Robotics Lab: Homework 3

Implement a vision-based task

Gaetano Torella
Stefano Riccardi
Marco Maffeo
Antonio D'Angelo

GitHub: https://github.com/marcomaffeo/homework3.git

This document contains the homework 3 of the Robotics Lab class.

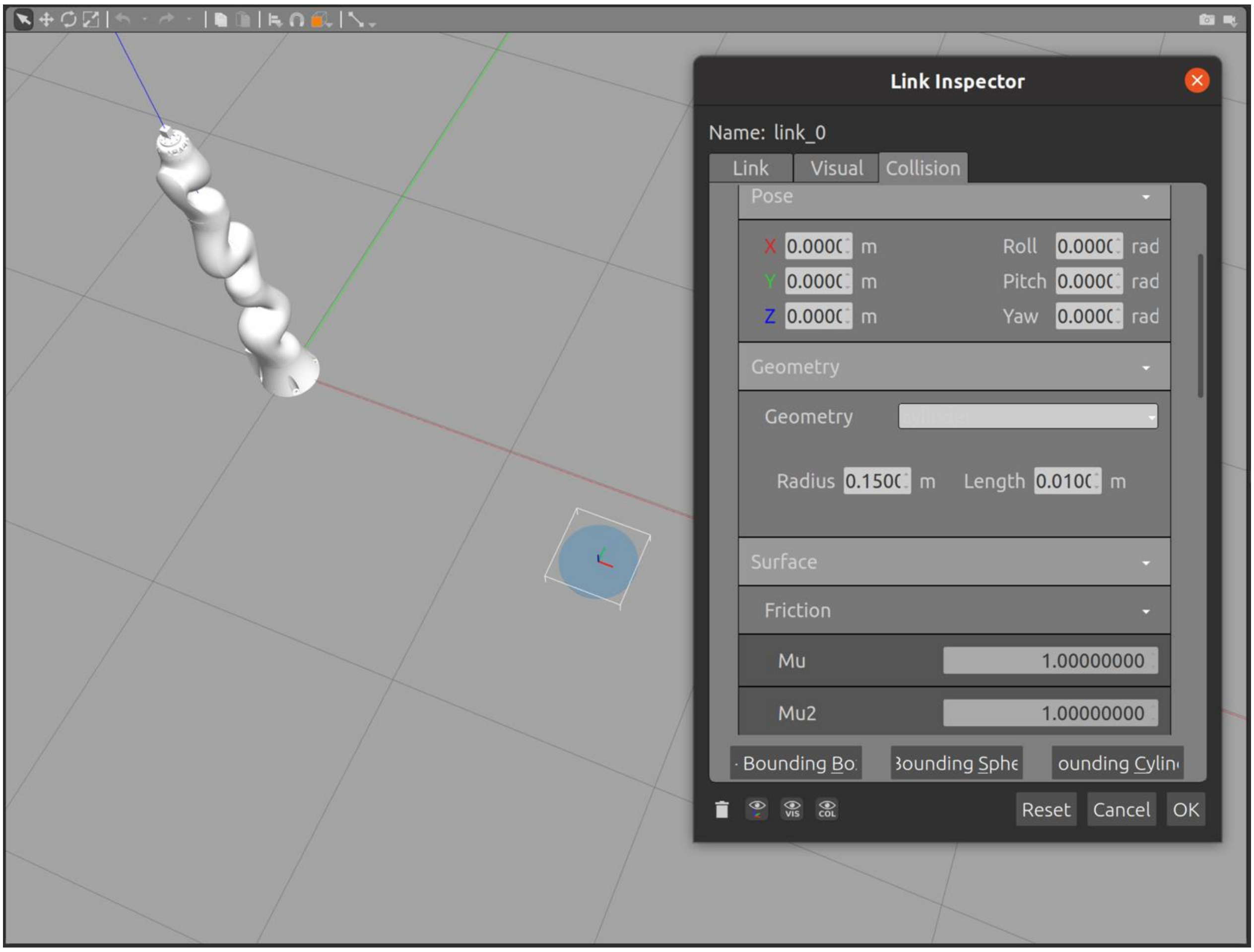
Implement a vision-based task

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) 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 instructor. The report is due in one week from the homework release.

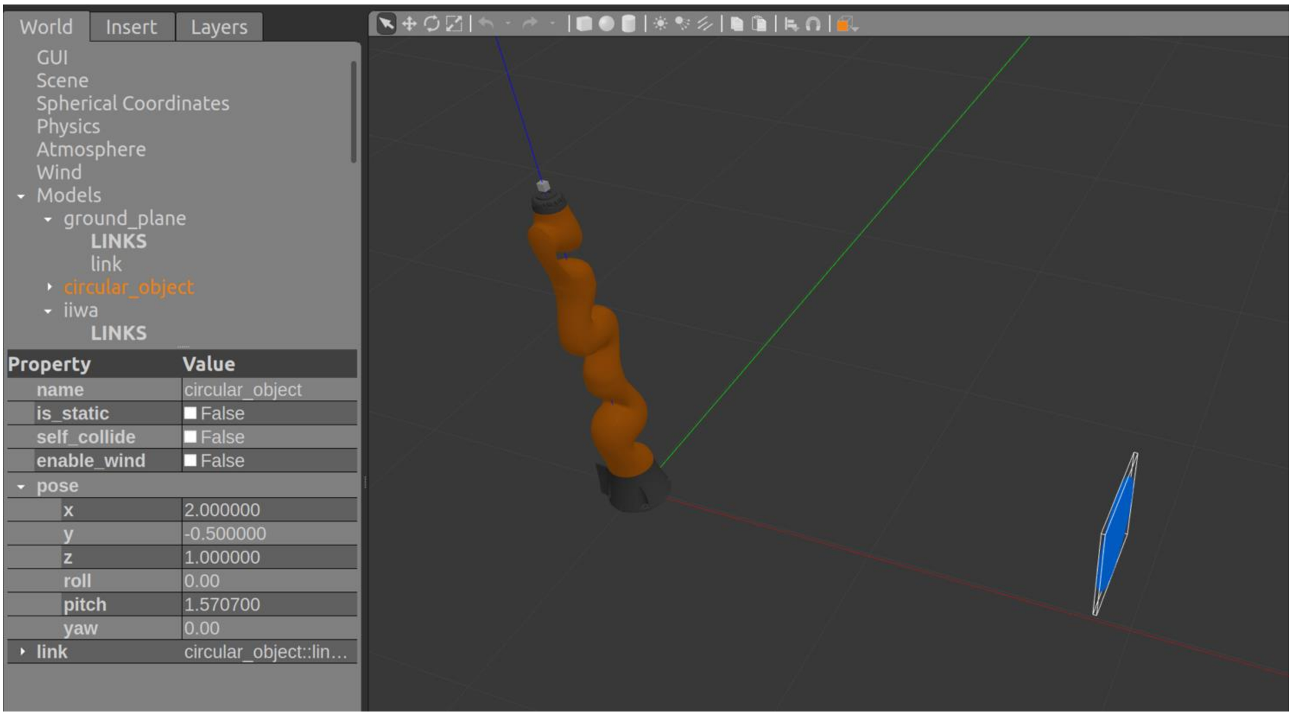
1. **Construct a gazebo world inserting a circular object and detect it via the *opencv_ros* package**

a) Go into the *iiwa_gazebo* package of the *iiwa_stack*. There you will find a folder model 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*.

Starting from the *iiwa_gazebo_aruco.launch* we decide to work directly in the Model Editor of the aruco marker so after delete it we add a Cylinder and then through the Link Inspector we change the Geometry and the Collision of this new link with 15 cm Radius and 1 cm Length. Finally, we change the color working on the RGB value and save the file inside the */iiwa_gazebo/models.*



Then we change the pose of our link and save it inside the */iiwa_gazebo/worlds/* as *iiwa_circular_object.world*. At the beginning we use the value of the reference pose but then we change it because the camera could not frame it very good. We can see below the values of poses chosen.



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 */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*).

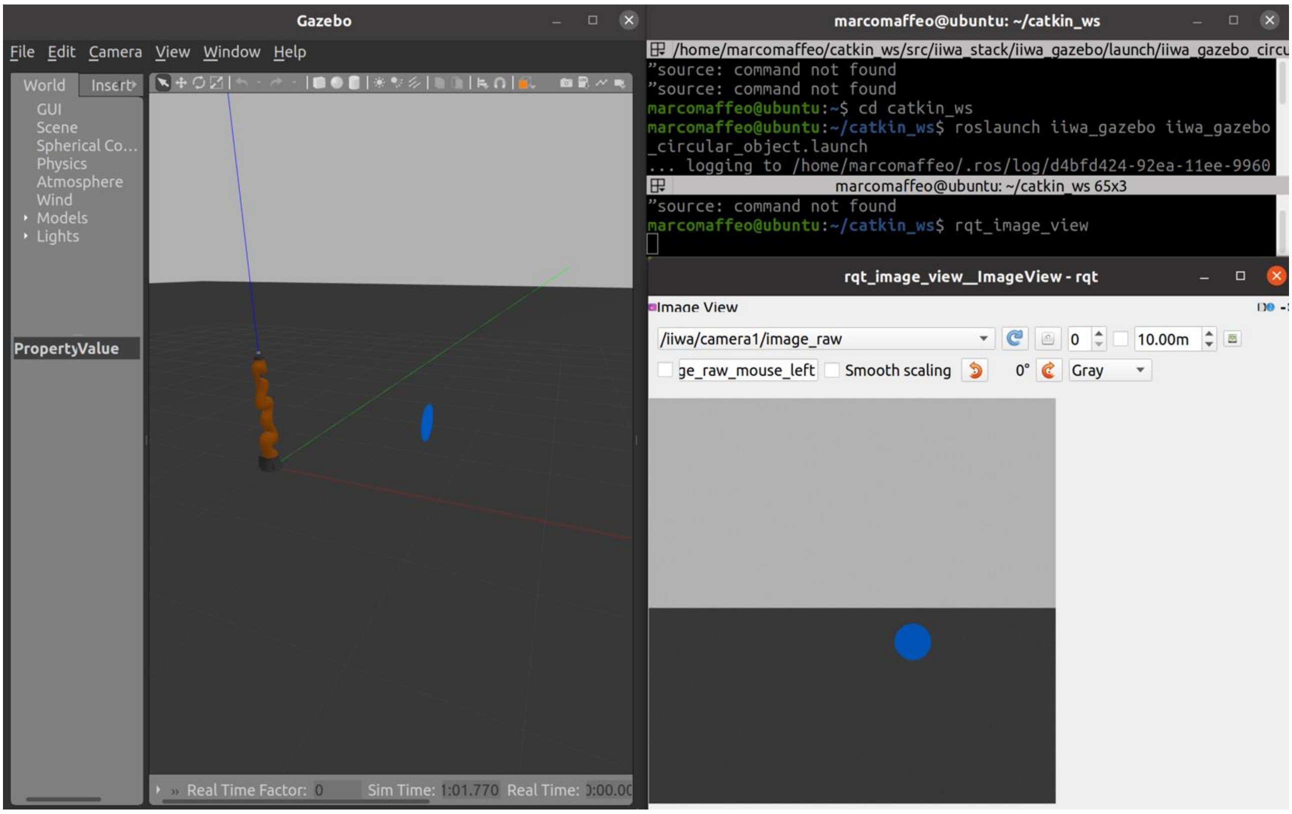
Inside the */iiwa_gazebo/launch/* we create a file named *iiwa_world_circular_object.launch* where we can recall the *iiwa_circular_object.world* that we created in the previous step as follow

```
<!-- 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>
```

Inside the */iiwa_gazebo/launch/* we create a file named *iiwa_gazebo_circular_object.launch* where we have the iiwa robot whit PositionJointInterface equipped with the camera and the *iiwa_world_circular_object.launch* as follow

```
<!-- 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>
```

Now we can easily check the result with *rqt_image_view* as follow.



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.

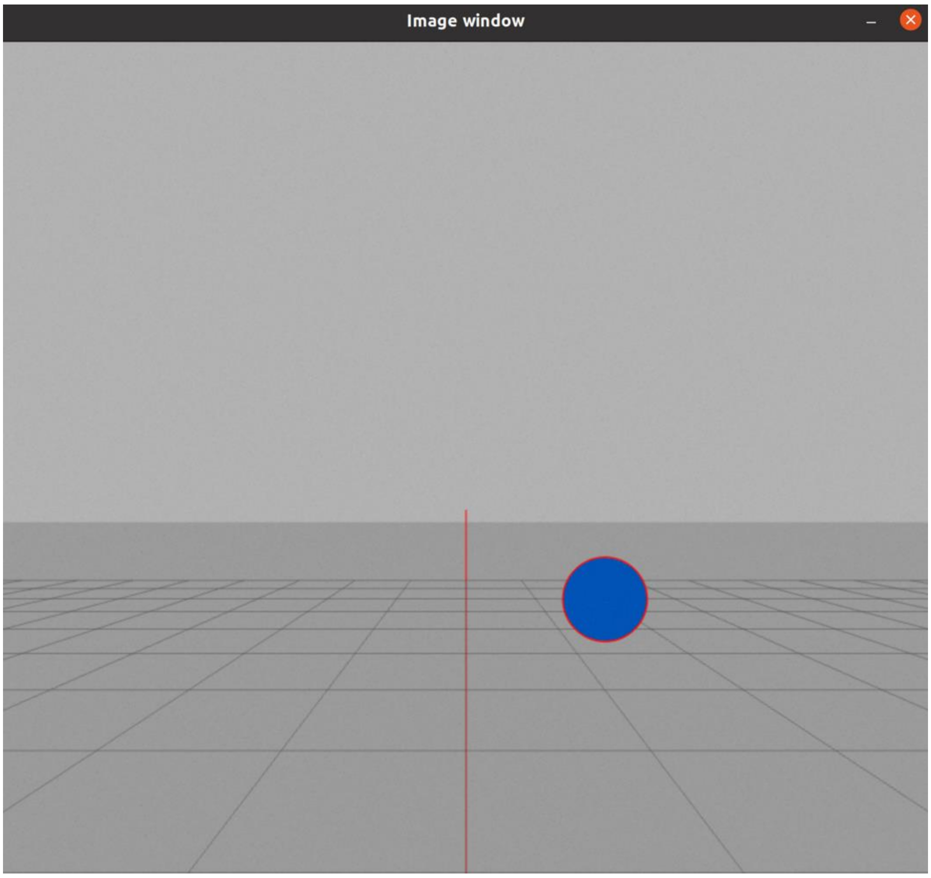
To detect the circular object, we must work inside the *opencv_ros_node.cpp* file using the OpenCV functions so taking Blob Detection Using OpenCV as a reference, we implemented the following code. Inside the public part of the class *ImageConverter* we add out camera called camera1

```
ImageConverter()
    : it_(nh_)
  {
    // Subscribe to input video feed and publish output video feed
    image_sub_ = it_.subscribe("/iiwa/camera1/image_raw", 1, &ImageConverter::imageCb, this);
    image_pub_ = it_.advertise("/image_converter/output_video", 1);

    namedWindow(OPENCV_WINDOW);
  }
```

Inside the void *imageCb* we implement the following code

```
// Set up the detector with default parameters.
    SimpleBlobDetector::Params params;
    params.minThreshold = 0;
    params.maxThreshold = 255;
    params.filterByCircularity = true;
    params.minCircularity = 0;
    params.maxCircularity = 1;

    Ptr<SimpleBlobDetector> detector = SimpleBlobDetector::create(params);

    Mat im = cv::Mat::zeros(cv_ptr->image.rows, cv_ptr->image.cols, cv_ptr->image.type());

    cvtColor(cv_ptr->image,im,COLOR_BGR2GRAY);

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

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

    // Show blobs
    imshow(OPENCV_WINDOW, cv_ptr->image);
    waitKey(0);

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

Finally, if we do the *rosrun opencv_ros opencv_ros_node* command we can see the Image window.



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.

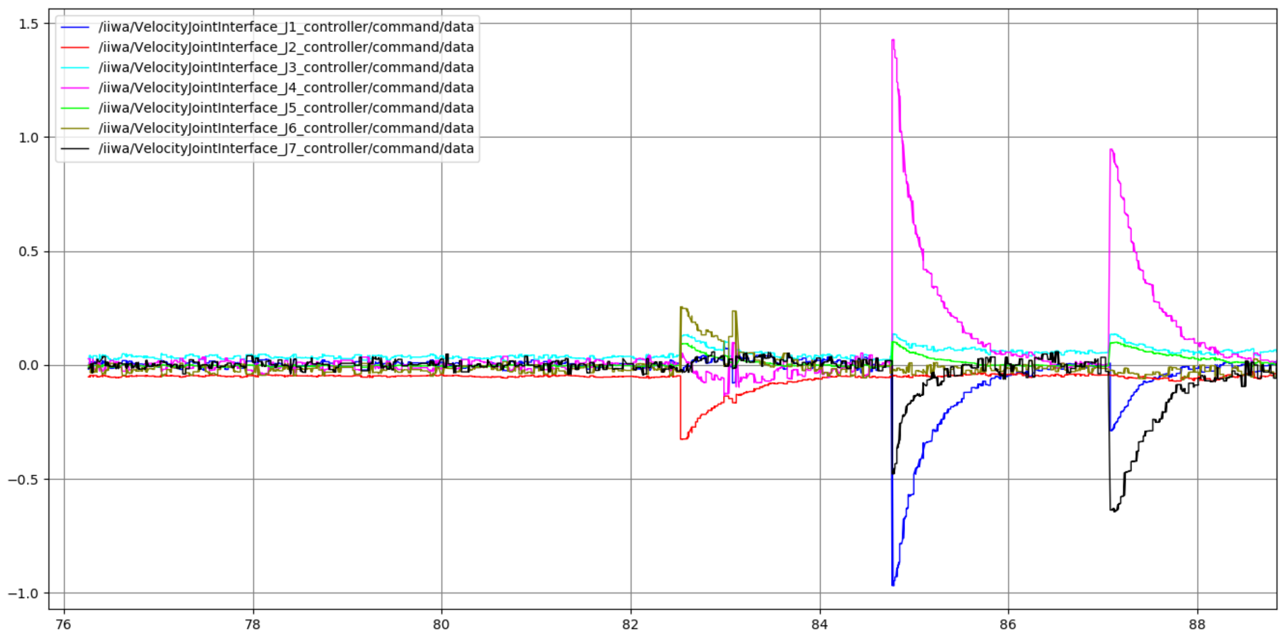
To implement a vision-based task that aligns the camera of the iiwa robot to the aruco marker we start creating a support frame called *frame_offset* initialized with the values of the *cam_T_object* frame; after that we shift this frame of 0.5 along the z-axis and rotate itself of 180° around the x-axis and then we have brought our frame at base-frame multiplying it by the *robot.getEEFrame().*

```
KDL::Frame frame_offset = cam_T_object;
frame_offset.p = cam_T_object.p - KDL::Vector(0,0,0.5);
frame_offset.M = cam_T_object.M* KDL::Rotation::RotX(-3.14);
KDL::Frame base_T_offset = robot.getEEFrame()* frame_offset;
```

After that we compute the orientation and linear errors, and used them in the velocity control law

```
Eigen::Matrix<double,3,1> e_o = computeOrientationError(toEigen(base_T_offset.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(base_T_offset.p),
toEigen(robot.getEEFrame().p));
x_tilde << e_p,  e_o_w[0], e_o[1], e_o[2];
. . .
dqd.data = lambda*J_pinv*x_tilde + 10*(Eigen::Matrix<double,7,7>::Identity() - J_pinv*J_cam.data) *
(qdi - toEigen(jnt_pos));
```

Here is the plot of the velocity commands moving the aruko marker:



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

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

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

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

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

First, we define a 3d vector *cPo* as the *pose* of the frame *cam_T_object* and we compute the unit-norm axis s as defined in the previous expression.

```
Eigen::Matrix<double,3,1> cPo = toEigen(cam_T_object.p);
Eigen::Matrix<double,3,1> s = cPo/cPo.norm();
```

We compute also the R and L matrix as follow

```
Eigen::Matrix<double,3,3> Rc = toEigen(robot.getEEFrame().M);
Eigen::Matrix<double,6,6> R_tot = Eigen::Matrix<double,6,6>::Zero();
R_tot.block(0,0,3,3) = Rc;
R_tot.block(3,3,3,3) = Rc;
Eigen::Matrix<double,3,6> L = Eigen::Matrix<double,3,6>::Zero();
Eigen::Matrix<double,3,3> L1 = (-1/cPo.norm()) * (Eigen::Matrix<double,3,3>::Identity() -
s*s.transpose());
Eigen::Matrix<double,3,3> L2 = skew(s);
L.block(0,0,3,3) = L1;
L.block(0,3,3,3) = L2;

L = L * (R_tot.transpose());
Eigen::MatrixXd L_J = L * toEigen(J_cam);
Eigen::MatrixXd L_J_pinv = L_J.completeOrthogonalDecomposition().pseudoInverse();
Eigen::MatrixXd N = (Eigen::Matrix<double,7,7>::Identity() - (L_J_pinv * L_J));
```

Finally, we implement the new control law:

```
dqd.data = lambda * L_J_pinv * Eigen::Vector3d(0,0,1) + 10 * N * (qdi - toEigen(jnt_pos));
```

To plot the s components, we declare three publishers for each s

```
ros::Publisher s1 = n.advertise<std_msgs::Float64>("/iiwa/s1", 1);
ros::Publisher s2 = n.advertise<std_msgs::Float64>("/iiwa/s2", 1);
ros::Publisher s3 = n.advertise<std_msgs::Float64>("/iiwa/s3", 1);
```

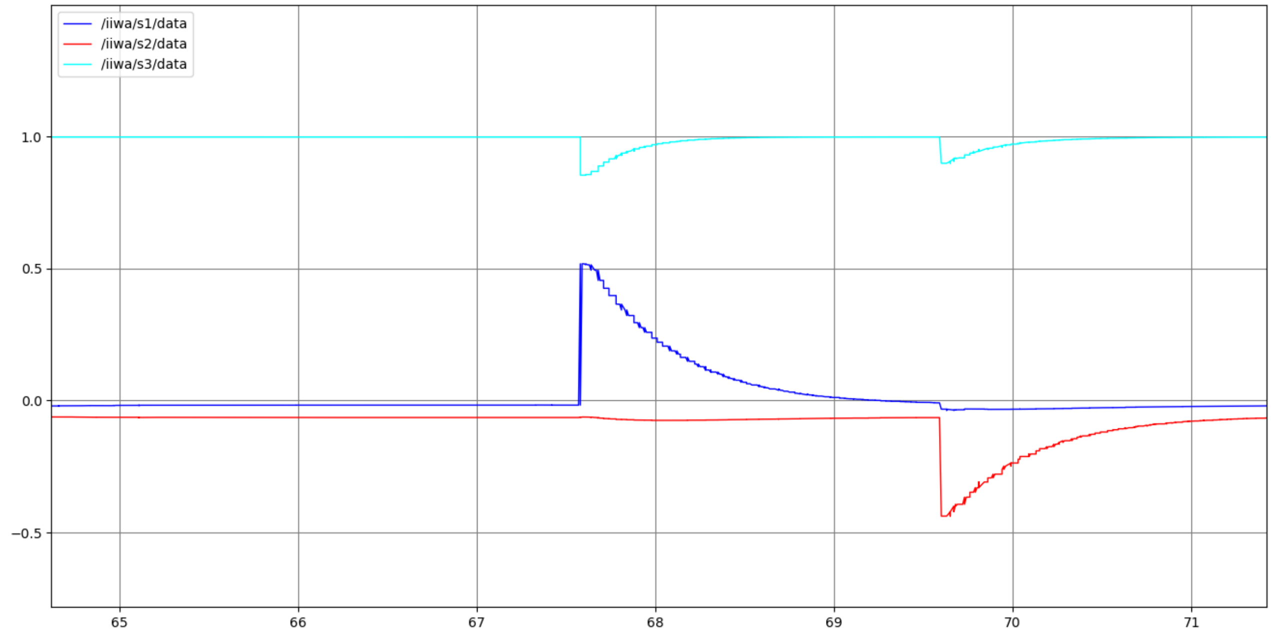
We define new standard messages

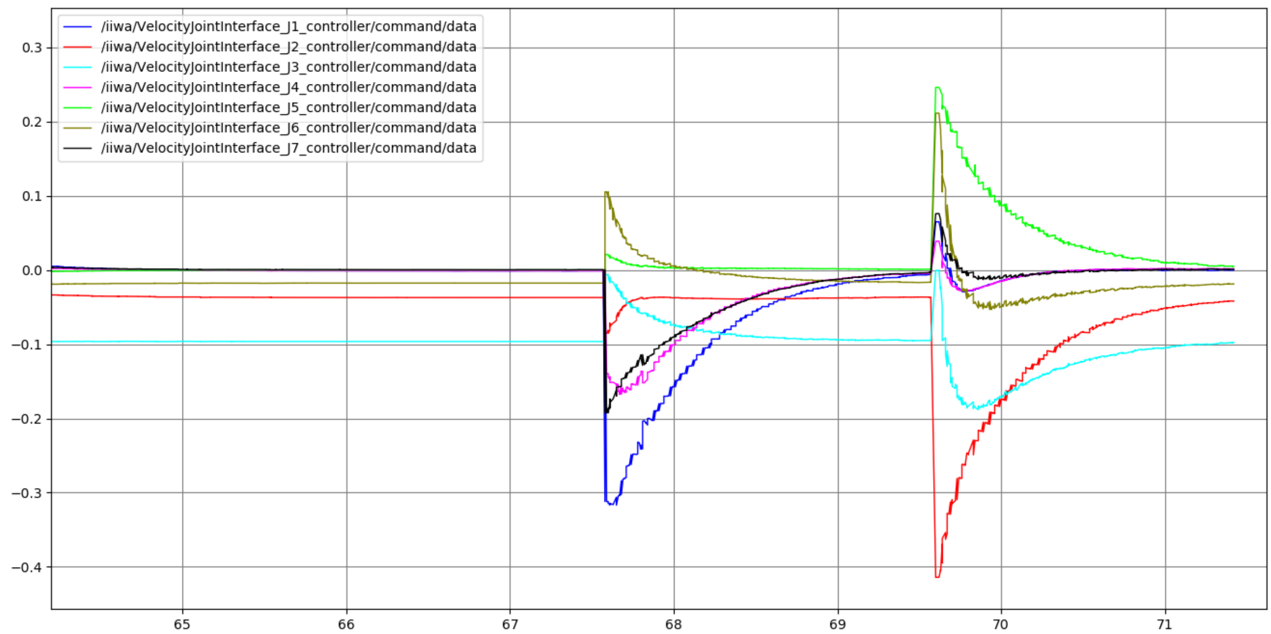
```
std_msgs::Float64 s1_msg,s2_msg,s3_msg;
```

and we publish these messages in the respective topic

```
s1_msg.data = s[0];
s2_msg.data = s[1];
s3_msg.data = s[2];
s1.publish(s1_msg);
s2.publish(s2_msg);
s3.publish(s3_msg);
```

These are the plot of the s components and of the velocity commands:





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 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.

For this purpose, we created a new file named *kdl_robot_vision_control_dynamic.cpp* in which we merged the two dynamics controller (Joint space inverse dynamics control and Cartisian space inverse dynamics control). Basically, the robot follows the circular and linear trajectory using the controllers of the previous homework and the look-at-point computed in the previous point of this homework. We created a new launch file named *iiwa_gazebo_aruco_dynamic.launch* where we imported the effort controller.

- *kdl_robot_vision_control_dynamic.cpp*

```
// Extract desired pose
des_cart_vel = KDL::Twist::Zero();
des_cart_acc = KDL::Twist::Zero();

// 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;

if (t <= init_time_slot) // wait a second
{
    //p = planner.compute_circolar_trajectory(0.0,acc_duration,s,dot_s,ddot_s);             //Point 2.b
    p = planner.compute_linear_trajectory(0.0,acc_duration,s,dot_s,ddot_s);    //Point 2.c
}
else if(t > init_time_slot && t <= traj_duration + init_time_slot)
{
    //p = planner.compute_circolar_trajectory(t-init_time_slot,acc_duration,s,dot_s,ddot_s);          //Point 2.b
    p = planner.compute_linear_trajectory(t-init_time_slot,acc_duration,s,dot_s,ddot_s);          //Point 2.c

    des_cart_vel = KDL::Twist(KDL::Vector(p.vel[0], p.vel[1], p.vel[2]),KDL::Vector::Zero());
    des_cart_acc = KDL::Twist(KDL::Vector(p.acc[0], p.acc[1], p.acc[2]),KDL::Vector::Zero());
}
else
{
    ROS_INFO_STREAM_ONCE("trajectory terminated");
    break;
}

des_pose.p = KDL::Vector(p.pos[0],p.pos[1],p.pos[2]);

// 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);

double Kp = 100; //to define
double Ko = 10; //to define

// Cartesian space inverse dynamics control
tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc, Kp, Ko, 2*sqrt(Kp), 2*sqrt(Ko));
```

- *iiwa_gazebo_aruco_dynamic.launch*

```
<arg name="hardware_interface" default="EffortJointInterface" />
<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="$(find iiwa_gazebo)/launch/iiwa_world_aruco.launch">
    <arg name="hardware_interface" value="$(arg hardware_interface)" />
    <arg name="robot_name" value="$(arg robot_name)" />
    <arg name="model" value="$(arg model)" />
</include>

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

    <include file="$(find iiwa_control)/launch/iiwa_control.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)" />
        <arg name="controllers" value="joint_state_controller
             iiwa_joint_1_effort_controller
             iiwa_joint_2_effort_controller
             iiwa_joint_3_effort_controller
             iiwa_joint_4_effort_controller
             iiwa_joint_5_effort_controller
             iiwa_joint_6_effort_controller
             iiwa_joint_7_effort_controller"/>
        <arg name="robot_name" value="$(arg robot_name)" />
        <arg name="model" value="$(arg model)" />
    </include>

</group>
```

Inside the *idCntr* function of the *kdl_robot_vision_control_dynamic.cpp* we changed the way we calculate the orientation error as follows:

```
Eigen::Matrix<double,3,1> dot_e_o = computeOrientationVelocityError(omega_d,
                                                                    omega_e,
                                                                    R_d,
                                                                    R_e);

Eigen::Matrix<double,3,1> e_o = computeOrientationError(toEigen(_desPos.M), toEigen(robot_->getEEFrame().M));
//Eigen::Matrix<double,3,1> e_o_w = computeOrientationError(toEigen(Fi.M), toEigen(robot.getEEFrame().M));
// x_tilde << e_p,  e_o_w[0], e_o[1], e_o[2];

Eigen::Matrix<double,6,1> x_tilde;
Eigen::Matrix<double,6,1> dot_x_tilde;
x_tilde << e_p, e_o;
dot_x_tilde << dot_e_p, dot_e_o;          //dot_e_o; -omega_e
dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;

// inverse dynamics
Eigen::Matrix<double,6,1> y;

KDL::Jacobian dotJacEE;
dotJacEE = robot_->getEEJacDotqDot1();

y << dot_dot_x_d - dotJacEE.data * robot_->getJntVelocities() + Kd*dot_x_tilde + Kp*x_tilde;

return M * (Jpinv*y + (I-Jpinv*J)*(/*- 10*grad */- 1*robot_->getJntVelocities()))
        + robot_->getGravity() + robot_->getCoriolis();

}
```

For the Joint space inverse dynamics control we change the way we calculate *qd* as follows:

```
des_pose.p = KDL::Vector(p.pos[0],p.pos[1],p.pos[2]);

// 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());

// Joint space inverse dynamics control
tau = controller_.idCntr(qd, dqd, ddqd, Kp, Kd);
```

To plot the results in terms of Cartesian error norm along the performed trajectory trajectories we have create two new topics *iiwa/position_norm_error* and *iiwa/orientation_norm_error* in the *kdl_robot_vision_control_dynamic.cpp* file

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

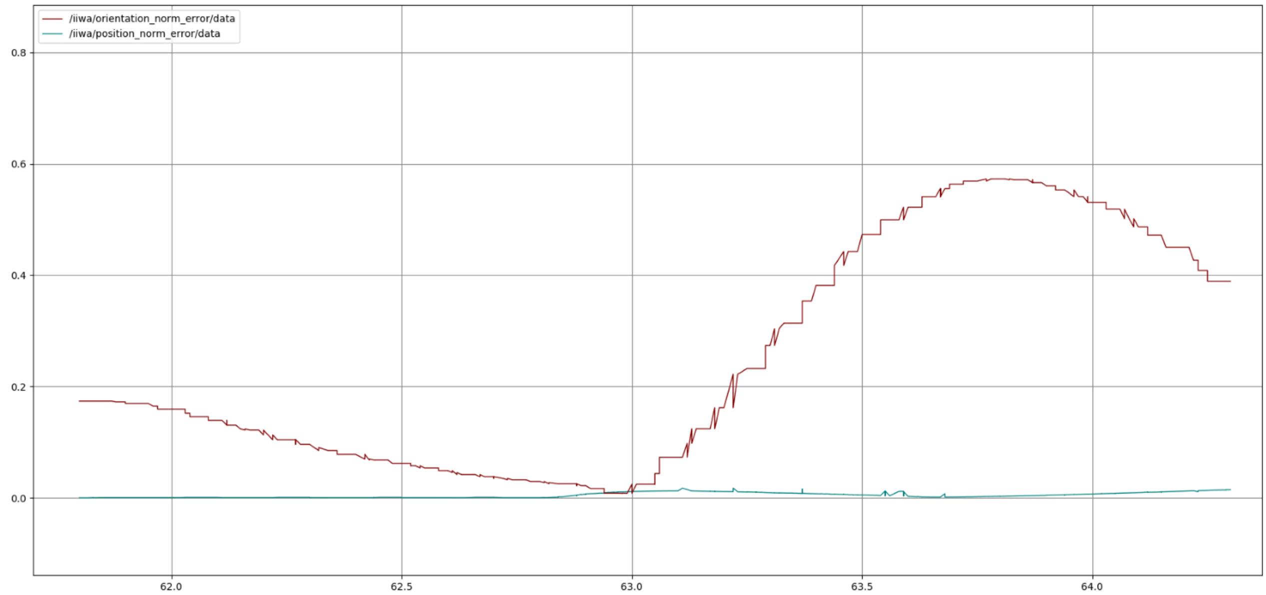
```
//Publish
position_err_msg.data = p_err;
position_norm_error.publish(position_err_msg);
orientation_err_msg.data = o_err;
orientation_norm_error.publish(orientation_err_msg);
```

We compute the two errors as follows:

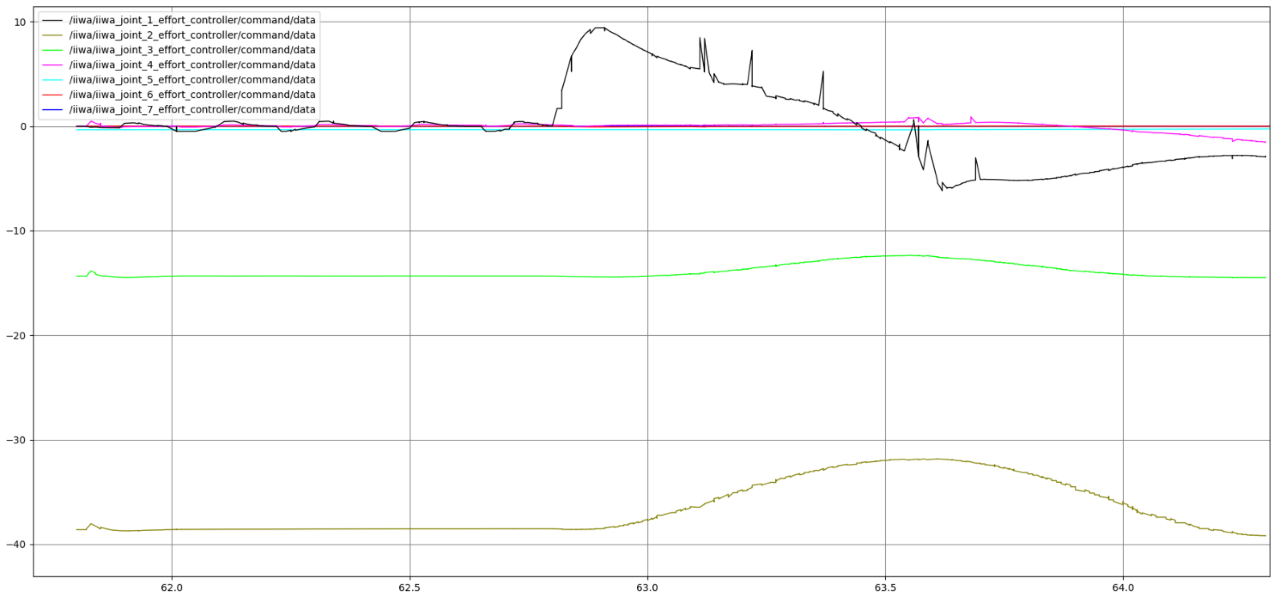
```
//Point 2.c compute Pose error:
double p_err = computeJointErrorNorm(toEigen(des_pose.p),toEigen(robot.getEEFrame().p));

//Orientiation error:
Eigen::Vector3d RPY_des;
des_pose.M.GetRPY(RPY_des[0],RPY_des[1],RPY_des[2]);
Eigen::Vector3d RPY_e;
robot.getEEFrame().M.GetRPY(RPY_e[0],RPY_e[1],RPY_e[2]);
Eigen::Vector3d o_e = RPY_des - RPY_e;
double o_err= o_e.norm();
std::cout << "errore orientamento: " << o_err;
```

We plot the results in terms of commanded joint torques and Cartesian error norm along the performed trajectory with the linear trajectory and Cartesian space inverse dynamics control.

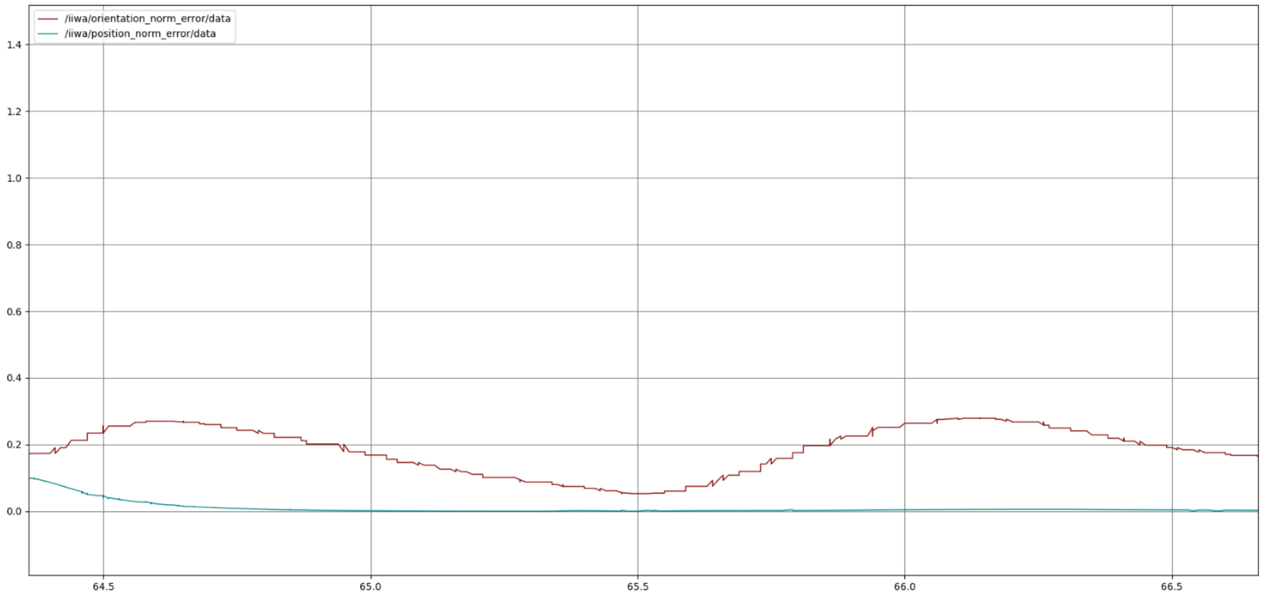
- Error norm for orientation and pose linear trajectory



- Joint effort command linear trajectory



- Error norm for orientation and pose circular trajectory



- Joint effort command circular trajectory