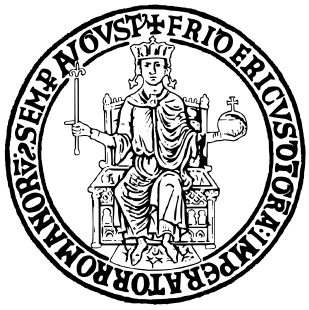
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Robotics Lab

Homework 3: Vision Based Task

Components
William Notaro
Chiara Panagrosso
Salvatore Piccolo
Roberto Rocco

Anno Accademico 2024–2025

Contents

The goal of this homework is to develop a ROS package to control a 7-degrees-of-freedom robotic manipulator arm into the Gazebo environment, using visual-based features. Starting from the recommended packages, the assigned tasks involve several steps. The first task requires correctly importing and identifying a blue sphere model in `Gazebo`. Subsequently, in a `Gazebo` world containing an `ArucoTag`, it is necessary to position the robot in a specific configuration relative to the tag and implement the tracking of the `ArucoTag` using both a velocity and effort based controllers. Finally, the robot must execute predefined trajectories while continuously observing the `ArucoTag`.

The code used for this project can be viewed and downloaded from the following GitHub repositories:

- William Notaro: `https://github.com/well-iam/RL24-Homework3`
- Chiara Panagrosso: `https://github.com/chiarapanagrosso/RL24-Homework3`
- Salvatore Piccolo: `https://github.com/SalvatorePiccolo/RL24_Homework3`
- Roberto Rocco:`https://github.com/Robertorocco/RL24-Homework3`

It is possible to view the videos of the simulation results of the various tasks on YouTube at the following links:

- Positioning: `https://youtu.be/iXu5GMZnYhI`
- Look-At-Point with velocity commands: `https://youtu.be/4gVAl3z1ZCI`
- Look-At-Point with effort commands in the joint space: `https://youtu.be/4Sj26DFo13Y`
- Look-At-Point with effort commands in the operational space:`https://youtu.be/hwt9C7kOoH8`
- Look-At-Point with effort commands in the operational space while following a linear trajectory:`https://youtu.be/Stkjby8hntU`

Chapter 1

Object Detection

In the first chapter, the task involves building a blue sphere model and correctly importing it inside a simulation environment in which we run `Gazebo`. Subsequently, a camera will be added to the robot's end effector, ensuring that it remains rigidly attached to the manipulator. Finally, leveraging the `OpenCV` package, the sphere will be accurately detected within the simulation by appropriately configuring the recognition features.

1.1 Sphere's Gazebo Model

The addition of a spherical object within the simulation in `Gazebo` requires the creation of 2 files, a `model.config` file and a `model.sdf` file, which should be placed inside a folder named after the model that we want to import into `Gazebo`. The `model.config` file is needed to provide metadata and configuration details about the model, while `model.sdf` defines the physical properties and structure of the model itself.

```
1
2  <?xml version="1.0" ?>
3  <sdf version="1.6">
4    <model name="spherical_object">
5      <!--<static>true</static> -->
6      <link name="link">
7        <visual name="visual">
8          <geometry>
9            <sphere>
10             <radius>0.15</radius>
11           </sphere>
12         </geometry>
13         <material>
14           <ambient>0 0 1 1</ambient>
15           <diffuse>0 0 1 1</diffuse>
16         </material>
17       </visual>
18       <collision name="collision">
19         <geometry>
20           <sphere>
21             <radius>0.15</radius>
22           </sphere>
23         </geometry>
24       </collision>
25     </link>
26   </model>
```

```
</sdf>
```

Listing 1.1: `sphere_model.sdf`

1.2 Sphere's Gazebo World

Once the two files for the model have been created, the model must be imported into the `empty.world` file used to start the simulation, in order to spawn the object inside the environment. To do this, you need to specify the URL of the model inside the `<include>` tag. `model` is a variable that contains the path where we will place all the models we want to add to Gazebo Simulation, so we specify the name `spherical_object` of the particular object that we want to import. This name must match the folder name in which we put the `model.config` and `model.sdf`, that is located in the path specified inside `model` variable.

```
<?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>

    <!-- Included spherical object -->
    <include>
      <uri>
        model://spherical_object
      </uri>
      <name>spherical_object</name>
      <pose>1 -0.5 0.6 0 0 0</pose>
    </include>
    <!-- Our Modifications ^^^^^ -->
    <gravity>0 0 0</gravity>
    ...
</sdf>
```

Listing 1.2: `sphere.world`

1.3 Camera Setting

The project requires the addition of the camera to the robot structure only through the setting of a `use_vision` flag. Inside the `iiwa.urdf.xacro` file of the manipulator in use, a conditional `<xacro>` has been added, meaning that the URDF includes the content of this file based on the value of the flag. The value of the `use_vision` argument is specified from the command line, so inside the configuration file `iiwa.config.xacro`, it is specified that the value of this parameter is passed as an argument and is set to false by default.

If `use_vision` is set to true, a camera is mounted on the manipulator's end-effector. An additional link called `camera_optical` is added, and this represents the frame of the image view by the camera.

```
1
2  <xacro:property name="camera_xacro" value="${use_vision}"/>
3
4  <xacro:if value="${use_vision}">
5
6    <joint name="${prefix}joint_camera" type="fixed">
7      <origin xyz="0.0 0.0 0.0" rpy="0.0 -1.57 3.14"/>
8      <parent link="${prefix}tool0"/>
9      <child link="${prefix}tool_camera"/>
10   </joint>
11
12
13   <link name="${prefix}tool_camera">
14     <visual>
15       <geometry>
16       <box size="0.02 0.008 0.008"/>
17       <box size="1 1 1"/>
18       </geometry>
19       <origin rpy="0 0 0" xyz="0 0 0"/>
20       <material name="green"/>
21     </visual>
22   </link>
23
24   <joint name="${prefix}joint_camera_optical" type="fixed">
25     <origin rpy="${-pi/2} 0 ${-pi/2}" xyz="0.0 0.0 0.0" />
26     <parent link="${prefix}tool_camera"/>
27     <child link="${prefix}camera_optical"/>
28   </joint>
29   <link name="${prefix}camera_optical"></link>
30
31   <gazebo>
32       <plugin filename="gz-sim-sensors-system" name="
      gz::sim::systems::Sensors">
33         <render_engine>ogre2</render_engine>
34       </plugin>
35   </gazebo>
36
37   <gazebo reference="tool_camera">
38     <sensor name="camera" type="camera">
39       <camera>
40       <horizontal_fov>1.047</horizontal_fov>
41         <image>
42           <width>320</width>
43           <height>240</height>
44         </image>
45       <clip>
46         <near>0.1</near>
47         <far>100</far>
48       </clip>
49       </camera>
50       <always_on>1</always_on>
51       <update_rate>30</update_rate>
52       <visualize>true</visualize>
53       <topic>camera</topic>
54     </sensor>
55   </gazebo>
56   </xacro:if>
57
```

```
58  </xacro:macro>
```

Listing 1.3: `iiwa.urds.xacro` - Conditional Xacro for Camera

1.4 Detection with OpenCV

Once the world is set up with the blue sphere inside and the robot positioned such that the camera can see the ball, the goal is to detect the object within the image. To do this, a source file `ros2_opencv_node.cpp` has been implemented, based on the implementation of the `ros2_opencv` package. The job of this node involves the creation of a subscriber to the topic `/videocamera`, where images from the camera mounted on the end-effector are published. Once the image data is received, the image is processed through the functions of the `"opencv2/opencv.hpp"` library. By setting certain parameters, `keypoints`, i.e. pixels, are detected within the image, which are the contour points that contain the portion of the image matching the specifications defined by the parameters. These parameters pertain to features such as the shape, size, circularity, and convexity of the area of the image we want to detect and are set so that the recognition of a blue sphere within the image can occur. Once we retrieve the `keypoints` using the `detect` method on the `detector` object set as previously specified, the camera image is reprocessed with the addition of red-colored `keypoints`. The image is then republished on a new topic, `/processed_image`, where it is possible to view the same camera image with the blue sphere detection highlighted.

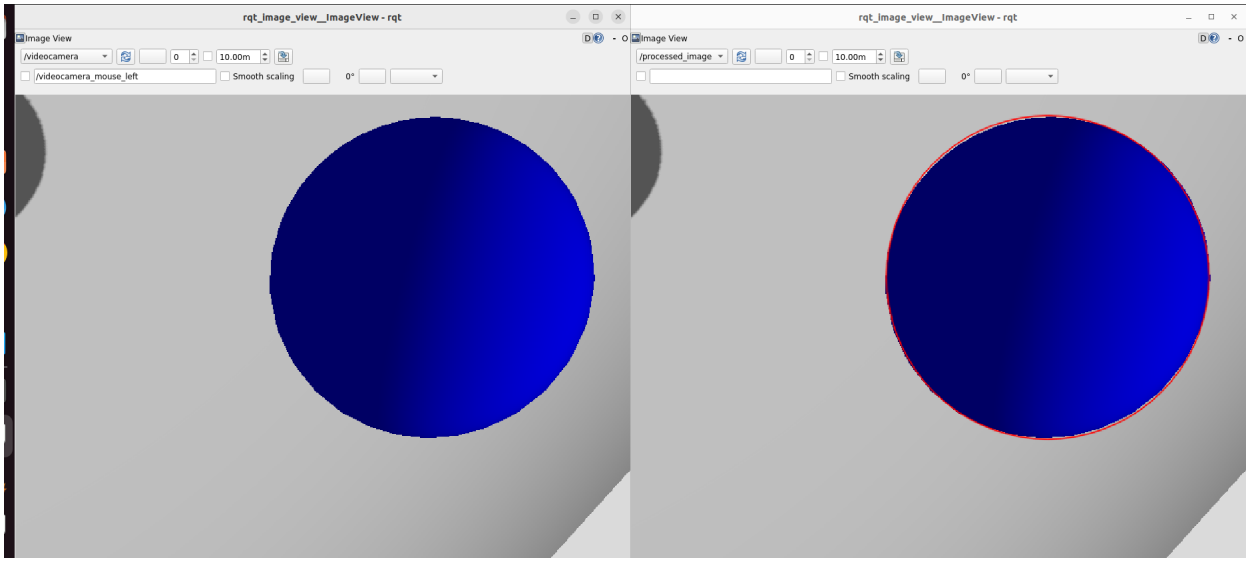


Figure 1.1: Image Comparison. Left: Camera Image, Right: Processed Image

1.5 Blue Sphere Detection Launch File

To easily demonstrate the functionality of image detection, a launch file called `sphere_detect` was created that starts the simulation of the robot in Gazebo and its visualization in RViz, allowing the camera image to be viewed as well. Within the launch file, an additional executable, `rqt_graph_view`, is launched, enabling the visualization of the processed image published on the `/processed_image` topic.

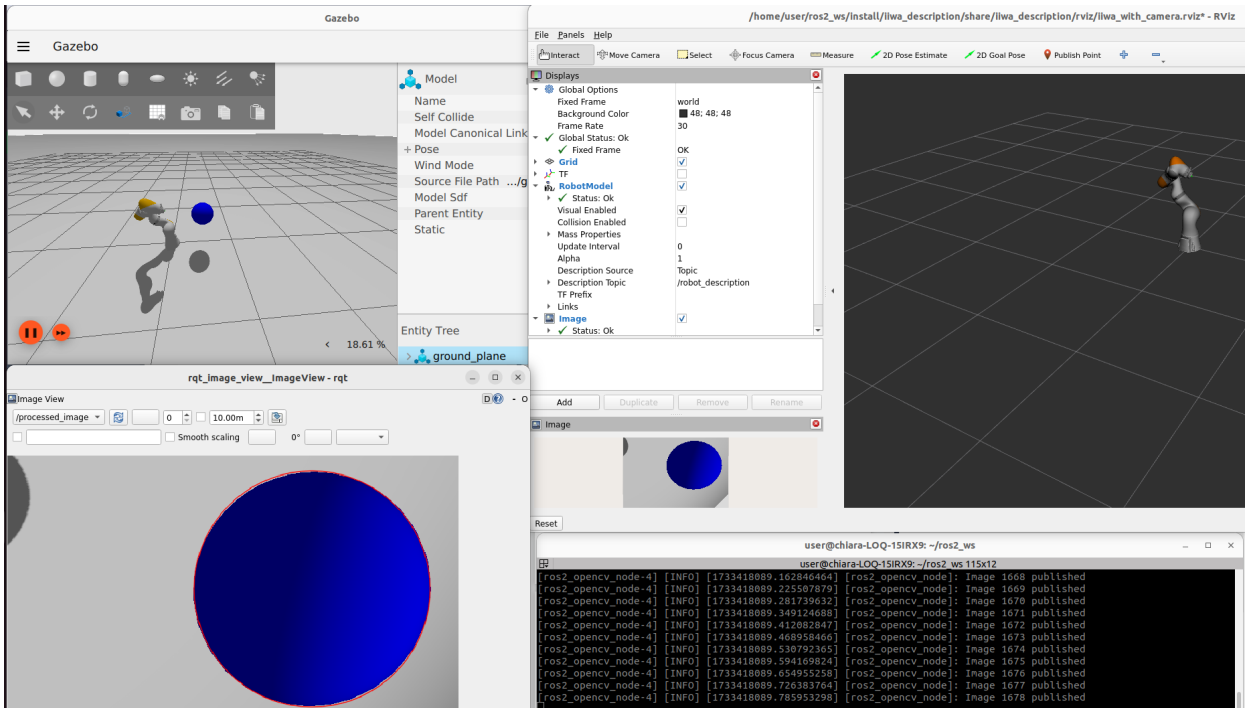


Figure 1.2: ROS2 Open CV Package Image Detection

Chapter 2

Vision-Based Controller

In the second chapter, the aim is to implement two vision based controllers that carry out two different tasks: the first controller is aimed at executing the task of positioning the end-effector at a specific point in space with a given orientation both relative to an Aruco Marker, whose absolute position is determined through a camera placed on the end-effector itself. The second controller, on the other hand, uses the position of the same Aruco Marker relative to the camera image frame with the goal of positioning the end-effector in such way that the marker is always exactly at the center of the image at an arbitrary distance (within the visibility limits of the camera's object), thus tracking its movement.

2.1 Positioning Task

Initially, the task requires actuating the robot to reach a specified pose. The detailed steps for the correct execution of this task are as follows:

- Identify the position of the `Aruco Marker` within the scene using the camera's view.
- Determine the reference frame of the `Aruco Marker` relative to the `World Frame` in terms of position and orientation.
- Receive as input an offset, expressed relative to the `Aruco Marker`, in terms of position (`x`, `y`, `z`) and orientation (`r`, `p`, `y`).
- Transform the input offset into the `World Frame` to compute the final point of the trajectory.
- Assign the computed point from the previous step as goal for the (linear) planner.
- Verify the error and ensure the camera is correctly aligned with the `Aruco Marker`.

This steps will be analyzed in the following.

2.1.1 Aruco Identification

An Aruco Marker is a black-and-white square marker with a unique ID encoded in its pattern. It is commonly used in computer vision for camera calibration, object tracking, and pose estimation due to its high detection accuracy and simplicity. Web provides a lot of Aruco Marker, which can be generated online given an ID and a dimension. Its possible to read this value from the `config.sdf` file that describes it: we're using and Aruco with

size 0.1x0.1 meters, with an ID value equal to 201. The position of the Aruco Marker with respect to the camera lens' frame is taken from the topic aruco/single/position and is passed to the function. The features parameters are defined by the formula provided by the homework's demands. The identification of the `Aruco Marker` is handled by the `aruco_pos` package, which in its `single` version allows for detecting a single `Aruco` in the scene. The marker parameters used to detect the Aruco are those specified earlier. The only required modification was connecting the simulated camera input in `Gazebo` to the `Aruco`, entirely through the bridge responsible for creating a `ROS` topic that enables video visualization of what the end-effector's camera perceives.

The correct functionality can be verified using `rqt_image_view`, a node that allows users to view video streams from topics published by `ROS`. By utilizing `simple_single.cpp`, `aruco` provides several useful topics for processing. Among these, the `/aruco_single/result` topic allows visualizing the `Aruco`'s identification within the scene.

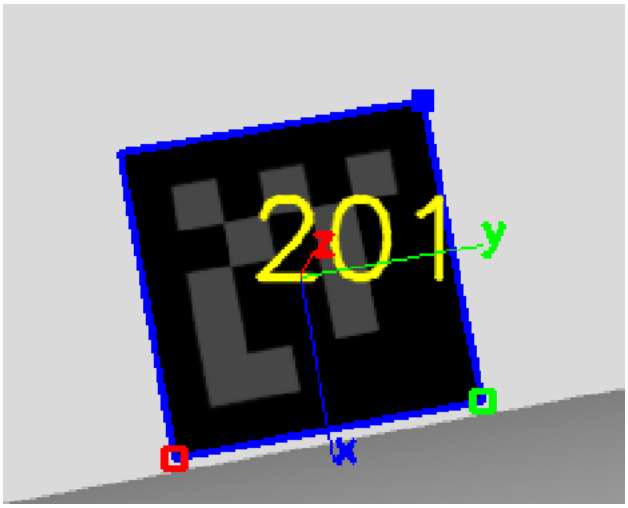


Figure 2.1: Aruco Identification

2.1.2 Aruco position in World Reference

To spawn the Aruco inside the simulated world its required to include the model inside the world file. Here is specified the pose of the marker with respect to the world frame. Its important that we position the manipulator with initial condition such that the `Aruco Marker` is in the camera's field of view, thus its position becomes accessible on the `/aruco_pose` topic. The reference frame in which the position is expressed can be selected by modifying the `reference_frame` parameter, which is accepted as input by the launch file of the `aruco_ros` package. By setting it to `'world'`, it gives us the absolute pose of the Aruco, i.e. the coordinates we put on the `empty.world`. The choice of the reference frame depends on the task to be executed: for the `'positioning'` task, the reference frame is set to the `world`, while for the `'look-at-point'` task, the reference frame is set to `camera_optical`, meaning that on the topic the pose published is referenced to the image that we see via the camera we put on the end-effector.

2.1.3 Final point computation

Regarding the functioning of the controller, the goal is to position the end-effector relative to the ArUco marker. The controller uses as references a position and orientation expressed

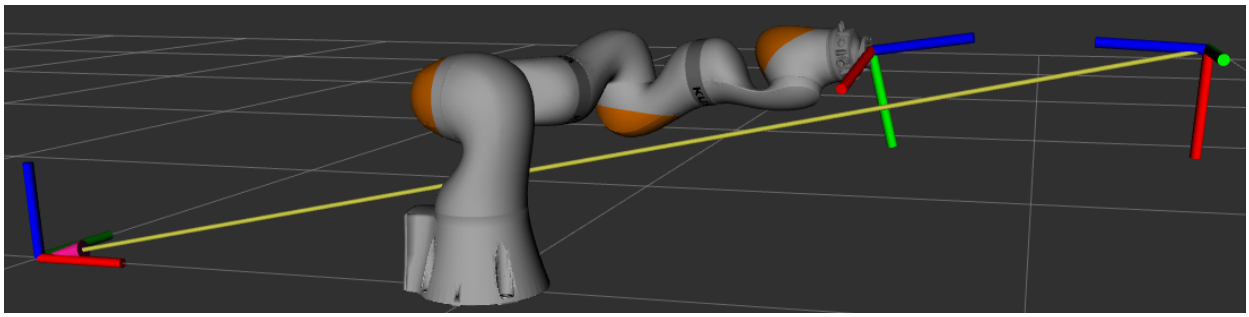


Figure 2.2: tool_camera frame and Aruco's frame using world as reference

in the ArUco marker's frame, which need to be transformed into the world frame.

Once the orientation and position of the marker in the world frame are known—accessible to the controller through a subscription to the topic `/aruco/single/pose`—the controller calculates the desired final pose of the end-effector frame in the world using the rotation matrix and position of the ArUco marker. This is done through the callback function of the subscription `aruco_position`. Here's a snippet of the code:

```
void aruco_position(const geometry_msgs::msg::PoseStamped::SharedPtr msg)
{
    if(!aruco_pose_available_){
    if(1){
    // Retrieve position DEBUG
    RCLCPP_INFO(this->get_logger(), "Position: [x: %f, y: %f, z: %f]",
                msg->pose.position.x, msg->pose.position.y, msg->pose.
   position.z);

    // Retrieve orientation (as quaternion) DEBUG
    RCLCPP_INFO(this->get_logger(), "Orientation: [x: %f, y: %f, z: %f, w:
    %f]",
                msg->pose.orientation.x, msg->pose.orientation.y,
                msg->pose.orientation.z, msg->pose.orientation.w);
    }


    tf2::Quaternion aruco_quat(
        msg->pose.orientation.x,
        msg->pose.orientation.y,
        msg->pose.orientation.z,
        msg->pose.orientation.w);

    aruco_quat.normalize();

    //Retrive Offset from Keyb:
    std::vector<float> input_pose(6);

    std::cout << "Enter position offset (x, y, z): ";
    for (int i = 0; i < 3; ++i)
    {
        std::cin >> input_pose[i];
    }

    std::cout << "Enter orientation offset (roll, pitch, yaw in radians):
   ";
    for (int i = 3; i < 6; ++i)
    {
```

```
36          std::cin >> input_pose[i];
37      }
38
39       // Step 1: Final orientation
40      tf2::Quaternion input_quat;
41      input_quat.setRPY(input_pose[3], input_pose[4], input_pose[5]); //
        Convert user input (Euler angles) to a quaternion
42      input_quat.normalize();
43
44
45      //Orientation
46      KDL::Rotation input_rot_matrix;
47      input_rot_matrix = KDL::Rotation::Quaternion(input_quat.x(),
48                                                   input_quat.y(),
49                                                   input_quat.z(),
50                                                   input_quat.w());
51
52      KDL::Rotation aruco_rot_matrix;
53      aruco_rot_matrix = KDL::Rotation::Quaternion(aruco_quat.x(),
54                                                   aruco_quat.y(),
55                                                   aruco_quat.z(),
56                                                   aruco_quat.w());
57
58      end_orientation_wrt_world_frame = aruco_rot_matrix*input_rot_matrix;
59
60      //Express input_pose in camera ref
61      // Create an KDL Vect from the std::vector
62      KDL::Vector input_pose_kdl(input_pose[0], input_pose[1], input_pose
        [2]);
63
64      KDL::Vector input_pose_wrt_world_frame;
65
66      input_pose_wrt_world_frame=aruco_rot_matrix*input_pose_kdl;
67
68      std::cout<<"Input pose wrt world frame:  "<<std::endl<<
        input_pose_wrt_world_frame<<std::endl;
69
70      //Step 2: Final position
71      end_position_wrt_world_frame<<
72          msg->pose.position.x + input_pose_wrt_world_frame[0],
73          msg->pose.position.y + input_pose_wrt_world_frame[1],
74          msg->pose.position.z + input_pose_wrt_world_frame[2];
75       aruco_pose_available_=true;
76      }
77  }
```

Listing 2.1: `aruco_position` callback function

The `aruco_pose_available_` is a flag used to ensure that this callback is executed just one time, the first time that the ArUco Pose is being published on the topic.

After determining the desired pose of the end-effector, the controller then computes the errors between the desired and actual pose to drive the control actions. The controller computes velocity commands that bring the end-effector to the final position following a linear trajectory with cubic curvilinear abscissa, and publishes them on the `/velocity_controller/commands`. The implementation of the controller is totally analogous to the velocity controller already implemented in the previous homework, and gives good performance in executing the desired task

2.2 Look-at-point Task

The look-at-point task's objective is to ensure good tracking of the Aruco Marker.

The control law used aims to move the end-effector in a way that the camera mounted on it always centers the Aruco Marker on the image frame, with an arbitrary distance from the camera. This means bringing the features parameters s, the camera frame unit vector that points to the Aruco Marker towards the desired features parameters sd, that is `s_d = [0, 0, 1]`.

2.2.1 Frames placement

As previously mentioned, to achieve tracking of the `Aruco Marker`, a specific frame named `camera_optical` was created, whose orientation is shown in `Rviz` in the figure. The goal was to define an additional frame (not linked to the `robot_` object) with its `z`-axis oriented towards the camera's lens. This frame is set as default by us for the `simple_single_launchpy` to use it as frame to interpret where is the ArUco Marker inside the image that we are publishing on the `/videocamera` topic. Its important to stress that this frame is in no way related to how the robot sees the aruco, that is instead related to the frame put on the `tool_camera` link inside the URDF. By default the camera is oriented in a way that it sees along the x-axis direction. As it can be seen in `Rviz`, if we don't specify any reference frame when launching the `simple_single_launchpy`, the position of the ArUco will be relative to the `camera_optical` frame located on the image the camera publishes. Therefore, the coordinates published on the `/aruco_single/position` topic will refer to this frame.

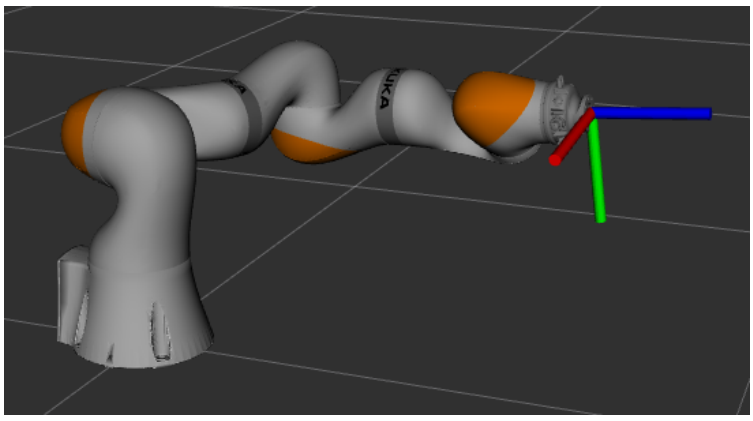


Figure 2.3: `camera_optical` frame positioning and orientation

2.2.2 Control

Once the frames are positioned as described, the system is ready to perform tracking of the `Aruco Marker`'s movements. A velocity controller is assigned, with the objective of driving `s`, a variable defined as the normalized position of the `Aruco Marker` relative to the camera, to a desired value `s_d = [0, 0, 1]`. This corresponds to ensuring that the camera consistently views the `Aruco Marker` at the center of its image, regardless of the marker's orientation or distance from the camera. In order to achieve what said, the following control law was developed:

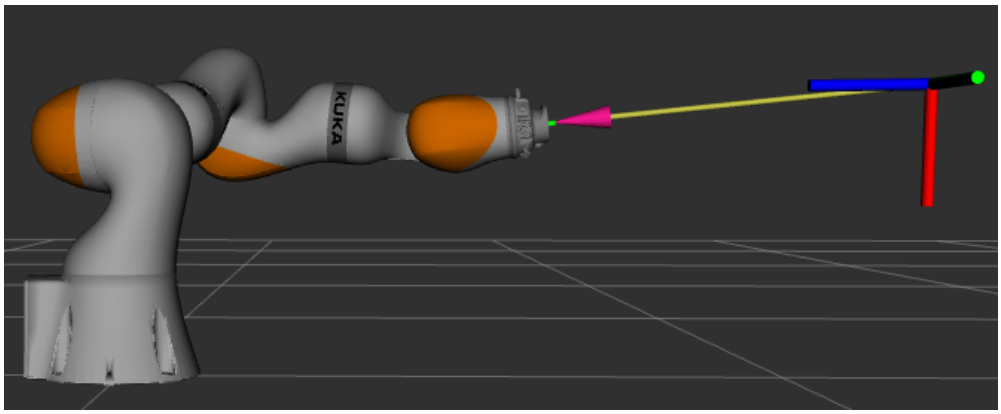


Figure 2.4: End Effector position expressed in `camera_optical` frame

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

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

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

that is the unit-norm vector connecting the origin of the camera frame and the position of the object c_{P_o}. The matrix J_c is the camera Jacobian, while $L(s)$(to be computed) maps linear/angular velocities of the camera to changes in s., so what the camera sees. The R is the rotation matrix from the world reference to the frame attached to the `tool_camera`.

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

To implement the previous control law, a function called `look_at_point_control()` was added inside the `#kdl_control.h` and its implementation follows the steps to compute $\dot{q}$, given as input the ArUco Position relative to the camera frame. The function computes from the imput also the value the interaction matrix L that depends on the features parameters, computed from the input by dividing it by its norm. Jacobian and Rotation matrix are instead obtained from the `robot_` variable cointaining all information about the manipulator structure and its frames, via `robot_- getEEJacobian()` and `robot_- getEEFrame()` methods called on that object Since the interaction matrix is low rectangular, in order to invert it the pseudoinverse must be used. The controller's implementation starts with getting the camera Jacobian and the rotation matrix of the camera with respect to the base frame of the robot manipulator.

```
Eigen::VectorXd look_at_point_control(const Eigen::Vector3d &cPo)
{
    double k;
    // Get EE(=camera) Jacobian
    Eigen::Matrix<double,6,7> Jc = robot_->getEEJacobian().data;

    // Get Camera rotation matrix
    Eigen::Matrix3d Rc = toEigen(robot_->getEEFrame().M);

    // Ensure the object is not at the camera origin to avoid division by zero
    double norm_cPo = cPo.norm();
        if (norm_cPo < 1e-6) {
            throw std::runtime_error("cPo is too close to the camera origin!");
        }
    // Compute s = cPo / ||cPo||
    Eigen::Vector3d s = cPo / norm_cPo;

    // Desired s_d = [0, 0, 1]
    Eigen::Vector3d s_d(0, 0, 1);

    // Compute L(s)
    Eigen::Matrix3d I = Eigen::Matrix3d::Identity();
    Eigen::Matrix3d S_s = skew(s);
    Eigen::Matrix3d L_linear = -1.0 / norm_cPo * (I - s * s.transpose()) * Rc;

    Eigen::MatrixXd L(3, 6);
    L.block<3, 3>(0, 0) = L_linear;
    L.block<3, 3>(0, 3) = S_s * Rc;

    // Compute the pseudoinverse of L * Jc
    Eigen::MatrixXd LJ = L * Jc;
    Eigen::MatrixXd LJ_pinv = LJ.completeOrthogonalDecomposition().pseudoInverse();
    }
    // Compute q_dot
    Eigen::VectorXd q_dot = k * LJ_pinv * s_d;

    return q_dot;
}
```

As for the positiong task, to employ the controller,it is required to have access to the position of the ArUco relative to the camera. This means that the the node responsible for calling the controller must also subscribe to the `aruco_single/ position` topic.

```
void aruco_subscriber(const geometry_msgs::msg::Vector3Stamped& sensor_msg){
            std::cout << "\nARUCO SUBSCRIBER TRIGGERED\n";
            aruco_pose_available_ = true;
            cPo[0] = sensor_msg.vector.x;
            cPo[1] = sensor_msg.vector.y;
            cPo[2] = sensor_msg.vector.z;
            std::cout << cPo << std::endl;

            std::string reference_frame = sensor_msg.header.frame_id;
            RCLCPP_INFO(this->get_logger(), "Pose is referenced to frame: %s", reference_frame.c_str());
        }
}
```

Listing 2.2: look-at-point callback on the /aruco_single/position

Notice that differently from the previous controller, the subscription is made only to the topic concerning the position of the ArUco marker, as the controller does not act on how the ArUco marker is oriented within the image but rather solely on its position within it. Another difference fron the previos subscription callback is that this is not just called once, but it is called everytime the aruco position is published on that topic. This is trivial to ensure that the s parameters feature is kept to the desired value and thus the controller is always tracking the Aruco Marker.

The presented algorithm performs well, with the tuning of the parameter `k` carried out based on several simulation runs. The goal was to find a sufficiently high value to ensure rapid convergence while avoiding overly fast oscillations. The performance of the algorithm is demonstrated in the video linked at the beginning of this report.

2.2.3 Plot

To report the velocity commands sent to the robot, we used `rqt_plot` to visualize the data published on the velocity command topic. By running `rqt_plot`, we selected the specific fields of the velocity command message to plot them over time. This provided a graphical representation of how the commands evolved during the execution of the task, which was recorded and included in the report to demonstrate the controller's performance.

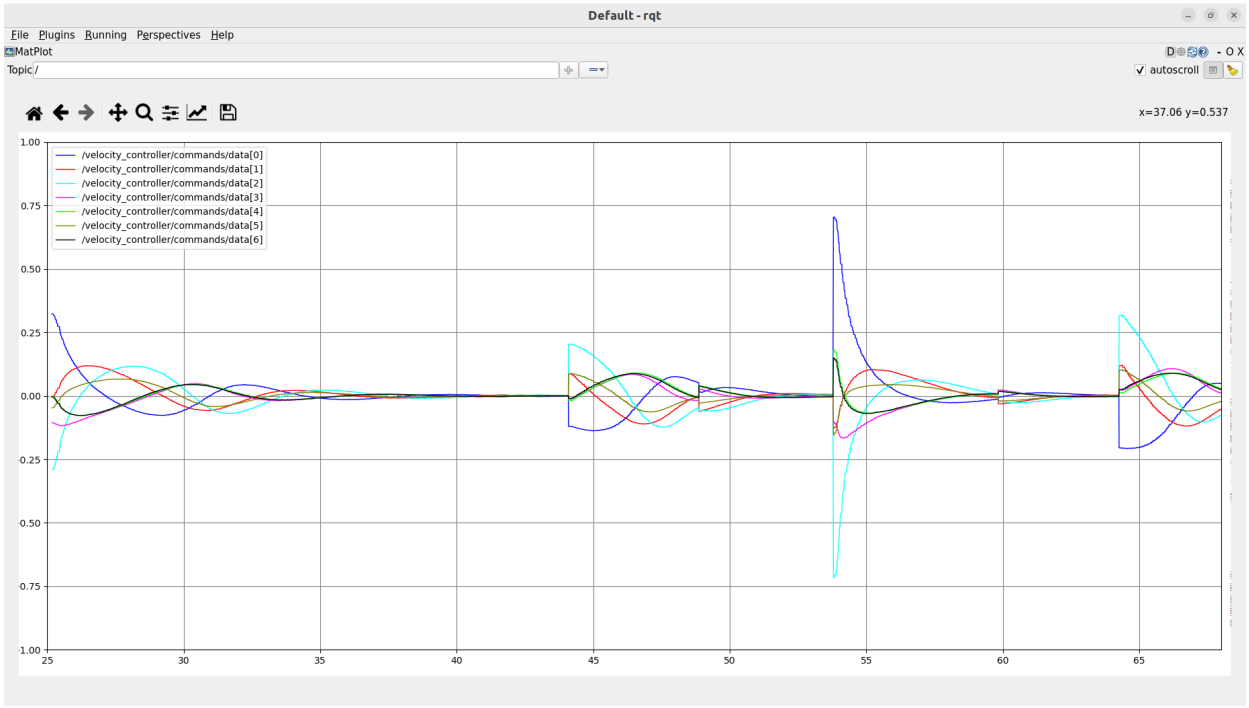


Figure 2.5: `velocity_control` Input

Chapter 3

Dynamic Vision-Based Controller

The final chapter focuses on the implementation of the dynamic version of the vision-based controllers. The general assumption is the use of the `q_dot` computed in the previous chapter, which ensures that the `Aruco Marker` remains at the center of the robot's camera view. Both `joint-space` and `operational-space` controllers will be discussed in the following sections. Finally, the task requires executing both a linear and a circular trajectory, ensuring that throughout the motion, the camera consistently keeps the `Aruco Marker` in its field of view.

3.1 Joint Space Controller

Here, it is required that the `look-at-point` task, previously executed with a velocity controller, be performed using a `joint-space` controller. Once `q_dot` is calculated as shown earlier, it is integrated and then differentiated to obtain `q` and `q_ddot`. Finally, these values are passed to the dynamic model, and the control law is computed.

Finally, the vector of desired commands will be retrieved into an appropriate variable in the main program and published by the `cmd_publisher` to the corresponding topic. The script is shown in Lst. 3.1.

```
1  else if(cmd_interface_ == "effort_joint"){
2
3      // Create cmd_publisher_look_at_point
4      //This is the topic for effort publication
5      cmdPublisher_ = this->create_publisher<FloatArray>("/effort_controller
       /commands", 10);
6      timer_ = this->create_wall_timer(std::chrono::milliseconds((int)(dt
       *1000)),
7                                       std::bind(&Iiwa_pub_sub::
       cmd_publisher_look_at_point, this));
8
9      // GET desired qdot
10     //LOOK-AT-POINT CONTROLLER
11     dqd_prev_ = joint_velocities_;
12     dqd_.data = controller_.look_at_point_control(cPo);
13
14     // GET desired q
15     //qd_prev_ = q;
16     qd_prev_ = joint_positions_;
17     qd_.data = qd_prev_.data + dqd_.data*dt;
18
19     // GET desired qddot
20     ddqd_.data.setZero();
```

```
21     //std::cout << "computed qddot: "; printJntArray(ddqd_, nj); std::cout
       << std::endl;
22
23     // Apply controller
24     desired_commands_ = toStdVector(controller_.idCntr(qd_, dqd_, ddqd_,
       Kp_, Kd_));
25     }
```

Listing 3.1: Joint Space's branch in kdl_control.cpp

3.2 Operational Space Controller

This code is designed to control the robot's end-effector by computing its position and velocity in space. It starts by using the robot's joint positions and velocities to calculate the end-effector's current pose and velocity through forward kinematics. Although the code defines the acceleration of the end-effector, this part is not being used. Finally, the Operational Space Controller is applied to calculate the appropriate joint commands, ensuring that the robot's end-effector reaches the desired position and velocity. The whole process is controlled using a feedback loop that adjusts the robot's movements accordingly.Lst. 3.3.

```
1  else if(cmd_interface_ == "effort_operational") {
2
3      // Create cmd publisher
4      //This is the topic for effort publication
5      cmdPublisher_ = this->create_publisher<FloatArray>("/
       effort_controller/commands", 10);
6      timer_ = this->create_wall_timer(std::chrono::milliseconds((int)(dt
       *1000)),
7                                       std::bind(&Iiwa_pub_sub::
       cmd_publisher_look_at_point, this));
8
9      // GET desired qdot
10     //LOOK-AT-POINT CONTROLLER
11     dqd_prev_ = joint_velocities_;
12     dqd_.data = controller_.look_at_point_control(cPo,Eigen::VectorXd::
       Zero(7),1.5);
13     // GET desired q
14     qd_prev_ = joint_positions_;
15     qd_.data = qd_prev_.data + dqd_.data*dt;
16     // GET desired qddot
17     //ddqd_.data.setZero();
18     ddqd_.data = (dqd_.data - dqd_prev_.data)/dt;
19
20     // Wrap q_des and dq_des in JntArrayVel structure
21     KDL::JntArrayVel desJntPosVel(qd_, dqd_);
22     // Define FrameVel structure for storing forward kinematics results
       and compute FK
23     KDL::FrameVel desCartPosVel; robot_->getDirectKinematicsPosVel(
       desJntPosVel, desCartPosVel);
24
25     // Extract desired pose
26     //KDL::Frame eeFrame_d(desCartPosVelAcc.M.R, desCartPosVelAcc.p.p);
27     KDL::Frame eeFrame_d(desCartPosVel.M.R, desCartPosVel.p.p);
28     // Extract desired velocity
29     //KDL::Twist eeVelTwist_d(desCartPosVelAcc.p.v, desCartPosVelAcc.M.w)
       ;
30     KDL::Twist eeVelTwist_d(desCartPosVel.p.v, desCartPosVel.M.w);
```

```
31      // Define acceleration structure and compute 2nd FK
32      //KDL::Twist eeAccTwist_d(desCartPosVelAcc.p.dv, desCartPosVelAcc.M.
    dw);
33      //KDL::Twist eeAccTwist_d = KDL::Twist::Zero();
34
35      // Define acceleration structure and compute 2nd FK
36      //using angle and axis we need the geometric jacobian, so vdot = J
    qdd + jdot qdot
37      //Define J_dot*q_dot
38      Eigen::Matrix<double,6,1> Jdotqdot = robot_->getEEJacDotqDot();
39      KDL::Twist Ades( toKDLTwist( (robot_->getEEJacobian().data) * ddqd_.
    data + Jdotqdot)) ;
40
41      std::cout<<"\n In effort_operational pre Controller\n";
42
43      // Apply Op. Space Controller
44      desired_commands_ = toStdVector(controller_.idCntr(eeFrame_d,
    eeVelTwist_d, Ades, Kp_j, Kpo_j,Kd_j,Kdo_j));
45
46  }
```

Listing 3.2: Operetional space control branch

3.3 Merge controller

The final task requires the use of an additional controller that enables the tracking of a trajectory, either linear or circular, calculated by the planner, while ensuring the `Aruco Marker` remains at the center of the camera's view. In simple terms, by using the `camera_optical_frame`, which has its `z`-axis aligned with the camera's depth, the goal is to keep this axis aligned with the center of the `Aruco Marker`, regardless of its relative orientation.

To perform both tasks simultaneously, an `operational-space` controller was used. This controller requires two inputs: a position error (provided by the planner) and an orientation error (determined by the camera's position in the world frame). By driving the first error to zero, trajectory tracking is ensured, while keeping the second error zero guarantees that the camera is always oriented towards the `Aruco Marker`.

The key part of the code is shown below: to ensure proper functionality, an additional reference frame has been added to the robot's kinematic tree structure, aligning its z-axis to point towards the `Aruco Marker`.

```
1
2 //Inside the branch if(task_== merge_controllers)
3
4
5 // REFERENCES
6 KDL::Frame eeFrame_d(KDL::Rotation::Identity(),toKDL(p.pos));
7 KDL::Twist eeVelTwist_d(toKDL(p.vel),KDL::Vector::Zero());
8 KDL::Twist eeAccTwist_d(toKDL(p.acc),KDL::Vector::Zero());
9 // Compute Desired Frame
10 Eigen::Matrix<double,3,1> aruco_pos_n = cPo;//(aruco_pose[0],aruco_pose
     [1],aruco_pose[2]);
11 aruco_pos_n.normalize();
12 Eigen::Vector3d r_o = skew(Eigen::Vector3d(0,0,1))*aruco_pos_n;
13 double aruco_angle = std::acos(Eigen::Vector3d(0,0,1).dot(aruco_pos_n));
14 KDL::Rotation Re = KDL::Rotation::Rot(KDL::Vector(r_o[0], r_o[1], r_o[2]),
       aruco_angle);
15 eeFrame_d.M = robot_->getEEFrame().M*Re;
16
17 // APPLY OP. SPACE CONTROLLER
18 desired_commands_ = toStdVector(controller_.idCntr(eeFrame_d, eeVelTwist_d
     , eeAccTwist_d, Kp_j, Kpo_j,Kd_j,Kdo_j));
19
20 // Create msg and publish
21 std_msgs::msg::Float64MultiArray cmd_msg;
22 cmd_msg.data = desired_commands_;
23 cmdPublisher_->publish(cmd_msg);
24 RCLCPP_INFO(this->get_logger(), "Starting trajectory execution ...");
25
26  }
```

Listing 3.3: Merge Controller Core

3.4 Effort commands and Cartesian Error plot

To conclude our report, we used the executable of rqt_plot to analyze in a more detailed way the behavior of some fundamental quantities, such as the values of the torque commands sent to the manipulator and the norm of the error between the desired position and the actual and effector position. In the following we present the results:

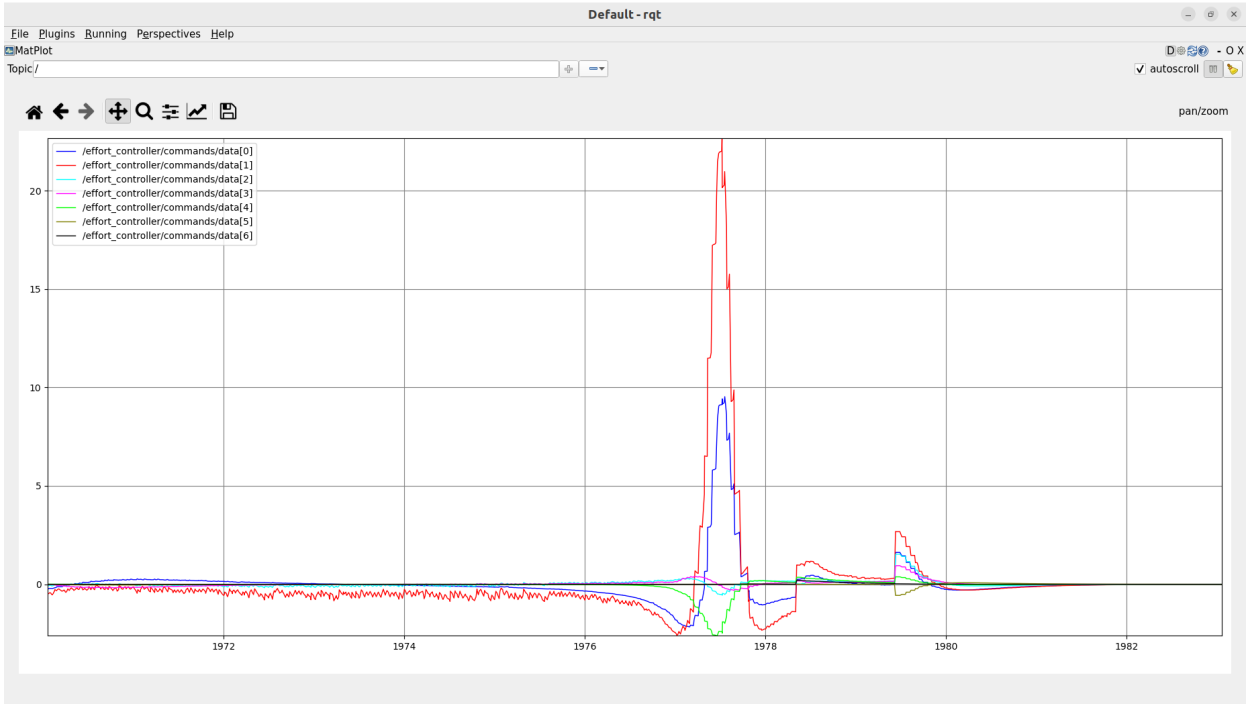


Figure 3.1: `effort_control` Input

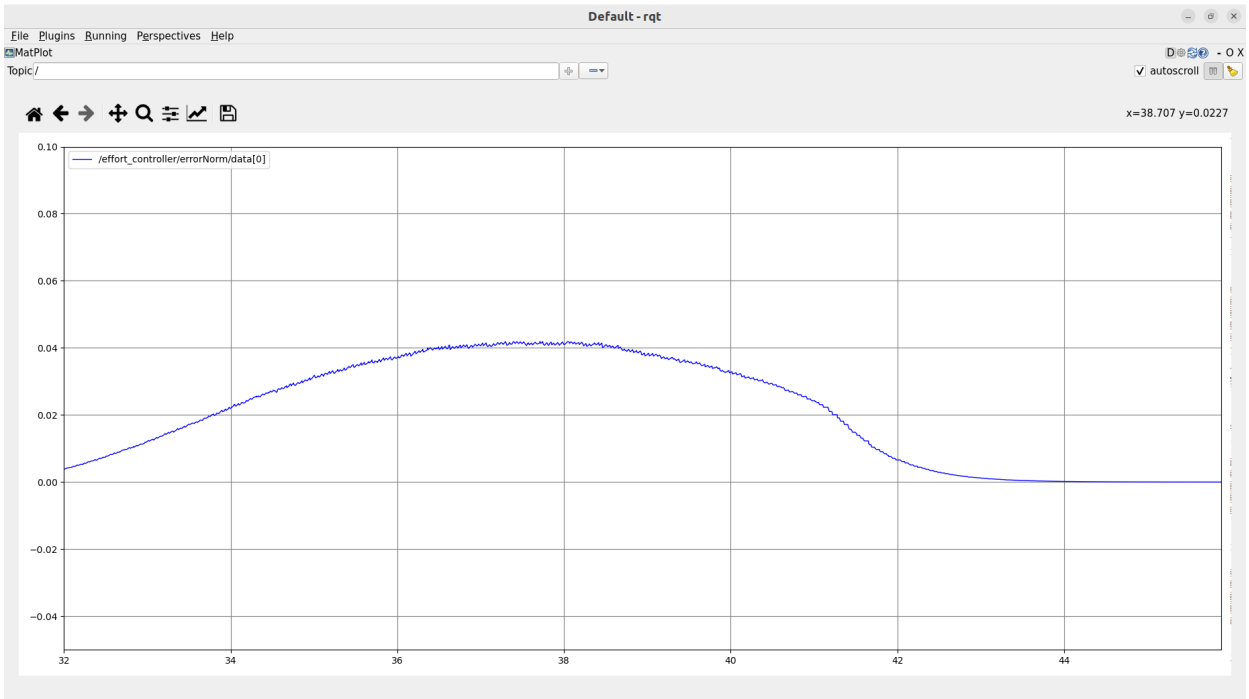


Figure 3.2: Cartesian Error Norm