Simulation, Control, and Estimation for an Inverted Pendulum

We consider an inverted pendulum consisting of a weight attached to a disk, which is actuated by a DC motor and rotates in a vertical plane, see Figure 1. The inverted pendulum is interesting due to its nonlinear dynamics, and is a commonly used example in the control field.

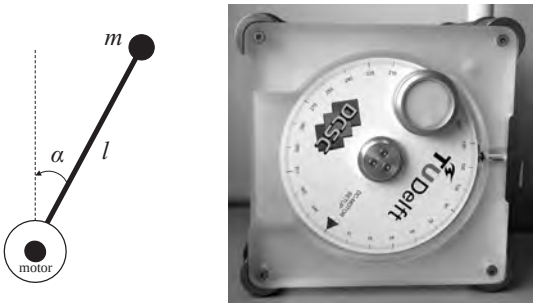

Figure 1: Inverted pendulum schematic (left) and the real system (right).

The continuous-time model of the pendulum dynamics is:

$$\ddot{\alpha} = 1/J \cdot [mgl \sin(\alpha) - b\dot{\alpha} - K^2\dot{\alpha}/R + Ku/R] \tag{1}$$

where $J = 1.91 \cdot 10^{-4}\,\mathrm{kgm}^2$, $m = 0.055\,\mathrm{kg}$, $g = 9.81\,\mathrm{m/s}^2$, $l = 0.042\,\mathrm{m}$, $b = 3 \cdot 10^{-6}\,\mathrm{Nms/rad}$, $K = 0.0536\,\mathrm{Nm/A}$, $R = 9.5\,\Omega$. The state vector is $x = [\alpha, v]^T$ where α is the angle and $v = \dot{\alpha}$ is the angular velocity, with the convention that $\alpha = 0$ represents the pointing-up position. The control action u is the voltage, and the motor supports a range of $[-10, 10]\,\mathrm{V}$. The goal is to stabilize the pendulum in the unstable equilibrium $x_{eq} = [0, 0]^T$ (pointing up).

The solution to the assignment consists of the resulting code and Simulink schemes, in a ZIP file; as well as a report in PDF. In your report, briefly describe the problems you solved, any significant choices you made during the implementation, outline your solution, and – most importantly – include the *results* you obtained (e.g., representative graphs of system trajectories) and *discuss* these results. Do not explain your code (or schemes) line by line, as that is not useful. Including literature research, the assignment is intended to take on the order of 20h of work, depending on your experience with Matlab, Simulink, and the methods involved.

Part 1: Modeling and Simulation

First, implement the original, nonlinear and continuous-time model of the pendulum (1) in Simulink. The model should take u as input, and produce the states α and v at the output. It should be possible to easily set (e.g. via Constant blocks) the initial conditions for the angle and angular velocity, and to observe the resulting state trajectories (via Scope, or export to workspace followed by plotting). Simulate trajectories with the model in the following conditions:

- $x(0) = [0.1, 0]^T$, $u(t) = 0$.
- $x(0) = [0, 0]^T$, $u(t)$ a random staircase sequence.
- $x(0) = [\pi, 0]^T$, $u(t)$ a similar random staircase sequence as above.

Think about how the trajectories should behave before running the simulation, and verify if your hypothesis is correct after running it. Debug the model to be sure that it is correct, as it will

form the basis of the other experiments. Hint: to implement the righthand side of (1), use a "user defined function" (Fcn) block.

Second, linearize the model analytically, on paper, around the unstable equilibrium $x_{eq} = [0, 0]^T, u_{eq} = 0$. Hint: to make things easier, linearize the differential equation (1) first, and only then rewrite the dynamics in the state space form. This should result in a continuous-time linear model of the form:

$$\dot{x} = A_c x + B_c u \tag{2}$$

Finally, discretize the linear continuous-time model with a sampling time of $T_s = 0.01\,\text{s}$, and using Euler discretization (and **not** c2d in Matlab, because that method will not preserve the physical meaning of the state variables!). You should end up with a discrete-time linear model of the form:

$$x_{k+1} = A_d x_k + B_d u_k \tag{3}$$

where $k = 0, 1, 2, \ldots$ is the current time step and $x_k = [\alpha_k, v_k]^T$.

Implement in Matlab code, without using Simulink this time, a simulator function for model (3) which takes as input the initial state x_0, an array $\mathbf{u}_N = (u_0, u_1, \ldots, u_{N-1})$ of N consecutive inputs, and produces the resulting state trajectories at the output. Use this function to simulate the linearized model with zero input and $x_0 = [0.1, 0]$. Compare the results with those obtained with the nonlinear system, at the first bullet point above. Do the results match your expectations? Explain.

Part 2: Control

Our goal here will be to control the pendulum so that it is maintained pointed upwards, in a range where the linearization stays valid. Consider the linearized continuous-time system (2), for which a state feedback control law is of the form:

$$u = -Fx \tag{4}$$

where the control gain F is computed such that the closed-loop system

$$\dot{x} = (A_c - B_c F)x \tag{5}$$

will be asymptotically stable. This can be achieved by *pole placement*, i.e., placing all the eigenvalues of the closed-loop system in the left-hand side of the complex plane, see Matlab function `place`.

After choosing suitable values for the closed-loop poles, compute the controller gain. To test that the controller is working correctly, i.e., that it stabilizes the linear system, implement system (2) together with the control law (4) in Simulink. Hint: use the state-space block, and set the output matrix `C` to identity so that the system outputs both states, which can then be seen on a Scope. Simulate the closed-loop system for different initial conditions. Once you have determined that the controller stabilizes the linear system, test the same controller on your Simulink implementation of the nonlinear system for several initial conditions in the range $0, \ldots, \pi$. Comparing to the linearized system, what happens as the initial conditions get further away from the linearization point (e.g., when they get close to π)?

Part 3: State Estimation

The real system only has a sensor for the angle (an encoder), and the angular velocity is not measurable. So, for example, the state feedback control above would not be directly implementable. Therefore, we will design and use an observer (state estimator) to recover this velocity from the angle measurements. In this part we will only use the linearized dynamics.

Consider the linear discrete-time model you obtained above together with the measurement

$$y_k = C_d x_k \tag{6}$$

Since only the angle is measured, $y_k = \alpha_k$ and so $C_d = [1,\ 0]$. The velocity needs to be estimated based on the system dynamics and this measurement. The dynamics of the observer are of the form

$$\begin{aligned} \hat{x}_{k+1} &= A_d\hat{x}_k + B_d u_k + L(y_k - \hat{y}_k) \\ \hat{y}_k &= C_d\hat{x}_k \end{aligned} \tag{7}$$

where the observer gain L is computed such that the estimation error dynamics

$$e_{k+1} = x_{k+1} - \hat{x}_{k+1} = (A_d - LC_d)e_k \tag{8}$$

is asymptotically stable. This is called a Luenberger observer.

Similarly to controller design, the gain L can be designed by suitably placing the poles of the error system. Comparing (8) to (5), one can observe that in both cases a matrix has to be computed, but the place of the matrix in the two equations is different. However, if one considers $(A_d - LC_d)' = A_d' - L'C_d'$ then this has the same form as (5). Therefore, for the transposed equation, one can use the same pole placement technique as for controller design, and the observer gain will be the transpose of the resulting matrix. Remember that this is a discrete-time system, and thus for stability the poles should be inside the unit circle!

After choosing suitable values for the poles, compute the observer gain. First test that the observer correctly estimate the states of the linearized discrete-time system for *zero input* and different initial conditions. Note that the "true" and estimated initial conditions should be different, since e.g. there is no practical way to know the initial angular velocity. Run the estimator also for the *controlled* system, where the inputs are computed with the controller found above: $u_k = Fx_k$ (even though this controller was designed with the continuous-time system in mind, it should work since the sampling time is small, which means that the discretized system approximates well the continuous-time one).