Robotics Lab

Report Homework 3

Federico Trenti P38000263
Matteo Russo P38000247
Alessandra Del Sorbo P38000289
Giulio Acampora P38000258

2024/2025

GitHub Links

- Link Trenti
- Link Russo
- Link Del Sorbo
- Link Acampora

Contents

Chapter 1

Construct a gazebo world inserting a blue colored circular object and detect it via the vision_opencv package

1.1 Creating a Spherical Object Model in Gazebo

For this task, we worked with the `iiwa_description` package, which was previously implemented as part of Homework 2. We extended the package by introducing a new folder, `gazebo/models`, to organize simulation models.

Within this folder, we created a new directory named `spherical_object`, containing two essential files:

- `model.sdf`: This file defines the physical and visual properties of the model, including its shape, size, and static behavior.
- `model.config`: This file provides metadata for the model, ensuring it can be properly loaded into Gazebo.

The model represents a static, blue sphere with a radius of 15 cm.

```
//model.sdf
<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.9'>
  <model name='spherical_object'>
    <static>true</static>
    <pose>0 0 0.001 0 0 0</pose>
    <link name='base'>
      <visual name="visual">
```

```
        <geometry>
          <sphere>
            <!-- Raggio della sfera in metri -->
            <radius>0.15</radius>
          </sphere>
        </geometry>
        <material>
          <!-- Colore blu -->
          <ambient>0 0 1 1</ambient>
          <diffuse>0 0 1 1</diffuse>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

```
//model.config
<?xml version="1.0"?>
<model>
  <name>spherical_object</name>
  <version>1.0</version>
  <sdf version="1.9">model.sdf</sdf>
  <author>
    <name>Jacob Dahl</name>
    <email>jake@arkelectron.com</email>
  </author>
  <description>Aruco tag model</description>
</model>
```

To incorporate this model into a Gazebo simulation, we created a new world file named `sphere.world`, where the spherical object was placed as a static entity at the following coordinates:

- **x**: 1.0
- **y**: -0.5
- **z**: 0.3

This new world file was saved in the `/gazebo/worlds/` folder.
Additionally, the launch file was modified to load the `sphere.world` file instead of the default world.
These changes ensure the proper integration of the spherical object into the Gazebo simulation environment.We then checked the configuration and functionality of the new model inside Gazebo.
Since the next tasks require launching `aruco.world` instead of `sphere.world`,

we modified the `iiwa.launch.py` file to include an argument that specifies the desired world file. This modification was applied to the nodes responsible for launching Gazebo, ensuring flexibility in selecting the appropriate simulation environment.

```
declared_arguments.append(
    DeclareLaunchArgument(
        'use_aruco',
        #default_value='false',

        default_value='true',
        description='Set to true to use aruco.world, false to use sphere.world',
    )
)
use_aruco=LaunchConfiguration('use_aruco')
```

```
# configurations of 'gz_args'
aruco_gz_args = DeclareLaunchArgument(
    'gz_args',
    default_value=[aruco_world, ' -r -v 1'],
    description='Arguments for gz_sim with aruco.world',
    condition=IfCondition(LaunchConfiguration('use_aruco'))
)
sphere_gz_args = DeclareLaunchArgument(
    'gz_args',
    default_value=[sphere_world, ' -r -v 1'],
    description='Arguments for gz_sim with sphere.world',
    condition=UnlessCondition(LaunchConfiguration('use_aruco'))
)

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

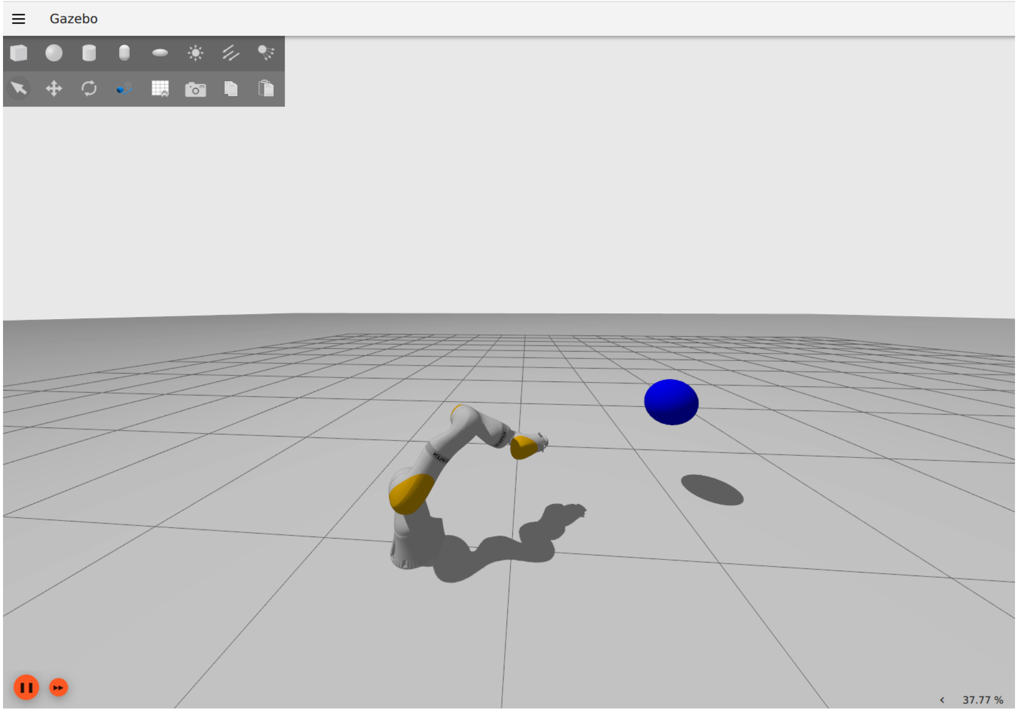

Figure 1.1: Manipulator IIWA in the Gazebo environment within sphere.world

1.2 Adding a Camera to the Robot's End-Effector

To enhance the robot's capabilities, a camera was mounted on the robot's end-effector. This camera was designed to be optionally loaded based on a configurable parameter in the `iiwa.launch.py` file, allowing greater flexibility during simulations.

```
//Modifications to iiwa.config.xacro
<?xml version="1.0"?>
<!-- Kuka iiwa 7DoF manipulator -->
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="iiwa">

    <!-- Enable setting arguments from the launch file -->
    <xacro:arg name="use_sim" default="false" />
    <xacro:arg name="use_fake_hardware" default="true" />
    <xacro:arg name="prefix" default="" />
    <xacro:arg name="robot_ip" default="192.170.10.2" />
```

```
    <xacro:arg name="robot_port" default="30200" />
    <xacro:arg name="initial_positions_file" default="initial_positions.yaml" />
    <xacro:arg name="command_interface" default="position" />
    <xacro:arg name="base_frame_file" default="base_frame.yaml" />
    <xacro:arg name="description_package" default="iiwa_description" />
    <xacro:arg name="runtime_config_package" default="iiwa_description" />
    <xacro:arg name="controllers_file" default="iiwa_controllers.yaml" />
    <xacro:arg name="namespace" default="/" />

    <xacro:arg name="use_vision" default="false"/>

    <xacro:property name="description_package" value="$(arg description_package)"/>

    <!-- Import iiwa urdf file -->
    <xacro:include filename="$(find ${description_package})/urdf/iiwa.urdf.xacro" />

    <!-- Import iiwa ros2_control description -->
    <xacro:include filename="$(find ${description_package})/ros2_control/iiwa.r2c_hardware.xacro" />

    <!-- Import all Gazebo-customization elements -->
    <xacro:include filename="$(find ${description_package})/gazebo/iiwa.gazebo.xacro" />

    <!-- Used for fixing robot -->
    <link name="world"/>
    <gazebo reference="world">
        <static>true</static>
    </gazebo>

    <xacro:property name="base_frame_file" value="$(arg base_frame_file)"/>
    <xacro:property name="base_frame" value="${xacro.load_yaml(base_frame_file)['base_frame']}"/>

    <xacro:property name="use_vision" value="$(arg
```

```
        use_vision)"/>
43
44     <xacro:iiwa parent="world" prefix="$(arg prefix)">
45         <origin xyz="${base_frame['x']} ${base_frame['y']} ${base_frame['z']}"
46                  rpy="${base_frame['roll']} ${base_frame['pitch']} ${base_frame['yaw']}" />
47     </xacro:iiwa>
48
49     <xacro:iiwa_r2c_hardware
50         name="iiwaRobot" prefix="$(arg prefix)"
51         robot_ip="$(arg robot_ip)" robot_port="$(arg robot_port)"
52         command_interface="$(arg command_interface)"
53         initial_positions_file="$(arg initial_positions_file)"
54         use_sim="$(arg use_sim)" use_fake_hardware="$(arg use_fake_hardware)"
55         />
56
57     <xacro:iiwa_gazebo
58         runtime_config_package="$(arg runtime_config_package)"
59         controllers_file="$(arg controllers_file)"
60         namespace="$(arg namespace)"
61         prefix="$(arg prefix)"
62         />
63
64 </robot>
```

In the `iiwa.config.xacro` file, the following lines were added on line 19 and 41:

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

These additions define a new argument, `use_vision`, with a default value of `false`. This argument allows the simulation to dynamically enable or disable the vision system. The launch file `iiwa.launch.py` was updated to support the `use_vision` argument. The following lines were added to the launch file on line 243:

```
' ',
'use_vision:=',
use_vision,
```

```
//Modifications to the Launch File iiwa.launch.py
 # Get URDF via xacro
    robot_description_content = Command(
        [
            PathJoinSubstitution([FindExecutable(name='xacro')]),
            ' ',
            PathJoinSubstitution(
                [FindPackageShare(description_package), 'config', description_file]
            ),
            ' ',
            'prefix:=',
            prefix,
            ' ',
            'use_sim:=',
            use_sim,
            ' ',
            'use_fake_hardware:=',
            use_fake_hardware,
            ' ',
            'robot_ip:=',
            robot_ip,
            ' ',
            'robot_port:=',
            robot_port,
            ' ',
            'initial_positions_file:=',
            initial_positions_file,
            ' ',
            'command_interface:=',
            command_interface,
            ' ',
            'base_frame_file:=',
            base_frame_file,
            ' ',
            'description_package:=',
            description_package,
            ' ',
            'runtime_config_package:=',
            runtime_config_package,
            ' ',
            'controllers_file:=',
            controllers_file,
            ' ',
```

```
                'namespace:=',
                namespace,
                ' ',
                'use_vision:=',
                use_vision,
            ]
        )
```

```
//Modifications to the Launch File iiwa.launch.py
declared_arguments.append(
    DeclareLaunchArgument(
        'use_vision',
        #default_value='false',

        default_value='true',
        description='Start robot with the camera.',
    )
)
use_vision=LaunchConfiguration('use_vision')
```

These lines ensure that the `use_vision` argument can be passed to the robot's configuration during the launch process. This integration makes it possible to load the robot with the camera into the simulation world when the argument `use_vision:=true` is specified.

1.2.1 Integration with the `sphere.world`

To verify the functionality of the camera, the robot was loaded into the newly created `sphere.world`, which contains the blue spherical object. The camera's field of view was adjusted to ensure that the imported object was visible. This was achieved by modifying the camera initial configuration until the object appeared correctly in the camera feed.

```
//In iiwa.urdf.xacro Modifications of rpy of use_vision
 <xacro:if value="${use_vision}">
    <joint name="camera_joint" type="fixed">
        <parent link="${prefix}tool0"/>
        <child link="camera_link"/>
        <origin xyz="0 0 0.00" rpy="0 -1.57 3.14"/>
    </joint>

    <link name="camera_link">
        <visual>
            <geometry>
                <box size="0.02 0.02 0.02"/>
            </geometry>
```

```
        </visual>
    </link>
  </xacro:if>

  <xacro:include filename="$(find iiwa_description)/
     urdf/iiwa.camera.xacro"/>
  <xacro:camera_ros2_control/>

  </xacro:macro>
</robot>
```

The orientation of the camera has been modified with the following coordinates:

- **r**: 0
- **p**: -1.57
- **y**: 3.14

The `rqt_image_view` tool was used to validate the camera's functionality. By observing the camera feed, it was confirmed that the blue spherical object was visible. If necessary, the robot's initial position or orientation was adjusted further to align the camera with the object.

1.3 Object Detection with OpenCV in ROS2

Once the spherical object was visible in the camera feed, the next task was to process the camera image in order to detect the object. For this purpose, the `ros2_opencv` package was used, specifically leveraging the `ros2_opencv_node.cpp` node to subscribe to the simulated camera image and perform object detection using OpenCV functions. The `ros2_opencv` package provides a straightforward method for processing image data.

We detect the contours of the blue object using the `cv::findContours` function. This implementation was inspired by the tutorial on contour detection provided on the Learn OpenCV website (`https://learnopencv.com/contour-detection-using-opencv-python-c/`).

1.3.1 ROS2 Node Construction

The node is built using ROS 2, which provides a modular framework for robotic applications. The node subscribes to the topic /camera/image_raw, which provides the raw image stream from the camera. Once an

image is received, the node processes it by detecting the blue object, draws its contours, and then publishes the processed image to the topic /processed_image.
The ROS 2 subscription and publication are managed as follows:

```
subscription_ = this->create_subscription<sensor_msgs::
    msg::Image>(
    "/camera/image_raw", 10, std::bind(&
        BlueObjectDetector::image_callback, this, _1));
publisher_ = this->create_publisher<sensor_msgs::msg::
    Image>("/processed_image", 10);
```

Here, create_subscription subscribes to the camera's image stream and links the incoming messages to the image_callback function. The publisher, created with create_publisher, publishes the processed image.

1.3.2 Image Conversion from ROS2 to OpenCV

The image received from ROS is in the sensor_msgs::msg::Image format, which is not directly usable by OpenCV. To process the image, we convert it to an OpenCV cv::Mat object using the cv_bridge package. This conversion is done as follows:

```
cv_bridge::CvImagePtr cv_ptr = cv_bridge::toCvCopy(msg,
    sensor_msgs::image_encodings::BGR8);
cv::Mat image = cv_ptr->image;
```

The cv_bridge::toCvCopy function converts the ROS image message into an OpenCV cv::Mat object, which represents the image in the BGR (Blue-Green-Red) color space.

1.3.3 Color Thresholding

To isolate the blue object in the image, we convert the image from the BGR color space to the HSV (Hue, Saturation, Value) color space. The HSV color space is often more effective for color-based segmentation than BGR because it separates the color information (Hue) from intensity (Value).

The conversion is done using the cv::cvtColor function:

```
cv::cvtColor(image, hsv_image, cv::COLOR_BGR2HSV);
```

We then apply a color threshold to isolate the blue regions in the image. The cv::inRange function is used to create a binary mask where pixels within the specified blue color range are set to 255 (white), and the others are set to 0 (black):

```
cv::inRange(hsv_image, cv::Scalar(100, 150, 50), cv::
    Scalar(140, 255, 255), mask);
```

The parameters (100, 150, 50) and (140, 255, 255) define the lower and upper bounds for the hue, saturation, and value of the blue color.

1.3.4 Morphological Operations

To clean up the binary mask and remove small noise, morphological operations are applied. First, we erode the mask, followed by dilation to fill in any holes. This is done using the cv::erode and cv::dilate functions:

```
cv::erode(mask, mask, cv::Mat(), cv::Point(-1, -1), 2);
cv::dilate(mask, mask, cv::Mat(), cv::Point(-1, -1), 2)
    ;
```

These operations improve the quality of the mask and help in detecting the contours more accurately.

1.3.5 Contour Detection

After obtaining the clean binary mask, we detect the contours of the blue object using the cv::findContours function:

```
cv::findContours(mask, contours, hierarchy, cv::
    RETR_TREE, cv::CHAIN_APPROX_SIMPLE);
```

The cv::RETR_TREE flag is used to retrieve all contours and their hierarchical relationships, while cv::CHAIN_APPROX_SIMPLE simplifies the contours by removing redundant points.

1.3.6 Drawing Contours

Once the contours are detected, we draw them on the original image to visualize the blue object. The contours are drawn using the cv::drawContours function:

```
cv::drawContours(processed_image, contours, -1, cv::
    Scalar(0, 255, 0), 2);
```

This draws the contours in green (cv::Scalar(0, 255, 0)) with a line thickness of 2.

1.3.7 Publishing the Processed Image

Finally, the processed image, with the detected contours, is converted back into a ROS message and published on the /processed_image topic:

```
auto output_msg = cv_bridge::CvImage(msg->header,
    sensor_msgs::image_encodings::BGR8, processed_image)
    .toImageMsg();
```

```
publisher_->publish(*output_msg);
```

This allows other nodes to subscribe to the processed image for further analysis or actions.

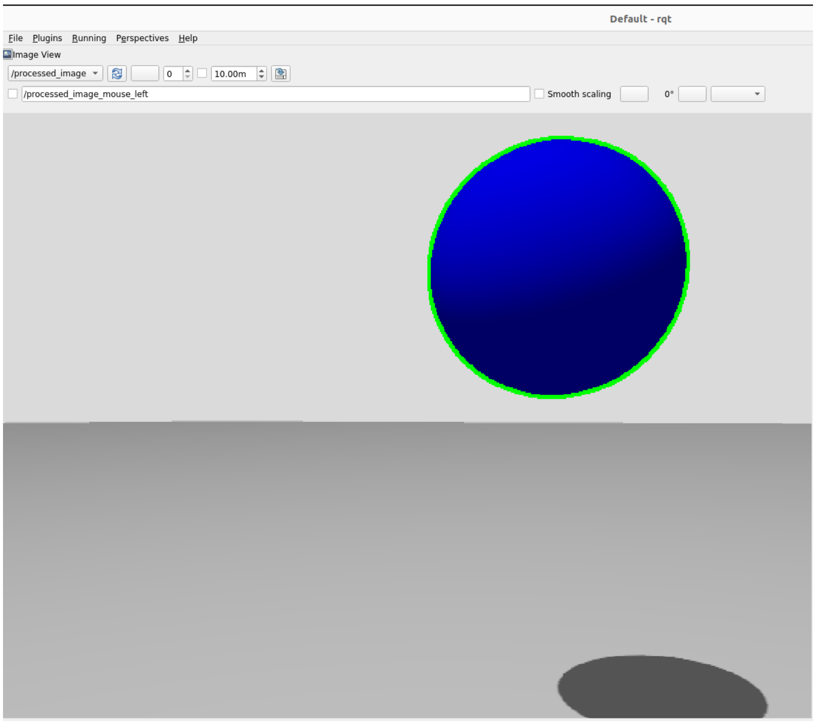

Figure 1.2: Detection of the spherical object using openCV functions

Chapter 2

Implement a look-at-point vision-based controller

2.1 Vision-based controller

The simulated environment consists of a robot equipped with a velocity command interface and a camera mounted on its end-effector. An Aruco marker is placed within the environment, serving as the visual target for the robot.
The control logic is implemented in a ROS 2 node named ros2_kdl_vision_control.cpp, which is part of the ros2_kdl_package. This node exploits the Kinematics and Dynamics Library (KDL) to compute the robot's motion and ensure the execution of the control tasks.

2.1.1 Camera alignment to the Aruco Marker and positioning Task

The first task of the vision-based controller is to align the camera of the IIWA robot with an Aruco marker. We achieved the alignment by maintaining a desired position and orientation offset relative to the marker.
We modified the `iiwa.urdf.xacro` file by introducing an additional frame named `camera_optical_joint`.

```
<xacro:if value="${use_vision}">
   <joint name="camera_joint" type="fixed">
       <parent link="${prefix}tool0"/>
       <child link="camera_link"/>
       <origin xyz="0 0 0.00" rpy="0.0 -1.57 3.14"/>
   </joint>

   <joint name="camera_optical_joint" type="fixed">
     <parent link="camera_link"/>
```

```
        <child link="camera_link_optical"/>
        <origin rpy="${-pi/2} 0 ${-pi/2}" xyz="0.0 0.0
           0.000" />
      </joint>

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

      <link name="camera_link">
          <visual>
              <geometry>
                  <box size="0.02 0.02 0.02"/>
              </geometry>
              <!-- <material name="red"/> -->
          </visual>
      </link>
  </xacro:if>

  <xacro:include filename="$(find iiwa_description)/
     urdf/iiwa.camera.xacro"/>
  <xacro:camera_ros2_control/>
```

This modification follows the approach described in this tutorial.
The tutorial explains that to accommodate different conventions, the standard approach is to create two distinct frames at the same location: one named `camera_link` (following the ROS2 convention) and the other named `camera_link_optical` (following the standard image convention).

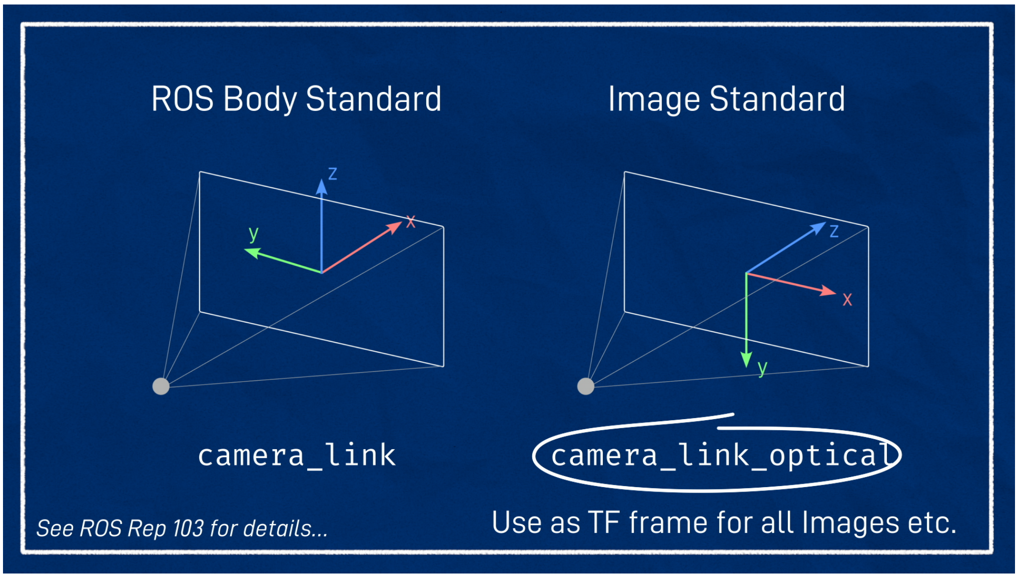

Figure 2.1: Different orientation of the camera_link_optical and `camera_link_optical`

In our case, we used the `single_simple` file from the `ros2_vision` package for implementing detection.
However, the frame conventions in `single_simple` and ROS 2 differ. Therefore, it was necessary to add the `camera_optical_joint` frame to align the conventions properly and ensure compatibility with our implementation.

```
// Initialize controller
KDLController controller_(*robot_);
if(task_ == "positioning"){
    // EE's trajectory initial position (just an offset
       )
    Eigen::Vector3d init_position(Eigen::Vector3d(
       init_cart_pose_.p.data) - Eigen::Vector3d
       (0,0,0.01));

    double offset_camera = 1; // Distanza desiderata
       dal marker

    Eigen::Vector3d end_position;
    end_position << marker_frame_.p.data[0] +
       offset_camera, marker_frame_.p.data[1],
       marker_frame_.p.data[2];

    std::cout << "END position: " << end_position[0]<<"
        "<< end_position[1]<<" "<<end_position[2] << "\
       n";


    // Plan trajectory
    double traj_duration = 1.5, acc_duration = 0.5;
    double t= 0.0;
    //std::cout << "DEBUG: Inizio dello switch,
       traj_chosen = " << traj_chosen << std::endl;

    // Trajectory rectilinear cubic

    planner_ = KDLPlanner(traj_duration, acc_duration,
       init_position, end_position); // currently using
        cubic_polynomial for rectiliniar path
    p = planner_.compute_trajectory(t);

    // compute errors
    Eigen::Vector3d error = computeLinearError(p.pos,
       Eigen::Vector3d(init_cart_pose_.p.data));
}
```

```
private:

    void cmd_publisher(){

        KDLController controller_(*robot_);
        iteration_ = iteration_ + 1;
        double total_time;
        double dt;

        if(task_=="positioning"){

        total_time = 1.5; //
        int trajectory_len = 150; //
        int loop_rate = trajectory_len / total_time;
        dt = 1.0 / loop_rate;
        t_+=dt;
        }else{

        // define trajectory
        total_time = 1.5*3; //
        int trajectory_len = 150*3; //
        int loop_rate = trajectory_len / total_time;
        dt = 1.0 / loop_rate;
        t_+=dt;
        }

        if (t_ < total_time)        // until the trajectory hasn't finished
        {
            if(task_ == "positioning") {
                if(cmd_interface_ == "velocity"){
                    if(t_ <= total_time) {
                        p = planner_.compute_trajectory(t_);
                        if(t_ >= total_time - dt) {                       // last dt before the end of the trajectory
                            p = planner_.compute_trajectory(t_);
                            final_pos = p;
                        }
                    }
                    else {
                        // std::cout << "tempo attuale" << t_;
```

```
                    p.pos = final_pos.pos;
                    p.vel = Eigen::Vector3d::Zero();
                    p.acc = Eigen::Vector3d::Zero();
                }

                // Compute EE frame
                KDL::Frame cartpos = robot_->getEEFrame();

                KDL::Rotation y_rotation = KDL::Rotation::RotY(M_PI);
                KDL::Rotation marker_frame_rotated;
                marker_frame_rotated = marker_frame_.M * y_rotation;



                // compute errors
                Eigen::Vector3d error = computeLinearError(p.pos, Eigen::Vector3d(cartpos.p.data));
                Eigen::Vector3d o_error = computeOrientationError(toEigen(marker_frame_rotated), toEigen(cartpos.M));
                std::cout << "The error norm is : " << error.norm() << std::endl;

                // Compute differential IK
                Vector6d cartvel; cartvel << p.vel + 5*error, o_error;
                joint_velocities_.data = pseudoinverse(robot_->getEEJacobian().data)*cartvel;
                joint_positions_.data = joint_positions_.data + joint_velocities_.data*dt;
            }
```

The trajectory of the end-effector is planned based on a predefined time horizon. A trajectory planner computes the desired position, velocity, and acceleration of the EE. During the active trajectory phase, the planner generates the desired trajectory based on the current time t relative to the total planned duration T_{total}.

At the end of the trajectory, the final position is stored and used to maintain the end-effector at the desired target. Linear and errors are computed to quantify the deviation of the EE from the desired position.

2.1.2 Look-at-Point Control Task

The second task involves implementing a look-at-point control law to maintain the camera's focus on a specific point in space. The control strategy is defined by the following equation:

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

where $s_d = [0, 0, 1]$ represents the desired unit vector aligning the camera's optical axis with the object's position relative to the camera frame, denoted by ${}^cP_o$.
The controller uses the normalized vector $s = \frac{{}^cP_o}{\|{}^cP_o\|} \in S^3$ to define this alignment.
The camera Jacobian matrix J_c is computed, while the matrix $L(s)$ maps the linear and angular velocities of the camera frame to changes in s:

$$L(s) = \begin{bmatrix} -\frac{1}{\|{}^cP_o\|}(I - ss^{\top}) & S(s) \end{bmatrix} R^{\top},$$

with

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

Here, $S(\cdot)$ is the skew-symmetric operator, and R_c is the camera rotation matrix. The null-space matrix $N = (I - (LJ)^{\dagger}LJ)$ is used to ensure redundancy resolution and to avoid undesired joint configurations.

```
else if(cmd_interface_ == "effort"){
        //LOOK AT POINT

        // EE's trajectory initial position (just an offset)
    Eigen::Vector3d init_position(Eigen::Vector3d(init_cart_pose_.p.data) - Eigen::Vector3d(0,0,0.1));

    // EE's trajectory end position (just opposite y)
    Eigen::Vector3d end_position; end_position << init_position[0], -init_position[1], init_position[2];

        // Plan trajectory
    double traj_duration = 1.5*3, acc_duration = 0.5*3, t = 0.0;
```

```
    planner_ = KDLPlanner(traj_duration, acc_duration, init_position, end_position); // currently using trapezoidal velocity profile

    // Retrieve the first trajectory point
    trajectory_point p = planner_.compute_trajectory(t);


    }
```

```
} else if(task_ == "look-at-point"){
    //LOOK AT POINT
    Eigen::Vector3d sd(0, 0, 1);
    //Calcolo vettore normalizzato
    Eigen::Vector3d s(marker_frame_.p.x(), marker_frame_.p.y(), marker_frame_.p.z());

    // Calcola la norma di P0
    double norm_s = s.norm();
    s.normalize();
    //Calcolo matrice S
    Eigen::Matrix3d S = skew(s);

    //Calcolo R
    KDL::Frame cartpos = robot_->getEEFrame();
    Eigen::Matrix3d Rc = toEigen(marker_frame_.M);

    Matrix6d R = Matrix6d::Zero();

    R.block(0,0,3,3)=Rc;
    R.block(3,3,3,3)=Rc;

    std::cout << "Matrice 6x6 R:\n" << R << std::endl;

    //Calcolo di L(s)

    // Calcola I - ss^T
    Eigen::Matrix3d ssT = s * s.transpose();
    Eigen::Matrix3d I3 = Eigen::Matrix3d::Identity();
    Eigen::Matrix3d I_minus_ssT = I3 - ssT;

    Eigen::Matrix3d scaled_matrix = (-1.0 / norm_s) * I_minus_ssT;

    Eigen::MatrixXd L  = Eigen::Matrix<double,3,6>::
```

```
       Zero();
    L.block(0,0,3,3)=scaled_matrix;
    L.block(0,3,3,3)=S;
    L=L*R.transpose();

    Eigen::MatrixXd Jc = robot_->getEEJacobian().data;

    Eigen::MatrixXd LJ_PseudoInv = pseudoinverse(L*Jc);

    Eigen::MatrixXd N = Eigen::MatrixXd::Identity(
       robot_->getNrJnts(), robot_->getNrJnts()) -
       pseudoinverse(L*Jc) * L*Jc;

    Eigen::Vector3d error_s = sd - s;

    Eigen::VectorXd q=robot_->getJntValues();
    if(cmd_interface_ == "velocity"){
        //Control law
        joint_velocities_.data = 5*LJ_PseudoInv*sd + N*
            (qi_- q);
        joint_positions_.data = joint_positions_.data +
            joint_velocities_.data*0.02;
    }
```

The robot was controlled to maintain the Aruco marker at the center of its camera view using the above control law. The control law ensure robust tracking and centering of the marker.
The robot successfully identifies the position of the Aruco marker and adjusts its end-effector's position to center the marker within the camera frame. This behavior is achieved through continuous monitoring of the marker's position and applying the control law.
When the marker is manually displaced during operation, the robot detects the change and recalculates its position, autonomously re-centering the marker in the camera view.
The resulting behavior of the robot can be observed in the accompanying video, available in our GitHub repository under the folder `videoHW3`.
While the robot effectively centers the Aruco marker in the camera frame, a limitation in the current implementation was observed. When the marker is displaced laterally (e.g., to the right), the camera's horizon tilts, even though the marker remains centered. We may have this issue because, for semplicity in the code we used the EE jacobian instead of the camera jacobian even if the orientation is different. The camera frame is not coincident to the end-effector frame as showed in the following image:

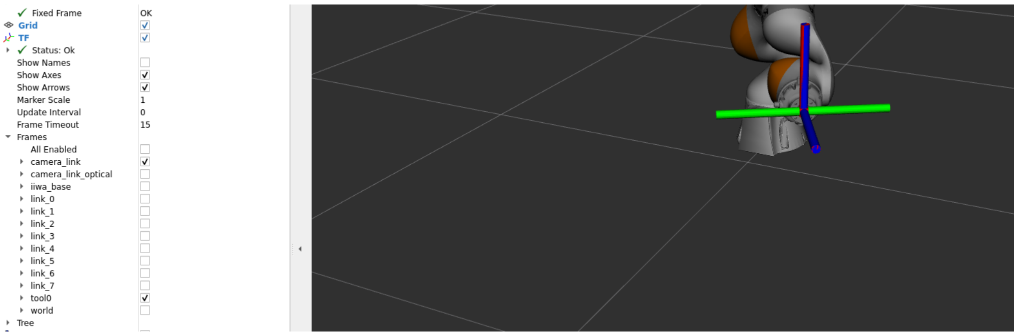

Figure 2.2: Difference between the camera optical frame and end-effector frame

2.2 Development of a Dynamic Vision-Based Controller

This chapter focuses on the development of a dynamic version of the vision-based controller and its integration with inverse dynamics controllers. The objective is to enable the robot to track reference velocities generated by the look-at-point vision-based control law, ensuring compatibility with the joint space and Cartesian space inverse dynamics controllers previously developed. Furthermore, the combined controller will allow the robot to jointly perform a linear position trajectory tracking task while maintaining the look-at-point vision-based task.

2.2.1 Controller Design

The dynamic vision-based controller exploits the look-at-point vision control law to generate reference velocities for the robot. These reference velocities are tracked using inverse dynamics controllers. A significant modification involves replacing the orientation error e_o, previously defined with respect to a fixed reference, with the orientation error generated by the vision-based controller.
In particular we computed the angle-axes of s_d by computing the scalar product ant the cross product.

```
KDL::Frame end_effector_frame;

    KDL::Rotation rotation_cam_ee = KDL::Rotation::RPY
       (0, -1.57, 3.14);


    Eigen::Vector3d s_axis = sd.cross(s);
    double s_angle =std::acos(sd.dot(s));
```

```

    KDL::Frame Marker_frame;
    Marker_frame.M =(robot_->getEEFrame()).M*
       rotation_cam_ee*(KDL::Rotation::Rot(toKDL(s_axis
       ), s_angle));
```

The code we use is the same developed in the previous homework with the following line modified:

```
Eigen::Vector3d o_error = computeOrientationError(
   toEigen(Marker_frame.M), toEigen(cartpos.M));
```

2.3 Simulation Results

The combined controller was tested through simulations. The robot was commanded to follow a predefined linear trajectory while keeping the marker centered in the camera frame.

2.3.1 Results

In the Github repository there is a video that shows that the trajectory is succesfully executed while still looking at the Aruco marker as expected.