DSP Scaling, a PID Example: Difference equations must be scaled to fit into the numerical space of a microprocessor. There are 4 parts to the scaling problem. First, the dynamic range of the multiply-accumulate, MAC, instructions must be less than the word length of the microprocessor. Second, the scaling of the multiply must be defined as Integer, Fractional or Mixed. Third, the value of the binary word that represents a unit of input must be decided. Finally the MAC coefficients must be scaled to get an output with sufficient authority from the integral term to swing the output over the range of Duty ratio required by the controller.

MAC dynamic range can be defined as the largest coefficient divided by the smallest coefficient times the A/D converter range or:

$$Range = 2^{ADCbits}\frac{MaxCoef}{MinCoef}$$

For a PID controller[1], the coefficients in continuous time may be defined as P,I and D for the proportional, Integral and Differential coefficients. Using standard z transform definitions [2] for integration and differentiation gives the following coefficients for the z transform domain.

$$P_z = K_{coef}P$$

$$I_z = K_{coef}IT$$

$$D_z = K_{coef}\frac{D}{T}$$

Where T is the sample interval and Kcoef is a scaling parameter to be determined. The dynamic range is then

$$Range = \frac{D_z}{I_z} = \frac{D}{IT^2}$$

As an example, using 16 bit signed arithmetic, if

$Range > 2^{14}$, and

$V_{PREVIOUS} < -2^{14}$, and

$V_{PRESENT} > 2^{14}$, then

$V_{PRESENT} - V_{PREVIOUS}$ overflows and the controller can get stuck with the output at the maximum value. If dynamic range is too large then there are 3 choices; limit the output, reduce A/D converter bits or limit the error signal range. It turns out that limiting the error signal range has beneficial side effects. That's because the current charging the output filter is also limited to:

$$I_{LIM} = C\frac{dv}{dt} = C\frac{V_{LIM}}{T}$$

Charge current limiting protects parts from overstress and automatically gives a soft-start characteristic. Limiting usually does the trick; the next choice is to carefully examine the A/D converter accuracy. It's quite possible the vendor has overstated the accuracy so that leaving out some bits has no impact. Some microprocessors have saturation limiting built into the MAC instruction. But derivative saturation can lead to turn on overshoot.

Arithmetic Scaling revolves around the definition of a binary word and examination of the arithmetic operations on that word. Three word definitions are possible; Integer, Mixed and Fractional. Figure scaling_word shows how the bit weighting works.

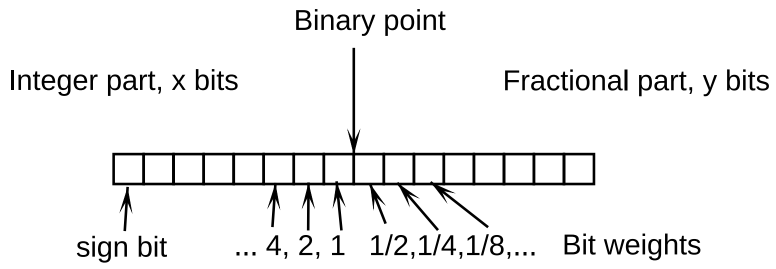


Figure scaling_word, Bit arrangement for 7.8 scaling.

If the number of bits to the left of the binary point are measured using x, and the bits to the right measured by y; then, x.y =“7.8” represents the scaling shown in the figure. This representation can also define integer scaling, “15.0” and fractional scaling, “0.15”. Notice that the sign bit isn’t included which makes the “x.y” notation work for both integers and fractions.

Addition and Subtraction work the same for all scaling methods. However multiplication works differently. Integer notation is the commonly accepted method used in programming languages from assembly language to high level languages such as the C programming language. The product fits into the originally scaled 16 bit register as long as it doesn’t overflow. The problem for the DSP designer is that integer multiplication can overflow; while fractional multiplication cannot overflow. Moreover, coefficient rounding is more pronounced with integer scaling. Some microprocessors have a fractional mode that keeps the result in the product accumulator scale as “0.30”. The product register in integer based multiplication scales as 1.30, shifted one bit right. If “one” is defined to be 2^y, then we want “one” * “one” to be “one”. That requires the product register to be shifted right by y bits following the MAC operations. The result can also be shifted into the upper accumulator by shifting the result left by (16-y) bits. The shift and store operation are usually available as single operations in assembly language so that one would expect saving ACCL(p >> 15) to be replaced by saving ACCH(p<<1) by a C compiler if the former shift is out of range.

Coefficient Accuracy: The three methods of arithmetic scaling result in different coefficient accuracy. The best accuracy is obtained using fractional scaling. The affect of coefficient errors is to misplace the pole-zero roots from their specified positions. But these specifications come from estimating component values that may have tolerances over temperature, age and operating points as high as 40%.

As you might expect, the definition of the arithmetic scaling has little effect on the net DSP computational result, at least for SMPS controllers because the specification accuracy. For more details see how the problem is solved using each of the 3 word definitions:

http://www.intusoft.com/ DSP/Drawings/MCHP2zLim.DWG

One: Next, it is necessary to define the binary value that represents a unit input. It’s interesting to note, that with mixed scaling, if the quantity is known as “one”, then casting “one” as a double and dividing variables by “one” will scale them back to their input units. If one = 256, then placing Verr/256.0 in a debugger window will scale it back into plant voltage. Input scaling is shown in Figure scaling_input.

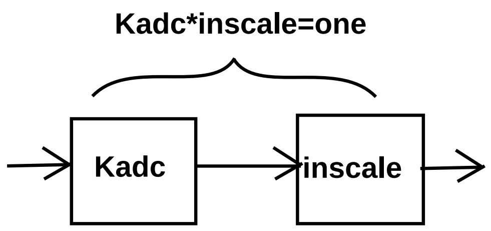


Figure scaling_input, Input scaling broken into 2 parts.

Notice that there are 2 parts to the input error scaling, the A/D scaling and an “inscale” multiplier. “inscale” is a shifting pre-scaler that increases the input value to “one” and its inverse is applied after the computations are complete. Kadc describes how the A/D converter measurement is processed, including scaling resistors, reference voltage and whether the ADC is left or right justified. “one”*FullScale must be greater than 2^ADCbits in order to retain all of the ADC precision, or

$$one > \frac{2^{ADCbits}}{FullScale}$$

FullScale is measured using the plant output voltage that saturates the A/D. Next, the coefficient multiplier must be selected to give maximum coefficient precision and provide the integrator sufficient authority to make the output controllable.

As the input limit is made smaller, there is more latitude in selection of inscale and Kcoef. These parameters tend to “center” the computation between underflow and overflow.

C Language programming: It is convenient, at least for prototyping, to do this arithmetic in a high level language. The C programming language is known to have a close relationship with hardware architectures. Unfortunately the C language integer types are 15.0 for int16. The fractional and mixed multiply, accumulate function can be computed using the C language as follows:

```
#define PRODUCT(a,x) (int32)a*x
Int16 mac2(const int16 coeof1, int16 var1,
        const int16 coef2, int16 var2,
           const int16 ybits)

{ // returns a scaled 16 bit result
  // for coef1*var1+coef2*var2
       return((int16) ( (     PRODUCT(coef1,var1) +
                              PRODUCT(coef2,var2)
                          ) >> ybits
                        )
              )
}
```

Notice the caste of the coefficient to 32 bits forces the C compiler to make the product a 32 bit word so that it can be shifted right, back into the int16 format. This can be made an inline function to maximize speed. It would be nicer to use C++ and override the arithmetic functions, but C++ adds extra code layer that slows down DSP operation and obscures the eventual hand translation into assembly language.

Selecting Kcoef: Whatever word scaling is chosen, the integrator should be scaled to operate between 0 and less that 0x7fff. Since the integrator ramps up slowly, it's possible to limit its values in a slower timing loop. Then selecting a limit value or ¾*0x7fff or 0x628F defines a slower timing loop requirement. The gain from the integrator output to the the duty ratio control value is then 1/(Kcoeff*one). With the limit set at .75*max, the duty ratio max value is 0x628F/one / Kcoeff, or Kcoef > 0x628f/one.

$$K_{coef} > \frac{0x628f}{one}$$

Then the conditions can be satisfied for any value of one from 1 to 32767; or from fractional through mixed to integer and the same controller operation is achieved.

The integrator output needs to be set to zero when input voltage falls below the specified under voltage; setting an under voltage lockout flag, UVLO, low. That keeps the integrator from locking up to its high limit when the control loop is unable to provide enough output voltage.

That is, $UVLO = \frac{V_{IN}}{D_{MAX}} < V_{OUT}$, where V_{IN} is the input voltage D_{MAX} and is the maximum duty ratio and V_{OUT} is the desired output. When UVLO is true, the PWM shuts down.

Slow timing loop: A slower interrupt can be selected for the integrator control by UVLO and saturation. The time is set by

$$T_{SLOW} < 2^{13} \frac{C}{one * I_{LIM} I_Z}$$

where 2^{13} is the integer value of integrator headroom.

The function of this interrupt is to limit the integrator and test for UVLO, resetting the integrator if UVLO is true. Typically, this loop needs to be evaluated every 10 or 20 T, sampling intervals.

Matrix Solution: The DSP control equations can be expressed using matrix algebra as shown in Figure scaling_2. Assume there are j states that need to be evaluated, with k of them having a delay history. The equations can be arranged as shown with all trivial solutions at the bottom of the matrix. Let Hn be the history value Hn. Then the j+k by j sub matrix at the top will have its right hand side equal to zero. After solving the matrix the Vn values are substituted into the Hn RHS for the next iteration. There may be more states than history because some of the states may include input and outputs. The main diagonal is scaled to be 1 so that there is no divide required in the solution. For large j, the matrix coefficients should be sparse and non zero values should be near the main diagonal. That's equivalent to having a number of blocks with a single input and output cascaded. If the original matrix had non-zero coefficients below the main diagonal, then the matrix solves an algebraic set of simultaneous equations. DSP's can be made to have to all zero values below the main diagonal by judicious use of backward euler integration to break up the signal flow. That has the side effect of adding delays and reducing controller bandwidth.

$$\begin{bmatrix} 1 & a12 & a13 & \dots & \dots & a1j & \dots & & a1(j+k) \\ 0 & 1 & a23 & & & a2j & \dots & & \dots \\ 0 & 0 & 1 & a34 & & a3j & \dots & & \dots \\ 0 & 0 & 0 & 1 & & a4j & \dots & & \dots \\ \dots & & & & \dots & ajj & \dots & & a(j+k)(j+k) \\ \dots & & & & & \dots & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} V1 \\ V2 \\ \dots \\ \dots \\ Vj \\ Hk \\ H3 \\ \dots \\ H1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \dots \\ \dots \\ 0 \\ Vkp \\ V3 \\ \dots \\ V1p \end{bmatrix}$$

Figure Scaling_2, A matrix solution has RHS(0 thru j)=0

LU decomposition, following the forward substitution gives us exactly what’s needed [2]. Then backward substitution is a multiply accumulate series for all non zero coefficients followed by division by the main diagonal value. If mixed precision is used, the main diagonal can be normalized to unity; eliminating the division. If integer or fractional scaling is used, the result can be multiplied by a predetermined constant, formed by dividing the scaling value by the main diagonal value, then applying the inverse of the scaling value to the outputs. The solution proceeds from the jth row and j+1 column, summing the products of the non-zero coefficient with their associated states. An array of coefficients is made in the order they will be used and a corresponding array of state-pointers can be made to make maximum use of the DSP multiply accumulate capability.

```
const int16  coef[numRowCoef];
iInt16 * varptr[numRowCoef];
int16 Vn;
while (numRowCoef--)
       Vn += *Coef++ *  *(*varptr++));
```

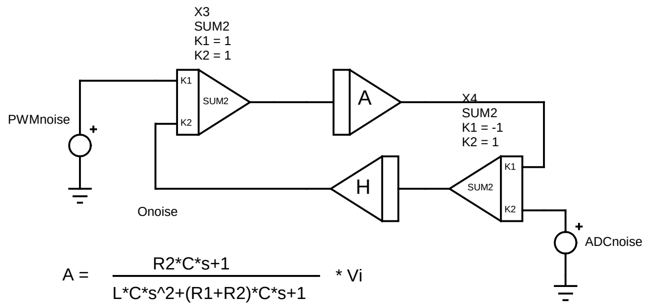
C compilers will figure this out; but there’s always hand coded assembly language to fall back on.

For Reduced Instruction Set Computers, RISC, it may be necessary to limit the range of variable index change from one computation to the next. This can be accomplished by moving the rows with H coefficient up until they are just below the first coefficient used in that column, and the moving the column left to place the unit value on the main diagonal. Such a movement doesn’t change the Lower triangle zero condition; but it tends to cluster coefficients along the main diagonal. Then the varptr usage shown above is replaced as shown below:

```
const int16  coef[numRowCoef];
iInt16  offset[numRowCoef];
int16 *varptr;
int16 Vn;
while (numRowCoef--)
       Vn += *Coef++ *  *(varptr + *offset++);
```

This form may need some adjustment depending on how the C compiler does its optimization. If the user identifies states that need a solution, then unwanted states can be eliminated by matrix manipulation. That reduces the number of MAC initializations and result storage; making a faster solution.

Noise: Quantizing of the input and output leads no noise. The RMS noise is equivalent to the LSB/sqrt(12). The PID controller A/D noise can be referred to the controller input and combined with the PWM noise. A/D's usually have more noise than this theoretical value, it's easy for a vendor to inflate accuracy by adding bits, so you need to make a few measurements around 0 and at your set point to test the A/D accuracy. Using a modified PID controller results in the block diagram as shown in Figure scaling_3. Referring the A/D noise to the PWM duty ratio results in amplifying A/D noise by the reciprocal of the power filter gain at the controller bandwidth. Taking the gain at the controller bandwidth gives a peak value; however, it is observed that quantizing noise tends to run close to the frequency domain peak. Moreover, its best to add the terms together rather than use an RSS approach because these values are no longer in a gausian world. They may very well tend to sync with one another rather than behave as independent random variables.



$$A = \frac{R2*C*s+1}{L*C*s^2+(R1+R2)*C*s+1} * Vi$$

choose H so that A*H = B/s

then onoise = PWMnoise + ADCnoise/A*(B/(s+B))
and if s = jB, and R2 is small,
. onoise = ADCnoise*.707*L*C*B^2/Vi+ PWMnoise
and if R2 is large
. onoise = ADCnoise*.707*L*B/(Vi*R2) + PWMnoise

B = Bandwidth (1/sec)
R2=capacitor ESR
R1=Inductor ESR
L=PWM inductor
C=PWM capacitor

Figure scaling_3, Calculation of quantizing noise for an ideal controller

The 2 computations form an upper and lower bound for noise without concern for B with respect to 1/(R2*C). For both cases, increasing B increases the quantizing noise at the PWM, for the case studies done here, it runs between 2 and 4 times larger, so that the PWM accuracy can be 1 or 2 bits less than the effective A/D accuracy. If round off error occurs anywhere in the computation, additional noise sources must be added at those points and summed into the resulting PWM noise.

[1] PID control equations

Given a plant described by Figure PID_1,

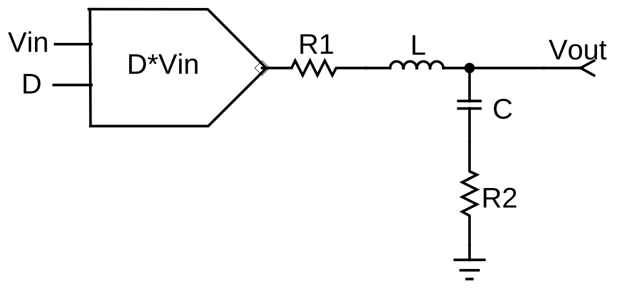


Figure PID_1, the plant model

Its transfer function, Aplant, is:

$$A_{plant} = K_p \frac{R_2Cs+1}{LCs^2+(R_1+R_2)Cs+1}$$

Where

C is the capacitor value
L is the inductor value
R1 is the inductor ESR
R2 the capacitor ESR
Kp is the PWM gain = Vin

And a controller that compensates for the plant by matching the plant poles and zeros, where P,I,D are controller coefficients shown below.

$$A_{comp} = P + \frac{I}{s} + Ds$$

$$A_{comp} = \frac{I}{s}\left(\frac{\frac{D}{I}s^2 + \frac{P}{I}s+1}{R_sCs+1}\right)$$

If the controller cancels the open loop poles and zeros, then the loop gain G is given by:

$$G = A_{plant}A_{comp} = K_pI = \frac{B}{s}$$

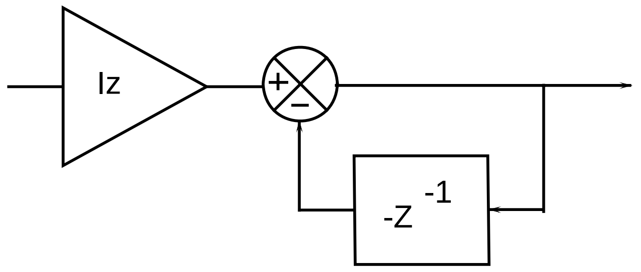
Then for bandwidth B,

$$I = \frac{B}{K_p}$$
$$D = LCI$$
$$P = (R_1+R_2)CI$$

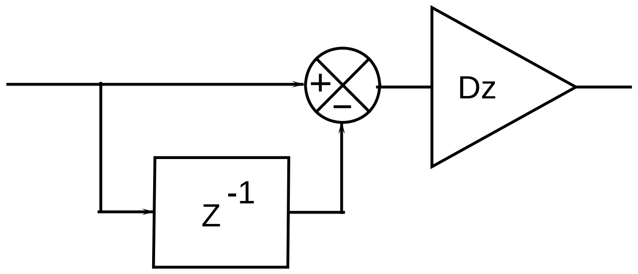
If R2, the capacitor ESR is large, it may be necessary to insert a canceling lag, $\frac{1}{R_2Cs+1}$ or other compensation in the controller.

[2] z transform PID controller

Using $\frac{1}{s} = \frac{T}{1 - z^{-1}}$ for forward euler integration where T is the sample time



and $s = \frac{1 - z^{-1}}{T}$ for differentiation



then substituting into the PID equation we find the z coefficient become

$$P_z = P$$
$$I_z = IT$$
$$D_z = \frac{D}{T}$$

[3] Quantizing Noise

Assume the measurement of an input variable is equally likely to be made anywhere in between A/D bits. Then drawing a straight line between –1/2LSB to +1/2LSB, the error is

$$E_{RMS}{}^2 = \frac{1}{\Delta}\int_{-\Delta/2}^{\Delta/2} \Delta^2 ds = \frac{\Delta^2}{12}$$

Where $\Delta = LSB$
Then

$$E_{RMS} = \frac{\Delta}{\sqrt{12}} = \frac{LSB}{\sqrt{12}}$$

This calculation assumes the measured input is uniformly spread across the measurement space. Unfortunately, in control systems, the noise tends to synchronize and move to the frequency where gain peaks. For most controllers that’s near the controller bandwidth.