LAB 8: Admittance Control and Planning

Michele Focchi and Matteo Saveriano

The goals of this assignment are:

- learning the basic procedure to design an admittance controller for the end-effector of a manipulator in contact with the environment with the purpose to control the interaction with a human.
- Implement an obstacle avoidance planning algorithm base on potential fields.

Tools that we are going to be using in this lab (all open-source):

- Python programming (2.7/3.5)[1]
- Robot Operating System (ROS)[2]
- Gazebo [3]
- Pinocchio Library for Rigid Body Dynamics Algorithms[4]
- Locosim[5]

In this assignment we will modify the set-point for the end-effector computed in previous labs, using an admittance model, to accommodate for an external force acting at the end-effector. An inverse kinematics scheme will map the new references into joint references. We will close the loop for tracking at the joint level. The implementation of and admittance control scheme is the only way to be able to achieve safe interactions with industrial robots that accept only position (and not torque) references. Indeed, most of the times, industrial robots are not equipped with an inner torque loop, but with a (stiff) position loop at the low level, plus a 6 axis force sensor located at the end-effector. Differently from the approaches seen until now, that enable a safe interaction with any part of the robot, this approach allows the robot to be compliant (and safe) *only* if the interaction occurs at the end-effector location, where we are able to *sense* the force.

[1] https://docs.python.org/
[2] https://www.ros.org/
[3] http://gazebosim.org/
[4] https://github.com/stack-of-tasks/pinocchio
[5] https://github.com/mfocchi/locosim

0 Preliminaries

The robot that we will use this time, is the 6DoFs Ur5 robot (see Fig. 1), but, differently from before, we will not integrate the dynamics by ourself with a forward Euler method, but this task will be delegated to a simulator called Gazebo. An intermediate controller (`ros_impedance_controller`) will receive the PD gains and the set-points (of desired joint positions and velocities) and feed-forwards for the torques generated by our Python node. The `ros_impedance_controller` will either interface to the Gazebo simulator or to the real robot. This has the big advantage of allowing us to use the *same* code in both situations, and safely test our algorithm before going on the real robot.

For this part of the assignment, we will use these four main files:

- `LOCOSIM/robot_control/base_controllers/base_controller_fixed.py` (Mother class with a generic controller for fixed based manipulators)
- `LOCOSIM/robot_control/base_controllers/L8_admittance.py` (Main file to run the visualization and all exercises (inherits from BaseControllerFixed))
- `LOCOSIM/robot_control/params.py` (Initializations of variables and simulation parameter for UR5)
- `LOCOSIM/robot_control/L8_conf.py` (Initializations of variables specific for L8 lab exercises)
- `LOCOSIM/robot_control/base_controllers/obstacle_avoidance/obstacle_avoidance.py` (Class that implements the artificial potential field planner)
- `LOCOSIM/robot_control/base_controllers/admittance_controller.py` (Class that implements the admittance model and controller)
- `LOCOSIM/robot_control/base_controllers/inverse_kinematics/inv_kinematics_pinocchio.py` (Class that implements numerical inverse kinematics for a generic fixed based robot based on Pinocccchio library)

1 Position Control

1.1 Position Control - Constant Joint Reference

Since the real prototype of the UR5 robot it does not accept desired torque command but only position and velocity commands, the first experience we are going to do is to create a constant set-point in position for our robot. Therefore, in the `params.yaml` configuration file, set `real_robot: True`, `control_mode: point`, `control_type: position`. Then, to check that everything works, set a constant reference for joint position equal $q^0 = \begin{bmatrix} 0.5 & -1.0 & 1.0 & -1.7 & -1.7 & 0.0 \end{bmatrix}^T$. Note that the real robot might start in a

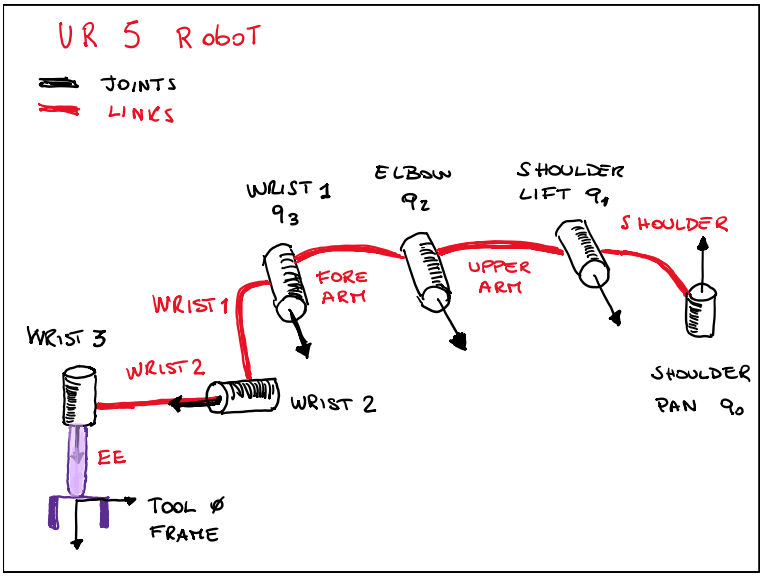

Figure 1: Sketch of the UR5 robot kinematics

different configuration and to avoid abrupt motions you need to implement a *homing* procedure to drive smoothly the joints to the q^0 set-point.

1.2 Position Control - Sinusoidal Joint Reference.

Now set a sinusoidal reference starting from the q^0 configuration with amplitude $A = 0.15$ rad and frequency $f = 0.25$ Hz equal for all the joints.

$$q^d(t) = q_0 + Asin(2\pi ft)$$

Try to avoid to set too large amplitude in order to avoid hitting some obstacles. Now that we are using Gazebo as a simulator you can set the fields `<dynamics damping="0.0" friction="0.0"/>` in the joint's tag to emulate the effect of viscous friction (damping parameter) and static friction (friction parameter) at the joint. Observe the effect of friction in the tracking of a sinuoidal trajectory as we did for the DC motor.

1.3 Position Control - Constant Cartesian Reference

Now let's set a constant Cartesian position (expressed in the base frame, check the base frame axes) for the end-effector $p_e^d = \begin{bmatrix} -0.3 & 0.5 & -0.5 \end{bmatrix}^T$. Employ the inverse kinematics function implemented in L1-2.6 and solve the redundancy setting a postural task $q^p = q_0$.

1.4 Position Control - Constant Cartesian Reference and Orientation.

In addition to the end-effector position, now we want also to specify a desired orientation with ${}_W R_e^d = eul2rot(\begin{bmatrix} -1.9 & -0.5 & -0.1 \end{bmatrix})$. In this case the task is 6D and there is no need of regularization or of a postural task, because the full Jacobian $J6 \in \mathbb{R}^{6\times 6}$ should

we used. Note that the now we have to append also the orientation error e_o to e_p. This can be computed with the angle-axis approach or equivalent way as we did for the L3 lab:

$$e_o = \text{logm}({}_w R_e^T \, {}_w R_e^d) \tag{1}$$

1.5 Position Control - Polynomial trajectory

If you implemented both 1.2 and 1.3 on the real robot, you will notice that the robot goes into protection mode because we are setting an abrupt change in the position set-point q^d that goes from q_0 to $IK(p_e^d)$. Taking inspiration from L1-2.5, design polynomial trajectories (if you want you can create a class for this) for the end-effector Cartesian coordinates to go from the actual position ($FK(q^0)$) to p_e^d. You can start with a quintic polynomial and set zero initial /final velocity and accelerations. Note that only for the quintic it will be possible to enforce the accelerations. Do the implementation also for a cubic polynomial and compare the results. We leave as an exercise the creation of polynomials also for the orientation (hint. use Euler Angles).

2 Admittance Control

An admittance control framework requires two components: an *admittance model* and an *inverse kinematics* function. The philosophy behind *admittance* control is to change the end-effector position set-point in order to mimic a desired admittance at the end-effector. [6] We recall that an admittance is a linear model that receives as input a *force* and outputs a displacement Δp_e (or a velocity). Then, the displacement Δp_e is added to the set-point creating a new reference for the end-effector $\tilde{p_e}^d = \Delta p_e + {p_e}^d$ This can be mapped to joints references via an inverse kinematics function, as shown in figure 2, or we can directly map Δp_e via a jacobian to joint displacements Δq and then sum them to q^d:

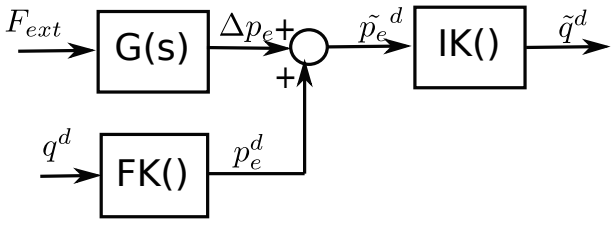

Figure 2: Block diagram for an admittance controller in the Cartesian space. FK(.) and IK(.) are the forward (or direct) and inverse kinematics blocks, respectively, while G(s) is the admittance model.

2.1 - Implement the admittance model

We want to implement an admittance model that represents a *linear* virtual spring and

[6] Mechanical admittance is the reciprocal of impedance

a *linear* virtual damper at the end-effector. The transfer function G(s) for this model in the Laplace domain is :

$$\begin{aligned} \Delta p_e &= \frac{F_{ext}}{D_x s + K_x} \\ G(s) &= \frac{\Delta p_e}{F_{ext}} = \frac{1}{D_x s + K_x} \end{aligned} \tag{2}$$

Where the input is the contact force $F_{ext} \in \mathbb{R}^3$ and the output the shift $\Delta p_e \in \mathbb{R}^3$ to be applied to the end-effector reference to emulate a virtual stiffness in response to F_{ext}. Note that this transfer function is in continuous time and we need a discretized version for a digital implementation on a computer. To get the discrete version, it is convenient to convert (2) first in the time domain:

$$D_x \Delta \dot{p_e} + K_x \Delta p_e = F_{ext}$$

Then, if we employ Backward Differentiation Formula $\dot{p_e} = (p_e(k) - p_e(k-1))dT^{-1}$ to approximate the derivative, then we can get the following difference equation:

$$\begin{aligned} &D_x \frac{\Delta p_e(k) - \Delta p_e(k-1)}{dT} + K_x \Delta p_e(k) = F_{ext}(k) \\ &\Delta p_e(k)(D_x dT^{-1} + K_x) - D_x \Delta p_e(k-1) dT^{-1} = F_{ext}(k) \\ &\Delta p_e(k) = (D_x dT^{-1} + K_x)^{-1}(F_{ext}(k) + D_x \Delta p_e(k-1) dT^{-1}) \end{aligned}$$

where (k)and (k-1) refer to the actual and previous samples of the variables.

2.2 Inverse Kinematics

Now that we wrote a function to implement the admittance model, we still miss to implement the inverse kinematics function. To be able to convert the modified reference $\tilde{p_e}^d(k) = p_e^d(k) + \Delta p_e(k)$ at each loop k into a joint reference q^d. Note that, since our admittance model (composed of linear spring and damper) involves only 3D vectors, and we have 6 DoFs in our robot, we will have redundancy, hence infinite solutions to the inverse kinematics problem. To solve this we will reuse the postural task implementation described in LAB L1-2.4 setting $q^p = q_0$.

2.3 - External push with a constant reference Now we are ready to test the implemented algorithm. Set a constant joint reference q_0 for the robot to follow (this will correspond to a constant end-effector position p^d that you will have to compute) and set the admittance gains to $K_x = 1000 I_{3\times3}$, $D_x = 300 I_{3\times3}$. We can have a feeling on how the controller works by applying an external force either in simulation (calling the

function `applyForce()` after 5 seconds) or pushing manually (gently!) the end-effector on the real robot (Important! keep always the E-stop in your hand to kill the robot in case some instability arise!). You should feel the robot deviating from the reference in the same direction of the force (we are setting a diagonal stiffness matrix), as if a spring is attached at the end-effector trying to push it back to the original position. Now try to reduce the stiffness gain to $K_x = 600 I_{3\times3}$, in order to make the robot more "soft". You should feel that for the same force you are applying you are getting a bigger deflection (almost double). Plot the signals of the end-effector position and force to evaluate the direction of the deflection is consistent with the force.

2.4 - External push with a sinusoidal reference Let's now apply a sinusoidal reference to the robot joints with amplitude $A = 0.15\ rad$ and frequency $f = 0.25\ Hz$, around the initial position q_0. And repeat the previous experiment. You will see the robot will deviate in a compliant way from the prescribed trajectory, allowing safe human-robot interaction.

2.5 - Payload estimation.
The final exercise of this lab will be to estimate a payload attached at the end effector in static conditions. We want to exploit the readings from the joint torques that are stored in the variable `tau`. Employing the principle of Virtual works we can estimate the an external force applied at the end-effector via the transpose of the end-effector Jacobian. Note that we will have also joint gravity torques to take into consideration that must be discarded. The derivation of the formula is the following, we start from the dynamic equation of the robot considering an external force F is applied from the environment, and set the joint accelerations and velocities to zero:

$$g = \tau + J^T F$$
$$F = -(J^T)^\dagger (\tau - g)$$

Note that you need to use the pseudo-inverse of the jacobian J to match dimensions. Then the payload weight is:

$$m_p = -F_z / g$$

Finally, apply some filtering to take out the noise.