Robotics Lab: Homework 2

Control your robot

Antonio Polito: https://github.com/P0l1702/
Maria Rosaria Imperato: https://github.com/MariaRosaria1
Gianmarco Ferrara: https://github.com/gianmarco-ferrara

This document contains the Homework 2 of the Robotics Lab class.

Control your robot

1. **Kinematic Control**

   (a) We created a config folder containing a `parameters.yaml` file containing the values of the variables to become ROS parameters:

```
/**:
    ros__parameters:
        traj_duration: 20.0
        acc_duration: 1.0
        total_time: 20.0
        trajectory_len: 100
        Kp: 3.0
        end_position_x: 1.0
        end_position_y: -0.4
        end_position_z: 0.8
```

   Then we modified the `ros2_kdl_node.cpp` by adding the declarations of these variables within the public `Iiwa_pub_sub` class constructor, responsible for initializing the parameters required by the `KDLPlanner` object.

```
declare_parameter("traj_duration", 1.5);
declare_parameter("acc_duration", 0.5);
get_parameter("traj_duration", traj_duration);
get_parameter("acc_duration", acc_duration);
RCLCPP_INFO(get_logger(),"Current trajectory duration is: '%f'", traj_duration);
RCLCPP_INFO(get_logger(),"Current acceleration duration is: '%f'", acc_duration);
```

   In contrast to the local planner parameters, other parameters like controller gains and execution time must be accessible by other class methods (e.g., the `cmd_publisher` timer callback).

```
declare_parameter("total_time", 1.5);
declare_parameter("trajectory_len", 150);
declare_parameter("Kp", 5.0);
get_parameter("total_time", this->total_time);
get_parameter("trajectory_len",this->trajectory_len);
get_parameter("Kp",this->Kp);
RCLCPP_INFO(get_logger(),"Current total time is: '%f'", this->total_time);
RCLCPP_INFO(get_logger(),"Current trajectory length is: '%d'", this->trajectory_len);
RCLCPP_INFO(get_logger(),"Current Kp is: '%f'", this->Kp);
```

   Then we created launch folder containing a `iiwa.control.launch.py` file that starts the `ros2_kdl_node`.

```
def generate_launch_description():
    pkg_name = 'ros2_kdl_package'
    config_file = os.path.join(
        get_package_share_directory(pkg_name),
        'config',
        'parameters.yaml'
    )

    params_file_arg = DeclareLaunchArgument(
        'params_file',
        default_value=config_file,
```

```
        description='Path to parameters YAML file'
    )

    cmd_interface_arg = DeclareLaunchArgument(
        'cmd_interface',
        default_value='position',
        description='Command interface (position | velocity | effort)'
    )

    ros2_kdl_node = Node(
        package=pkg_name,
        executable='ros2_kdl_node',
        name='ros2_kdl_node',
        output='screen',
        emulate_tty=True,
        parameters=[
            LaunchConfiguration('params_file'),
            {'cmd_interface': LaunchConfiguration('cmd_interface')}
        ]
    )

    return LaunchDescription([
        params_file_arg,
        cmd_interface_arg,
        ros2_kdl_node
    ])
```

Before launching, we had to modify the `CMake` file by including the `launch` and `config` folders. In order to test, first, we launched the `iiwa.launch.py` from the `iiwa_bringup` package. Then, we lunched the the `iiwa.control.launch.py` from the `ros2_kdl_package` package.

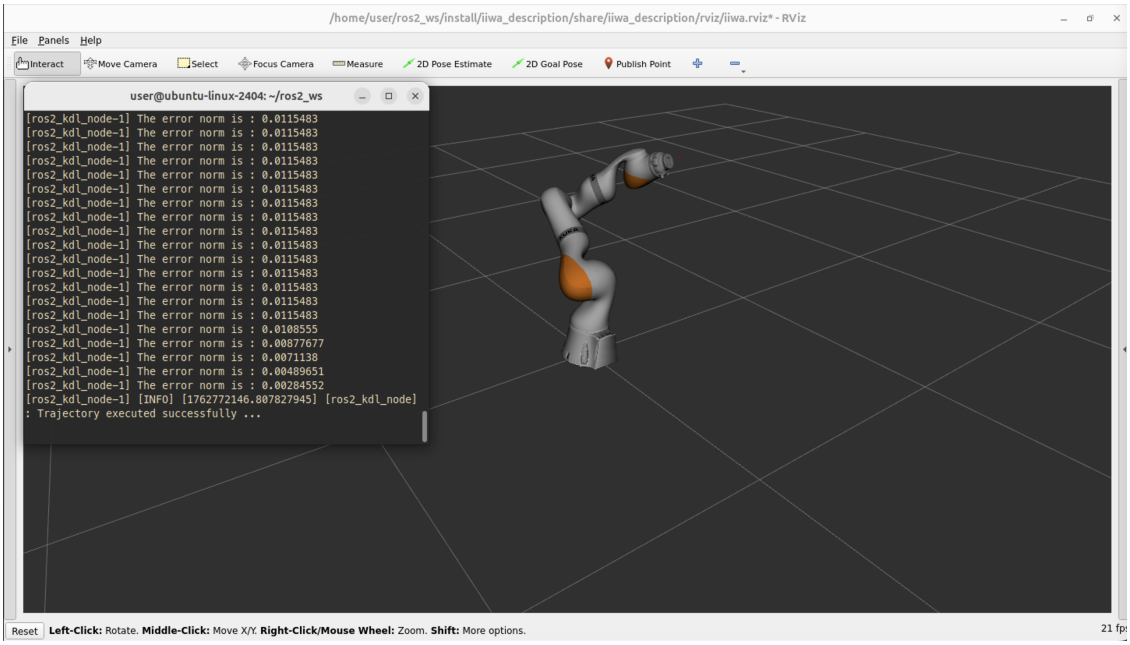


Figure 1: Position control. Video available at link

(b) We created a new controller called `velocity_ctrl_null` that implements the specified velocity control law. In details, we added the public method `velocity_ctrl_null` in the `kdl_control` class definition in the `kdl_control.h` file.

```
class KDLController
{
public:
    KDLController(KDLRobot &_robot);

    Eigen::VectorXd idCntr(KDL::JntArray &_qd,
                           KDL::JntArray &_dqd,
                           KDL::JntArray &_ddqd,
                           double _Kp,
                           double _Kd);
    Eigen::VectorXd velocity_ctrl_null(KDL::Frame& _desPos, double _Kp, double _lambda);
private:
    KDLRobot* robot_;
};
```

Then we implemented it in the `kdl_control.cpp` file:

```
Eigen::VectorXd KDLController::velocity_ctrl_null(KDL::Frame& _desPos, double _Kp, double _lambda)
{
    unsigned int n = robot_->getNrJnts();
    Eigen::VectorXd q = robot_->getJntValues();
    KDL::Frame current_pose = robot_->getEEFrame();
    Eigen::MatrixXd J = robot_->getEEJacobian().data;

    KDL::Twist e_twist = KDL::diff(current_pose, _desPos);
    Eigen::Matrix<double, 6, 1> ep;
    ep(0) = e_twist.vel.x();
    ep(1) = e_twist.vel.y();
    ep(2) = e_twist.vel.z();
    ep(3) = e_twist.rot.x();
    ep(4) = e_twist.rot.y();
    ep(5) = e_twist.rot.z();

    Eigen::Matrix<double, 6, 6> Kp_matrix = Eigen::Matrix<double, 6, 6>::Identity() * _Kp;
    Eigen::MatrixXd J_pinv = J.completeOrthogonalDecomposition().pseudoInverse();
    Eigen::VectorXd q0_dot = Eigen::VectorXd::Zero(n);
    Eigen::MatrixXd joint_limits = robot_->getJntLimits();
    for (unsigned int i = 0; i < n; ++i)
    {
        double q_i = q(i);


        double q_min_i = joint_limits(i, 0);
        double q_max_i = joint_limits(i, 1);
        double den_H = (q_max_i - q_i) * (q_i - q_min_i);
        if (std::abs(den_H) < 1e-6)
        {
            q0_dot(i) = 0.0;
        }
        else
        {
            double C_i = std::pow(q_max_i - q_min_i, 2) / _lambda;
```

```
            double numerator = C_i * (2*q_i-q_max_i - q_min_i );
            double common_denominator_sq = std::pow(den_H, 2);
            q0_dot(i) = -numerator / common_denominator_sq;
        }
    }
    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);
    Eigen::MatrixXd null_space_projector = I - (J_pinv * J);
    Eigen::VectorXd q_dot = (J_pinv * Kp_matrix * ep) + (null_space_projector * q0_dot);
    return q_dot;
}
```

We added `lambda` in the `parameters.yaml` file and modified the `ros2_kdl_node.cpp` by adding two new private variables `ctrl_mode_` and `lambda_`.

```
    declare_parameter("ctrl", "velocity_ctrl");
    get_parameter("ctrl", this->ctrl_mode_);
    RCLCPP_INFO(get_logger(), "Control mode selected: '%s'", this->ctrl_mode_.c_str());


    declare_parameter("lambda", 1.0);
    get_parameter("lambda", this->lambda_);
    RCLCPP_INFO(get_logger(), "Lambda value for null-space control: '%f'", this->lambda_);
```

where `lambda_` will be passed to the `velocity_ctrl_null` and `ctrl_mode_` will allow the user to select the controller.

```
    else if(cmd_interface_ == "velocity"){
        if (this->ctrl_mode_ == "velocity_ctrl")
        {
            Vector6d cartvel; cartvel << p_.vel + Kp*error, o_error;
            joint_velocities_cmd_.data =
            pseudoinverse(robot_->getEEJacobian().data)*cartvel;
        }
        else if (this->ctrl_mode_ == "velocity_ctrl_null")
        {
            joint_velocities_cmd_.data =
            this->controller_->velocity_ctrl_null(desFrame, this->Kp, this->lambda_);
        }
    }
```

We also modified the launch file by adding `ctrl_arg` and `cmd_interface_arg` arguments:

```
    ctrl_arg = DeclareLaunchArgument(
        'ctrl',
        default_value='velocity_ctrl',
        description='Controller type (velocity_ctrl | velocity_ctrl_null)'
    )
    cmd_interface_arg = DeclareLaunchArgument(
        'cmd_interface',
        default_value='position',
        description='Command interface (position | velocity | effort)'
    )
```

Passing them to the node parameters and launching it:

```
    ros2_kdl_node = Node(
    package=pkg_name,
    executable='ros2_kdl_node',
    name='ros2_kdl_node',
```

```
    output='screen',
    emulate_tty=True,
    parameters=[
        LaunchConfiguration('params_file'),
        {'ctrl': LaunchConfiguration('ctrl')},
        {'cmd_interface': LaunchConfiguration('cmd_interface')}])
    return LaunchDescription([
    params_file_arg,
    ctrl_arg,
    cmd_interface_arg,
    ros2_kdl_node])
```

So when we launch we must specifiy the `cmd_interface_arg` (`position|velocity|effort`) and the `ctrl_arg` (`velocity_ctrl` for proportional velocity control or `velocity_ctrl_null` for the other control law).

We tested both the control laws with the trajectory whose parameters are specified in the `.yaml` file reported before.

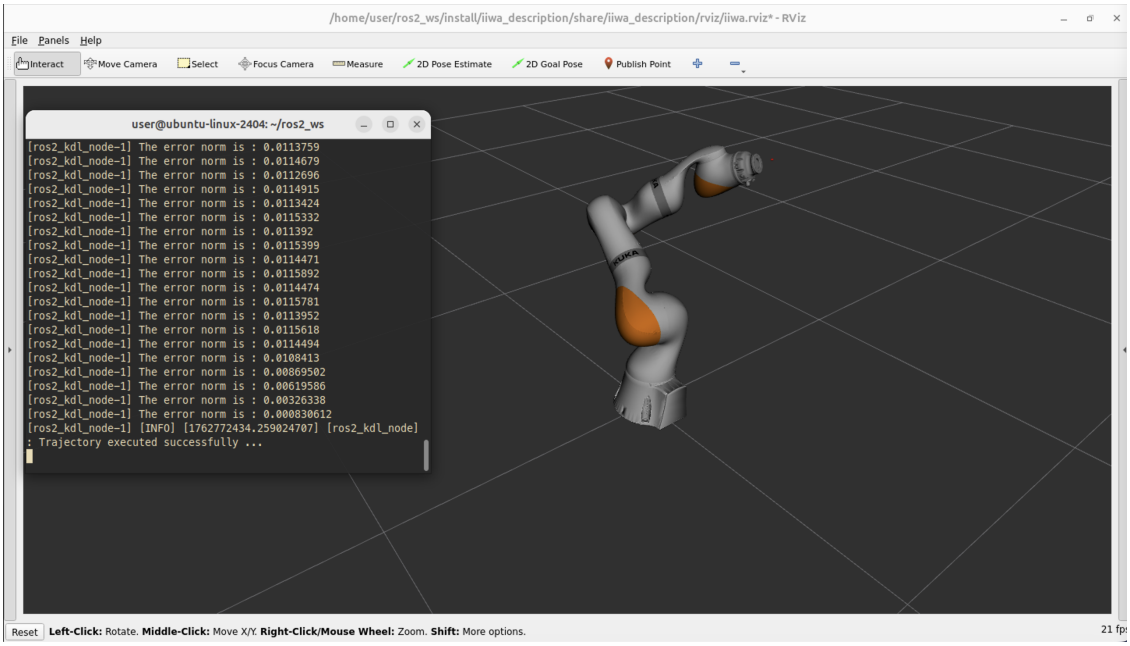


Figure 2: Jacobian pseudo-inverse velocity control. Video available at link

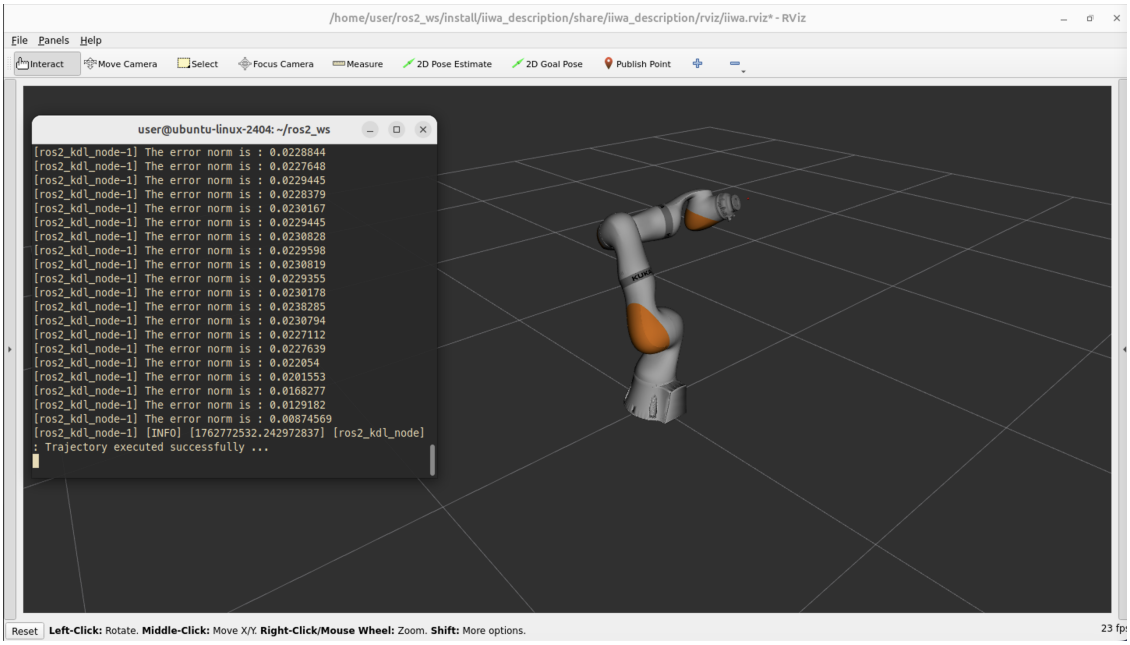
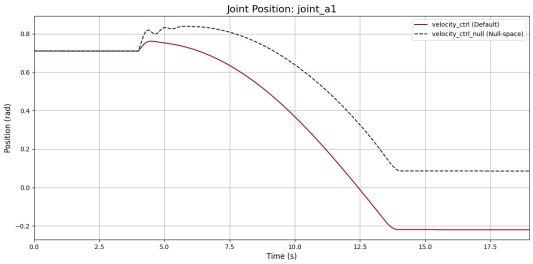
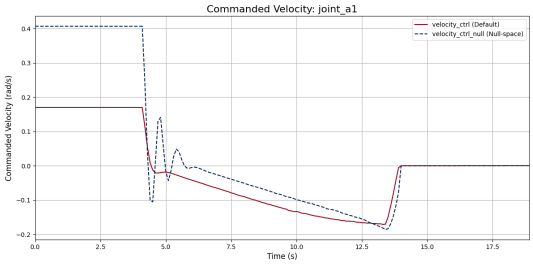
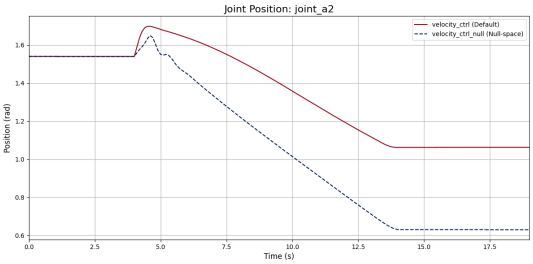


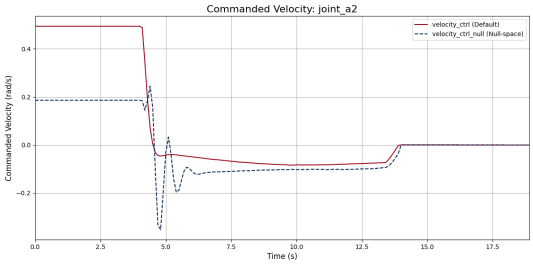
Figure 3: Alternative velocity control. Video available at link

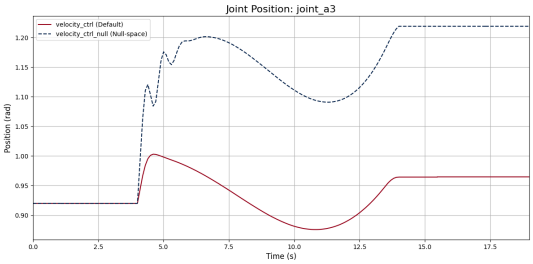
In the following we report the time histories of joints positions (left column) and commanded velocities (right column); the continous red lines refer to the proportional velocity control, while the dashed blue lines refer to the alternative velocity control law. Note that in both cases the joint variables reach the steady-state in about 10 seconds, as specified by the parameter `traj_duration`.

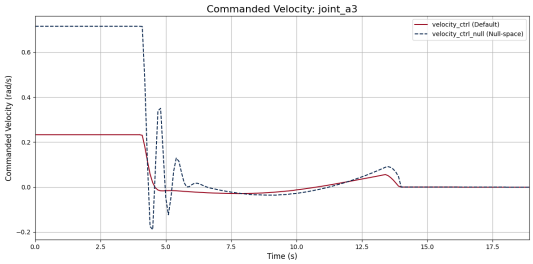

Joint Position: joint_a1
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Position (rad)
Time (s)

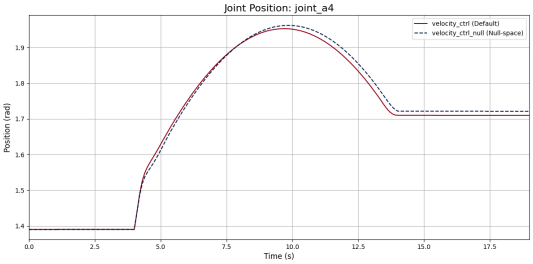

Commanded Velocity: joint_a1
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Commanded Velocity (rad/s)
Time (s)

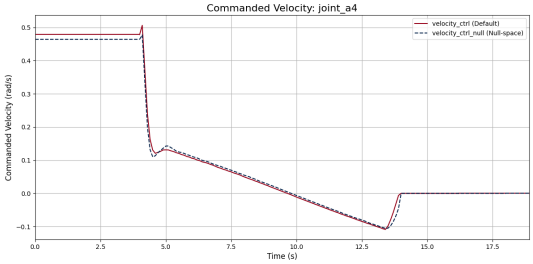

Joint Position: joint_a2
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Position (rad)
Time (s)

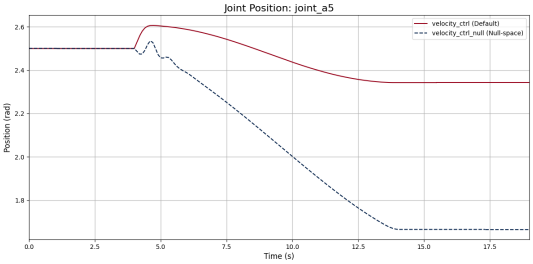

Commanded Velocity: joint_a2
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Commanded Velocity (rad/s)
Time (s)

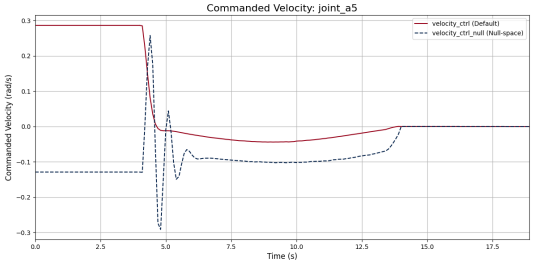

Joint Position: joint_a3
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Position (rad)
Time (s)

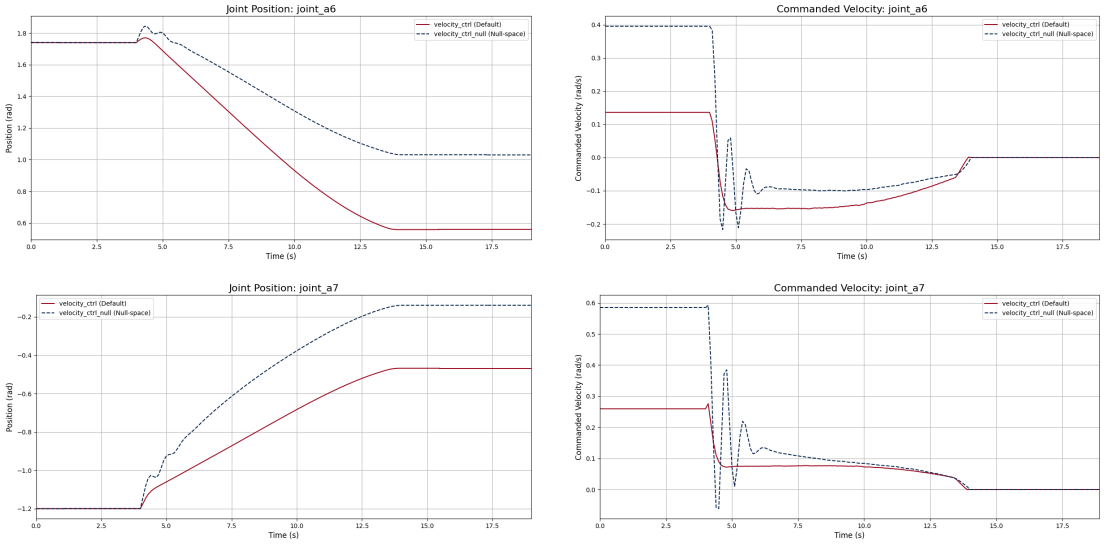

Commanded Velocity: joint_a3
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Commanded Velocity (rad/s)
Time (s)


Joint Position: joint_a4
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Position (rad)
Time (s)


Commanded Velocity: joint_a4
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Commanded Velocity (rad/s)
Time (s)


Joint Position: joint_a5
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Position (rad)
Time (s)


Commanded Velocity: joint_a5
velocity_ctrl (Default)
velocity_ctrl_null (Null-space)
Commanded Velocity (rad/s)
Time (s)



To obtain these plots we run a simulation with each control law and we saved the corresponding data in a bag file:

```
ros2 bag record -o bag_name <topic>
```

Then we converted it in csv format and used a Phyton code (avaible at the Git repository) to plot the time history for the variables of interest.

(c) We created the `kdl_interfaces` package that contains an `action` directory in which there is a `Tracjectory.action` file. In this action file we defined the structure of goal, result and feedback messages.

```
geometry_msgs/Point end_position
float64 traj_duration
float64 acc_duration
float64 total_time
int32 trajectory_len
float64 kp
float64 lambda_val
string traj_type
string s_type
string ctrl
---
bool success
string message
---
float64 position_error_magnitude
geometry_msgs/Point current_position
```

Following the tutorial at the ROS2 documentation, we recasted the `kdl_node` from the `ros2_kdl_package` into a new node called `kdl_action_server`. According to the standard `rclcpp_action` interface, in the server are defined four main callback functions:

- **handle_goal()**: called when a new goal request is received. It verifies the validity of the goal parameters and decides whether to accept or reject it.
- **handle_cancel()**: executed when a client requests to cancel the current goal.
- **handle_accepted()**: invoked immediately after a goal has been accepted; it spawns a new thread to execute the trajectory.

- **execute_trajectory()**: the main execution routine, running in a separate thread. It publishes the joint or Cartesian commands at each control step and periodically sends feedback to the client about the current robot position and tracking error (thus, it behaves like a node adding feedback messages).

The code implementing the server is available at the Git repo (for the sake of brevity it is not reported here).

Then we developed a client in order to test the above mentioned functionalities. First of all, in the client the goal message is built by using the parameters that are read from the `parameters.yaml` file (passed in the launch file) - the same that was read by the `ros2_kdl_node` in the previous point. Then the message is sent to the server. Thus, three main callbacks are provided:

- **goal_response_callback()**: invoked immediately after the goal is sent, this callback informs whether the server has accepted or rejected the goal request.
- **feedback_callback()**: executed periodically during the action execution, it provides intermediate updates from the server, such as position error or the current end-effector position.
- **result_callback()**: triggered once at the end of the action, it receives the final result from the server, indicating whether the goal was successfully completed, failed, or canceled.

In summary, the `send_goal_options` object defines how the client manages the asynchronous communication with the action server throughout the goal execution process.

The code implementing the client is available at the Git repo (for the sake of brevity it is not reported here).

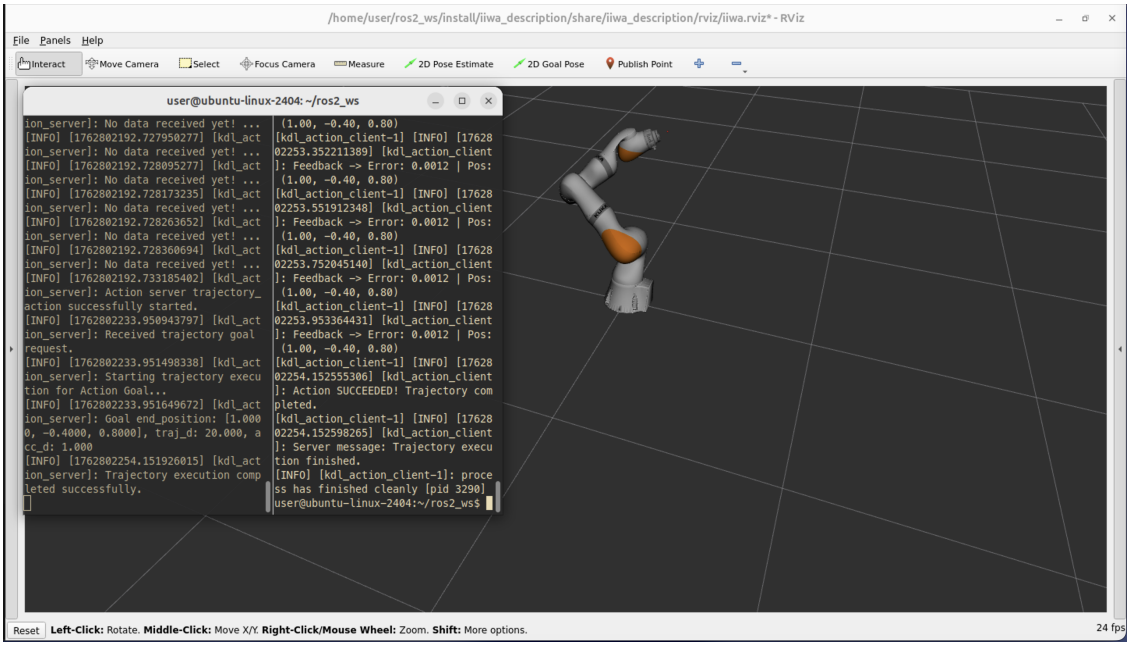


Figure 4: Jacobian pseudo-inverse velocity control with client-server communication. Video available at link

2. Vision-based Control

(a) We created a folder `gazebo/models` containing the aruco maker model `aruco_iiwa.png` generated at the link. Then we created a new model into the `model.sdf` file named `arucotag`:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.9">
```

```
  <model name="arucotag">
    <static>true</static>
     <pose>1.14 -0.08 0.79 1.90 0.00 2.33</pose><link name="base">
      <visual name="tag_visual">
        <geometry>
          <box>
            <size>0.1 0.1 0.001</size>
          </box>
        </geometry>
        <material>
          <ambient>1 1 1 1</ambient>
          <diffuse>1 1 1 1</diffuse>
          <pbr>
            <metal>
              <albedo_map>model://arucotag/aruco_iiwa.png</albedo_map>
            </metal>
          </pbr>
        </material>
      </visual>
      <collision name="tag_collision">
        <geometry>
          <box>
            <size>0.1 0.1 0.001</size>
          </box>
        </geometry>
      </collision>
    </link>
  </model>
</sdf>
```

After that we imported it into a gazebo world file named `aruco_world.world`.

```
    <include>
      <uri>
        model://arucotag
      </uri>
      <name>arucotag</name>
        <pose>1.14 -0.07 0.79 1.90 0.00 2.33</pose>
    </include>
```

In order to test the successful loading of the model into the `aruco_world.world` file we tested it with:

```
    ign gazebo -r src/ros2_iiwa/iiwa_description/gazebo/worlds/aruco_world.world
```

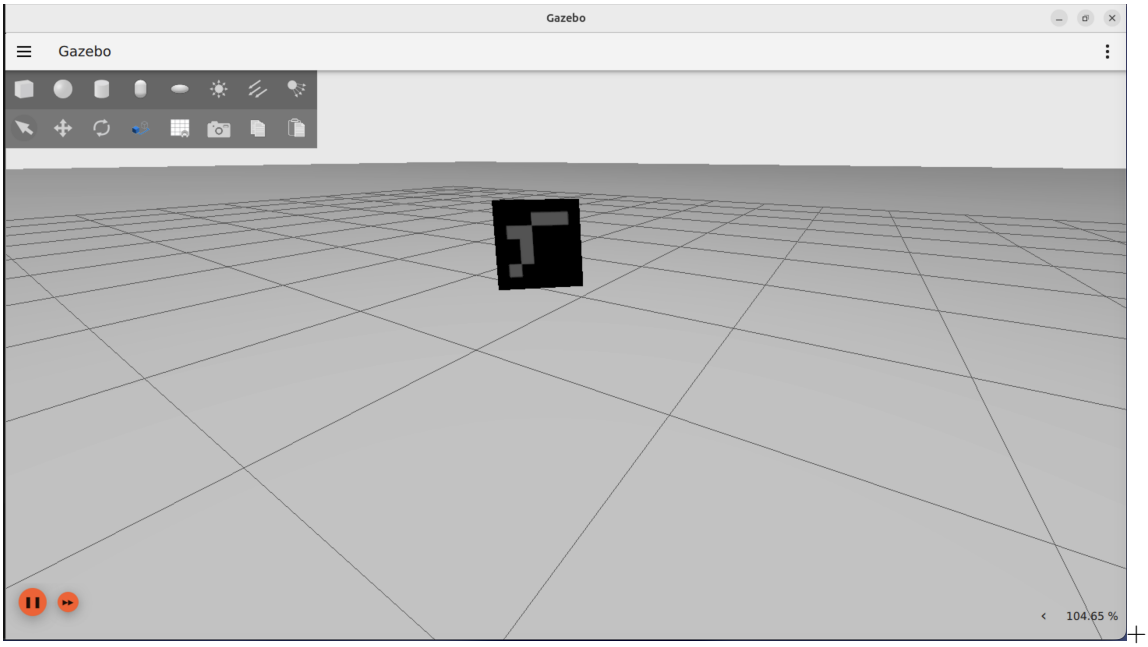


Figure 5: Aruco marker loaded in the gazebo world

In order to implement a vision controller, a camera is needed. So we had to implement a camera link and a camera joint into the `iiwa.config.xacro` file.

```
<link name="$(arg prefix)camera_link">
    <visual>
      <geometry>
        <box size="0.005 0.005 0.005"/>
      </geometry>
      <material name="black"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.005 0.005 0.005"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>
    </inertial>
  </link>

  <joint name="$(arg prefix)camera_joint" type="fixed">
    <parent link="$(arg prefix)link_7"/>
    <child link="$(arg prefix)camera_link"/>
    <origin xyz="0 0 0.2" rpy="0 -1.5708 0"/>
  </joint>

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

```
</gazebo>

<gazebo reference="$(arg prefix)camera_link">
  <sensor name="camera_sensor" type="camera">
    <camera>
      <horizontal_fov>1.39626</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
    </camera>
    <always_on>1</always_on>
    <update_rate>30</update_rate>
    <visualize>true</visualize>

    <topic>/camera</topic>

  </sensor>
</gazebo>
```

Then we spawned the needed nodes with the `iiwa_world_vision.launch.py`.

```
  pkg_ros_gz_sim = get_package_share_directory('ros_gz_sim')
  pkg_iiwa_description = get_package_share_directory('iiwa_description')


  gazebo_launch = IncludeLaunchDescription(
      PythonLaunchDescriptionSource(
          os.path.join(pkg_ros_gz_sim, 'launch', 'gz_sim.launch.py')
      ),
      launch_arguments={
          'gz_args': '-r ' + aruco_model_sdf,
          'publish_clock': 'true'
      }.items(),
  )

  xacro_file_name = 'iiwa.config.xacro'
  xacro = os.path.join(get_package_share_directory('iiwa_description'), "config", xacro_file_name)

  robot_description_param = {"robot_description": Command(['xacro ', xacro])}

  load_robot_description_node = Node(
      package='robot_state_publisher',
      executable='robot_state_publisher',
      output='screen',
      parameters=[robot_description_param, {"use_sim_time": True}],
  )

  spawn_entity_node = Node(
```

```
        package='ros_gz_sim',
        executable='create',
        output='screen',
        arguments=[
            '-topic', '/robot_description',
            '-name', 'iiwa'
        ],
    )

    clock_bridge = Node(
     package="ros_ign_bridge",
     executable="parameter_bridge",
     arguments=['/clock@rosgraph_msgs/msg/Clock[ignition.msgs.Clock'],
     output="screen",)

    camera_bridge = 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',
    )

    joint_state_broadcaster_node = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["joint_state_broadcaster", "--controller-manager", "/controller_manager"],
    )

    robot_controller_spawner = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["velocity_controller", "--controller-manager", "/controller_manager"],
    )

    spawn_jsb_handler = RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=spawn_entity_node,
            on_exit=[joint_state_broadcaster_node],
        )
    )

    spawn_controller_handler = RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=joint_state_broadcaster_node,
            on_exit=[robot_controller_spawner],
        )
    )
    pkg_aruco_ros = get_package_share_directory('aruco_ros')
```

```
    aruco_launch = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        os.path.join(pkg_aruco_ros, 'launch', 'single.launch.py')
    ),
    launch_arguments={
        'marker_size': '0.1',
        'marker_id': '14'
    }.items(),
)


    return LaunchDescription([
        gazebo_launch,
        load_robot_description_node,
        spawn_entity_node,
        clock_bridge,
        camera_bridge,
        spawn_jsb_handler,
        spawn_controller_handler
    ])
```

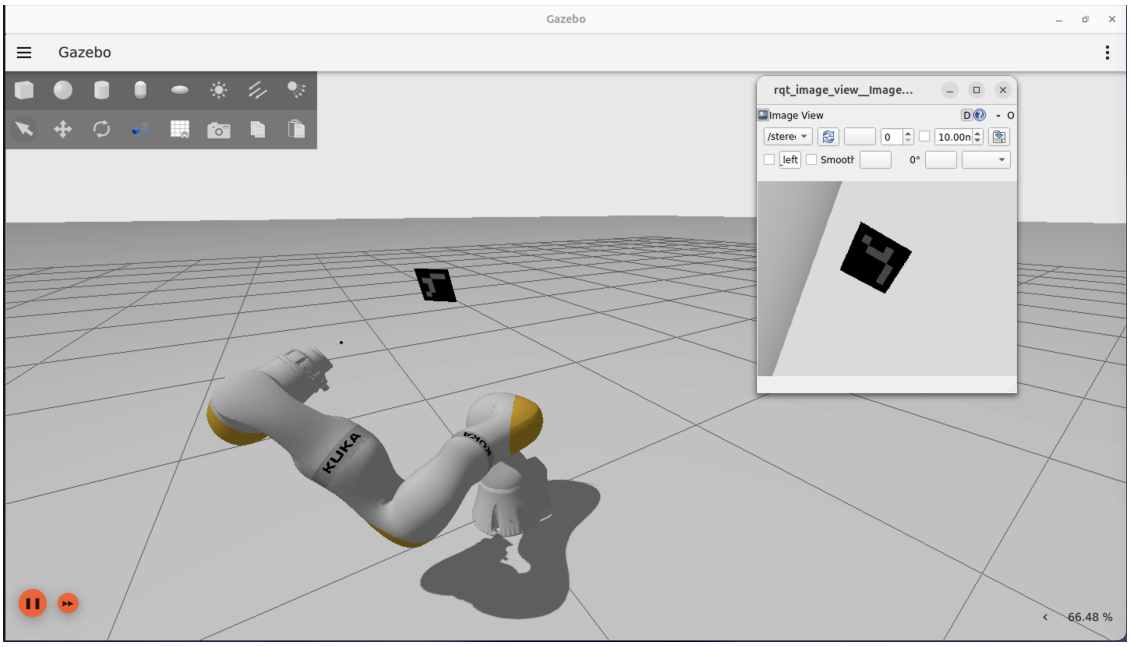


Figure 6: Aruco marke and iiwa robot with camera loaded in the gazebo world

(b) In the previous point we have spawned the robot with the velocity command interface into the world containing the aruco tag. It is then possible launching the `single.launch.py` from the `aruco_ros` package to detect the tag with the command:

```
ros2 launch aruco_ros single.launch.py marker_size:=0.1 marker_id:=14
```

in order to load the correct aruco tag.

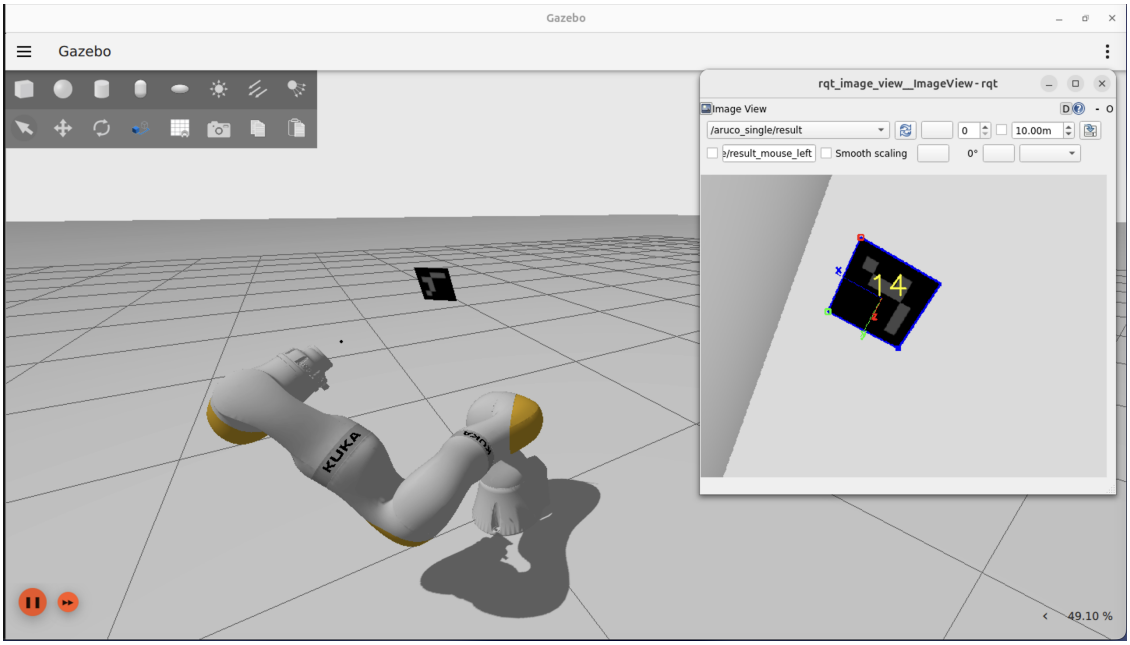


Figure 7: Marker detection by `aruco_ros`

In order to start the vision control, we created a `ros2_kdl_node_vision` node. Then we created a subscriber to the `/aruco_single/pose` topic.

```
image_sub_ = this->create_subscription<geometry_msgs::msg::PoseStamped>("/aruco_single/pose",
100, std::bind(&Iiwa_pub_sub::imageCallback, this, std::placeholders::_1));
```

With the `imageCallback` function defined as:

```
    void imageCallback(const geometry_msgs::msg::PoseStamped& msg) {
        RCLCPP_INFO_ONCE(this->get_logger(), "===> Aruco pose received! <===");
        aruco_pose_available_ = true;
        const auto position = msg.pose.position;
        const auto orientation = msg.pose.orientation;
        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
        );
        pose_in_camera_frame.M = kdl_rotation;
        pose_in_camera_frame.p = kdl_position;
    }
```

Then the function `cmd_publisher()` computes the control law.

```
    void cmd_publisher(){
        robot_->update(toStdVector(joint_positions_.data),toStdVector(joint_velocities_.data));
        double total_time = 50.0;
        int trajectory_len = total_time * 100;
        double dt = 1.0 / (trajectory_len / total_time);
        t_ += dt;
        if (t_ < total_time){
        KDL::Frame cartpos = robot_->getEEFrame();
        J_cam = robot_->getEEJacobian();
        Eigen::VectorXd q0_dot = Eigen::VectorXd::Zero(nj);
        KDL::Frame cartpos_camera = cartpos * KDL::Frame(
```

```
        KDL::Rotation::RotY(-1.5708),
        KDL::Vector(0.0, 0.0, 0.02)
        );
        KDL::Jacobian J_c_camera(J_cam.columns());
        KDL::changeBase(J_cam, cartpos_camera.M, J_c_camera);
        joint_velocities_.data = controller_->vision_ctrl(
        pose_in_camera_frame,
        cartpos_camera,
        J_c_camera,
        q0_dot
        );
        robot_->update(toStdVector(joint_positions_.data),toStdVector(joint_velocities_.data));
        for (long int i = 0; i < nj; ++i) {
            desired_commands_[i] = joint_velocities_(i);
        }
    }
        else{
            RCLCPP_INFO_ONCE(this->get_logger(), "===> Trajectory completed.
            Sending zero velocity commands. <===");
            for (long int i = 0; i < nj; ++i) {
            desired_commands_[i] = 0.0;
        }
}
}
```

In the `kdl_control` class of the `ros2_kdl_package` we created a vision-based controller called `vision_ctrl` for the simulated iiwa robot.

```
    Eigen::VectorXd KDLController::vision_ctrl(
    KDL::Frame pose_in_camera_frame,
    KDL::Frame camera_frame,
    KDL::Jacobian camera_jacobian,
    Eigen::VectorXd q0_dot)
{
    Matrix6d R = spatialRotation(camera_frame.M);
    Eigen::Vector3d c_P_o = toEigen(pose_in_camera_frame.p);
    Eigen::Vector3d s = c_P_o / c_P_o.norm();
    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;
    Eigen::MatrixXd J_c = camera_jacobian.data;
    Eigen::MatrixXd LJ = L * J_c;
    Eigen::MatrixXd LJ_pinv = LJ.completeOrthogonalDecomposition().pseudoInverse();
    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(J_c.cols(), J_c.cols());
    Eigen::MatrixXd N = I - (LJ_pinv * LJ);
    Eigen::Vector3d s_d(0, 0, 1);
    double k = 1;
    Eigen::VectorXd joint_velocities = k * LJ_pinv * s_d + N * q0_dot;
    Eigen::Vector3d s_error = s - s_d;
    std::cout << "Error norm (s_error): " << s_error.norm() << std::endl;
    return joint_velocities;
}
```

The tracking capability of the controller was tested by manually moving the aruco marker via the gazebo user interface.

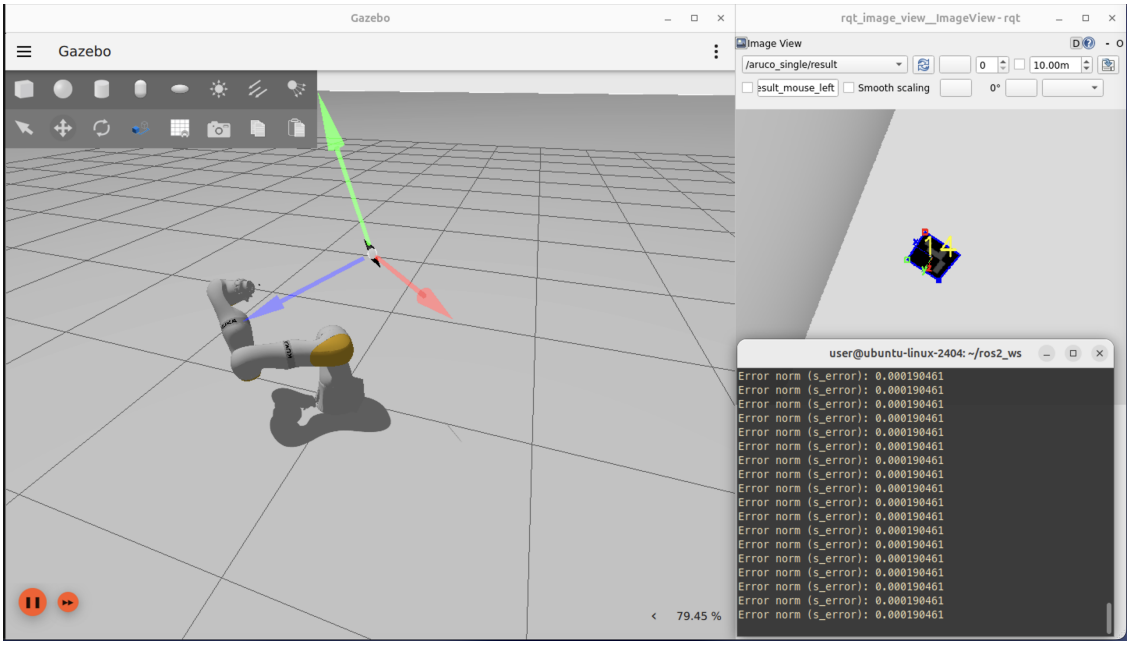
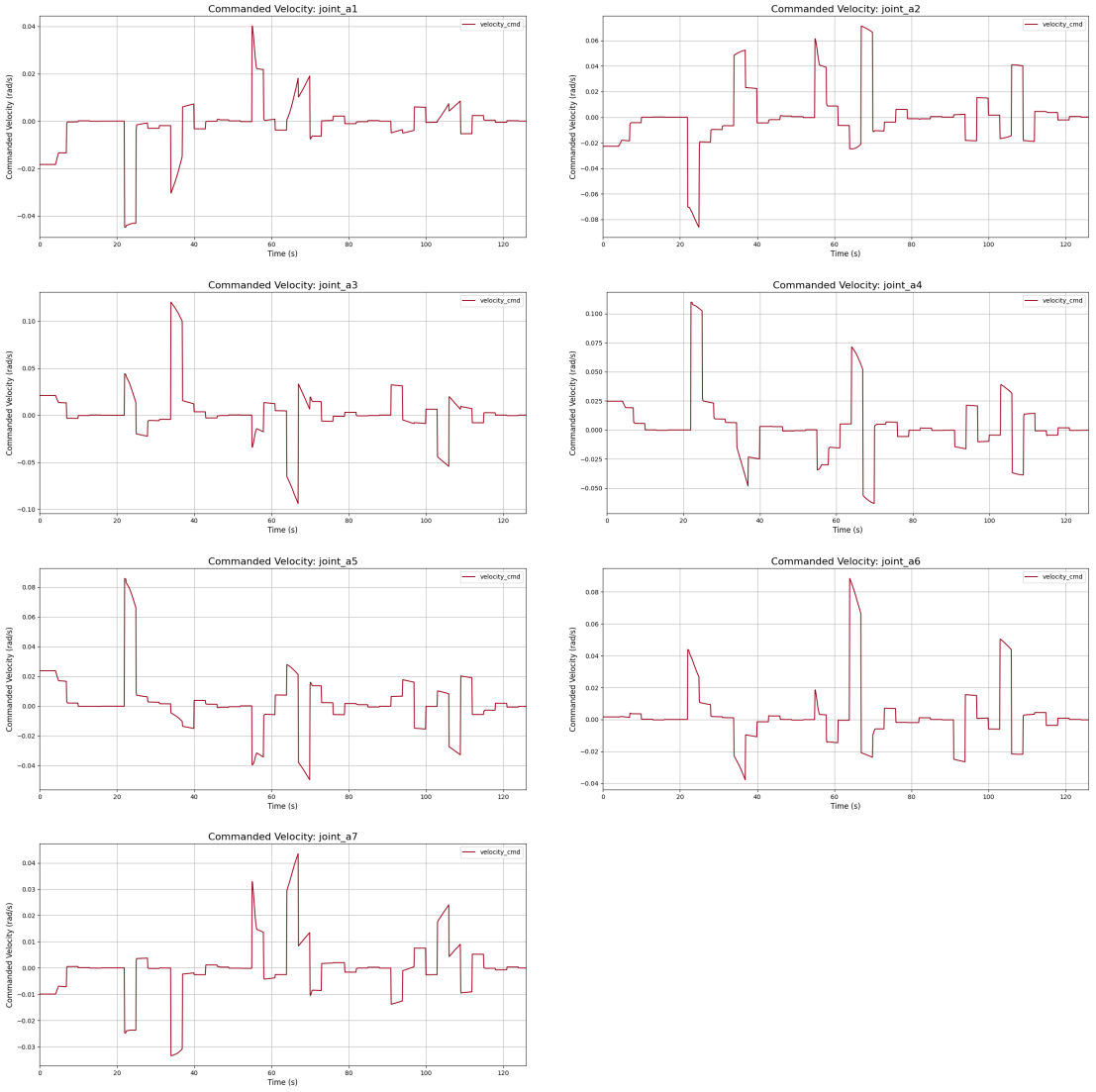


Figure 8: Robot executing a look-at-point task (vision-based control). Video available at link

We report here the time history of the velocity commands sent to the robot.



(c) A ROS2 service bridge was created to allow updating the position of the Aruco marker within the Gazebo simulation environment. This was achieved by bridging the native Ignition service `/set_pose` through a `parameter bridge`, as shown below:

```
service_bridge_node = Node(
    package='ros_gz_bridge',
    executable='parameter_bridge',
    name='set_pose_bridge',
    arguments=[
        '/world/default/set_pose@ros_gz_interfaces/srv/SetEntityPose'
    ],
    output='screen')
```

This bridge exposes the Gazebo service to the ROS2 interface. The service can then be tested using the following command, which moves the entity named `arucotag` to the desired position and orientation in the simulation:

```
    ros2 service call /world/default/set_pose ros_gz_interfaces/srv/SetEntityPose "{
    entity: {
        name: 'arucotag',
```

```
    type: 1
},
pose: {
    position: {x: 1.0, y: 1.5, z: 0.5},
    orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}
}
}"
```

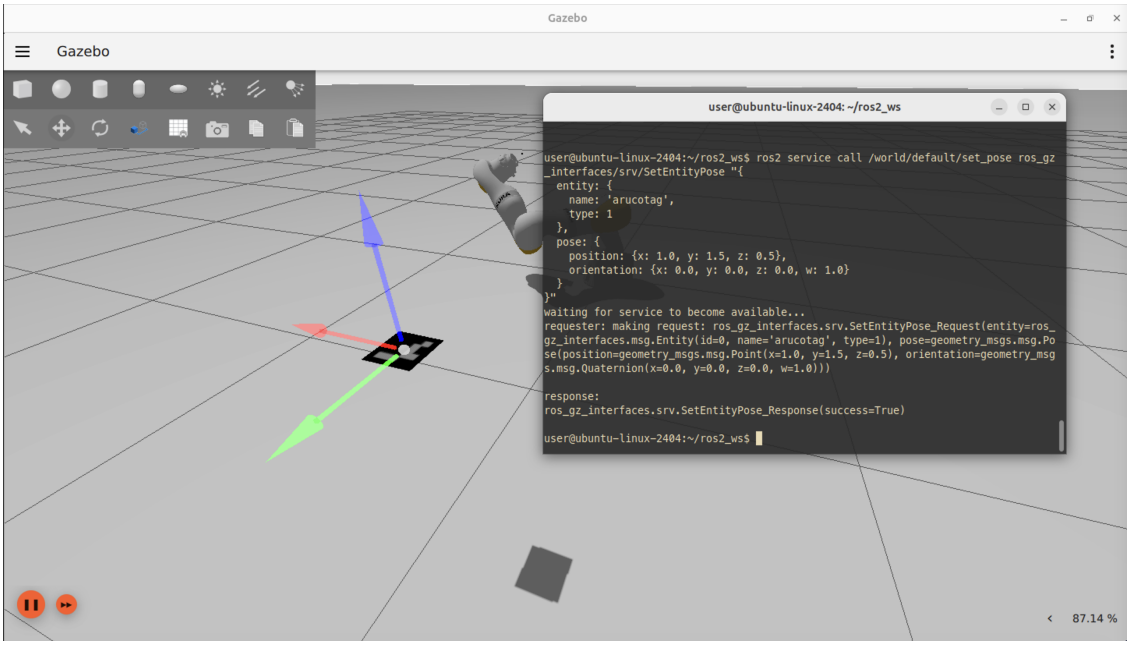


Figure 9: Service call.