CSE 490P Homework:
Simple control methods applied to 2-link arm

Emo Todorov
University of Washington

Due Jan 28 (not Jan 23 as previously advertised)

1 Dynamical system

The goal of this exercise is to implement and test the simple control methods discussed in class on a model of a 2-link arm moving the in vertical plane (with gravity). The equations of motion are in the general form

$$\dot{v} = M(q)^{-1}(u - c(q, v))$$

where q, v are the joint angle and velocity vectors (both 2D, since the arm has two joints – shoulder and elbow), $M(q)$ is the 2-by-2 joint-space inertia matrix, $c(q, v)$ is the 2D vector of Coriolis, centripetal and gravitational forces, and u is the 2D vector of joint torques that your controller can apply. The Cartesian position of the hand/end-effector y is given by the forward kinematics $y = f(q)$. The functions $M(q), c(q, v), f(q)$ are given in the attached MATLAB file arm2.m. To obtain the Jacobian $J(q) = \frac{\partial f}{\partial q}$ you should compute the derivatives of $f(q)$ analytically.

2 Control and numerical integration

Suppose you have implemented a controller in the general form

$$u = g(q, v; y^*)$$

The function g somehow "decides" what controls u to generate given the current position and velocity, and also the spatial target y^*. Note that y^* is not part of the dynamics; instead it is a 2D vector provided externally by the user. In practice y^* will change occasionally (when the user wants the

hand to move to a new location) but the controller has no way of knowing when that will happen, so it treats it as a known constant.

Once a controller is specified, we have an autonomous dynamical system

$$\begin{aligned} \dot{q} &= v \\ \dot{v} &= M(q)^{-1}\left(g(q, v; y^*) - c(q, v)\right) \end{aligned}$$

In order to integrate this system forward in time (i.e. simulate what the arm will do), we can discretize the time axis at small intervals h, and implement discrete-time semi-implicit Euler integration as

$$\begin{aligned} q(t+1) &= q(t) + hv(t+1) \\ v(t+1) &= v(t) + hM(q(t))^{-1}\left(g(q(t), v(t); y^*) - c(q(t), v(t))\right) \end{aligned}$$

The second equation should be evaluated first, because the update for $q(t+1)$ needs $v(t+1)$. This is what "semi-implicit" means. We could also use the explicit integrator $q(t+1) = q(t) + hv(t)$ but it tends to be less stable. The time step h can be 0.001 seconds (i.e. 1 msec). Note that you have to use consistent units: meters, seconds, Newtons.

3 Controller A: Push towards the goal

The first control method is based on the idea of pushing the hand towards the target. It has the form

$$u = k_1 J(q)^T (y^* - f(q)) - k_2 v$$

Here k_1, k_2 are positive constants that you have to adjust manually to make the system work well. The first term acts like a virtual spring pulling the hand towards the target. The stiffness of the spring is k_1. The second term is damping, which is needed to avoid oscillations at the target.

4 Controller B: Proportional-derivative control

The second control method relies on making a plan for the movement, and then executing the plan. These two steps are as follows.

4.1 Planning

We will make a plan in joint space. First we have to transform the desired hand location y^* to a corresponding joint configuration q^* such that

$y^* = f(q^*)$. In other words, we need to invert the forward kinematics function f and compute the inverse kinematics. This comes down to solving a nonlinear system of equations, which you can do numerically with the `fsolve` command in MATLAB.

Now that we have a target q^* in joint space, we can connect the initial configuration and the target with a straight line. This gives us a geometric path, but does not tell us how fast we should move at each point along the path (i.e. we do not yet have a speed profile). To design a speed profile we choose some bell-shaped function of time, such as the Gaussian

$$N(t; m, s) = \exp\left(-0.5(t-m)^2/s^2\right)$$

with mean m and covariance s. We want to generate a speed profile of desired duration T (where the choice of T is up to us), so the peak/mean should occur in the middle of the time interval: $m = T/2$. The covariance s should be around $T/4$. We interpret $N(t; m, s)$ as being proportional to the speed at time t. One complication is that at $t = 0$ the above function is not exactly 0, but we can subtract this bias and define the speed as

$$\text{speed}(t) = d(N(t; m, s) - N(0; m, s))$$

The scaling constant d is needed to make sure that the integral of this speed profile matches the length of the trajectory (i.e. the distance between the initial configuration and the target in joint space). Then we construct a desired trajectory, i.e. a sequence of positions $q^*(1), q^*(2), \ldots$ that lie on a straight line and are spaced apart so that the effective speed at each point, i.e. the quantity $v^*(t) = (q^*(t+1) - q^*(t))/h$, is proportional to the above speed function. Be careful with the computations at the two endpoints.

4.2 Execution

Once we have a plan, we can implement a proportional-derivative (PD) controller as

$$u(t) = k(q^*(t) - q(t)) + b(v^*(t) - v(t))$$

where k, b are positive coefficients that need to be adjusted to make the system work well.

5 What to do in the assignment

1. Write MATLAB functions for the forward kinematics (already given to you, just copy from arm2.m), inverse kinematics (using fsolve), forward dynamics, and numerical integration.

2. Choose initial joint configuration q_0 and Cartesian target position y^*, and make sure they do not coincide. You should repeat steps 2–4 for multiple pairs of initial and target positions, to make sure your code always works.

3. Implement the two controllers. In A you need to compute the Jacobian matrix, in B you need to compute the desired/planned trajectory. After than you have simple formulas for each controller.

4. Plug each controller in the simulation, integrate the dynamics and see what it does. Adjust the two parameters of each controller to make it work well.

5. Make plots of the joint angles over time, to illustrate how well your controllers work.