Robotics Lab: Homework 2

Control your robot

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

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:

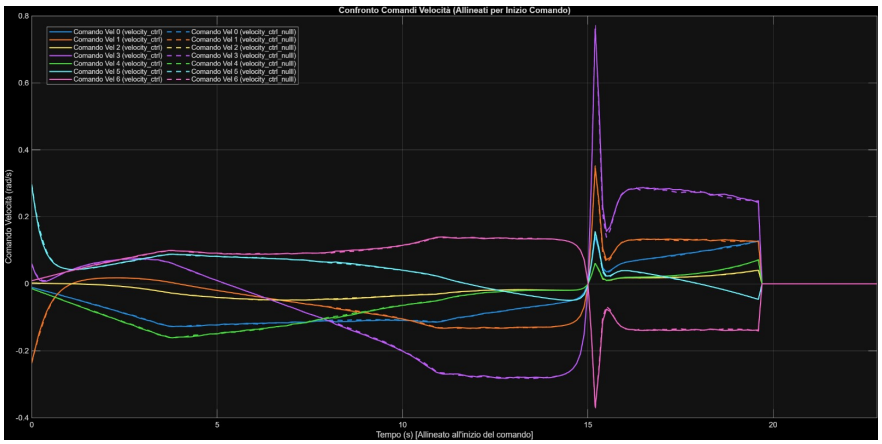

Figure 1: velocity command comparison

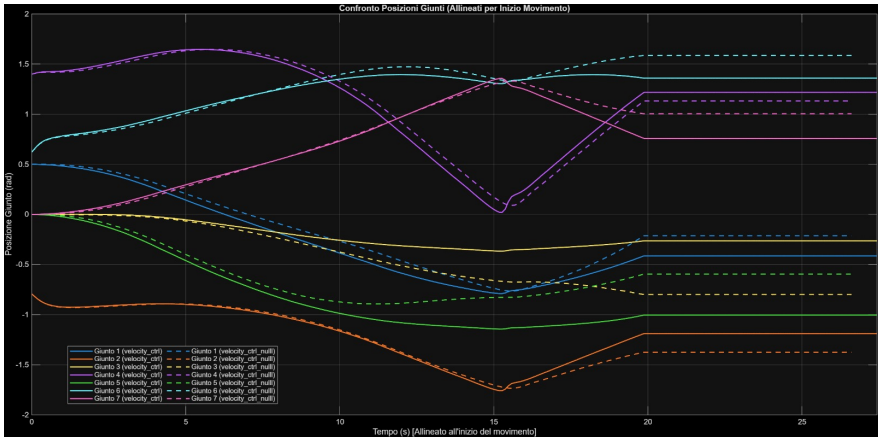

Figure 2: joints position values

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
```

2. We added sensors and controllers to our robot and spawned it in Gazebo

   (a) We had to create a package named `armando_gazebo` using ros2 CLI. Within this package, we created a launch folder containing an `armando_world.launch` file and filled it with commands that load the URDF into the `/robot_description` topic and spawn the robot using the create node in the `ros_gz_sim` package

```
user@alex:~/ros2_ws/src$ tree
   armando_gazebo
        CMakeLists.txt
        LICENSE
        launch
            armando_world.launch.py
        package.xml
        urdf
            armando_camera.xacro
```

   We took the main functions from the `gazebo.launch.py` from the repository https://github.com/RoboticsLab2025/ros2_urdf.git to get the robot to spawn in gazebo. In line 27 of our `armando_world.launch.py` file we modified the default value of `package_arg` setting it to "armando_description". We also modified the default value of `model_arg` (line 33) to "urdf/amr.urdf"; note that this file in the next points will become "arm.urdf.xacro".

   In the `package.xml` file, located in the `armando_gazebo` folder we added this two commands:

```
<depend>urdf-launch</depend> (line 17)

<gazebo_ros gazebo_model_path="${prefix}/.." /> (line 21)
```

   Here is the output of the command

```
ros2 launch armando_gazebo armando_world.launch.py
```

   (b) We add a `PositionJointInterface` as a hardware interface to our robot using the `ros2_control` framework.

   This is the part of the code we put in the `armando_hardware_interface.xacro` file that allowed us to use the framework. The file is in the `armando_description/urdf` folder and it contains a macro that defines the hardware interface for the joints of our robot.

```
<gazebo>
  <plugin filename="ign_ros2_control-system" name="ign_ros2_control::
  IgnitionROS2ControlPlugin">
  <parameters>$(find armando_description)/config/armando_controllers.yaml</
  parameters>
    <controller_manager_prefix_node_name>controller_manager</
  controller_manager_prefix_node_name>
  </plugin>
</gazebo>
<xacro:include filename = "$(find armando_description)/urdf/ros2_control/
  joint_initial_pos.xacro"/>
<ros2_control name="HardwareInterface_Ignition" type="system">
```

```
<hardware>
  <plugin>ign_ros2_control/IgnitionSystem</plugin>
</hardware>
```

Then we added this line to `arm.urdf.xacro` to include it in our main:

```
 <xacro:include filename="$(find armando_description)/urdf/
armando_hardware_interface.xacro"/>
```

In the end, to load the URDF in our launch file we added this line to `armando_display.launch.py`:

```
        xacro_path = os.path.join(armando_description_path, "urdf", "arm.urdf.
xacro")
```

(c) We added inside the `arm.urdf.xacro` the commands to enable the Gazebo ROS2 control plugin and load the joint position controllers from the `armando_controllers.yaml` file :

```
<gazebo>
    <plugin filename="ign_ros2_control-system" name="ign_ros2_control::
    IgnitionROS2ControlPlugin">
    <parameters>$(find armando_description)/config/armando_controllers.yaml</
    parameters>
      <controller_manager_prefix_node_name>controller_manager</
    controller_manager_prefix_node_name>
    </plugin>
  </gazebo>
```

In the `.yaml` file we have:

```
  position_controller:
  ros__parameters:
    command_interfaces:
      - position
    state_interfaces:
      - position
      - velocity
    joints:
      - j0
      - j1
      - j2
      - j3
```

To spawn the joint state broadcaster and the position controllers we added these lines in the `armando_world.launch.py`:

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

    position_controller = Node(
        package="controller_manager",
        executable="spawner",
        arguments=[
            "position_controller",
            "--controller-manager",
            "/controller_manager"
        ],
        condition=UnlessCondition(PythonExpression(["'",control_mode_selected,"'
    ", " == 'trajectory'"]))
    )
```

Note that this is the final version of our code, so it's already implemented the condition section which controls if the mode selected is "position" or "trajectory".
To load the controllers after Gazebo is started we added these lines in `armando_world.launch.py`:

```
delay_joint_state_broadcaster = RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=gz_spawn_entity,
            on_exit=[joint_state_broadcaster],
        )
    )
    delay_controller = RegisterEventHandler(
    event_handler=OnProcessExit(
        target_action=gz_spawn_entity,
        on_exit=[position_controller,trajectory_controller],
    )
)
```

This graph, obtained by launching the command `rqt_graph` while Gazebo is running, shows that the hardware interface is correctly loaded and connected:

Also from the terminal we have these messages: