Robotics Lab: Homework 2

Control your robot

Alessandro Prisco: https://github.com/Alexsoft55-unina/RL2025_Homework2
Eros Cribello: https://github.com/eroscribello/Homework2_RL25.git
Andrea Russo: https://github.com/Andrearusso01/Homework2_group.git
Aldo Rosicarelli: https://github.com/aldetor00/Homework2_group.git

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

Control your robot

1. Kinematic control

   (a) We modified the `ros2_kdl_node`, located in a file called `launching.launch.py`, such that the following variables became ROS2 parameters: `traj_duration, acc_duration, total_time, trajectory_len`, Kp, and the three components of the trajectory `end_position`.

```
ros2_kdl_node = Node(
      package='ros2_kdl_package',
      executable='ros2_kdl_node',
      name='ros2_kdl_node',
      parameters=[
          PathJoinSubstitution([
              FindPackageShare('ros2_kdl_package'),
              'config',
              'kdl_params.yaml'
          ]),
          {'cmd_interface': cmd_interface_val},
          {'ctrl': ctrl_val}
      ]
  )
```

   This is our `kdl_params.yaml` (located in the config folder) and it contains the parameters' definition:

```
ros2_kdl_node:
  ros__parameters:
    traj_duration: 8.0
    acc_duration: 5.0
    total_time: 8.0
    trajectory_len: 200
    Kp: 1
    end_position: [0.5, 0.3, 0.5]
```

   Then we added the launch command to the README file in our repo:

```
ros2 launch ros2_kdl_package launching.launch.py cmd_interface:=velocity ctrl:=
    velocity_ctrl
```

   (b) We created 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

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

where λ is a scaling factor, and q_i^+ and q_i^- are the i–th upper and lower joint limit, respectively.

```
KDL::JntArray KDLController::velocity_ctrl_null(Eigen::Matrix<double,6,1>
    error_position,int Kp)

{
    unsigned int nj = robot_->getNrJnts();

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

    Eigen::MatrixXd I;
    I = Eigen::MatrixXd::Identity(nj,nj);

    Eigen::MatrixXd JntLimits_ (nj,2);
    JntLimits_ = robot_->getJntLimits();

    Eigen::VectorXd q_min(nj);
    Eigen::VectorXd q_max(nj);
    q_min = JntLimits_.col(0);
    q_max = JntLimits_.col(1);

    Eigen::VectorXd q(nj);
    q  = robot_->getJntValues();

    double lambda = 50;

    Eigen::VectorXd q0_dot(nj);
    for (unsigned int i = 0; i<nj; i++) {

        double L =(q_max(i) - q_min(i))*(q_max(i) - q_min(i));

        double G = (2*q(i) - q_max(i) - q_min(i));

        double D = (q_max(i)- q(i))*(q(i)- q_min(i));

        q0_dot(i) = 1/lambda*L*G/(D*D);

    }

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

    Eigen::VectorXd qd_vec(nj);
    qd_vec = J_pinv * error_position * Kp + (I-J_pinv*J)*q0_dot;
```

```
    KDL::JntArray qd(nj);

    qd.data = qd_vec;

    return qd;
}
```

We tested the new control mode and compared to the previous velocity control. Here are reported the plots of the commanded velocities and the joint position values:

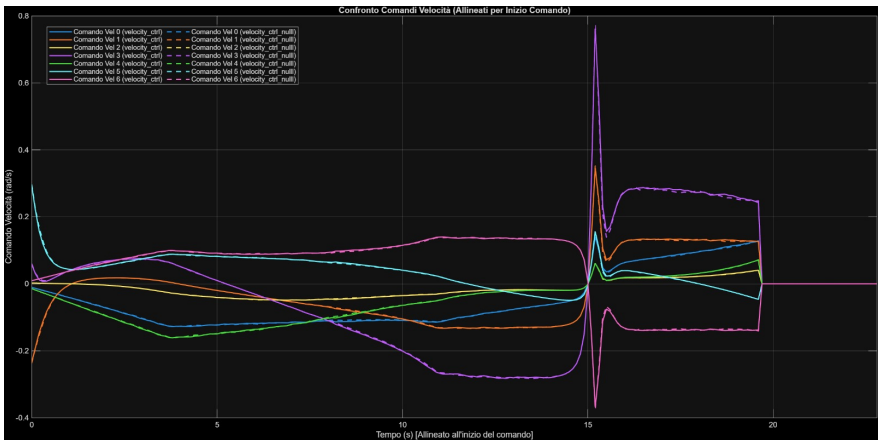

Figure 1: velocity command comparison

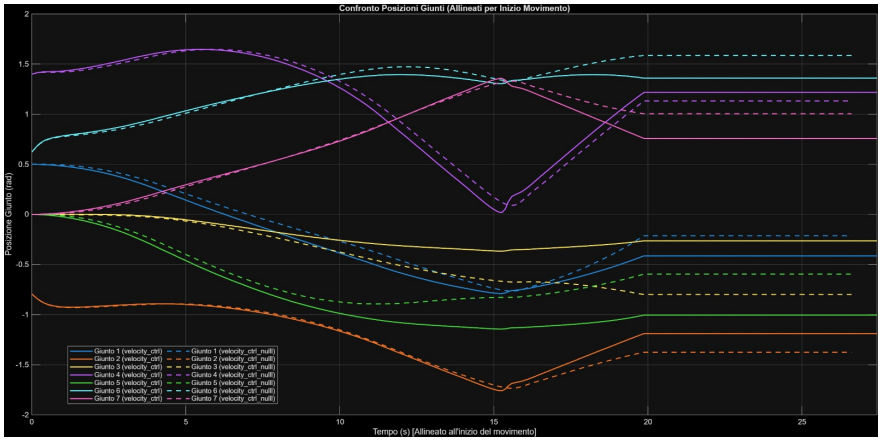

Figure 2: joints position values

Adding these lines in the launch file, we can switch between the two velocity controllers:

```
cmd_interface_arg = DeclareLaunchArgument(
        'cmd_interface',
        default_value='velocity',
        description='Select controller: position, velocity or effort'
    )

    ctrl_arg = DeclareLaunchArgument(
        'ctrl',
        default_value='velocity_ctrl',
        description='Select velocity controller: velocity_ctrl or
    velocity_ctrl_null'
    )

    cmd_interface_val = LaunchConfiguration('cmd_interface')
    ctrl_val = LaunchConfiguration('ctrl')
```

Launching this command we can use the new velocity controller:

```
ros2 launch ros2_kdl_package launching.launch.py cmd_interface:=velocity ctrl:=
    velocity_ctrl_null
```

(c) We made our `ros2_kdl_node` an action server that executes the same linear trajectory and publishes the position error as feedback.
First of all we made a custom interface for this action and we put it in the action folder:

```
# Goal
int32 order
---
# Result
bool success
---
# Feedback
float64[3] position_error
```

Then we added those lines in the CMakeLists file:

```
find_package(rclcpp_action REQUIRED)
find_package(action_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/ExecuteTrajectory.action"
  DEPENDENCIES builtin_interfaces
)

rosidl_target_interfaces(ros2_kdl_node
```

```
    ${PROJECT_NAME} "rosidl_typesupport_cpp"
  )
```

Specifically, the `find_package(rclcpp_action REQUIRED)` and `rosidl_generate_interfaces ()` commands allow the use of the APIs for managing actions and the automatic generation of goal, feedback, and result messages from the .action file, respectively.
Then in the package.xml file we added those lines:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

The `rosidl_default_generators` package automatically generates the source code for the ExecuteTrajectory action, while `action_msgs` contains the standard messages used for communication between client and server. Finally, the `<member_of_group>rosidl_interface_packages</member_of_group>` tag indicates that the package defines ROS 2 interfaces, so the build system can recognize and connect them correctly.
Finally in the `ros2_kdl_node.cpp` file we added some lines to define the action type and the handler:

```
using ExecuteTrajectory = ros2_kdl_package::action::ExecuteTrajectory;
using GoalHandleExecuteTrajectory = rclcpp_action::ServerGoalHandle<
  ExecuteTrajectory>;
```

Before creating the action server we removed the wall_timer since we don't want to send the commands periodically based on a timer but we want to activate this action with a client.
Then we created the actual server and we binded the 3 foundamental callback functions:

```
this->action_server_ = rclcpp_action::create_server<ExecuteTrajectory>(
    this,
    "ExecuteTrajectory",
    std::bind(&Iiwa_pub_sub::handle_goal, this, _1, _2),
    std::bind(&Iiwa_pub_sub::handle_cancel, this, _1),
    std::bind(&Iiwa_pub_sub::handle_accepted, this, _1));
```

Those functions are needed to handle new goal requests, goal cancellation and acceptation
Here we can see the `handle_goal` and the `handle_cancel`.

```
rclcpp_action::GoalResponse handle_goal(const rclcpp_action::GoalUUID & uuid,
                                        std::shared_ptr<const
  ExecuteTrajectory::Goal> goal){
    RCLCPP_INFO(this->get_logger(), "Received goal request with order %d",
  goal->order);
    return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
}

rclcpp_action::CancelResponse handle_cancel(const std::shared_ptr<
  GoalHandleExecuteTrajectory> goal_handle){
    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
    return rclcpp_action::CancelResponse::ACCEPT;
}
```

The most important one is the `handle_accepted`:

```
void handle_accepted(const std::shared_ptr<GoalHandleExecuteTrajectory>
  goal_handle){
    std::thread{std::bind(&Iiwa_pub_sub::execute, this, _1), goal_handle}.
  detach();
}
```

It executes a thread running the `execute()` function:
In this function we define the feedback and the result

```
auto feedback = std::make_shared<ExecuteTrajectory::Feedback>();
auto result = std::make_shared<ExecuteTrajectory::Result>();
```

We use a while loop to continously update the robot state and compute the control commands:

```
while (rclcpp::ok() && t < total_time && ctrl_ != "vision") {
    if (goal_handle->is_canceling()) {
        goal_handle->canceled(result);
        return;
    }
    ...
    t+=dt;
    rate.sleep();
}
```

Note: In the next chapter we will add the vision control too so in the execute function we have a different behaviour depending on the control mode.
This is how we compute the trajectory and the error

```
if(traj_type_ == "linear"){
    if(s_type_ == "trapezoidal")
        p_ = planner_.linear_traj_trapezoidal(t);
    else
        p_ = planner_.linear_traj_cubic(t);
}

Eigen::Vector3d error = computeLinearError(p_.pos, Eigen::Vector3d(cartpos.p.
  data));
```

Then we send the error as feedback for the client:

```
feedback->position_error = {error(0), error(1), error(2)};
goal_handle->publish_feedback(feedback);
```

Then we use the control method we made in the previous steps and finally we send the commands:

```
robot_->update(toStdVector(joint_positions_.data), toStdVector(
  joint_velocities_.data));
```

```
std_msgs::msg::Float64MultiArray cmd_msg;
  cmd_msg.data.assign(joint_velocities_cmd_.data.data(),
                              joint_velocities_cmd_.data.data() +
  joint_velocities_cmd_.rows());
  cmdPublisher_->publish(cmd_msg);
```

If instead the selected control is vision we enter in this loop:

```
while(rclcpp::ok() && ctrl_ == "vision"){
    ...
    goal_handle->publish_feedback(feedback);
    cmdPublisher_->publish(cmd_msg);
}
```

At the end of the execution we send the success message:

```
result->success = true;
goal_handle->succeed(result);
```

Now we only have to create the action client that will send the order to the server. We made a new node called `ros2_kdl_node_client.cpp` and we created a client for the `ExecuteTrajectory` action;

```
client_ = rclcpp_action::create_client<ExecuteTrajectory>(this, "
  ExecuteTrajectory");
```

Then we created the `send_goal()` function to define the goal message and we send it to the server:

```
void send_goal(int order)
{
  auto goal_msg = ExecuteTrajectory::Goal();
  goal_msg.order = order;

  RCLCPP_INFO(this->get_logger(), "Sending Goal: %d", order);

  auto options = rclcpp_action::Client<ExecuteTrajectory>::SendGoalOptions();
  options.goal_response_callback =
      std::bind(&ExecuteTrajectoryClient::goal_response_callback, this, std::
  placeholders::_1);
  options.feedback_callback =
      std::bind(&ExecuteTrajectoryClient::feedback_callback, this, std::
  placeholders::_1, std::placeholders::_2);
  options.result_callback =
      std::bind(&ExecuteTrajectoryClient::result_callback, this, std::
  placeholders::_1);

  client_->async_send_goal(goal_msg, options);
}
```

After the goal is sent the client runs the following function:

```
 void goal_response_callback(const GoalHandleExecuteTrajectory::SharedPtr &
  goal_handle)
{
  if (!goal_handle) {
    RCLCPP_ERROR(get_logger(), "Goal rifiutato dal server");
  } else {
    RCLCPP_INFO(get_logger(), "Goal accettato, in attesa del risultato...");
  }
}
```

Then when the action starts it executes this function:

```
void feedback_callback(
  GoalHandleExecuteTrajectory::SharedPtr,
  const std::shared_ptr<const ExecuteTrajectory::Feedback> feedback)
{
  RCLCPP_INFO(get_logger(),
              "Feedback: position_error = [%f, %f, %f]",
              feedback->position_error[0],
              feedback->position_error[1],
              feedback->position_error[2]);
}
```

that delivers info during the execution (in this case the `position_error`)
Finally we have the `result_callback()` function:

```
void result_callback(const GoalHandleExecuteTrajectory::WrappedResult & result
  )
{
  switch (result.code) {
    case rclcpp_action::ResultCode::SUCCEEDED:
      RCLCPP_INFO(get_logger(), "Goal Completed Succesfully: %s",
                  result.result->success ? "true" : "false");
      break;
    case rclcpp_action::ResultCode::ABORTED:
      RCLCPP_ERROR(get_logger(), "Goal Aborted");
      break;
    case rclcpp_action::ResultCode::CANCELED:
      RCLCPP_WARN(get_logger(), "Goal Cancelled");
      break;
    default:
      RCLCPP_ERROR(get_logger(), "Unknown Result");
      break;
  }
  rclcpp::shutdown();
}
```

This is called at the end of the execution and it just checks the goal state.

2. Vision-based control

(a) We constructed a gazebo world, named `empty.world`, inserting an aruco tag and detected it via the `aruco_ros` package (link here). With these lines we added the aruco tag:

```
<include>
 <uri>
   model://aruco_tag
 </uri>
 <name>aruco_tag</name>
 <pose>0 -0.707 0.707 0 1.57 0</pose>
</include>
```

Then we created a folder `gazebo/models`, in the `iiwa_description` package of the `ros2_iiwa` stack, containing the aruco marker model for gazebo.

We created a new model named `aruco_tag` and imported it into a new Gazebo world as a static object in a position that is visible by the camera. Then we saved the new world into the `/gazebo /worlds/`folder.

This is the `model.sdf` file:

```
  <?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.9'>
  <model name='aruco_tag'>
    <static>true</static>
    <link name='base_link'>
      <visual name='aruco_visual'>
        <geometry>
          <box>
            <size>0.1 0.1 0.01</size>
          </box>
        </geometry>
        <material>
          <diffuse>1 1 1 1</diffuse>
          <specular>0.4 0.4 0.4 1</specular>
          <pbr>
            <metal>
              <albedo_map>model://aruco_tag/aruco-20.png</albedo_map>
            </metal>
          </pbr>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

And this is the code of our `aruco_world`:

```
<?xml version="1.0" ?>
<sdf version="1.4">
<world name="aruco_world">
```

```
<physics name="1ms" type="ignored">
    <max_step_size>0.001</max_step_size>
    <real_time_factor>1.0</real_time_factor>
    </physics>
    <plugin
      filename="ignition-gazebo-physics-system"
      name="gz::sim::systems::Physics">
    </plugin>
    <plugin
      filename="ignition-gazebo-user-commands-system"
      name="gz::sim::systems::UserCommands">
    </plugin>
    <plugin
      filename="ignition-gazebo-scene-broadcaster-system"
      name="gz::sim::systems::SceneBroadcaster">
    </plugin>
    <plugin
      filename="ignition-gazebo-contact-system"
      name="gz::sim::systems::Contact">
    </plugin>

    <light type="directional" name="sun">
      <cast_shadows>true</cast_shadows>
      <pose>0 0 10 0 0 0</pose>
      <diffuse>0.8 0.8 0.8 1</diffuse>
      <specular>0.2 0.2 0.2 1</specular>
      <attenuation>
        <range>1000</range>
        <constant>0.9</constant>
        <linear>0.01</linear>
        <quadratic>0.001</quadratic>
      </attenuation>
      <direction>-0.5 0.1 -0.9</direction>
    </light>

    <model name="ground_plane">
      <static>true</static>
      <link name="link">
        <collision name="collision">
          <geometry>
            <plane>
              <normal>0 0 1</normal>
              <size>100 100</size>
            </plane>
          </geometry>
        </collision>
        <visual name="visual">
          <geometry>
```

```
            <plane>
              <normal>0 0 1</normal>
              <size>100 100</size>
            </plane>
          </geometry>
          <material>
            <ambient>0.8 0.8 0.8 1</ambient>
            <diffuse>0.8 0.8 0.8 1</diffuse>
            <specular>0.8 0.8 0.8 1</specular>
          </material>
        </visual>
      </link>
    </model>


<include>
  <uri>
    model://aruco_tag/
  </uri>
  <name>aruco_tag</name>
  <pose>-0.32 -0.83 0.43 2.54 1.57 3.14</pose>
</include>

<gravity>0 0 -9.81</gravity>

  </world>
</sdf>
```

(b) We spawned the robot in our new gazebo world modifying the launch file:

```
    iiwa_simulation_world = PathJoinSubstitution(
      [FindPackageShare(description_package),
          'gazebo/worlds', 'aruco_world']
)

declared_arguments.append(
    DeclareLaunchArgument(
        'gz_args',
        default_value=[
            TextSubstitution(text='-r -v 1 '),
            iiwa_simulation_world
        ],
        description='Arguments for gz_sim'
    )
)
```

Then in `kdl_control` class we added a new vision-based controller called `vision_ctrl` , it can be activated from the terminal as well as the other velocity_ctrl by inserting "vision" in the `ctrl` ROS parameter.

Then we created a subscribed to the aruco topic `/aruco_single/pose` in the source code `ros2_kdl_node.cpp`

```
auto qos_profile = rclcpp::QoS(rclcpp::KeepLast(10)).reliable();
MarkerPoseSubscriber_ = this->create_subscription<geometry_msgs::msg::
    PoseStamped>(
            "/aruco_single/pose", qos_profile, std::bind(&Iiwa_pub_sub::
    aruco_pose_subscriber, this, std::placeholders::_1));

            ...

            void aruco_pose_subscriber(const geometry_msgs::msg::PoseStamped&
    pose_stamped_msg)
    {
        cPo_(0) = pose_stamped_msg.pose.position.x;
        cPo_(1) = pose_stamped_msg.pose.position.y;
        cPo_(2) = pose_stamped_msg.pose.position.z;
    }
```

As already said, in `kdl_control.cpp` we added the `vision_ctrl` class:

```
KDL::JntArray KDLController::vision_ctrl(int Kp, Eigen::Vector3d cPo,Eigen::
    Vector3d sd )
{
    unsigned int nj = robot_->getNrJnts();

    Eigen::Matrix<double,3,3> Rc;
    Rc = toEigen(robot_->getEEFrame().M);//assumiamo che la matrice di rotazione
     siano approssimabili
    Eigen::MatrixXd K(nj,nj);
    K = 3*Kp*K.Identity(nj,nj);

    Eigen::Matrix<double,6,6> R =Eigen::Matrix<double,6,6>::Zero();

    R.block<3, 3>(0, 0) = Rc;
    R.block<3, 3>(3, 3) = Rc;//.transpose();

    Eigen::Vector3d s;
    for (int i=0; i<3; i++){
        s(i) = cPo(i)/cPo.norm();
    }


    RCLCPP_INFO(rclcpp::get_logger("KDLController"),
                "vector s: %f %f %f",
                s(0),s(1),s(2));

    Eigen::Matrix<double,3,3> L1;
    L1 = -1/cPo.norm()* (Eigen::Matrix3d::Identity() - s*s.transpose());

    Eigen::Matrix3d S_skew = Eigen::Matrix3d::Zero();
```

```
S_skew <<     0, -s.z(),  s.y(),
             s.z(),      0, -s.x(),
            -s.y(),  s.x(),      0;


Eigen::Matrix<double,3,3> L2;
L2 = S_skew;

Eigen::Matrix<double,3,6> L;

L.block<3, 3>(0, 0) = L1;
L.block<3, 3>(0, 3) = L2;

L = L*R;

Eigen::MatrixXd J;
J = robot_->getEEJacobian().data;
Eigen::MatrixXd Jc;
Jc  = J; //assumiamo che i due jacobiani siano uguali


Eigen::MatrixXd I;
I = Eigen::MatrixXd::Identity(nj,nj);

Eigen::MatrixXd JntLimits_ (nj,2);
JntLimits_ = robot_->getJntLimits();

Eigen::VectorXd q_min(nj);
Eigen::VectorXd q_max(nj);
q_min = JntLimits_.col(0);
q_max = JntLimits_.col(1);

Eigen::VectorXd q(nj);
q  = robot_->getJntValues();

double lambda = 50;

Eigen::VectorXd q0_dot(nj);
for (unsigned int i = 0; i<nj; i++) {

    double L =(q_max(i) - q_min(i))*(q_max(i) - q_min(i));

    double G = (2*q(i) - q_max(i) - q_min(i));

    double D = (q_max(i)- q(i))*(q(i)- q_min(i));

    q0_dot(i) = 1/lambda*L*G/(D*D);

}
```

```
    Eigen::MatrixXd N (nj,nj);

    N = I - pseudoinverse(J)*J;

    Eigen::MatrixXd J_pinv = pseudoinverse(L*J);
    KDL::JntArray qd(nj);
    qd.data =  K*J_pinv*sd + N * q0_dot;


    return qd;
}
```

Since the camera and the end effector are very near, we chose as Jacobian camera, the Jacobian of the end effector. We did the same for the Rotation matrix, we were aware that they are different joints, but since we are not much interested about the orientation of the camera, with respect to the aruco tag, this approximation can be acceptable.
Then we computer the matrix R, the assignment required that R is

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

However we tried to impose this equation, but we noticed that the control performed much better without imposing the Transposition on the R_c matrix. Then we computed the unit-norm axis s and calculated the $L(s)$ matrix. Eventually we imposed that

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

Where K is a diagonal matrix and $\mathbf{s}_d = [0, 0, 1]$ which is the desired pointing direction.
While the current direction is:

$$\mathbf{s} = \frac{{}^c\mathbf{P}o}{\|{}^c\mathbf{P}o\|}$$

$$L(\mathbf{s}) = \left[-\frac{1}{\|{}^c\mathbf{P}_o\|}\left(\mathbf{I} - \mathbf{s}\mathbf{s}^{\top}\right)\mathbf{S}(\mathbf{s})\right]\mathbf{R}$$

At this link it's possible to see the video of robot following the aruco tag. The plots shown in the video are here retrieved 3

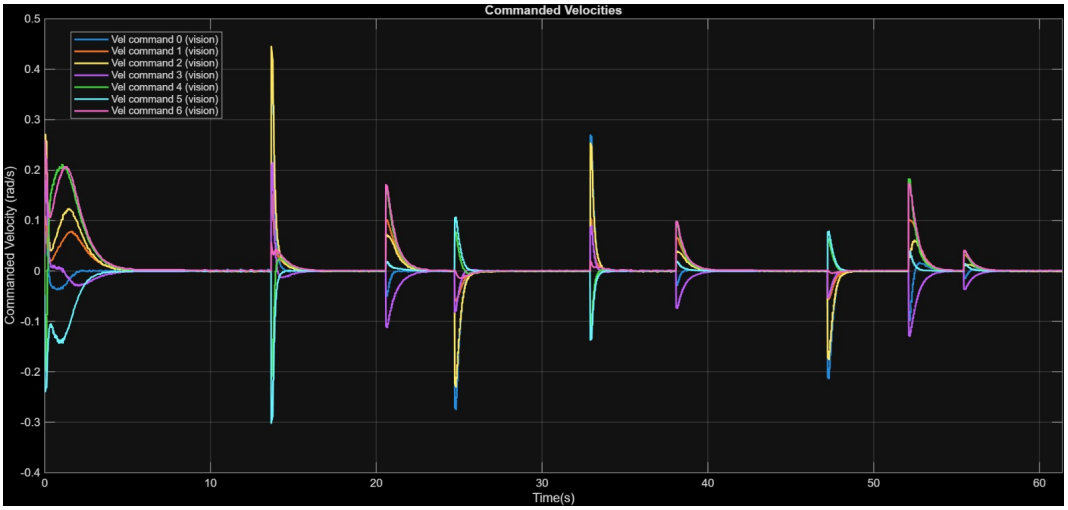

Figure 3: Commanded velocities with Vision Control

(c) We created a ROS 2 service to update the aruco marker position in Gazebo. We started from the `/set_pose` ign service and then created a `parameter_bridge` in the previously created launch file.
This is the bridge code:

```
bridge_set_pose = Node(
      package='ros_gz_bridge',
      executable='parameter_bridge',
      arguments=[
          f'/world/aruco_world/set_pose@ros_gz_interfaces/srv/SetEntityPose@gz
  .msgs.Pose@gz.msgs.Boolean',
      ],
      remappings=[
          (f'/world/aruco_world/set_pose', '/set_aruco_pose'),
      ],
      output='screen'
  )
```

Then we tested the bridged service via ROS 2 service call:

```
ros2 service call /set_aruco_pose ros_gz_interfaces/srv/SetEntityPose '{"entity
    ": {"name": "aruco_tag", "type": 2}, "pose": {"position": {"x": 1.0, "y":
    0.0, "z": 0.5}, "orientation": {"x": 0.0, "y": 0.0, "z": 0.0, "w": 1.0}}}'
```

This video shows how it works: https://youtu.be/Mql9K-V0ZbU