CHAPTER 23

PID Feedback Control

The cruise control system on a car is an example of a feedback control system. The actual speed v of the car is measured by a sensor (e.g., the speedometer) to yield a sensed value s; the sensed value is compared to a reference speed r set by the driver; and the error $e = r - s$ is fed to a control algorithm that calculates a control for the motor that drives the throttle angle. This control therefore changes the car's speed v and the sensed speed s. The controller tries to drive the error $e = r - s$ to zero.

Figure 23.1 shows a block diagram for a typical feedback control system, like the cruise control system. Typically, a computer (the PIC32 in our case) calculates the error between the reference and the sensor value and implements the control algorithm. The controller produces a control signal that is input into the *plant*, the physical system that we want to control. Often we model the plant's dynamics with differential equations derived from Newton's laws. The plant produces an output that the sensor measures. A system that uses a sensor measurement to determine its control signal is a *closed-loop* control system (the sensor feedback causes the block diagram to form a closed loop).

In this chapter, for simplicity, we assume that the sensor is perfect. Therefore, the sensed output s and the plant's actual output are the same.

A common test of a controller's performance is to start with the reference r equal to zero, then suddenly switch r to 1 and keep it constant for all time. (Equivalently, for the cruise control case, r could start at 50 mph then change suddenly to 51 mph.) Such an input is called a *step input* and the resulting error $e(t)$ is called the *step error response*. Figure 23.2 shows a typical step error response and illustrates three key metrics by which controller performance is measured: overshoot, 2% settling time t_s, and steady-state error e_{ss}. The settling time t_s is the amount of time it takes for the error to settle to within 0.02 of its final value, and the steady-state error e_{ss} is the final error. A controller performs well if the system has a short settling time and zero (or small) overshoot, oscillation, and steady-state error.

Perhaps the most popular feedback control algorithm is the *proportional-integral-derivative* (PID) controller. Entire books are devoted to the analysis and design of PID controllers, but PID control can also be used effectively with a little intuition and experimentation. This chapter provides a brief introduction to help with that intuition.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00023-8

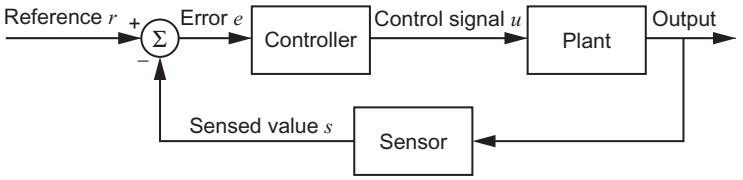

Figure 23.1
A block diagram of a typical control system.

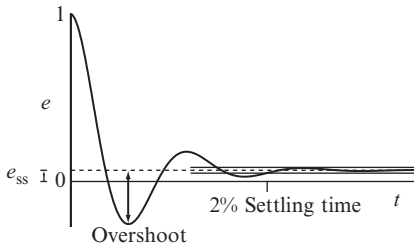

Figure 23.2
A typical error response to a step change in the reference, showing overshoot, 2% settling time t_s, and steady-state error e_{ss}. The error is one immediately after the step change in the reference and converges to e_{ss}.

23.1 The PID Controller

Assume the reference signal is $r(t)$, the sensed output is $s(t)$, the error is $e(t) = r(t) - s(t)$, and the control signal is $u(t)$. Then the PID controller can be written

$$u(t) = K_p e(t) + K_i \int_0^t e(z)\,\mathrm{d}z + K_d \dot{e}(t), \tag{23.1}$$

where K_p, K_i, and K_d are called the proportional, integral, and derivative gains, respectively. To make the discussion of the control law (23.1) concrete, let us assume r is the desired position of a mass moving on a line, s is the sensed position of the mass, and u is the linear force applied to the mass by a motor. Let us look at each of the proportional, derivative, and integral terms individually.

Proportional. The term $K_p e(t)$ creates a force proportional to the distance between the desired and measured position of the mass. This force is exactly what a mechanical spring does: it creates a force that pulls or pushes the mass proportional to a position displacement. Thus the proportional term $K_p e$ acts like a spring with a rest length of zero, with one end attached to the mass at s and the other end attached to the desired position r. The larger K_p, the stiffer the virtual spring.

Because we define the error as $e = r - s$, K_p should be positive. If K_p were negative, then in the case $s > r$ (the mass's position s is "ahead" of the reference r), the force

$K_p(r - s) > 0$ would try to push the mass even further ahead of the reference. Such a controller is called *unstable*, as the actual error tends toward infinity, not zero.

Derivative. The term $K_d\dot{e}(t)$ creates a force proportional to $\dot{e}(t) = \dot{r}(t) - \dot{s}(t)$, the difference between the desired velocity $\dot{r}(t)$ and the measured velocity $\dot{s}(t)$. This force is exactly what a mechanical damper does: it creates a force that tries to zero the relative velocity between its two ends. Thus the derivative term $K_d\dot{e}$ acts like a damper. An example of a spring and a damper working together is an automatic door closing mechanism: the spring pulls the door shut, but the damper acts against large velocities so the door does not slam. Derivative terms are used similarly in PID controllers, to damp overshoot and oscillation typical of mass-spring systems.

As with K_p, K_d should be nonnegative.

Integral. The term $K_i \int_0^t e(z)\,\mathrm{d}z$ creates a force proportional to the time integral of the error. This term is less easily explained in terms of a mechanical analog, but we can still use a mechanical example. Assume the mass moves vertically in gravity with a gravitational force $-mg$ acting downward. If the goal is to hold the mass at a constant height r, then a controller using only proportional and derivative terms would bring the mass to rest with a nonzero error e satisfying $K_p e = mg$, the upward force needed to balance the gravitational force. (Note that the derivative term $K_d\dot{e}$ is zero when the mass is at rest.) By increasing the stiffness of K_p, the error e can be made small, but it can never be made zero—nonzero error is always needed for the motor to produce nonzero force.

Using an integral term allows the controller to produce a nonzero force even when the error is zero. Starting from rest with error $e = mg/K_p$, the time-integral of the error accumulates. As a result, the integral term $K_i \int_0^t e(z)\,\mathrm{d}z$ grows, pushing the mass upward toward r, and the proportional term $K_p e$ shrinks, due to the shrinking error e. Eventually, the term $K_i \int_0^t e(z)\,\mathrm{d}z$ equals mg, and the mass comes to rest at r ($e = 0$). Thus the integral term can drive the steady-state error to zero in systems where proportional and derivative terms alone cannot.

As with K_p and K_d, K_i should be nonnegative.

It is possible to implement a PID controller purely in electronics using op amps. However, nearly all modern PID controllers are implemented digitally on computers.[1] Every `dt` seconds, the computer reads the sensor value and calculates a new control signal. The error derivative $\dot{e}$ becomes an error difference, and the error integral $\int_0^t e(z)\,\mathrm{d}z$ becomes an error sum. Pseudocode for a digital implementation of PID control is given below.

```
eprev = 0;              // initial "previous error" is zero
eint = 0;               // initial error integral is zero
now = 0;                // "now" tracks the elapsed time
```

[1] A digital PID controller is a type of digital filter, just like those discussed in Chapter 22.

```
every dt seconds do {
s = readSensor();              // read sensor value
r = referenceValue(now);       // get reference signal for time "now"
e = r - s;                     // calculate the error
edot = e - eprev;              // error difference
eint = eint + e;               // error sum
u = Kp*e + Ki*eint + Kd*edot; // calculate the control signal
sendControl(u);                // send control signal to the plant
eprev = e;                     // current error is now prev error for next iteration
now = now + dt;                // update the "now" time
}
```

A few notes about the algorithm:

- **The timestep `dt` and delays.** Generally, the shorter `dt` is, the better. If computing resources are limited, however, it is enough to know that the timestep `dt` should be significantly shorter than time constants associated with the dynamics of the plant. So if the plant is "slow," you can afford a longer `dt`, but if the system can go unstable quickly, a short `dt` is needed. The primary reason is that near the end of a control cycle, the control applied by the controller is in response to old sensor data. Control based on old measurements can cause the system to become unstable.
  For many robot control systems, `dt` is 1 ms.
- **Error difference and sum.** The pseudocode uses an error difference and an error sum. Instead, the error derivative can be approximated as `edot = (e - eprev)/dt` and the error integral could be approximated using `eint = eint + e*dt`. There is no need to do these extra divisions and multiplications, however, which simply scale the results. This scaling can be incorporated in the gains K_d and K_i.
- **Integer math vs. floating point math.** As we have seen, addition, subtraction, multiplication, and division of integer data types is much faster than with floating point types. If we wish to ensure that the control loop runs as quickly as possible, we should use integers where possible. Consider that raw sensor signals (e.g., encoder counts or ADC counts) and control signals (e.g., the period register of an output compare PWM signal) are typically integers anyway. If necessary, control gains can be scaled up or down to maintain good resolution while only using integer values during calculations, while also making sure that integer overflow does not occur. After calculations, the control signal can be scaled back to an appropriate range. The idea of scaling to allow integer math is exactly the same used for fixed-point math in DSP in Chapter 22.6.
  For many applications, since the PID controller involves only a few additions, subtractions, and multiplications, integer math is not necessary (especially on the PIC32, with its relatively fast clock speed).
- **Control saturation.** There are practical limits on the control signal `u`. The function `sendControl(u)` enforces these limits. If the control calculation yields `u=100`, for example, but the maximum control effort available is 50, the value sent by `sendControl(u)` is 50.

If large controller gains `Kp`, `Ki`, and `Kd` are used, the control signal may often be saturated at the limits.

- **Integrator anti-windup.** Imagine that the integrator error `eint` is allowed to build up to a large value. This windup creates a large control signal that tries to create error of the opposite sign, to try to dissipate the integrated error. To limit the oscillation caused by this effect, `eint` can be bounded. This *integrator anti-windup* protection can be implemented by adding the following lines to the code above:

```
eint = eint + e;               // error sum
if (eint > EINTMAX) {          // ADDED: integrator anti-windup
 eint = EINTMAX;
} else if (eint < -EINTMAX) { // ADDED: integrator anti-windup
 eint = -EINTMAX;
}
```

  Choosing `EINTMAX` is a bit of an art, but a good rule of thumb is that `Ki*EINTMAX` should be no more than the maximum control effort available from the actuator.
- **Sensor noise, quantization, and filtering.** The sensor data take discrete or *quantized* values. If this quantization is coarse, or if the time interval `dt` is short, the error `e` is unlikely to change much from one cycle to the next, and therefore `edot = e - eprev` is likely to take only a small set of different values. This effect means that `edot` is likely to be a jumpy, low-resolution signal. The sensor may also be noisy, adding to the jumpiness of the `edot` signal. Digital low-pass filtering, or averaging `edot` over several cycles, yields a smoother signal, at the expense of added delay from considering older `edot` values.

Although the PID control algorithm is quite simple, the challenge is finding control gains that yield good performance. Tuning these gains is the topic of Section 23.3.

23.2 Variants of the PID Controller

Common variants of the PID controller are P, PI, and PD controllers. These controllers are obtained by setting K_i and/or K_d equal to zero. Which variant to use depends on the performance specifications, sensor properties, and the dynamics of the plant, particularly its *order*. The order of a plant is the number of integrations from the control signal to the output. For example, consider the case where the control is a force to drive a mass, and the objective is to control the position of the mass. The force directly creates an acceleration, and the position is obtained from two integrations of the acceleration. Hence this is a second-order system. For such a system, derivative control is often helpful. If the system lacks much natural damping, derivative control can add it, slowing the output as it approaches the desired value. For zeroth-order or first-order systems, derivative control is generally not needed. Integral control should always be considered if it is important to eliminate steady-state error.

Table 23.1: Recommended PID variants based on the order of the plant

Control	Output of Plant	Order	Recommended Controller
Force	Position of mass	2	PD, PID
Force	Velocity of mass	1	P, PI
Current	Voltage across capacitor	1	P, PI
Current	Brightness of LED	0	P, PI

A rough guide to choosing the PID variant is given in Table 23.1. While PID control can be effective for plants of order higher than two, we will not consider such systems, which can have unintuitive behavior. An example is controlling the endpoint location of a flexible robot link by controlling the torque at the joint. The bending modes of the link introduce more state variables, increasing the system's order.

23.3 Empirical Gain Tuning

Although there is no substitute for analytic design techniques using a good model of the system, useful controllers can also be designed using empirical methods.

Empirical gain tuning is the art of experimenting with different control gains on the actual system and choosing gains that give a good step error response. Searching for good gains in the three-dimensional K_p-K_i-K_d space can be tricky, so it is best to be systematic and to obey a few rules of thumb:

- **Steady-state error.** If the steady-state error is too big, consider increasing K_p. If the steady-state error is still unacceptable, consider introducing a nonzero K_i. Be careful with K_i, though, as large K_i can destabilize the system.
- **Overshoot and oscillation.** If there is too much overshoot and oscillation, consider increasing the damping K_d, or the ratio K_d/K_p.
- **Settling time.** If the settling time is too long, consider simultaneously increasing K_p and K_d.

It is a good idea to first get the best possible performance with simple P control ($K_i = K_d = 0$). Then, starting from your best K_p, if you are using PD or PID control, experiment with K_d and K_p simultaneously. Experimenting with K_i should be saved until last, when you have your best P or PD controller, as nonzero K_i can lead to unintuitive behavior and instability.

Assuming you will not break your system by making it unstable, you should experiment with a wide range of control gains. Figure 23.3 shows an example exploration of a PD gain space for a plant with unknown dynamics. The control gains are varied over a few orders or magnitude. For larger control gains, the actuator effort $u(t)$, indicated by dotted lines, is often saturated. As expected, as K_p increases, oscillation and overshoot increases while the steady-state error decreases. As K_d increases, oscillation and overshoot is damped.

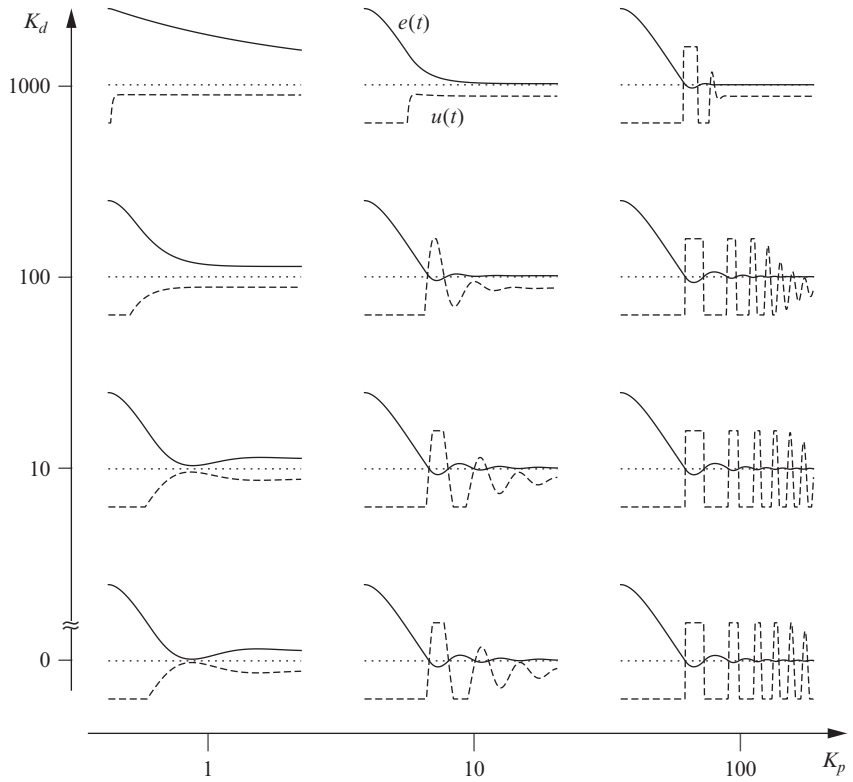

Figure 23.3
The step error response for a mystery system controlled by different PD controllers. The solid lines indicate the error response $e(t)$ and the dotted lines indicate the control effort $u(t)$. Which controller is "best?"

There are practical limits to how large controller gains can be. Large controller gains, combined with noisy sensor measurements or long cycle times `dt`, can lead to instability. They can also result in controls that chatter between the actuator limits.

23.4 Model-Based Control

With a feedback controller, like the PID controller, no control signal is generated unless there is error. If you have a reasonable model of the system's dynamics, why wait for error before applying a control? Using a model to anticipate the control effort needed is called *feedforward control*, because it depends only on the reference trajectory $r(t)$, not sensor feedback.
A model-based feedforward controller can be written as

$$u_{\text{ff}}(t) = f(r(t), \dot{r}(t), \ddot{r}(t), \ldots),$$

where $f(\cdot)$ is a model of the inverse plant dynamics that computes the control effort needed as a function of $r(t)$ and its derivatives (Figure 23.4).

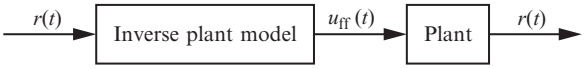

Figure 23.4
An ideal feedforward controller. If the inverse plant model is perfect, the output of the plant exactly tracks the reference $r(t)$.

Since feedforward control is not robust to inevitable model errors, it can be combined with PID feedback control to get the control law

$$u(t) = u_{\text{ff}}(t) + K_p e(t) + K_i \int_0^t e(z)\,\mathrm{d}z + K_d \dot{e}(t). \tag{23.2}$$

In control of a robot arm, this control law is called *computed torque control*, where u is the set of joint torques and the model $f(\cdot)$ computes the joint torques needed given the desired joint angles, velocities, and accelerations.

A related control strategy is to use the reference trajectory $r(t)$ and the error $e(t)$ to calculate a desired change of state. For example, if the plant is a second-order system (the control u directly controls $\ddot{s}$), then the desired acceleration $\ddot{s}_d(t)$ of the plant output can be written as the sum of the planned acceleration $\ddot{r}(t)$ and a PID feedback term to correct for errors:

$$\ddot{s}_d(t) = \ddot{r}(t) + K_p e(t) + K_i \int_0^t e(z)\,\mathrm{d}z + K_d \dot{e}(t).$$

Then the actual control $u(t)$ is calculated using the inverse model,

$$u(t) = f(s(t), \dot{s}(t), \ddot{s}_d(t)). \tag{23.3}$$

If the inverse model is good, an advantage of the control law (23.3) over (23.2) is that the effect of the constant PID gains is the same at different states of the plant. This property can be important for systems like robot arms, where the inertia about a joint can change depending on the angle of outboard joints. For example, the inertia about your shoulder is large when your elbow is fully extended and smaller when your elbow is bent. A shoulder PID controller designed for a bent elbow may not work so well when the elbow is extended if the output of the PID controller is a joint torque. By treating the PID terms as accelerations instead of joint torques, and by passing these accelerations through the inverse model, the shoulder PID controller should have the same performance regardless of the configuration of the elbow.[2]

Feedforward plus feedback control laws like (23.2) and (23.3) provide the advantage of smaller errors with less control effort as compared to feedback control alone. The cost is in

[2] Provided the inverse model is good and the control effort does not saturate.

developing a good model of the plant dynamics and in increased computation time for the controller.

23.5 Chapter Summary

- Performance of a control system is often evaluated by the overshoot, 2% settling time, and steady-state error of the step error response.
- The PID control law is $u(t) = K_p e(t) + K_i \int_0^t e(z)\,\mathrm{d}z + K_d \dot{e}(t)$.
- The proportional gain K_p acts like a virtual spring and the derivative gain K_d acts like a virtual damper. The integral gain K_i can be useful for eliminating steady-state error, but large values of K_i may cause the system to become unstable.
- Common variants of PID control are P, PI, and PD control.
- To reduce steady-state error, K_p and K_i can be increased. To reduce overshoot and oscillation, K_d can be increased. To reduce settling time, K_p and K_d can be increased simultaneously. Stability considerations place practical limits on controller gains.
- Feedback control requires error to produce a control signal. Model-based feedforward control can be used in conjunction with feedback control to anticipate the controls needed, thereby reducing errors.

23.6 Exercises

1. Provided with this chapter is a simple MATLAB model of a one-joint revolute robot arm moving in gravity. Perform empirical PID gain tuning by doing tests of the error response

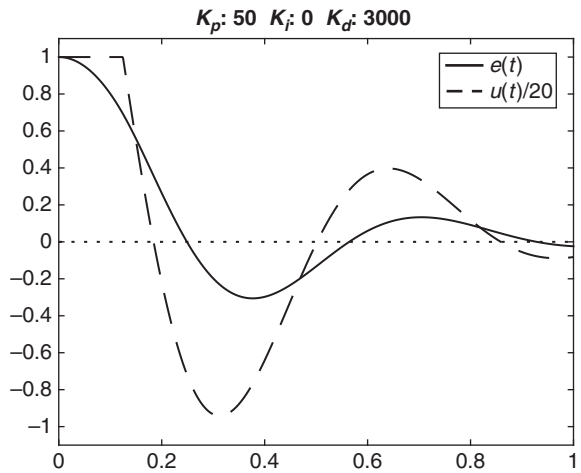

Figure 23.5
The step error and control response of the one-joint robot for $K_p = 50, K_i = 0, K_d = 3000$.

to a step input, where the step input asks the joint to move from $\theta = \theta = 0$ (hanging down in gravity) to $\theta = 1$ radian. Tests can be performed using

```
pidtest(Kp, Ki, Kd)
```

Find good gains K_p, K_i, and K_d, and turn in a plot of the resulting step error response. An example output of `pidtest(50, 0, 3000)` is given in Figure 23.5.

Code Sample 23.1 `pidtest.m`**. Empirical Gain Tuning in MATLAB for a Simulated One-Joint Revolute Robot in Gravity.**

```
function pidtest(Kp, Ki, Kd)

INERTIA = 0.5;        % The plant is a link attached to a revolute joint
MASS = 1;             % hanging in GRAVITY, and the output is the angle of the joint.
CMDIST = 0.1;         % The link has INERTIA about the joint, MASS center at CMDIST
DAMPING = 0.1;        % from the joint, and there is frictional DAMPING.
GRAVITY = 9.81;
DT = 0.001;           % timestep of control law
NUMSAMPS = 1001;      % number of control law iterations
UMAX = 20;            % maximum joint torque by the motor

eprev = 0;
eint = 0;
r = 1;                % reference is constant at one radian
vel = 0;              % velocity of the joint is initially zero
s(1) = 0.0; t(1) = 0; % initial joint angle and time
for i=1:NUMSAMPS
  e = r - s(i);
  edot = e - eprev;
  eint = eint + e;
  u(i) = Kp*e + Ki*eint + Kd*edot;
  if (u(i) > UMAX)
    u(i) = UMAX;
  elseif (u(i) < -UMAX)
    u(i) = -UMAX;
  end
  eprev = e;
  t(i+1) = t(i) + DT;

  % acceleration due to control torque and dynamics
  acc = (u(i) - MASS*GRAVITY*CMDIST*sin(s(i)) - DAMPING*vel)/INERTIA;

  % a simple numerical integration scheme
  s(i+1) = s(i) + vel*DT + 0.5*acc*DT*DT;
  vel = vel + acc*DT;
end

plot(t(1:NUMSAMPS),r-s(1:NUMSAMPS),'Color','black');
hold on;
plot(t(1:NUMSAMPS),u/20,'--','Color','black');
set(gca,'FontSize',18);
legend({'e(t)','u(t)/20'},'FontSize',18);
plot([t(1),t(length(t)-1)],[0,0],':','Color','black');
axis([0 1 -1.1 1.1])
title(['Kp: ',num2str(Kp),'  Ki: ',num2str(Ki),'  Kd: ',num2str(Kd)]);
hold off
```

Further Reading

Franklin, G. F., Powell, D. J., & Emami-Naeini, A. (2014). *Feedback control of dynamic systems* (7th ed.). Upper Saddle River, NJ: Prentice Hall.

Ogata, K. (2002). *Modern control engineering* (4th ed.). Upper Saddle River, NJ: Prentice Hall.

Phillips, C. L., & Troy Nagle, H. (1994). *Digital control system analysis and design* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.