LAB03 - Differential and Inverse Kinematics

Marco Camurri

1 Introduction

In this Lab, you will learn

- how to compute the Geometric and Analytical Jacobian of a 4-DoF robot serial manipulator and visualize it using the RViz software
- how to numerically compute the inverse kinematics of the robot and use it to make it move through a predefined trajectory with quintic splines

2 Kinematics

2.1 Direct Kinematics

If needed, reehearse the direct kinematics exercises from the previous LAB.

2.2 Geometric Jacobian (lecture E3).

Now that you have devised a function to compute the direct kinematics, let's proceed to compute the end-effector Jacobian of the manipulator by modifying the function `computeEndEffectorJacobian` inside `kin_dyn_utils.py`.

Recall that the expression to compute the geometric Jacobian is:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{P1} & \dots & \mathbf{J}_{Pi} \\ \mathbf{J}_{O1} & \dots & \mathbf{J}_{Oi} \end{bmatrix}$$

where

$$\begin{bmatrix} \mathbf{J}_{Pi} \\ \mathbf{J}_{Oi} \end{bmatrix} = \begin{cases} \begin{bmatrix} \mathbf{z}_i \\ \mathbf{0} \end{bmatrix} & \text{if } i \text{ is prismatic} \\ \begin{bmatrix} \mathbf{z}_i \times (\mathbf{w}\mathbf{p}_e - \mathbf{w}\mathbf{p}_i) \\ \mathbf{z}_i \end{bmatrix} & \text{if } i \text{ is revolute} \end{cases}$$

In our case, all of the joints are revolute. Be also reminded that the vector $\mathbf{z}_i$ represents the moving axis of joint i that drives link i, expressed in the world frame.

As in the previous case, compare your results with the built-in function of Pinocchio: `robot.frameJacobian(q, frame_ee, False, pin.ReferenceFrame.LOCAL_WORLD_ALIGNED)` for different values of `q`.

2.3 Analytical Jacobian (lecture E3).

In case we want to use Euler Angles to express orientations in our task variables, we can compute the Analytic Jacobian directly from the Geometric Jacobian via a simple transformation.

The analytical Jacobian $\mathbf{J}_a$ can be computed as follows:

$$\mathbf{J}_a = \mathbf{T}_{RPY}\mathbf{J}$$

where $\mathbf{J}$ is the geometric Jacobian and $\mathbf{T}_a$ is the transformation matrix given by

$$\mathbf{T}_a = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_{RPY}^{-1} \end{bmatrix}$$

where $\mathbf{T}_{RPY}^{-1}$ is the transformation matrix from angular velocity $\boldsymbol{\omega}$ to Euler angle rate $\dot{\mathbf{\Phi}}$, i.e.,

$$\boldsymbol{\omega} = \mathbf{T}_{RPY}\dot{\mathbf{\Phi}}$$

and implement it in the function `geometric2analyticJacobian`..

Let's have a look to the structure of the Analytical Jacobian and compare it with the Geometrical one, both computed at the q_0 configuration.

Note there is a zero row in the angular part(the linear part is the same) in two different directions.

Why is that?

```
Geometric Jacobian:
[ 0.11015 -0.43244 -0.2698   0.0414 ]
[-0.68538  0.       0.      -0.     ]
[ 0.      -0.68538 -0.29273 -0.05395]
[ 0.      -0.      -0.      -0.     ]
[ 0.      -1.      -1.      -1.     ]
[ 1.       0.       0.       0.     ]

Analytic Jacobian:
[ 0.11015 -0.43244 -0.2698   0.0414 ]
[-0.68538  0.       0.      -0.     ]
[ 0.      -0.68538 -0.29273 -0.05395]
[ 0.       1.       1.       1.     ]
[ 0.       0.       0.       0.     ]
[ 1.       0.       0.       0.     ]
```

This fact gives us the chance to better understand the difference between the two Jacobians.

Let's have a look at the orientation of the end-effector frame in the q_0 configuration, illustrated in Fig. 1

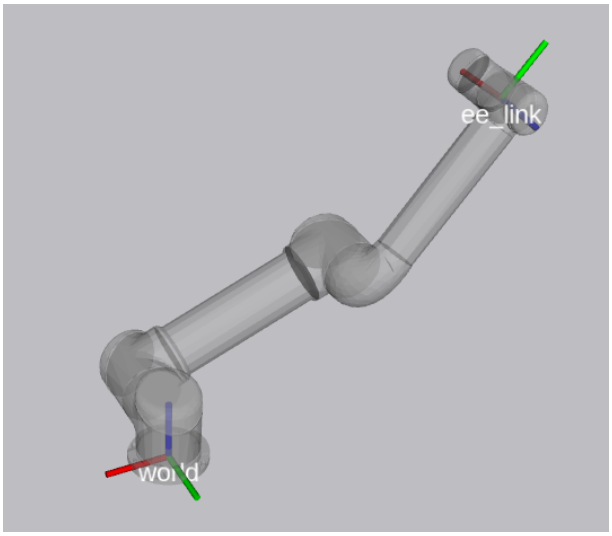

Figure 1: End-effector and world frames at the initial configuration $q_0 = (\pi, -\pi/8, -\pi/6, 0)$

In the case of the Geometric Jacobian, the row of zeros is the 3rd one. This is reasonable, because in the Geometric we have ω as output which is defined in in the world frame (Euclidean). Considering the kinematic structure of the robot, there is no motion of the joints that can create a change of of orientation for the end effector about the X axis (red arrow).

In the case of the Geometric Jacobian, the zero row is the 4th instead. This is also sensible if we consider that the Geometric Jacobian maps into Euler Rates, which are subsequent rotations of the end-effector axes. In this case, it is intuitive to see that there is no motion of the joints that can create a local rotation about the pitch (Y axis green) of the end-effector.

Is this just a case specific to the q_0 configuration, or there is something more structural?

Let's recompute the Jacobians after subtracting 90° to the `shoulder_pan_joint` (see Fig. 2):

[fig:init_conf90]

```
Geometric Jacobian:
[[-0.68538  0.       0.      -0.     ]
```

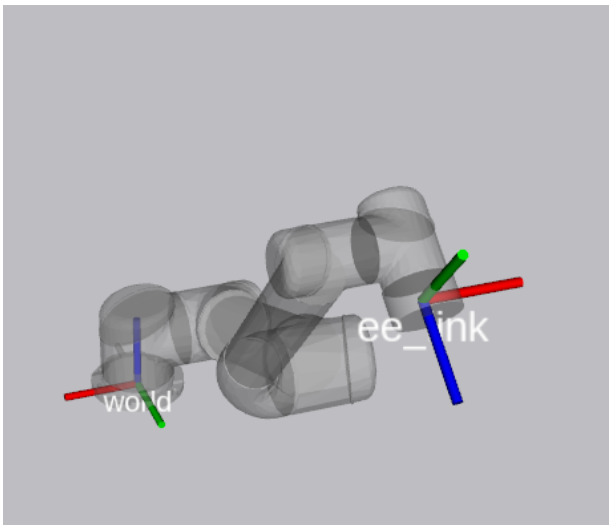

Figure 2: End-effector and world frames at the configuration $q = (\pi/2, -\pi/8, -\pi/6, 0)$

```
[-0.11015  0.43244  0.2698  -0.0414 ]
[ 0.      -0.68538 -0.29273 -0.05395]
[ 0.      -1.      -1.      -1.     ]
[ 0.       0.       0.       0.     ]
[ 1.       0.       0.       0.     ]]

Analytic Jacobian:
[[-0.68538  0.       0.      -0.     ]
[-0.11015  0.43244  0.2698  -0.0414 ]
[ 0.      -0.68538 -0.29273 -0.05395]
[ 0.       1.       1.       1.     ]
[ 0.       0.       0.       0.     ]
[ 1.       0.       0.       0.     ]]
```

In this case, the rows of zeros in both Jacobians is the 4-th one, which related to the fact that, in this configuration, no joint can change orientation of the end-effector frame w.r.t to the Y axis of the base link (in the case of the Geometric Jacobian). The linear part is also the same but different from the previous case. If we move only 45° the instead we get these results:

```
Geometric Jacobian:
[-0.40675 -0.30578 -0.19078  0.02927]
[-0.56253  0.30578  0.19078 -0.02927]
[ 0.      -0.68538 -0.29273 -0.05395]
[ 0.      -0.70711 -0.70711 -0.70711]
```

```
[ 0.      -0.70711 -0.70711 -0.70711]
[ 1.       0.       0.       0.     ]

Analytic Jacobian:
[-0.40675 -0.30578 -0.19078  0.02927]
[-0.56253  0.30578  0.19078 -0.02927]
[ 0.      -0.68538 -0.29273 -0.05395]
[ 0.       1.       1.       1.     ]
[ 0.      -0.      -0.      -0.     ]
[ 1.       0.       0.       0.    ]
```

Again, the linear part changed w.r.t the previous case, but while the Analytic Jacobian still maintain the 4th row of zeros, the Geometric Jacobian no longer has it. This highlights the fact that, being the Euler Angles subsequent rotations, and the Euler Rates the infinitesimal changes of these angles, they are expresssed about the end-effector axes, hence they have a "local" meaning. This is independent by the overall orientation of the arm w.r.t to the base link (dictated by the). Instead, the Geometric Jacobian (orientation part) always spits out the vector ω of angular velocity, which represents the rate of change of the orientation of the end-effector axes with respect to the base frame axes, hence it is affected by the `shoulder_pan_joint`.

2.4 Numerical inverse kinematics (lecture E4)

Now you have all the necessary ingredients to compute the manipulator's inverse kinematics. Since we have 4-DoF, the *analytical* solution of the inverse kinematics problems is not straightforward, so we suggest to take the iterative approach given by the algorithm described in Fig. 3.

To start simple, consider your task space variables to be the Cartesian space position (i.e., x, y, z) and the roll angle ψ of the end-effector. Why can't the full orientation be defined in the task? (hint: check which row in the Jacobians goes to zero). Therefore, you will need to adequately choose the rows of interest of the *analytical* Jacobian $\mathbf{J}_a$. Since we have 4 task variables and 4 DoFs then $\mathbf{J}_a$ is a square matrix and we expect $\mathbf{J}_a^T\mathbf{J}_a$ to be always invertible (out of singularities). After implementing an inverse kinematics function, try to find the corresponding joint angles for the task $p = \begin{bmatrix}-0.5 & -0.2 & 0.5 & \frac{\pi}{3}\end{bmatrix}^T$, where the first three components are the x, y, z Cartesian coordinates of the end-effector and the fourth component corresponds to the roll angle ψ of the end-effector frame. Set your hyper-parameters for the algorithm as follows: $\alpha = 1$ (Newton step size), $\epsilon = 1 \times 10^{-6}$ and $\beta = 0.5$ where β is the coefficient of step-size reduction in the line search, and ϵ is the termination tolerance. First, implement the algorithm without line search (i.e., with constant step size α=1) and then compare it with the adjustable step size. Is there any difference in the number of iterations? (you should notice that the line search might take a smaller number of iterations, 10 instead of 18). What is the influence of the initial condition? Check that

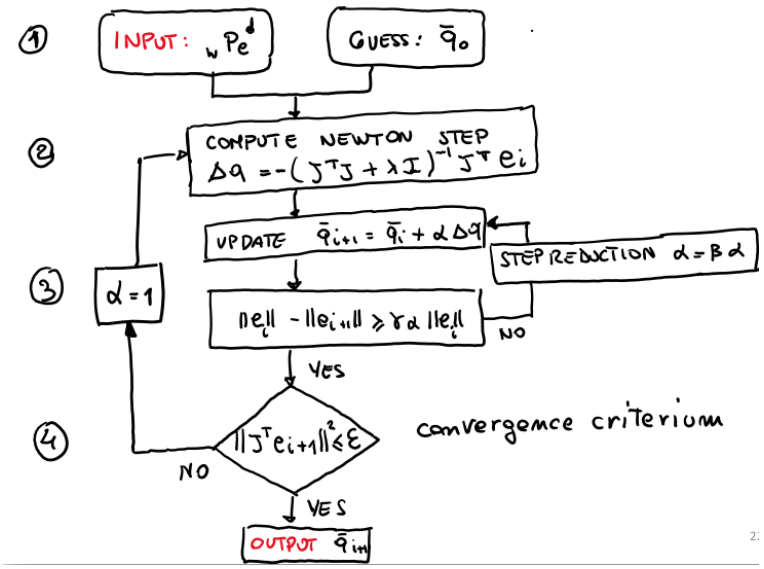

Figure 3: Numerical inverse kinematics algorithm or closed-loop inverse kinematics (CLIK)

the algorithm converges to different joint configurations depending on how you initialize it and that the number of iterations changes. Try to set it in order to converge to the elbow up or elbow down configuration. Find also an initial configuration that requires a higher number of iterations to converge.

Sanity check: Using the joint angles coming out of the numerical inverse kinematics, compute the direct kinematics and obtain the error between the asked task space position and the computed one using the joint angles obtained from the inverse kinematics and verify the error is close to zero.

Despite the correct result (no error at the end-effector), you might notice that the values of the joint variables q are very large. Why is this? You will have to implement a "wrapping" function to keep the joint angles between -2π and 2π.

Now, compare different outputs of the numerical inverse kinematics function using the line search algorithm but limiting the maximum number of iteration to 3, 4 and 5, how much is the difference between the final position and the target?

Figure [fig:ik] shows the position of the robot after 3, 4 and 5 iterations.

You can notice that the algorithm is converging very fast (the rate of convergence of Gauss-Newton method approaches the quadratic) and the robot is very close to the target already after 4 iterations.

Now, try to ask to reach a position of the end-effector *outside* of the workspace, for example, $p = \begin{bmatrix} -1.0 & -0.2 & 0.5 & \frac{\pi}{3} \end{bmatrix}^T$. Start without the line search, is the

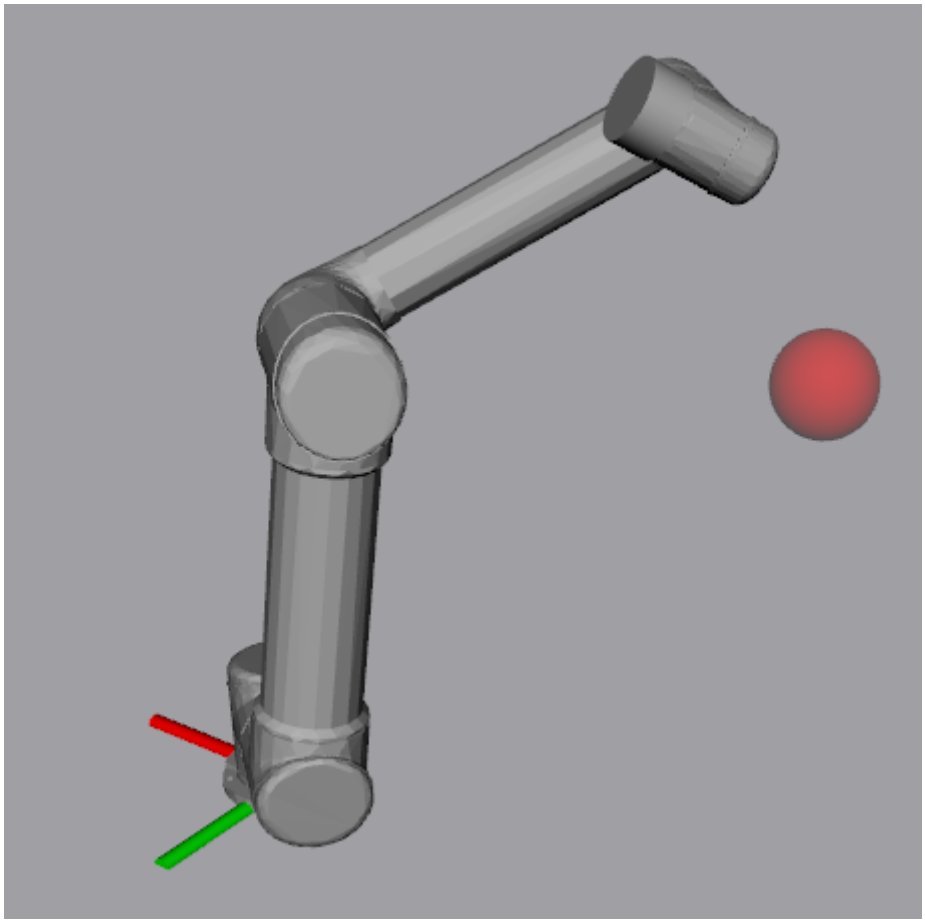

Figure 4: Position after 3 iterations

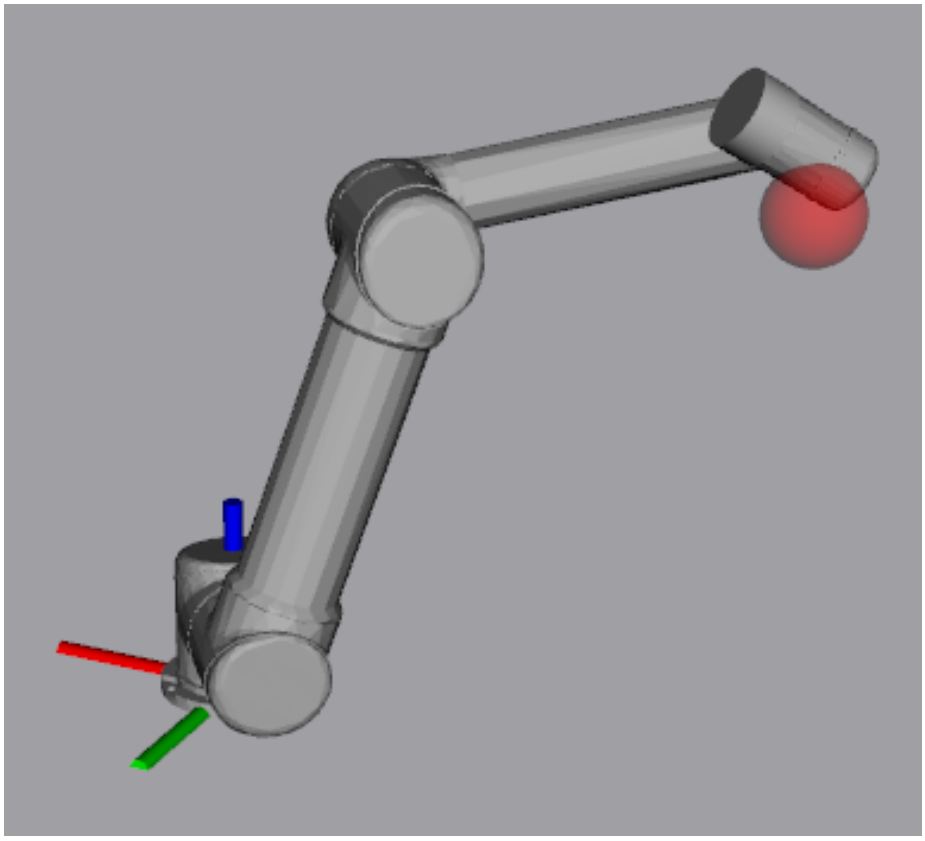

Figure 5: Position after 4 iterations

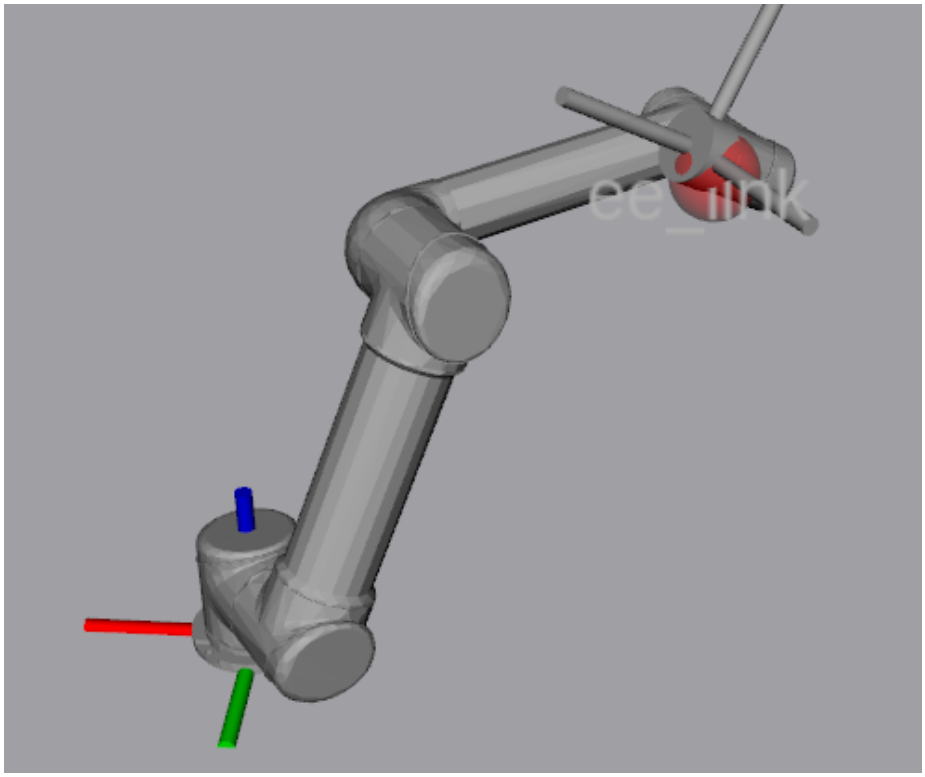

Figure 6: Position after 5 iterations

error going to zero? verify that the algorithm never converges (the maximum number of iterations is reached). If you enable the line search the robot tries to do "the best that he can" to reach the target, reducing the error, but still the maximum number of iteration is reached. Increasing the regularization term λ (e.g. to 0.01) you can see that the gradient goes under the tolerance ϵ before reaching the maximum value of iterations (and the error will be minimum).

A limitation of this approach is that joint limits can be considered only at the end. An improvement would be to set up a *constrained* optimization problem where we enforce the end-stop limits during the optimization.

2.5 Fifth order joint polynomial trajectory.

Now that we have a function that computes the inverse kinematics, we can use our initial joint position $\mathbf{q}_0$(defined in the file) and the final joint position vector $\mathbf{q}_f$ (computed with our numerical inverse kinematics function) to compute a trajectory and move from one point to the other. An easy way to do so is by using a fifth order *polynomials* parameterized by time to represent the position and its derivatives (i.e., velocity and acceleration) of a given joint with respect to time, i.e.:

$$\begin{aligned} q_i(t) &= a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \\ \dot{q}_i(t) &= a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4 \\ \ddot{q}_i(t) &= 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3 \end{aligned}$$

we can then define a set of *initial* and *final* conditions for the joint variables q_i, $\dot{q}_i(t)$ and $\ddot{q}_i(t)$ to define the behavior of the trajectory. Using this *boundary conditions* we can obtain the values of the coefficients a_i by solving a 6×6 linear system of equations (we have 6 equations and six unknowns). For the initial conditions we use the $\mathbf{q}_0$ and assume that the arm starts from a no-motion condition (i.e., $\dot{\mathbf{q}}_0 = \mathbf{0}$ and $\ddot{\mathbf{q}}_0 = \mathbf{0}$). For the final conditions we can use the joint values obtained by the inverse kinematics $\mathbf{q}_f$ and again assume that the robot will come to a full stop at the end of the motion (i.e., $\dot{\mathbf{q}}_f = \mathbf{0}$ and $\ddot{\mathbf{q}}_f = \mathbf{0}$). We initialize time $t_0 = 0$ and the final time t_f we are free to choose (the shorter the time, the faster the robot will move). Let's select $t_f = 3s$. With these considerations, replacing $t = 0$ and $t = t_f$ the equations for initial and final conditions become (per joint):

$$\begin{aligned} q_0 &= a_0 \\ 0 &= a_1 \\ 0 &= 2a_2 \\ q_f &= a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 + a_4 t_f^4 + a_5 t_f^5 \\ 0 &= a_1 + 2a_2 t_f + 3a_3 t_f^2 + 4a_4 t_f^3 + 5a_5 t_f^4 \\ 0 &= 2a_2 + 6a_3 t_f + 12a_4 t_f^2 + 20a_5 t_f^3 \end{aligned}$$

which can be rewritten in matrix form as:

$$\begin{bmatrix} q_0 \\ 0 \\ 0 \\ q_f \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 \\ 0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 0 & 0 & 2 & 6t_f & 12t_f^2 & 20t_f^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}$$

Then we can pre-multiply both sides of the equation by the inverse of the square (6X6) matrix and on the right-hand side of the equation obtain the values for the coefficients of the polynomials. With these coefficients we can generate the whole trajectory $q(t)$ starting for a time $t = 0$ (where $q = q_0$) up to $t = t_f$ (where $q = q_f$).

Implement this trajectory by modifying the function . Plot the trajectories by un-commenting the final lines of the file and see if you notice any problem. Hint: you can set a of 50 to better see the motion. You will see that the robot is doing very strange loops (too see them you need to restore λ to its default value). If you remove the *wrapping* feature you will get something even worse. This is because of the initialization that has a strong influence. Try to initialize the ik with q_0 instead than with q_i and you will see that the trajectory improves significantly. However, if you set a target that is more far away (e.g. $p_e^d = \begin{bmatrix} 0.5 & -0.2 & 0.5 \end{bmatrix}^T$) you will still see loops. Using this approach to generate the trajectory is simple, but has no guarantee on what is happening (e.g. where the end-effector will be) in between the initial and the final configurations. Namely, you only now for sure where the end-effector is at the beginning and the end of the motion, which in real-life scenarios can lead to potential collisions or unwanted paths.

Cartesian trajectory: The solution is to design a trajectory directly in Cartesian space via parametric curves. For instance you can design a trajectory of a line in 3D Cartesian space parameterized by time and, for each position given by the curve along the trajectory, you will need to compute the inverse kinematics to provide the joint values (hint: you need to do it also at the velocity and at the acceleration level). As initial guess at each time instant you can set the configuration at the previous sample. Compare the results of the trajectory with respect to the one designed in 2.5.

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:

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

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:

- (Main file to run the visualization and the exercises)
- (Initializations of variables and simulaion parameters)
- (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, . We will use the following equations from lecture F1.2. Refer to Figure 7 for variable definitions.

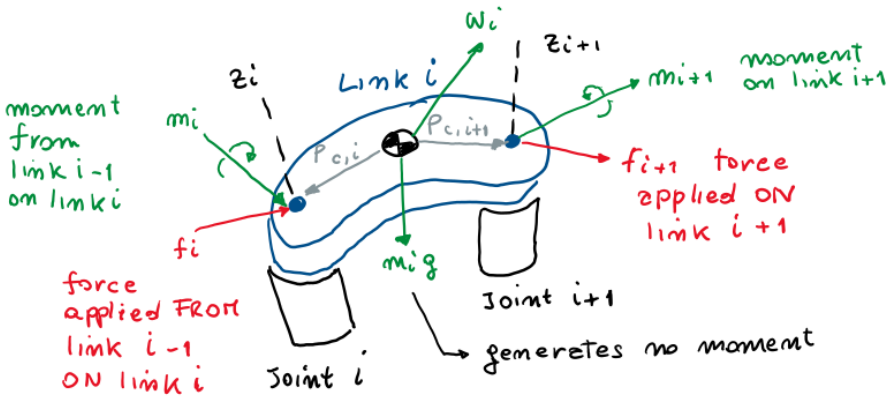

Figure 7: Sketch of a link i supported by joint i.

So, for the forward pass we have[^15]:

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

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

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

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 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., .

3.2 Compute each of 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 , the vector of Coriolis and centrifugal forces with and each of the columns of the joint space inertia matrix with , where 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 loop in the provided main file to run your visualization by using the following equations:

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

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: , which takes as only argument the vector of joint positions. To compute $\mathbf{M}$ use: [ˆ16]. In the case of vector $\mathbf{C}$, Pinocchio only provides the sum $\mathbf{h} = \mathbf{c} + \mathbf{g}$ through the function: [ˆ17]. 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 frames. Whereas in 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:

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

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.