Robotics Lab: Homework 3 Report

a.y. 2024/2025

Students

Annese Antonio	P38000296
Bosco Stefano	P38000245
Ercolanese Luciana	P38000197
Varone Emanuela	P38000284

Contents

Goal: implement a vision-based task

This document contains a report of Homework 3 of the Robotics Lab class.The goal of this homework is to implement a vision-based controller for a 7-degrees-of-freedom robotic manipulator arm into the Gazebo environment.

1 Blue colored circular object and vision_opencv package

In this section, is displayed how construct a gazebo world inserting a blue colored circular object and how detect it via the vision_opencv package.

1.0.1 1.a `spherical_object model`

Into the gazebo/models folder of the iiwa_description package we created a new model, named spherical_object, that represents a 15 cm radius blue colored spherical object, as seen in Fig.1:

```
<?xml version="1.0" ?>
<sdf version="1.9">
  <model name="spherical_object">
    <static>true</static>
    <link name="base">
      <visual name="base_visual">
        <geometry>
          <sphere>
            <radius>0.15</radius>
          </sphere>
        </geometry>
        <material>
          <diffuse>0 0 1 1</diffuse>
          <specular>0.1 0.1 0.1 1</specular>
        </material>
      </visual>
      <collision name="base_collision">
        <geometry>
          <sphere>
            <radius>0.15</radius>
          </sphere>
        </geometry>
      </collision>
    </link>
  </model>
</sdf>
```

Figure 1: model.sdf

Then, we imported it into a new Gazebo world as a static object in x = 1, y = -0.5, z = 0.6 (Fig.2):

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <!-- We use a custom world for the rrbot so that the camera angle is launched correctly -->

  <world name="default">
    <!-- Included light -->
    <include>
      <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/models/Sun</uri>
    </include>

    <!-- Included model -->
    <include>
      <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/models/Ground Plane</uri>
    </include>

    <include>
      <uri>
        model://spherical_object
      </uri>
      <name>spherical_object</name>
      <pose>1 -0.5 0.6 0 0 0</pose>
    </include>

    <gravity>0 0 0</gravity>

    <!-- Focus camera on tall pendulum -->
    <gui fullscreen='0'>
      <camera name='user_camera'>
        <pose>4.927360 -4.376610 3.740080 0.000000 0.275643 2.356190</pose>
        <view_controller>orbit</view_controller>
      </camera>
    </gui>
  </world>
</sdf>
```

Figure 2: sphere.world

1.0.2 1.b Equip the robot with a camera

Into the urdf folder of the iiwa_description package we created a new file, named camera.xacro, which defines a macro to create a "camera" type sensor, added to the simulated robot in Gazebo (Fig.3):

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="my_camera" params="parent">

    <joint name="camera_joint" type="fixed">
      <parent link="${parent}"/>
      <child link="camera_link"/>
      <origin xyz="0.02 0.0 0.0" rpy="0 0 0"/>
    </joint>

    <link name="camera_link">
      <visual>
        <geometry>
          <box size="0.01 0.01 0.01"/>
        </geometry>
          <material name="blue">
            <color rgba="0 0 1 1"/>
          </material>
      </visual>
    </link>

    <joint name="camera_optical_joint" type="fixed">
      <parent link="camera_link"/>
      <child link="camera_link_optical"/>
      <origin xyz="0.0 0.0 0.0" rpy="${-pi/2} 0 ${-pi/2}"/>
    </joint>

    <link name="camera_link_optical"></link>


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

    <gazebo reference="camera_link">
      <sensor name="camera" type="camera">
      <pose> 0 0 0 0 0 0 </pose>
        <camera>
          <horizontal_fov>1.047</horizontal_fov>
          <image>
            <width>320</width>
            <height>240</height>
          </image>
          <clip>
            <near>0.1</near>
            <far>100</far>
          </clip>
        </camera>
        <always_on>1</always_on>
        <update_rate>30</update_rate>
        <visualize>true</visualize>
        <topic>camera</topic>
      </sensor>
    </gazebo>
  </xacro:macro>
</robot>
```

Figure 3: camera.xacro

This part of the code defines a new joint called camera_optical_joint and a link called camera_link_optical. It is an addition that is used to change the camera reference system, making it compatible with the optical standard used by simulators such as Gazebo.

- `camera_link` is the reference system of the "basic" camera: it is the attachment point of the camera to the robot, with its own orientation;
- `camera_link_optical` represents the standard optical coordinate system.There is a standard

convention in the camera world:

-The Z axis points forward (the direction in which the camera "looks").

-The X-axis points to the right.

-The Y axis points downward.

- `camera_optical_joint` connects the two references: It is a fixed joint, so there is no movement between camera_link and camera_link_optical, only the orientation changes and defines the relative position between the two. The transformation rpy = $\{-\pi/2, 0, -\pi/2\}$ changes the orientation of the reference system to align it with the standard optical one:
  - $-\pi/2$ ($-90°$ rotation on the x-axis): rotates the x-axis downward.
  - 0: no rotation on the y-axis.
  - $-\pi/2$ ($-90°$ rotation on the z-axis): correctly aligns the z-axis forward.
- `camera_link_optical`: this link has no geometry and is used only as a new reference for the optical system.

Next, after importing the camera.xacro file into iiwa.urdf.xacro and calling the my_camera macro, we positioned and connected the camera to the robot's tool0 link, so that it appears attached to the end effector (Fig.4):

```
<xacro:include filename="$(find iiwa_description)/urdf/camera.xacro"/>
<xacro:my_camera parent="${prefix}tool0"/>
```

Figure 4: Importing the camera.xacro file into iiwa.urdf.xacro

Then, we modified the iiwa.launch.py file to load the robot with the camera into the new world specifying the argument use_vision:=true, as seen in (Fig.5):

```
154     declared_arguments.append(
155         DeclareLaunchArgument(
156             'use_vision',
157             default_value='true',
158             description='Load camera',
159         )
160     )
```

```
180     use_vision = LaunchConfiguration('use_vision')
```

```
400     bridge_camera = Node(
401         package='ros_ign_bridge',
402         executable='parameter_bridge',
403         arguments=[
404             '/camera@sensor_msgs/msg/Image@gz.msgs.Image',
405             '/camera_info@sensor_msgs/msg/CameraInfo@gz.msgs.CameraInfo',
406             '--ros-args',
407             '-r', '/camera:=/videocamera',
408         ],
409         output='screen',
410         condition=IfCondition(use_vision)
411     )
```

Figure 5: use_vision

To ensure that the robot sees the imported object with the camera, its initial configuration had to be changed.To determine the initial configuration of the robot that met our needs, we introduced the following node into the iiwa.launch.py file (Fig.6):

```
joint_state_publisher_node = Node(
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
)
```

Figure 6: joint_state_publisher_node

When this node is launched, a graphical user interface (GUI) opens to move the joints of the simulated robot. The data is published to the /joint_states topic, which can be read by other nodes, such as RViz for visualizing the robot model.Proceeding in this way, it was possible to set the following as the initial configuration of the robot (Fig.7):

```
initial_positions:
  joint_a1: 0.4
  joint_a2: 1.3
  joint_a3: 1.46
  joint_a4: 1.12
  joint_a5: 2.77
  joint_a6: 1.8
  joint_a7: -1.8
```

Figure 7: initial_positions.yaml

To test the changes made, we ran the following commands in the terminal:

- `ros2 launch iiwa_bringup iiwa.launch.py use_sim:="true"`

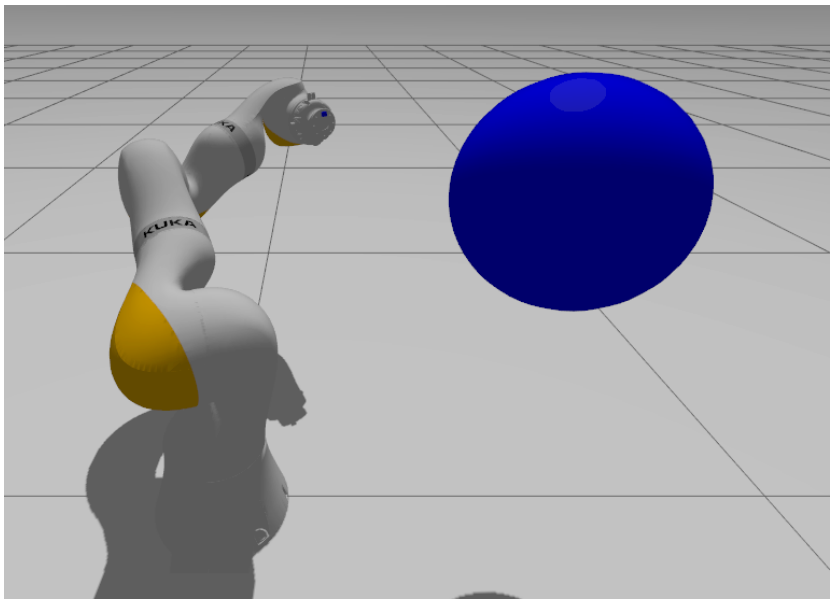

Figure 8: iiwa in Gazebo

- `ros2 run rqt_image_view rqt_image_view`

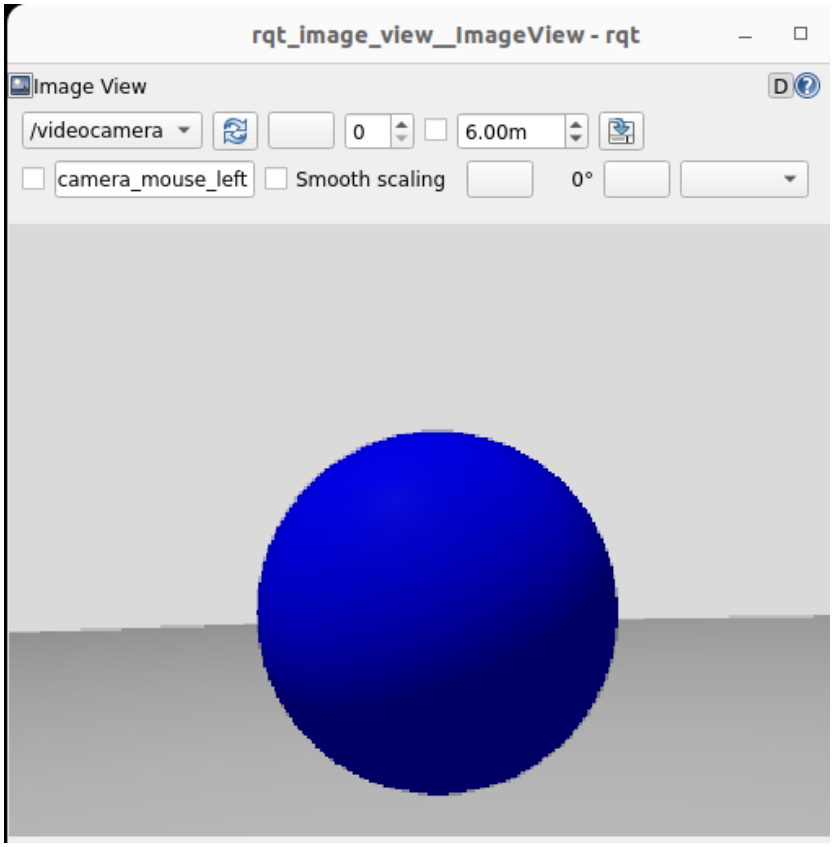

Figure 9: rqt_image_view

1.0.3 1.c `ros2_opencv package`

Once the object is visible in the camera image, we used the ros2_opencv package, specifically the ros2_opencv_node.cpp, to:

- `subscribe to the simulated image`

```
class ImageProcessorNode : public rclcpp::Node {
public:
  ImageProcessorNode() : Node("opencv_image_processor") {

    //Subscriber for the simulated image
    subscription_ = this->create_subscription<sensor_msgs::msg::Image>(
        "/videocamera", 10,
        std::bind(&ImageProcessorNode::image_callback, this, std::placeholders::_1));
```

Figure 10: Subscriber for the simulated image

The node subscribes to the /videocamera topic with a queue of size 10. When an image arrives, the image_callback function is invoked.

- `detect the spherical object in it using openCV functions`

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

//Change thresholds
params.minThreshold = 0;
params.maxThreshold = 255;

//Filter by color
params.filterByColor=false;
params.blobColor=0;

// Filter by Area.
params.filterByArea = false;
params.minArea = 0.1;

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

// Filter by Convexity
params.filterByConvexity = true;
params.minConvexity = 0.9;

// Filter by Inertia
params.filterByInertia = false;
params.minInertiaRatio = 0.01;

//Set up detector with params
cv::Ptr<cv::SimpleBlobDetector> detector = cv::SimpleBlobDetector::create(params);

std::vector<cv::KeyPoint> keypoints;

//Use the detector to find blobs in the image
detector->detect(cv_ptr->image,keypoints);
```

```
//Use the detector to find blobs in the image
detector->detect(cv_ptr->image,keypoints);

//Draw the found blobs as red circles on a copy of the original image
cv::Mat im_with_keypoints;
cv::drawKeypoints(cv_ptr->image, keypoints, im_with_keypoints, cv::Scalar(0, 0, 255),

//Update GUI Window
//Show the processed image in a window called “Image window”
cv::imshow("Image window", im_with_keypoints);
cv::waitKey(3);
```

Figure 11: openCV functions

The provided code applies a blob detection algorithm with configured parameters, draws the blobs found on the image and displays the processed image in a graphics window.

- `republish the processed image`

```
//Publisher for the processed image
publisher_ = this->create_publisher<sensor_msgs::msg::Image>("/processed_image", 10);
```

```
//Converts the processed image to a ROS message and publishes it on the topic /processed_image
auto processed_msg = cv_bridge::CvImage(std_msgs::msg::Header(), "bgr8", im_with_keypoints).toImageMsg();
publisher_->publish(*processed_msg);
```

Figure 12: Publisher for the processed image

The publisher publish the processed image on the /processed_image topic.
After running the node, this is what we can see:

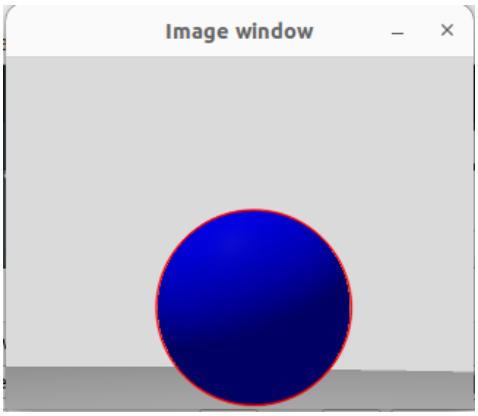

Figure 13: Processed image

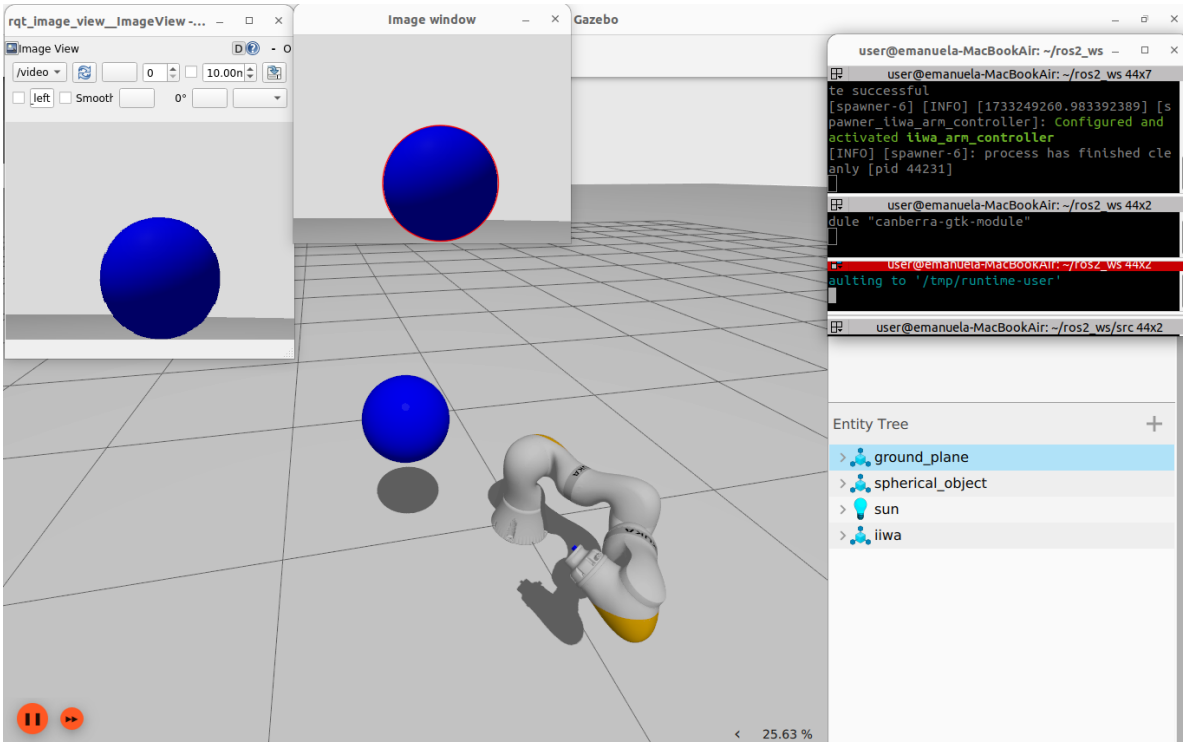

Figure 14: Complete view

2 Implementation of a look-at-point vision-based controller

The objective of this chapter is to implement a look-at-point vision-based controller.The controller should be able to perform the following two tasks:

- `aligns the camera to the aruco marker with a desired position and orientation offsets`;
- `performs a look-at-point task`

Then is required to develop a dynamic version of the vision-based contoller.

2.0.1 2.a Positioning and look-at-point tasks

To spawn the robot with the velocity command interface into a world containing an aruco tag, we created a new launch file, called iiwa_aruco.launch.py (Fig.15), which loads into the simulation the empty.world file, containing a model called arucotag, placed in a specific location and orientation (Fig.16).

```
iiwa_simulation_world = PathJoinSubstitution(
    [FindPackageShare(description_package),
        'gazebo/worlds', 'empty.world']
)
```

```
declared_arguments.append(DeclareLaunchArgument('gz_args', default_value=iiwa_simulation_world,
                                    description='Arguments for gz_sim'),)
```

```
gazebo = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                                   'launch',
                                   'gz_sim.launch.py'])]),
        launch_arguments={'gz_args': LaunchConfiguration('gz_args')}.items(),
        condition=IfCondition(use_sim),
)
```

Figure 15: iiwa_simulation_world in iiwa_aruco.launch.py file

```
<include>
  <uri>
    model://arucotag
  </uri>
  <name>arucotag</name>
  <pose>1.30 -0.35 0.62 1.57 0.01 2.16</pose>
</include>

<gravity>0 0 0</gravity>
```

Figure 16: model://arucotag in empty.world file

Running the command `ros2 launch iiwa_bringup iiwa_aruco.launch.py use_sim:="true" command_interface:="v robot_controller:="velocity_controller"`, we can see that the arucotag appears correctly loaded in the world (Fig.17):

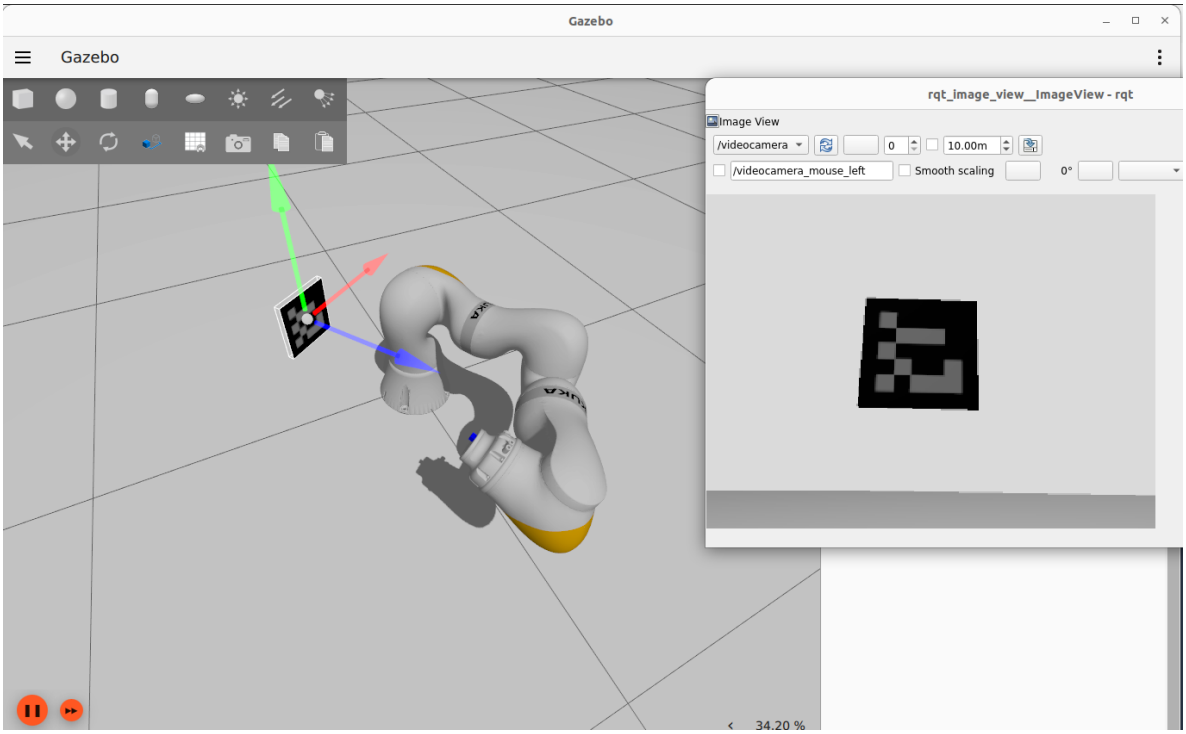

Figure 17: arucotag correctly loaded in the world

Next we created a new launch file, named aruco_cam.launch.py, which configures and starts a node, called aruco_single, for aruco marker detection using the aruco_ros package (Fig.18):

```
# Node configuration
aruco_single_node = Node(
    package='aruco_ros',
    executable='single',
    name='aruco_single',
    output='screen',
    parameters=[{
        'image_is_rectified': True,
        'marker_size': LaunchConfiguration('marker_size'),
        'marker_id': LaunchConfiguration('marker_id'),
        'reference_frame': LaunchConfiguration('reference_frame'),
        'camera_frame': 'camera_link',
        'marker_frame': LaunchConfiguration('marker_frame'),
        'corner_refinement': LaunchConfiguration('corner_refinement'),
    }],
    remappings=[
        ('/camera_info', LaunchConfiguration('camera_info_topic')),
        ('/image', LaunchConfiguration('camera_topic')),
    ]
)
```

Figure 18: Node configuration

The node receives a number of parameters:

- `marker_size`: the size of the marker, whose default value is '0.1';
- `marker_id`: the ID of the marker, whose default value is '201';

- `reference_frame`: the optional reference frame to calculate the pose;
- `marker_frame`: the name of the coordinate frame of the detected marker;
- `corner_refinement`:the method to improve the accuracy of marker edge detection

The node performs topic remapping to connect properly.

`NOTE`: in ROS, remapping allows a topic used by one node to be redirected to another real topic in the running ROS2 system.

In our case,

- the aruco_single node uses the /camera_info topic to obtain camera calibration information (such as the intrinsic matrix and distortion coefficients).With ('camera_info',LaunchConfiguration('camera_info_topic')), it is specified that the actual topic to be used will not be fixed, but will depend on the value of the launch argument camera_info_topic.
- The aruco_single node uses the /image topic to receive frames captured by the camera.With ('/image', LaunchConfiguration('camera_topic')), the actual topic , from which the node will read the frames, will be determined by the value of camera_topic, whose defaul value is /videocamera.

Running the commands:

- `ros2 launch aruco_ros aruco_cam.launch.py`
- `ros2 topic list`

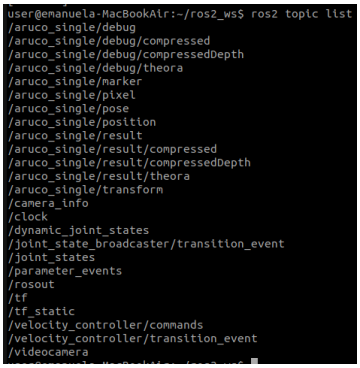

Figure 19: topic list

- `ros2 run rqt_image_view rqt_image_view`

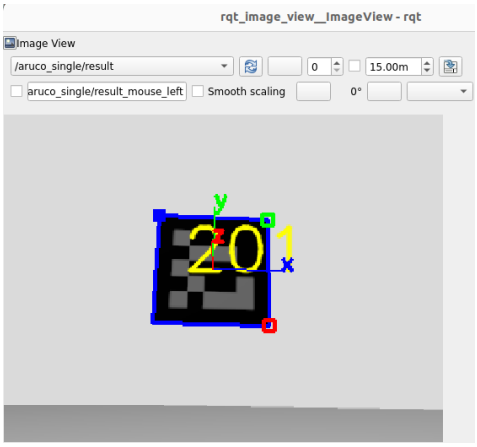

Figure 20: /aruco_single/result

we can see that aruco marker detection is successfully performed.
NOTE: the /aruco_single/result topic provides the position of the marker relative to the camera and includes orientation information.
In the ros2_kdl_package package we created a ros2_kdl_vision_control.cpp node that implements a vision-based controller for the simulated iiwa robot.
The controller should be able to perform the following two tasks:

- `i.` aligns the camera to the aruco marker with a desired position and orientation offsets;
- `ii.` performs a look-at-point task using the following control law:

$$\dot{q} = k(LJ_c)^{\dagger} s_d + N\dot{q}_0, \tag{1}$$

where $s_d = [0, 0, 1]$ is a desired value for

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

that is a unit-norm axis connecting the origin of the camera frame and the position of the object ${}^{c}P_o$. The matrix J_c is the camera Jacobian (to be computed), while $L(s)$ maps linear/angular velocities of the camera to changes in s

$$L(s) = \left[-\frac{1}{||{}^{c}P_o||} \left(I - ss^T\right) \quad S(s) \right] R \in \mathbb{R}^{3\times 6} \quad \text{with} \quad R = \begin{bmatrix} R_c & 0 \\ 0 & R_c \end{bmatrix}, \tag{3}$$

where $S(\cdot)$ is the skew-simmetric operator, R_c the current camera rotation matrix. Finally, $N = \left(I - (LJ)^{\dagger}LJ\right)$ is the matrix spanning the null space of the LJ matrix.

To switch between tasks, we introduced a new argument, named "task_", as seen in (Fig.21):

```
class VisionControlNode : public rclcpp::Node
{
public:
    VisionControlNode()
    :Node("ros2_kdl_vision_control"),
    node_handle_(std::shared_ptr<VisionControlNode>(this))
    {

        //Declaration of task parameter
        declare_parameter<std::string>("task", "positioning");
        get_parameter("task", task_);
        RCLCPP_INFO(get_logger(),"Current task is: '%s'", task_.c_str());

            if (!(task_ == "positioning" || task_ == "look-at-point"))
            {
                RCLCPP_INFO(get_logger(),"Selected task is not valid!"); return;
            }
```

Figure 21: Declaration of task parameter

The `positioning task` is implemented as follows:

- `Definition of frames and transformations:`

```
// From object to base frame with offset on rotation and position
KDL::Frame marker_wrt_camera(marker.M,
KDL::Vector(marker.p.data[0],marker.p.data[1]-0.12,marker.p.data[2]-0.27));
cam_link_to_optical= KDL::Frame(KDL::Rotation::Quaternion(-0.5,0.5,-0.5,0.5),KDL::Vector(0,0,0));
marker_wrt_world =  robot_->getEEFrame() * cam_link_to_optical* marker_wrt_camera;
```

Figure 22

This part of the code defines:
- `marker_wrt_camera:` the frame of the marker with respect to the camera with an offset in position.The position is changed by subtracting 0.12 and 0.27 from the y and z axes, respectively.
`-cam_link_to_optical:` the transformation between the camera frame and its optical frame by applying a rotation through a quaternion.
`-cam_link_to_optical:` the transformation between the camera frame and its optical frame by applying a rotation through a quaternion.
`-marker_wrt_world:` the position of the marker in the world. It starts from the robot's end-effector frame, then, the optical camera transformation is applied and, finally, the position of the marker with respect to the camera is transformed.

- `Calculation of desired position and orientation:`

```
// Compute desired Frame
KDL::Frame desFrame;
desFrame.M = end_orientation*KDL::Rotation::RotY(-1.57)*KDL::Rotation::RotZ(0)*KDL::Rotation::RotX(1.57);
desFrame.p = toKDL(end_position);
```

Figure 23

This part of the code defines the desired frame for the robot:
`-Desired orientation:`it starts from the marker orientation and applies additional rotations (-90° on the y axis, +90° on the x axis).
`-Desired position:` this is the position of the marker.

- `Computation of errors:`

```
// compute errors
Eigen::Vector3d error = computeLinearError(Eigen::Vector3d(desFrame.p.data), Eigen::Vector3d(cartpos.p.data));
Eigen::Vector3d o_error = computeOrientationError(toEigen(cartpos.M), toEigen(desFrame.M));
```

Figure 24

This part of the code defines:
`-Linear error:` difference between the current position of the robot (cartpos.p) and the desired position (desFrame.p).
`-Angular error:` difference between the current orientation (cartpos.M) and the desired orientation (desFrame.M)

- `Robot control:`

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

        Vector6d cartvel; cartvel << 0.05*p.vel + kp*error, ko*o_error;
        joint_velocities_.data = pseudoinverse(robot_->getEEJacobian().data)*cartvel;
        joint_positions_.data = joint_positions_.data + joint_velocities_.data*dt;
```

Figure 25

This part of the code generates control commands (cartesian velocities) and converts them to joint velocities using the robot's Jacobian.Then, updates joint positions in positioning mode.
The `attached video ("Positioning.mp4")` shows the positioning task executed by the robot.

The `look-at-point task` is implemented the follows:

- `Computation of matrices:`

```
//Computation of matrices
Eigen::Matrix<double,3,1> c_Po = toEigen(marker_wrt_camera.p);
Eigen::Matrix<double,3,1> s = c_Po/c_Po.norm();
Eigen::Matrix<double,3,3> Rc = 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,3,6> L = Eigen::Matrix<double,3,6>::Zero();
Eigen::Matrix<double,6,6> Rc_grande = Eigen::Matrix<double,6,6>::Zero();
Rc_grande.block(0,0,3,3) = Rc;
Rc_grande.block(3,3,3,3) = Rc;
L.block(0,0,3,3) = L_block;
L.block(0,3,3,3) = skew(s);
L=L*Rc_grande;

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

Figure 26

This part of the code calculates:
- `c_Po:` position of the marker relative to the camera;
- `s:` normalized vector pointing from the robot to the marker;
- `L_block:` a matrix that projects onto a space tangent to the point on the marker, which is needed to compute motions that point to the target;
- `Rc_grande:` expands the current rotation matrix (Rc) into a 6×6 form to include translation and rotation;
- `L:` matrix for tangent transformation, which includes the robot's contribution of position and rotation;
- `LJ:` projection of the robot's Jacobian in tangent space;
- `LJ_pinv:` pseudoinverse of LJ, computed by orthogonal decomposition;
- `N:` matrix of null space, representing the unconstrained joint motions bound to the target point.

- `Robot control:`

```
else if(task_ == "look-at-point"){

        dqd.data=k*LJ_pinv*sd +1*N*(qdi.data-joint_positions_.data);
```

Figure 27

This part of the code calculates and applies joint velocities to perform the required task, managing robot kinematics with multiple priorities: making the end-effector "look" toward the marker and preserving desired poses through null-space.

The `attached video ("LOOK+POSITIONING.mp4")` shows the look-at-point task executed by the robot.

2.0.2 2.b Develop a dynamic version of the vision-based contoller

The objective of this section is to 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. To switch between the joint space and the Cartesian space inverse dynamics controllers, we introduced a new argument, named "cmd_interface", as seen in (Fig.28):

```
// declare cmd_interface parameter (effort or effort_cartesian)
declare_parameter<std::string>("cmd_interface", "effort");
get_parameter("cmd_interface", cmd_interface_);

RCLCPP_INFO(get_logger(),"Current cmd interface is: '%s'", cmd_interface_.c_str());

if (!(cmd_interface_ == "effort" || cmd_interface_ == "effort_cartesian"))
{
    RCLCPP_INFO(get_logger(),"Selected cmd interface is not valid!"); return;
}
```

Figure 28

After setting the final position and orientation as the position of the marker relative to the world and rotation matrix of the marker relative to the world, respectively (Fig.29):

```
end_position<<marker_wrt_world.p.data[0],marker_wrt_world.p.data[1]+0.014,marker_wrt_world.p.data[2];
end_orientation = marker_wrt_world.M;
```

Figure 29

we defined a trajectory schedule using a trapezoidal velocity profile (Fig.30):

```
double traj_duration = 1.5, acc_duration = 0.5, t = 0.0;
planner_ = KDLPlanner(traj_duration, acc_duration, init_position, end_position); //currently using trapezoidal velocity profile

trajectory_point p = planner_.compute_trajectory(t);
```

Figure 30

The computation of the orientation error is a crucial step for successful performance of the task.We proceeded as shown below (Fig.31):

```
// compute errors
Eigen::Vector3d error ;
Eigen::Vector3d o_error;
KDL::Frame des_cartpos;

double cos_theta = sd.dot(s); // Prodotto scalare tra i due vettori
cos_theta = std::max(-1.0, std::min(1.0, cos_theta)); // Assicura che il valore sia tra -1 e 1 per evitare errori numerici
///double angular_error = std::acos(cos_theta);
double cam_o_error = std::acos(cos_theta);
Eigen::Vector3d orientation_error_cam = sd.cross(s);
KDL::Rotation axang = (KDL::Rotation::Rot(toKDL(orientation_error_cam), cam_o_error)); //costriusco la matrice di rotazione tra i

// Trasformazione nel frame base
Eigen::Vector3d orientation_error = computeOrientationError( toEigen(robot_->getEEFrame().M*axang),toEigen(cartpos.M));
```

Figure 31

The orientation error calculation was implemented following this steps: We calculate the angle and axis of rotation needed to align two intermediate orientations (through the vectors s and sd).The correct orientation is transformed into the base frame. Finally, we calculate the orientation error as the difference between the current transformed orientation and the desired orientation. At the end, we proceeded to calculate the torque (Fig.32):

```
joint_velocity_old.data=joint_velocities_.data;

Vector6d cartvel; cartvel << p.vel + 10*error, 10*orientation_error;

//Update joint velocities, using the pseudoinverse of the end-effector Jacobian to map the desired Cartesian velocity (cartvel) in joint
dq_des.data = pseudoinverse(robot_->getEEJacobian().data)*cartvel ;

//Calculate the new joint positions by integrating the velocities (joint_velocities_) with the time step dt:
q_des.data = joint_positions_.data + joint_velocities_.data*dt;

//Calculate joint acceleration by discrete numerical derivative:
joint_acceleration_d_.data=(joint_velocities_.data-joint_velocity_old.data)/dt;

//Use the first method (idCntr) to calculate the required joint torques:
torque_values = controller_.idCntr(q_des,dq_des,joint_acceleration_d_, _Kp, _Kd);
```

Figure 32

The method pursued to extrapolate the plot consisted of saving the joint torque command topics in a bag file and plotting it using MATLAB. The figure (Fig. 31) provided below show the plot obtained for the linear-trapezoidal trajectory:

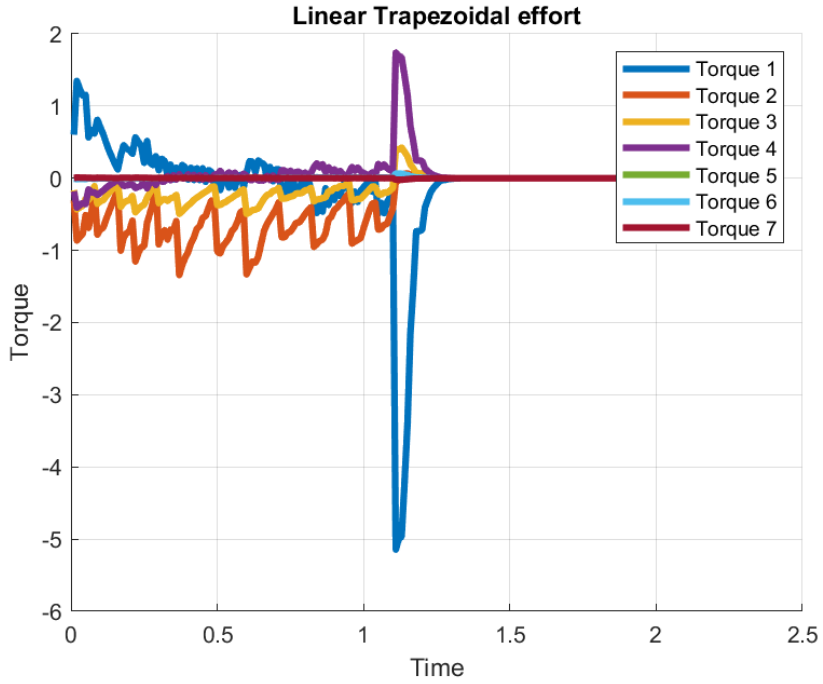

Figure 33

The `attached video ("Effort_JS.mp4")` shows the look-at-point task executed by the robot.

3 GitHub repositories links:

Students

Annese Antonio	https://github.com/antann1/HomeworkRL_3.git
Bosco Stefano	https://github.com/SteBosco/HMW3_RL_2024.git
Ercolanese Luciana	https://github.com/LErcolanese/RL_Homework3.git
Varone Emanuela	https://github.com/Emanuela-var/Homework3_RL24.git