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

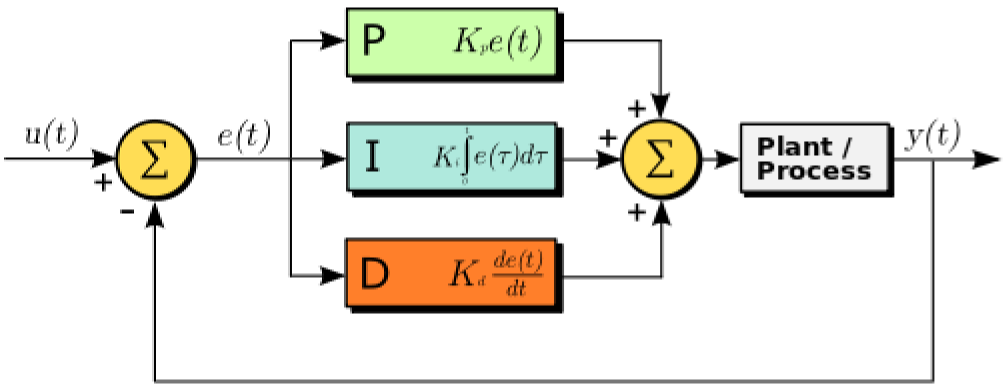
Jason Sachs ● March 25, 2012

In Part I we talked about some of the issues around discrete-time proportional-integral (PI) controllers:

- various forms and whether to use the canonical form for z-transforms (don't do it!)
- order of operation in the integral term: whether to scale and then integrate (my recommendation), or integrate and then scale.
- saturation and anti-windup

In this part we'll talk about the issues surrounding fixed-point implementations of PI controllers. First let's recap the conceptual structure and my "preferred implementation" for floating-point.

For a PID controller without saturation:



(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).)

For a PI controller with saturation, using floating-point arithmetic:

```
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
 */
```

Here **e** is the error, and **x** is the output, and **Kp** and **Ki2** are the proportional and integral gains, where Ki2 = Ki * the timestep dt.

Fixed-point basics

In fixed-point arithmetic, we typically use 16-bit or 32-bit quantities (rarely 8-bit or 64-bit) to represent engineering quantities. Since physical quantities have units, you are forced to make a decision for the scaling factor that relates the integer stored in software to the physical quantity. This probably sounds abstract, so let's use a specific example:

Suppose I have a 12-bit analog-to-digital converter measuring an analog input voltage from 0 to 3.0V, and the input voltage comes from a 20:1 voltage divider. This becomes fairly straightforward: 0 counts on the ADC represents 0V and 4096 counts (well, really 4095 counts but ignore that for now) represents 60V. That's a scaling factor of 14.65mV/count.

In fixed-point arithmetic, for conceptual purposes we often imagine a binary point (analogous to a decimal point) that is scaled by 2^Q for some number Q. In my example above, the choice of Q=12 is convenient: 4096 counts = 2^{12} * 1.0 which represents 1.0 times a certain full-scale ADC scaling factor, which is 60V in this case. This "Q" number is used to refer to a fixed-point representation: Q12 means that floating-point numbers are scaled by 2^{12} to be represented as integers, along with a further scaling factor for engineering units. The full specification of this engineering encoding is 60V Q12. To convert from integer to engineering values, we divide by 4096 and multiply by 60V; to convert from engineering values, we divide by 60V and multiply by 4096 and round to the nearest integer.

If I measured some voltage V1, and I wanted to multiply by a gain K where K = 1.237, I could also represent K by a Q12 number. 1.237*4096 = approximately 5067, so 5067 counts represents 1.237 in Q12 encoding.

A floating-point example (and here I'm going to use {} to denote floating-point quantities): {V1} = 38.2V, {K} = 1.237, {V2} = {V1} * {K} = 47.25V. Very simple.

To do this in fixed point, we'd start out with V1 = {V1} / 60V * 4096 = 2608 counts, and K = {K} * 4096 = 5067 counts.

In order to get {V2} = {V1} * {K}, we substitute:

{V2} = V2 * 60V / 4096
{V1} = V1 * 60V / 4096
{K} = K / 4096

V2 * 60V / 4096 = (V1 * 60V / 4096) * (K/4096)

and simplifying:

V2 = V1 * K / 4096

For our example, V1 = 2608 counts and K = 5067 counts, so V2 = V1 * K / 4096 = 3226 counts. Let's see if that makes sense, when we translate back to engineering units:

{V2} = V2 * 60 / 4096 = 3226 * 60 / 4096 = 47.26V

The resulting computation almost matches the floating-point case; the discrepancy is due to the limited resolution of fixed-point.

An important note: Please understand that in fixed-point math, the floating point numbers {V1} = 38.2V, {V2} = 47.26V, and {K} = 1.237 are never actually calculated unless we need to convert them into meaningful numbers for human beings or other consumers of data; instead, we only deal with the fixed-point quantities of V1 = 2608, V2 = 3226, K = 5067.

The generalization of this multiply step

{C} = {A} * {B}

with fixed-point representation

{A} = A * A_U / 2^{Q_A}
{B} = B * B_U / 2^{Q_B}
{C} = C * C_U / 2^{Q_C}

can be simplified as follows. Note the Q numbers may all be different, and so may the engineering unit scaling factors A_U, B_U, C_U.

C * C_U / 2^{Q_C} = (A * A_U / 2^{Q_A}) * (B * B_U / 2^{Q_B})

C = A * B * (A_U * B_U / C_U) / $2^{(Q_A + Q_B - Q_C)}$

We usually choose the unit factors to be related by a power of two, so that the net calculation

C = (A * B) >> k

can be performed, where an integer k = Q_A + Q_B - Q_C + $\log_2$ (A_U*B_U/C_U)

How do you pick a Q number, unit scaling factor, and integer bit size?

These are very important steps in the design of fixed point systems. What you need to look at are issues of resolution and overflow. Overflow relates to the minimum and maximum values that can be represented by an integer quantity, whereas resolution relates to the smallest possible incremental value.

If I have an unsigned 16-bit integer representing voltage with units 60V Q12, the minimum voltage is 0, the voltage per count is 60V / 4096 = 14.65mV/count, and the maximum voltage is 65535 * 60V / 4096 = 959.99V.

We could choose a larger scaling factor for Q12 representation, and it would raise both the maximum voltage and the voltage per count.

If you choose too large a scaling factor, you won't have the resolution you need. If you choose too small a scaling factor, the "ceiling" of your numeric range will be too low and you won't be able to represent large enough quantities.

Please realize also that these two quantities (the Q number and unit scaling factor) are not unique for the same representation: 60V Q12 = 15V Q10 = 960V Q15: in all three cases, the integer 2608 represents 38.2V. So the choice of whether to call a certain representation Q12 or Q10 or Q15 or something else is really arbitrary, it just forces a different engineering unit scaling factor.

If you are running into both problems (poor resolution and overflow) in different situations, it means you need to use a larger number of bits to store your engineering quantities.

My rule of thumb is that 16-bit integers are almost always sufficient to store most ADC readings, output values, and gain and offset parameters. 32-bit integers are almost always sufficient for integrators and intermediate calculations. There have been times that I've needed 48-bit or even 64-bit intermediate storage values, but this is rare except in situations of wide dynamic range.

If you have an ADC between 14 and 16 bits, you can use a 16-bit integer to store the raw reading, but you may still wish to use 32-bit storage to handle gain/offset calibration for your ADC -- otherwise, with gains that are slightly offset from 1 (e.g. 1.05 or 0.95), you may see problems in the low-order bits trying to store the result in a 16-bit number -- if the raw ADC count increases by one, sometimes the scaled result increases by 1 count and sometimes by 0 or 2 counts. This is a resolution problem and the cure is to use extra bits to minimize quantization error without running into overflow.

For PI controller inputs and outputs, the obvious choice is to pick the scaling factor such that the integer overflow range corresponds to the maximum input or output value, but sometimes this works well and sometimes it doesn't.

Back to the PI controller in fixed-point

OK, ready to implement a PI controller in fixed point? Here goes:

```
int16 x, sat;
int32 x_integral, p_term;
int16 Kp, Ki2;
const int16 N = [controls proportional gain scaling: see discussion later]
int16 x_min16, x_max16;
const int32 nmin = -(1 << (15+N));
const int32 nmax = (1 << (15+N)) - 1;

/* ... other code skipped ... */

int16 e = u - y;
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 + (int32)Ki2*e;
const int32 x_min32 = ((int32)x_min16) << 16;
const int32 x_max32 = ((int32)x_max16) << 16;
(x_integral,sat) = satlimit(x_integral, x_min32, x_max32);
p_term = limit((int32)Kp*e, nmin, nmax);
x = limit((p_term >> N) + (x_integral >> 16), x_min16, x_max16);

/* 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
 */
```

There are a couple of subtleties here.

Integrator scaling

In the implementation I've given above, the integrator is a 32-bit value with 65536=2^16 counts of the integrator equivalent to 1 count of the output. Or stated another way, the bottom 16 bits of the integrator are extra resolution to accumulate increases over time, and the top 16 bits of the integrator are the integrator's output.

You need more bits in integrator state variables than regular input/outputs, to handle the effects of small timesteps. This is also true for low-pass filter state variables. What you want is for the largest anticipated integral gain to not cause overflow, and the smallest anticipated integral gain to do something useful.

If Ki2 is very small (e.g. 1 count), the integrator will take a while to accumulate errors, but you will not lose any resolution. If we implemented the integrator as a 16-bit value like

```
x_integral = x_integral + (Ki2*e) >> 16;
```

then for low values of the integral gain, small error values would just disappear and never accumulate in the integrator.

A corollary of the "you need more bits in integrator state variables" rule, is that you should never execute a PI loop at a rate that is more slowly or more quickly than is reasonable. If you run the PI loop too slowly, the bandwidth and/or stability of the loop will suffer. That's pretty straightforward. But if you run the PI loop too quickly, that has a problem too: integrators that integrate 100,000 times per second have to integrate smaller amounts than integrators that integrate 1,000 times per second. The faster the rate you execute an integrator or filter, the more resolution you need in state variables. If you're executing your control loop less than 5 times the intended bandwidth, that's too slow, and if you're executing your control loop more than 500 times the intended bandwidth, that's probably too fast.

So one solution for dealing with numerical problems is to run the control loop at a slower rate -- just make sure you filter your inputs so you don't run into aliasing problems.

The integrate-then-scale vs. scale-then-integrate debate, yet again

When I brought up the issue of when to apply the integral gain (integrate-then-scale vs. scale-then-integrate), I said there were 5 reasons to prefer scale-then-integrate. Here is reason #5:

Scaling and then integrating is a very natural way to pick fixed-point scaling factors. Once you choose a scaling factor for the input error, and a scaling factor for the PI controller output, it very naturally comes together and the integrator usually just works if you use an extra 16 bits of integrator resolution, as I discussed above.

If you integrate and then scale, the integrator is measured in this weird intermediate scaling factor that you have to manage, and you have to handle resolution and overflow twice: once in the integrator, and once in the final gain scaling step. I find it more cumbersome to design around, so it's yet another reason not to implement the integration step before applying the integral gain.

Proportional gain scaling

I've written this control loop's proportional term with a variable shift term N between 0 and 16. Picking N=0 isn't appropriate in some cases, and picking N=16 isn't appropriate in some cases. You need to figure out what range of proportional gains you want, and make sure that the highest gain can be implemented without overflow, but the smallest gain has sufficient resolution: if the smallest gain translates to fixed point as a count of 1, and you want to adjust it by 10%, you're stuck -- the gain is either 0, 1, or 2 counts. Minimum gain values should be at least 5 or 10 counts when converted to fixed points. If you need a proportional gain adjustment range of more than 3000:1 (which is very unusual), you'll probably need to use 32-bit scaling factors and 32x32-bit math rather than 16x16 math.

The relationship between gain scaling factor is fixed by the choice of input and output scaling factors and the choice of N. As an example, suppose the input scaling factor is 2A = 32768 counts, and the output scaling factor is 14.4V = 32768 counts, and N = 8.

For a gain of 10V/A, you'd scale an input of 1A = 16384 counts, to an output of 10V = 22756 counts.

(16384 * K) >> 8 = 22756 means that K = 355.56 counts corresponds to 10V/A. If your system gain needs to be between 1V/A (=35.556 counts) and 100V/A (=3555.6 counts), the choice of N=8 is a good one. If your system gain needs to be between 0.1V/A (=3.5556 counts) and 10V/A, then N=8 is too small; N=10 or N=12 is a better one. If your system gain needs to be between 10V/A and 1000V/A (=35556 counts) then N=8 is too large; N=6 or N=4 is a better one.

That Darned Derivative!

Let's take a break for a second and come back to the derivative term that's been left out of all this discussion: we've been talking about PI controllers but have mentioned PID controllers a few times.

The D term is something I rarely if ever use, because it provides low gain for slowly-varying errors and high gain for the high frequency content in the error term, which is usually just noise.

If you can get by with a PI controller without the D term, do so -- your system will be simpler and you won't have to worry about it.

There are systems that do benefit from a D term. They usually involve systems with long delay or phase lag where there's a quantity that you'd like to observe, but can't.

For example, consider a thermal controller where you have a heater block and a temperature sensor. If the heater block is large and the temperature sensor is mounted on the outside, it may take a long time before the sensor sees any change in power applied to the heating element. Ideally you would also measure the temperature deep inside the heater block and use a relatively fast control loop to regulate that, and then a slower control loop to regulate the outside temperature. But if you can't add an extra temperature sensor, you need a way to notice when the heater block is starting to heat up. If you use a PI loop, the proportional term isn't fast enough: by the time the temperature error drops appreciably, you've already applied power for a long time to the heating element and even if you suddenly turn the heating element off, the temperature at the sensor is going to continue heating up as heat diffuses to the outside of the heating block. The integral term is even slower -- integral terms are there to handle DC and low-frequency errors; they're intentionally the slowest-responding part of your control loop. So a derivative term can help you throttle back the output of your controller when the sensor reading starts to increase but the error is still positive.

Because of noise content, it usually makes sense to implement the derivative term with a rolloff or low-pass filter: take differences between readings, but then filter out the really high-frequency content so you are left with frequencies that you care about. In a thermal control loop, unless the system in question is really small, the response times are measured in seconds, so anything higher than 10 or 20Hz probably isn't going to be useful.

Just remember that you shouldn't use a D term unless you need to.

OK, now back to the discussion of arithmetic and overflow.

Overflow in addition and subtraction

Let's look at this simple line:

```
int16 e = u - y;
```

There can't possibly be any errors here, right?

WRONG! If y = 32767 counts and u = -32767 counts, then e = -65534 counts. But that can't fit in an int16 variable, which only holds values between -32768 and +32767 counts; -65534 counts will alias to +2 counts which is an incorrect calculation.

What we have to do instead is one of three things:

1. use 32-bit math (which is kind of a pain) to leave room for the calculation to reach its full range of +/-65535 counts

2. ensure that under the worst-case conditions, we can never get overflow (e.g. if we are absolutely positive u and y are limited to the range 0-4095 counts) -- which isn't always possible

3. saturate the calculation:

```
int16 e = limit((int32)u - y, -32768, +32767);
```

Some processors have built-in arithmetic saturation, but it tends to be inaccessible from high-level languages like C.

Other lines that are causes for concern are the following:

```
x_integral = x_integral + (int32)Ki2*e;
x = limit((p_term >> N) + (x_integral >> 16), x_min16, x_max16);
```

That's right -- you need to look at any calculation that has an addition or subtraction.

The upper line (the integrator update) is not a problem as long as the right-hand side doesn't cause overflow, and the biggest factors here are the maximum values of x_integral (determined by x_max32 and x_min32) and the maximum integral gain and error. You may have to limit the Ki2*e product before adding it in, or use a temporary 64-bit variable first.

The lower line (limiting the sum of the proportional term and integral term) is not a concern as long as the intermediate value below is a 32-bit value:

```
(p_term >> N) + (x_integral >> 16)
```

In C, this will be the case because p_term and x_integral are both 32-bit values.

Overflow in multiplication

As long as you indicate to the compiler that you want a 16x16 = 32-bit multiply (or 8x8=16 or 32x32=64, as the case may be), you will never get an overflow. (Try it! Check with the extremes for both signed and unsigned integers.) Unfortunately, the arithmetic rules for C/C++ are to make the result of a multiplication the same type as its promoted inputs, so in lines like this:

```
int16 a = ..., b = ...;
int32 c = a*b;
```

the result for c can be incorrect, because a*b has an implicit type of int16. To get intermediate values in C to calculate 16x16=32 correctly, you have to promote one of the operands to a 32-bit integer:

```
int16 a = ..., b = ...;
int32 c = (int32)a*b;
```

A good compiler will see that and understand that you want to do a 16x16=32 multiply. A mediocre compiler will promote the 2nd operand b to 32-bit, do a 32x32 bit multiply, and take the bottom 32 bits of the result -- which gives the correct answer, but wastes CPU cycles in a library function call. If you have this kind of compiler, you have a few choices. One choice is to use a better compiler, and a second choice is that you may have access to a compiler intrinsic function that does the right thing:

```
int32 c = __imul1632(a,b);
```

where __imul1632 is the appropriate intrinsic function that does a 16x16 multiply; I made that function name up but there may be one on your system. When the compiler sees an intrinsic function, it replaces it with the appropriate short series of assembly instructions rather than actually making a library function call.

The third choice is that you should complain very loudly to your compiler vendor to make sure there is a mechanism for you to use in C to calculate a 16x16=32 multiply.

Once you've done the multiply, you either need to store it in a 32-bit value, or you need to shift and cast back down to a 16-bit value:

```
int16 a = ..., b = ...;
int16 c = ((int32)a*b) >> 16;
```

This calculation is overflow-free. But shift counts of less than 16 are prone to overflow, and you'll need to limit the intermediate result:

```
int16 a = ..., b = ...;
const int32 limit12 = (1 << (12+15)) - 1;
int16 c = limit((int32)a*b, -limit12, limit12) >> 12;
```

This is true even for a shift of 15, for one unfortunate case...

Other pathological cases of overflow

If you square -2^15 = -32768, you'll get 2^30. Shift right by 15 and you get 2^15, which will not fit in a signed 16-bit integer. (-32768)*(-32768) is the one calculation that will not fit in a c = (a*b)>>15 calculation. If you know definitively that one or both operands cannot be -32768 (e.g. if one number is a nonnegative gain), then you don't have to worry, but otherwise you'll have to limit the results before right-shifting:

```
int16 a = ..., b = ...;
const int32 limit15 = (1 << (15+15)) - 1;
int16 c = min32((int32)a*b, limit15) >> 15;
// min32(x,max) = minimum of the two values
```

The other pathological cases of overflow also involve this "evil" value of -32768.

```
int16 evil = -32768;
int16 good1 = -evil;
```

```
int16 good2 = abs(evil);
```

Unfortunately good1 also evaluates to -32768, even though logically it should be 32768 (which doesn't fit into a signed 16-bit integer). If you use an abs() function, make sure you understand how it handles -32768: if it's implemented like these, then you have the same problem:

```
#define abs(x) ((x)>=0 ? (x) : (-x))
inline int16 abs(x) { return x>=0 ? x : -x; }
```

If it's implemented using a built-in assembly instruction, you need to check how that instruction handles the input. The TI C2800 DSP has an ABS instruction, and it behaves differently if an overflow mode bit is set; if OVM is set, then ABS(-32768) = +32767, but otherwise, ABS(-32768) = -32768.

A step back

Whoa! Where were we again? Weren't we talking about a PI controller? Let's zoom back out to the big picture, and summarize.

As far as the choice of whether to use floating-point vs. fixed-point arithmetic, perhaps now you can understand that

- if we use floating-point arithmetic, we can represent engineering values directly in our software, we have a few subtleties to deal with, and it may cost us in resources (some processors don't have floating-point instructions, so floating-point math has to be handled in library function calls that are much slower; other processors do have floating-point instructions but they usually run more slowly than integer math)
- if we use fixed-point arithmetic, the math is really simple and fast for the computer to perform, but as control system designers we have to do a lot of grunt-work to ensure that we don't run into resolution or overflow errors in our systems.

If you do decide to use fixed-point arithmetic, consider these issues:

- choose scaling factors for input and output wisely
- add an extra 16 bits to the integrator size for the added resolution
- the scaling factor for proportional gain may need to have 32 bits if you have a system with large dynamic range
- check **all** your arithmetic steps for overflow, even simple add, subtract, and negate operations

In any case, be aware that you can implement a PI controller and have it just work right -- it just may take some extra work on your part to make sure it does.

Good luck on your next control system!

Previous post by Jason Sachs:
How to Build a Fixed-Point PI Controller That Just Works: Part I
Next post by Jason Sachs:
Hot Fun in the Silicon: Thermal Testing with Power Semiconductors

---