Assignment 01: Simulation and control of a DC Motor with Coulomb and viscous friction

Andrea Del Prete*- andrea.delprete@unitn.it

March 23, 2020

1 Description

The goals of this assignment are:

- implementing a class to simulate a DC motor with a speed reducer subject to Coulomb and viscous friction
- implementing a PID controller with Coulomb friction compensation
- testing the performance of the controller under various conditions

2 Submission procedure

You are encouraged to work on the assignments in groups of 2 people. However, also working alone is acceptable. Groups of more than 2 people are not allowed. The mark of each assignment contributes to 10% of you final mark for the class (i.e. 3 points out of 30).

When you are done with the assignment, please submit a single compressed file (e.g., zip) containing:

- A pdf file with the answers to the questions and the names of the people of the group; you are encouraged to include plots and/or numerical values obtained through simulations to support your answers.
- The complete *arc* folder containing all the python code that you developed.

If you are working in a group (i.e., 2 people) only one of you has to submit. Submitting the pdf file without the code is not allowed and would result in zero points. Your code should be consistent with your answers (i.e. it should be possible to produce the results that motivated your answers using the code that you submitted). If your code does not even run then your mark will be zero, so make sure to submit a correct code.

3 DC Motor Simulator

The template code for this part of the assignment is located in

`code_template/arc/01_actuators/dc_motor_w_gear.py`

*Advanced Optimization-based Robot Control, Industrial Engineering Department, University of Trento.

This file contains both the class `MotorWGear` and a `__main__` function to test the class. The only part that needs to be implemented is the method `simulate` of `MotorWGear`, which takes as input the motor current `i` and the simulation method (which can be either 'time-stepping' or 'standard').

The recommended method is 'time-stepping', but if you cannot do it then you could also use the 'standard' method (i.e. explicit Euler). Implementing the 'standard' method rather than 'time-stepping' is simpler, but it gives you less points. The dynamics of the DC motor with speed reducer is:

$$\begin{aligned} V &= Ri + K_b \dot{q}_m \\ \tau_m &= K_b i \\ (I_j + N^2 I_m)\ddot{q}_j + (b_j + N^2 b_m)\dot{q}_j + N\tau_{cm} + \tau_{cg} &= N\tau_m \end{aligned} \tag{1}$$

where τ_{cm} is the motor Coulomb friction, τ_{cg} is the gear Coulomb friction, and the other terms have already been defined in class. Both friction terms are bounded:

$$\begin{aligned} |\tau_{cm}| &\leq \tau_{cm}^{max} \\ |\tau_{cg}| &\leq \tau_{cg}^{max} \end{aligned} \tag{2}$$

Note that τ_{cm}^{max} and τ_{cg}^{max} are called `tau_coulomb` and `tau_coulomb_gear` in the code, and they are specified in the motor parameters (see function `get_motor_parameters`). For applying time-stepping, we can see the sum of the two Coulomb friction torques as a single term

$$\tau_c \triangleq N\tau_{cm} + \tau_{cg}, \tag{3}$$

which is bounded by:

$$|\tau_c| \leq N\tau_{cm}^{max} + \tau_{cg}^{max} \tag{4}$$

The motor dynamics can then be rewritten as:

$$I\ddot{q}_j + b\dot{q}_j + \tau_c = NK_b i, \tag{5}$$

where:

- $I \triangleq (I_j + N^2 I_m)$
- $b \triangleq (b_j + N^2 b_m)$

Using time-stepping, we can represent the acceleration as:

$$\ddot{q}_j = \frac{\dot{q}'_j - \dot{q}_j}{\Delta t} \tag{6}$$

where $\dot{q}'_j$ is the next joint velocity (i.e. the joint velocity at the end of the current time step). Using this expression we can find the Coulomb friction torque τ_c as the minimizer of $||\dot{q}'_j||^2$.

4 Motor Control

The template code for this part of the assignment is located in

`code_template/arc/01_actuators/friction_compensation.py`

This file contains both the class `PositionControl` and a `__main__` function to test the class. The only part that needs to be implemented is the method `compute` of `PositionControl`, which takes as input the joint state and the reference joint state (state = position and velocity).

The control law to implement is a standard PID plus a friction compensation term:

$$u(t) = k_p(q_{ref} - q) + k_d(\dot{q}_{ref} - \dot{q}) + k_i \int_0^t (q_{ref}(s) - q(s))\mathrm{d}s + u_{friction} \tag{7}$$

We want to test two strategies for Coulomb friction compensation. The first one is to use the sign function:

$$u_{friction} = \mathrm{sign}(\dot{q}) i_c \tag{8}$$

where i_c is the motor current corresponding to the total Coulomb friction torque of our actuator (i.e. motor + gear). The second strategy is to use the hyperbolic tangent:

$$u_{friction} = \tanh(k_{tanh}\dot{q}) i_c \tag{9}$$

where k_{tanh} is a user-specified parameter defining how closely the tanh function should approximate the sign function.

5 Tests

After you have implemented the motor simulator and the controller, you can start testing them. The code for running the tests is already written in the `__main__` function in `friction_compensation.py`. Please do not change any parameter (such as gains and time step) in the main function. This code runs three simulations, which differ only for the friction compensation strategy:

- using the sign function
- using the tanh function with $k_{tanh} = 0.01$
- using the tanh function with $k_{tanh} = 2$

For each simulation the script plots the resulting trajectories and prints the average tracking error, which gives us a measurement of how well the controller was able to track the reference trajectory (i.e. a sinusoid). To make the simulation more realistic, some random noise is added to the velocity estimation before feeding it to the controller.

Try to answer the following questions:

1. Which friction compensation strategy resulted in the smallest tracking error? Why? How much was it?
2. Which friction compensation strategy resulted in the largest tracking error? Why? How much was it?
3. What is the main issue with using the sign function for friction compensation?
4. Overall, which one of the three strategies would you implement on a real actuator? Why?
5. (Optional) Discuss ways for improving the tracking performance (e.g., increasing gains), highlighting pros and cons.