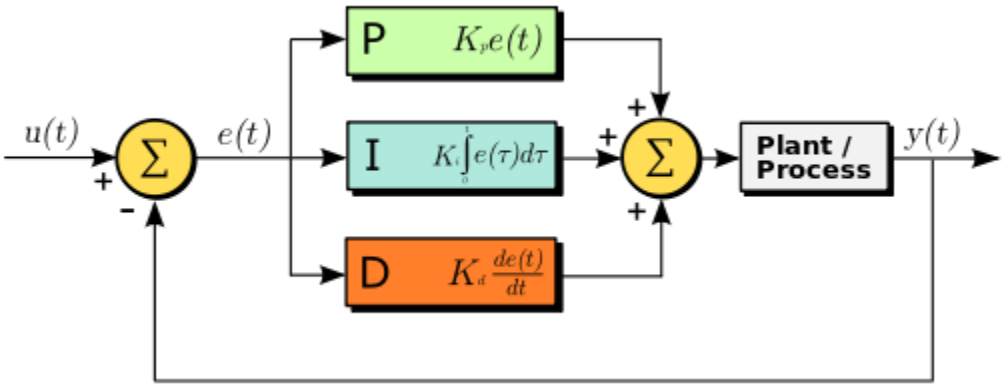
How to Build a Fixed-Point PI Controller That Just Works: Part I

Jason Sachs ● February 26, 2012

This two-part article explains five tips to make a fixed-point PI controller work well. I am *not* going to talk about loop tuning -- there are hundreds of articles and books about that; any control-systems course will go over loop tuning enough to help you understand the fundamentals. There will always be some differences for each system you have to control, but the goals are the same: drive the average error to zero, keep the system stable, and maximize performance (keep overshoot and delay low). Save your questions about stability and performance for someone else, and let's focus on implementation details.

Let's start with the basics. You have some quantity **y(t)** you'd like to control to match a command signal **u(t)**. You can sense **y(t)** with a sensor. In order to control **y(t)**, you have an actuator that you send a signal **x(t)**.



(Diagram from the Wikipedia entry on PID controllers. Actuator signal = output of the rightmost summing block = input to plant/process is not labeled, but corresponds to x(t).)

The way to do this with a PI controller is to calculate the error e(t) = u(t)-y(t), then calculate x(t) = Kp * e(t) + Ki * integral(e(t)). (We're not going to include a D term in the controller; most systems don't need one, which is something I'll discuss in Part II. The integral term is important because it lets you reach equilibrium with zero steady-state error; the integrator output is what's able to drive the output with zero error, since at that point the proportional term is zero.)

Or at least that's the textbook approach.

1. Canonical discrete-time form: a PI controller in the digital world

Analog systems can implement the PI controller directly, but digital systems have to restate the problem slightly.

In a digital PI controller, you will almost always have a controller that executes at a fixed timestep dt, and instead of continuous-time signals, you have discrete variables that are updated at that fixed timestep. The usual method, at least conceptually, is to number the timesteps 1, 2, 3, 4, ... n. This doesn't mean you need to make an array in your program, just that for analysis purposes you can think of it as an array with an index that increments for each timestep.

The PI controller can be analyzed using z-transforms if it's put into a canonical form:

```
x[n] = x[n-1] + a*e[n] + b*e[n-1]
```

In other words, the actuator control signal is equal to its old value plus some gains times the previous and current error terms.

You can relate **a** and **b** to the proportional gain Kp and integral gain Ki as follows:

x(t) = Ki*integral(e(t)) + Kp*e(t)

dx/dt = Ki*e(t) + Kp*de/dt

(x[n]-x[n-1])/dt = Ki*e[n] + Kp*(e[n]-e[n-1])/dt (approximating derivatives with discrete differences)

x[n] - x[n-1] = Ki*e[n]*dt + Kp*(e[n]-e[n-1])

and therefore, by matching terms, get a = (Ki*dt + Kp), b = -Kp.

To implement this in a program, the PI controller becomes:

```
e_prev = e;
e = u - y;
x_prev = x;
x = x_prev + a*e + b*e_prev;
```

And to analyze with MATLAB or another numerical toolset, the controller can be written in terms of z-transforms:

$(1 - z^{-1})x = (a + bz^{-1})e$

$x = e * (a + bz^{-1})/(1 - z^{-1})$

Got it?

Good. Now throw it out, because although this form may be useful for modeling in numerical analysis software, it has problems in real systems.

Why?

Well, for one thing, the canonical form gains **a** and **b** have indirect significance: you'd like to tune proportional and integral gains directly, or maybe proportional gain and time constant (= Kp/Ki) directly, and instead you're forced to deal with this **a** and **b** nonsense. If I use **a** and **b** and someone asks me what the integral gain is, or I want to double the integral gain, I have to get out my calculator and figure it out.

The other two reasons have to do with nonlinearity. In linear systems, you can prove that this canonical form is equivalent to one of several other possible implementations of a PI controller. I'm going to come back to this issue a little bit later, as it's rather subtle, and instead will talk about an aspect of PI loop implementation that's a bit easier to understand.

2. Which comes first, the integrator or the gain?

Let's look at the integral term of the controller, Ki*integrator(e(t)).

Integration and scaling are two linear operations, and that means in a perfect linear system,

Ki*integrator(e(t)) = integrator(Ki*e(t))

In a digital controller, you could implement this in one of the following ways:

```
// option A: integrate, then scale
e_integral = e_integral + e*dt;
x = Kp*e + Ki*e_integral;


// option B: scale, then integrate
x_integral = x_integral + Ki*e*dt;
x = Kp*e + x_integral

;
```

In a linear system, the two are exactly equivalent.

But there are five reasons to choose option B over option A. Two of these reasons are related to nonlinearity, and one reason is related to time-varying systems, and we'll talk about those in a little bit, but the remaining two are a bit more straightforward.

Reason #1: integrator unit significance

Let's talk about a typical example. You've got a current controller based on an actuator (perhaps a switching circuit) with a voltage command. The sensor **y**, the command **u**, and the error term **e**, are all currents measured in amperes. The actuator command **x** is a voltage measured in volts.

Now what about the integral terms? In option A, the integral term `e_integral` is ampere-seconds = coulombs. In option B, the integral term `x_integral` is a voltage measured in volts.

Here's the lesson I want you to learn:

- **If you integrate and then scale, the integrator is measured in units of the integrated input unit.**
- **If you scale and then integrate, the integrator is measured in the same units as the output.**

If you choose option B, the integrator's significance is more directly related to the controller output. It's a subtle difference, but one that in my experience is very important.

Reason #2: folding the timestep into the integrator implementation

There's a related reason that has to do with this timestep value dt. If you choose option B, you can fold the timestep into the integral gain, and make Ki2 = Ki * dt:

```
// option B2: scale, then integrate; Ki2 = Ki*dt
x_integral = x_integral + Ki2*e;
x = Kp*e + x_integral

;
```

This saves you a multiply step, requiring less time to execute in a processor. If you try to do this with option A, it looks like this:

```
// option A2: integrate, then scale; Ki2 = Ki*dt
e_integral = e_integral + e;
x = Kp*e + Ki2*e_integral
```

```
;
```

This will work just as well as option B2, saving a multiply step, but now the units of the variable `e_integral` are kind of weird. Dimensional analysis tells us they're the same units as the input value: if e is measured in amperes and you add it to a variable, then that variable must also be measured in amperes. But on the other hand, you're summing a variable over time, so the real units of `e_integral` are coulombs scaled by 1/dt. That's right, the units of the integrator now depend on the timestep value, and are a numerically-scaled integral of the input unit. Ick. This makes my head hurt.

Reason #3: time-varying gain

Here's a thought experiment. Let's say I've got two controllers. One is option A (integrate, then scale), and the other is option B (scale, then integrate). The system is stable, and the output has reached equilibrium.

Now all of a sudden I decide to double the integral gain. (Assume the new gain would still keep the system stable.) What happens?

In option A, the integrator's output value doubles, so my output suddenly jumps and presents a disturbance to the system, which will decrease as the system regains equilibrium.

In option B, the integrator's output value remains essentially the same. The output doesn't jump and the system remains at equilibrium.

Got that? If you change the integral gain suddenly with option A, your controller will see a glitch; with option B, it won't.

It's that simple.

One corollary or side-effect of this is when you set the integral gain to zero. With scale-then-integrate, you have an integrator that will keep its output constant, and you can use it as an offset that you can manually adjust. With integrate-then-scale, the integrator's effect on the output is zero. I prefer an offset that I can adjust.

Reasons #4 and #5: nonlinearities

So far, we've talked only about linear systems. It's time to start talking about nonlinearities, and to do so we'll skip ahead to the third important concept I want to bring up, and come back to the integrate-then-scale vs. scale-then-integrate debate in a bit.

3. Integrator windup

Let's go back to our example of a current controller.

Suppose we command u = 1A of current, and it takes a command to our output actuator x = 7.2V to reach equilibrium. Great.

Now suppose we want u = 2A of current. To do this x = 14.4V.

Now suppose we want u = 10A of current. To do this x = 72V.

Now suppose we want u = 1000A of current. To do this x = 7200V.

Now suppose we want u = 1,000,000A of current... what? You're telling me we're not going to be able to get a million amperes of current out of our system?

Linear systems have the wonderful property that their output scales identically to their input. But no real

system that I know of is perfectly linear: at some point we run into some physical limit that prevents us from exceeding a maximum, or we run into a phenomenon that is negligible at small values of input/output values, but starts to grow faster than linearly as the amplitude increases. An example of the former is op-amp saturation (an op-amp cannot output more than its supply rails); an example of the latter is mechanical vibrations -- at small amplitudes, most materials have a linear stress-strain relationship, but at larger amplitudes, you can run into buckling or plasticity.

Suppose we have a current controller that has an actuator supplied by a 14.4V power supply, and we can get up to 2A of current out of it.

Now we increase the commanded current to 2.1A, a 5% increase. The error is 0.1A; the proportional term will jump 5% immediately, and the integral term will gradually increase. But our actuator is only able to deliver 14.4V, and the 0.1A error persists as long as we wait. We'll soon ask for 15V, 20V, 30V, 50V... supposing we wait until our controller output is 100V, and we decide to set the commanded current to 0.

What's our output current? Well, the error will be 0A command - 2A current = -2A until the actuator command drops below 14.4V, but it's going to take a while for the integrator to decrease down to the point where the output decreases below 14.4V. In the meantime the output current will still be 2A.

This phenomenon is called integrator windup, and it's present in all real systems with an actuator saturation limit where the controller that doesn't take this into account somehow.

There are many ways to prevent integrator windup (look up "anti-windup" and you'll find lots of articles), but the two most common are the following:

- limit the integrator output
- add another feedback term that represents the error due to windup

The first approach, which I've used extensively in motor controllers, is to disallow integrating in the direction the actuator has saturated, but allow integration in the other direction, which would bring the integrator away from saturation. Here's a sample program implementation, which I'll call my Preferred Implementation, because it's the one I generally use in most control systems:

```
if ((sat < 0 && e < 0) || (sat > 0 && e > 0))
 ;
/* do nothing if there is saturation, and error is in the same direction;
 * if you're careful you can implement as "if (sat*e > 0)"
 */
else
 x_integral = x_integral + Ki2*e;
(x_integral,sat) = satlimit(x_integral, x_minimum, x_maximum);

x = limit(Kp*e + x_integral, x_minimum, x_maximum)

;


/* satlimit(x, min, max) does the following:
 * if x is between min and max, return (x,0)
 * if x < min, return (min, -1)
 * if x > max, return (max, +1)


*

 * limit(x, min, max) does the following:
 * if x is between min and max, return x
```

```
 * if x < min, return min
 * if x > max, return max
 */
```

The second approach, which I'll call the "shortcoming-feedback" anti-windup approach, is less familiar to me, but I've seen it in many papers:

```
x_integral = x_integral + Ki2*(e - Ks*x_shortcoming);
x_ideal = Kp*e + x_integral;
x = limit(x_ideal, x_minimum, x_maximum);
x_shortcoming = x_ideal - x;
```

The idea here is that you integrate normally, but as soon as you have to limit the output, take the discrepancy between the limited output and the ideal output, and feed it back into the integrator. If the output hasn't reached the limit, the integrator behaves normally, but if it has reached the limit, it will stop integrating because this discrepancy will balance out the error.

There are lots of subtleties about anti-windup compensation that I won't get into here -- I just wanted to bring the issue to your attention, and you can read more about it on your own. (For an interesting reference, see this short paper from Warsaw Polytechnic which presents 11 different variants on anti-windup PI controllers; the two I've mentioned are in figures #10 and #7)

Consequences of integrator windup in PI controller structure

If you use the stop-integrating-when-the-integrator-reaches-the-saturation-limit approach, you have to know what the saturation limit is. That seems kind of obvious, but let's go back and look at some of our implementation alternatives:

Canonical form:

x[n] = x[n-1] + a*e[n] + b*e[n-1]

```
e_prev = e;
e = u - y;
x_prev = x;
x = x_prev + a*e + b*e_prev;
```

Where's the integrator? Well, it's buried in this equation and combined with the proportional term. But you can try to take the same approach with the output x:

```
e_prev = e;
e = u - y;
x_prev = x;
dx = a*e + b*e_prev;
if ((dx > 0 && sat > 0) || (dx < 0 && sat < 0))
 ;
/* do nothing if there's saturation and we would otherwise increase the output
 * in the same direction
 */
else
 x = x_prev + dx;

(x,sat) = limit(x, x_minimum, x_maximum);
```

This looks like it might work, and if you try it in your system, it might appear to work.

But it has a little problem.

Suppose we have a noisy sensor, which isn't that out of the ordinary. Instead of the error being a nice smooth input that gradually decays down to zero, the error will be bouncing around, sometimes a small negative amount and sometimes a small positive amount.

Let's say we're exactly at the upper saturation limit for our output actuator. In our noisy system, the output change dx will sometimes be positive and sometimes negative, but its average value will be zero, since both e and e_prev have average values of zero. The proper behavior for a controller is to keep the controller output at that limit when the error is at or above zero.

The problem is that positive values of dx will be added in and then limited, but negative values of dx will reduce the output. What you'll find is that the controller's output value will bounce around near but slightly below its upper limit, effectively giving up some saturation range. (Let's call this phenomenon "limit hesitation" since I don't think it has a name.)

We can get around this problem by raising the controller's saturation limit above its physical limit (e.g. in our example, if the current controller's voltage limit has a real saturation point of 14.4V, then our controller's output limit could be set slightly above this point, perhaps to 15V or 16V) and then we'll still be able to reach the actuator's physical limit.

But that's kind of a hack.

The implementation I gave when I introduced the anti-windup section (my "Preferred Implementation") doesn't have this problem to the same degree, because unlike the canonical-form implementation, it keeps the integrator and proportional terms separate. In the presence of noise, if you look at the integrator term and proportional term, noise will show up in the proportional term just as much as in the input term, but it will be diminished in the integrator term, because the integrator term's transfer function lets low-frequency components through but attentuates high-frequency content. The "limit hesitation" is therefore much smaller when you keep integrator and proportional terms separate, and that's a major reason why I don't use the canonical-form implementation (aside from the conceptual opaqueness).

The "shortcoming-feedback" anti-windup controller appears to have a similar problem when used with canonical-form implementation as well, because the integrator and proportional terms are combined. Again, I'm not as familiar with this type of anti-windup controller, so I can't tell you whether limit hesitation can be overcome with shortcoming-feedback.

Integrate-then-scale implementation (reason #4 not to use it):

This approach allows you to use anti-windup almost as easily as the scale-then-integrate implementation, but there's a problem with that too:

```
// option A2: integrate, then scale; Ki2 = Ki*dt
e_integral = e_integral + e;
x = Kp*e + Ki2*e_integral

;
```

Here's the problem: what's the saturation limit for the integral term? You have to calculate it as a function of the integral gain: e_integral_limit = x_limit / Ki2.

If you don't mind doing that divide, this approach will work; it should perform just as well in the presence of noise (with respect to limit hesitation) as the scale-then-integrate implementation.

Of course, I prefer to just use the scale-then-integrate approach.

4. Fixed-point scaling issues

So far, everything we've talked about applies to both floating-point and fixed-point control loops. In fixed-point controllers, numbers are stored as integers rather than floating-point values, and for each quantity you must select a scaling factor that relates integer counts to the engineering value they represent. You have to deal with overflow and resolution issues: if the scaling factor is too small, the integer count will overflow, but if the scaling factor is too large, the resolution will be poor.

We'll talk about the subtleties of this issue in part II of this article.

Previous post by Jason Sachs:
10 More (Obscure) Circuit Components You Should Know
Next post by Jason Sachs:
How to Build a Fixed-Point PI Controller That Just Works: Part II