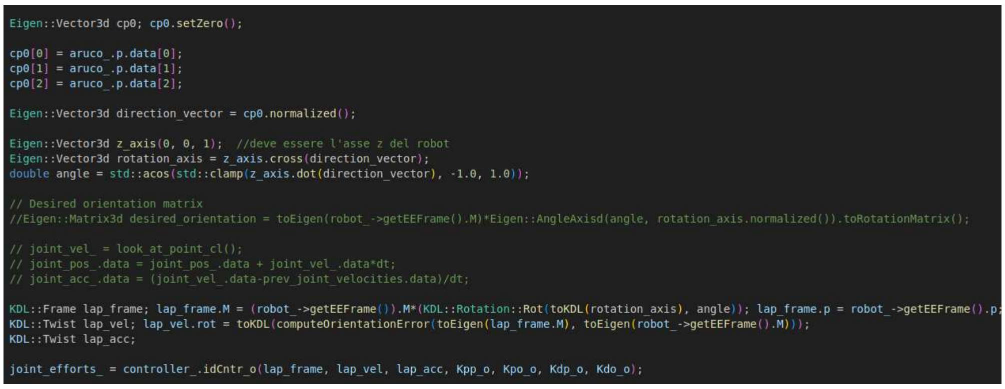


```
if(task_ == "positioning"){

    Eigen::MatrixXd Jac = robot_->getEEJacobian().data;

    Vector6d dxd_; dxd_ << p.vel, 0.0, 0.0, 0.0; //x_dot
    Vector6d ddxd_; ddxd_ << p.acc, 0.0, 0.0, 0.0; //x_ddot

    KDL::Twist desVel; desVel.vel=toKDL(p.vel);
    KDL::Twist desAcc; desAcc.vel=toKDL(p.acc);
    KDL::Frame desFrame; desFrame.p = toKDL(p.pos);

    Vector6d e_, e_dot;
    computeErrors(desired_frame_, robot_->getEEFrame(), desVel, robot_->getEEVelocity(), e_, e_dot);

    //Second order IK
    joint_acc_.data = pseudoinverse(Jac)*(ddxd_ - robot_->getEEJacDot() + Kp*e_ + Kd*e_dot);
    joint_vel_.data = joint_vel_.data + joint_acc_.data*dt;
    joint_pos_.data = joint_pos_.data + joint_vel_.data*dt + 0.5*joint_acc_.data*dt*dt;

    desFrame.M = desired_frame_.M;

    joint_efforts_ = controller_.idCntr_o(desFrame, desVel, desAcc, Kpp_o, Kpo_o, Kdp_o, Kdo_o);
```

Figure 2.10: Code snippet of positioning with operational space control

2.2.4 Operational space controller - Look-at-point

Instead, regarding the look-at-point task, I computed the desired frame by specifying its position as the next point of the trajectory and its rotation matrix as the desired rotation matrix, obtained by computing the desired orientation with the angle-axis representation. Then, I computed the desired twist by setting the linear velocity as the one determined by the planner and the angular velocity as the orientation error. Then, the desired acceleration is set to zero. The code is as follow.



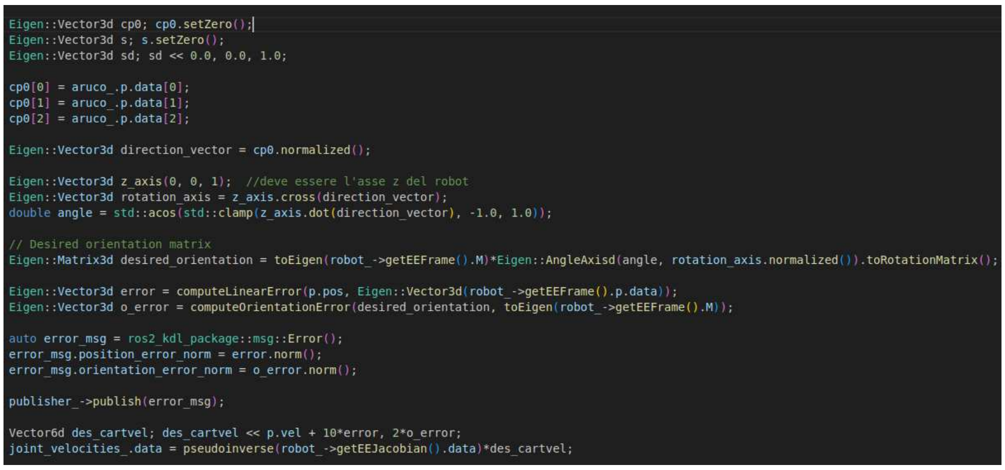
```
Eigen::Vector3d cp0; cp0.setZero();

cp0[0] = aruco_.p.data[0];
cp0[1] = aruco_.p.data[1];
cp0[2] = aruco_.p.data[2];

Eigen::Vector3d direction_vector = cp0.normalized();

Eigen::Vector3d z_axis(0, 0, 1);  //deve essere l'asse z del robot
Eigen::Vector3d rotation_axis = z_axis.cross(direction_vector);
double angle = std::acos(std::clamp(z_axis.dot(direction_vector), -1.0, 1.0));

// Desired orientation matrix
//Eigen::Matrix3d desired_orientation = toEigen(robot_->getEEFrame().M)*Eigen::AngleAxisd(angle, rotation_axis.normalized()).toRotationMatrix();

// joint_vel_ = look_at_point_cl();
// joint_pos_.data = joint_pos_.data + joint_vel_.data*dt;
// joint_acc_.data = (joint_vel_.data-prev_joint_velocities.data)/dt;

KDL::Frame lap_frame; lap_frame.M = (robot_->getEEFrame()).M*(KDL::Rotation::Rot(toKDL(rotation_axis), angle)); lap_frame.p = robot_->getEEFrame().p;
KDL::Twist lap_vel; lap_vel.rot = toKDL(computeOrientationError(toEigen(lap_frame.M), toEigen(robot_->getEEFrame().M)));
KDL::Twist lap_acc;

joint_efforts_ = controller_.idCntr_o(lap_frame, lap_vel, lap_acc, Kpp_o, Kpo_o, Kdp_o, Kdo_o);
```

Figure 2.11: Code snippet of look-at-point with operational space control

2.3 Merged controllers

The last point of the homework asked to merge the two controllers and enable the joint tracking of a linear position trajectory and the look-at-point vision-based task.

2.3.1 Joint space controller with look at point task and linear trajectory



```cpp
Eigen::Vector3d cp0; cp0.setZero();
Eigen::Vector3d s; s.setZero();
Eigen::Vector3d sd; sd << 0.0, 0.0, 1.0;

cp0[0] = aruco_.p.data[0];
cp0[1] = aruco_.p.data[1];
cp0[2] = aruco_.p.data[2];

Eigen::Vector3d direction_vector = cp0.normalized();

Eigen::Vector3d z_axis(0, 0, 1);  //deve essere l'asse z del robot
Eigen::Vector3d rotation_axis = z_axis.cross(direction_vector);
double angle = std::acos(std::clamp(z_axis.dot(direction_vector), -1.0, 1.0));

// Desired orientation matrix
Eigen::Matrix3d desired_orientation = toEigen(robot_->getEEFrame().M)*Eigen::AngleAxisd(angle, rotation_axis.normalized()).toRotationMatrix();

Eigen::Vector3d error = computeLinearError(p.pos, Eigen::Vector3d(robot_->getEEFrame().p.data));
Eigen::Vector3d o_error = computeOrientationError(desired_orientation, toEigen(robot_->getEEFrame().M));

auto error_msg = ros2_kdl_package::msg::Error();
error_msg.position_error_norm = error.norm();
error_msg.orientation_error_norm = o_error.norm();

publisher_->publish(error_msg);

Vector6d des_cartvel; des_cartvel << p.vel + 10*error, 2*o_error;
joint_velocities_.data = pseudoinverse(robot_->getEEJacobian().data)*des_cartvel;
```

Figure 2.12: Code snippet of merged controller in joint space

As it is possible to see from *Figure 2.12* , I first computed the vector that connects the aruco to the camera, in order to make the *z-axis* of the camera frame align to it. Then I proceeded by computing the error, both linear and orientation. The first one is the error between the next desired position computed by the planner and the actual end effector position, while the second one is the error between the desired orientation (computed in order to align the *z-axis* with the vector that unites the camera and the aruco) and the actual orientation of the end effector. In the following figures it is possible to see the evolution in time of the torque commands and the error norm.

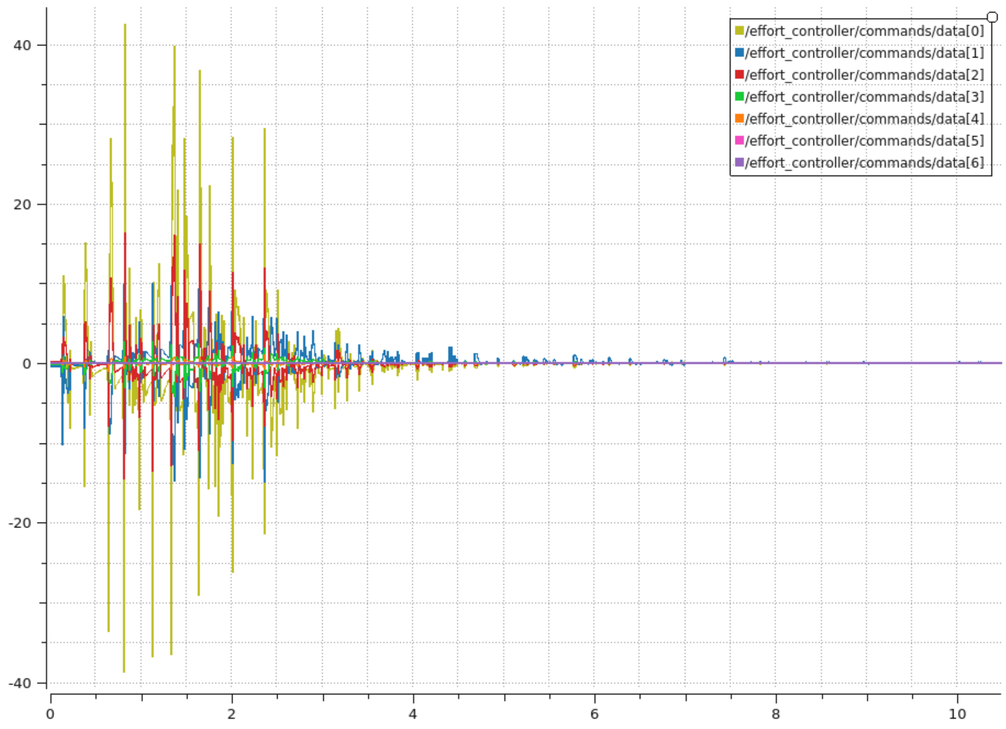


Figure 2.13: Torque commands for merged controller in joint space

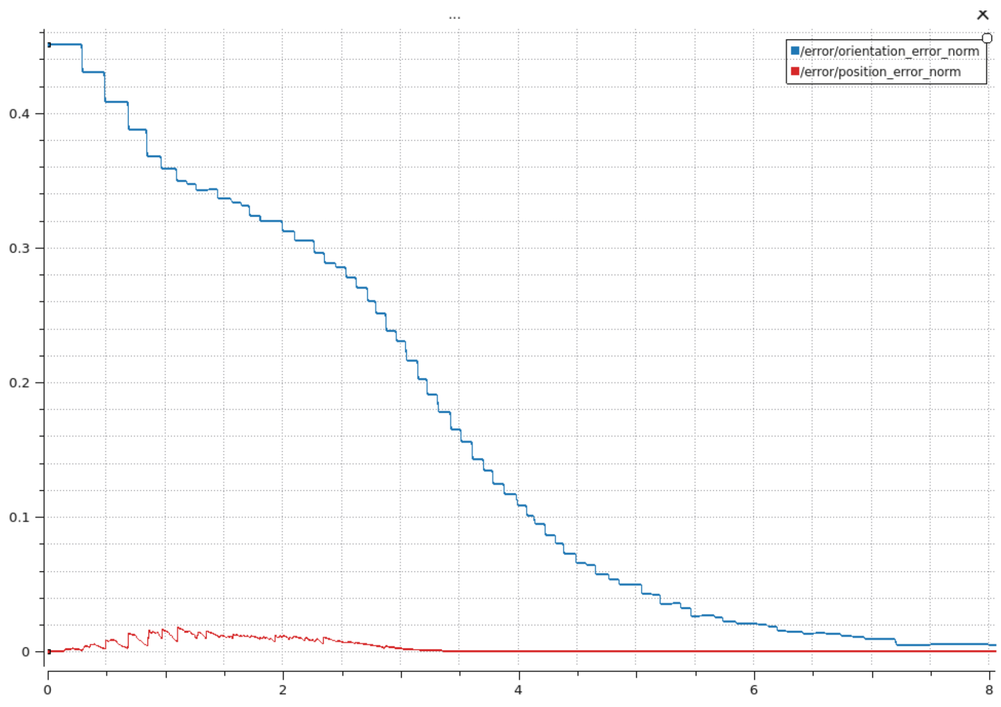
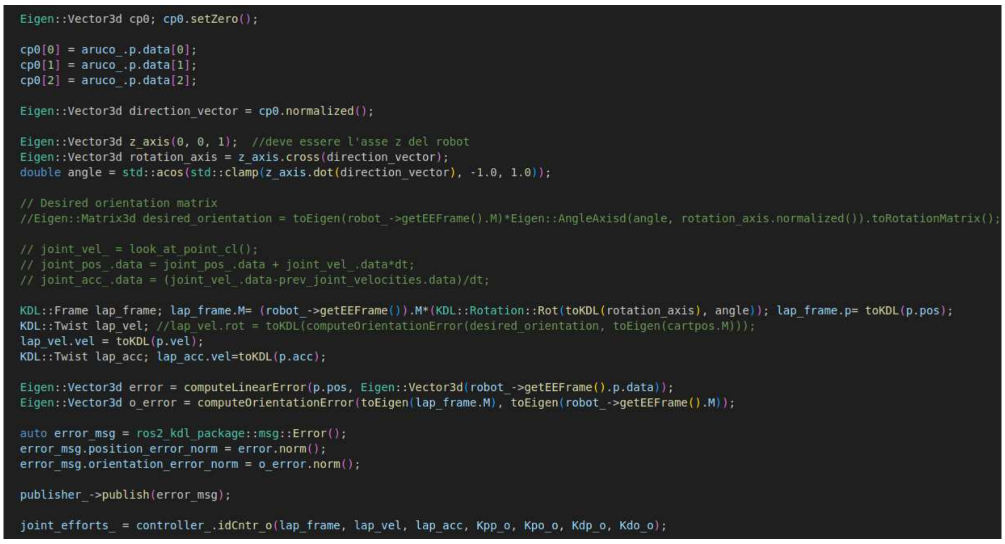


Figure 2.14: Error norm for merged controller in joint space

The related video is in the repository.

2.3.2 Operational space controller with look at point task and linear trajectory



```cpp
Eigen::Vector3d cp0; cp0.setZero();

cp0[0] = aruco_.p.data[0];
cp0[1] = aruco_.p.data[1];
cp0[2] = aruco_.p.data[2];

Eigen::Vector3d direction_vector = cp0.normalized();

Eigen::Vector3d z_axis(0, 0, 1);  //deve essere l'asse z del robot
Eigen::Vector3d rotation_axis = z_axis.cross(direction_vector);
double angle = std::acos(std::clamp(z_axis.dot(direction_vector), -1.0, 1.0));

// Desired orientation matrix
//Eigen::Matrix3d desired_orientation = toEigen(robot_->getEEFrame().M)*Eigen::AngleAxisd(angle, rotation_axis.normalized()).toRotationMatrix();

// joint_vel_ = look_at_point_cl();
// joint_pos_.data = joint_pos_.data + joint_vel_.data*dt;
// joint_acc_.data = (joint_vel_.data-prev_joint_velocities.data)/dt;

KDL::Frame lap_frame; lap_frame.M= (robot_->getEEFrame()).M*(KDL::Rotation::Rot(toKDL(rotation_axis), angle)); lap_frame.p= toKDL(p.pos);
KDL::Twist lap_vel; //lap_vel.rot = toKDL(computeOrientationError(desired_orientation, toEigen(cartpos.M)));
lap_vel.vel = toKDL(p.vel);
KDL::Twist lap_acc; lap_acc.vel=toKDL(p.acc);

Eigen::Vector3d error = computeLinearError(p.pos, Eigen::Vector3d(robot_->getEEFrame().p.data));
Eigen::Vector3d o_error = computeOrientationError(toEigen(lap_frame.M), toEigen(robot_->getEEFrame().M));

auto error_msg = ros2_kdl_package::msg::Error();
error_msg.position_error_norm = error.norm();
error_msg.orientation_error_norm = o_error.norm();

publisher_->publish(error_msg);

joint_efforts_ = controller_.idCntr_o(lap_frame, lap_vel, lap_acc, Kpp_o, Kpo_o, Kdp_o, Kdo_o);
```

Figure 2.15: Code snippet of merged controller in operational space

For what concerns the operational space merged controller, firstly I computed the vector that connects the aruco to the camera, as done before. Then, I set the desired frame position as the next point of the trajectory and its rotation matrix as the desired rotation matrix, obtained by computing the desired orientation with the angle-axis representation. Then, I computed the desired twist by setting the linear velocity as the one determined by the planner and the angular ones to zero. In the end, the desired acceleration is set to zero. In the following figures it is possible to see the evolution in time of the torque commands and the error norm.

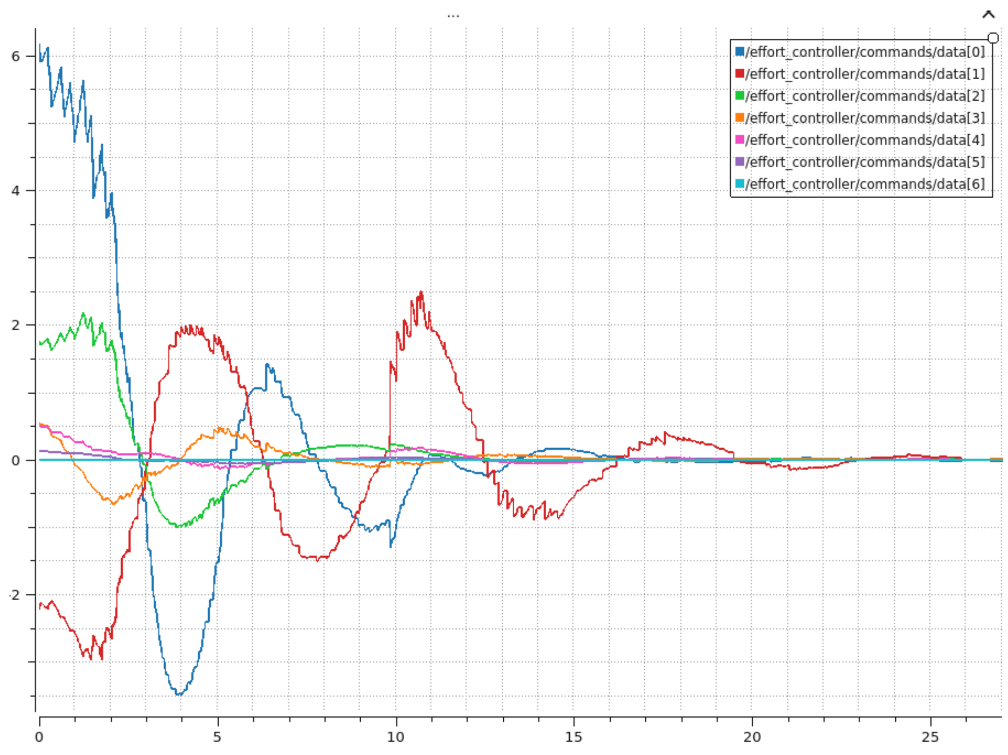


Figure 2.16: Torque commands for merged controller in joint space

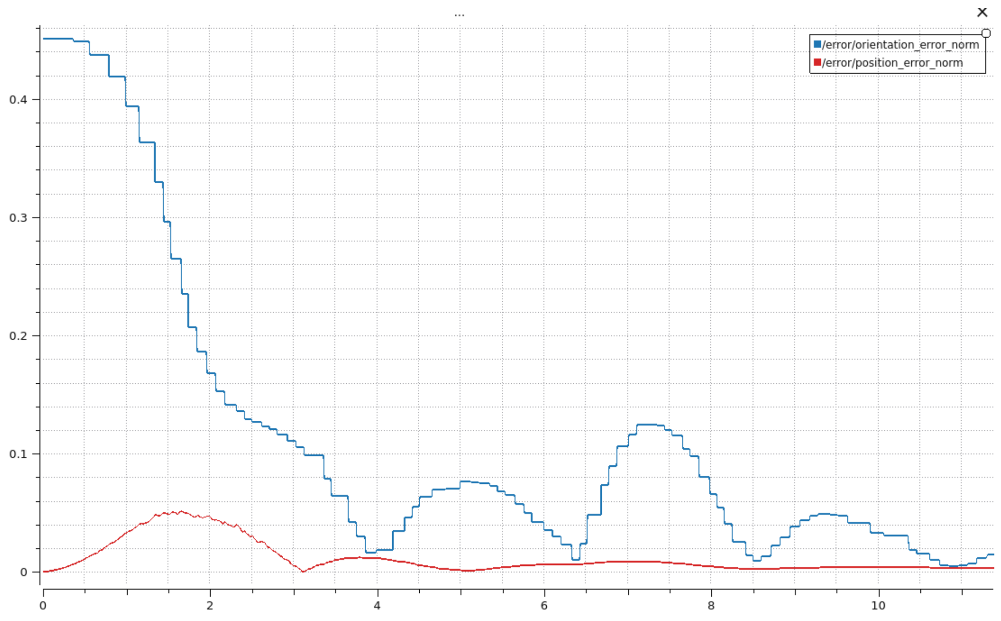


Figure 2.17: Error norm for merged controller in joint space

The related video is always in the reposiory:
https://github.com/danieffe/Vision-based-controller-for-a-7-DOF-robotic-manipulator-arm.git