EE106A Discussion 10: Control

1 Control

Now that we've learned how to represent the dynamics of systems we can construct controllers to make these systems follow joint angle trajectories. First we'll examine the PID controller, a *model-free* controller that is ubiquitous in industry and often used in research. We'll then examine how we can use the model information in our dynamics to augment this controller.

1.1 What is control?

Say that we have some dynamic system (differential equation)

$$\dot{x} = f(x) + g(x)u$$

The derivative of your system state $\dot{x}$ is some affine function of your system state x and a control input u. This control input could be a velocity (like you saw in lab 7 or lab 4), a torque (like you saw in class), a steering wheel angle, or essentially anything else. You can model traffic patterns as control systems with your traffic lights as inputs, or the economy as a control system with interest rates as inputs. The goal of control is to find some policy to set u such that the state x tracks some desired reference input x_d. Generally this policy will use the system's current state to determine u. This is called feedback.

$$u = K(x)$$

While controls can be used on a wide variety of systems, the rest of this document will assume that we're controlling a mechanical system and that our input is a vector of forces and/or torques. Thus the systems we'll be controlling will look like this:

$$\ddot{x} = f(x, \dot{x}) + M^{-1}u$$

2 PID Control

While much of control theory is focused on finding a feedback policy that will provably track desired trajectories, meet certain robustness criteria, or will result in behavior that optimizes some value function, PID control is an intuitive model-free method meant to be used when you don't have a good model of your system. It's relatively easy to tune and can be quite robust, but unless you have a system model, you can't prove anything about it. If you want to track a desired trajectory $[x_d(t), \dot{x}_d(t)]$, your control input should be

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \dot{e}(t)$$

Where the state error is defined as $e(t) = x_d(t) - x(t)$ and the velocity error is defined as $\dot{e}(t) = \dot{x}_d(t) - \dot{x}(t)$. These error terms encode how far away the system state is from the desired trajectory at any given time. K_p, K_i, and K_d are called your *control gains* and are each $n \times n$ diagonal matrices

of weights. By varying these gains, you can change how the controller responds to the various error terms. This is called *tuning* the controller.

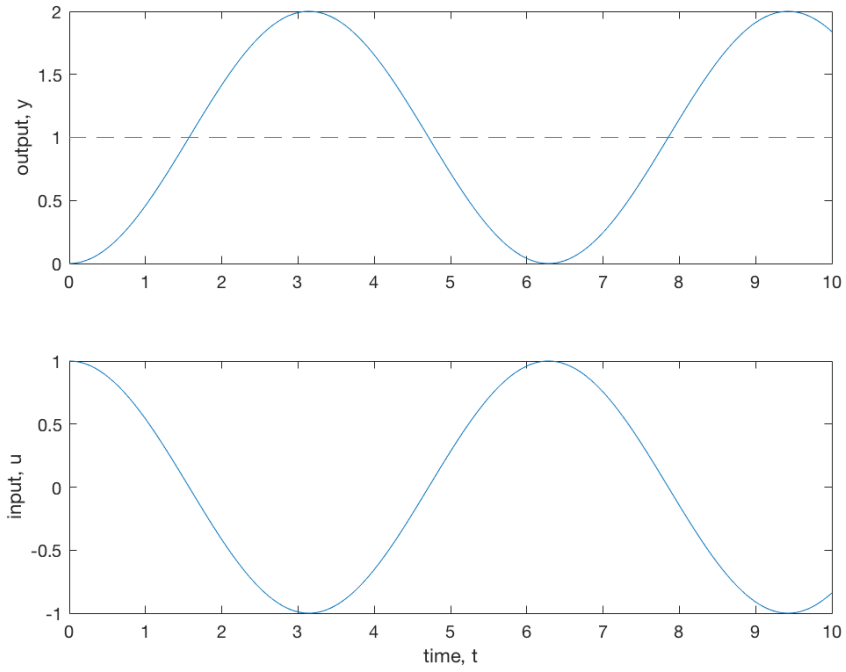
The PID controller has three parts: a proportional term K_p, which sets u proportionally to the state error; an integral term K_i, which sets u proportionally to an integral of the state error from the start of the trajectory to the current time; and a derivative term K_d, which sets u proportionally to the velocity error (the derivative of the state error). All three terms need not be included. P, PI, PD, and PID controllers are all quite common.

2.1 The Proportional Term

The proportional term is the workhorse term in any PID controller. Essentially what the P term does is create a virtual spring force (with stiffness K_p) which pulls the state towards the desired trajectory.

Problem 1: *You want to control the simple system of a mass on a frictionless rail. Your dynamics are* $m\ddot{x} = u$. *Assume that* $x_d(t)$ *and* $\dot{x}_d(t)$ *are uniformly zero. Your system starts at* $x(0) = 1$. *Sketch the system response given a proportional controller* $u = K_p e$ *where* $K_p = 1$

It'll oscillate back and forth forever:



In general increasing the proportional term will increase the amount of oscillation in the system response as well as increasing the *percent overshoot*, the amount by which the system response will exceed the setpoint in a step response. However, increasing the proportional gain will also increase the response speed.

2.2 The Derivative Term

The derivative term is the stabilizing term in a PID controller. Essentially what the D term does is add a damper (with viscosity K_d) between the state and its desired trajectory. A damper you see almost

every day are the one-way dampers mounted to the tops of doors which stop them from closing too fast. Remember that the dynamics of a damper are $F = -b\dot{x}$, so the damper exerts higher force the faster the state is changing. Similarly the derivative term in a PID controller will exert more control input the faster the error changes (in the direction opposite the change).
The derivative term provides stabilization, decreasing oscillation and overshoot, but will also slow down the system response.

2.2.1 Implementation Pitfalls: Noise

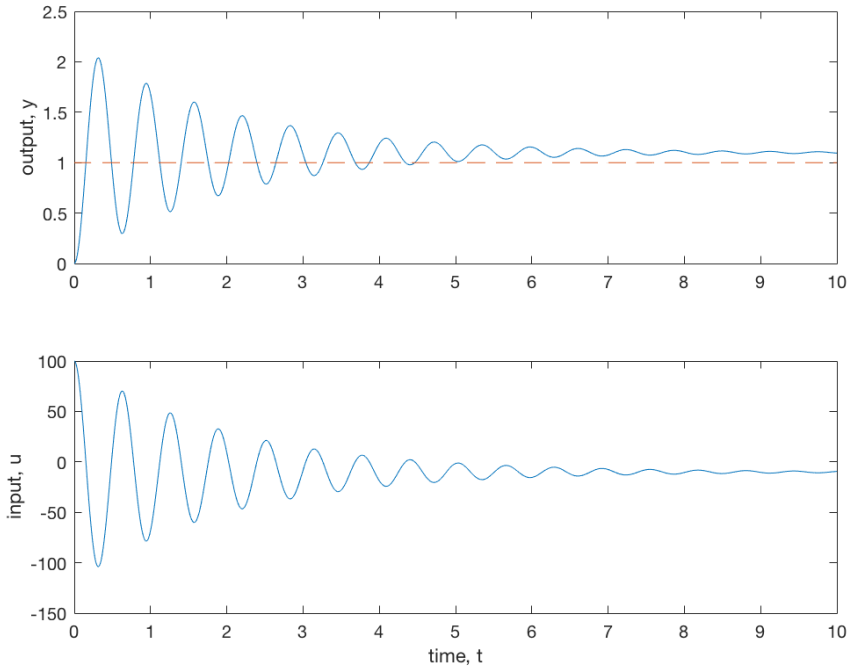
The derivative term has a major implementation-related pitfall. Often, a system's sensors will only measure the position of the system x, not its velocity $\dot{x}$. This means that in order to estimate $\dot{x}$ you need to approximate the derivative $\dot{x}(t) \approx \frac{x(t)-x(t-\delta t)}{\delta t}$. If you've taken EECS 16B, you'll recognize this difference equation as a *high-pass filter*. A high-pass filter amplifies high frequency components of an input signal; the faster the input changes the higher the output will be.
The problem emerges when you consider sensor noise. Sensor noise tends to be rather low in magnitude, but high in frequency. A common frequency for noise is 60 Hz (the frequency of AC current in the US), but the resonant frequencies of any RLC circuits or mechanical components you're using are also quite common. A derivative term will naturally amplify any sensor noise and potentially cause instability in your controller. Thus, you generally want to apply a *low-pass filter* such as a moving average on your error derivative before feeding it into your derivative term. For example, the PID controller in lab 7 averages the past three measurments of $\dot{e}$ before feeding it into the controller. The low pass filter will attenuate the high frequency signals such as sensor noise and give you a cleaner control signal.

2.3 The Integral Term

The integral term doesn't have as clear an effect as the proportional and derivative terms, nor does it have a neat physical analog. In order to appreciate the effect of the integral term, we need to consider the effect of disturbance forces on the system.

Problem 2: *Imagine you want to control the same mass-rail system as you did in Problem 1. However, now the rail has friction, and is oriented vertically, so you'll have to deal with gravity. Your dynamics are now* $m\ddot{x} + g + \dot{x} = u$. *Assume that* $x_d(t)$ *and* $\dot{x}_d(t)$ *are uniformly zero. Your system starts at* $x(0) = 1$. *Approximately sketch the system response given a proportional controller* $u = K_p e$ *where* $K_p = 100$. *Where does the system stabilize as* $t \to \infty$.

It'll converge, but it won't end up converging to the desired setpoint. Instead, it'll be slightly offset.



When there are constant disturbance or drift forces, it's impossible for a PD controller to settle at $x(t \to \infty) = x_d(t \to \infty)$. This is where the integral term is useful. By adding up error over time, the integrator can chip away at this constant disturbance and cause the state to converge to the desired trajectory. However, in doing so integral controllers tend to decrease system stability (even proving stability for a system with integrators becomes much harder). If tuned improperly, they can easily cause significant oscillation.

2.3.1 Implementation Pitfalls: Windup

Integral control also has a major implementation pitfall, which is called windup. Imagine that you're controlling the same mass-rail system as before, but now you have actuation constraints. Real-world actuators usually can't produce arbitrary torques or velocities at any given time. There are often constraints on the maximum velocity, acceleration, and jerk of a given actuator. If you start with a very high error, your proportional term will likely saturate your actuators, and you'll end up taking a longer time to reach the goal state. During this time, the integral term will keep increasing, even though this increase won't change u. When it finally hits the goal state the integral term will be very large, and the system will overshoot until the integral winds down again. Then on the way back the integrator will wind up again, this time in the opposite direction, and you'll get overshoot again. If the wind up gets bad enough, the integrator can easily destabilize the system, overshooting the goal more and more each time.

Problem 3: *Brainstorm some potential anti-windup methods. What are their drawbacks?*
Some possible solutions:

- Time-discount your integrals (what we did in lab)
- Fixed integration horizon
- Limit the size of the integral

- Figure out if you're saturating your input, and don't change your integral term in the direction of saturation if you are.

All of these methods have benefits and shortcomings, and picking a method that works is just another thing you have to tune.

3 Model-Based Control

All the control we've done thus far has been model-free. However, we often have some idea of the dynamics of our system, even if it's only approximate. If we incorporate this information into our controller, we can usually get significant benefits without many drawbacks. Furthermore, we can often precalculate expected inputs when we calculate our state trajectories. We can easily incorporate this information into our controller. This is called *feed-forward control*.

3.1 Feedforward Control

Imagine that we have a second order (force-controlled) system that we want to move from point A to B. We can generate a trajectory $x_d(t)$ that will navigate the system from A to B. Since this is a function of time, we can take the derivative to find $\dot{x}_d(t)$ and do so again to find $\ddot{x}_d(t)$ (we could also use Euler differentiation to approximate the derivative of a set of waypoints). If we have a good estimate of the system's inertia $m(x)$, we can plug in the trajectory to find $m(t)$, then find $F_d(t) = m(t)\ddot{x}_d(t)$. Now we have an estimate of the input force required at every time step, assuming that we start where the plan starts, and no disturbances occur. We can incorporate this information into our PID control law by adding a feedforward term

$$u(t) = u_f f(t) K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t)$$

In this example $u_f f(t) = F_d(t)$. Feedforward control doesn't oscillate and can't cause instability (assuming that the system itself is stable). Because of this, we want the feedforward term to do a majority of the control work, and use our feedback terms for small corrections. A good feedforward term will *drastically* improve performance and require significantly less tuning, with essentially no downsides.

3.2 Feedback Linearization

Sometimes, we can do better than plain feedforward control when we're dealing with highly nonlinear systems, like robot manipulators. If you look at a general control-affine (the dynamics have an affine dependence on the control) nonlinear system

$$\dot{x} = f(x) + g(x)u$$

you have two nonlinear terms. The *drift vector* $f(x)$ incorporates all the internal forces in the system (coriolis and gravitational forces for example) while the *control vectors* $g(x)$ determine how your control inputs affects your dynamics. If we can figure out a set of inverse dynamics

$$u = a(x) + b(x)\dot{x}_d = -g^{-1}(x)f(x) + g^{-1}(x)\dot{x}_d$$

we can use this as our controller and exactly cancel out the nonlinearities of our system to get

$$\dot{x} = \dot{x}_d$$

It's not always possible to invert the dynamics like this (if you want to figure out if it's possible, you'll need to take a graduate nonlinear controls class), but it so happens that the dynamics of any open-chain robot manipulator are invertible. You'll examine the feedback linearization of these systems in your homework, and should you choose to take EECS C106B, you'll be implementing it on hardware.