Digital PID Controllers

Dr.Varodom Toochinda

June 2011

Proportional-Integral-Derivative (PID) control is still widely used in industries because of its simplicity. No need for a plant model. No design to be performed. The user just installs a controller and adjusts 3 gains to get the best achievable performance. Most PID controllers nowadays are digital. In this document we discuss digital PID implementation on an embedded system. We assume the reader has some basic understanding of linear controllers as described in our other document.

Different forms of PID

A standard "textbook" equation of PID controller is

$$u(t) = K\left(e(t) + \frac{1}{T_i} \int_0^t e(\tau)d\tau + T_d \frac{de(t)}{dt} \right) \tag{1}$$

where the error e(t), the difference between command and plant output, is the controller input, and the control variable u(t) is the controller output. The 3 parameters are K (the proportional gain), T_i (integral time), and T_d (derivative time).

Performing Laplace transform on (1), we get

$$G(s) = K\left(1 + \frac{1}{sT_i} + sT_d \right) \tag{2}$$

Another form of PID that will be discussed further in this document is sometimes called a parallel form.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt}e(t) \tag{3}$$

With its Laplace transform

$$G(s) = K_p + \frac{K_i}{s} + sK_d \tag{4}$$

We can easily convert the parameters from one form to another by noting that

$$K_p = K$$
$$K_i = \frac{K}{T_i}$$
$$K_d = KT_d$$

(5)

Discrete-time PID Algorithm

For digital implementation, we are more interested in a Z-transform of (3) *

$$U(z) = \left[K_p + \frac{K_i}{1 - z^{-1}} + K_d(1 - z^{-1}) \right] E(z)$$

(6)

Rearranging gives

$$U(z) = \left[\frac{(K_p + K_i + K_d) + (-K_p - 2K_d)z^{-1} + K_d z^{-2}}{1 - z^{-1}} \right] E(z)$$

(7)

Define

$$K_1 = K_p + K_i + K_d$$
$$K_2 = -K_p - 2K_d$$
$$K_3 = K_d$$

(7) can then be rewritten as

$$U(z) - z^{-1}U(z) = \left[K_1 + K_2 z^{-1} + K_3 z^{-2} \right] E(z)$$

(8)

which then converted back to difference equation as

$$u[k] = u[k-1] + K_1 e[k] + K_2 e[k-1] + K_3 e[k-2]$$

(9)

a form suitable for implementation. Listing 1 shows how to code this algorithm in C. We assume that the plant output is returned from a function readADC( ), and the control variable u is outputted using writeDA( ). Note that u must be bounded above and below depending on the DAC resolution. For instance, UMAX = 2047 and UMIN = -2048 for 12-bit DAC.

---

* To simplify the discussion, we omit the effect of sampling period on the PID parameters. Also, this particular form is called backward Euler. Other variations are forward Euler and trapezoidal methods.

```c
double e, e1, e2, u, delta_u;

k1= kp + ki + kd;

k2=-kp – 2*kd;

k3= kd;

void pid( )

{
        e2 = e1;                          // update error variables

        e1 = e;

        y = readADC( );                   // read variable from sensor

        e = setpoint – y;                 // compute new error

        delta_u = k1*e + k2*e1 + k3*e2;   // PID algorithm (3.17)

        u = u + delta_u;

        if (u > UMAX) u = UMAX;           // limit to DAC range

        if(u < umin) u = UMIN;

        writeDA(u);                       // send to DAC hardware
}
```

Listing 1: C code for the PID algorithm

FPGA Implementation

Listing 2 demonstrates a Verilog code to implement (9) on a CPLD or FPGA chip, assuming that the values of 3 parameters are hard-coded. One could modify this code to make the parameters user-adjustable. It is left as an exercise.

```verilog
module PID #(parameter  W=15) // bit width – 1
(output signed  [W:0]  u_out,  // output
 input   signed  [W:0]  e_in,    // input
input    clk,
input    reset);
parameter      k1=107;          // change these values to suit your system
parameter      k2 = 104;
parameter      k3 = 2;


reg     signed  [W:0]  u_prev;
reg     signed  [W:0]  e_prev[1:2];


assign  u_out = u_prev + k1*e_in – k2*e_prev[1] + k3*e_prev[2];


always @ (posedge clk)
        if (reset == 1) begin
                u_prev <= 0;
                e_prev[1] <= 0;
                e_prev[2] <= 0;
        end
        else begin
                e_prev[2] <= e_prev[1];
                e_prev[1] <= e_in;
                u_prev <= u_out;
        end
endmodule
```

Listing 2: PID implementation on FPGA using Verilog

PID Tuning

Adjusting the PID gains to achieve a good response could be problematic, especially for an inexperienced user. As a result, most commercial PID controllers have functions to tune the 3 parameters automatically. This is normally called "autotunig" feature. There are some variants of autotunig methods suggested in the literature. Here we mention one of them, the relay feedback, which is closely related to a manual tuning scheme known as Ziegler-Nichols Frequency Domain (ZNFD) method.

So we start by explaining ZNFD procedure. First we have to caution that, to conform to the derivation from [1], our ZNFD discussion refers to the "textbook" PID equation (1), not the parallel form (3). This does not pose any problem since the two forms are closely related by (5).

To tune a PID controller manually by ZNFD method, we start by turning off both the integral and derivative terms. From (1) we see this can be done by letting $T_i \to \infty$ and $T_d \to 0$. So now the PID is left only with the proportional gain K. We crank K up to the point that the closed-loop system starts to oscillate. At this point, the plant output will swing in a constant sinusoid motion, not growing and not dying out. Write this value down on a paper as K_u. Then find a way to measure the period of oscillation. Note this period as T_u. That's all. Suggested values of the 3 parameters can be found from Table 1. Example 1 demonstrates this procedure in simulation.

Controller Form	K	T_i	T_d
P	$0.5K_u$	-	-
PI	$0.4K_u$	$0.8T_u$	-
PID	$0.6K_u$	$0.5T_u$	$0.125T_u$

Table 1: suggested PID parameters from ZNFD method

Example 1: We want to experiment ZNFD method on this plant

$$P(s) = \frac{1}{(s+1)^3} \qquad (10)$$

Figure 1 shows a SIMULINK setup used for this simulation. We turn off the I and D terms and adjust K until K = 8, the output oscillates. Figure 3 captures the oscillation. Hence $K_u = 8$, and from Figure 3 $T_u = 3.5$. Using Table 1, we get K = 4.8, $T_i = 1.75$ and $T_d = 0.4375$. Figure 3 shows a step response when these values are used. Note that the overshoot is quite excessive (50%). In a sense, ZNFD just gives us some good values to start with. We can often fine-tune the PID to improve the response.
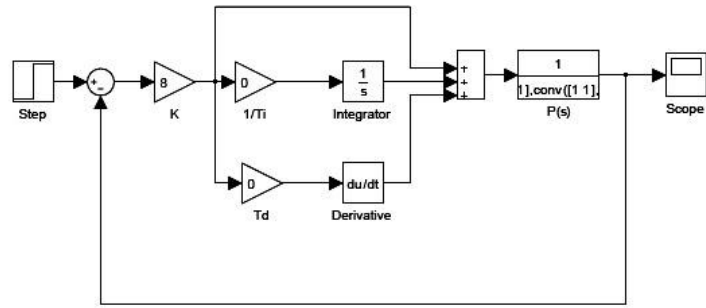
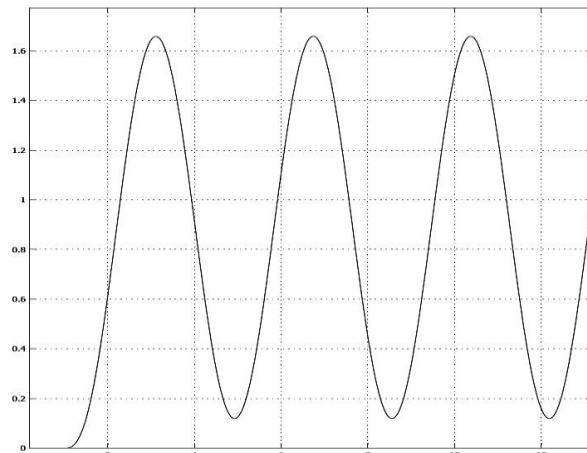Figure 1: A SIMULINK setup for Example 1

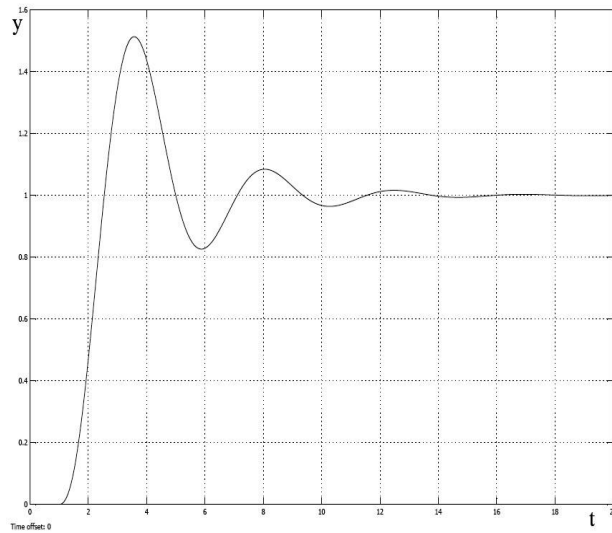Figure 2: Oscillation captured from scope

Figure 3: Step response from PID values given by ZNFD method

The ZNFD method could be explained using a Nyquist diagram in Figure 4. The diagram shows how a point x on the curve is moved related to the P, I , and D terms. Using the P term alone, x could be moved in radial direction only. The I and D terms help provide more freedom to move perpendicular to the radius. It can be shown that by using ZNFD method, the critical point (-$1/K_u$, 0) is moved to the point -0.6 – 0.28i. The distance of this point to the critical point is 0.5. So the sensitivity peak is at least 2. This explains the high overshoot in the step response.
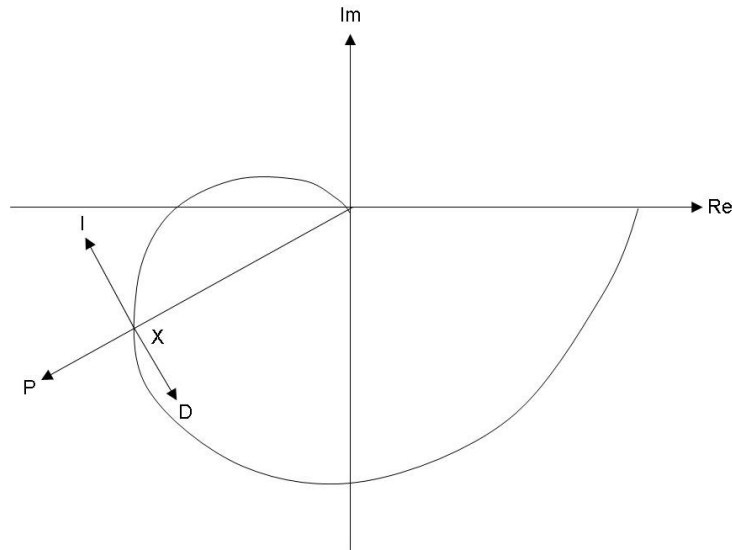

Figure 4: How a point on Nyquist curve is moved with PID control

Automatic Tuning

As simple as it sounds, the ZNFD method may be difficult to perform in certain industrial applications. It is problematic to adjust the gain until the close-loop system oscillates. A little beyond that results in instability. Automatic tuning scheme exploits some component that could make the system oscillate, but does not become unstable. A relay is one such component.

Example 2: In Figure 5, we put a relay in place of the PID controller. The relay output swings between $\pm$ 1. Using the same plant (10), the simulated response in Figure 6 shows the plant output oscillates, with the same period as in Figure 2. We see that the oscillation is automatic and the magnitude of plant output is related to the relay output. We can keep things under control, so this scheme is suitable for PID autotuning.
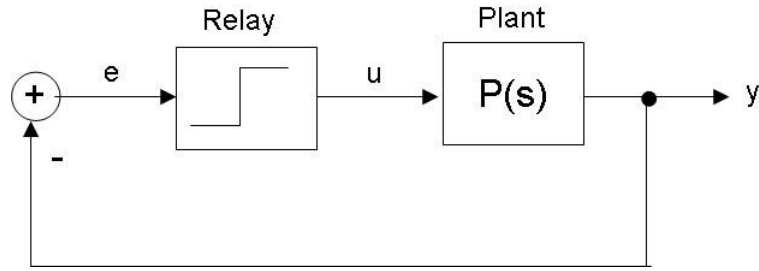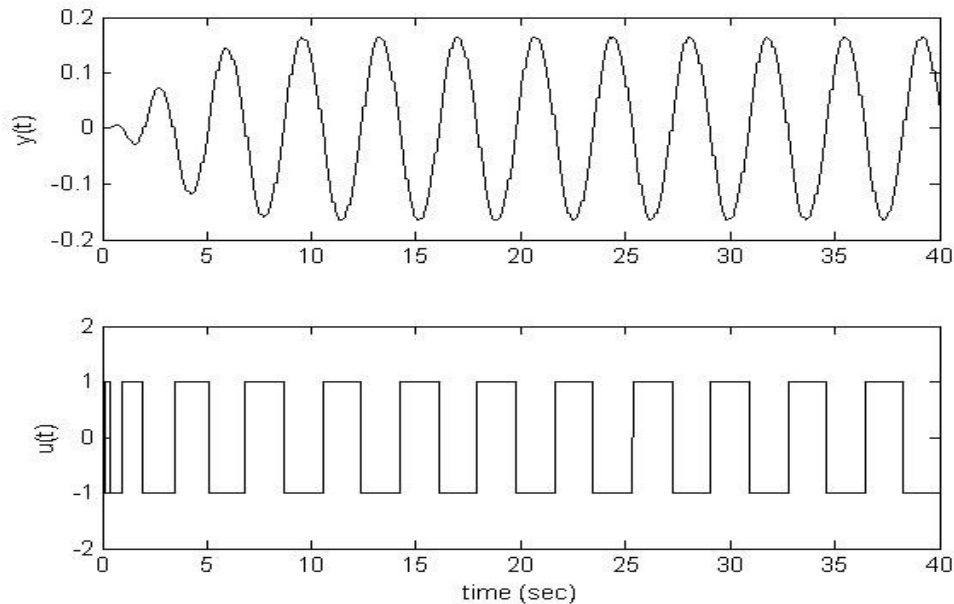
Figure 5: A relay feedback diagram

Figure 6: Oscillatory response from the relay feedback

Since a relay is a nonlinear element, we have to find some linear representation using some math tool. A suitable approach is to use "describing function." Using such technique, the relay is replaced by a dependent gain $N(a)$, where a is the size of relay input. And the condition for oscillation is

$$N(a)P(i\omega) = -1 \qquad (11)$$

One can easily check this condition graphically by plotting $-1/N(a)$ on the Nyquist plot. If the graph of $-1/N(a)$ and $P(i\omega)$ intersects, that means the relay feedback should oscillate, where the magnitude and frequency of oscillation equals the values at that intersection point. So, if we can measure the magnitude and frequency of oscillation, we can determine the intersection point.

In case the relay is a non-hysteresis type, its describing function is

$$N(a) = \frac{4d}{\pi a} \qquad (12)$$

where d is the magnitude of relay signal and a is the magnitude of oscillating output. Note that (12) is a real function, so the system should oscillate if the Nyquist curve intersects the negative real axis. Hence, using relay feedback could help us find the intersection point between the Nyquist curve and the negative real axis, which is the same point acquired using the manual ZNFD method.

From Figure 6, we can measure $a \approx 0.15$ and $d = 1$. From (12), it can be computed that the Nyquist curve must intersect the negative real axis at $-1/N(a)$, or -0.118, which is close to the value found in Example 1, which is -1/8 or -0.125.

Autotuning Implementation

To transform all these to C code, the components needed are a relay, and a function to read the magnitude and frequency of oscillation. A relay could be implemented in software as follows

```
#define        RELAYOUT    204     // change this to your desired output value

void    relay( void)
{
        int  e;
        e = read_input( );              // read from specified input source
        if (e < 0)       out_dac(RELAYOUT); // send output. Note opposite phase
        else out_dac( -RELAYOUT);
}
```

And functions to detect a and Tu and compute magnitude and frequency of oscillation

```
/************ Global Variables ********************************/
/*These two variables are what we want to find  */
double p;               // magnitude of P
double w;               // frequency of P

double  tu;             // oscillation period  (w = 1/tu)
double d;               // relay amplitude. This is constant for a particular relay
```

```
double a;              // peak process output amplitude.
double yold;           // keep previous process output
double ts;             //sampling period
int i=0;               // a counter to keep the number of iterations between two peaks
/*********************************************************/

void detect_a_tu(void)         / this must be a timer ISR running each ts seconds
{
        double y;                      // use to keep process output each sampling period
        y = read_input( );             // read input from specified channel
        if (y>a) a = y;                // compare new input with a, if greater keep it as new a
        if (yold<0 && y>=0)   {        // detect zero crossing
                tu=i*ts;
                i=0;
        }
        yold=y;
        i++;
}

void compute_pw(void)    // run this after we get values for a and tu
{
        double Na;
        Na = (4*d)/(pi*a);
        p = -1/Na;       // gain of P(jw) at point of intersection
        w = 2*pi/tu;     // frequency in rad/s
}
```

The functions are straightforward. detect_a_tu( ) has to be implemented as a timer ISR. The magnitude a is detected by comparing the new read value with the previous largest value, and keep the larger. The period tu is found by detecting two zero crossings and compute the time between them. Then the function compute_pw( ) just compute the magnitude and frequency of the point of intersection between Nyquist plot and the negative real axis. After this point is found, we can determine where to move it to give a good gain and phase margins. The values in Table 1 can be used.

Reference

[1] K.J. Astrom and T.Hagglund. PID Controllers, 2nd ed., Instrument Society of America, 1995.

Revision History

July 2009:   First appearance of this document

June 2011:   Fix typo in Verilog code (page 4). Correct variable name should be u_out not u_in

May 2014:   In regard to a valid reader comment, a footnote is added on page 2 to indicate that the PID parameters are indeed affected by sampling period. Here I choose to omit that for simplicity of discussion. There are also variations in the Z-domain form of the PID controllers. In future article, we 'd investigate these issues in more detail.

Visit   http://controlsystemslab.com/category/articles/control-engineering/pid/   for recent updates.