Robotics Lab: Homework 3

Ludovica Ruggiero P38000207

repository:	ludruggiero/RL/tree/main/homework3
videos:	output/videos.md
collaborators:	Luigi Catello, Stefano Covone, Cristiana Punzo

Contents

1 Detection of a circular object using OpenCV **2**
1.a Circular object in Gazebo world 2
1.b Launch file creation 4
1.c Object Detection with OpenCV 5

2 Vision-Based Control Task **6**
2.a ArUco Marker tracking 6
2.b Improved look-at-point algorithm 9
2.c Dynamic controller with look-at-point task 12

The goal of this assignment is to implement a vision-based controller for a 7-degrees-of-freedom robotic manipulator arm within the Gazebo environment. In Section 1, the `opencv_ros` library has been employed to detect a circular object spawned in the Gazebo environment. In Section 2, the provided look-at-point vision-based control example has been modified to first follow the ArUco marker, and then, in subsection 2.c, to ensure the camera focuses on the marker while the robot executes a linear or circular trajectory using a dynamic controller.

1 Detection of a circular object using OpenCV

1.a Circular object in Gazebo world

Firstly, a `circular_object` model has to be defined in the Gazebo environment, creating a new model in the `iiwa_gazebo/models` folder. Starting from the `aruco_marker` model, in the `model.sdf` file the proprieties of the object, such as the material, dimension and pose were defined as follows:

```
<?xml version="1.0"?>
<sdf version="1.6">
  <model name="circular_object">
    <static>true</static>
    <link name="link">
      <visual name="front_visual">
        <pose>0 0 0.005 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.15</radius>
            <length>0.01</length>
          </cylinder>
        </geometry>
        <material>
          <script>
            <uri>model://circular_object/materials/scripts</uri>
            <uri>model://circular_object/materials/textures</uri>
            <name>CircularObject</name>
          </script>
        </material>
        ...
    </link>
  </model>
</sdf>
```

The main difference with respect to the square `aruco_marker` model is about the geometry of the object, which now is a cylinder with a radius of 15 cm.

The name of the object was then defined in the `model.config` file. The material `CircularObject` is defined within the `circular.material` file:

```
material CircularObject
{
```

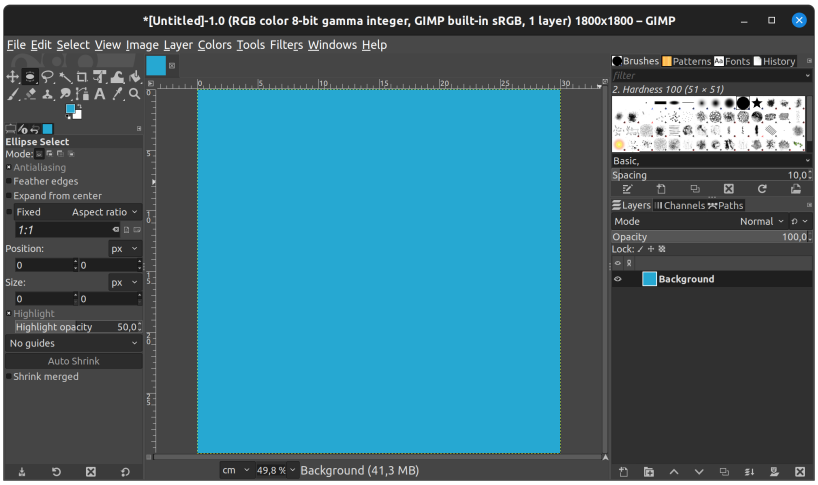


Figure 1.1: GIMP texture creation

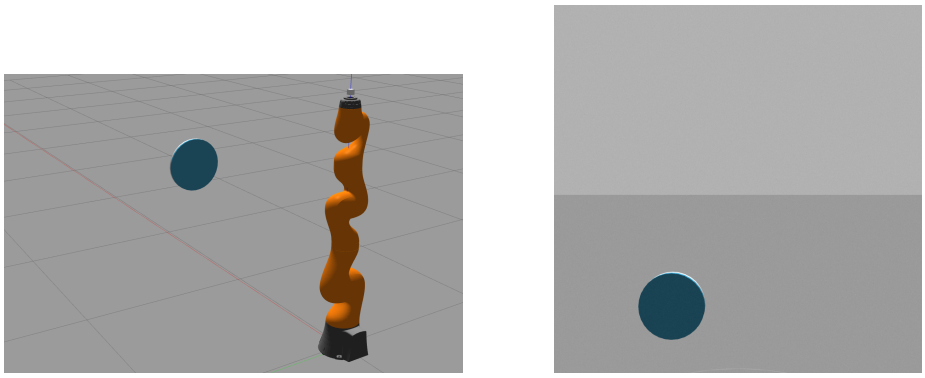
```
        ...
        texture circular.png
        ...
    }
```

The texture `circular.png` is a PNG file created via GIMP, where a square of dimensions 30×30 cm, filled with a solid color, is obtained, as shown in Figure 1.1.

This object was then imported in Gazebo and the new world saved as `iiwa_circular.world` in the `worlds` folder. In Figure 1.2 the obtained environment and the camera input is shown.

Another possible object was created, respecting the required specification: a sphere with radius 15 cm. It was directly defined as an `SDF` file, including a `sphere` component in its geometry:

```
<?xml version="1.0"?>
<sdf version="1.6">
  <model name="spherical_object">
    <link name="link">
      <pose>1 -0.5 0.6 0 0 0</pose>
      <gravity>0</gravity>
      <inertial>...</inertial>
      <visual name='visual'>
        <pose>0.00005 0 0 0 0 0</pose>
        <geometry>
          <sphere>
            <radius>0.15</radius>
          </sphere>
        </geometry>
        ...
```



(a) Gazebo world (b) Camera view

Figure 1.2: Gazebo world with Iiwa robot and circular object

For the material of the sphere, not needing a precise texture, the default `Gazebo/Blue` was chosen.

```
        ...
         <material>
          <lighting>1</lighting>
          <script>
            <uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Blue</name>
          </script>
          ...
        </material>
        <transparency>0</transparency>
        <cast_shadows>1</cast_shadows>
      </visual>
      ...
    </link>
  </model>
</sdf>
```

1.b Launch file creation

The previously created world needed two files to be correctly opened in Gazebo: `iiwa_world_circular_object.launch` and `iiwa_gazebo_circular_object.launch`.

The file `iiwa_world_circular_object.launch` for loading the previously saved world in Gazebo has a code similar to `iiwa_world_aruco.launch`, but it was modified to load the new world. In particular the argument `world_name` was set to `iiwa_circular.world`.

```
<arg name="world_name" value="$(find iiwa_gazebo)/worlds/iiwa_circular.world"/>
```

The `iiwa_gazebo_circular_object.launch` first calls the previously defined launch file, then adds the `PositionJointInterfaces` to the joints. To this end, the following lines of `iiwa_gazebo_aruco.launch` have been modified as follows:

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

1.c Object Detection with OpenCV

In order to detect the round object, `OpenCV` functions have been used. To this end, the `opencv_ros_node.cpp` file was appropriately modified. To use the `SimpleBlobDetector`, the following library was included:

```
#include "opencv2/opencv.hpp"
```

In the `ImageConverter` class, the constructor was changed for the subscriber to listen to the correct topic:

```
image_sub_ = it_.subscribe("/iiwa/camera1/image_raw", 1,
     &ImageConverter::imageCb, this);
```

The `imageCb()` callback function is divided in five parts. First, the image is converted in suitable format for OpenCV, using `BGR8` encoding.

```
cv_bridge::CvImagePtr cv_ptr;
try {
  cv_ptr = cv_bridge::toCvCopy
       (msg, sensor_msgs::image_encodings::BGR8);
} catch ...
```

Since the Blob Detector only works with grayscale images, as a second step the image is reconverted.

```
cv::Mat im;
cv::cvtColor(cv_ptr->image, im, cv::COLOR_BGR2GRAY);
```

Then the `SimpleBlobDetector` is instantiated and its parameters tuned. The chosen values are the result of a trial-and-error process.

```
// Change thresholds
params.minThreshold = 1;
params.maxThreshold = 250;
params.thresholdStep = 1;
// Filter by Color
```

```
params.filterByColor = true;
params.blobColor = 0;
params.minRepeatability = 4;
params.minDistBetweenBlobs = 1;
// Filter by Area.
params.filterByArea = true;
params.minArea = 100;
params.maxArea = 10000000;
// Filter by Circularity
params.filterByCircularity = true;
params.minCircularity = 0.7;
// Filter by Convexity
params.filterByConvexity = true;
params.minConvexity = 0.87;
// Filter by Inertia
params.filterByInertia = true;
params.minInertiaRatio = 0.4;
...
cv::Ptr<cv::SimpleBlobDetector> detector =
       cv::SimpleBlobDetector::create(params);
```

Then, the detector is started and the detected blobs collected in a specific vector.

```
detector->detect(im, keypoints);
```

Lastly, a red circle is drawn to outline the identified object.

```
cv::drawKeypoints(cv_ptr->image, keypoints,
       im_with_keypoints, cv::Scalar(0,0,255),
       cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS );
```

The detected circular object was displayed in the OpenCV window, as shown in Figure 1.3, and the image was then converted into a ROS message to be published on a new topic.

```
// Update GUI Window
cv::imshow(OPENCV_WINDOW, im_with_keypoints);
cv::waitKey(1);

// Output modified video stream
sensor_msgs::ImagePtr output_image_msg =
       cv_bridge::CvImage(std_msgs::Header(), "bgr8",
       im_with_keypoints).toImageMsg();
image_pub_.publish(output_image_msg);
```

2 Vision-Based Control Task

2.a ArUco Marker tracking

The goal was to create a controller enabling the robot to track the marker's trajectory while preserving a defined position and orientation offset between the ArUco marker frame and the end-effector camera frame.

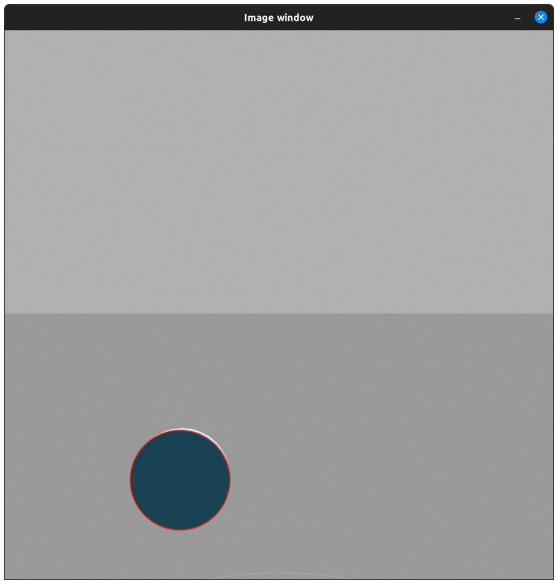


Figure 1.3: Detection algorithm output

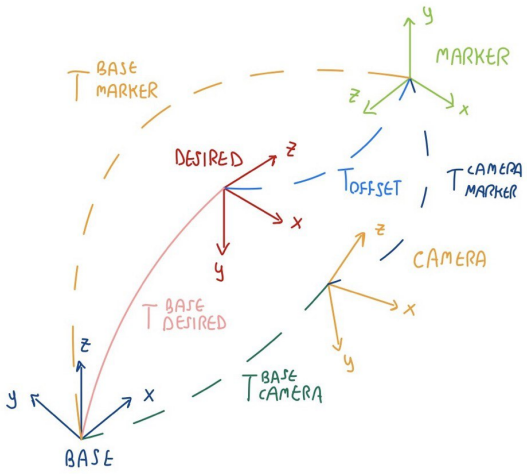


Figure 2.1: Definition of orientation error

At first, the transformation matrix describing the desired offset from the marker has been computed. It aligns the camera to the marker as shown in Figure 2.1, keeping a constant distance of 0.3 m from it:

$$T_{\text{OFFSET}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0.3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

To define the desired pose of the camera, the transformation matrix from the base frame to the desired camera frame is obtained as:

$$T_{\text{D}} = T_{\text{C}} T_{\text{M}}^{\text{C}} T_{\text{OFF}} \tag{2}$$

where T_{C} expresses the pose of the camera with respect to the base frame, T_{M}^{C} represents the pose of the marker with respect to the camera frame and T_{OFF} is the offset matrix. The above formula is defined in the following lines of code:

```
// 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]));
KDL::Frame base_T_object = robot.getEEFrame()*cam_T_object;

// compute offset transformation
KDL::Rotation R_off = KDL::Rotation::RotX(3.14);
KDL::Vector P_off(0,0,0.3);
KDL::Frame T_offset(R_off,P_off);
KDL::Frame T_desired = base_T_object*T_offset;
```

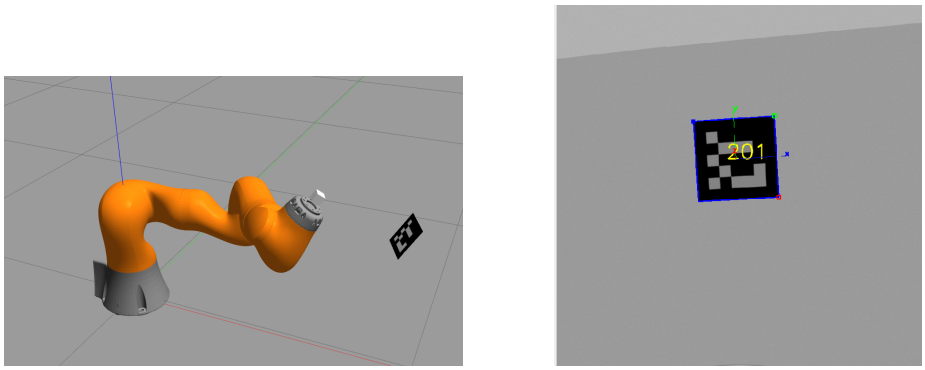
Then, the controller needed to evaluate the following control law:

$$\dot{q} = 1.5 J^{\dagger} \tilde{x} + 10(I - J^{\dagger} J) q_o \tag{3}$$

where $\tilde{x}$ represents the error between the desired position and orientation and the actual one. The second term belongs to the null space of J and, therefore, is only responsible for some internal motion that does not affect the end effector pose. The implemented code is:

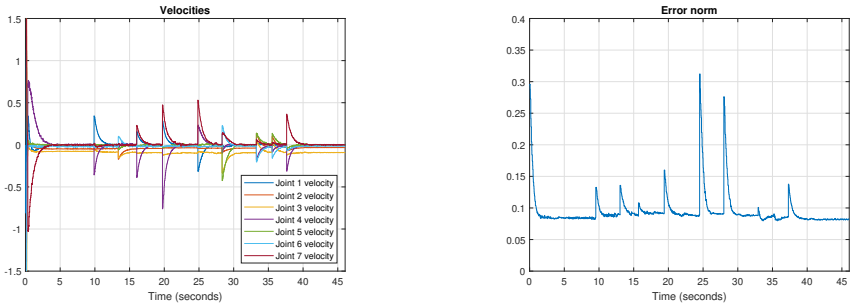
```
Eigen::Matrix<double, 3, 1> e_o_n =
       computeOrientationError(toEigen(T_desired.M),
       toEigen(robot.getEEFrame().M));
Eigen::Matrix<double, 3, 1> e_p =
       computeLinearError(toEigen(T_desired.p),
       toEigen(robot.getEEFrame().p));
Eigen::Matrix<double,6,1> x_tilde; x_tilde << e_p, e_o_n;

// resolved velocity control law
Eigen::MatrixXd J_pinv =
       J_cam.data.completeOrthogonalDecomposition()
```



(a) Gazebo world (b) Camera view

Figure 2.2: The robot is now able to track the marker pose



(a) Velocity commands (b) Norm of tracking error

Figure 2.3: The joint velocity commands and the norm of the tracking error relative to the tracking simulation

```
                                        .pseudoInverse();
dqd.data = 1.5 * J_pinv * x_tilde +
           10 * (Eigen::Matrix<double, 7, 7>::Identity()
                - J_pinv * J_cam.data) * (qdi - toEigen(jnt_pos));
```

The robot is now able to track the ArUco marker both in position and orientation, as shown in Figure 2.2. To further prove the tracking performance of the manipulator, a screen recording of the Gazebo simulation is available here. The simulation results are shown in Figure 2.3 in terms of joint velocities commands and norm of the tracking error.

2.b Improved look-at-point algorithm

The control law

$$\dot{q} = k(LJ)^{\dagger} s_d + N\dot{q}_0 \tag{4}$$

where $s_d = [0 \quad 0 \quad 1]$, implements an improved look-at-point algorithm [1]. To obtain this control law, we started by building the matrix

$$L(s) = \left[-\frac{1}{\|^c P_o\|}(I - ss^T) \quad S(s) \right] R \in \mathbb{R}^{3\times 6}$$

which maps linear/angular velocities of the camera to changes in s; where $S(s)$ is the skew-simmetric operator, and R is a block matrix defined as:

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

The matrix L was defined by two 3x3 blocks:

```
Eigen::Matrix<double,3,3> R_c = toEigen(robot.getEEFrame().M);
Eigen::Matrix<double,3,6> L;
L.block(0,0,3,3) = (-1/cam_T_object.p.Norm()) *
        (Eigen::Matrix<double,3,3>::Identity()
        - (aruco_pos_n * aruco_pos_n.transpose())) *
        R_c.transpose();
L.block(0,3,3,3) = skew(aruco_pos_n) * R_c.transpose();
```

Once obtained, it was useful to calculate the pseudo-inverse matrix $LJ^{\dagger}$.

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

The matrix $N = (I - (LJ)^{\dagger}LJ)$ is the null projector of the LJ matrix, computed as:

```
Eigen::MatrixXd Null_projector =
        Eigen::Matrix<double,7,7>::Identity() -
        (LJ_pinv * (L * J_cam.data));
```

Finally, the control law defined in the equation (4) was computed as:

```
dqd.data = 2 * LJ_pinv * sd +
        Null_projector * (qdi - toEigen(jnt_pos));
```

This control law resulted effective in controlling the manipulator to look at the ArUco marker. Table 1 shows the simulation results in terms of velocity commands and components of **s**. Moreover, a video of the performance of the improved controller is available here.

Null space projector from the publication

The paper [1] defines a Null Space Projector which allows the human operator to impose 4 different motion of the manipulator through the pseudo-velocities $\lambda = \begin{bmatrix} \lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 \end{bmatrix}^T$:

- a translation along s with a velocity proportional to λ_1;
- a rotation about s with a velocity proportional to λ_2;

- a motion on the surface of a sphere centered in the marker, whose velocity along x is proportional to λ_3 while the one along y is proportional to λ_4.

The null space projector N has the expression:

$$N = \begin{bmatrix} \mathbf{s} & 0 & -S(\mathbf{s})\mathbf{e}_y & S(\mathbf{s})\mathbf{e}_x \\ \mathbf{0} & \mathbf{s} & -(I - \mathbf{s}\mathbf{s}^T)\mathbf{e}_y/d & (I - \mathbf{s}\mathbf{s}^T)\mathbf{e}_x/d \end{bmatrix} \in \mathbb{R}^{6\times 4} \tag{5}$$

Since the control law in [1] is an operational space control law, to map the twist into joint velocities the overall control law was defined as:

$$\dot{q} = k(LJ)^{\dagger} s_d + J^{\dagger} N \lambda \tag{6}$$

However this control law result inefficient in simulation, since it constraints only 6 DOFs of the manipulator, resulting in an internal motion causing the camera roll to drift. This is probably due to the fact that the robot used in [1] is a 6 DOFs manipulator, whereas the Iiwa in our simulations has 7 DOFs.

To constraint the seventh DOF, the null term from (4) was added, resulting in the overall control law:

$$\dot{q} = k(LJ)^{\dagger} s_d + J^{\dagger} N \lambda + [I - (LJ)^{\dagger} LJ](q_0 - q) \tag{7}$$

To impose a motion to the manipulator, a period pseudo-velocity $\lambda_3 = 0.1\sin(t)$ was assigned. The following code implements the control law discussed above:

```
Eigen::Matrix<double,6,1> n1,n2,n3,n4;
Eigen::Vector3d ex, ey, s;
s = aruco_pos_n;
ex << 1,0,0;
ey << 0,1,0;
double d = cam_T_object.p.Norm();
Eigen::Matrix<double,3,3> Ps =
       Eigen::Matrix<double,3,3>::Identity() - (s * s.transpose());

Eigen::Matrix<double,6,4> N;
N.col(0).topRows(3) = s;
N.col(0).bottomRows(3) = Eigen::Vector3d::Zero();
N.col(1).topRows(3) = Eigen::Vector3d::Zero();
N.col(1).bottomRows(3) = s;
N.col(2).topRows(3) = -skew(s)*ey;
N.col(2).bottomRows(3) = -Ps*ey;
N.col(3).topRows(3) = skew(s)*ex;
N.col(3).bottomRows(3) = Ps*ex;

Eigen::Vector4d lambda;
lambda[1] = 0;
lambda[2] = 0;
lambda[3] = 0;
lambda[4] = 0.1*std::sin(t);
```

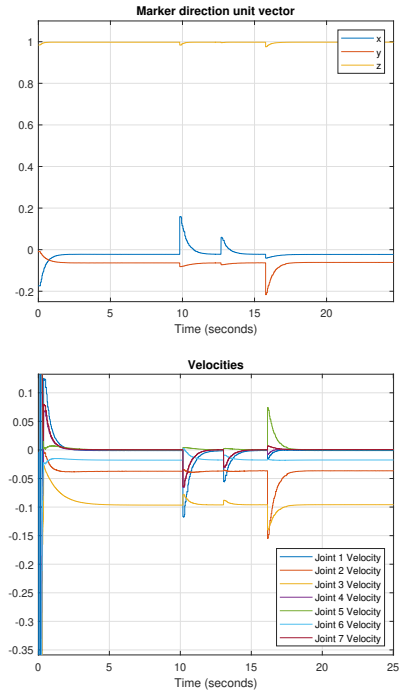
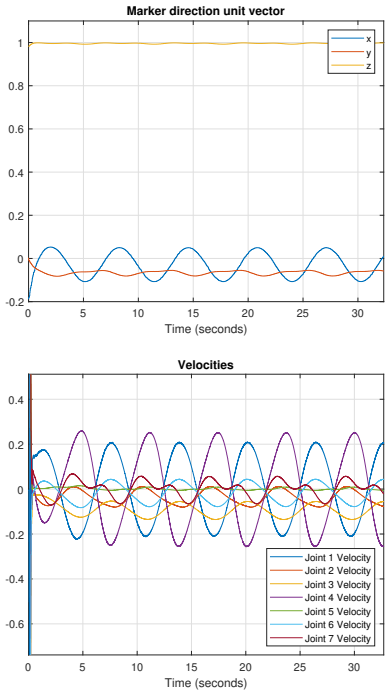
Improved look-at-point	With pseudo-velocity command
	

Table 1: Control law from [1]. In the first column, the spikes are due to the motion of the marker.

```
Eigen::MatrixXd J_dagger =
       J_cam.data.completeOrthogonalDecomposition().pseudoInverse();
dqd.data = 2 * LJ_pinv * sd + J_dagger*N*lambda +
       0.5*Null_projector * (qdi - toEigen(jnt_pos));
```

The result is visible in this video. A comparison of the results is shown in Table 1, where both velocity commands and **s** components are plotted. In order to show the components of **s** a new ROS Publisher was defined, which publishes a `geometry_msgs::Vector3` containing the three unit vector components:

```
ros::Publisher s_pub =
    n.advertise<geometry_msgs::Vector3>("/iiwa/s", 1);
```

2.c Dynamic controller with look-at-point task

The last point of the assignment consists in implementing the dynamics controllers developed in our previous work to merge both path following and look-at-point task.

Firstly, a new launch file `iiwa_gazebo_aruco_effort.launch` was created, launching the `aruco.world` with effort controllers, in order to use the dynamics controllers from the previous homework. Starting from the `iiwa_gazebo_effort.launch`, the main differences regard the inclusion of the marker model in the Gazebo path and the upload of the gazebo world containing the ArUco marker:

```
<?xml version="1.0"?>
<launch>
...
    <arg name="hardware_interface" default="EffortJointInterface" />
    ...
    <env name="GAZEBO_MODEL_PATH" value="$(find iiwa_gazebo)
           /models:$(optenv GAZEBO_MODEL_PATH)" />
    <!-- Loads the Gazebo world. -->
    <include file="$(find iiwa_gazebo)/launch/iiwa_world.launch">
        ...
        <arg name="world_name"
             value="$(find iiwa_gazebo)/worlds/iiwa_aruco.world"/>
    </include>

    <include file="$(find iiwa_control)/launch/iiwa_control.launch">
              ...
              iiwa_joint_7_effort_controller"/>
    </include>
</launch>
```

The `kdl_control_test` node has been re-implemented with all the functionalities related to the recognition of the ArUco marker. The node now includes a subscriber to the marker pose with an associated callback function:

```
void arucoPoseCallback(const geometry_msgs::PoseStamped & msg);
ros::Subscriber aruco_pose_sub =
       n.subscribe("/aruco_single/pose", 1, arucoPoseCallback);
```

The robot initial configuration has been changed to the one in `kdl_robot_vision_control` so as to obtain an initial pose where the robot can look at the marker. The camera has been added to the end-effector of the robot:

```
// Specify an end-effector: camera in flange transform
KDL::Frame ee_T_cam;
ee_T_cam.M = KDL::Rotation::RotY(1.57)*KDL::Rotation::RotZ(-1.57);
ee_T_cam.p = KDL::Vector(0,0,0.025);
robot.addEE(ee_T_cam);
```

This caused some problems with the joint space controller, because the function for the kinematic inversion appeared to be unaffected by the added robot end effector. This problem was solved converting the camera pose to the flange pose, going as input to the inverse kinematics function:

```
KDL::Frame flangePose = des_pose * (robot.getFlangeEE().Inverse());
robot.getInverseKinematics(flangePose, des_cart_vel,
des_cart_acc,qd, dqd, ddqd);
```

Once that all this changes were made, the robot was basically capable of moving along the desired trajectory as in the previous homework.

To accomplish the look-at-point task, the orientation reference of the control algorithm has to be modified. The desired orientation is now computed in order to align the z-axes of the camera to the z-axes of the marker. The error matrix R_e is defined with the angle/axes representation, where:

$$r_o = z \times s \qquad \theta = \arccos(z \cdot s) \tag{8}$$

The above equations are implemented in the following code:

```
// look at point: compute rotation error from angle/axis
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]));
Eigen::Matrix<double,3,1> aruco_pos_n = toEigen(cam_T_object.p);
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.p = KDL::Vector(p.pos[0],p.pos[1],p.pos[2]);
des_pose.M = robot.getEEFrame().M * Re;
```

The code was tested in both joint and Cartesian space, for both linear and circular trajectory. The manipulator managed to keep the marker at the center of the camera view consistently. The torques and errors, whose plots were obtained using the MATLAB function developed in the previous homework, are shown in Table 2. Additionally, the videos recordings for all four cases are available here.

References

[1] Firas Abi-Farraj, Nicolò Pedemonte, and Paolo Robuffo Giordano. "A visual-based shared control architecture for remote telemanipulation". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 4266–4273. DOI: 10.1109/IROS.2016.7759628.

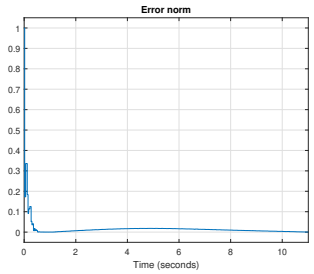
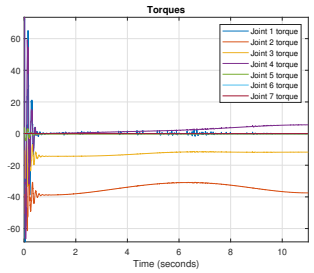
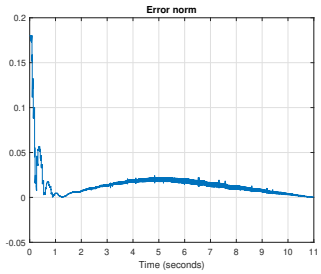
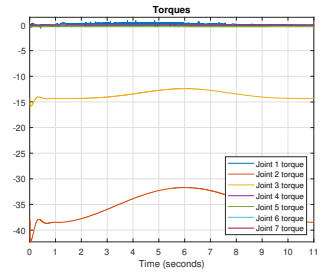
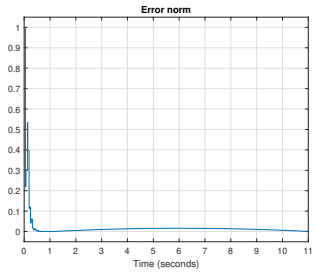
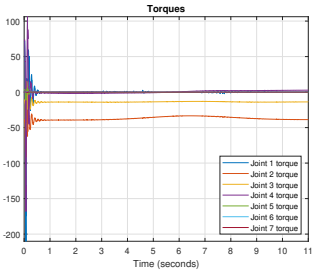
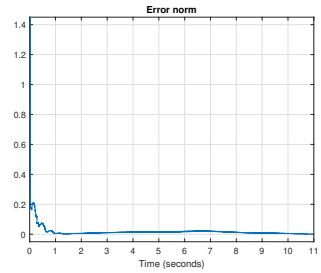
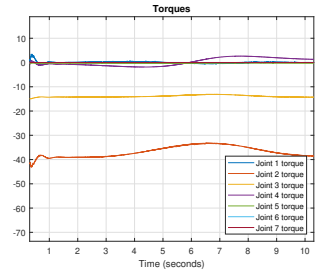
	Cartesian space	Joint space
Linear trajectory	 	 
Circular trajectory	 	 

Table 2: Error and torques plots from dynamics controllers