UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Robotics Lab Project

HOMEWORK 3

Academic Year 2024/2025

Relatore
Ch.mo prof. Mario Selvaggio
Candidato
Emmanuel Patellaro
P38000239
Repository
`https://github.com/EmmanuelPat6/Homework_3.git`

Abstract

This project focuses on developing a **Vision-Based Controller** for a 7-degrees-of-freedom robotic manipulator arm into the Gazebo Environment. At starting point, *ros2_ kdl_ package* package
(`https://github.com/RoboticsLab2024/ros2_kdl_package`) and *ros2_ iiwa_ packa*
package
(`https://github.com/RoboticsLab2024/ros2_iiwa`) have been used. First of all, a world in Gazebo was created containing a blue colored spherical object. Then, this object has been detected via the *vision_ opencv* package
(`https://github.com/RoboticsLab2024/ros2_vision/tree/main`). At the end, a **Look-at-Point Vision-Based Controller** has been implemented both with velocity control and dynamic control.

Contents

Implement a Vision-Based Task

Object Detection

For the trajectory planning of any robot and for many types of tasks, it is often crucial to have a clear vision and perception of the surrounding environment. To this end, the **Object Detection** through camera sensors becomes of paramount importance.

Object Creation

To construct a Gazebo World containing a blue colored spherical object, the world **empty.world** was modified and a new world **new_world.world** was created. Let's see it:

```xml
<?xml version="1.0" ?>
<sdf version="1.4">

  <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://arucotag
      </uri>
      <name>arucotag</name>
      <pose>0 -0.707 0.707 0 1.57 0</pose>
```

```
    </include>

    <include>
      <uri>
        model://spherical_object
      </uri>
      <name>spherical_object</name>
      <pose>1.0 -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>
```

To allow this, in the directory /**iiwa_description_gazebo**/**models** a new directory **spherical_object** containing all the data to enable the creation of the spherical object was created. In particular, in the file **model.config**

```
<?xml version="1.0"?>
<model>
  <name>sphere</name>
  <version>1.0</version>
  <sdf version="1.9">model.sdf</sdf>
  <author>
    <name>Emmanuel Patellaro</name>
    <email>e.patellaro@studenti.unina.it</email>
  </author>
  <description>Sphere</description>
</model>
```

and in **model.sdf**

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.9">
  <model name="spherical_object">
    <static>true</static>
    <link name="sphere_link">
      <visual name="sphere_visual">
        <geometry>
          <sphere>
            <radius>0.15</radius>
          </sphere>
        </geometry>
        <material>
```

```
          <diffuse>0.2 0.2 0.8 1</diffuse>
          <specular>0.4 0.4 0.4 1</specular>
        </material>
      </visual>
      <collision name="sphere_collision">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <sphere>
            <radius>0.15</radius>
          </sphere>
        </geometry>
      </collision>
    </link>
  </model>
</sdf>
```

At this point, it is possible to launch the new world creating a new launch file called **iiwa_sphere.launch.py**, similar to **iiwa.launch.py** file, but changing the argument related to the chosen world.

```
iiwa_new_world = PathJoinSubstitution(
    [FindPackageShare(description_package),
        'gazebo/worlds', 'new_world.world']
)
declared_arguments.append(DeclareLaunchArgument('gz_args',
   default_value=iiwa_new_world,
                            description='Arguments for gz sim with 
                                camera'),)

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),
)
```

```
ros2 launch iiwa_bringup iiwa_sphere.launch.py use_sim:=true
```

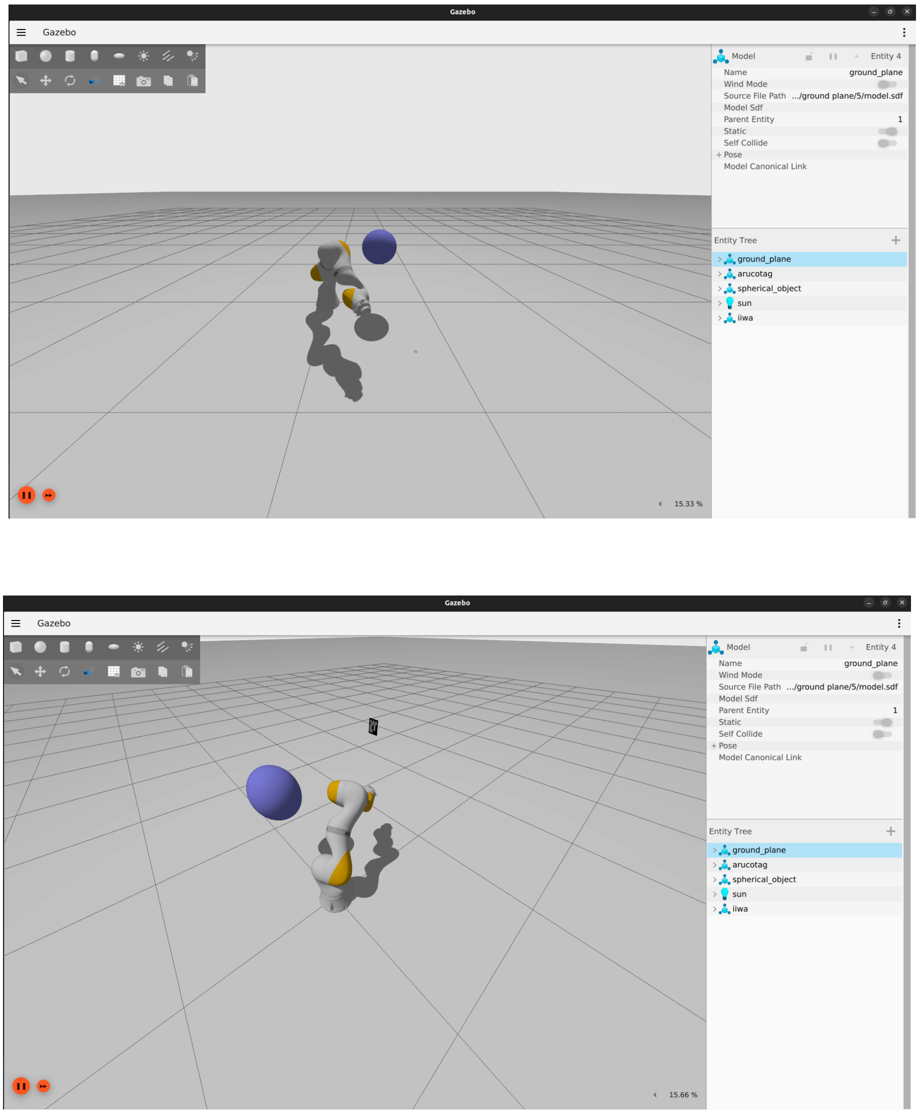

Figure 1: Spherical Object in Gazebo

Camera Sensor

As mentioned in the introduction of this chapter, a **Camera Sensor** was added to the manipulator to **detect** the newly inserted object. The same steps as in Homework 1 were followed. For the **camera.xacro** file in **iiwa_description/urdf** you have:

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

```
  <link name="camera_link">
      <visual>
          <geometry>
              <box size="0.0000010␣0.00000003␣0.00000003"/>
          </geometry>
          <material name="red"/>
      </visual>
  </link>

  <joint name="camera_joint" type="fixed">
      <parent link="tool0"/>
      <child link="camera_link"/>
      <origin xyz="0.0␣0.0␣0.0" rpy="3.14␣-1.57␣0.0"/>
  </joint>

</robot>
```

while for **iiwa_camera.xacro** in **iiwa_description/gazebo**

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

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="camera_gaz">

  <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">
      <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>

</robot>
```

Instead, in **iiwa.urdf.xacro**

```
<!-- Import camera -->
<xacro:if value="$(arg␣use_vision)">
    <xacro:include filename="$(find␣iiwa_description)/urdf/camera.
       xacro"/>
    <xacro:include filename="$(find␣iiwa_description)/gazebo/
       iiwa_camera.xacro"/>
</xacro:if>
```

is added. The variable **use_vision** is implemented to load the robot with the camera into the new world specifying the argument **use_vision:=true**. In order to do this, in some parts of the code, this variable was added. Here some example. In **iiwa.config.xacro**

```
<xacro:arg name="use_vision" default="false" />
...
...
...
use_vision="$(arg␣use_vision)"
```

and so on. At the end, in the launch file, a new node was created in order to launch the camera when it is necessary

```
bridge_camera = Node(
    package='ros_ign_bridge',
    executable='parameter_bridge',
    arguments=[
        '/camera@sensor_msgs/msg/Image@gz.msgs.Image',
        '/camera_info@sensor_msgs/msg/CameraInfo@gz.msgs.CameraInfo'
            ,
        '--ros-args',
        '-r', '/camera:=/videocamera',
     ],
     output='screen'
)
```

To choose the orientation of the camera, I used **tf** in **RViz2** to ensure that the direction of the red axis aligned with the one emerging from the end-effector, which represents the camera's direction.

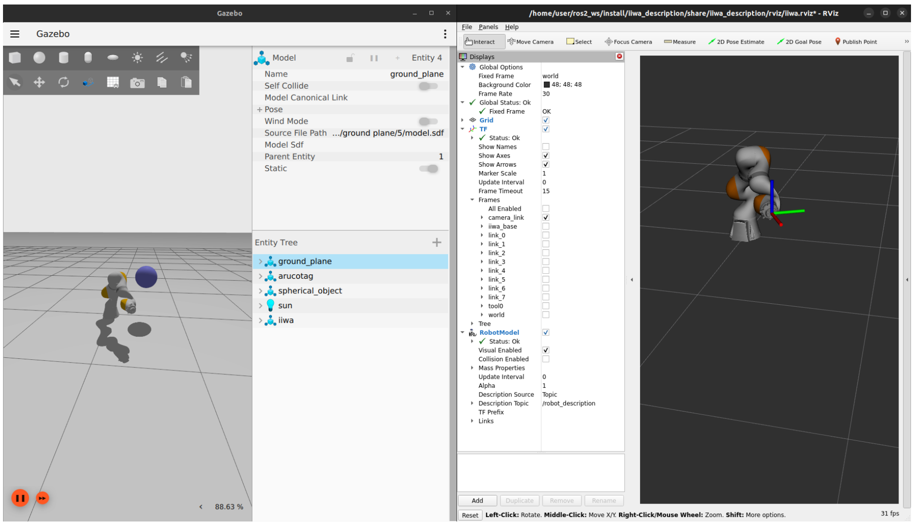

Figure 2: use_vision := true

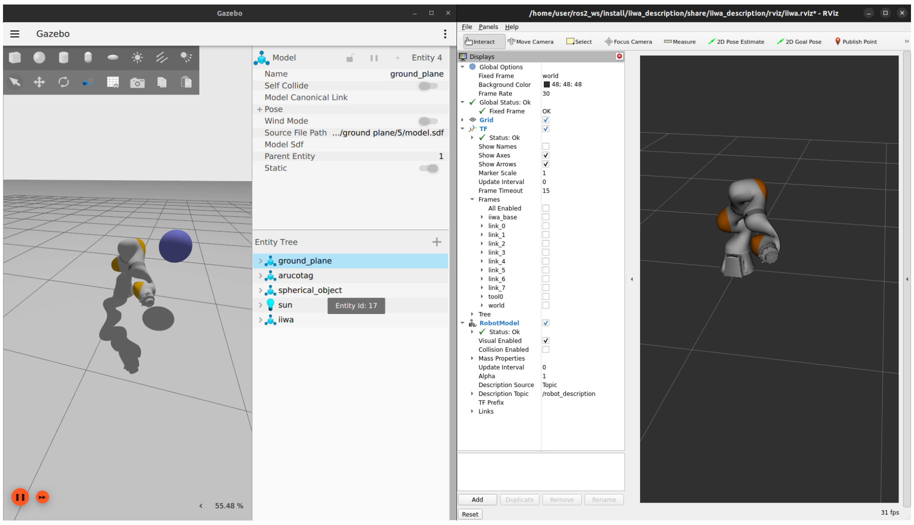

Figure 3: use_vision:=false

Furthermore, using **rqt_image_view**, the initial position of the robot was adjusted to allow the spherical object to be visible within the camera's view. So, in **inital_position.yaml** in **/iiwa_description/config**

```
initial_positions:
  joint_a1: 0.4124
  joint_a2: 1.2447
  joint_a3: 1.4574
  joint_a4: 1.0436
  joint_a5: 2.7578
  joint_a6: 1.7968
  joint_a7: -1.8123
```

Changing this, the new robot pose is the following in the figure below.

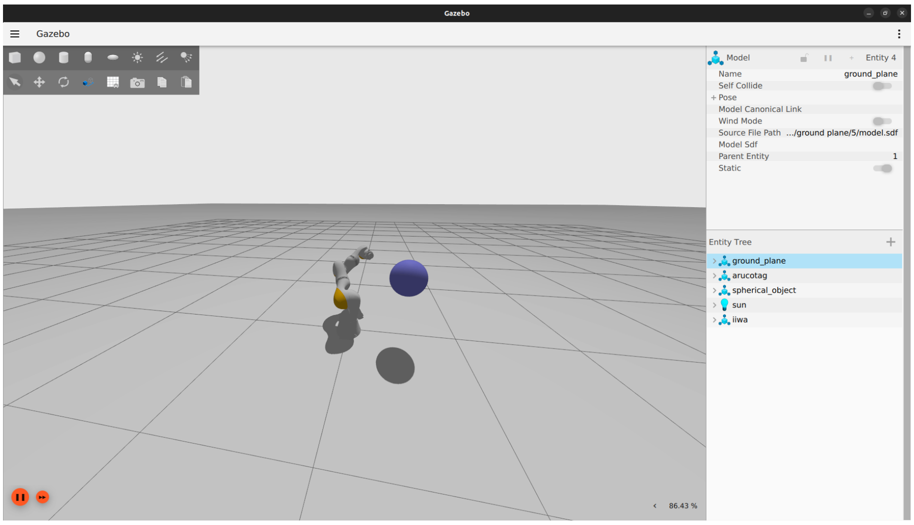

Figure 4: Manipulator looking at the object

Running

```
ros2 launch iiwa_bringup iiwa_sphere.launch.py
use_sim:=true use_vision:=true
```

and

```
ros2 run rqt_image_view rqt_image_view
```

it is possible to see the object viewed by the robot through the camera.

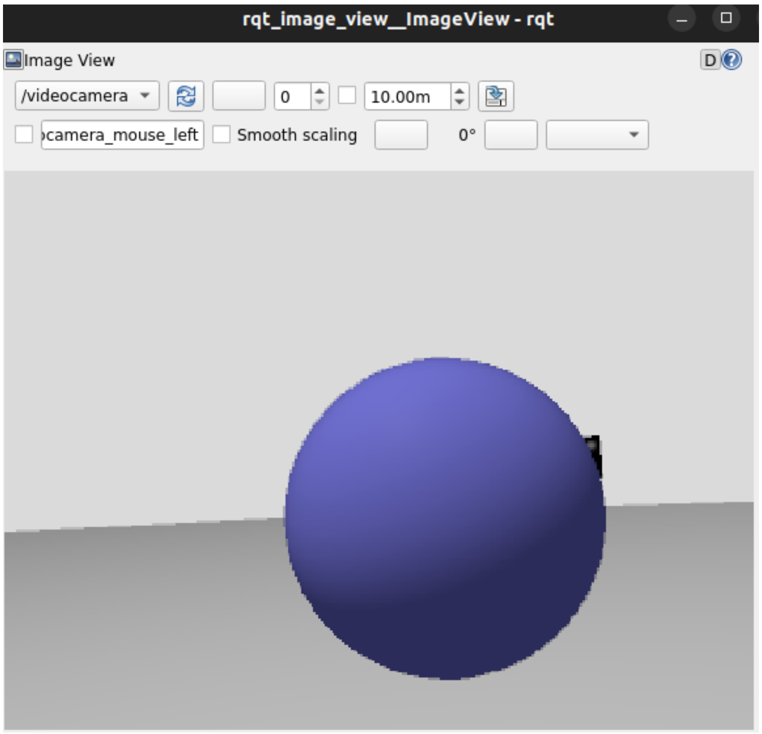

Figure 5: Spherical Object through the Camera

Detection

To conclude this chapter, the **detection** of the object created in the previous section has been implemented. The **ros2_opencv** package (and specifically the **ros2_opencv_node.cpp** file) has been used to **subscribe** to the simulated image, **detect** the spherical object in it (using openCV functions), and **republish** the processed image. Let's see the **ros2_opencv_node.cpp** file.

```
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/image.hpp"
#include "std_msgs/msg/header.hpp"
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.hpp>
#include <opencv2/opencv.hpp>

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));

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

private:

  void image_callback(const sensor_msgs::msg::Image::SharedPtr msg)
  {
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
      //Converts the ROS message to an OpenCV object with BGR8
          encoding
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings
          ::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
      RCLCPP_ERROR(this->get_logger(), "cv_bridge␣exception:␣%s", e.
          what());
      return;
    }

    //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 = 0.5;

    //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 = true;
    params.minInertiaRatio = 0.9;
```

```
    //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);

    //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), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS
        );

    //Show the processed image in a window called 'Sphere Detection'
    cv::imshow("Sphere Detection", im_with_keypoints);
    cv::waitKey(0);

    //Conversion and Publishing
    auto processed_msg = cv_bridge::CvImage(std_msgs::msg::Header(),
        "bgr8", im_with_keypoints).toImageMsg();
    publisher_->publish(*processed_msg);
  }

  rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr
     subscription_;
  rclcpp::Publisher<sensor_msgs::msg::Image>::SharedPtr publisher_;

};


int main(int argc, char *argv[]) {

  rclcpp::init(argc, argv);
  auto node = std::make_shared<ImageProcessorNode>();
  rclcpp::spin(node);

  rclcpp::shutdown();
  return 0;
}
```

All that remains is to visualize the results. Various attempts are shown. By adjusting the detection parameters, the results were progressively improved until reaching the final outcome.

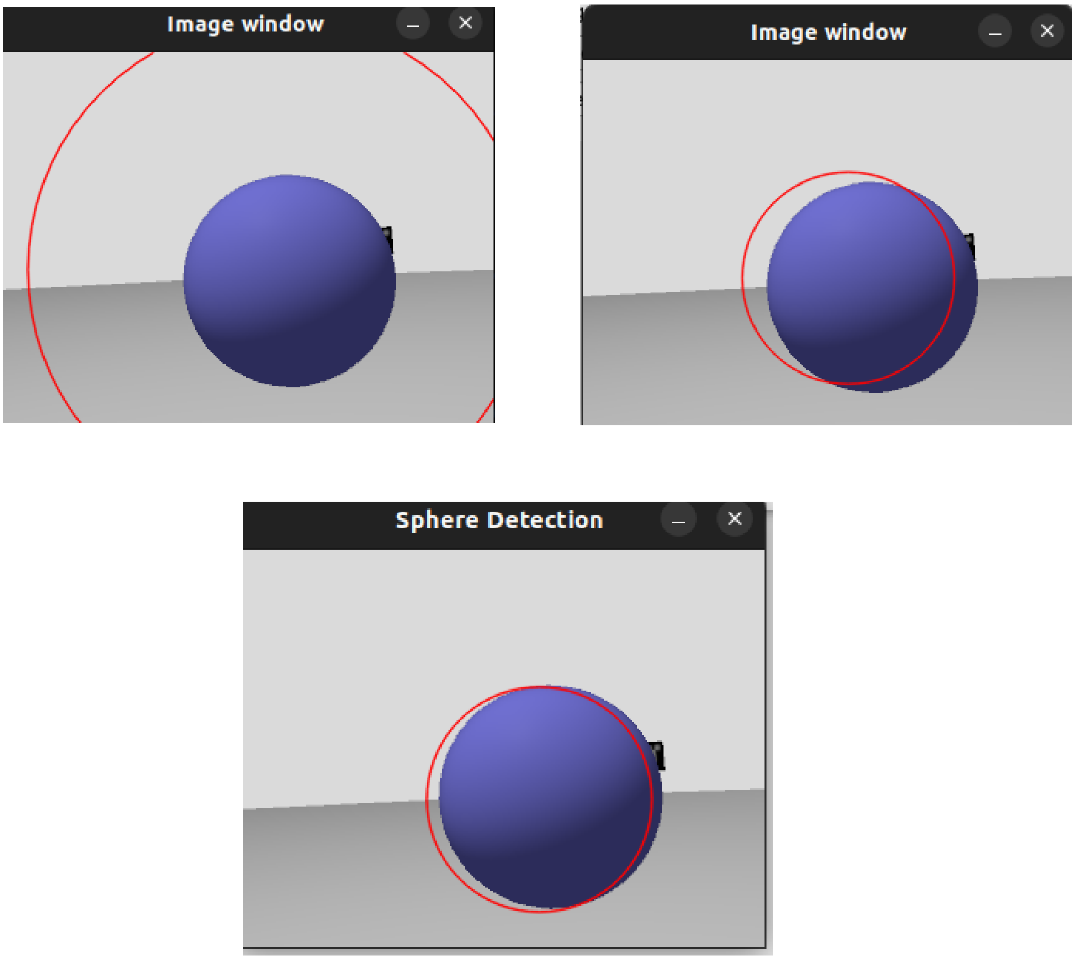

Figure 6: Detection Parameters Tuning

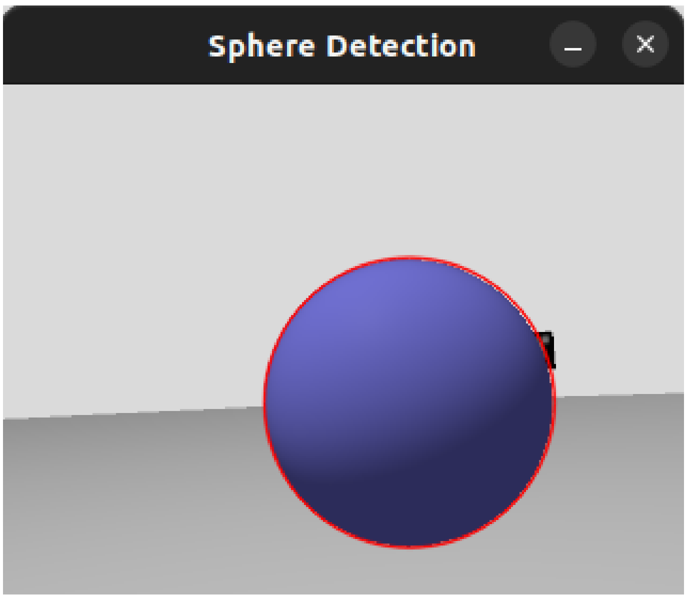

Figure 7: Spherical Object Detection

It is possible to check that, changing the initial position of the manipulator, the detection still remains optimal

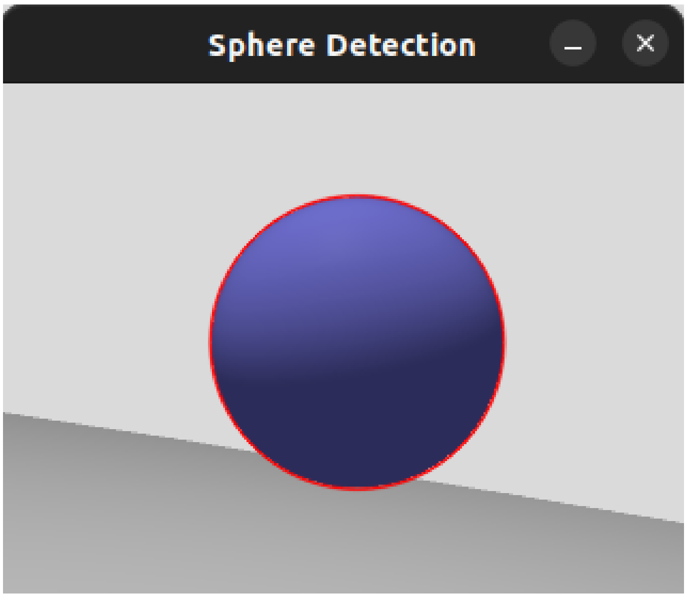

Figure 8: Spherical Object Detection with another Initial Position

The instructions are:

```
ros2 launch iiwa_bringup iiwa_sphere.launch.py
use_sim:=true use_vision:=true
```

```
ros2 run rqt_image_view rqt_image_view
```

```
ros2 run ros2_opencv ros2_opencv_node
```

Vision-Based Controller

In this chapter a **Vision-Based Controller** will be implemented. Thanks to this type of controller, the robot can move based on the visualization and detection of a specific **ArUco tag** (in this case the **201** one).

Vision-Based Velocity Controller

First, a **Vision-Based Velocity Controller** was created. The world used was the **empty.world** provided by Prof. Mario Selvaggio in the starting repository. The world is the following

```xml
<?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://arucotag
      </uri>
      <name>arucotag</name>
        <pose>1.10 -0.47 0.65 1.57 0.0 2.1</pose>
```

```
      <!--<pose>0.15 -0.5 0.45 -0.3 1.57 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>
```

with **Zero Gravity** and with the **ArUco tag**, defined by the following **model.sdf** file

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.9'>
  <model name='arucotag'>
    <static>true</static>
    <pose>0 0 0.001 0 0 0</pose>
    <link name='base'>
      <visual name='base_visual'>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>0.1 0.1</size>
          </plane>
        </geometry>
        <material>
          <diffuse>1 1 1 1</diffuse>
          <specular>0.4 0.4 0.4 1</specular>
          <pbr>
            <metal>
              <albedo_map>model://arucotag/aruco-201.png</albedo_map>
            </metal>
          </pbr>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

Then, the file **single.launch.py** in **ros2_vision/aruco_ros/aruco_ros/launch** directory, was modified in so that the manipulator could detect the ArUco Tag through its Camera:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, OpaqueFunction
```

```
from launch.substitutions import LaunchConfiguration
from launch.utilities import perform_substitutions
from launch_ros.actions import Node


def launch_setup(context, *args, **kwargs):

    eye = perform_substitutions(context, [LaunchConfiguration('eye')
        ])

    aruco_single_params = {
        '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')
            ,
    }

    aruco_single = Node(
        package='aruco_ros',
        executable='single',
        parameters=[aruco_single_params],
        remappings=[('/camera_info', '/camera_info'),
                    ('/image', '/videocamera')],
    )

    return [aruco_single]


def generate_launch_description():

    marker_id_arg = DeclareLaunchArgument(
        'marker_id', default_value='201',
        description='Marker␣ID.␣'
    )

    marker_size_arg = DeclareLaunchArgument(
        'marker_size', default_value='0.1',
        description='Marker␣size␣in␣m.␣'
    )

    eye_arg = DeclareLaunchArgument(
        'eye', default_value='left',
        description='Eye.␣',
        choices=['left', 'right'],
    )

    marker_frame_arg = DeclareLaunchArgument(
        'marker_frame', default_value='aruco_marker_frame',
        description='Frame␣in␣which␣the␣marker␣pose␣will␣be␣refered.␣
            '
    )
```

```
    reference_frame = DeclareLaunchArgument(
        'reference_frame', default_value='camera_link',
        description='Reference␣frame.␣'
        'Leave␣it␣empty␣and␣the␣pose␣will␣be␣published␣wrt␣param␣
           parent_name.␣'
    )

    corner_refinement_arg = DeclareLaunchArgument(
        'corner_refinement', default_value='LINES',
        description='Corner␣Refinement.␣',
        choices=['NONE', 'HARRIS', 'LINES', 'SUBPIX'],
    )

    # Create the launch description and populate
    ld = LaunchDescription()

    ld.add_action(marker_id_arg)
    ld.add_action(marker_size_arg)
    ld.add_action(eye_arg)
    ld.add_action(marker_frame_arg)
    ld.add_action(reference_frame)
    ld.add_action(corner_refinement_arg)

    ld.add_action(OpaqueFunction(function=launch_setup))

    return ld
```

The instructions to run are

```
ros2 launch iiwa_bringup iiwa.launch.py

command_interface:="velocity"

robot_controller:="velocity_controller"

use_sim:=true use_vision:=true
```

```
ros2 launch aruco_ros single.launch.py
```

```
ros2 run rqt_image_view rqt_image_view
```

In **rqt_image_view** select the topic /**aruco_single**/**result**. In fact, very important the topic /**aruco_single**/**pose** from which it is possible to extract the ArUco Tag Position.

```
user@patpc:~/ros2_ws$ ros2 topic info /aruco_single/pose
Type: geometry_msgs/msg/PoseStamped
Publisher count: 1
Subscription count: 0
```

Figure 9: Info /aruco_single/pose

```
user@patpc:~/ros2_ws$ ros2 topic echo /aruco_single/pose
header:
  stamp:
    sec: 300
    nanosec: 499000000
  frame_id: camera_link
pose:
  position:
    x: -0.11312488466501236
    y: -0.030817689374089244
    z: 0.7616513967514038
  orientation:
    x: 0.7176781662336802
    y: 0.6924957700589428
    z: -0.022584558332570975
    w: 0.06983978727671374
```

Figure 10: echo /aruco_single/pose

The detection is shown in the figures below.

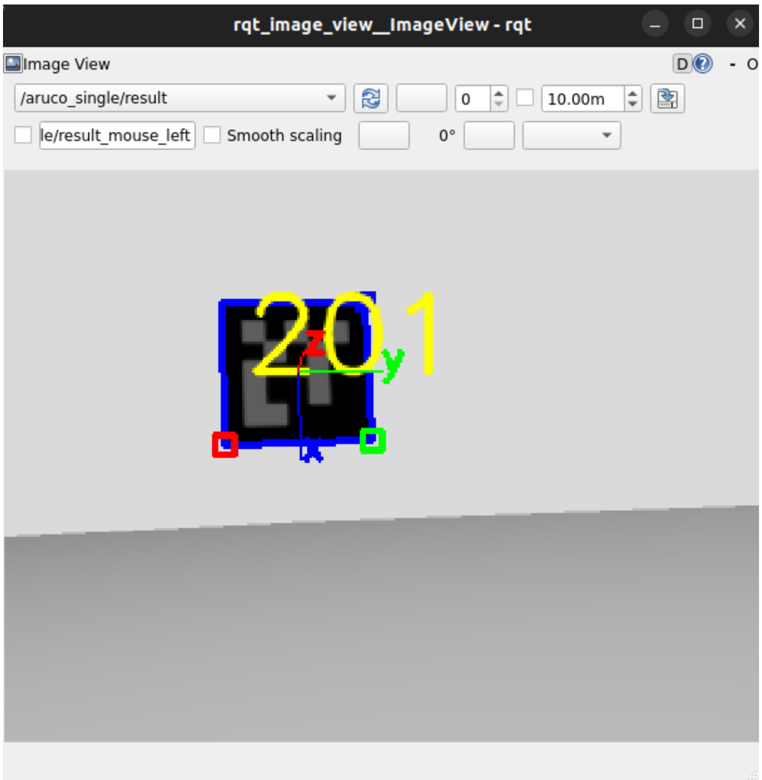

Figure 11: ArUco Detection

Since the camera has poor resolution, to improve subsequent simulations, the quality of the camera was enhanced. In **iiwa_camera.xacro** some parameters were increased:

```
<image>
  <width>640</width>
  <height>480</height>
</image>
```

Let's see the controller implementation. The controller should be able to perform the following two tasks:

- **Positioning Task**: aligns the camera to the aruco marker with a desired position and orientation offests
- **Look at Point Task**: look to the ArUco tag which can move in the surrounding environment.

To avoid repeating code multiple times, the final implementation of the two tasks will be shown only once. The control law for the **look-at-point** task is

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

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

$$s = \frac{{}^cP_o}{||{}^cP_o||} \in \mathcal{S}^2$$

that is a unit-norm axis connecting the origin of the **Camera Frame** and the **Position of the Object** ${}^cP_o$. The matrix J_c is the **Camera Jacobian**, while $L(s)$ maps Linear/Angular Velocities of the camera to changes in s.

$$\begin{bmatrix} -\frac{1}{||{}^cP_o||}(I - ss^T) & S(s) \end{bmatrix} R \in \mathcal{R}^{3x6}$$

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

where $S(.)$ is the **Skew-Symmetric Operator**, R_c the current **Camera Rotation Matrix**. Finally $N(I - (LJ)^{\dagger}LJ$ is the matrix spanning the **Null-Space** of the LJ amtrix, in order to avoid strange joints movement of the manipulator during the tracking. Now the **ros2_kdl_vision_control.cpp** file (only some important parts)

```
class VisionBasedControlNode : public rclcpp::Node
{
    public:
        VisionBasedControlNode()
        : Node("ros2_kdl_vision_control"),
        node_handle_(std::shared_ptr<VisionBasedControlNode>(this))
        {
            // declare task to be executed (positioning, look-at-
               point)
            declare_parameter("task", "positioning"); // defaults to
               "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;
            }

...
...
...

            // Subscriber to aruco
            arucoSubscriber_ = this->create_subscription<
               geometry_msgs::msg::PoseStamped>(
                "/aruco_single/pose", 10, std::bind(&
                   VisionBasedControlNode::aruco_subscriber, this,
                   std::placeholders::_1));

            // Wait for the joint_state topic
            while(!aruco_available_){
                RCLCPP_INFO(this->get_logger(), "No ArUco data
                   received yet! ...");
                rclcpp::spin_some(node_handle_);
            }
...
```

```
...
...

        //Aruco to Base Frame with a Position Offset and an
           Orientation Offset
        KDL::Frame cam_T_object_offset(aruco_frame_.M*KDL::
           Rotation::RotY(-0.2),
                   KDL::Vector(aruco_frame_.p.data[0]-0.05,
                      aruco_frame_.p.data[1], aruco_frame_.p.
                      data[2] - 0.4));

        base_T_object = robot_->getEEFrame() *
           cam_T_object_offset;

...
...
...

        // Create cmd publisher
        if(task_=="positioning"){
            cmdPublisher_ = this->create_publisher<FloatArray>("/
               velocity_controller/commands", 10);
            timer_ = this->create_wall_timer(std::chrono::
               milliseconds(100),
                                        std::bind(&
                                           VisionBasedControlNode
                                           ::
                                           cmd_publisher_positioning
                                           , this));

            // Send joint velocity commands
            for (long int i = 0; i < joint_velocities_.data.size
                (); ++i) {
                desired_commands_[i] = joint_velocities_(i);
            }

            RCLCPP_INFO(this->get_logger(), "Starting␣Positioning
               ␣with␣Velocity␣Controller...␣\n");
        }

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

            cmdPublisher_ = this->create_publisher<FloatArray>("/
               velocity_controller/commands", 10);
            timer_ = this->create_wall_timer(std::chrono::
               milliseconds(100),
                                        std::bind(&
                                           VisionBasedControlNode
                                           ::
                                           cmd_publisher_look_at_point
                                           , this));

            // Send joint velocity commands
            for (long int i = 0; i < joint_velocities_.data.size
                (); ++i) {
                desired_commands_[i] = joint_velocities_(i);
```

```
                }

                RCLCPP_INFO(this->get_logger(), "Starting␣look-at-
                    point␣with␣Velocity␣Controller...␣\n");

            }

            // Create msg and publish
            std_msgs::msg::Float64MultiArray cmd_msg;
            cmd_msg.data = desired_commands_;
            cmdPublisher_->publish(cmd_msg);

        }

    private:

        void cmd_publisher_positioning(){

...
...
...

                Vector6d cartvel; cartvel << 0.02*p.vel + 5*error,
                    0.05*o_error;
                joint_velocities_.data = pseudoinverse(robot_->
                    getEEJacobian().data)*cartvel;
                joint_positions_.data = joint_positions_.data +
                    joint_velocities_.data*dt;

                // Update KDLrobot structure
                robot_->update(toStdVector(joint_positions_.data),
                    toStdVector(joint_velocities_.data));

                // Send joint velocity commands
                for (long int i = 0; i < joint_velocities_.data.size
                    (); ++i) {
                     desired_commands_[i] = joint_velocities_(i);
                }

                // Create msg and publish
                std_msgs::msg::Float64MultiArray cmd_msg;
                cmd_msg.data = desired_commands_;
                cmdPublisher_->publish(cmd_msg);

            }

            else{
                RCLCPP_INFO_ONCE(this->get_logger(), "Positioning␣
                    Task␣Executed␣Successfully␣...␣\n");
                for (long int i = 0; i < joint_velocities_.data.size
                    (); ++i) {
                     desired_commands_[i] = 0.0;
                }

                // Create msg and publish
                std_msgs::msg::Float64MultiArray cmd_msg;
```

```
        cmd_msg.data = desired_commands_;
        cmdPublisher_->publish(cmd_msg);
    }

}


void cmd_publisher_look_at_point(){

    // Publisher for look-at-point task

    //Object Frame
    cam_T_object = KDL::Frame(aruco_frame_.M, KDL::Vector(
       aruco_frame_.p.data[0], aruco_frame_.p.data[1],
       aruco_frame_.p.data[2]));

    Eigen::Matrix<double,3,1> cP_o = toEigen(cam_T_object.p);

    //Unit Norm Axis Connecting the Origin of the Camera
       Frame and the Position of the Object
    Eigen::Matrix<double,3,1> s = cP_o/cP_o.norm();

    //tool0 (End-Effector) Frame
    KDL::Frame base_T_tool0 = robot_->getEEFrame();

    //Rotation and Traslation of the Camera Specified in
       iiwa_description)/urdf/camera.xacro for the Camera
       Joint
    KDL::Frame tool0_T_cam(KDL::Rotation::RPY(3.14, -1.57,
       0.0), KDL::Vector(0.0, 0.0, 0.0));

    //Base to Camera Frame
    KDL::Frame base_T_cam = base_T_tool0 * tool0_T_cam;

    //Camera Rotation Matrix
    Eigen::Matrix<double,3,3> R_c = toEigen(base_T_cam.M);

    Eigen::Matrix<double,6,6> R_c_big = Eigen::Matrix<double
       ,6,6>::Zero();

    //Diagunal Matrix
    R_c_big.block(0,0,3,3) = R_c;
    R_c_big.block(3,3,3,3) = R_c;

    //L Matrix which maps Linear adn Angular Velocities of
       the Camera to changes in s
    Eigen::Matrix<double,3,3> L_block = (-1/cP_o.norm())*(
       Eigen::Matrix<double,3,3>::Identity() - s*s.transpose
       ());
    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());

    //Desired Value
```

```cpp
Eigen::Vector3d sd;
sd = Eigen::Vector3d(0,0,1);


KDL::Jacobian J_cam = robot_->getEEJacobian();

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

// Matrix spanning the Null-Space of LJ
Eigen::MatrixXd N = Eigen::Matrix<double,7,7>::Identity()
     - (LJ_pinv*LJ);

//s Error
double s_error=(sd-s).norm();

//A variable that allows me to stop the Joint Velocity
    Command when the
//manipulator loses sight of the ArUco tag during
    movement, before reaching the desired position
double s_error_old;

std::cout << "s Error: "<< s_error << std::endl;

for (long int i = 0; i < joint_velocities_.data.size();
    ++i) {
     std::cout << "Joint " << (i + 1) << " Velocity 
        Command: " << joint_velocities_(i) << std::endl;
}

//If error > 0.02 AND ArUco tag is available
if (s_error>0.02 && s_error!=s_error_old)
{
     joint_velocities_.data =2*LJ_pinv*sd + N*(q_in.data -
          joint_positions_.data);
     s_error_old=s_error;
}
//If error<=0.02 OR ArUco tag is not available
else if(s_error<=0.02 || s_error==s_error_old)
{
     for (long int i = 0; i < joint_velocities_.data.size
        (); ++i)
         joint_velocities_.data[i]=0;
         if (s_error<0.02) std::cout <<"Look-at-Point Task
             Successfully Executed" << std::endl;
}

for (long int i = 0; i < joint_velocities_.data.size();
    ++i) {
     desired_commands_[i] = joint_velocities_(i);
}

// Create msg and publish
std_msgs::msg::Float64MultiArray cmd_msg;
cmd_msg.data = desired_commands_;
```

```
            cmdPublisher_->publish(cmd_msg);

            }

...
...
...

        //Subscriber to Aruco Pose
        void aruco_subscriber(const geometry_msgs::msg::PoseStamped&
           aruco_pose_msg)
        {

            aruco_available_ = true;

            //Position
            aruco_x = aruco_pose_msg.pose.position.x,
            aruco_y = aruco_pose_msg.pose.position.y,
            aruco_z = aruco_pose_msg.pose.position.z;

            //Quaternion
            aruco_q1 = aruco_pose_msg.pose.orientation.x,
            aruco_q2 = aruco_pose_msg.pose.orientation.y,
            aruco_q3 = aruco_pose_msg.pose.orientation.z,
            aruco_q4 = aruco_pose_msg.pose.orientation.w;

            KDL::Rotation rot_= KDL::Rotation::Quaternion(aruco_q1,
               aruco_q2,aruco_q3,aruco_q4);
            KDL::Vector trasl_(aruco_x,aruco_y,aruco_z);

            aruco_frame_.p = trasl_;
            aruco_frame_.M = rot_;

        }

...
...
...
```

The instructions to run are several

```
ros2 launch iiwa_bringup iiwa.launch.py

command_interface:="velocity"

robot_controller:="velocity_controller"

use_sim:=true use_vision:=true


ros2 launch aruco_ros single.launch.py
```

```
ros2 run rqt_image_view rqt_image_view
```

```
ros2 run ros2_kdl_package ros2_kdl_node_vision_control
```

After that the Positioning is completed (wait the message **"Positioning Task Executed Successfully ...**) it is possible to run in this last terminal, after pressing **ctrl+C**, the final instruction

```
ros2 run ros2_kdl_package ros2_kdl_node_vision_control
--ros-args -p task:=look-at-point
```

Now it is possible to move the **ArUco tag** with the realtive interface.

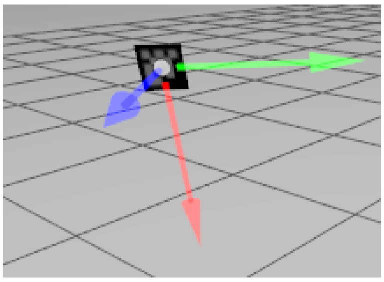

Figure 12: Moving ArUco tag

If during movement the manipulator loses sight of the tag, a mechanism has been implemented to stop the robot, in order to prevent any erratic movements. Here a video in which is shown the implementation of both tasks with the Velocity Controller

`https://youtu.be/li2pN2s-sjc`

In the video were also reporting, on the right, the **Joint Velocity Commands** during the all simulation. It can be observed that at the end of each trajectory, the center of the Camera always aligns with the center of the Tag (regardless of the orientation).

Vision-Based Effort Controller

In the same way as the previous chapter, a **Vision-Based Effort Controller** was implemented. The world is the same as before. This was achieved by tracking the velocities generated by the Control Law presented in the previous section, using both the **Joint Space** and the **Operational Space Inverse Dynamics Controller** developed in the previous Homework. By combining the effort controller studied earlier with the one developed in the Homework 2 for Linear Trajectory Tracking, the goal is to ensure that the robot follows a linear trajectory while continuously keeping its focus on the **ArUco Tag**.
This was implemented by replacing the **Orientation Error** with respect to a fixed frame with the error generated by the Vision-Based Look-at-Point Controller. Specifically, the variables s and s_d were used to represent the error between the **Camera Center** and the center of the **ArUco Tag**. Due to the low performance of my PC, the videos were recorded using a mobile phone, as a different behavior would have occurred with screen recording.
Linear Trajectory Looking at the ArUco Tag with Joint Space Inverse Dynamic Controller

`https://youtu.be/KYrDLQJmb4o`

Positioning with Joint Space Inverse Dynamics Controller

`https://youtu.be/rCUsp9KdgcQ`

Due to lack of time, the appropriate tests with the inverse dynamics controller in the operational space were not conducted. However, the behavior is expected to be similar with an appropriate choice of parameters.

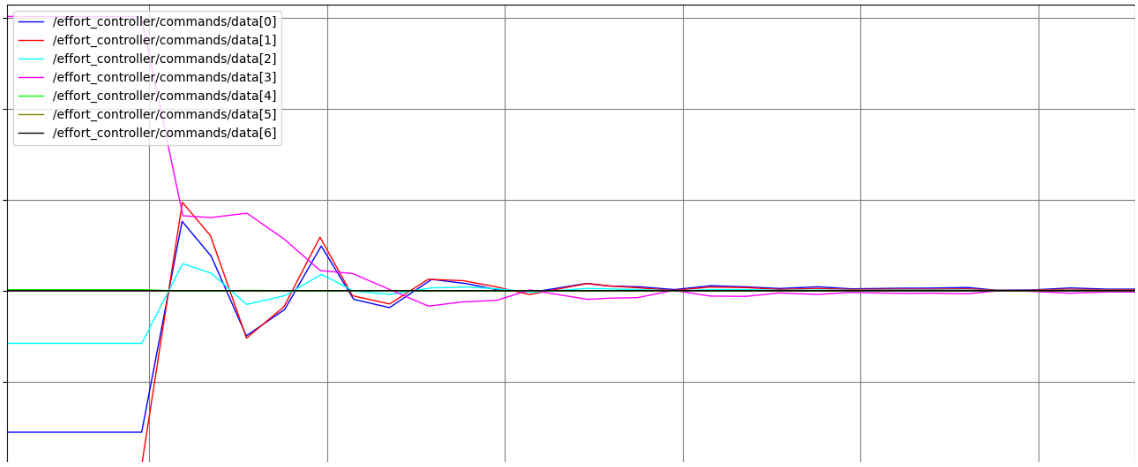

Figure 13: Effort Commands for Positioning

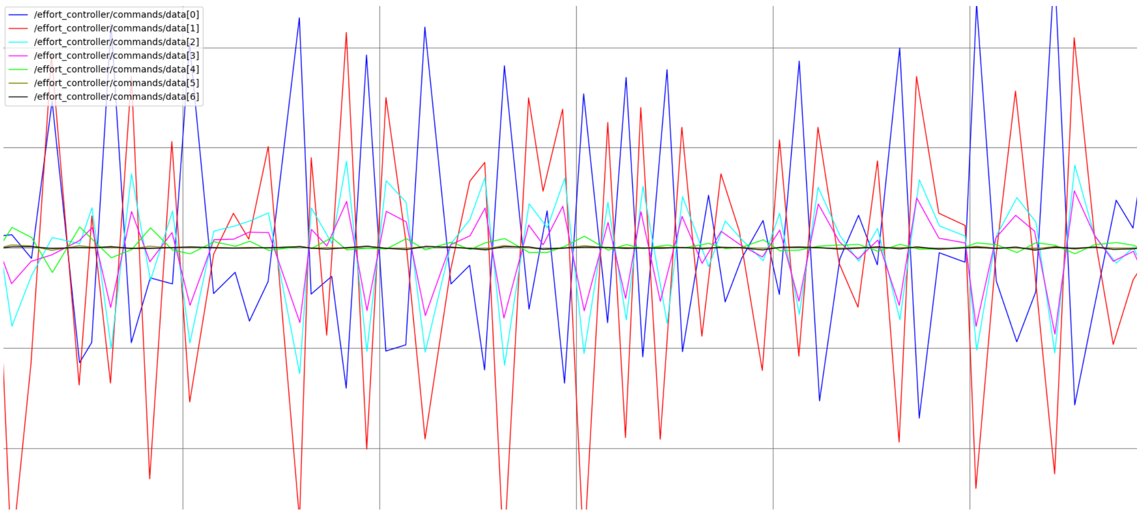

Figure 14: Effort Commands for Look-at-Point