Report - Homework 3

Implement a vision-based task

Emanuele Cuzzocrea P38000196

Group:
Emanuele Cuzzocrea
Silvia Leo
Vito Daniele Perfetta
Giulia Gelsomina Romano

The code of this project and the related video can be found in my public github repository:
`https://github.com/EmanueleCuzzocrea/Homework3.git`

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.

A circular_object model has been created inside the .../iiwa_stack/iiwa_gazebo/models folder. This has been filled with model.sfd, model.config files and materials folder. In this last one, texture and scripts folders have been inserted in order to apply the red texture to the circular object. The requested model properties have been assigned inside model.sdf in the following way

```
<?xml version="1.0"?>
<sdf version="1.6">
  <model name="circular_object">
    <static>true</static>
    <pose>1 -0.5 0.6 0 -1.57 0</pose>
    <link name="link">

      <inertial>
        <pose>0 0 0 0 0 0</pose>
        <mass>0.001</mass>
        <inertia>
          <ixx>3.7499999999999997e-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 0.0005 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.15</radius>
            <length>0.001</length>
          </cylinder>
        </geometry>
        <material>
          <script>
            <uri>model://circular_object/materials/scripts</uri>
            <uri>model://circular_object/materials/textures</uri>
            <name>Red_</name>
          </script>
        </material>
      </visual>

      <visual name="rear_visual">
        <pose>0 0 -0.0005 0 0 0</pose>
        <geometry>
          <cylinder>
```

```
            <radius>0.15</radius>
            <length>0.001</length>
          </cylinder>
        </geometry>
      </visual>

      <collision name="collision">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.15</radius>
            <length>0.001</length>
          </cylinder>
        </geometry>
      </collision>

    </link>
  </model>
</sdf>
```

From now on, it is possible to include the circular object model inside the gazebo environment. Starting from the already existing world iiwa_aruco.world, the circular object has been properly positioned and oriented, while the robot and marker objects have been deleted before saving the world as iiwa_circular_object.world.

(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).

Using iiwa_gazebo_aruco.launch as reference, a new iiwa_gazebo_circular_object.launch has been created in such way to include the iiwa_world_circular_object.launch

```
...
    <!-- 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>
...
```

In the same way iiwa_world_circular_object.launch has been implemented as follows in order to include iiwa_circular_object.world

```
...
    <!-- 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>
```

```
...
```

It was noticed that starting from the default configuration the circular object was not entirely visualized by the camera. So that, the initial pose was changed with the following values using the additional file kdl_robot_init.cpp

```
...
    robot_init_config.request.joint_positions.push_back(3.14);
    robot_init_config.request.joint_positions.push_back(2.09);
    robot_init_config.request.joint_positions.push_back(-1.57);
    robot_init_config.request.joint_positions.push_back(1.57);
    robot_init_config.request.joint_positions.push_back(1.04);
    robot_init_config.request.joint_positions.push_back(-1.57);
    robot_init_config.request.joint_positions.push_back(1.57);
...
```

Used commands

Run each command in a different terminal

```
$ roslaunch iiwa_gazebo iiwa_gazebo_circular_object.launch
```

```
$ rosrun kdl_ros_control kdl_robot_init ./src/iiwa_stack/
   iiwa_description/urdf/iiwa14.urdf
```

```
$ rqt_image_view
```

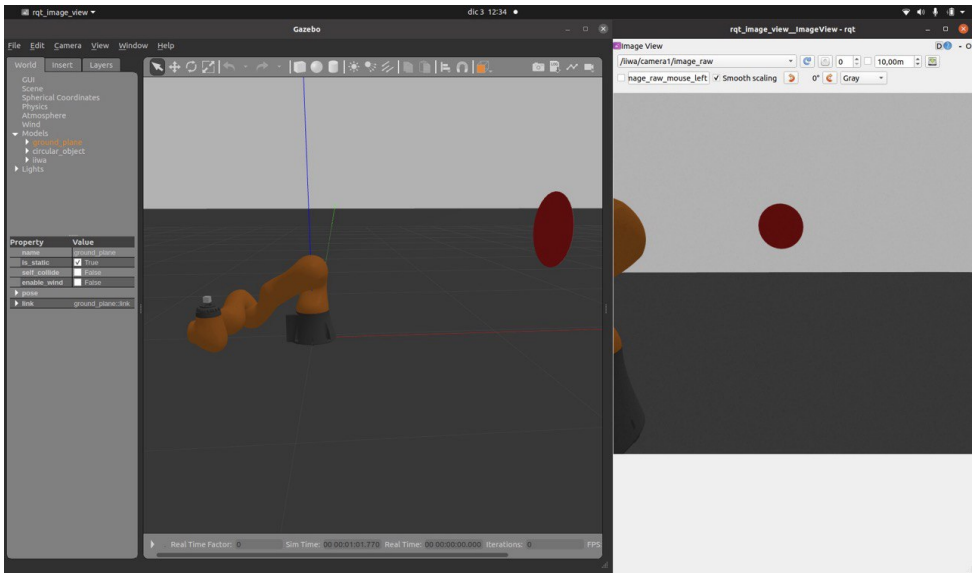

Figure 1: Assigned initial configuration

(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.

In order to detect the circular object, a blob detection function has been used in the opencv_-ros_node.cpp

```
...
  void imageCb(const sensor_msgs::ImageConstPtr& msg) {
```

```
    // Read image
    cv_bridge::CvImagePtr cv_ptr;
    try {
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::
          BGR8);
    }
    catch (cv_bridge::Exception& e) {
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
    }

    // Convert the image to grayscale
    Mat gray;
    cvtColor(cv_ptr->image, gray, COLOR_BGR2GRAY);

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

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

    // Filter by Area.
    params.filterByArea = false;
    //params.minArea = 10;

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

    // Filter by Convexity
    params.filterByConvexity = false;
    // params.minConvexity = 0.87;

    // Filter by Inertia
    params.filterByInertia = false;
    // params.minInertiaRatio = 0.7;

    Ptr<SimpleBlobDetector> detector = SimpleBlobDetector::create(params
        );
    // Detect blobs
    std::vector<KeyPoint> keypoints;
    detector->detect(gray, keypoints);

    // Draw blobs
    Mat im_with_keypoints;
    drawKeypoints(gray, keypoints, im_with_keypoints, Scalar(0, 0, 255),
        DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

    cv_ptr->image=im_with_keypoints;

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

It can be noticed that the image information provided by the camera has converted in grey scale since the blob detection function works better in this condition.

Used commands

Run each command in a different terminal

```
$ roslaunch iiwa_gazebo iiwa_gazebo_circular_object.launch
```

```
$ rosrun kdl_ros_control kdl_robot_init ./src/iiwa_stack/
   iiwa_description/urdf/iiwa14.urdf
```

```
$ rosrun opencv_ros opencv_ros_node
```

```
$ rqt_image_view
```

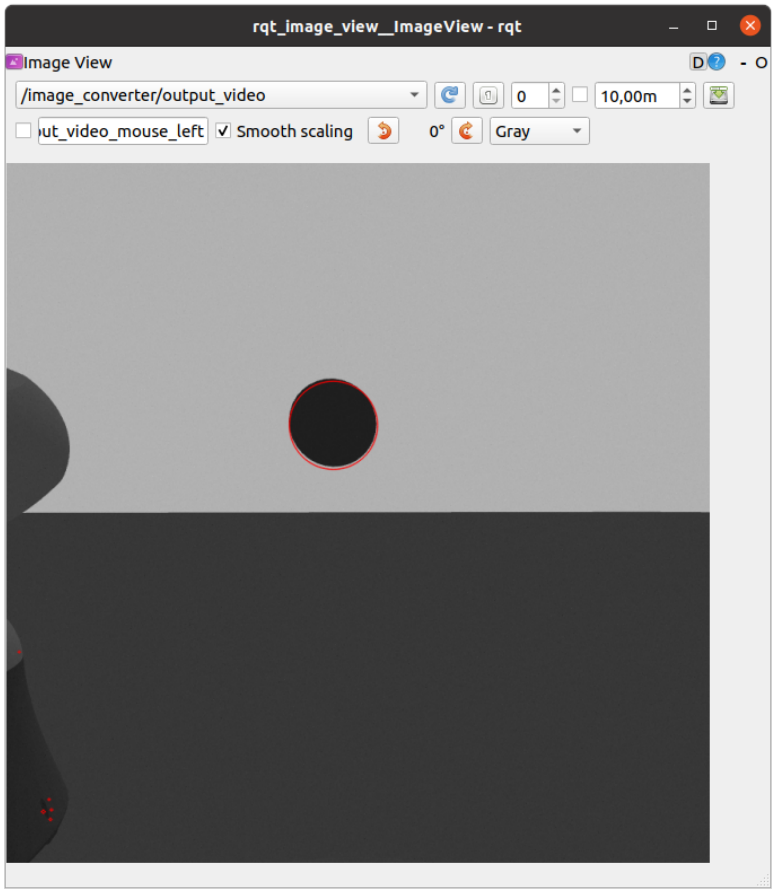

Figure 2: Circular object detection

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.

Knowing that the marker position can be expressed in the base frame with the following formula

$$p_m^b = p_c^b + R_c^b p_m^c$$

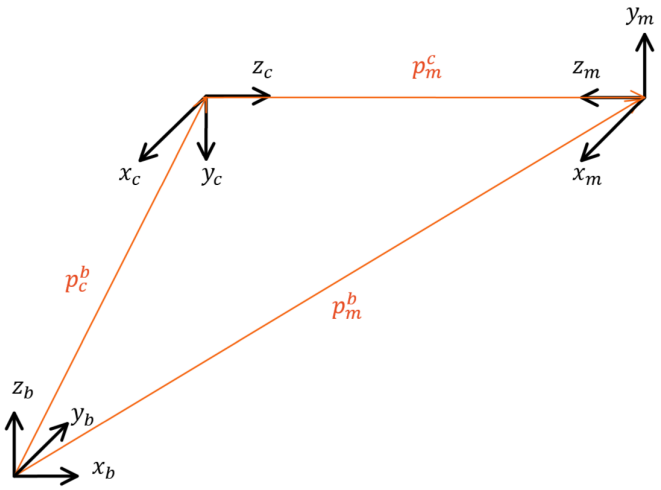

Figure 3: Marker position in the base frame

It is possible to define the desired camera position in the base frame assigning a position offset directly in the marker frame

$$p_{c_d}^b = p_m^b + R_c^b R_m^c p_{offset}^m$$

where p_{offset}^m has been chosen as $[0, 0, 0.4]$

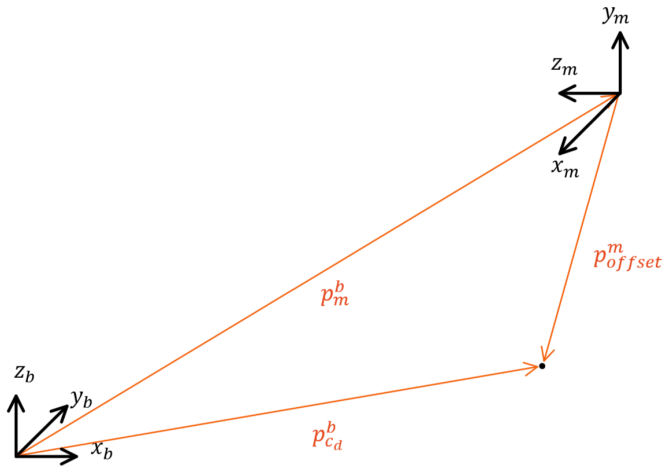

Figure 4: Desired camera frame position

For what concern the orientation offset, it can be assigned using a frame that initially coincides to the fixed marker frame and aligning it to the camera frame. It has been chosen an offset of 30°

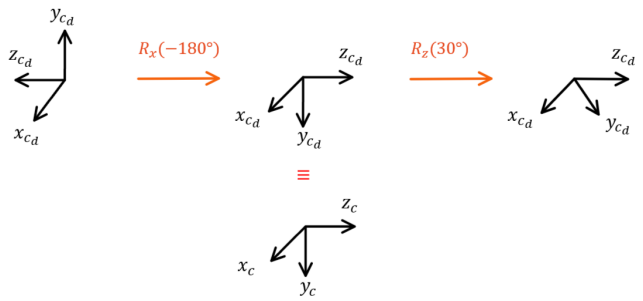

Figure 5: Desired camera frame orientation

This solution has been implemented in the following way

```
...
            ///look-at-point vision-based control with offsets ///

             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); // rotation to align zc to
                 s

             // offsets definition
             Eigen::Matrix<double,3,1> offset_pos = Eigen::Matrix<double,
                 3,1>::Zero();
             offset_pos<<0.0,0.0,0.4;
             double offset_or=0.5;

             Eigen::Matrix<double,3,1> pm_base =toEigen(robot.getEEFrame
                ().p) + toEigen(robot.getEEFrame().M)* toEigen(
                cam_T_object.p);//marker position wrt base frame
             Eigen::Matrix<double,3,1> pcd_base= pm_base +toEigen(robot.
                getEEFrame().M*cam_T_object.M)*offset_pos; // desired
                camera position wrt base frame

             // compute errors
             Eigen::Matrix<double,3,1> e_o = computeOrientationError(
                toEigen(robot.getEEFrame().M*Re), toEigen(robot.
                getEEFrame().M));
             Eigen::Matrix<double,3,1> e_o_w = computeOrientationError(
                toEigen(robot.getEEFrame().M*cam_T_object.M*KDL::Rotation
                ::RotX(-3.14)*KDL::Rotation::RotZ(offset_or)), toEigen(
                robot.getEEFrame().M));

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

             Eigen::Matrix<double,6,1> x_tilde; x_tilde << e_p,  e_o_w
                [0], e_o[1], e_o[2];

             //std::cout << "xtilde: "<< std::endl << x_tilde << std::
                endl; //debug

             // resolved velocity control low
             Eigen::MatrixXd J_pinv = J_cam.data.
```

```
            completeOrthogonalDecomposition().pseudoInverse();
        dqd.data = 0.5*lambda*J_pinv*x_tilde + 10*(Eigen::Matrix<
            double,7,7>::Identity() - J_pinv*J_cam.data)*(qdi -
            toEigen(jnt_pos));

...
```

Used commands

Run each command in a different terminal

```
$ roslaunch iiwa_gazebo iiwa_gazebo_aruco.launch
```

```
$ roslaunch aruco_ros usb_cam_aruco.launch camera:=/iiwa/camera1
```

```
$ rosrun kdl_ros_control kdl_robot_vision_control ./src/iiwa_stack/
    iiwa_description/urdf/iiwa14.urdf
```

```
$ rqt_image_view
```

```
$ rqt_plot
```

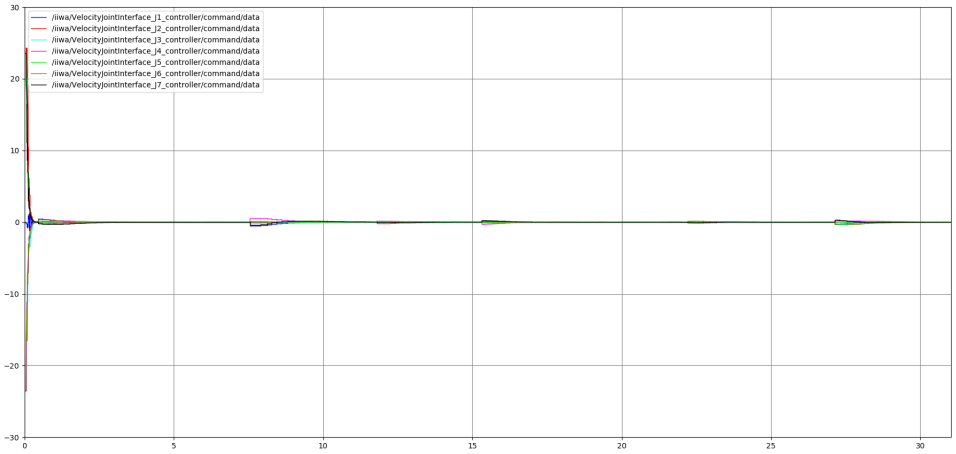

Figure 6: Velocity commands

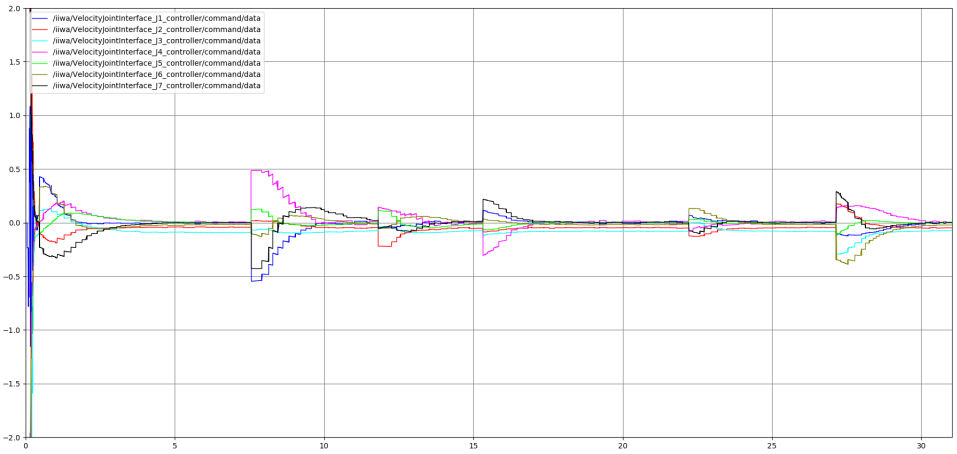

Figure 7: Velocity commands (zoomed view)

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

$$s = \frac{{}^cP_0}{||{}^cP_0||} \in S^2 \tag{1}$$

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

$$L(s) = \left[-\frac{1}{||{}^cP_0||}(I - ss^T) \quad S(s) \right] R^T \quad \text{with} \quad R = \begin{bmatrix} R_c & 0 \\ 0 & R_c \end{bmatrix} \tag{2}$$

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

This algorithm can be implemented as follows

```
...
            ///look-at-point vision-based control improved algorithm ///
            Eigen::Matrix<double,3,1> aruco_pos=toEigen(cam_T_object.p);
            Eigen::MatrixXd I3 = Eigen::MatrixXd::Identity(3, 3);
            Eigen::MatrixXd I7 = Eigen::MatrixXd::Identity(7, 7);

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

            Eigen::Matrix<double,3,3> L1=-(1/aruco_pos.norm())*(I3-(
                aruco_pos_n*aruco_pos_n.transpose()));
            Eigen::Matrix<double,3,3> L2=skew(aruco_pos_n);
            Eigen::Matrix<double,3,6> L=Eigen::Matrix<double, 3, 6>::
                Zero();
            L << L1,L2;
            Eigen::Matrix<double,6,6> R=Eigen::Matrix<double, 6, 6>::
                Zero();
            R.block(0,0,3,3)= Rc;
            R.block(3,3,3,3)= Rc;
            L=L*R.transpose();

            double k=2;
            Eigen::Matrix<double,3,1> sd=Eigen::Matrix<double, 3, 1>::
                Zero();
            sd<<0.0,0.0,1.0;

            Eigen::MatrixXd LJ = L*J_cam.data;
            Eigen::MatrixXd LJ_pinv = LJ.completeOrthogonalDecomposition
                ().pseudoInverse();


            dqd.data = k*LJ_pinv*sd+10*(I7 - LJ_pinv*LJ)*(qdi - toEigen(
                jnt_pos));
            std::cout << "s: "<< std::endl << aruco_pos_n << std::endl;
...
```

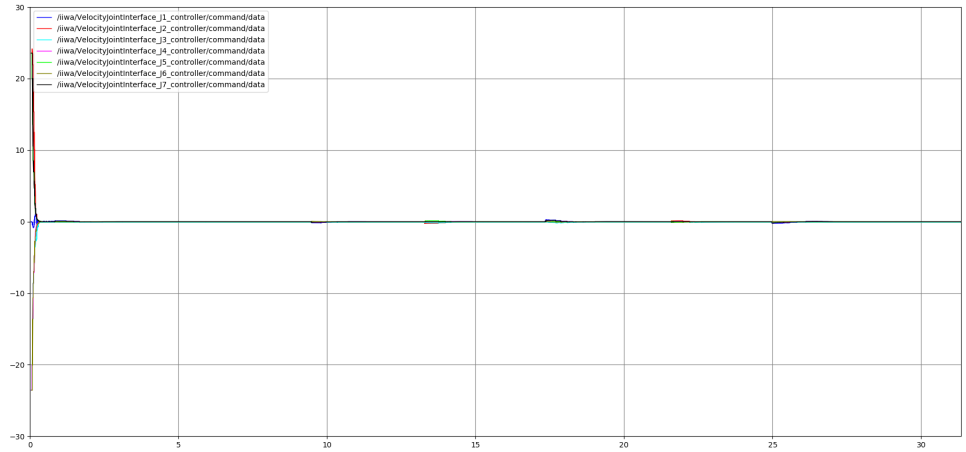

Figure 8: Velocity commands

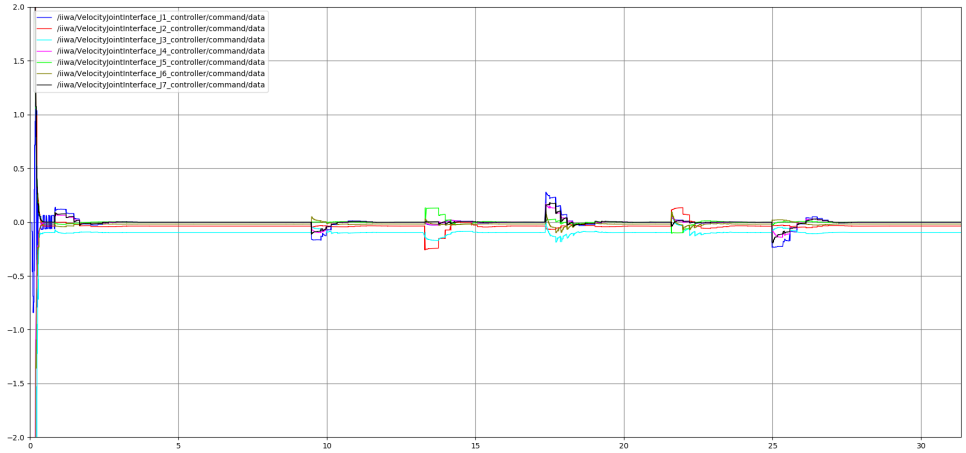

Figure 9: Velocity commands (zoomed view)

Figure 10: s components

It can be observed in the corresponding video that if the Null component is not considered, the iiwa robot still looks at the marker, even though it sags.

(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.

A new launch file named iiwa_gazebo_aruco_traj.launch has been coded starting from iiwa_-gazebo_aruco.launch, and making the following changes in order to merge these two controllers

```
...
    <arg name="hardware_interface" default="EffortJointInterface" />
    <arg name="robot_name" default="iiwa" />
    <arg name="model" default="iiwa14"/>
    <arg name="trajectory" default="false"/>
...
    <!-- 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
                   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>
...
```

In order to define the desired rotation matrix for the camera orientation, the following analysis has been performed. Starting from the unit vector s pointing towards the marker, two additional unit vectors have been defined in order to get an orthonormal frame. First of all, a unit vector w perpendicular to s and directed downward has been computed. To complete the right-handed frame, the unit vector u has been defined as the cross product of w and s. This frame depends only on the camera's position, not on its orientation. As a result, the unit vector s will serve as a fixed reference for the camera's z_c-axis, w will be the reference for y_c and u will be the reference for x_c. The analytical steps taken for the computation of the reference frame are shown below.
Computation of the height of the unit vector w with respect to the base frame

$$p_s^b = p_c^b + R_c^b p_s^c$$

$$a = p_s^b.z - p_c^b.z$$

$$\alpha = 1 \cdot \arcsin a$$

$$\beta = 1.57 - \alpha$$

$$b = \sin \beta$$

$$c = p_c^b.z - b$$

Computation of the x and y components of the unit vector w with respect to the camera frame

$$d = \cos \beta$$

$$e = d/\cos\alpha$$

$$p_w^c = e \cdot p_s^c$$

Final computation of the unit vector w with respect to the camera frame

$$p_w^b = p_c^b + R_c^b p_w^c$$

$$p_w^b.z = c$$

$$p_w^c = -R_b^c p_c^b + R_b^c p_w^b$$

Computation of the unit vector u with respect to the camera frame, and construction of the reference rotation matrix

$$p_u^c = p_w^c \times p_s^c$$

$$R_{c_d}^c = \begin{bmatrix} p_u^c & p_w^c & p_s^c \end{bmatrix}$$

$$R_{c_d}^b = R_c^b R_{c_d}^c$$

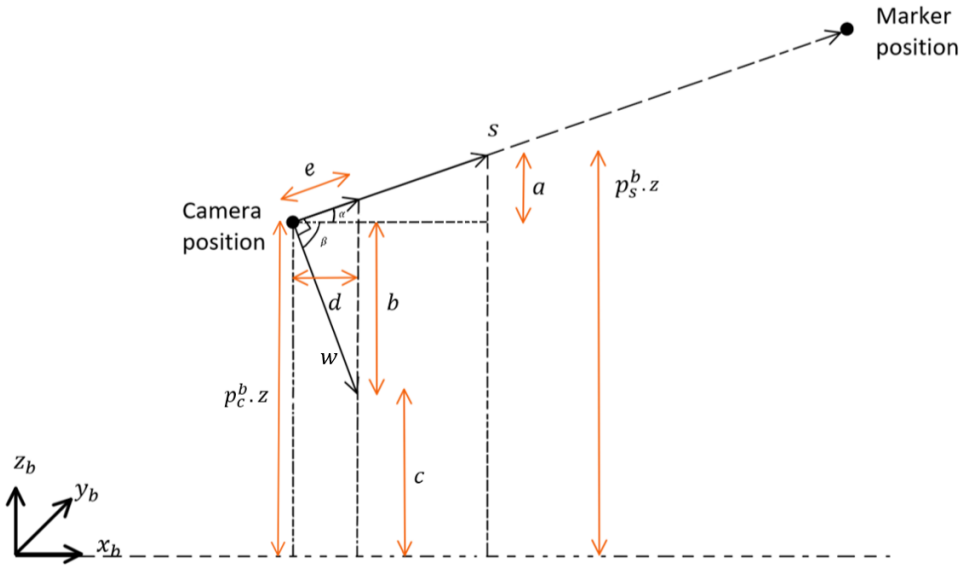

Figure 11: Construction of the reference frame

This reasoning has been implemented in the following code in kdl_robot_test.cpp

```
...
            Eigen::Matrix<double,3,1> s_b = toEigen(robot.getEEFrame().p
                ) + toEigen(robot.getEEFrame().M)*aruco_pos_n;
            Eigen::Matrix<double,3,1> camera_b = toEigen(robot.
                getEEFrame().p);
            double _a = s_b(2) - camera_b(2);
            double alpha = std::asin(_a); //s has norm = 1
            double beta = 1.57 - alpha;
            double _b = std::sin(beta); //w has norm = 1
            double _c = camera_b(2) - _b;
            double _d = std::cos(beta);
            double _e = _d/std::cos(alpha);

            Eigen::Matrix<double,3,1> w = _e*aruco_pos_n; // assigning x
                and y to w (the z component is corrected below)
            Eigen::Matrix<double,3,1> w_b = toEigen(robot.getEEFrame().p
                ) + toEigen(robot.getEEFrame().M)*w;
            w_b(2) = _c;
```

```
            w= -(toEigen(robot.getEEFrame().M)).transpose()*toEigen(
                robot.getEEFrame().p) + (toEigen(robot.getEEFrame().M)).
                transpose()*w_b;
            w.normalize();

            Eigen::Matrix<double,3,1> u = skew(w)*aruco_pos_n;
            u.normalize();
            KDL::Rotation RefOrFrame(
                u(0), w(0), aruco_pos_n(0),
                u(1), w(1), aruco_pos_n(1),
                u(2), w(2), aruco_pos_n(2)
            );
             Eigen::MatrixXd des_pose_n(3,3);

            for (int i = 0; i < 3; ++i)
                 for (int j = 0; j < 3; ++j)
                      des_pose_n(i,j) = RefOrFrame(i,j);

            des_pose_n = matrixOrthonormalization(des_pose_n);

            for (int i = 0; i < 3; ++i)
                 for (int j = 0; j < 3; ++j)
                      RefOrFrame(i,j) = des_pose_n(i,j);

...
```

Once the desired rotation matrix for the camera orientation has been computed, the joint space and the Cartesian space inverse dynamics controllers developed in the previous homework have been modified as follow

```
...
           /////// INVERSE KINEMATICS ///////
            //position and orientation compensation between flange and
                end effector (camera)
            des_pose.p = KDL::Vector(p.pos[0],p.pos[1],p.pos[2])-
                ee_T_cam.p;
            //des_pose.M = robot.getEEFrame().M*RefOrFrame*KDL::Rotation
                ::RotZ(0.5)*KDL::Rotation::RotZ(1.57)*KDL::Rotation::RotY
                (-1.57); //offset 30deg
            des_pose.M = robot.getEEFrame().M*RefOrFrame*KDL::Rotation::
                RotZ(1.57)*KDL::Rotation::RotY(-1.57);

            //retrieve current values
            qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3],
                jnt_pos[4], jnt_pos[5], jnt_pos[6];
            dqd.data << jnt_vel[0], jnt_vel[1], jnt_vel[2], jnt_vel[3],
                jnt_vel[4], jnt_vel[5], jnt_vel[6];

            //update desired values
            ddqd = robot.getInvAccKin(qd,dqd,des_cart_acc);
            dqd = robot.getInvVelKin(qd, des_cart_vel);
            qd = robot.getInvKin(qd, des_pose);

            /////// JOINT SPACE INVERSE DYNAMICS CONTROL  ///////
            // Gains
            double Kp = 20, Kd = 10;
            tau = controller_.idCntr(qd, dqd, ddqd, Kp, Kd);
...
```

It is relevant to notice that a position and orientation compensation has been done in order to transform the Cartesian camera pose reference into the flange one. This is done because the inverse kinematics functions compute the desired joint values assuming the flange as end effector.

```
...
            ///////  OPERATIONAL SPACE INVERSE DYNAMICS CONTROL  ///////

            des_pose.p = KDL::Vector(p.pos[0],p.pos[1],p.pos[2]);
            //des_pose.M = robot.getEEFrame().M*RefOrFrame*KDL::Rotation
                ::RotZ(0.5);
            des_pose.M = robot.getEEFrame().M*RefOrFrame;

            // Gains
            double Kp = 50, Kd = 10;

            tau = controller_.idCntr(des_pose, des_cart_vel,
                des_cart_acc, Kp, Kd, 2*sqrt(Kp), 2*sqrt(Kd));
...
```

In the operational space inverse dynamics control, the previous compensation isn't necessary anymore because the inverse kinematics functions haven't been used.

An alternative way to proceed is presented below. In this case, instead of assigning the desired rotation matrix, it is possible to exploit the orientation error as defined in the look-at-point vision-based control in 2a

```
...
            //Solution assigning orientation error
            //double offset_or=0.5;
            Eigen::Matrix<double,3,1> e_o = computeOrientationError(
                toEigen(robot.getEEFrame().M*Re), toEigen(robot.
                getEEFrame().M));
            //Eigen::Matrix<double,3,1> e_o_w = computeOrientationError(
                toEigen(robot.getEEFrame().M*cam_T_object.M*KDL::Rotation
                ::RotX(-3.14)*KDL::Rotation::RotZ(offset_or)), toEigen(
                robot.getEEFrame().M));
            Eigen::Matrix<double,3,1> e_o_w = computeOrientationError(
                toEigen(robot.getEEFrame().M*cam_T_object.M*KDL::Rotation
                ::RotX(-3.14)), toEigen(robot.getEEFrame().M));
            //Eigen::Matrix<double,3,1> e_o_w = computeOrientationError(
                toEigen(init_cart_pose.M), toEigen(robot.getEEFrame().M))
                ;

            Eigen::Matrix<double,3,1> e_or; e_or << e_o_w[0], e_o[1],
                e_o[2];

            ...

            tau = controller_.idCntr(des_pose.p,e_or,des_cart_vel,
                des_cart_acc, Kp, Kd, 2*sqrt(Kp), 2*sqrt(Kd)); //
                neglecting L,assuming wd=0
...
```

The prototype and the implementation of the controller_.idCntr function have been coded respectively in kdl_control.h and kdl_control.cpp

```
...
Eigen::VectorXd idCntr(KDL::Vector &_desPos_p,
                       Eigen::Matrix<double,3,1> &_e_or,
                       KDL::Twist &_desVel,
```

```
                                KDL::Twist &_desAcc,
                                double _Kpp,
                                double _Kpo,
                                double _Kdp,
                                double _Kdo);
...
```

```
...
// compute orientation errors
Eigen::Matrix<double,3,1> dot_e_o =-omega_e;

Eigen::Matrix<double,6,1> x_tilde = Eigen::MatrixXd::Zero(6, 1);
...
```

It is possible to notice that the L matrix, defined in the angular velocity error, will be neglected if the torque is assigned without using the rotation matrix.

Used commands

Run each command in a different terminal

```
$ roslaunch iiwa_gazebo iiwa_gazebo_aruco_traj.launch
```

```
$ roslaunch aruco_ros usb_cam_aruco.launch camera:=/iiwa/camera1
```

```
$ rosrun kdl_ros_control kdl_robot_test ./src/iiwa_stack/
   iiwa_description/urdf/iiwa14.urdf
```

```
$ rqt_image_view
```

```
$ rqt_plot
```

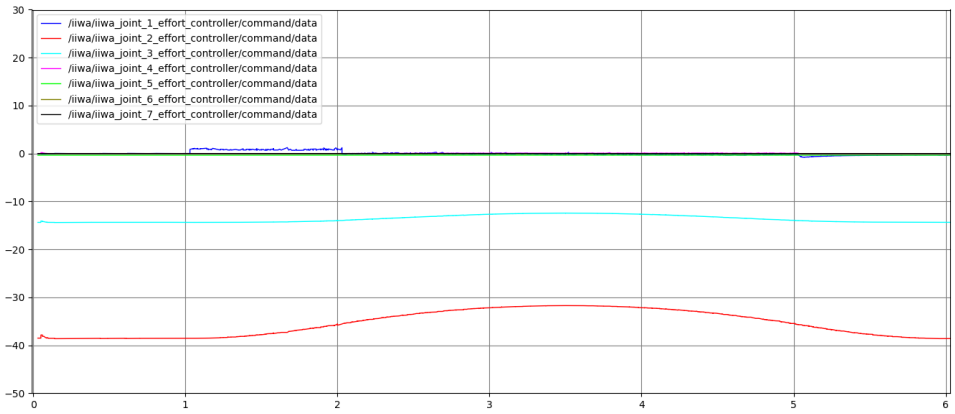

Figure 12: Linear trajectory Trapezoidal Velocity. Inverse dynamics joint space control

Figure 13: Cartesian error norm: Linear trajectory Trapezoidal Velocity. Inverse dynamics joint space control

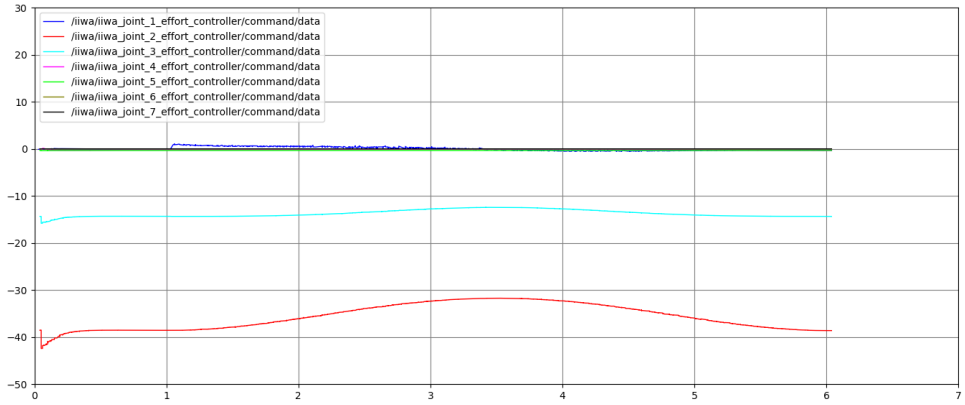

Figure 14: Linear trajectory Cubic Polynomial. Inverse dynamics joint space control

Figure 15: Cartesian error norm: Linear trajectory Cubic Polynomial. Inverse dynamics joint space control

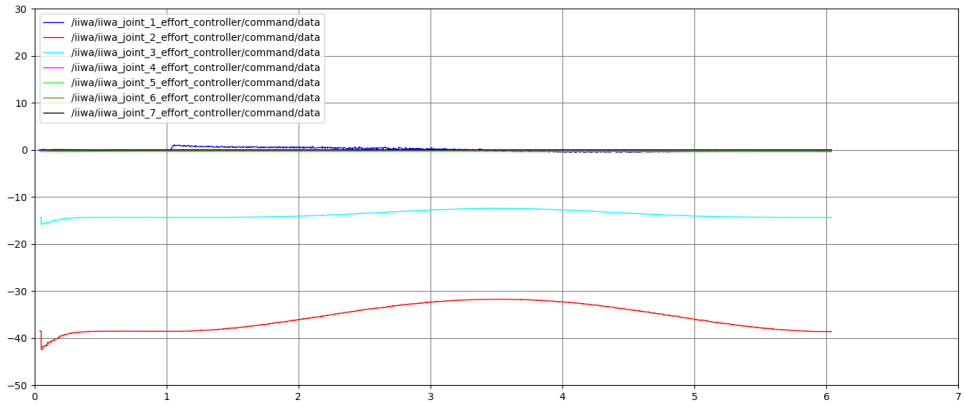

Figure 16: Circular trajectory Trapezoidal Velocity. Inverse dynamics joint space control

Figure 17: Cartesian error norm: Circular trajectory Trapezoidal Velocity. Inverse dynamics joint space control

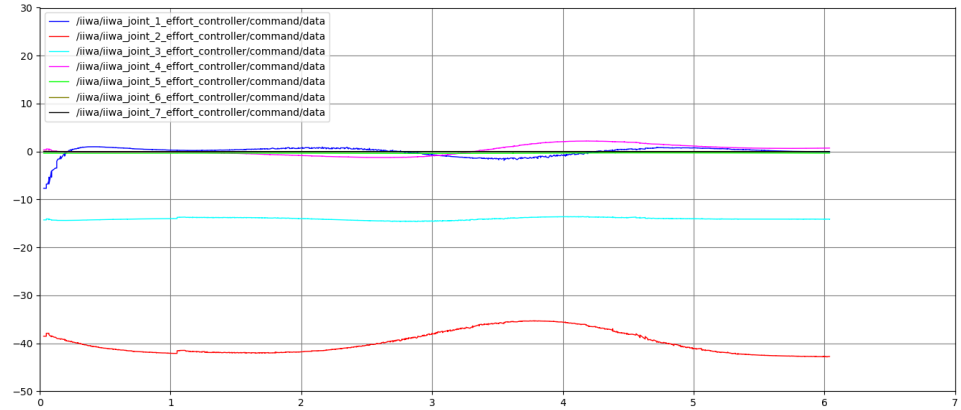

Figure 18: Circular trajectory Cubic Polynomial. Inverse dynamics joint space control

Figure 19: Cartesian error norm: Circular trajectory Cubic Polynomial. Inverse dynamics joint space control

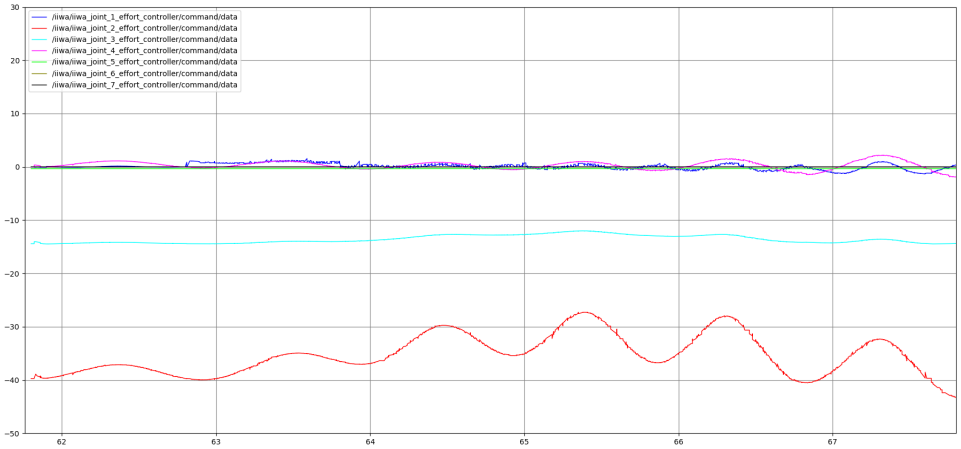

Figure 20: Linear trajectory Trapezoidal Velocity. Inverse dynamics operational space control

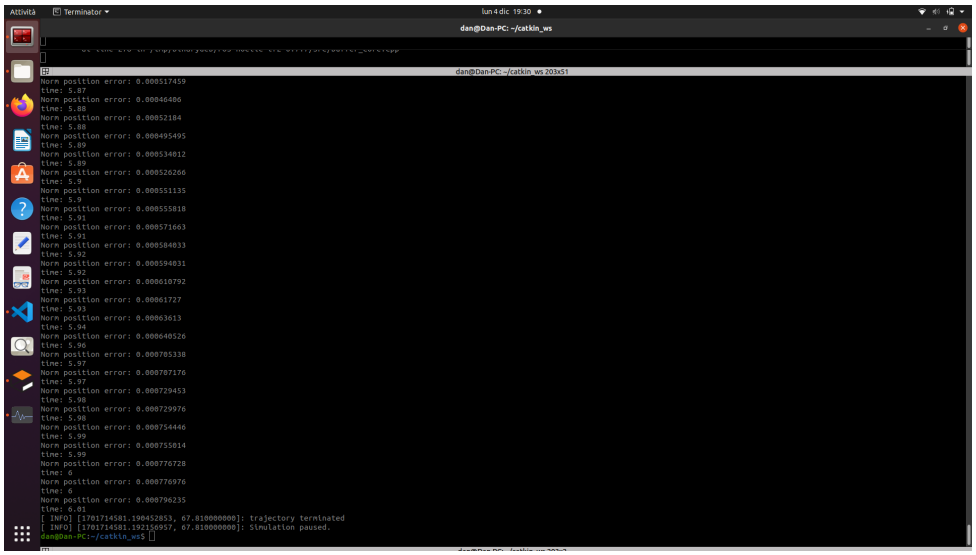

Figure 21: Cartesian error norm: Linear trajectory Trapezoidal Velocity. Inverse dynamics operational space control

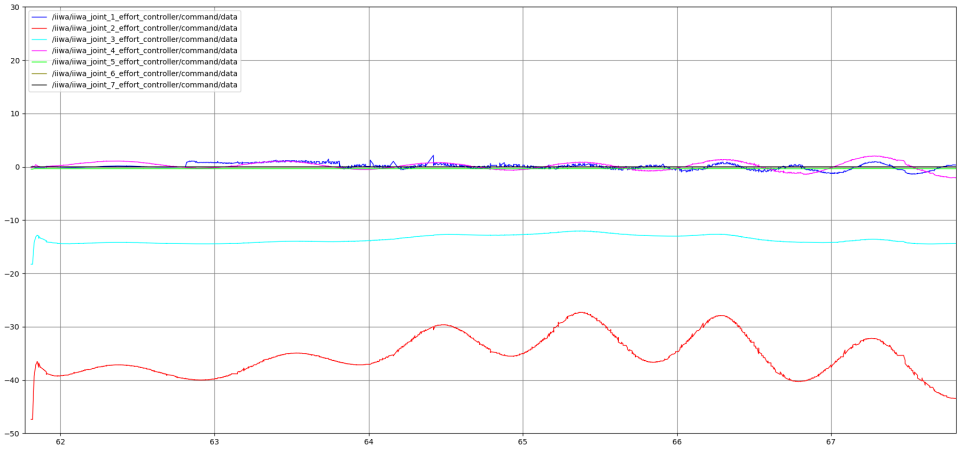

Figure 22: Linear trajectory Trapezoidal Velocity. Inverse dynamics operational space control assigning orientation error

Figure 23: Cartesian error norm: Linear trajectory Trapezoidal Velocity. Inverse dynamics operational space control assigning orientation error

As shown in the previous results, the iiwa robot follows the assigned trajectories while implementing the look-at-point vision control in all the analyzed cases.