Robotics Lab course

Homework 3
Implement a vision-based task

Student:
Vincenzo Palomba
P38000180

Teacher:
Prof. Mario Selvaggio
Assistants:
Andrea Capuozzo
Claudio Chiariello

GitHub Repository

It is possible to find all the code used in this report at: https://github.com/vincip99/RoboticsLabHW3.

Construct a gazebo world inserting a blue colored circular object and detect it via the `vision_opencv` package. A template for the implementation is provided

(a) Go into the `iiwa_description` package of the `ros2_iiwa` stack. There you will find a folder `gazebo/models` containing the aruco marker model for gazebo. Taking inspiration from this, create a new model named `spherical_object` that represents a 15 cm radius blue colored spherical object and import it into a new gazebo world as a static object in $x = 1, y = -0.5, z = 0.6$. Save the new world into the `/gazebo/worlds/` folder.

A new directory named `spherical_object` was created into the `gazebo/models` folder of the `iiwa_description` package in order to store the model definition files for the spherical object.

A `model.config` file was created in the `spherical_object` directory to provide metadata to gazebo about the new model such as model name, sdf file and description.

An SDF file, `model.sdf`, was created in the `spherical_object` directory to define the geometry, visual properties, and collision properties of the model:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.9'>
  <model name='spherical_object'>
    <static>true</static>
    <pose>0 0 0 0 0 0</pose>
    <link name='base'>
      <visual name='base_visual'>
        <geometry>
          <sphere>
            <radius>0.015</radius>
          </sphere>
        </geometry>
        <material>
          <diffuse>0 0 1 1</diffuse>
          <specular>0.4 0.4 0.4 1</specular>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

Listing 1: Model SDF File: spherical_object.sdf

A new world file, named `spherical_object_world.world`, was created in the `gazebo/worlds/` directory. The world included, above other basic gazebo description command, the spherichal object with its pose:

```
<world name="spherical_object_world">
<include>
  <uri>
    model://spherical_object
  </uri>
  <name>spherical_object</name>
  <pose>1.0 -0.5 0.6 0 0 0</pose>
</include>
```

Listing 2: Spherical world file: spherical_object_world.world

Finally, to correctly load the new world the `iwaa.launch.py` launch file was modified accordingly, to allow the use of a parameter in order to choose various gazebo worlds:

```
gz_world = LaunchConfiguration('gz_world') # use gazebo world param

models_path = os.path.join(get_package_share_directory('iiwa_description'),
    ↪ 'gazebo', 'models')
```

Listing 3: iiwa launch file: iiwa.launch.py

(b) Equip the robot with a camera at the end-effector that loads optionally setting arguments from `iiwa.launch.py` file (it is recomended to use `<xacro:if value="${use_vision}"}>`). Modify the launch file to load the robot with the camera into the new world specifying the argument `use_vision:=true`. make sure the robot sees the imported object with the camera, otherwise modify its initial configuration (Hint: check it with `rqt_image_view`).

First, in order to add the camera to the robot's end effector, a `camera.xacro` file was created into the `urdf` folder of the `iiwa_description` package.

```
  <xacro:macro name="end_effector_camera" params="use_vision␣
   ↪ prefix">
    <!-- Optional Camera -->
    <xacro:if value="${use_vision}">
      <link name="${prefix}camera_link">
        <visual>
          <geometry>
            <box size="0.02␣0.02␣0.02"/>
          </geometry>
          <material name="camera_material"> <!-- Add the name
             ↪ attribute here -->
            <color rgba="0.0␣0.0␣0.0␣1.0"/>
          </material>
        </visual>
        <origin xyz="0␣0␣0.01" rpy="0␣0␣0"/>
      </link>
      <joint name="camera_joint" type="fixed">
        <parent link="${prefix}tool0"/>
        <child link="${prefix}camera_link"/>
        <origin xyz="0␣0␣0.01" rpy="0␣-1.57␣-3.14"/>
      </joint>
      <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>
              <plugin filename="gz-sim-sensors-system"
                 ↪ name="gz::sim::systems::Sensors">
                <render_engine>ogre2</render_engine>
              </plugin>
              <always_on>1</always_on>
              <update_rate>30</update_rate>
              <visualize>true</visualize>
```

```
                <topic>camera</topic>
            </sensor>
        </gazebo>
      </xacro:if>
    </xacro:macro>
```

Listing 4: Xacro File: camera.xacro

Notice how the directive `<xacro:if>` was used to enable conditional loading of the camera based on the `use_vision` argument. To actually use this as an argument, it was necessary to add into the `iiwa.config.xacro` the following row:

```
<!-- Enable setting camera argument from the launch file -->
<xacro:arg name="use_vision" default="false" />
```

Listing 5: xacro config file: iiwa.config.xacro

Plus, in the robot's urdf file (`iiwa.urdf.xacro`) the camera was included, as well as the necessary arguments.

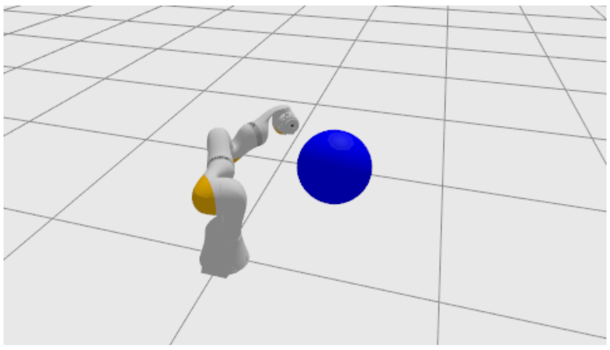
```
<!-- Camera macro inclusion -->
<xacro:include filename="$(find␣
   ↪ iiwa_description)/urdf/camera.xacro"/>
<xacro:end_effector_camera use_vision="$(arg␣use_vision)"
   ↪ prefix="$(arg␣prefix)" />
```

Listing 6: urdf: iiwa.urdf

Finally the launch file was modified to pass the `use_vision` argument and to add the camera bridge:

```
use_vision = LaunchConfiguration('use_vision') # use vision param

# Bridge
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:=/stereo/left/image_rect_color',
        '-r', '/camera_info:=/stereo/left/camera_info'
    ],
    output='screen',
    condition=IfCondition(use_vision)
)
```

Listing 7: iiwa launch file: iiwa.launch.py



(c) Once the object is visible in the camera image, use the `ros2_opencv` package (and specifically the `ros2_opencv_node.cpp` node) to subscribe to the simulated image, detect the spherical object in it using openCV functions, and republish the processed image.

Once the robot is equipped with a camera, the next task is to subscribe to the camera feed and process the image to detect the spherical object using OpenCV functions. The `ros2_opencv` package is used for this purpose, specifically the `ros2_opencv_node.cpp` node. This node will subscribe to the camera's image topic, apply **OpenCV** functions to detect the spherical object, and then republish the processed image.

Initially, the incoming image, received as a ROS2 `sensor_msgs::msg::Image`, is converted to an OpenCV compatible data structure `cv::Mat` format using the `cv_bridge` library. This ensures integration between ROS2 and OpenCV's processing functions. Subsequently, the image is converted to grayscale using `cv::cvtColor`, reducing data complexity while preserving essential intensity information.

```
// Convert ROS2 image to OpenCV Mat
cv_bridge::CvImagePtr cv_ptr = cv_bridge::toCvCopy(msg, "bgr8");
cv::Mat img_ = cv_ptr->image;

// Convert to grayscale
cv::Mat gray_img;
cv::cvtColor(img_, gray_img, cv::COLOR_BGR2GRAY);
```

Listing 8: Vision Control file: ros2_opencv_node.cpp

A **blob detector** is configured using OpenCV's `SimpleBlobDetector` with parameters optimized for detecting the spherical object:

- **Thresholding**: The minimum and maximum intensity thresholds are set to capture all pixel intensities, ensuring no potential blob is overlooked.
- **Area Filtering**: The detector filters blobs by their pixel area. The area bounds are derived from the sphere's diameter in pixels, incorporating a $\pm 20\%$ tolerance for scaling and perspective effects.
- **Circularity Filtering**: To ensure that the blobs are approximately circular, a minimum circularity value of 0.785 (pi/4) is used.
- **Convexity Filtering**: Convexity ensures that the detected shape does not have indentations. A minimum convexity of 0.87 is set.
- **Inertia Ratio Filtering**: The inertia ratio ensures the roundness of the detected blob by penalizing elongated shapes. A minimum ratio of 0.6 is defined.

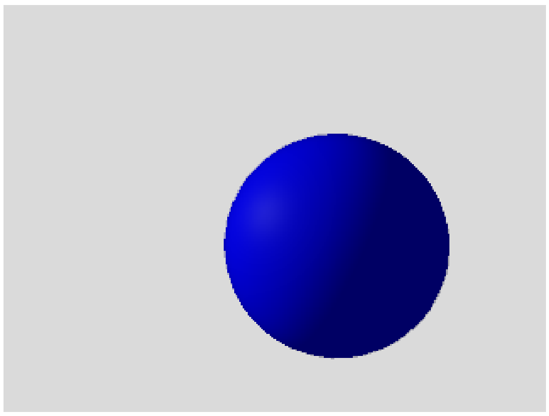
```
// Hardcoded parameters for blob detection
cv::SimpleBlobDetector::Params params;
params.minThreshold = 0.0;  // Minimum intensity threshold for blob detection
params.maxThreshold = 255.0; // Maximum intensity threshold

// Filter by area
params.filterByArea = true;
params.minArea = M_PI * pow(sphere_diameter_pixels / 2.0 * 0.8, 2);  // Allow
    ↪ for some tolerance
params.maxArea = M_PI * pow(sphere_diameter_pixels / 2.0 * 1.2, 2);  // Allow
    ↪ for some tolerance
// Filter by Circularity
params.filterByCircularity = true;
params.minCircularity = 0.785; // Near circular shape
// Filter by Convexity
params.filterByConvexity = true;
params.minConvexity = 0.87;
// Filter by Inertia
params.filterByInertia = true;
params.minInertiaRatio = 0.6; // Tolerance for roundness
```

Listing 9: Vision Control file: ros2_opencv_node.cpp

The grayscale image is processed by the detector, which identifies blobs matching the configured parameters. Detected blobs are annotated on the grayscale image using the`cv::drawKeypoints` function, where red circles represent the detected blobs. The processed image, containing the annotated blobs, is converted back to a ROS2 image message using `cv_bridge` and published to a specified topic.

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

// Detect blobs
std::vector<cv::KeyPoint> keypoints;
detector->detect(gray_img, keypoints);

// Draw detected blobs as red circles
// DrawMatchesFlags::DRAW_RICH_KEYPOINTS ensures the size corresponds to blob
    ↪ size
cv::Mat im_with_keypoints;
cv::drawKeypoints(gray_img, keypoints, im_with_keypoints, cv::Scalar(0, 0,
    ↪ 255), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

// Convert the processed image to ROS2 message
auto msg_ = cv_bridge::CvImage(std_msgs::msg::Header(), "bgr8",
    ↪ im_with_keypoints).toImageMsg();

// Publish the image to the topic defined in the publisher
publisher_->publish(*msg_);
RCLCPP_INFO(this->get_logger(), "Image␣%ld␣published", count_);
count_++;
```

Listing 10: Vision Control file: ros2_opencv_node.cpp


(a) Sphere

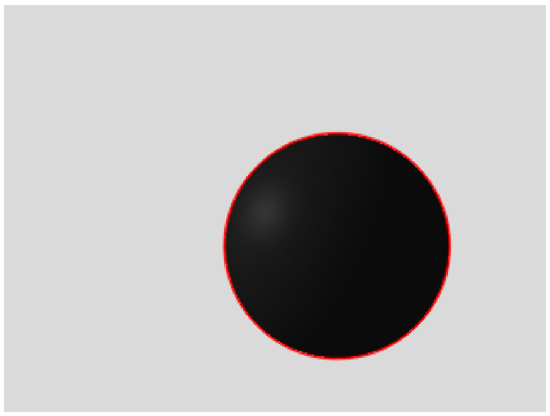

(b) Detected sphere

Figure 1: Comparison of the original sphere and the detected sphere.

Implement a look-at-point vision-based controller

(a) Spawn the robot with the velocity command interface into a world containing an aruco tag. In the `ros2_kdl_package` package create 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 he following two tasks (it is recommended to switch between the two on the basis of a `task:=positioning|look-at-point` argument passed to the node)

  i. aligns the camera to the aruco marker with a desired position and orientation offsets

  To implement the requested functionality, the task involves creating a node that uses vision-based control to align the robot's end-effector-mounted camera with an ArUco marker while maintaining desired position and orientation offsets.

A callback function was implemented into the node, in order to subscribe to the `/aruco_pose` topic to receive the marker's pose relative to the robot

```
void imageCallback(const geometry_msgs::msg::PoseStamped& msg) {
    aruco_pose_available_ = true;

    // Extract pose information
    const auto position = msg.pose.position;
    const auto orientation = msg.pose.orientation;

    // Convert to KDL::Frame
    KDL::Vector kdl_position(position.x, position.y, position.z);
    KDL::Rotation kdl_rotation = KDL::Rotation::Quaternion(
        orientation.x, orientation.y, orientation.z, orientation.w
    );

    // Set pose_in_camera_frame with the converted position and rotation
    pose_in_camera_frame.M = kdl_rotation;
    pose_in_camera_frame.p = kdl_position;

    KDL::Vector adjusted_position(
        kdl_position.data[0] + 0.03, // Add an offset to X
        kdl_position.data[1],        // Y remains the same
        kdl_position.data[2] - 0.20  // Subtract an offset from Z
    );

    // Construct the rotation matrix with an additional rotation about Y
    KDL::Rotation adjusted_rotation = kdl_rotation *
        ↪ KDL::Rotation::RotY(-1.57);
    pose_in_tool_frame = KDL::Frame(adjusted_rotation, adjusted_position);
}
```

Listing 11: Vision Control file: ros2_kdl_vision_control.cpp

The function is designed to be triggered by a message of type PoseStamped, which typically contains both the position and orientation of an object in 3D space. The boolean flag `aruco_pose_available_` is set to true whenever this message is received, indicating that the pose of the ArUco marker has been successfully captured.

The pose of the detected object is then stored in `pose_in_camera_frame`. This frame represents the pose in the camera's coordinate system.

Lastly, the pose of the object is transformed into the tool frame (the robot's end-effector frame). The orientation is transformed using a series of rotations: `KDL::Rotation::RotX(3.14)` and `KDL::Rotation::RotY(1.57)`. These rotations represent a fixed orientation adjustment applied to the object's detected pose in the camera frame to account for the position and orientation of the tool (end-effector).

Finally, to transform the object pose into the world frame:

```
aruco_pose_ = cart_pose_ * pose_in_tool_frame;
```

Listing 12: Vision Control file: ros2_kdl_vision_control.cpp

Then, into the command publisher callback, the positioning task commands are computed via inverse differential kinematics:

```
if(task_ == "positioning"){
    // Compute differential IK
    Vector6d cartvel; cartvel << p.vel + 5*error, o_error;
    joint_velocities_.data =
        ↪ pseudoinverse(robot_->getEEJacobian().data)*cartvel;
}
```

Listing 13: Vision Control file: ros2_kdl_vision_control.cpp

ii. performs a look-at-point task using the following control law

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

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

$$s = \frac{{}^cP_o}{||{}^cP_o||}$$

hat 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 (to be computed), while $L(s)$ maps linear/angular velocities of the camera to changes in s

$$L(s) = \left[-\frac{1}{||{}^cP_o||}(I - ss^T)\ S(s) \right] R$$

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

Show the tracking capability by manually moving the aruco marker around via the gazebo user interface and reporting the velocity commands sent to the robot.

To leverage modularity in the code, a function that performs the look-at-point control has been developed:

```
Eigen::VectorXd KDLController::look_at_point_control(KDL::Frame
    ↪ pose_in_camera_frame, KDL::Frame camera_frame, KDL::Jacobian
    ↪ camera_jacobian, Eigen::VectorXd q0_dot)
{
    //////////////////////
    // Compute L matrix //
    //////////////////////

    // Convert the camera rotation to Eigen and build the 6x6 spatial
        ↪ rotation matrix
    Matrix6d R = spatialRotation(camera_frame.M);

    // Compute the direction vector s
    Eigen::Vector3d c_P_o = toEigen(pose_in_camera_frame.p);
    Eigen::Vector3d s = c_P_o / c_P_o.norm();

    // Interaction matrix L
    Eigen::Matrix<double, 3, 6> L = Eigen::Matrix<double, 3, 6>::Zero();
    Eigen::Matrix3d L_11 = (-1 / c_P_o.norm()) *
        ↪ (Eigen::Matrix3d::Identity() - s * s.transpose());
    L.block<3, 3>(0, 0) = L_11;
    L.block<3, 3>(0, 3) = skew(s);
    L = L * R;

    ///////////////////////
    // Compute Jacobians //
    ///////////////////////

    Eigen::MatrixXd J_c = camera_jacobian.data; // Camera Jacobian in the
        ↪ camera frame
    Eigen::MatrixXd LJ = L * J_c;               // Combined matrix L * J_c
    Eigen::MatrixXd LJ_pinv =
        ↪ LJ.completeOrthogonalDecomposition().pseudoInverse(); //
        ↪ Moore-Penrose pseudoinverse of L * J_c

    // Compute null-space projector N
    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(J_c.cols(), J_c.cols());
    Eigen::MatrixXd N = I - (LJ_pinv * LJ);

    /////////////////////////
    // Compute Joint Velocities
    /////////////////////////

    Eigen::Vector3d s_d(0, 0, 1); // Desired unit vector pointing forward
    double k = -10;               // Gain for the primary task
    Eigen::VectorXd joint_velocities = k * LJ_pinv * s_d + N * q0_dot;

    Eigen::Vector3d s_error = s - s_d;
    std::cout << s_error.norm() << std::endl;

    // Return computed joint velocities
    return joint_velocities;

}
```

Listing 14: Control file: kdl_control.cpp

The first part of the code computes the interaction matrix L, which defines how changes in joint velocities affect the visual task in the camera frame, while the camera rotation matrix is converted into an `Eigen` matrix and used to build the spatial rotation matrix R. Moreover, the direction vector s is computed as the normalized vector pointing from the camera to the object (target), based on the position P_o^c of the object in the camera frame.
The desired unit vector s_d is set to point forward (along the z-axis), representing the target direction for the visual task, while a gain k is used to scale the control law, ensuring proper task tracking. The joint velocities are then computed as the combination of the pseudoinverse of LJ_c, the desired direction s_d, and the null-space projector N applied to the joint velocity (which is the robot's current joint velocities).
The output is a set of joint velocities that will move the robot's end effector towards the target while following the visual task constraints.

```
// Define the pose in the camera frame
KDL::Frame cartpos_camera = cartpos * KDL::Frame(
    KDL::Rotation::RotY(-1.57) * KDL::Rotation::RotZ(-3.14)
);

// Transform the Jacobian J_cam into the camera frame
KDL::Jacobian J_cam_camera(J_cam.columns());
KDL::changeBase(J_cam, cartpos_camera.M, J_cam_camera);

joint_velocities_.data =
    ↪ controller_.look_at_point_control(pose_in_camera_frame,
    ↪ cartpos_camera, J_cam_camera, q0_dot);
```

Listing 15: Vision Control file: ros2_kdl_vision_control.cpp

Then as one can see, this code snippet transforms a robot's control problem into the camera frame to implement a vision-based "look-at-point" task. The pose of the robot (`cartpos`) is transformed into the camera frame by applying a fixed rotation. The robot's Jacobian (`J_cam`) is converted into the camera frame using the transformed rotation matrix from `cartpos_camera`. The transformed pose and Jacobian are passed to a look-at-point control law, which computes the required joint velocities to align the robot's end-effector (or tool) with the target in the camera frame.

(b) Develop a dynamic version of the vision-based controller. Track the reference velocities generated by the look-at-point vision-based control law with the joint space and the Cartesian space inverse dynamics controllers developed in the previous homework. Merge the two controllers and enable the joint tracking of a linear position trajectory and the look-at-point vision-based task.
Regarding the joint space controller, the following code has been developed:

```
// Define the pose in the camera frame
KDL::Frame cartpos_camera = cartpos * KDL::Frame(
    KDL::Rotation::RotY(-1.57) * KDL::Rotation::RotZ(-3.14)
);
// Transform the Jacobian J_cam into the camera frame
KDL::Jacobian J_cam_camera(J_cam.columns());
KDL::changeBase(J_cam, cartpos_camera.M, J_cam_camera);

prev_joint_velocities_.data = joint_velocities_.data;

Eigen::Vector3d c_P_o = toEigen(pose_in_camera_frame.p);
Eigen::Vector3d s = c_P_o / c_P_o.norm();
Eigen::Vector3d s_error = s - s_d;
error_norm = s_error.norm();
std::cout << error_norm << std::endl;

// Desired joint velocities
```

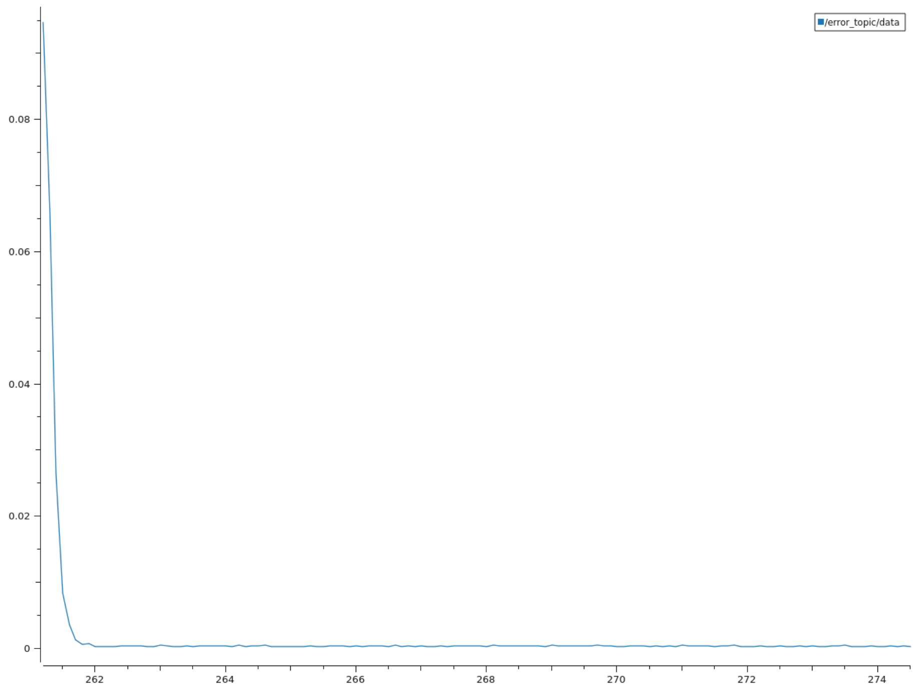


Figure 2: Error norm of s for velocity controlled look-at-point controller

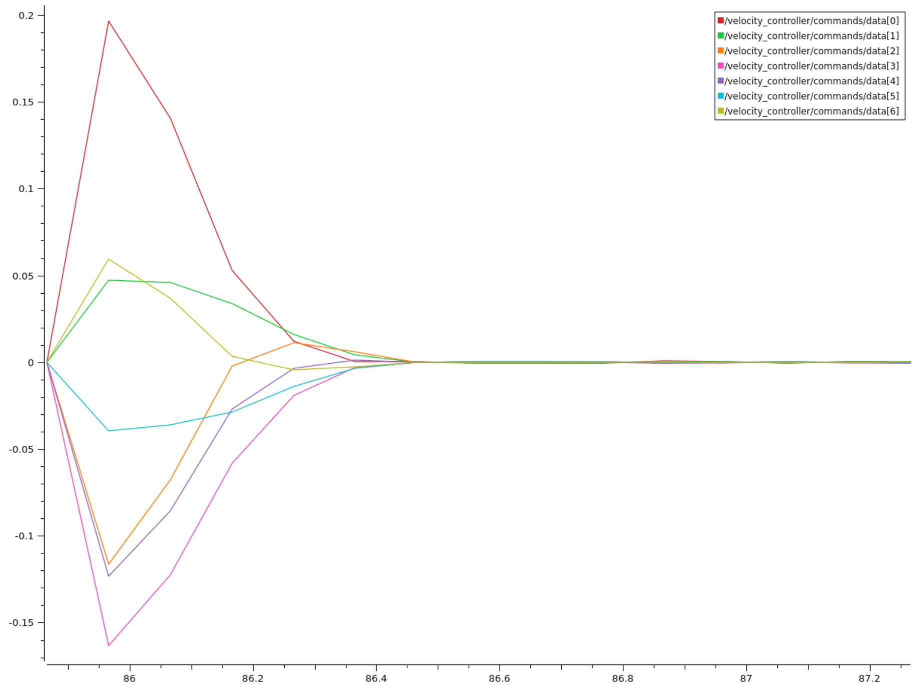


Figure 3: Velocity command for velocity controlled look-at-point controller

```
q_dot_des.data = controller_.look_at_point_control(pose_in_camera_frame,
    ↪ cartpos_camera, J_cam_camera, q0_dot);
// Desired joint positions
q_des.data = joint_positions_.data + q_dot_des.data * dt;
// Desired joint acceleration
q_ddot_des.data = (joint_velocities_.data - prev_joint_velocities_.data) /
    ↪ dt;

double cos_theta = s.dot(s_d); // Dot product between vectors
cos_theta = std::max(-1.0, std::min(1.0, cos_theta)); // Clamp to [-1, 1]
// double angular_error = std::acos(cos_theta);
```

```
double theta = std::acos(cos_theta);
Eigen::Vector3d axis = s_d.cross(s);
//Eigen::Vector3d orientation_error = cam_orient_error * s;
KDL::Rotation R_des = KDL::Rotation::Rot(toKDL(axis), theta);

joint_efforts_.data = controller_.idCntr(q_des, q_dot_des, q_ddot_des, Kp,
    ↪ Kd);
```

Listing 16: Vision Control file: ros2_kdl_vision_control.cpp

The target's direction is computed in the camera frame, and the desired alignment direction is defined. The alignment error is determined as the difference between the current direction and the desired one, providing a feedback mechanism for the control loop.

To ensure proper alignment, the robot's end-effector pose and Jacobian are transformed into the camera frame. Using this transformed data, the `look_at_point_control` function calculates joint velocities that drive the robot's tool towards the target (joint positions and accelerations are numerically computed). The updated joint velocities are then used to incrementally adjust joint positions, while joint accelerations are calculated as the time derivative of velocities.

Finally, an inverse dynamics controller computes the required joint torques to execute the desired motion. This is achieved using proportional and derivative control gains, ensuring stability and responsiveness.

While, after the computation of the orientation error via angle axis, the inverse dynamics operational space controller was developed similarly to the previous case:

```
double cos_theta = s.dot(s_d); // Dot product between vectors
cos_theta = std::max(-1.0, std::min(1.0, cos_theta)); // Clamp to [-1, 1]
// double angular_error = std::acos(cos_theta);
double theta = std::acos(cos_theta);
Eigen::Vector3d axis = s_d.cross(s);
//Eigen::Vector3d orientation_error = cam_orient_error * s;
KDL::Rotation R_des = KDL::Rotation::Rot(toKDL(axis), theta);
```

Listing 17: Vision Control file: ros2_kdl_vision_control.cpp

```
desVel = KDL::Twist(KDL::Vector(p.vel[0], p.vel[1], p.vel[2]),
    ↪ KDL::Vector::Zero());
desAcc = KDL::Twist(KDL::Vector(p.acc[0], p.acc[1], p.acc[2]),
    ↪ KDL::Vector::Zero());
// desPos.M = desired_frame.M;
desPos.M = (robot_->getEEFrame().M)*R_des;

joint_efforts_.data = controller_.idCntr(desPos, desVel, desAcc, Kpp, Kpo,
    ↪ Kdp, Kdo);
```

Listing 18: Vision Control file: ros2_kdl_vision_control.cpp

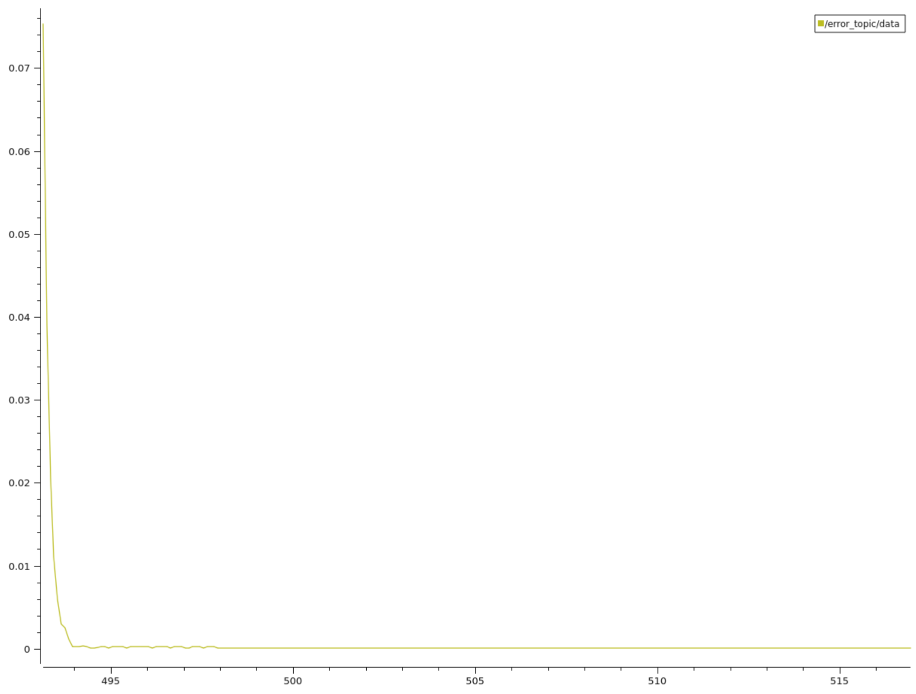


Figure 4: Error norm of s for effort controlled look-at-point controller

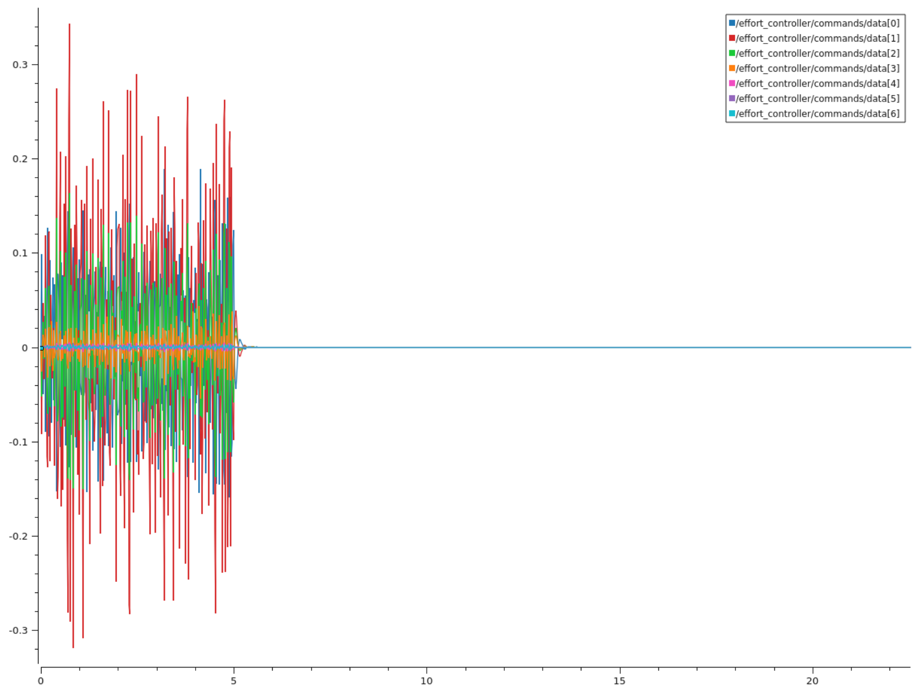


Figure 5: Velocity command for effort controlled look-at-point controller