Homework 3

Andrea Morghen, Maria Vittoria Cinquegrani, Valentina Giannotti

December 6, 2023

GitHub link: https://github.com/Andremorgh/RL_HW_03
GitHub link: https://github.com/MVCinquegrani/ROS_Homework3
GitHub link: https://github.com/ValentinaGiannotti/Homework3

1 Construct a Gazebo World and Detect a Circular Object using `opencv_ros` Package

1a Create new worlds

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.*

Taking inspiration from the `iiwa_stack` `models` we decided to create a new cylinder-shaped model named `circular_object`, the XML code is defined below:

```
<?xml version="1.0"?>
<sdf version="1.6">
  <model name="circular_object">
        <static>1</static>
    ...

      <visual name="front_visual">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.15</radius>
            <length>0.001</length>
          </cylinder>
        </geometry>
        <material>
          <ambient>1 0 0 1</ambient>
        </material>
      </visual>

    ...

      <visual name="rear_visual">
        <pose>0 0 -0.002 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.15</radius>
            <length>0.001</length>
          </cylinder>
        </geometry>
        <material>
```

```
        <ambient>1 0 0 1</ambient>
      </material>
    </visual>

    ...

      <kinematic>1</kinematic>

    ...

    <pose>1 -0.5 0.6 0 1.57 -3.14</pose>
  </model>

    ...
</sdf>
```

The <visual> tags define the visual aspects of the model. Since the cylinder is typified by two faces, there are two visual elements one for each of them: front_visual and rear_visual. We have specified and adjusted position, orientation, geometry, and color for each side so that the elements do not overlap. In particular the <ambient> tags specify the color of the object; in this case it is red.

To complete the model we created the model.config:

```
<?xml version="1.0"?>
<model>
  <name>circular_object</name>
  <version>1.0</version>
  <sdf version="1.6">model.sdf</sdf>

  <description>
    <p>circular_object</p>
  </description>
</model>
```

In order to create the new world with the iiwa and the object preloaded we launched the iiwa_gazebo_aruco.launch file. Then, directly into the Gazibo environment, we deleted the aruco marker and added the object as a static one at the right position, so that one of the cylinder sides faces the camera.

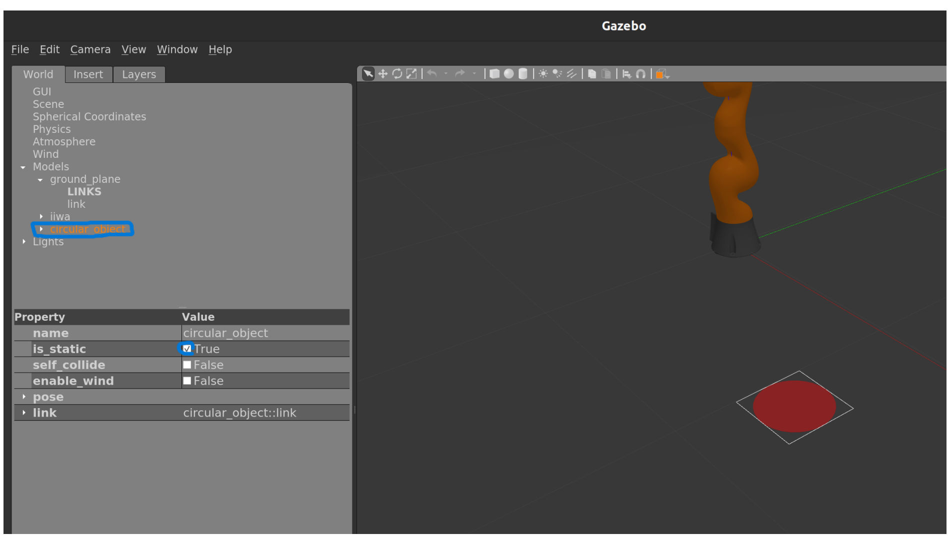


Figure 1: Circular object added to gazebo.

Figure 2 shows the final result in Gazibo.

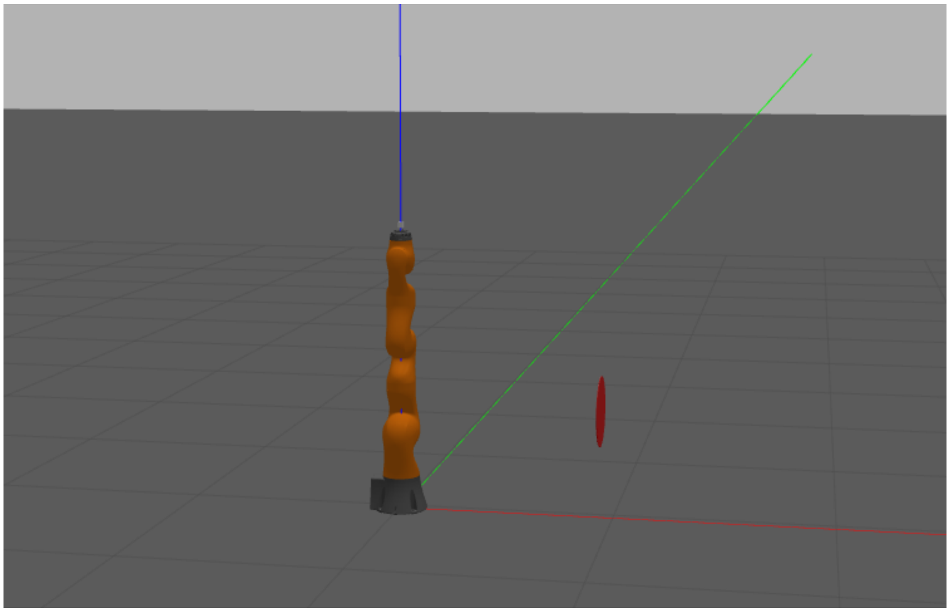


Figure 2: Final configuration.

After saving the new world into the `/iiwa gazebo/worlds/` folder as `iiwa_gazebo_circular_object.launch` we obtained the following result.

```
<sdf version='1.7'>
  ...

    <model name='circular_object'>
      <link name='link'>
        <visual name='front_visual'>
          <pose>0 0 0 0 0 0</pose>
          <geometry>
            <cylinder>
              <radius>0.15</radius>
              <length>0.001</length>
            </cylinder>
          </geometry>
          <material>
            <!-- colore desiderato -->
            <ambient>1 0 0 1</ambient>
          </material>
        </visual>
        <visual name='rear_visual'>
          <pose>0 0 -0.002 0 -0 0</pose>
          <geometry>
            <cylinder>
              <radius>0.15</radius>
              <length>0.001</length>
            </cylinder>
          </geometry>
          <material>
            <ambient>1 0 0 1</ambient>
          </material>
        </visual>

        ...
</sdf>
```

1b Create a new launch file

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`).

In order to create a new launch file useful for starting the Gazibo environment with the elements we added, we modified the lunch files of Homework 1. The launch file `launch/iiwa_gazebo_circular_object.launch` is shown below, with comments describing the steps taken.

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

    <arg name="hardware_interface" default="PositionJointInterface" />
    <arg name="robot_name" default="iiwa" />
    <arg name="model" default="iiwa14"/>
    <arg name="trajectory" default="false"/>

    <env name="GAZEBO_MODEL_PATH"
    value="$(find iiwa_gazebo)/models:$(optenv GAZEBO_MODEL_PATH)" />

    <!-- Loads the Gazebo world. -->
    <include file="$(find iiwa_gazebo)/launch
    /iiwa_world_circular_object.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)" />
        <arg name="robot_name" value="$(arg robot_name)" />
        <arg name="model" value="$(arg model)" />
    </include>

    <!-- Spawn controllers - it uses a JointTrajectoryController -->
    <group  ns="$(arg robot_name)" if="$(arg trajectory)">

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

    <!-- 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>

</launch>
```

We modified the position configuration of the iiwa so it can see the object, we used the Position_controllers like first homework. We set the q values:
(0.0); (1.57); (-1.57); (-1.57); (1.57); (-1.57); (+1.57);

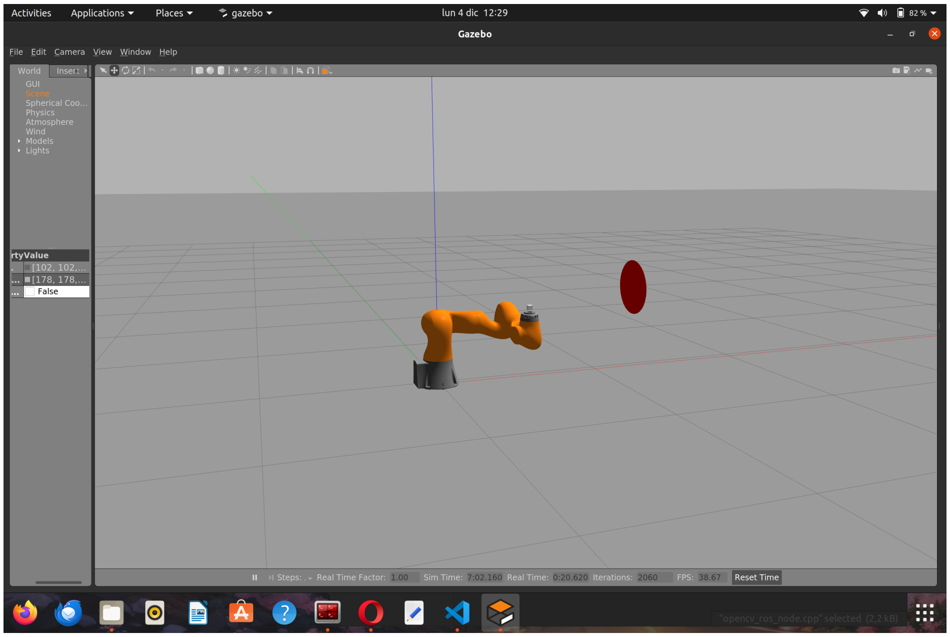


Figure 3: Recognized image

1c Detect the object

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 in the camera image, first of all we cloned the `opencv_ros` repository and we modified the `opencv_ros_node.cpp` by

- importing all the necessary libraries
- subscribing to input video the simulated camera image, that we can see typing `rqt_image_view` from terminal.
- using the OpenCV Blob Detection functions to identify the circular object.

The steps taken to perform the last point are better explained in the code comments below.

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d.hpp>
#include <iostream>

using namespace cv;
static const std::string OPENCV_WINDOW = "Image window";

class ImageConverter
{
    ...
public:
    ...
  {
    // Subscribe to input video feed as simulated camera image
    image_sub_ = it_.subscribe("/iiwa/camera1/image_raw", 1,&ImageConverter::
        imageCb, this);
    ...
  }
    ...

  void imageCb(const sensor_msgs::ImageConstPtr& msg)
  {
    cv_bridge::CvImagePtr cv_ptr;
    try     // to read image
      {
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
      }
    ...

    //Define the image variable using the OpenCV library data structure Mat
    Mat im;

    //Turns the image into greyscale and save it in im
    cvtColor (cv_ptr->image, im, COLOR_BGR2GRAY);

    //Set up the detector with default parameters.
    SimpleBlobDetector::Params params;
    Ptr<SimpleBlobDetector> detector = SimpleBlobDetector::create(params);

    //Detect blobs.
```

```
    std::vector<KeyPoint> keypoints;
    detector->detect(im, keypoints);
    //cv::imshow("Original Image", cv_ptr->image);

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

    //Show blobs
    imshow(OPENCV_WINDOW, im_with_keypoints );

    cv::waitKey(10);

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

We also modified the `camera.launch` file to execute the .cpp file just implemented:

```
<launch>
  <node name="opencv_ros_node" pkg="opencv_ros" type="opencv_ros_node"
  output="screen" >
  </node>
  <node name="image_view" pkg="image_view" type="image_view" respawn="false"
  output="screen">
    <remap from="image" to="/iiwa/camera1/image_raw"/>
    <param name="autosize" value="true" />
  </node>
</launch>
```

We execute the following command from the terminal to allow the camera detect the circular object. The image result is shown in Figure 4:

```
roslaunch \texttt{opencv\_ros} camera.launch
```

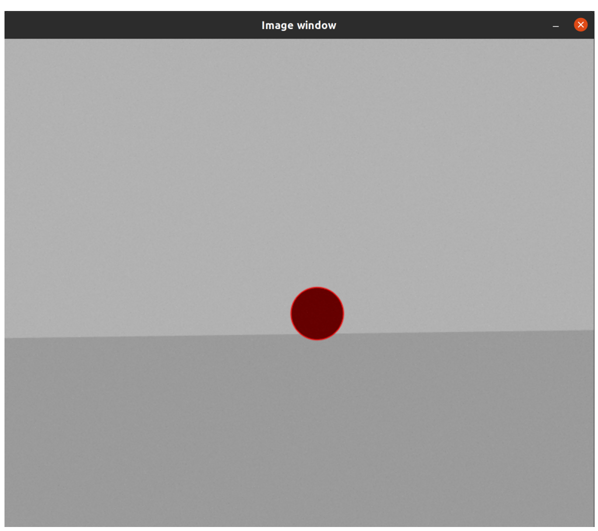


Figure 4: Recognized image

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

2a Enhancing KDL Robot Vision Control

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 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.*

Below is the `usb_cam_aruco.launch` file in which we modified the `camera` default argument:

```
<launch>

    <arg name="markerId"          default="201"/>
    <arg name="markerSize"        default="0.1"/>
    <arg name="camera"            default="iiwa/camera1"/> <!--MODIFIED-->
    <arg name="marker_frame"      default="aruco_marker_frame"/>
    <arg name="ref_frame"         default=""/>
    <arg name="corner_refinement" default="LINES" />

    <node pkg="aruco_ros" type="single" name="aruco_single">
        <remap from="/camera_info" to="/$(arg camera)/camera_info" />
        <remap from="/image" to="/$(arg camera)/image_raw" />
        <param name="image_is_rectified" value="True"/>
        <param name="marker_size"        value="$(arg markerSize)"/>
        <param name="marker_id"          value="$(arg markerId)"/>
        <param name="reference_frame"    value="$(arg ref_frame)"/>
        <!-- frame in which the marker pose will be refered -->
        <param name="camera_frame"
        value="stereo_gazebo_$(arg camera)_camera_optical_frame"/>
        <param name="marker_frame"       value="$(arg marker_frame)" />
        <param name="corner_refinement"  value="$(arg corner_refinement)" />
    </node>

</launch>
```

To implement a vision-based task that aligns the camera to the ArUco marker we start by modifying the `kdl_robot_vision_control` node. The main steps implemented are listed below, for a more detailed analysis see the code provided further on.

- First of all, we define the desired camera orientation and position, which must be kept constant while the marker moves, by choosing position and orientation offsets appropriately.
- Then we compute the error and we use it into the control law.

```
#include ...

// Global variables
...
const double KP = 15;
const double toRadians = M_PI/180.0;

// Functions
...

// Main
int main(int argc, char **argv)
{
    ...
```

```
while (ros::ok())
{

    if (robot_state_available && aruco_pose_available)
    {
        ...

        //------------------------------------------------------------//

        //// HomeWork 2.a ////
        // keep fixed position and orientation offset

        // DESIRED ORIENTATION COMPUTATION
        Eigen::Vector3d desired_orient(0.0,10.0, -10.0);
        Eigen::Matrix<double,3,3> Rot_des_ZYX = toEigen(KDL::Rotation::
            EulerZYX((-90.0+desired_orient[0])*toRadians, desired_orient
            [1]*toRadians,(-90.0+desired_orient[2])*toRadians));
            //Compute a desired ZYX rotation matrix using Euler angles

        // DESIRED POSITION COMPUTATION
        Eigen::Vector3d offset(0.0, 0.0, 0.5);
        Eigen::Matrix<double,3,1> p_cam_object = toEigen(robot.getEEFrame
            ().M)*toEigen(cam_T_object.p);
            // Compute the object position with respect to the camera

        Eigen::Matrix<double,3,1> p_bas_object = toEigen(robot.getEEFrame
            ().p) + p_cam_object;
            //Compute the object position with respect to the base
                reference (ee_position+offset)

        Eigen::Vector3d p_e_des_offset = p_bas_object- Rot_des_ZYX*offset;
            //Compute the desired pose of the end-effector (position and
                rotation)

        // DEBUG HW.2a
        // std::cout << robot.getEEFrame().M << std::endl;
        // std::cout << Rot_des_ZYX << std::endl;
        // std::cout << e_rot_ee_offset << std::endl;
        // std::cout << p_e_des_offset << std::endl;


        // ERROR COMPUTATION
        Eigen::Matrix<double,3,1> e_rot_ee_offset =
            computeOrientationError(Rot_des_ZYX, toEigen(robot.getEEFrame
            ().M));
            //Computes orientation error

        Eigen::Matrix<double,3,1> e_p_ee_offset = computeLinearError(
            p_e_des_offset, toEigen(robot.getEEFrame().p));
            //Computes position error

        Eigen::Matrix<double,6,1> x_tilde = Eigen::Matrix<double,6,1>::
            Zero(); //Define the error variable

        x_tilde << e_p_ee_offset,  e_rot_ee_offset[0], e_rot_ee_offset[1],
             e_rot_ee_offset[2];
            //Chain the position error and the orientation error to obtain
                 the total error to be used in the control law

        // HW.2a CONTROL LAW
        dqd.data = lambda*J_pinv*x_tilde+ 10*(Eigen::Matrix<double,7,7>::
```

```
                Identity() - J_pinv*J_cam.data)*(qdi - toEigen(jnt_pos));
            //------------------------------------------------------------//
            ...
            }
        ...
    }
    return 0;
}
```

The variable `desired_orient` is the chosen orientation offset from the object, The variable `offset` is instead the distance offset from the object

The commands that are executed in different windows of the terminal to run the code are as follows:

```
//launch the gazebo simulation with effort controllers
roslaunch iiwa_gazebo iiwa_gazebo_aruco.launch

//launch the aruco recognition cam
roslaunch aruco_ros usb_cam_aruco.launch

//open the rqt_image_view node to see the output of the recognition
rosrun rqt_image_view rqt_image_view

//run the vision control effort node
rosrun kdl_ros_control kdl_robot_vision_control src/iiwa_stack/
    iiwa_description/urdf/iiwa14.urdf
```

We plotted joins velocities and we obtained the following result:

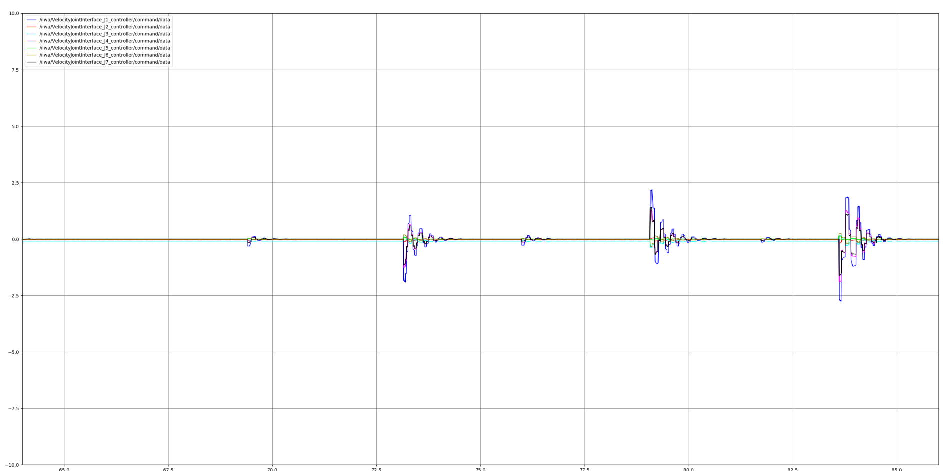


Figure 5: velocity

2b Improved look-at-point algorithm

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

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

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

$$L(s) = \left[-\frac{1}{||{}^{c}P_o||}(I - ss^T) S(s) \right], \quad R \in \mathbb{R}^{3\times6} \tag{1}$$

with

$$R = \begin{bmatrix} R_c & 0 \\ 0 & R_c \end{bmatrix} \tag{2}$$

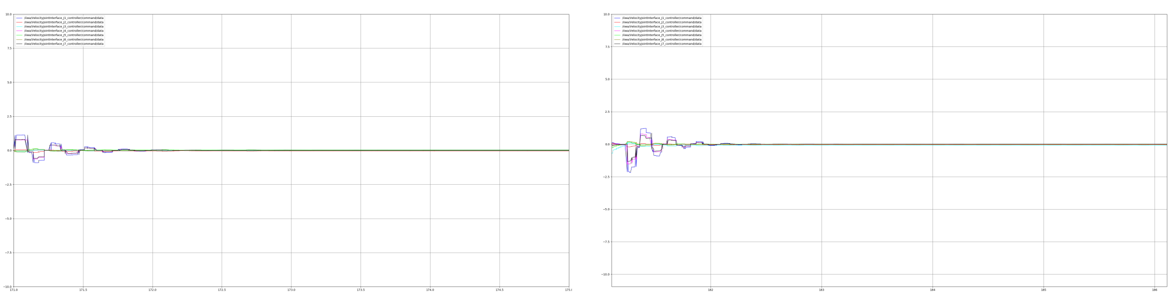
where $S(\cdot)$ is the skew-symmetric operator, R_c is the current camera rotation matrix. Implement the following control law $\dot{q} = k(LJ)^{\dagger} s_d + N\dot{q}_0$, where s_d is a desired value for s, e.g., $s_d = [0, 0, 1]$, and $N = (I - (LJ)^{\dagger}LJ)$ being the matrix spanning the null space of the LJ matrix. Verify that for a chosen $\dot{q}_0$, the s measure does not change by plotting joint velocities and the s components.

To implement the required control law, we computed the N and L matrices whit the provided equation. Then we implemented the control low whit the required shape. Below is the file `kdl_robot_vision_control.cpp` with the edits made:

```
#include ...

// Global variables
...

// Functions
...

// Main
int main(int argc, char **argv)
{
    ...
    while (ros::ok())
    {
        if (robot_state_available && aruco_pose_available)
        {
            ...
            //------------------------------------------------------------//

            //// HomeWork 2.b ////

            // L Matrix(3x6) computation
            Eigen::Matrix<double,3,1> c_Po = toEigen(cam_T_object.p);
            Eigen::Matrix<double,3,1> s = c_Po/c_Po.norm();
            Eigen::Matrix<double,3,3> R_c = 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,6,6> R_c_big = Eigen::Matrix<double,6,6>::
                Zero();
            R_c_big.block(0,0,3,3) = R_c;
            R_c_big.block(3,3,3,3) = R_c;
             Eigen::Matrix<double,3,6> L = Eigen::Matrix<double,3,6>::Zero();
            L.block(0,0,3,3) = L_block;
            L.block(0,3,3,3) = skew(s);
            L = L*(R_c_big.transpose());

            // N matrix(7x7) computation
            Eigen::MatrixXd LJ = L*toEigen(J_cam);
```

```
            Eigen::MatrixXd LJ_pinv = LJ.completeOrthogonalDecomposition().
                pseudoInverse();
            Eigen::MatrixXd N = Eigen::Matrix<double,7,7>::Identity() - (
                LJ_pinv*LJ);

            // DEBUG HW.2b
            // std::cout << c_Po << std::endl;
            // std::cout << R_c << std::endl << std::endl;
            // std::cout << s.transpose() << std::endl;
            // std::cout << L << std::endl;
            // std::cout << LJ_pinv << std::endl;

            // HW.2b control law
            dqd.data = 20*LJ_pinv*Eigen::Vector3d(0,0,1) + 10*N*(qdi - toEigen
                (jnt_pos));

            //------------------------------------------------------------//

            ...

            }

        ...

    }
    return 0;
}
```

An improved look-at-point control law has been implemented in this section. The commands used to run this part of the code are similar to those used in the previous point. To verify that for a chosen $\dot{q}0$; the s measure does not change we compared joint velocities plot and the s component for the case of presence and absence of the null in the control law.

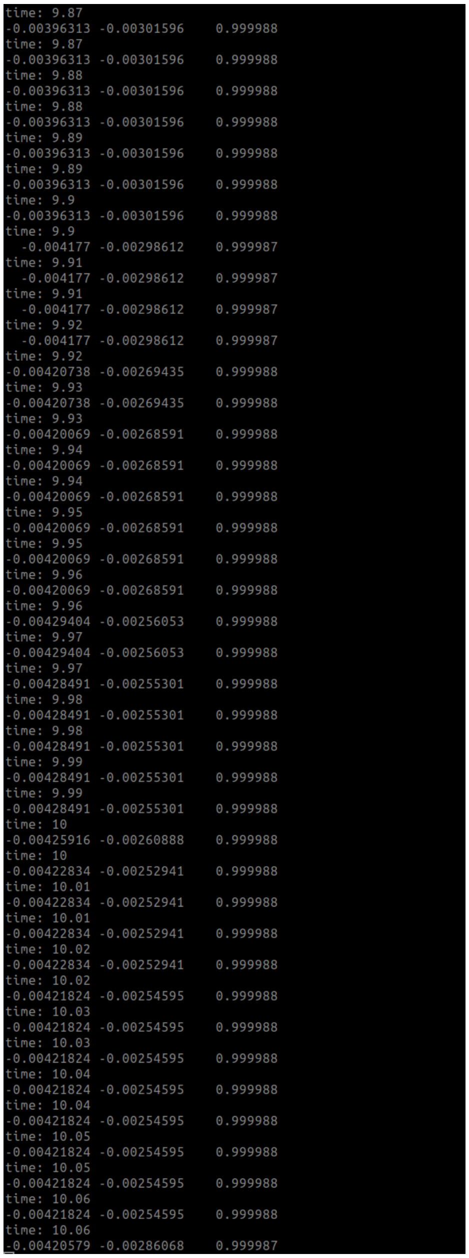


(a) Velocity without null

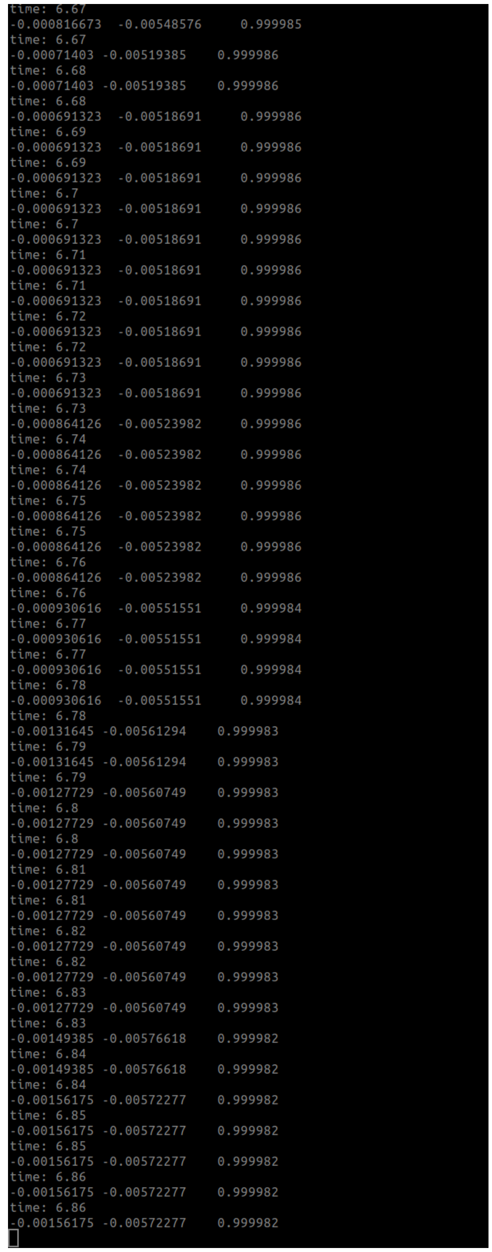
(b) Velocity with null

Figure 6: Simulation of the velocity

Looking at Figures 9 and 8 it is easy to see that for the chosen $\dot{q}_0$, the s measure and joint velocities do not change considerably.



(a) s vector without null



(b) s vector with null

Figure 7: Simulation of the s

2c Develop a Dynamic Version of the Vision-Based Controller

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.

First of all we define the linear position trajectory using curvilinear abscissa:

$$p(s) = p_i + s(p_f - p_i) \tag{3}$$

Here the derivatives of the equation 3:

$$\dot{p}(s) = \dot{s}(p_f - p_i) \tag{4}$$
$$\ddot{p}(s) = \ddot{s}(p_f - p_i) \tag{5}$$

Hence the `KDLPlanner::compute_trajectory` function in the `kdl_planner.cpp` file calculates the linear trajectory based on the provided time as we did in the Homework 2. Using a cubic polynomial curvilinear abscissa, it computes the position, velocity, and acceleration of the linear trajectory. The calculated values are then assigned to the `trajectory_point` structure and returned.

Taking inspiration from the `iiwa_gazebo_aruco.launch` file we created in `iiwaa_gazebo` a new launch file named `iiwa_gazebo_aruco_effort.launch` shown below:

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

   <arg name="hardware_interface"
   default="EffortJointInterface" /> <!-- modified -->
    <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 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
                   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"/> <!-- modified -->
            <arg name="robot_name" value="$(arg robot_name)" />
            <arg name="model" value="$(arg model)" />
        </include>

    </group>


</launch>
```

Hence we changed the names of the `hardware_interface` to `EffortJointInterface` and we also changed the names of all the controller nodes (`iiwa_joint_1_effort_controller`, etc.) as they had different names in the `.yaml` file.

Then we modified the `kdl_robot_vision_control` file, in `kdl_robot` repository, and we save it as a new file called `kdl_robot_vision_control_effort`, shown below.

```
#include ...

// Global variables
...

// Functions
...

// Main
int main(int argc, char **argv)
{
    ...

    // Subscribers
    ...
    // Publishers
    //names have been changed according to the file yalm
    ros::Publisher joint1_effort_pub = n.advertise<std_msgs::Float64>("/iiwa/
       iiwa_joint_1_effort_controller/command", 1);
    ...

    ros::Publisher joint7_effort_pub = n.advertise<std_msgs::Float64>("/iiwa/
       iiwa_joint_7_effort_controller/command", 1);
    ros::Publisher error_pub = n.advertise<std_msgs::Float64>("/iiwa/
       traj_error", 1);

    ...

    // Messages
    std_msgs::Float64 tau1_msg, tau2_msg, tau3_msg, tau4_msg, tau5_msg,
       tau6_msg, tau7_msg, error_msg;

    ...
```

```
    // Torques
    Eigen::VectorXd tau;
    tau.resize(robot.getNrJnts());

    ...

  //------------------------------------------------------------------//

    ////Init planner and trajectory parameters////

    // EE's trajectory initial position
    KDL::Frame init_cart_pose = robot.getEEFrame();
    Eigen::Vector3d init_position(init_cart_pose.p.data);

    // EE trajectory end position
    Eigen::Vector3d end_position;
    end_position << init_cart_pose.p.x(), init_cart_pose.p.y()+0.50,
        init_cart_pose.p.z();

    // trajectory parameters
    double traj_duration = 1.5, acc_duration = 0.5, t = 0.0, init_time_slot =
        1.0, traj_radius = 0.15, error = 0.0;
    // traj_choice=1->rectilinear traj
    // traj_choice=2->circular traj
    int traj_choice=2;

    // Plan trajectory
    KDLPlanner planner(traj_duration, traj_radius, acc_duration, init_position
        , end_position);

    // Retrieve the first trajectory point
    trajectory_point p = planner.compute_trajectory(t,traj_choice);
    //------------------------------------------------------------------//

    // Init controller
    KDLController controller_(robot);

    // Retrieve initial simulation time
    ros::Time begin = ros::Time::now();
    ROS_INFO_STREAM_ONCE("Starting control loop ...");

    // Init trajectory
    KDL::Frame des_pose = KDL::Frame::Identity(); KDL::Twist des_cart_vel =
        KDL::Twist::Zero(), des_cart_acc = KDL::Twist::Zero();
    des_pose.M = robot.getEEFrame().M;

    while (ros::ok())
    {
        if (robot_state_available && aruco_pose_available){

            // Update robot
            robot.update(jnt_pos, jnt_vel);

            // Update time
            t = (ros::Time::now()-begin).toSec();
            std::cout << "time: " << t << std::endl;

        //------------------------------------------------------------------//

            ////Extract desired pose velocity and acceleration////
```

```
    des_cart_vel = KDL::Twist::Zero();
    des_cart_acc = KDL::Twist::Zero();

    if (t <= init_time_slot){
        p = planner.compute_trajectory(0.0,traj_choice);
    }
    else if(t > init_time_slot && t <= traj_duration + init_time_slot)
        {
        p = planner.compute_trajectory(t-init_time_slot,traj_choice);
        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]);

//-----------------------------------------------------------------//

    // 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);

    //// select the type of control ////

    // Joint space inverse dynamics control
    qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3], jnt_pos
        [4], jnt_pos[5], jnt_pos[6];
    des_pose.M=robot.getEEFrame().M*Re*ee_T_cam.M.Inverse();
    robot.getInverseKinematics(des_pose, des_cart_vel, des_cart_acc,
        qd, dqd, ddqd);
    double Kp = 50, Kd = 2*sqrt(Kp);
    tau = controller_.idCntr(qd, dqd, ddqd, Kp, Kd, error);

    // Cartesian space inverse dynamics control
    des_pose.M=robot.getEEFrame().M*Re;
    double Kp = 100;
    double Ko = 100;
    tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc, Kp,
         Ko, 2*sqrt(Kp), 2*sqrt(Ko), error);

    //// select the type of control ////

    // // DEBUG
    // std::cout << "des_pose.M: "<< des_pose.M << std::endl;
    // std::cout << "Re: " << std::endl << Re << std::endl;
```

```
            // std::cout << "aruco_pos_n:"<<std::endl<<aruco_pos_n<<std::endl;
            // std::cout << "tau: "<<std::endl<< tau.transpose() << std::endl;
            // std::cout << "des_pose_q: "<<std::endl<<des_pose.q<< std::endl;
            // std::cout << "current_pose: "<< std::endl << robot.getEEFrame()
                << std::endl;
        }
        else{
            tau = Eigen::Matrix<double,7,1>::Zero();
        }
        // Set torques
        tau1_msg.data = tau[0];
        ...
        tau7_msg.data = tau[6];
        error_msg.data = error;

        // Publish
        joint1_effort_pub.publish(tau1_msg);
        ...
        joint7_effort_pub.publish(tau7_msg);
        error_pub.publish(error_msg);

        ros::spinOnce();
        loop_rate.sleep();
    }
    if(pauseGazebo.call(pauseSrv))
        ROS_INFO("Simulation paused.");
    else
        ROS_INFO("Failed to pause simulation.");

    return 0;
}
```

We customized the given code to meet our specific requirements, incorporating additional lines of controller code for both joint space and operational space, as computed in the preceding homework. Notably, significant emphasis was placed on the implementation of joint space inverse dynamics control. Indeed, in order to be able to use the kinematic inversion function previously used without the camera, the desired position of the camera with respect to the end-effector was specified.

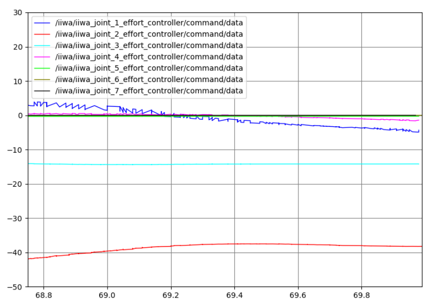
The cmake file was subsequently modified to take account of the kdl_robot_vision_control_effort.cpp file as follows:

```
cmake_minimum_required(VERSION 2.8.3)
project(kdl_ros_control)
...
###########
## Build ##
###########
...
add_executable(kdl_robot_vision_control_effort src/
    kdl_robot_vision_control_effort.cpp
src/kdl_robot.cpp
src/kdl_control.cpp
src/kdl_planner.cpp
)
...
target_link_libraries(kdl_robot_vision_control_effort
   ${catkin_LIBRARIES}
)
...
```

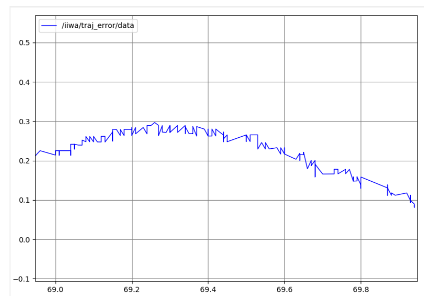
The commands that are executed in different windows of the terminal to run the code are as follows:

```
//launch the gazebo simulation with effort controllers
roslaunch iiwa_gazebo iiwa_gazebo_aruco_effort.launch

//launch the aruco recognition cam
roslaunch aruco_ros usb_cam_aruco.launch

//open the rqt_image_view node to see the output of the recognition
rosrun rqt_image_view rqt_image_view

//run the vision control effort node
rosrun kdl_ros_control kdl_robot_vision_control_effort src/iiwa_stack/
    iiwa_description/urdf/iiwa14.urdf
```

Then the results of the Joint space inverse dynamics control and the Cartesian space inverse dynamics control are plotted in Figure 8 and Figure 9 respectively:

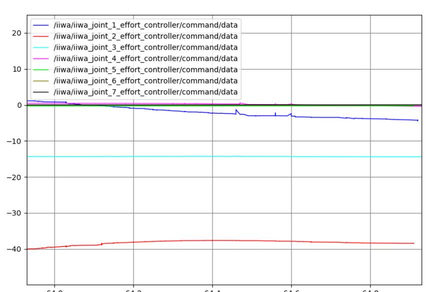


(a) Torque in the joint space control

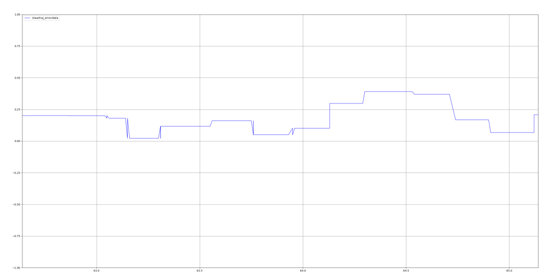


(b) orientation error in the joint space control

Figure 8: Simulation Joint space inverse dynamics



(a) Torque in the Cartesian space control



(b) orientation error in the cartesian space control

Figure 9: Simulation Cartesian space inverse dynamics