Imitation Learning for Robotics : Assignment 1 (Part 2)

Professor: Ravi Prakash

Contact: ravipr@iisc.ac.in

(1) Theoretical exercise

Optimal trajectories formulation

Given a 2-joint planar robotic arm with unit link length, we are interested in its end-effector position along X direction, i.e. with forward kinematics:

$$x(t) = h(\mathbf{q}(t))$$

A trajectory is a path with $N+1$ waypoints $\{(q_1(0), q_2(0)), (q_1(1), q_2(1)), \ldots, (q_1(N), q_2(N))\}$, and a time-indexing $\{0, t(1), \ldots, t(N)\}$ that specifies the time at which the robot arrives the corresponding waypoint.

Write the optimisation problem to get the trajectories with $N+1$ waypoints and uniform time indexing, i.e. $\delta t = t(n) - t(n-1) = t(N)/N, \forall n \in [1, \ldots, N]$ that moves the robot from initial configuration x_0 given by $(q_1(0), q_2(0))$ to a desired position x^* subject to joint maximum increment $\delta q_{\max}$ per second with following desired properties:

(1a) Minimizing the time to reach x^*

(1b) Following the shortest path in Cartesian space

(1c) Following the shortest path in joint space

Programming exercises

Preliminaries: Instructions for proper installation of Python libraries

For this first session, you will need to use the provided Jupyter Notebook. Please follow the installation instructions to install the required libraries.

To make sure everything works properly, you will need Python 3.10 or newer. You will need to install the following libraries using pip:

- `numpy`
- `scipy`

- `pybullet`
- `matplotlib`
- `notebook` (or `jupyterlab`)

Installation command:

```
pip install numpy scipy pybullet matplotlib notebook
```

Programming Exercise 1.1 Closed Form Trajectory

1. **Compute a closed-form time-dependent trajectory generator based on a third-order polynomial.**

You can add your implementation in the marked section of **Cell 2 (Exercise 1.1)** in the notebook. You have to manually create a trajectory in position (Cartesian space) that starts at `p0` and ends at `pT`, based on a third-order polynomial for each dimension (x, y, z).

Programming Exercise 1.2: Computing Optimal Trajectories

The aim of this exercise is to familiarize you with basic optimization methods for the motion of a manipulator. Go to **Cell 3 (Exercise 1.2)** in the notebook. The code sets up a **7-DOF Franka Emika Panda** robot in the PyBullet simulator. Edit the code to do the following:

1. Write the optimization problems for this redundant manipulator moving in three dimensions. Assume that the joints are attached serially. The target position is fixed in 3D space.
   (a) Minimizing the time to reach x^*
   (b) Following the shortest path in Cartesian space
   (c) Following the shortest path in joint space

We use the `scipy.optimize.minimize` function to perform the numerical optimization. You only have to define a cost function corresponding to each problem of the exercise. In Python, the optimization variable `x` is passed as a 1D (flattened) array containing the sequence of joint configurations followed by the final time T_f. You will need to "unpack" this array to use it.

$$\mathbf{x}_{\text{flat}} = [\underbrace{q_0^1, q_0^2, \ldots, q_0^7}_{\text{Step 0}}, \ldots, \underbrace{q_N^1, q_N^2, \ldots, q_N^7}_{\text{Step N}}, T_f]$$

The helper function `unpack(x)` in the code will reshape this for you.

Here $\mathbf{q} = [q^1, \ldots, q^7]$ represents the joint positions of the robot. T_f is the final time of the trajectory. This trajectory is made of N points.

The three functions to modify are cost_min_time(), cost_min_task_dist() and cost_min_joint_dist(). You can play with different cost functions by modifying the `minimize` call.
You also have to fill in the velocity constraint and uncomment it in the optimization call. WARNING Some objective functions might take a few seconds to be solved.

You can use the helper wrapper to get the end-effector position for a given configuration:

```
# Input: 7-element joint configuration array
# Output: 3-element [x, y, z] position array
pos = get_ee_pos_wrapper(q)
```

Programming Exercise 1.3: Disturbed trajectory

The aim of this programming exercise is to determine how to recompute a path under disturbances with the optimization techniques seen in programming exercise 1.2. Go to **Cell 4 (Exercise 1.3)**.

1. **Initialize the end-effector at one of the locations chosen in the previous programming exercise and generate a path following optimization (c).**
   The code generates an initial trajectory using a simple cost function.

2. **Generate a disturbance midpath by suddenly displacing one of the joints of the robot.**
   Choose an index mid-trajectory (mid_idx), and apply the disturbance at this time by modifying the joint configuration at this index (e.g., adding 0.8 rad to the elbow joint). The updated joint configuration will be used as the initial configuration for the next optimization.

3. **Redo the optimization to generate a new path to the target using the residual time to the target.**
   A new trajectory is generated from the disturbed configuration. Create the complete trajectory by stitching the two solutions (the initial path up to the disturbance, and the new replanned path).

The solution returns an array of joint configurations which you should stack to create the full path.

Exercise 1.4: Kinematically feasible trajectories

The aim of this exercise is to familiarize you 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 **7 degrees of freedom** robot arm.

Algorithm 1 Generate Kinematically Feasible Data Trajectories

a) Initialization

$x(0)$ and x^*: Randomly define the initial robot's position and final target.

$q(0)$ is the known start configuration.

$t = 0$, Initialize time.

b) Main Loop

while $\epsilon \leq ||F(q(t)) - x^*||$ **do**

Compute error $e = x^* - F(q(t))$

Compute Jacobian $J(q)$

Compute Damped Pseudo-Inverse $J^\dagger = J^T(JJ^T + \lambda^2 I)^{-1}$

Compute $\dot{q} = J^\dagger e$

Update $q(t + dt) = q(t) + \dot{q}dt$

$t = t + dt$

end while

Note: $F(\cdot)$ and $J(\cdot)$ are the forward kinematics and the Jacobian matrices, respectively.

Instructions

1. STEP 1 OF THE EXERCISE:

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

Programming steps: Go to **Cell 5 (Exercise 1.4)**. You will have to implement the main loop of Algorithm 1.

Try first to create a straight line between the initial configuration q_0 and the final position `target_pos`. To do this, you can use the Jacobian matrix **J** of the kinematic model.

You can use the following methods from the provided `sim` object:

```
# Get current end-effector position
pos, orn = sim.get_ee_pose()
# Get Jacobian (Linear and Angular). Use J_lin (3x7)
J_lin, J_ang = sim.get_jacobian(q)
```

As the Jacobian is a 3-by-7 matrix (3 dimensions of Cartesian velocity vs 7 joint velocities), you cannot directly invert it. There are two common methods to solve this:

1. Moore-Penrose Pseudo-Inverse: The simplest solution is to use the standard pseudo-inverse $J^\dagger$:

$$\mathbf{J}^\dagger = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1} \tag{1}$$

2. Damped Pseudo-Inverse: A more robust approach is to use the damped pseudo-inverse, which avoids matrix inversion issues near singularities (where the robot loses a degree of freedom):

$$\mathbf{J}_d^\dagger = \mathbf{J}^T \left(\mathbf{J}\mathbf{J}^T + \lambda^2 \mathbf{I} \right)^{-1} \tag{2}$$

You can tune the stability with the parameter λ.

Task: Test these two inversion methods in your code (hint: try moving the `target_pos` to different locations or near singularities).

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 `tolerance` variable).

(2) Answer the following questions:

(a) What is the effect of using different Jacobian inversion methods (e.g. Pseudo-inverse vs Damped)?

(b) Can you think of other features that could be added to this algorithm (e.g. null-space control)?

(c) Compare the solutions with the ones from the optimal control (Exercise 1.2). What are the advantages and disadvantages of each?

Submission Instructions: Submit the completed Jupyter Notebook (.ipynb) and a separate PDF with the answers to the theory questions.

Adapted from *Learning Adaptive and Reactive Control for Robots*, Aude Billard, LASA, EPFL
