Chapter 10 – Digital PID

Goals

- To show how PID control can be implemented in a digital computer program
- To deliver a template for a PID controller that you can implement yourself on the micro-processor platform of your choice
- To consider practical constraints on digital control
- To present a quick description of pulse-width modulation

Introduction

In most books on digital control one is quickly introduced to the *z-transform* and the *z-plane*, as opposed to the s-plane, which we have used primarily for the root locus in designing PID controllers. I have chosen not to use the z-transform here to explain digital controls. Rather the explanation here is more practical. It explains digital controls more directly, discusses how to implement a digital PID, and describes several phenomena to be aware of when implementing digital control systems.

Analog vs. digital world

With digital control we are interfacing a computer (micro-controller) with the outside (analog) world. Figure 10.1 shows a control loop with the boundary demarcated between the two different regimes.

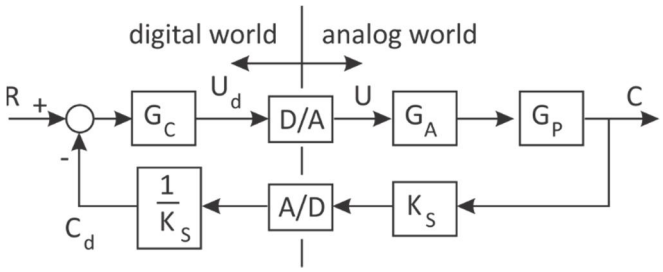


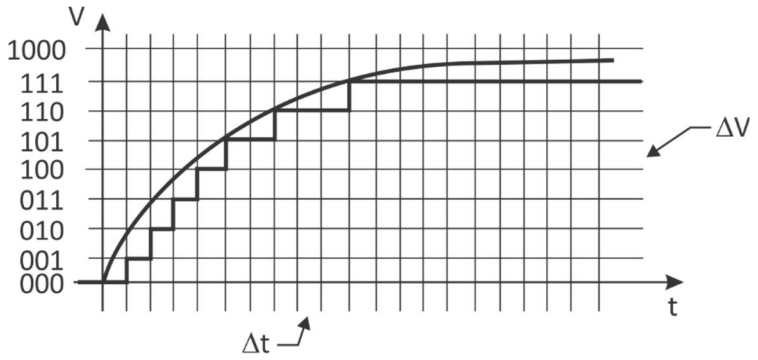
Figure 10.1 – Control loop, showing boundary between the outside (analog) world and the digital world inside the micro-processor

The actuator, plant, and sensor are real devices in the real (analog) world. The actual value being controlled (C) is sensed by a sensor that, here, is modelled as a simple gain. Usually the sensor produces an electrical signal—often a voltage—which then is sent into the computer via an analog-to-digital converter (A/D). There the original signal, whether it be temperature, pressure, position, velocity, or whatever, is recovered in digital form inside the computer by dividing the representative voltage through by the sensor gain. This is shown as C_d in the diagram, with the "d" indicating that it is the digital representation of the real, sensed variable C. At this point the computer algorithm for the comparison with the desired value (R) can take place, the error determined in digital format, and the controller algorithm executed to come up with U_d , the digital version of the analog value to be sent out to the actuator. This is converted by the D/A converter into a real signal (U, usually a voltage), which starts the actuation chain.

Of course the A/D and D/A converters should have a value of 1, so that the voltage or current sensed is converted accurately into and out of the computer, respectively. Figure 10.1 also shows why, with computer-controlled loops, they generally become unity-feedback loops. What can happen is that one of the converters can malfunction, and it ceases to have 1 as its transfer function.

The analog world is a smear; values can change continuously; the size of the smallest change from one value to another is infinitesimal. The digital world, on the other hand, is a collection of values that are used to model the outside world discreetly. Digital bits are either off or on. To model a changing voltage, for example, a group of bits is used to model an analog voltage. These bits are either off or on, and that status represents the level at which the analog signal has arrived. For example, if a sensing device only recorded increments of 1 volt, to the micro-processor, there would be no difference between 3.2 volts, 3.5 volts, and 3.9 volts. All would be 3.0 volts.

A more detailed example of this: Take the first-order response shown in Figure 10.2. The t-axis is segmented into instances, each Δt, the scan rate, apart. A scheme can be devised to model the level arrived at using three bits, for example. Three bits can be used to represent eight different levels of voltage. Counting in the binary number system from 0 to 7, these values are 000, 001, 010, 011, 100, 101, 110, 111. An analog-to-digital (A/D) converter senses the voltage coming in and turns on the bits lower than the voltage that it senses.



0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t9	t > t10
000	001	010	011	100	101	101	110	110	110	111	111

Figure 10.2 – Time slices and voltage-level slices

As can be seen with a careful comparison, at the time instances shown, the bits will be turned on to represent that the real, analog voltage has arrived at the voltage level that corresponds to a particular bit. Thus the sensed voltage has the values represented in the table; this is graphically shown as the stairstep curve shape shown under the analog voltage curve.

Of course 3 bits does not give a good resolution of the analog curve. Three binary bits gives $2^3 = 8$ possible values. Typically more bits are used to give a higher resolution of analog values. Ten, for instance, would give $2^{10} = 1024$ different levels, and it is easy to see that 1024 levels would allow the curve to be represented with much better fidelity. Three bits were used here simply to show the nature

of the problem of making a digital representation of an analog curve. Here the three bits would be mapped to a range, where 111 represented the maximum value of voltage measured by the sensor. To capture values above this range, yet another bit would be needed, as shown at the top of the 3-digit values.

Of course a continuously changing signal can be better captured by making the time increments smaller and smaller and by making the resolution of the bits representing the signal ever finer. And, indeed, over the decades-long history of digital control, this is exactly what has happened. Micro-controllers have become ever faster and more and more bits have been assigned to capture analog quantities. This has always been accompanied by greater cost, but as digital-control technology has progressed, these costs have dropped dramatically too. Thus we have today the happy circumstance that for a great many applications in the control of physical systems, one is not even aware of this discretization of time and of the digital representation of the quantity either being measured or controlled.

The structure of digital control

Digital control is implemented in specialized micro-controllers developed for hardened industrial use. With the coming of the electronic hobbyist movement, small, inexpensive micro-processors have become available that are fast enough to provide adequate PID control for many mechanical, thermal, or fluid systems. These micro-controllers are designed to be user-friendly, even for novice programmers. Thus you can program your own PID. Here's how.

The control program in the micro-controller often runs in a fixed time-step loop. This loop is very fast, especially compared with the reaction times of most mechanical systems. Typical loops execute once every 1-10 milliseconds. Generally this is adequate for controlling most mechanical systems. A problem, called *latency*, does exist if the loop is too slow for the process it's controlling. This is discussed at the end of this chapter along with other problems that can accompany digital control. Digital control has many advantages over analog control (control with standard electrical components and op-amps). But analog control still beats digital control in that one critical area of speed. Analog controllers respond virtually with no delay, and that is necessary for very fast systems.

As was stated above, digital controllers run a looping computer program that provides the control. A rough layout of this computer program's structure is given in Figure 10.3. These three steps are carried out, one after the other, whenever the controller is running in automatic mode. With this broad structure in mind, let's look at the details of writing a PID controller within this structure.

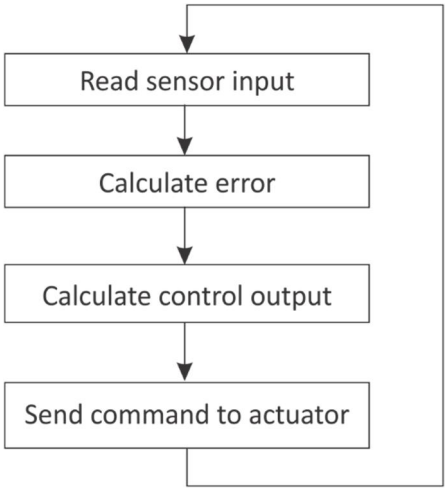


Figure 10.3 – Digital control loop structure

Implementation of PID control in a computer program

The following description of a PID controller is general and is not computer-language specific. This should allow you to implement a controller, whether it be in C, Java, Python, Matlab m-code, Arduino code, or whatever computer language you wish. The code is pseudo-code–that is, code that just gives the ideas that make up the implementation. The pseudo-code is interspersed with explanations. Then at the end the entire pseudo-code controller is given as it would appear in a micro-controller. You should be able to use this as a template for your implementation in whatever computer language.

Some variables are needed:

`Vs`	the voltage coming from the sensor
`VsPrev`	the voltage from the sensor in the previous pass through the loop
`KSens`	Sensor gain (volts/physical quantity)
`BSens`	Sensor bias (units: physical quantity)
`KP`	the controller's proportional gain
`KI`	the controller's integral gain
`KD`	the controller's derivative gain
`dt`	the controller scan rate
`err`	the error (input to the controller)
`errPrev`	the error on the previous pass through the loop
`errDot`	d(err)/dt, the change in the error with time
`errAccum`	the accumulated area under the error curve
`r`	the desired value
`cd`	the digital version of the actual value (c)
`pAction`	the proportional action from the controller
`iAction`	the intergral action from the controller
`dAction`	the derivative action from the controller
`u`	the action out of the controller in % of actuator capability
`Vu`	the voltage out from the controller to the actuator

VRange	the voltage range of the actuator
AUTO	a constant set up to denote that the loop is in automatic mode

The following code reads the sensor and calculates the proportional action of the controller. Note that the sensor has a gain but a *bias* as well. Bias is an additive constant, because the voltage from the sensor may not be 0 when the physical quantity measured is 0.

```
Vs = getV();
cd = Vs / KSens;
cd = cd + BSens;
err = r – cd;
pAction = err * KP;
```

For the other two control actions—I and D—one needs to calculate `errAccum` and `errDot`. `errDot` is just the change in the error with time. We need the error from the previous pass through the loop to calculate this. We also need the *scan rate* of the controller, that is the time between consecutive passes through the loop.

```
errDot = (err – errPrev)/dt;
dAction = errDot * KD;
```

`errAccum` is the accumulated area under the error curve since the controller was put into service. It is accumulated slice by slice with each time step. We'll use the Trapezoidal Rule to calculate the slice of error under the error curve between the previous and the present time steps.

```
errAccum = errAccum + (err + errPrev)/2*dt;
iAction = errAccum * KI;
```

Now we have all the actions calculated. The action out of the controller is simply the sum of the three actions.

```
u = pAction + iAction + dAction;
Vu = u * VuRange;
putActuator(Vu);
```

Prior to restarting the loop, we need to preserve the previous values for next time.

```
VsPrev = Vs;
errPrev = err;
```

Now the loop is finished and goes back to the beginning for another pass.

We need to set the loop up properly to execute until it's commanded to stop. A command to stop is a command for the loop to go into manual mode. Thus during each pass through the loop, the operation mode (automatic or manual) needs to be checked to see if we need to exit the loop. We need also to set `errAccum` to 0 prior to entering the loop. Thus the loop should be bracketed by the statements

```
errAccum = 0;
while (getAutoManStatus() == AUTO) {

     Put loop here

};
```

Thus the entire loop is

```
errAccum = 0;
while (getAutoManStatus() == AUTO) {
     Vs = getVs();
     cd = Vs / KSens;
     cd = cd + BSens;
     err = r - cd;
     pAction = err * KP;
     errDot = (err - errPrev)/dt;
     dAction = errDot * KD;
     errAccum = errAccum + (err + errPrev)/2*dt;
     iAction = errAccum * KI;
      u = pAction + iAction + dAction;
      Vu = u * VuRange;
      putActuator(Vu);
      VsPrev = Vs;
      errPrev = err;
};
```

Problems encountered with digital control

As was explained at the first of this chapter, a problem inherent to digital control is that the scan rate of the controller can be too slow for the process it is trying to control. Thermal processes usually are slow, so this problem is not encountered there. But hydraulic-actuation processes can be very quick, and the controller must keep pace with the changing nature of the process. If the scan rate of the controller is too slow, up and down changes in the process can be missed in between sampling instances of the sensor. Figure 10.4 illustrates a possible scenario of this type.

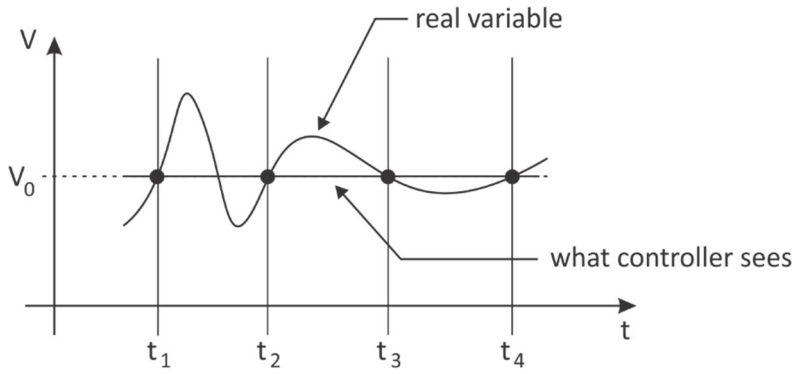


Figure 10.4 - Slow sampling rate misses changes in measured variable

In this case, even though the variable being sampled is changing up and down, it looks as if it is at a steady level because the sampling instances just happen to catch it as it crosses a steady value. An instrument panel displaying the value of V would thus show no change in V, even though V is actually experiencing quite an active excursion from V_0 . Though this is somewhat of a contrived example, it is easy to see that the first peak value would be missed or measured to be lower unless the sampling instance just happened to coincide with the occurrence of the peak. Obviously the solution to this problem is to reduce Δt , the scan rate of the micro-processor.

Pulse-Width Modulation

In the digital world it is easy to turn things on and off, harder to *modulate* them, i.e. to adjust them to values between on and off. A digital board will usually work with standard voltage levels—0 to 5 volts, -5 to +5 volts, -10 to +10 volts. So in a 0 to 5 volt system, you can produce an ON (5 volts) by telling the board controller to write put that system voltage on a particular output pin on that board. But if instead you want a voltage level of, say 3.2 volts, it is not as easy to produce this as you might think.

The standard way of doing this is to use *pulse-width modulation* (PWM). An output signal is produced that is a square wave, a sequence of ONs and OFFs as shown in Figure 10.5. This figure shows three different PWM signals, all for a processor that works with 0-5 volts as its output.

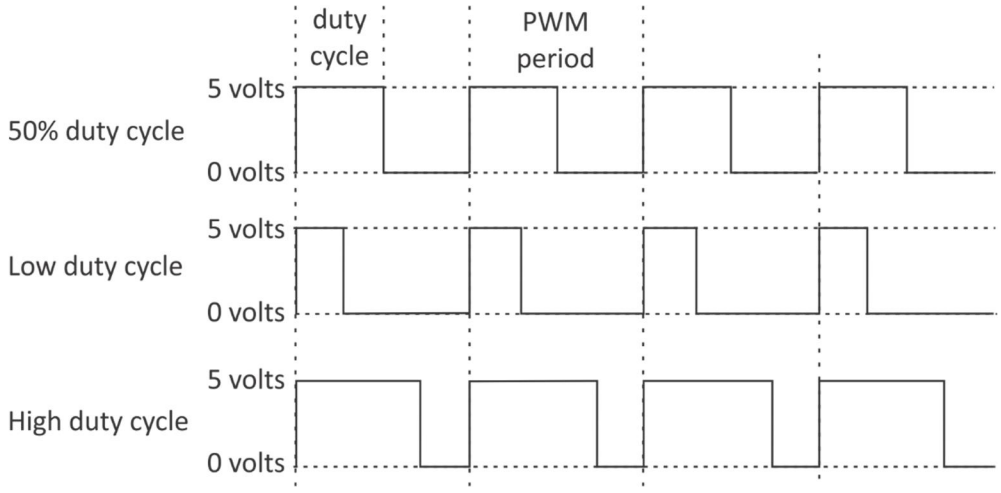


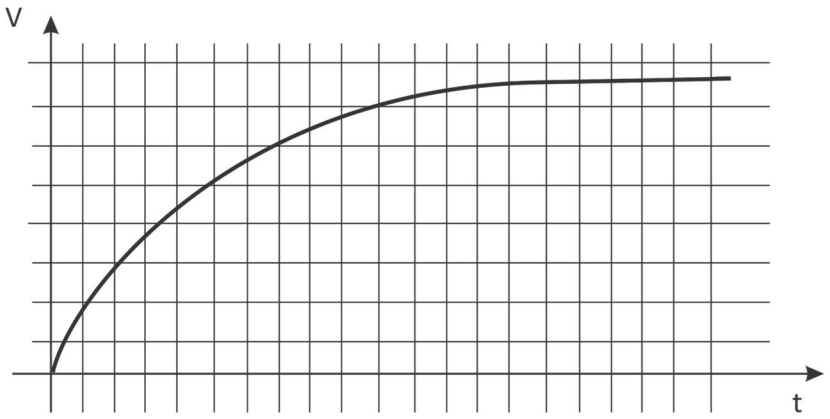
Figure 10.5 - Pulse-width modulated signals

The signal is a sequence of pulses. The frequency of the pulses is fixed and depends on the microprocessor being used to generate it. The faster the microprocessor, the faster the PWM frequency can be. The top signal exhibits a *duty cycle* of 50%. This means that the output voltage is on 50% of the PWM period and off 50% of the period. This would correspond to an equivalent output voltage of 2.5 volts. The middle signal shows a signal with a duty cycle less than 50%. This would correspond with an output voltage of less that 2.5 volts. The bottom figure shows a signal with a duty cycle of more than 50%, so an output voltage of somewhere between 2.5 and 5 volts. To get 5 volts out, the duty cycle needs to be 100%. For 0 volts, the duty cycle is 0%. The width of the pulse can be *modulated*, that is adjusted, to give any output voltage between 0 and 5 volts.

Obviously the device that this pulsed signal is fed into will "feel" the pulses. An LED will blink on and off very quickly. A motor will be subjected to this pulsed signal. A valve will be commanded to open and close very rapidly. But physical reality, especially mechanical parts, cannot react so rapidly due to their inertia. Thus they behave as if the signal they receive actually is a steady analog voltage with the value that corresponds to the modulated pulse width. An LED which can switch on and off to follow the pulse is perceived to be brighter the higher the duty cycle. The human eye perceives the modulation of pulses to an LED to be brightening and dimming of the LED.

Problems

10.1 Take the digital representation of the curve of Figure 10.1 and change the algorithm so that the voltage level represented by a measurement is increased automatically by half the ΔV to the next voltage level. Draw on the chart below the digital representation of the curve with this modification in the algorithm.



Is this an improvement over the original algorithm, illustrated in Figure 10.1?

10.2 Sometimes a user wants to flush out the accumulated error used in the integral-action calculation and start over from zero. Let's modify the existing loop to allow this. Define a logical variable named `reset`. Right after the sensor voltage is gotten, use a function `getReset()` to set `reset`. Then check this with an if statement and reset the accumulator to 0.0 if `reset` is true. To test a logical variable, the syntax is `if(a) {…};` .

10.2 A micro-controller can only output a limited range of voltage. That is, `VuMin < Vu < VuMax`. Modify your loop to include a check of output voltage and set up statements to enforce these limits.

10.3 Anti-wind-up for integral control is explained in Chapter 9. But briefly, the way it works is that, if the controller output is saturated, then the accumulation of error is suspended. Modify the PID algorithm given in this chapter to include anti-wind-up.

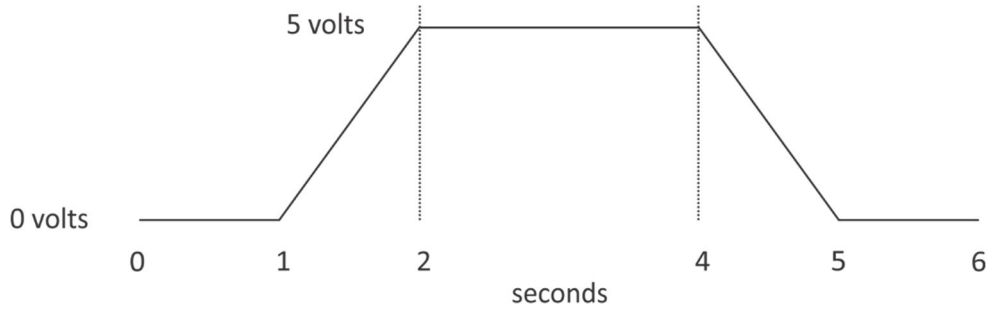
10.4 Implement manual-mode operation of the controller. Include a check each time through the loop to `getMode()`. Store this into a variable called `mode`. Then let there be two constants `MAN` and

`AUTO`, so that the algorithm goes into the right place to either have `Vu` calculated by the controller or to have it increased, decreased, or left alone manually. Implement this by having a call to `getIncr()` which returns either 1, 0, or -1. Then increase or decrease `Vu` by 1% of the range either up or down.

You will need to have an outer loop that checks the controller status with `getCntlStatus()`. This will return either the value `ON` or `OFF`. If it returns `OFF`, then the loop is exited.

10.5 Whatever `Vu` is, it is some percentage of the range of output of the controller. Use `VuMin` and `VuMax` to calculate `UPercent`. Then display this percent output with a call to `displayUPercent()`.

10.6 Let's say that we have a motion-control application where we want to move a robot arm from point A to point B. The motor that drives the arm receives a PWM signal from a microprocessor for the motion. The signal is generated with a period of 0.1 seconds. To get from point A to point B, the signal has the time profile shown below.



Make a plot of the PWM signal for this cycle of motion, showing the signal from 0 to 6 seconds.