LAB 4: Kinematics and Dynamics Lab:

Marco Camurri, Michele Focchi, Octavio Villarreal

In this assignment, you will learn:

- how to compute the inverse kinematics of a 5-DoF serial manipulator and visualize it using the RViz software
- compute and analyze the forward/inverse dynamics of a 4-DoF serial manipulator using the Recursive Newton-Euler Algorithm (RNEA)

For this lab, you need the following open-source software:

- Python interpreter (version 3.8)[1]
- Robot Operating System (ROS)[2]
- Pinocchio library for Rigid Body Dynamics Algorithms[3]
- Code available at `https://github.com/idra-lab/intro_robotics_labs`

2.6 Dealing with the redundancy: the postural task.

Now, let's consider a ur5 robot with 6 DoFs. In this case we want to set *only* the *position* of the end-effector to be at $p_e^d = \begin{bmatrix} 0.5 & -0.2 & 0.5 \end{bmatrix}^T$. Hence, the robot (6 DoFs) becomes redundant for the task (3 variables) and we expect to have an infinite number of solutions for the inverse kinematics problem. In particular the matrix $J^T J$ becomes singular (semi-positive definite). Instead of using a regularization that would select a configuration that depends on the initial guess, we want to have a solution that is as *close* as possible to a desired configuration q^p that we call *postural* configuration. This could be, for instance, a configuration that tries to keep the joints in the middle of their range or minimize gravity torques. To achieve this, we can implement a "postural task" at the kinematic level to solve the redundancy. This will select, among the infinite solutions, the one (joint configuration $\tilde{q}^\star$) that is closer (in an Euclidean sense) to the desired configuration q^p. To implement this, we setup the following optimization problem:

$$\tilde{q}^\star = \underset{q}{\operatorname{argmin}} \frac{1}{2} \left|\left| \begin{bmatrix} p(q) \\ wq \end{bmatrix} - \begin{bmatrix} p^d \\ wq^p \end{bmatrix} \right|\right|^2 = \frac{1}{2} \left|\left| \begin{bmatrix} e_x \\ e_q \end{bmatrix} \right|\right|^2$$

[1] https://docs.python.org/3.8
[2] https://www.ros.org/
[3] https://github.com/stack-of-tasks/pinocchio

where w is a scalar weight and q^p the postural configuration. The solution of this optimization problem (which is still non convex do to the non linear dependency of p_e from q) can be obtained through the Newton method, in a similar fashion to what we did in lab L1 - 2.4. The difference is the definition of the error $\bar{e} = \begin{bmatrix} e_x^T & e_q^T \end{bmatrix}^T$ (than now will be a $3+n$ vector) and the way the newton step Δq is computed:

$$\begin{aligned} \Delta q &= -\left(\underbrace{\begin{bmatrix} J^T & wI_{3\times 3} \end{bmatrix}}_{\bar{J}^T} \underbrace{\begin{bmatrix} J \\ wI_{3\times 3} \end{bmatrix}}_{\bar{J}} \right)^{-1} \left(J^T e_x + w^2 e_q \right) \\ &= -\left(\underbrace{J^T J + w^2 I_{3\times 3}}_{\bar{J}^T \bar{J}} \right)^{-1} \underbrace{\left(J^T e_x + w^2 e_q \right)}_{\bar{J}^T \bar{e}} \end{aligned}$$

So, with respect to the exercise 2.4: 1) we use w^2 instead of λ to regularize the Hessian $\bar{J}^T \bar{J}$ and make it full rank and therefore invertible, 2) we added ad term $w^2 e_q$ in the computation of the Newton step, 3) we changed the stopping criteria of the IK algorithm checking if the norm of this new gradient $\bar{J}^T \bar{e}$ goes below ϵ rather than $J^T \bar{e}$. Set different postural configurations and observe that the optimal configuration is the one that places the end-effector in the desired position and is close to the postural configuration selected. Try to (slightly) change the initial configuration and you will see that the ik solution is no longer changing and is always the same.

3 Robot dynamics

In this last part of the assignment we will use the Recursive Newton-Euler Algorithm (RNEA) to compute the robot dynamics without any joint torque. We will then proceed to compute each of the terms in the dynamics equation as shown during lecture F1.2. We will start by writing our own function for the forward and backward passes of the RNEA and compute each of the terms of the dynamics equation. Then we will use the previously computed terms to obtain the forward dynamics (i.e., joint accelerations) and integrate them once to obtain velocity and twice for position, using a forward/explicit Euler scheme. We will then compare our results with the built in functions of Pinocchio.

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

- `L1_2_dynamics.py` (Main file to run the visualization and the exercises)
- `L1_conf.py` (Initializations of variables and simulaion parameters)
- `utils/kin_dyn_utils.py` (Kinematics and dynamics functions)

3.1 Build the function for the RNEA (lecture F1.2).

Write a function that computes the forward and backward passes of the RNEA in the case of revolute joints, `RNEA`. We will use the following equations from lecture F1.2. Refer to Figure 1 for variable definitions.

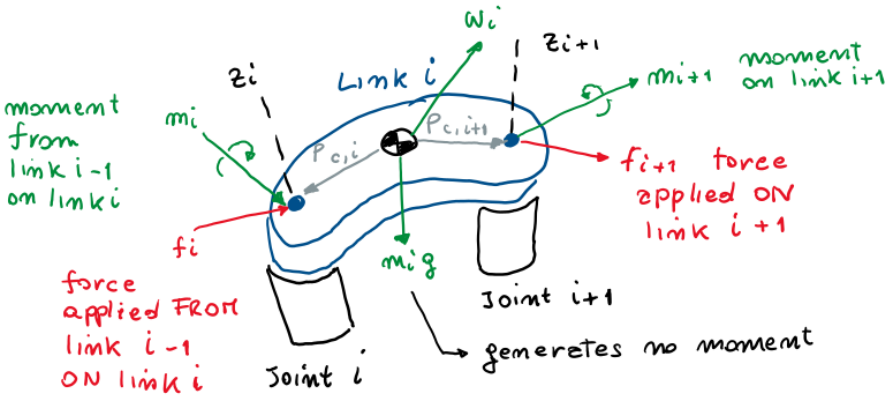

Figure 1: Sketch of a link i supported by joint i.

So, for the forward pass we have[4]:

$$\begin{aligned}
\boldsymbol{\omega}_i &= \boldsymbol{\omega}_{i-1} + \dot{\mathbf{q}}_i \mathbf{z}_i \\
\dot{\boldsymbol{\omega}}_i &= \dot{\boldsymbol{\omega}_{i-1}} + \ddot{\mathbf{q}}_i \mathbf{z}_i + \dot{\mathbf{q}}_i \boldsymbol{\omega}_{i-1} \times \mathbf{z}_i \\
\mathbf{v}_i &= \mathbf{v}_{i-1} + \boldsymbol{\omega}_{i-1} \times \mathbf{p}_{i-1,i} \\
\mathbf{a}_i &= \mathbf{a}_{i-1} + \dot{\boldsymbol{\omega}}_{i-1} \times \mathbf{p}_{i-1,i} + \boldsymbol{\omega}_{i-1} \times (\boldsymbol{\omega}_{i-1} \times \mathbf{p}_{i-1,i}) \\
\mathbf{a}_{ci} &= \mathbf{a}_i + \dot{\boldsymbol{\omega}}_i \times \mathbf{p}_{i,c} + \boldsymbol{\omega}_i \times (\boldsymbol{\omega}_i \times \mathbf{p}_{i,c})
\end{aligned}$$

where the last equations is the acceleration of the CoM, and vector $\mathbf{p}_{i,c}$ is the vector from joint frame i to the CoM frame. This expression will be used to compute the velocities and the accelerations along the entire kinematic chain. It is important to note that the forward pass will be computed from the base link, which is not moving and it is only subject to the acceleration of gravity (i.e., $\boldsymbol{\omega}_0 = 0$, $\dot{\boldsymbol{\omega}}_0 = 0$, $\mathbf{v}_0 = 0$ and $\mathbf{a}_0 = -\mathbf{g}$). This simplifies the computation of the forces since the acceleration of gravity is already considered in the forward pass. For the backward pass, we will begin our computation from the end-effector frame. Consider that if the end effector is performing free-motion, then the forces and moments are equal to zero (i.e., $\mathbf{f}_{ee} = 0$ and $\mathbf{m}_{ee} = 0$), however, if the end-effector is in contact with an object, a measurement or approximation of the forces and moments is needed. The backward pass equations for the moments and forces are

$$\begin{aligned}
\mathbf{f}_i &= \mathbf{f}_{i+1} + \mathbf{m}_i \mathbf{a}_{ci} \\
\mathbf{m}_i &= \mathbf{m}_{i+1} - \mathbf{p}_{c,i} \times \mathbf{f}_i + \mathbf{p}_{c,i+1} \times \mathbf{f}_{i+1} + \mathbf{I}_i \dot{\boldsymbol{\omega}}_i + \boldsymbol{\omega}_i \times \mathbf{I}_i \boldsymbol{\omega}_i
\end{aligned}$$

[4]all quantities in both the forward and backward passes are expressed with respect to the world frame

Remember that all quantities are expressed with respect to the world, so you will need to provide appropriate transformations for the parameter vectors and matrices (e.g., the position of the CoM $\mathbf{p}_{ci}$ and the inertia tensor $\mathbf{I}_i$). The function `RNEA` should receive as input arguments the gravity vector $\mathbf{g}_0$ ($\begin{bmatrix}0 & 0 & -9.81\end{bmatrix}^\top$), the vector of current joint positions $\mathbf{q}$, the vector of current joint velocities $\dot{\mathbf{q}}$ and the vector of current joint accelerations $\ddot{\mathbf{q}}$, i.e., `RNEA(g0,q,qd,qdd)`.

3.2 Compute the terms of the dynamics equation (lecture F1.2).

Create functions to compute the vector of gravitational force $\mathbf{g}$, the joint space inertia matrix $\mathbf{M}$, and the vector of Coriolis and centrifugal terms $\mathbf{c}$. Remember that the RNEA function previously written can be used to obtain all of these terms by setting some quantities to 0. In essence, compute the gravity vector through the previous function by setting its arguments as `RNEA(g0,q,0,0)`, the vector of Coriolis and centrifugal forces with `RNEA(0,q,qd,0)` and each of the columns of the joint space inertia matrix with `RNEA(0,q,0,ei)`, where `ei` is a vector to select the i-th column of $\mathbf{M}$.

3.3 Compute the robot joint accelerations (lecture F1.2).

Now that the equations of motion can be computed, simulate the robot dynamics under zero joint torque. To do this, you need to obtain the joint acceleration based on the equations of motion, namely:

$$\ddot{\mathbf{q}} = \mathbf{M}^{-1}(\boldsymbol{\tau} - \mathbf{g} - \mathbf{c})$$

with $\boldsymbol{\tau} = 0$ (for the time being).

3.4 Simulate the robot dynamics using a forward Euler scheme (lecture F1.2).

With the previously computed joint acceleration, we can integrate once to obtain joint velocity and twice for position. Implement an improved forward/explicit Euler integration scheme in the included `while` loop in the provided main file to run your visualization `L1_2_dynamics.py` by using the following equations:

$$\dot{\mathbf{q}}(t+\delta t) = \dot{\mathbf{q}}(t) + \ddot{\mathbf{q}}(t)\delta t$$
$$\mathbf{q}(t+\delta t) = \mathbf{q}(t) + \dot{\mathbf{q}}(t)\delta t + \frac{\delta t^2}{2}\ddot{\mathbf{q}}(t)$$

where δt is the time step of our simulation. Check the visualizer, you will see the robot falling but never stopping, why is this the case? Is this realistic? What can be done to

stop the motion after a while?.

3.5 Add a damping term to the equation.

The robot is not stopping since there is no friction or damping that allows the joint velocities to come to a stop. Implement a damping term by defining τ as

$$\boldsymbol{\tau} = -b\dot{\mathbf{q}}$$

where b is the damping coefficient of the joints. For now we keep b as a scalar, however, you can set different damping coefficients for each of the joints. Try different values of b and visualize the behavior of the robot changing.

3.6 Compare results against Pinocchio.

Compute the terms of the dynamics equation $\mathbf{M}$, $\mathbf{c}$ and $\mathbf{g}$ using the built in functions from Pinocchio. To compute $\mathbf{g}$ use the following function: `robot.gravity(q)`, which takes as only argument the vector of joint positions. To compute $\mathbf{M}$ use: `robot.mass(q, False)`[5]. In the case of vector $\mathbf{C}$, Pinocchio only provides the sum $\mathbf{h} = \mathbf{c} + \mathbf{g}$ through the function: `robot.nle(q, qd, False)`[6]. Run again the simulation using Pinocchio and verify that the outputs of the function that you wrote coincides with those of the built-in functions. Now remove entirely the implementation with your functions and visualize the robot falling. You will notice that the robot is moving faster that with your own implementation. Why is this the case? This is mainly due to the fact that Pinocchio is implemented in C++ but used in Python through bindings and because our implementation is done with respect to the world frame. Instead, inside Pinocchio computations are done with respect to the link frame, and this results in "sparser" matrix multiplications (i.e. the inertia tensors are full of zeros and this results in faster matrix multiplications).

3.7 Model end-stops.

Our joints have not an infinite range of motion and are usually limited by end-stops. Model this feature as an additional torque/force that gets activated when the joint exceed its limits. Set the joint range for all the joints to:

$$q_{\max} = \begin{bmatrix} 2\pi & 0.5 & 2\pi & 2\pi \end{bmatrix}^T \qquad q_{\min} = \begin{bmatrix} -2\pi & -0.5 & -2\pi & -2\pi \end{bmatrix}^T$$

[5]For the functions of the mass and nonlinear terms the second argument is a boolean related to the update of the joint variables, in our case we are keeping it as false since we are updating the joint states manually

[6]Nle stands for non linear effects.

If you run the simulation you will see that the second joint cannot move to the vertical as before due to the presence of the end-stop that limits its motion.