Robotics Homework_02

Control Your Robot

Mario Selvaggio

Federica Pirozzi P38000350: `https://github.com/Federica2103`
Francesco Lionetti P38000365: `https://github.com/FrancescoLionetti`

12 November 2025

Control your robot

The goal of this homework is the development and implementation of kinematic and vision-based controllers for the robotic manipulator arm (KUKA iiwa) in a simulation environment.
To begin, we downloaded the packages into the ROS2 workspace using the following commands:

```
$ cd ~/ros2_ws/src
$ git clone https://github.com/RoboticsLab2025/ros2_kdl_package.git
$ git clone https://github.com/RoboticsLab2025/ros2_iiwa.git
```

1. **Kinematic control**

   (a) *Request: Modify the* `ros2_kdl_node` *such that the following variables become ROS2 parameters:* `traj_duration`, `acc_duration`, `total_time`, `trajectory_len`, `Kp`, *and the three components of the trajectory* `end_position`. *Create a launch file that starts the* `ros2_kdl_node` *loading a* `.yaml` *file (from a* `config` *folder) that contains the aforementioned parameters' definition. Add the launch command to the* `README.md` *file in your repo.*

   `EXECUTION`

   The main goal of the first task is to make the `ros2_kdl_node` more flexible and configurable, avoiding the need to modify the source code every time the controller or trajectory parameters must be changed. In the original implementation, some variables were directly defined inside the C++ code. Although functional, this approach lacked modularity: every change required editing the source file and recompiling the entire package, making the testing and tuning process slower and less efficient.

   To overcome this limitation, this point introduces the use of **ROS 2 parameters**, allowing these variables to be dynamically loaded when the node starts. Specifically, the node is modified to declare and retrieve these parameters through the ROS 2 parameter API, enabling users to configure the system behavior simply by editing an external configuration file rather than the source code. All the required parameters are defined inside a dedicated YAML file placed in the `config` folder.

   In addition, a **launch file** is created to automatically start the `ros2_kdl_node` while loading the YAML configuration at runtime.

   The modifications made in the `.cpp` file are the following:

   - To share parameter values between the **constructor** (where they are read) and the **timer callback** (where they are used), private member variables were added to the class (e.g., `traj_duration_`, `Kp_`, `end_position_vec_`).
   - The variables (`traj_duration`, `acc_duration`, `total_time`, `trajectory_len`, `Kp`, `end_position`) were declared using `declare_parameter()` and their values were read from the `.yaml` file (or defaults) and saved into the member variables using `get_parameter()`.
   - The `end_position` parameter is read as a `std::vector<double>` and then manually converted to an `Eigen::Vector3d`. Fallback logic was added to use a default value if the parameter is invalid.
   - `KDLPlanner` is now initialized using the `traj_duration_` and `acc_duration_` member variables.
   - `RCLCPP_INFO` logs were added to print all loaded parameters to the screen, confirming the operation's success.

ros2_kdl_node.cpp

```
//Declare parameters asked in the homework
//Total duration of trajectory (in seconds)
declare_parameter("traj_duration", 1.5);  //Register
    traj_duration with default value 1.5
get_parameter("traj_duration", traj_duration);  //Read
    traj_dur value and update member variable

//Duration of acceleration/deceleration phases (in
    seconds)
declare_parameter("acc_duration", 0.5);
get_parameter("acc_duration", acc_duration);

//Total simulation time (in seconds)
declare_parameter("total_time", 1.5);
get_parameter("total_time", total_time);

//Control loop total execution time (seconds)
declare_parameter("trajectory_len", 150);
get_parameter("trajectory_len", trajectory_len);

//Proportional gain for position control (Kp)
declare_parameter("Kp", 5);
get_parameter("Kp", Kp);

//Desired final position [x, y, z] in meters (in the base
     frame)
declare_parameter("end_position_vec", std::vector<double
    >{});
get_parameter("end_position_vec", end_position_vec);

//Print messages
RCLCPP_INFO(get_logger(), "Parameters loaded:");
RCLCPP_INFO(get_logger(), "  traj_duration: %f",
    traj_duration);   //Print node's read value
RCLCPP_INFO(get_logger(), "  acc_duration: %f",
    acc_duration);
RCLCPP_INFO(get_logger(), "  total_time: %f", total_time)
    ;
RCLCPP_INFO(get_logger(), "  trajectory_len: %d",
    trajectory_len);
RCLCPP_INFO(get_logger(), "  Kp: %d", Kp);
if(end_position_vec.size() == 3) {
     RCLCPP_INFO(get_logger(), "  end_position_vec: [%f, %
         f, %f]",
          end_position_vec[0], end_position_vec[1],
              end_position_vec[2]);
     } else {
          RCLCPP_WARN(get_logger(), "  end_position_vec:
              not set or invalid size. Will use default
              logic.");
     }

---

Eigen::Vector3d end_position;
if (end_position_vec.size() == 3) {
      end_position << end_position_vec[0],
```

```
            end_position_vec[1], end_position_vec[2];
    } else {
        //Revert to original logic if parameter read failed
            or is incorrect
        RCLCPP_WARN(get_logger(), "Using original relative
            logic for end_position.");
         end_position << init_position[0], -init_position[1],
              init_position[2];
    }

    ---

    //Member variables for parameters
    double traj_duration;
    double acc_duration;
    double total_time;
    int trajectory_len;
    int Kp;
    std::vector<double> end_position_vec;
```

We created a config folder to put the yaml file in it with the following commands:

```
$ cd ~/ros2_ws/src/ros2_kdl_package
$ mkdir config
$ cd config
$ touch params.yaml
```

ros2_kdl_params.yaml

```
param_node:
ros__parameters:
    traj_duration: 1.5
    acc_duration: 0.5
    total_time: 1.5
    trajectory_len: 150
    Kp: 5
    end_position_vec: [0.4, 0.3, 0.5]
```

The assigned parameter values strictly adhere to the default configurations specified within the `ros2_kdl_node` definition, ensuring consistency and system stability. This initial setup facilitates immediate verification of the control node's baseline functionality. A crucial modification was implemented solely for the **end-effector's final Cartesian position**. This specific parameter was intentionally altered from its default state to a distinctly different target location. The primary purpose of this adjustment is not merely a test of control accuracy, but rather to provide a clear, visual demonstration of the robot's kinematic motion and the trajectory generation capabilities under the current configuration.

kdl_node_launch.py

```
def generate_launch_description():

# Define 'params' variable holding the path to the
```

```
        parameter YAML file
    params = PathJoinSubstitution(
        [FindPackageShare('ros2_kdl_package'), 'config', '
            param.yaml']
    )

    # Node Definition
    ros2_kdl_node = Node(
        package='ros2_kdl_package',      # Executable path
        executable='ros2_kdl_node',      # Cpp executable name
        name='param_node',               # Node name within
            the system
        output='both',                   # Prints output/logs
            to both screen and file
        parameters=[params]              # Pass path to
            parameters file
    )

    return LaunchDescription([
        ros2_kdl_node
    ])
```

We also modified the `CMakeLists.txt` file to install the launch and config folders when the package is built:

CMakeLists.txt

```
# Install launch files
install(DIRECTORY
launch config
DESTINATION share/${PROJECT_NAME}
)
```

After these modifications, we built the workspace and started the simulation.

```
TERMINAL 1:
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
$ ros2 launch iiwa_bringup iiwa.launch.py

TERMINAL 2:
$ source install/setup.bash
$ ros2 launch ros2_kdl_package kdl_node.launch.py
```

The simulation result can be viewed in the following video link: `https://youtu.be/RPZHueAKIWU`.

(b) *Request: Create a new controller in the* ***`kdl_control`*** *class called* ***`velocity_ctrl_null`*** *that implements the following velocity control law:*

$$\dot{q} = J^{\dagger} K_p e_p + (I - J^{\dagger} J)\dot{q}_0 \tag{1}$$

where $J^{\dagger}$ *is the Jacobian pseudoinverse,* e_p *is the position error and* $\dot{q}_0$ *is the joint velocity that keeps the manipulator far from joint limits*

$$\dot{q}_0 = \nabla V \sum_{i=1}^{n} \frac{1}{\lambda} \frac{(q_i^+ - q_i^-)^2}{(q_i(t) - q_i^-)(q_i^+ - q_i(t))} \tag{2}$$

where λ is a scaling factor, q_i^+ and q_i^- are the i-th upper and lower joint limit, respectively. Test this new control mode (with velocity the corresponding cmd interface) and compare to the previous velocity control by reporting the plots of the commanded velocities and the joint position values. Switch between the two velocity controllers creating an additional parameter `ctrl:=velocity_ctrl/velocity_ctrl_null` *passed as argument to the node. Insert this in the previous launch file.*

EXECUTION

To implement the new `velocity_ctrl_null` controller within the `kdl_control` class, we first defined the function's interface. This interface was specified in the `kdl_control.h` header file, which is located in the `include` folder.

kdl_control.h

```
//The new function is being implemented, to which we pass
     the desired position vector (p_d) in order to then
    calculate the position error (e_p), the damping
    factor (lambda), and the proportional gain (K_p).
Eigen::VectorXd velocity_ctrl_null(const Eigen::Vector3d&
     p_des, double Kp, double lambda);
```

Subsequently, we implemented the function in the `kdl_control.cpp` file, which is located in the `src` folder. As required by the assignment, this function must implement the following control law:

$$\dot{q} = J^{\dagger} K_p e_p + (I - J^{\dagger} J)\dot{q}_0 \tag{3}$$

This formula can be viewed as a composition of a **primary task** (reaching the desired position p_{des}) and a **secondary task** in the null space to avoid joint limits, utilizing a repulsive gradient (`gradientJointLimits`). This is the logic that was followed to implement the body of the function.

kdl_control.cpp

```
Eigen::VectorXd KDLController::velocity_ctrl_null(const
    Eigen::Vector3d& p_des, double Kp, double lambda)
{
// Current state from the robot
//Pose end-effector (spatial)
KDL::Frame F_e    = robot_->getEEFrame();
//Current position
Eigen::Vector3d p_e = toEigen(F_e.p);
//Position error
Eigen::Vector3d e_p = p_des - p_e;

//Positional Jacobian (first 3 rows of the 6xN matrix)
Eigen::MatrixXd J  = robot_->getEEJacobian().data;  //6xN
J = J.topRows(3);                                   //3xN

//Pseudoinvers
Eigen::MatrixXd Jpinv = pseudoinverse(J);

//Primary Task: Differential IK on position
Eigen::VectorXd qdot_task = Jpinv * (Kp * e_p); // NX1

//Null-space: repulsive gradient to avoid joint limits
double cost_val = 0.0;
Eigen::MatrixXd jntLim = robot_->getJntLimits(); //(min,
```

```
        max)
    Eigen::VectorXd q      = robot_->getJntValues();    //Nx1
    Eigen::VectorXd grad   = gradientJointLimits(q, jntLim,
        cost_val); //Nx1

    Eigen::VectorXd qdot0  = -(1.0 / lambda) * grad;    //Nx1

    //Null-space projector
    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(J.cols(), J
        .cols());
    Eigen::MatrixXd Nproj = I - Jpinv * J;

    //Final command
    Eigen::VectorXd qdot = qdot_task + Nproj * qdot0;    //Nx1
    return qdot;
}
```

Afterward, we had to modify the `ros2_kdl_node.cpp` file. The modifications consist of the following:

- Declaration of the λ variable, which the code will read from the YAML file.
- Declaration of the `ctrl` variable, which is used to specify at runtime (via the CLI) whether the user desires to execute a **base control** or the **null space control**.
- Initialization of the new controller.
- An **if-else control structure** within the `cmd_publisher` to enable the selection.

ros2_kdl_node.cpp

```
    //Lambda-scaling factor of velocity_ctrl_null command
    declare_parameter("lambda", 5.0);
    get_parameter("lambda", lambda);

    //Ctrl command
    declare_parameter("ctrl", "velocity_ctrl");
    get_parameter("ctrl", ctrl);
    ---
    //We only print the $\lambda$ parameter to the screen if
        we are using null space control.
    if(cmd_interface_ == "velocity"){
        RCLCPP_INFO(get_logger(), "  ctrl: %s", ctrl.c_str())
            ;
            if (ctrl == "velocity_ctrl_null") {
                RCLCPP_INFO(get_logger()," lambda: %f",
                    lambda);
                }
        }
    ---
    controller = std::make_unique<KDLController>(*robot_);
    ---
    else if(cmd_interface_ == "velocity"){
        if (ctrl == "velocity_ctrl_null") {
            //New null-space control (point 1b)
            Eigen::VectorXd qdot = controller->
                velocity_ctrl_null(p_.pos, static_cast<double
                >(Kp), lambda);
            joint_velocities_cmd_.data = qdot;
        }
        else {
```

```
            //Old velocity control (used in point 1a)
            Vector6d cartvel;
            cartvel << p_.vel + Kp * error, o_error;
            joint_velocities_cmd_.data = pseudoinverse(robot_
                ->getEEJacobian().data) * cartvel;
        }
    }
                ---
    std::unique_ptr<KDLController> controller;
    std::string ctrl;
    double lambda;
```

In order to make the control function correctly, we had to modify the **YAML file** to command the λ parameter and, crucially, the **launch file** to read the choices made via the terminal regarding the desired type of control.

kdl_node_launch.py

```
    # We define the variables that we can pass from the
        terminal when launching with 'launch'
    # Create an argument named 'cmd_interface' that passes '
        position' as its default value
    cmd_interface = DeclareLaunchArgument(
        'cmd_interface',
        default_value='position',
        description='Command interface (position, velocity,
            effort)'
    )

    #Create an argument named 'ctrl' to specify the desired
        controller type
    ctrl = DeclareLaunchArgument(
        'ctrl',
        default_value='velocity_ctrl', # Il vecchio
            controllore
        description='Controller type (velocity_ctrl,
            velocity_ctrl_null)'
    )

    #Retrieving the argument values
    command = LaunchConfiguration('cmd_interface')
    ctrl_type = LaunchConfiguration('ctrl')

    #Node definition
    ros2_kdl_node = Node(
        package='ros2_kdl_package',
        executable='ros2_kdl_node',
        name='param_node',
        output='both',
        parameters=[params,
                {'cmd_interface': command},
                {'ctrl': ctrl_type}
        ]
    )

    return LaunchDescription([
        cmd_interface,
```

```
        ctrl,
        ros2_kdl_node
    ])
```

We can simulate the two controllers using the following commands:

```
TERMINAL 1:
$ cd ~/ros2_ws
$ source install/setup.bash
$ ros2 launch iiwa_bringup iiwa.launch.py command_interface
    :="velocity" robot_controller:="velocity_controller"

TERMINAL 2:
$ source install/setup.bash
$ ros2 launch ros2_kdl_package kdl_node.launch.py
    cmd_interface:=velocity ctrl:=velocity_ctrl

TERMINAL 2 (for null space control):
$ source install/setup.bash
$ ros2 launch ros2_kdl_package kdl_node.launch.py
    cmd_interface:=velocity ctrl:=velocity_ctrl_null
```

This link shows the simulation video: `https://youtu.be/gbvvS5XD8CU`.

As requested, we compared the two controllers by plotting the commanded velocities and joint positions over time. The following plots illustrate the differences between the standard velocity control and the null space velocity control.

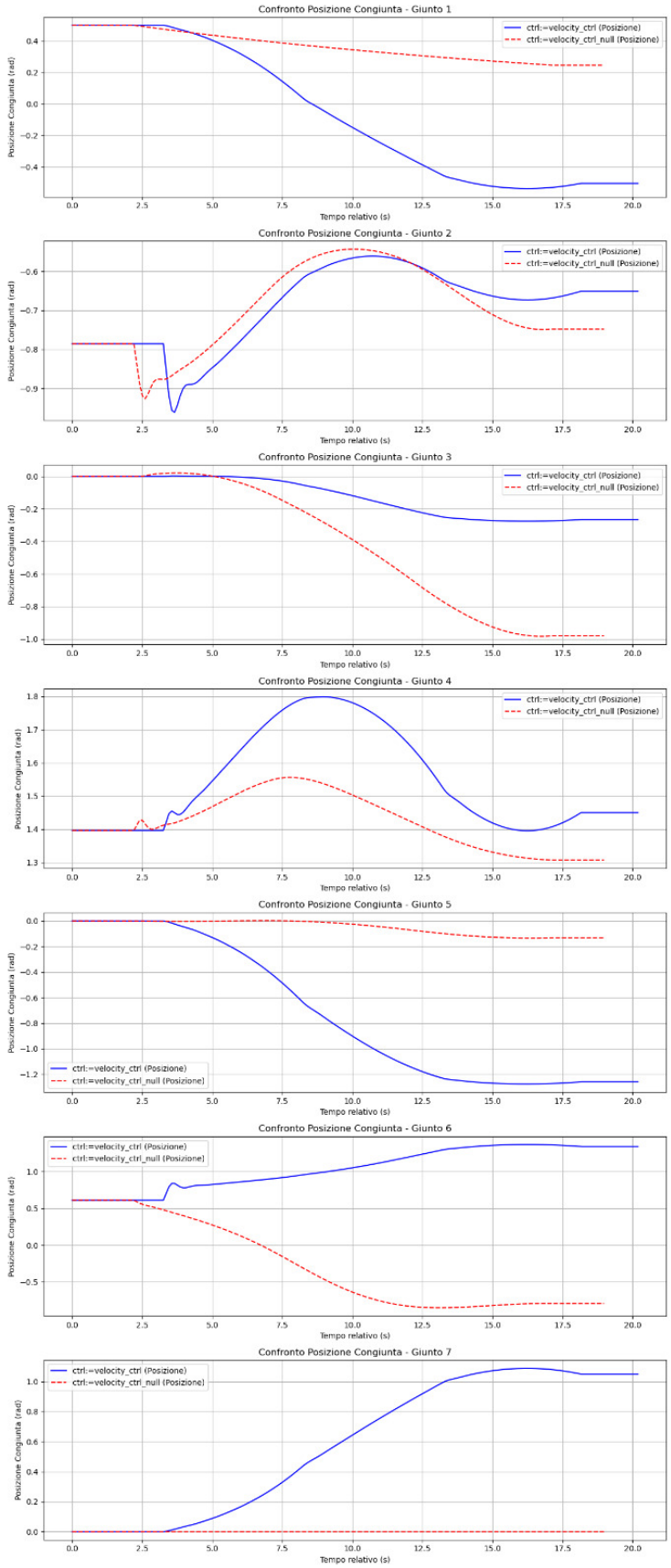

Figura 1: Comparison of Joint Positions

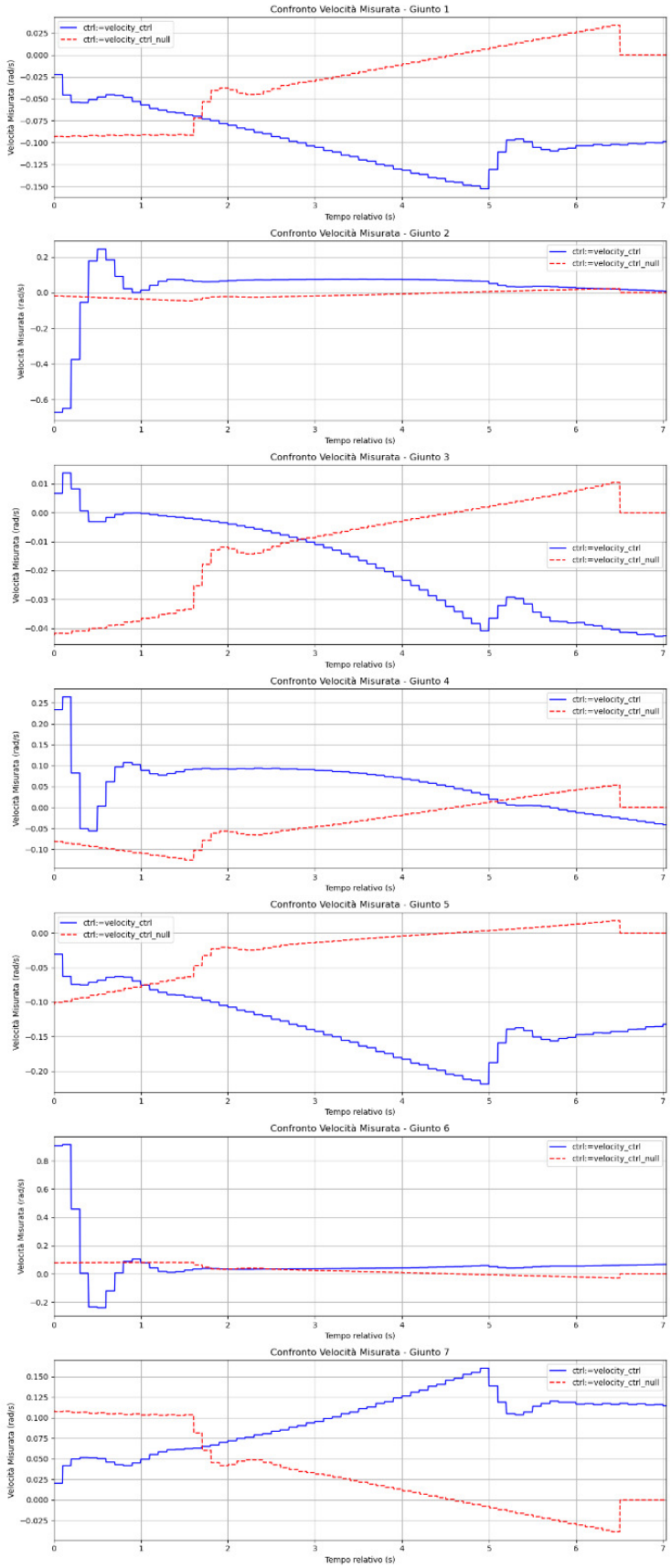

Figura 2: Comparison of Commanded Velocities

We can observe that the null space controller effectively avoids joint limits, as evidenced by the smoother joint position trajectories and reduced velocity spikes compared to the standard velocity controller.

(c) *Request: Following this tutorial, make our* ***ros2_kdl_node*** *an action server that executes the same linear trajectory and publishes the position error as feedback. Write an action client to test your code and add the command to run both of them to the* ***README.md*** *file in your repo.*

`EXECUTION`

The aim of point **1c** is to convert the controller to an **asynchronous ROS 2 Action** architecture. To do this we need to create three different elements:

- **Action definition file**: This file defines the structure of the action, including the goal, feedback, and result messages. We created a new folder named `action` inside the `ros2_kdl_package` directory and added a file named `KDLControl.action` with the following content:
- **Action server**: This is implemented within the `action_server_node.cpp` file. The server handles incoming action requests, executes the control logic, and provides feedback during execution.
- **Action client**: This is implemented in a new file named `action_client_node.cpp`. The client sends action requests to the server and processes the results.

First of all we created the action definition file, named `MoveArm.action`, with the following content:

MoveArm.action

```
# Goal: La posizione target (x, y, z)
geometry_msgs/Point target_position
---
# Result: Un booleano che indica il successo
bool success
---
# Feedback: L'errore di posizione corrente
float64 position_error_norm
```

Next, the action server was implemented. We choose separate the action server from the original `ros2_kdl_node.cpp` to keep the code modular and maintainable. The action server handles incoming goals, executes the control logic, and provides feedback during execution. We created a new file named `action_server_node.cpp` with the following content:

action_server_node.cpp

```
#include "rclcpp_action/rclcpp_action.hpp"
#include "ros2_kdl_package/action/move_arm.hpp"
---
using MoveArm = ros2_kdl_package::action::MoveArm;
using GoalHandleMoveArm = rclcpp_action::
    ServerGoalHandle<MoveArm>;
---
TrajectoryActionServer()
 : Node("action_server_node"),
 // ... (altre inizializzazioni)
{
 // ... (caricamento parametri, kdl, subscriber, etc
     .) ...
```

```
            // --- CREAZIONE ACTION SERVER ---
             using namespace std::placeholders;
             this->action_server_ = rclcpp_action::
                 create_server<MoveArm>(
                this,
                "move_arm",
                std::bind(&TrajectoryActionServer::
                    handle_goal, this, _1, _2),
                std::bind(&TrajectoryActionServer::
                    handle_cancel, this, _1),
                std::bind(&TrajectoryActionServer::
                    handle_accepted, this, _1));
             RCLCPP_INFO(get_logger(), "Action server '
                 move_arm' avviato e pronto.");
         }
        ---
        rclcpp_action::GoalResponse handle_goal(const
            rclcpp_action::GoalUUID & uuid, std::shared_ptr<
            const MoveArm::Goal> goal)
        {
             RCLCPP_INFO(this->get_logger(), "Ricevuto nuovo
                 goal per la traiettoria!");
             (void)uuid;
             // ... (logica per rifiutare se un altro goal
                 attivo) ...
             return rclcpp_action::GoalResponse::
                 ACCEPT_AND_EXECUTE;
        }

        rclcpp_action::CancelResponse handle_cancel(const std
            ::shared_ptr<GoalHandleMoveArm> goal_handle)
    {
             RCLCPP_INFO(this->get_logger(), "Richiesta di
                 cancellazione ricevuta");
             (void)goal_handle;
             return rclcpp_action::CancelResponse::ACCEPT;
     }

         void handle_accepted(const std::shared_ptr<
             GoalHandleMoveArm> goal_handle)
     {
         std::thread{std::bind(&TrajectoryActionServer::
             execute_trajectory, this, goal_handle)}.detach()
             ;
         }
        ---
        void execute_trajectory(const std::shared_ptr<
            GoalHandleMoveArm> goal_handle)
        {
             // ... (impostazione iniziale della traiettoria)
                 ...
             // --- LOOP DI ESECUZIONE (Controllo Cinematico)
                 ---

             while (rclcpp::ok() && t_ <= total_time)
             {
                 // 1. Gestione Annullamento Cliente
                 if (goal_handle->is_canceling()) {
                     // ... (stop robot) ...
```

```
                goal_handle->canceled(result);
                RCLCPP_INFO(get_logger(), "Goal
                    cancellato dal client.");
                return;
            }
            // ... (calcolo traiettoria, update robot,
                calcolo errore) ...
            // 5. Pubblicazione Feedback
            auto feedback_msg = std::make_shared<MoveArm::
                Feedback>();
            feedback_msg->position_error_norm = error.norm();
            goal_handle->publish_feedback(feedback_msg);
            // ... (calcolo legge di controllo e
                pubblicazione comandi) ...
            // ... (incremento tempo e sleep) ...
            }
            // --- FINE TRAIETTORIA ---
            // ... (stop robot) ...
            // Imposta il risultato dell'azione
            auto result = std::make_shared<MoveArm::Result>()
                ;
            result->success = (t_ > total_time);
            if (result->success) {
                goal_handle->succeed(result);
            } else {
                goal_handle->abort(result);
            }
        }
```

Finally, we implemented the action client in a new file named `action_client_node.cpp`:

action_client_node.cpp

```
    // Definizione del tipo di azione e dell'handle del
        Goal
    using MoveArm = ros2_kdl_package::action::MoveArm;
    using GoalHandleMoveArm = rclcpp_action::
        ClientGoalHandle<MoveArm>;

    class TrajectoryActionClient : public rclcpp::Node
    {
    public:
    explicit TrajectoryActionClient(const rclcpp::
        NodeOptions & options)
        : Node("trajectory_action_client", options)
{
            // 1. Creazione del client connesso all'
                azione "move_arm"
            this->client_ptr_ = rclcpp_action::
                create_client<MoveArm>(
            this,
            "move_arm");
}
---
void send_goal(double x, double y, double z)
{
    using namespace std::placeholders;
```

```
    // 1. Attende che l'Action Server sia disponibile (
        timeout di 10s)
    if (!this->client_ptr_->wait_for_action_server(std::
        chrono::seconds(10))) {
        RCLCPP_ERROR(this->get_logger(), "Action server non
            disponibile dopo 10s");
        return;
    }

    // 2. Popola il messaggio di Goal con la posizione target
         (x, y, z)
    auto goal_msg = MoveArm::Goal();
    goal_msg.target_position.x = x;
    goal_msg.target_position.y = y;
    goal_msg.target_position.z = z;

    RCLCPP_INFO(this->get_logger(), "Invio goal: [x: %f, y: %
        f, z: %f]", x, y, z);

    auto send_goal_options = rclcpp_action::Client<MoveArm>::
        SendGoalOptions();

    // 3. Collegamento dei callback per il ciclo di vita dell
        'azione
    send_goal_options.goal_response_callback =
        std::bind(&TrajectoryActionClient::
            goal_response_callback, this, _1);
    send_goal_options.feedback_callback =
        std::bind(&TrajectoryActionClient::feedback_callback,
             this, _1, _2);
    send_goal_options.result_callback =
        std::bind(&TrajectoryActionClient::result_callback,
            this, _1);

    // 4. Invio asincrono del Goal
    this->client_ptr_->async_send_goal(goal_msg,
        send_goal_options);
}

---

void feedback_callback(
    GoalHandleMoveArm::SharedPtr,
    const std::shared_ptr<const MoveArm::Feedback> feedback)
{
    // Stampa del feedback (norma dell'errore di posizione)
    RCLCPP_INFO(this->get_logger(), "Feedback ricevuto:
        Errore = %f", feedback->position_error_norm);
}

---

void result_callback(const GoalHandleMoveArm::WrappedResult &
     result)
{
    switch (result.code) {
        case rclcpp_action::ResultCode::SUCCEEDED:
            RCLCPP_INFO(this->get_logger(), "Goal Raggiunto!
                Successo: %s", result.result->success ? "true
```

```
                " : "false");
            break;
        case rclcpp_action::ResultCode::ABORTED:
            RCLCPP_ERROR(this->get_logger(), "Goal annullato
                (aborted)");
            break;
        case rclcpp_action::ResultCode::CANCELED:
            RCLCPP_ERROR(this->get_logger(), "Goal cancellato
                ");
            break;
        default:
            RCLCPP_ERROR(this->get_logger(), "Stato
                sconosciuto");
            break;
    }
    // Terminazione del nodo dopo il risultato
    rclcpp::shutdown();
}

---

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);

    // Controlla il numero corretto di argomenti (x, y, z)
    if (argc != 4) {
        RCLCPP_ERROR(rclcpp::get_logger("main"), "Uso: ros2
            run ros2_kdl_package action_client_node <x> <y> <
            z>");
        return 1;
    }

    auto action_client = std::make_shared<
        TrajectoryActionClient>(rclcpp::NodeOptions());

    // Converte gli argomenti in double e invia il goal
    try {
        double x = std::stod(argv[1]);
        double y = std::stod(argv[2]);
        double z = std::stod(argv[3]);
        action_client->send_goal(x, y, z);
    } catch (const std::invalid_argument& e) {
        RCLCPP_ERROR(rclcpp::get_logger("main"), "Argomenti
            non validi. x, y, z devono essere numeri.");
        return 1;
    }

    rclcpp::spin(action_client);
    return 0;
}
```

Then, we modified the `CMakeLists.txt` file to build the action server and client nodes and to generate the action messages:

CMakeLists.txt

```
# find dependencies per le Actions
find_package(rclcpp_action REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
find_package(action_msgs REQUIRED)
find_package(rosidl_typesupport_cpp REQUIRED)
---
# Genera i file sorgenti (C++, Python, ecc.) per l'
    interfaccia d'azione MoveArm
rosidl_generate_interfaces(${PROJECT_NAME}
    "action/MoveArm.action"
DEPENDENCIES geometry_msgs std_msgs sensor_msgs
)

ament_export_dependencies(rosidl_default_runtime)

# Include la cartella in cui vengono generati gli header
    C++
include_directories(${CMAKE_CURRENT_BINARY_DIR}/
    rosidl_generator_cpp)
---
add_executable(action_client_node src/action_client_node.
    cpp)
target_include_directories(action_client_node PUBLIC
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
$<INSTALL_INTERFACE:include>)
ament_target_dependencies(action_client_node
rclcpp
rclcpp_action
geometry_msgs
action_msgs
)

# Collega il typesupport generato per MoveArm.action
rosidl_get_typesupport_target(
    client_node_target
    ${PROJECT_NAME}
    "rosidl_typesupport_cpp")

target_link_libraries(action_client_node
    ${client_node_target})

    ---

add_executable(action_server_node src/action_server_node.
    cpp src/kdl_robot.cpp src/kdl_planner.cpp src/
    kdl_control.cpp)
# ...
ament_target_dependencies(action_server_node
rclcpp
rclcpp_action
geometry_msgs
action_msgs
orocos_kdl # Necessario per la logica KDL
kdl_parser # Necessario per la logica KDL
)
```

```
    # Collegamento del typesupport
    rosidl_get_typesupport_target(
    server_node_target
    ${PROJECT_NAME}
    "rosidl_typesupport_cpp")

    target_link_libraries(action_server_node
        ${server_node_target})

    ---

    install(TARGETS ros2_kdl_node action_client_node
        action_server_node
        DESTINATION lib/${PROJECT_NAME})
```

Also the `package.xml` file was modified to include the necessary dependencies for actions:

package.xml

```
    <depend>rclcpp</depend>
    <depend>orocos_kdl</depend>
    <depend>kdl_parser</depend>
    ---
    <depend>rclcpp_action</depend>
    <depend>geometry_msgs</depend>
    <depend>action_msgs</depend>
    <depend>std_msgs</depend>
    <depend>sensor_msgs</depend>
```

Finally, we created a launch file named `kdl_action_launch.py` to start the action server:

kdl_action_launch.py

```
    import os
from ament_index_python.packages import
    get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    pkg_share = get_package_share_directory('ros2_kdl_package
        ')

    param_file_path = os.path.join(pkg_share, 'config', '
        param.yaml')

    action_server_node = Node(
        package='ros2_kdl_package',
        executable='action_server_node',
        name='action_server_node',
        output='screen',
        # PASSAGGIO CHIAVE: Carica i parametri dal file YAML
        parameters=[param_file_path]
    )

    return LaunchDescription([
```

```
        action_server_node
    ])
```

After completing these modifications, we built the workspace again to compile the new action server and client nodes and to generate the action messages. Finally, we ran both the action server and client using the following commands:

```
TERMINAL 1:
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
$ ros2 launch iiwa_bringup iiwa.launch.py command_interface
    :="velocity" robot_controller:="velocity_controller"

TERMINAL 2:
$ source install/setup.bash
$ ros2 launch ros2_kdl_package kdl_action_launch.py

TERMINAL 3:
$ source install/setup.bash
$ ros2 run ros2_kdl_package action_client_node 0.4 0.3 0.5
```

The action server will execute the trajectory to the specified target position (0.4, 0.3, 0.5) while providing feedback on the position error norm during execution.
The simulation result can be viewed in the following video link: `https://youtu.be/O8keFosJe9E`.

2. **Vision_based control**

(a) *Request: Construct a Gazebo world inserting an aruco tag and detect it via the* ***aruco_ros*** *package (link here). Go into the* ***iiwa_description*** *package of the* ***ros2_iiwa*** *stack. There, create a folder* ***gazebo/models*** *containing the aruco marker model for Gazebo. Create a new model named* ***aruco_tag*** *and import it into a new Gazebo world as a static object in a position that is visible by the camera. Save the new world into the* ***/gazebo/worlds/*** *folder.*

`EXECUTION`

To address task 2a, a Gazebo simulation environment was constructed, featuring a static ArUco tag and a camera-equipped robot.

First, an ArUco marker image (ID 18) was generated from the `ORIGINAL_ARUCO` dictionary using the online generator available at `https://chev.me/arucogen/`. The marker size was set to 100x100 pixels and saved as a PNG.

To create the Gazebo model, a new folder structure `gazebo/models/aruco_tag` was created within the `iiwa_description` package. This directory contains the necessary files to define the model. A `model.config` file provides metadata for Gazebo, such as the model's name, version, and author information. A `model.sdf` file defines the model's properties; it is set as `<static>true</static>` and its geometry is a thin box (0.1m x 0.1m x 0.001m). The PNG image is applied as a visual texture using the `<albedo_map>` tag.

A new file named `aruco.world` was created inside the `iiwa_description/gazebo/worlds/` folder. This world file includes standard elements (e.g., sun, ground plane) and, most importantly, includes the `model://aruco_tag` defined above. The marker is instantiated as a static object at a specific pose, ensuring it is visible from the robot's camera perspective.

model.config

```
<?xml version="1.0"?>
<model>
<name>aruco_tag</name>
<version>1.0</version>
<sdf version="1.9">model.sdf</sdf>
<author>
    <name>Federica_Pirozzi</name>
    <email>federi.pirozzi@studenti.unina.it</email>
</author>
<description>Aruco tag model</description>
</model>
```

model.sdf

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.9">
<model name="aruco_tag">
    <static>true</static>
    <pose>0.16 -0.48 0.41 1.36 0.00 -1.17</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://aruco_tag/aruco-18.
                        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>
```

aruco.world

```
<?xml version="1.0" ?>
<sdf version="1.4">
<!-- We use a custom world for the robot so that the
    camera angle is launched correctly -->

<world name="default">
<!-- Included light -->
<include>
  <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/models
      /Sun</uri>
</include>

<!-- Included model -->
<include>
  <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/models
      /Ground Plane</uri>
</include>

<include>
  <uri>
    model://aruco_tag
  </uri>
  <name>aruco_tag</name>
   <pose>0.16 -0.48 0.41 1.36 0.00 -1.17</pose>
</include>

<gravity>0 0 0</gravity>

<!-- Focus camera on tall pendulum -->
<gui fullscreen='0'>
  <camera name='user_camera'>
    <pose>4.927360 -4.376610 3.740080 0.000000 0.275643
```

```
            2.356190</pose>
          <view_controller>orbit</view_controller>
        </camera>
      </gui>
      </world>
      </sdf>
```

To enable the robot to perceive the tag, a camera sensor was added. A new Xacro macro file, `camera.xacro`, was created. This file defines a new `camera_link` and attaches a Gazebo camera sensor plugin to it, specifying its resolution (320x240), horizontal FOV, and the Gazebo topic (`/camera`) on which it publishes. This macro was then included in the main `iiwa.urdf.xacro` file, rigidly attaching the `camera_link` to the robot's end-effector, `tool0`.

camera.xacro

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

    <xacro:macro name="end_effector_camera" params="prefix">

      <joint name="camera_joint" type="fixed">
        <parent link="${prefix}tool0"/>
        <child link="${prefix}camera_link"/>
        <axis xyz="0 1 0"/>
        <origin xyz="0 0 0" rpy="0 -1.57 3.14"/>
      </joint>

      <link name="${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>

       <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:macro>

</robot>
```

iiwa.urdf.xacro

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

The last step was to create the launch file `gaz.launch.py` within the `ros2_kdl_package` to load the new Gazebo world and start all necessary nodes.

In this launch file, we first declared an argument named `gz_args` that specifies the Gazebo world to be loaded, pointing to our custom `aruco.world`. We then included the standard Gazebo launch file from the `ros_gz_sim` package, passing this `gz_args` argument.

Next, we defined the `robot_state_publisher` node to publish the robot's state from the URDF, enabling simulation time synchronization. We also included nodes to spawn the `iiwa` robot model into the Gazebo world and event handlers to sequentially load the `joint_state_broadcaster` and `velocity_controller`.

Additionally, critical bridges were set up. A `clock_bridge` synchronizes simulation time with ROS 2 time. A `camera_bridge` was configured to relay camera data; this bridge translates the GazEOF image topic (`/camera`) into a ROS 2 `sensor_msgs/msg/Image` and remaps it to `/stereo/left/image_rect_color`, which is the default input topic expected by the `aruco_ros` package.

Finally, we added an `rqt_image_view` node to visualize the camera feed, specifically subscribing to the `/aruco_single/result` topic, where the ArUco detection results are published.

iiwa.launch.py

```
    def generate_launch_description():

    #Load Robot Description (XACRO)
    # Build the complete path to the xacro file
    xacro_file_name = "iiwa.config.xacro"
    xacro = os.path.join(
        get_package_share_directory("iiwa_description"), "
            config", xacro_file_name)

    #Define parameters for nodes.
```

```
    params={"robot_description": Command(["xacro ", xacro])}

    # Declare a launch argument named 'gz_args' to specify
        which world to load
    declared_arguments = [
        DeclareLaunchArgument("gz_args",
            default_value=["-r ", PathJoinSubstitution([
                get_package_share_directory("iiwa_description
                "), "gazebo", "worlds", "aruco.world"])],
            description="Arguments for gz_sim",
        )
    ]

    # This includes the standard launch file from the
        ros_gz_sim package
    # It starts the Gazebo simulator using the parameters
        defined above
    gz_ign = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                [PathJoinSubstitution([FindPackageShare("
                    ros_gz_sim"), "launch",
                                           'gz_sim.launch.py'])]),
            launch_arguments={"gz_args": LaunchConfiguration(
                "gz_args"),
                              "publish_clock": "true",
                              }.items()
    )

    # Define and configure the robot_state_publisher node
    robot_state_publisher_node = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        output="screen",
        parameters=[params,
                    {"use_sim_time": True},  #to synchronize
                        gazebo time and ros2 time
            ],
    )

    # This node calls the `create` executable from ros_gz_sim
    # (read from the '/robot_description' topic) into the
        running Gazebo simulation.
    spawn_entity_node = Node(
        package="ros_gz_sim",
        executable="create",
        output="screen",
        arguments=["-topic", "/robot_description", "-name", "
            iiwa"],
    )

    # Define the node that will spawn the '
        joint_state_broadcaster'
    # This controller publishes joint states for all
        interfaces.
    joint_state_broadcaster_node = Node(
        package="controller_manager",
        executable="spawner",
        arguments=[
            "joint_state_broadcaster",
```

```
            "--controller-manager",
            "/controller_manager",
        ],
    )

    # Define the node that will spawn the '
        velocity_controller'
    robot_controller_spawner = Node(
        package="controller_manager",
        executable="spawner",
        arguments=[
            "velocity_controller",
            "--controller-manager",
            "/controller_manager",
        ],
    )

    # Create an event handler that waits for '
        spawn_entity_node' to exit
    spawn_jsb_handler = RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=spawn_entity_node,
            on_exit=[joint_state_broadcaster_node],
        )
    )

    # Create another handler that waits for '
        joint_state_broadcaster_node' to exit
    spawn_controller_handler = RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=joint_state_broadcaster_node,
            on_exit=[robot_controller_spawner],
        )
    )

    # Clock bridge to keep simulation time and ROS2 time
        synchronized
    clock_bridge = Node(
        package="ros_ign_bridge",
        executable="parameter_bridge",
        arguments=["/clock@rosgraph_msgs/msg/Clock[ignition.
            msgs.Clock"],
        output="screen",
    )

    # Camera bridge
    bridge_camera = Node(
        package="ros_ign_bridge",
        executable="parameter_bridge",
        arguments=[
            # Bridge Gazebo's '/camera' topic to a ROS 2 '
                sensor_msgs/msg/Image'
            "/camera@sensor_msgs/msg/Image@gz.msgs.Image",
            # Bridge the '/camera_info' topic as well
            "/camera_info@sensor_msgs/msg/CameraInfo@gz.msgs.
                CameraInfo",
            "--ros-args",
            # Remap the ROS 2 topic '/camera' to '/stereo/
                left/image_rect_color'
```

```
            "-r",
            "/camera:=/stereo/left/image_rect_color",
            # Remap the ROS 2 topic '/camera_info' to '/
                stereo/left/camera_info'
            "-r",
            "/camera_info:=/stereo/left/camera_info",
            # These remappings are likely to match the input
                topics expected by 'aruco_ros'
        ],
        output="screen",
    )

    # Node to visualize the ArUco detection results
    rqt_image = Node(
        package="rqt_image_view",
        executable="rqt_image_view",
        name="rqt_image_view",
        output="screen",
        arguments=["/aruco_single/result"],
    )

    # Collect all declared arguments, nodes, and event
        handlers
    return LaunchDescription(
        declared_arguments
        + [
        gz_ign,
        robot_state_publisher_node,
        spawn_entity_node,
        clock_bridge,
        bridge_camera,
        rqt_image,
        bridge_service,
        spawn_jsb_handler,
        spawn_controller_handler,
        ]
    )
```

Finally, we launched Gazebo with the following command:

```
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
$ ros2 launch ros2_kdl_package gaz.launch.py
```

In another terminal, we launched the `aruco_ros` package to detect the ArUco marker:

```
$ source install/setup.bash
$ ros2 launch aruco_ros single.launch.py marker_size:=0.1
    marker_id:=18
```

After launching both Gazebo and the ArUco detection node, by setting the `rqt_image_view` to the `/aruco_single/result` topic, we were able to see the camera feed with the detected ArUco marker highlighted. This confirmed that the world, model, camera, and bridges were all configured correctly.

The simulation result can be viewed in the following video link: `https://youtu.be/Rl0hU6uW39A`.

(b) *Request: Spawn the robot with the velocity command interface into the world containing the aruco tag. In the* ***`kdl_control`*** *class of the* ***`ros2_kdl_package`*** *create a vision-based controller called* ***`vision_ctrl`*** *for the simulated iiwa robot that is activated setting the* ***`ctrl`*** *ROS parameter to* ***`vision`****. Create a subscriber to the aruco marker pose published by the* ***`aruco_ros`*** *package*[1]*. The controller should be able to perform a look-at-point task using the following control law*

$$\dot{q} = K(L(s)J_c)^{\dagger} s_d + N\dot{q}_0, \tag{4}$$

where K is a diagonal gain matrix, $s_d = [0,\ 0,\ 1]$ is a desired value for

$$s = \frac{c_o^P}{\|c_o^P\|} \in \mathbb{S}^2, \tag{5}$$

that is a unit-norm axis connecting the origin of the camera frame and the position of the object c_o^P. The matrix J_c is the camera Jacobian (different from the end-effector one), while $L(s)$ maps linear/angular velocities of the camera to changes in s as follows

$$L(s) = \begin{bmatrix} -\frac{1}{\|c_o^P\|}(I - ss^{\mathrm{T}}) & S(s) \end{bmatrix} R \in \mathbb{R}^{3\times 6} \quad with \quad R = \begin{bmatrix} R_c^{\mathrm{T}} & 0 \\ 0 & R_c^{\mathrm{T}} \end{bmatrix}, \tag{6}$$

where $S(\cdot)$ is the skew-symmetric operator, R_c the current camera rotation matrix. Finally, $N = (I - (L(s)J_c)^{\dagger}L(s)J_c)$ is the matrix spanning the null space of the $L(s)J_c$ matrix. Show the tracking capability of the controller by manually moving the aruco marker around in Gazebo. The user interface and robot plot the joint velocities and the end-effector commands sent to the robot. To spawn the robot with the velocity command interface into the Gazebo world, we modified the default arguments in the launch file that we used in point 2.a. This ensures that the `VelocityController` from `ros2_control` is loaded and active for the `iiwa` robot.

To implement the vision-based controller, we defined a new control law in the `ros2_kdl_package` within the `kdl_control` class, as requested. This function, `vision_ctrl`, implements the "look-at-point" task using the control law specified in Equation 3 of the homework. The function receives the pose of the marker in the camera frame (T_cam_aruco), the current pose of the camera in the base frame (T_base_cam), and the camera's geometric Jacobian (J_base_cam).It computes the unit-norm feature vector s, the interaction matrix Lmat, and the pseudoinverse. It then returns the joint velocities qdot required to drive s to the desired feature sd = [0, 0, 1], using a zero vector for the null-space task qdotzero.

kdl_control.cpp

```
    Eigen::VectorXd KDLController::vision_ctrl(KDL::Frame
        pose_in_camera_frame, KDL::Frame camera_frame, KDL::
        Jacobian camera_jacobian, Eigen::VectorXd q0_dot)
{
    // 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,
```

[1] https://github.com/pal-robotics/aruco_ros

```
        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; // 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);

    Eigen::Vector3d s_d(0, 0, 1); // Desired unit vector
        pointing forward
    double k = -2;                // Gain for the task
    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.norm() << std::
        endl;

    // Return computed joint velocities
    return joint_velocities;

}
```

Then we created a new node called `ros2_vision_control_node.cpp` in `ros2_kdl_package` to implement the look-at-point task using the vision-based controller. This node activates when the `ctrl` ROS parameter is set to `vision`. The node subscribes to `/joint_states` for the robot's state and to `/aruco_single/pose` to receive the marker pose from the `aruco_ros` package.
A 10Hz timer executes the control loop. In each cycle, the node updates the robot's KDL model, calculates the current camera pose T_base_cam and Jacobian J_base_cam (by applying the fixed T_ee_cam transform to the flange), and then calls the `controller_->vision_ctrl(...)` function. A timeout logic is included to stop the robot if the marker is not visible. The resulting joint velocities qdot are published to `/velocity_controller/commands`.

ros2_vision_control_node.cpp

```
class VisionControlNode : public rclcpp::Node
{
    public:
        VisionControlNode()
        : Node("ros2_vision_control_node"),
        node_handle_(std::shared_ptr<VisionControlNode>(this)
            )
        {
            // Declare and Get ROS Parameters
```

```
declare_parameter("cmd_interface", "velocity");
get_parameter("cmd_interface", cmd_interface_);
RCLCPP_INFO(get_logger(),"Current cmd interface
    is: '%s'", cmd_interface_.c_str());

if (cmd_interface_ != "velocity")
{
    RCLCPP_ERROR(get_logger(),"This control mode
        only supports 'velocity'!");
    rclcpp::shutdown();
    return;
}

declare_parameter("ctrl_mode", "vision");
get_parameter("ctrl_mode", ctrl_mode_);
RCLCPP_INFO(get_logger(),"Current control mode is
    : '%s'", ctrl_mode_.c_str());

if (ctrl_mode_ == "vision")
{
    RCLCPP_INFO(get_logger(),"Controllo visione
        attivo");
}

//Initialize State Flags
aruco_pose_available_ = false;
joint_state_available_ = false;
//Initialize the timestamp for the pose timeout
    logic
last_pose_time_ = this->get_clock()->now();

//Setup KDL Robot from robot_description
//Create a parameter client to get the robot
    description from robot_state_publisher
auto parameters_client = std::make_shared<rclcpp
    ::SyncParametersClient>(node_handle_, "
    robot_state_publisher");
while (!parameters_client->wait_for_service(1s))
    {
    if (!rclcpp::ok()) {
        RCLCPP_ERROR(this->get_logger(), "
            Interrupted while waiting for the
            service. Exiting.");
        rclcpp::shutdown();
    }
    RCLCPP_INFO(this->get_logger(), "service not
        available, waiting again...");
}
auto parameter = parameters_client->
    get_parameters({"robot_description"});

KDL::Tree robot_tree;
if (!kdl_parser::treeFromString(parameter[0].
    value_to_string(), robot_tree)){
    std::cout << "Failed to retrieve
        robot_description param!";
    rclcpp::shutdown();
    return;
}
```

```
robot_ = std::make_shared<KDLRobot>(robot_tree);

// Get joint count and set joint limits
nj_ = robot_->getNrJnts();
KDL::JntArray q_min(nj_), q_max(nj_);
q_min.data <<
    -2.96,-2.09,-2.96,-2.09,-2.96,-2.09,-2.96;
q_max.data <<
    2.96,2.09,2.96,2.09,2.96,2.09,2.96;
robot_->setJntLimits(q_min,q_max);

// Resize KDL arrays
joint_positions_.resize(nj_);
joint_velocities_.resize(nj_);
joint_velocities_cmd_.resize(nj_);
joint_velocities_cmd_.data.setZero();

//Setup Subscribers and Wait for Initial State
jointSubscriber_ = this->create_subscription<
    sensor_msgs::msg::JointState>(
    "/joint_states", 10, std::bind(&
        VisionControlNode::joint_state_subscriber
        , this, std::placeholders::_1));

// Wait until the first /joint_states message is
    received
while(!joint_state_available_){
    RCLCPP_INFO(this->get_logger(), "No data
        received yet! ...");
    rclcpp::spin_some(node_handle_);
}

//Initial Robot and Controller Setup
robot_->update(toStdVector(joint_positions_.data)
    ,toStdVector(joint_velocities_.data));

// Initialize the controller class
controller_ = std::make_shared<KDLController>(*
    robot_);

//Setup Publisher and Control Loop Timer
cmdPublisher_ = this->create_publisher<FloatArray
    >("/velocity_controller/commands", 10);

// Create a 10Hz timer (100ms) that calls the
    cmd_publisher function
timer_ = this->create_wall_timer(std::chrono::
    milliseconds(100),
                                 std::bind(&
                                     VisionControlNode
                                     ::cmd_publisher,
                                     this));

// Send an initial zero-velocity command to
    ensure the robot is stationary
for (long int i = 0; i < joint_velocities_.data.
    size(); ++i) {
    desired_commands_[i] = 0.0;
```

```
        }

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

        //Setup Vision Subscriber
        image_sub_ = this->create_subscription<
            geometry_msgs::msg::PoseStamped>(
            "/aruco_single/pose", 10, std::bind(&
                VisionControlNode::imageCallback, this,
                std::placeholders::_1));

        RCLCPP_INFO(this->get_logger(), "Starting vision
            control execution (LOGICA ESEMPIO)...");
    }

private:

    void cmd_publisher(){

        //// This prevents "ghost movement" if the marker
             hasn't been seen for a while.
        rclcpp::Time now = this->get_clock()->now();
        if ((now - last_pose_time_).seconds() > 0.5)
        {
            // If last pose is older than 0.5s, mark it
                as unavailable
            aruco_pose_available_ = false;
        }

        // Update the model with the latest joint
            positions/velocities
        robot_->update(toStdVector(joint_positions_.data)
            ,toStdVector(joint_velocities_.data));

        // Get the pose and Jacobian of the DEFAULT End-
            Effector
        KDL::Frame cartpos = robot_->getEEFrame();
        KDL::Jacobian J_cam = robot_->getEEJacobian();

        // Main Control Logic
        if (ctrl_mode_ == "vision")
        {
            // Check if the marker pose is still valid
            if (!aruco_pose_available_) {
                RCLCPP_WARN_ONCE(get_logger(), "Aruco
                    marker pose not available or timed
                    out. Stopping.");
                joint_velocities_cmd_.data.setZero();
            } else {

                //This logic calculates T_base_cam and
                    J_base_cam in every cycle.
                // Define the static transformation from
                    the EE to the Camera
                KDL::Frame T_ee_cam = KDL::Frame(
                    KDL::Rotation::RotY(-1.57) * KDL::
                        Rotation::RotZ(3.14)
```

```
                    );

                    // Current camera pose in the base frame:
                        T_base_cam = T_base_ee * T_ee_cam
                    KDL::Frame cartpos_camera = cartpos *
                        T_ee_cam;

                    // Transform the EE Jacobian (J_cam) to
                        the Camera Jacobian (J_cam_camera)
                    // J_cam_camera = R_base_cam * J_cam_base
                    // This gives us the Jacobian of the
                        camera frame, expressed in the base
                        frame.
                    KDL::Jacobian J_cam_camera(nj_);
                    KDL::changeBase(J_cam, cartpos_camera.M,
                        J_cam_camera);

                    Eigen::VectorXd q0_dot = Eigen::VectorXd
                        ::Zero(nj_);

                    // Calculate joint velocities using the
                        vision controller
                    joint_velocities_cmd_.data = controller_
                        ->vision_ctrl(
                        pose_in_camera_frame_, // T_cam_aruco
                        cartpos_camera,        // T_base_cam
                        J_cam_camera,          // J_base_cam
                        q0_dot
                    );
                }
            }
            else
            {
                // If control mode is not 'vision', log a
                    warning and send zero velocities
                RCLCPP_WARN_ONCE(get_logger(), "Control mode
                    '%s' not recognized. Sending zero
                    velocities.", ctrl_mode_.c_str());
                joint_velocities_cmd_.data.setZero();
            }


            // Copy the calculated KDL joint velocities to
                the output vector
            for (long int i = 0; i < joint_velocities_.data.
                size(); ++i) {
                desired_commands_[i] = joint_velocities_cmd_(
                    i);
            }

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

        // allback for the Aruco marker pose topic (/
            aruco_single/pose)
        void imageCallback(const geometry_msgs::msg::
```

```
    PoseStamped& msg) {
    aruco_pose_available_ = true;
    // Update the timestamp every time a new pose is
        received
    last_pose_time_ = this->get_clock()->now();

    // Extract position and orientation from the
        message
    const auto position = msg.pose.position;
    const auto orientation = msg.pose.orientation;

    // Convert from geometry_msgs to KDL types
    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
    );

    // Store the pose (T_cam_aruco) in the class
        member
    pose_in_camera_frame_.M = kdl_rotation;
    pose_in_camera_frame_.p = kdl_position;
}

//Callback for the robot's joint state topic (/
    joint_states)
void joint_state_subscriber(const sensor_msgs::msg::
    JointState& sensor_msg){
    joint_state_available_ = true;
    for (unsigned int i  = 0; i < sensor_msg.position
        .size(); i++){
        joint_positions_.data[i] = sensor_msg.
            position[i];
        joint_velocities_.data[i] = sensor_msg.
            velocity[i];
    }
}

// ROS 2 Members
rclcpp::Subscription<sensor_msgs::msg::JointState>::
    SharedPtr jointSubscriber_;
rclcpp::Subscription<geometry_msgs::msg::PoseStamped
    >::SharedPtr image_sub_;
rclcpp::Publisher<FloatArray>::SharedPtr
    cmdPublisher_;
rclcpp::TimerBase::SharedPtr timer_;
rclcpp::Node::SharedPtr node_handle_;

// KDL and State Members
std::shared_ptr<KDLRobot> robot_;
std::shared_ptr<KDLController> controller_;

KDL::JntArray joint_positions_;
KDL::JntArray joint_velocities_;
KDL::JntArray joint_velocities_cmd_;

KDL::Frame pose_in_camera_frame_; // T_cam_aruco
```

```
        // State Variables and Parameters
        std::vector<double> desired_commands_ = {0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0};
        unsigned int nj_;
        bool joint_state_available_;
        bool aruco_pose_available_;
        std::string cmd_interface_;
        std::string ctrl_mode_;

        // FIX ESSENZIALE: Membro per il timestamp
        rclcpp::Time last_pose_time_;
};
```

To test the controller, we launched Gazebo with the ArUco tag, the `aruco_ros` node, and the vision control node with the following commands:

```
TERMINAL 1:
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
$ ros2 launch ros2_kdl_package gaz.launch.py

TERMINAL 2:
$ source install/setup.bash
$ ros2 launch aruco_ros single.launch.py marker_size:=0.1
    marker_id:=18

TERMINAL 3:
$ source install/setup.bash
$ ros2 run ros2_kdl_package ros2_vision_control_node
    cmd_interface:=velocity ctrl:=vision
```

We can see the tracking capability of the controller, responding to the marker being moved manually in the Gazebo interface, in the following video link: `https://youtu.be/rzPom3kmrsM`.
We also plot the joint velocities and the end-effector commands sent to the robot using `rqt_plot` by subscribing to the topics `/velocity_controller/commands`:

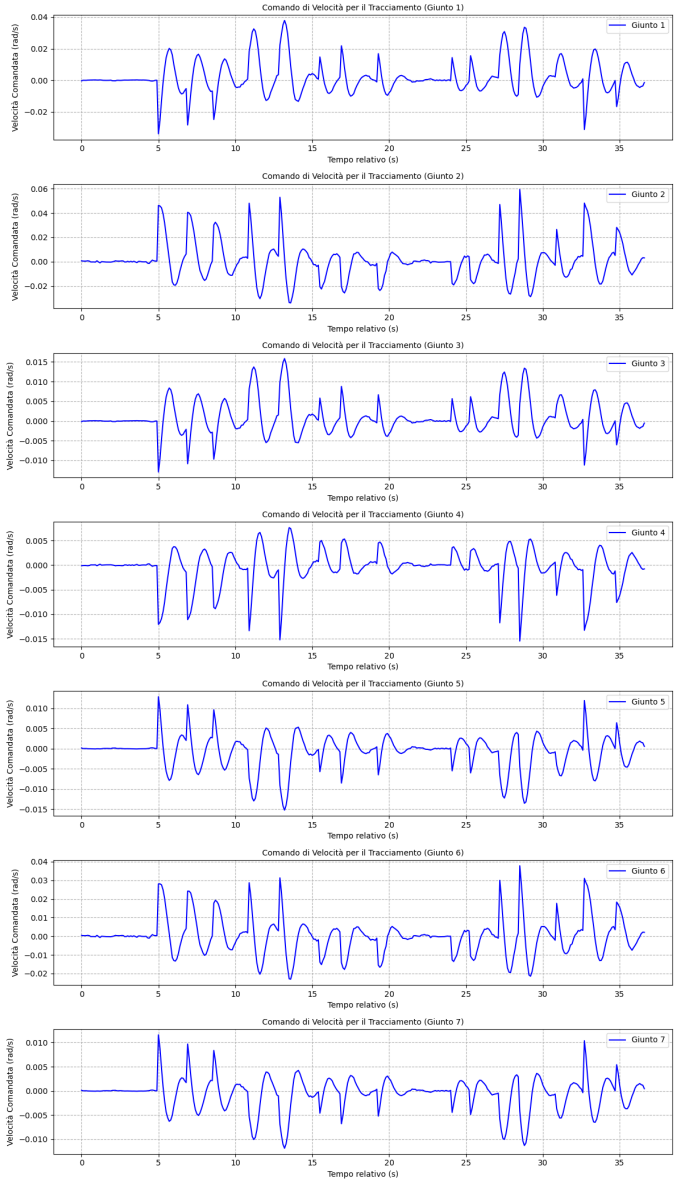

Figura 3: Joint velocities command plot.

(c) *Request: Create a ROS 2 service to update the aruco marker position in Gazebo. To do so, starting from the* ***/set_pose ign*** *service, create a* ***parameter_bridge*** *in the previously created launch file. Test the bridged service via a ROS 2 service call.*

To do this point, we added a parameter bridge for the `/set_pose` service in the `gaz.launch.py` file created in point 2.a. This bridge allows us to call the Gazebo service from ROS 2.

gaz.launch.py

```
bridge_service = Node(
    package="ros_gz_bridge",
    executable="parameter_bridge",
    name="set_pose_bridge",
    arguments=["/world/default/set_pose@ros_gz_interfaces
        /srv/SetEntityPose"],
```

```
            output="screen",
        )
```

To test this service, after we launched Gazebo and aruco nodes, we have to launch the following command in a terminal:

```
$ ros2 service call /world/default/set_pose ros_gz_interfaces/
    srv/SetEntityPose "{entity: {name: 'aruco_tag', type: 1},
    pose: {position: {x: 1.3, y: 0.6, z: 0.2}, orientation: {x:
    0.0, y: 0.0, z: 0.0, w: 1.0}}}"
```

We can see that the aruco tag in Gazebo moves to the new position specified in the service call.

The results can be viewed in the following video link: `https://youtu.be/kmpNUJriw_E`.