Learning for Adaptive and Reactive Robot Control
Instructions for exercises of lecture 2

Professor: Aude Billard
Contact: aude.billard@epfl.ch

Installation

You can now download the `solutions` folder for last week from Moodle, unzip it and place it in the `lecture1-introduction` folder. Your `book-code` folder should have the following architecture :

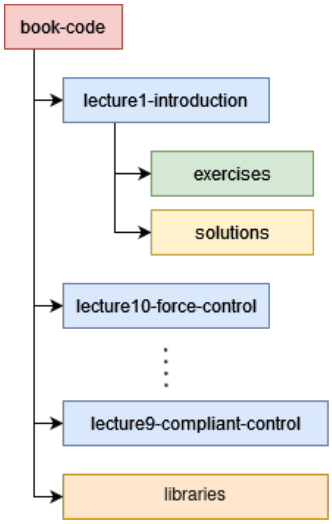

Figure 1: Workspace directory architecture

1 Exercise 1: Kinematically feasible trajectories

Book correspondence: page 40.

1.1 Goal:

The aim of this exercise is to familiarize the reader with dataset generation, using kinematic models and dynamic constraints. Specifically, in this exercise, you will code a program that allows to sample kinematically and dynamically feasible trajectories for a 4 degrees of freedom robot arm. These trajectories will then be used in the future practice sessions as demonstrations from which we will learn a stable dynamical system.

1.2 Instructions

1.2.1 STEP 1 OF THE EXERCISE:

1. Generate a joint space trajectory that starts from a random configuration, and reaches a prespecified target location in Cartesian space, following Algorithm 2.1, see below.

2. Repeat the previous step 10 times, each time by initializing the robot at different initial configurations. This will generate a dataset of trajectories. Save each trajectory. A trajectory consists of a series of position and velocities followed by the robot until it reaches the goal.

Algorithm 2.1 Generate Kinematically Feasible Data Trajectories

---

Initialization:
  $x(0)$ and x^* *Randomly define the initial robot's position and final target position.*
  $q(0) = F^{-1}(x(0))$ *Inverse kinematics for the initial joint position.*
  $t = 0$. *Initialize time.*
Main loop:
  While $\epsilon \leq \|F(q(t)) - x^*\|$:
    $\dot{q}(t+dt) = \arg\max_{\dot{q}} \|\dot{q}\|$. *Maximizing the velocity of the joints.*
  subject to:
    $\frac{\dot{q}}{\|\dot{q}\|} = J^+(q(t)) \frac{x^* - F(q(t))}{\|x^* - F(q(t))\|}$. Moving on the straight line toward the target at the task space.
    $\dot{q}_{min}[i] \leq \dot{q}[i] \leq \dot{q}_{max}[i] \;\; \forall i \in \{1, \ldots, D\}$. Kinematic feasibility at the velocity level.
    $q_{min} \leq q(t)[i] + \dot{q}[i]dt \leq q_{max}[i] \;\; \forall i \in \{1, \ldots, D\}$. Kinematic feasibility at the position level.
  $q(t+dt) = q(t) + \dot{q}(t+dt)\, dt$
  $t = t + dt$

$F(.)$ and $J(.)$ are the forward kinematics and the Jacobin matrices, respectively. $+$ is Moore–Penrose inverse.

Matlab steps: Open `chp2_algo1_inverse_kinematic.m`. You will have to implement the main loop of algorithm 2.1 at line 48 of the MATLAB file.
Try first to create a straight line between the initial configuration `q0` and the final position `targetPosition` without maximizing the velocity of the joints, nor checking for joint position and velocity limits.
To do this, you can use the Jacobian matrix $\mathbf{J}$ of the kinematic model, which relates the joint velocity $\dot{\mathbf{q}}$ to Cartesian velocity $\dot{\mathbf{x}}$:

$$\dot{\mathbf{x}} = \mathbf{J} * \dot{\mathbf{q}} \tag{1}$$

You can use the following MATLAB functions:

```
% Input 4x1 joint position vector q, output 3x1 Cartesian position vector:
position = robot.fastForwardKinematic(q);

% Input 4x1 joint position vector q, output 3x4 Jacobian matrix:
jacobian = robot.fastJacobian(q);
```

As the Jacobian is a 3-by-4 matrix (four joint velocity, three dimensions of Cartesian velocity), you cannot directly invert it. The simplest solution is to rely on MATLAB `pinv()` function (Moore-Penrose pseudo-inverse) to get the Moore-Penrose inverse $\mathbf{J}^+$ and solve this equation to get $\dot{\mathbf{q}}$. A better approach is to use the damped pseudo-inverse:

$$\mathbf{J}_d^+ = \mathbf{J}' * (\mathbf{J}\mathbf{J}' + \lambda^2 \mathbf{I})^{-1} \tag{2}$$

This method has the advantage of avoiding matrix inversion issues near singularities, which typically happen when the robot is fully extended, and thus loses a degree of freedom. You can tune this effect with the parameter λ: for values close to zero, the dampening effect is reduced, so the Jacobian inversion is more precise but less stable near singularities. Test these two inversion methods to see when the damped pseudo-inverse is the most useful (*hint*: try moving the `targetPosition`)

You can now compute an increment $\Delta\mathbf{q}$ that moves the end-effector in the direction of your target. Repeat this step until you are sufficiently close to the target (Cartesian distance less than `toleranceDistance` variable). You can store all the joint velocities $\dot{\mathbf{q}}$ and joint positions $\mathbf{q}$ in the variable `trajectory` as a 8-by-N array.

$$\texttt{trajectory} = \begin{bmatrix} \mathbf{q}_0 & \cdots & \mathbf{q}_N \\ \dot{\mathbf{q}}_0 & \cdots & \dot{\mathbf{q}}_N \end{bmatrix} \tag{3}$$

with $\mathbf{q}_N$ and $\dot{\mathbf{q}}_N$ the joint position and velocity respectively at `targetPosition`.
WARNING: Don't fill the 3D array `trajectories`. It will be automatically filled afterward by resampling `nPoints` from your array `trajectory`.
Once you can successfully generate one straight Cartesian trajectory, you can modify the variable `nTraj` to generate a batch of `nTraj` trajectories.

As a final improvement, you can scale each joint velocity vector by a scalar to ensure that the maximum joint speed is just at the limit `maxJointSpeed`:

$$||\mathbf{q}||_\infty = max(||\dot{q}_1||, ||\dot{q}_2||, ||\dot{q}_3||, ||\dot{q}_4||) = \texttt{maxJointSpeed} \tag{4}$$

which will ensure that all joint speed $\dot{q}_i$ are bounded:

$$-\texttt{maxJointSpeed} \leq \dot{q}_i \leq \texttt{maxJointSpeed} \tag{5}$$

Answer the following questions:

- What is the effect of using different Jacobian inversion methods ?
- Can you think of other features that could be added to this algorithm ?

1.2.2 STEP 2 OF THE EXERCISE:

1. Generate joint space trajectories using optimal control.
2. Compare the solutions with the ones from the closed-loop inverse kinematic algorithm. What are the advantages and disadvantages of each ?

Matlab Steps: Open `chp2_algo1_optimal_control.m`. Add the cost functions you derived during exercise 1 at the bottom of the file. You can choose which cost function the MPC will use at line 28.

References

[1] Aude Billard, Sina Mirrazavi, and Nadia Figueroa. *Learning for Adaptive and Reactive Robot Control: A Dynamical Systems Approach*. MIT press, 2022.