PID Without a PhD[1]

Tim Wescott

PID (proportional, integral, derivative) control is not as complicated as it sounds. Follow these simple implementation steps for quick results.

At work, I am one of three designated "servo guys," and the only one who implements control loops in software. As a result, I often have occasion to design digital control loops for various projects. I have found that while there certainly are control problems that require all the expertise I can bring to bear, a great number of control problems can be solved with simple controllers, without resorting to any control theory at all. This article will tell you how to implement and tune a simple controller without getting into heavy mathematics and without requiring you to learn any control theory. The technique used to tune the controller is a tried and true method that can be applied to almost any control problem with success.

PID control

The PID controller has been in use for over a century in various forms. It has enjoyed popularity as a purely mechanical device, as a pneumatic device, and as an electronic device. The digital PID controller using a microprocessor has recently come into its own in industry. As you will see, it is a straightforward task to embed a PID controller into your code.

PID stands for "proportional, integral, derivative." These three terms describe the basic elements of a PID controller. Each of these elements performs a different task and has a different effect on the functioning of a system.

In a typical PID controller these elements are driven by a combination of the system command and the feedback signal from the object that is being controlled (usually referred to as the "plant"). Their outputs are added together to form the system output.

Figure 1 shows a block diagram of a basic PID controller. In this case the derivative element is being driven only from plant feedback. The plant feedback is subtracted from the command signal to generate an error. This error signal drives the proportional and integral elements. The resulting signals are added together and used to drive the plant. I haven't described what these elements do yet-we'll get to that later. I've included an alternate placement for the proportional element (dotted lines)-this can be a better location for the proportional element, depending on how you want the system to respond to commands.

---

[1] Taken from the EE Times embedded controller website (http://www.embedded.com/2000/0010/0010feat3.htm) on 10/23/2009

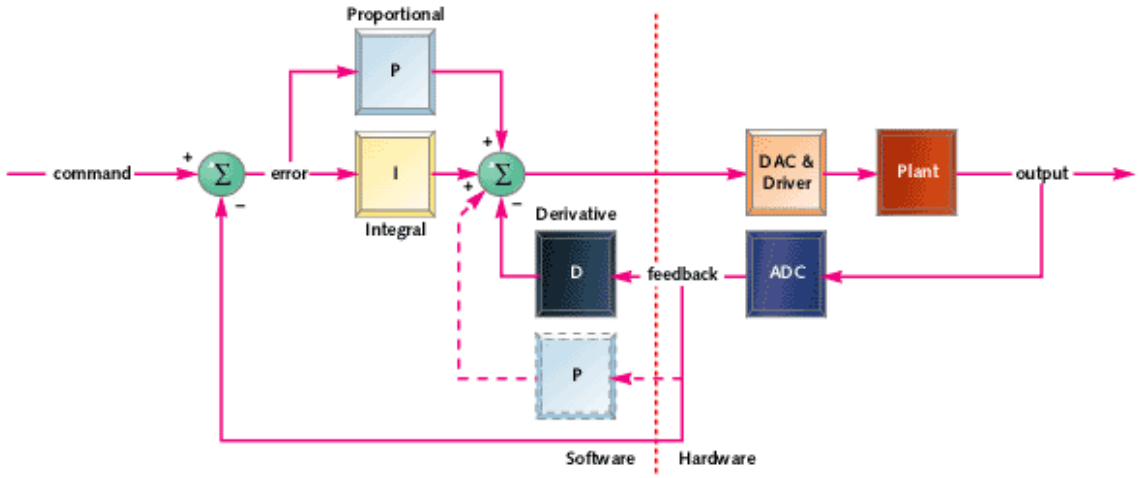


Figure 1: A basic PID controller.

Sample plants

In order to discuss this subject with any sense of reality we need some example systems. I'll use three example plants throughout this article, and show the effects of applying the various controllers to them:

- A motor driving a gear train
- A precision positioning system
- A thermal system

Each of these systems has different characteristics and each one requires a different control strategy to get the best performance.

Motor and gear

The first example plant is a motor driving a gear train, with the output position of the gear train being monitored by a potentiometer or some other position reading device. You might see this kind of mechanism driving a carriage on a printer, or a throttle mechanism in an automobile cruise control system, or almost any other moderately precise position controller. Figure 2 shows a diagram of such a system. The motor is driven by a voltage that is commanded by software. The motor output is geared down to drive the actual mechanism. The position of this final drive is measured by the potentiometer.

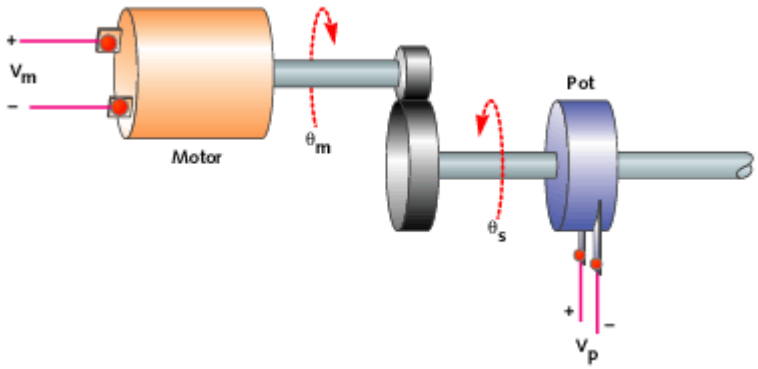
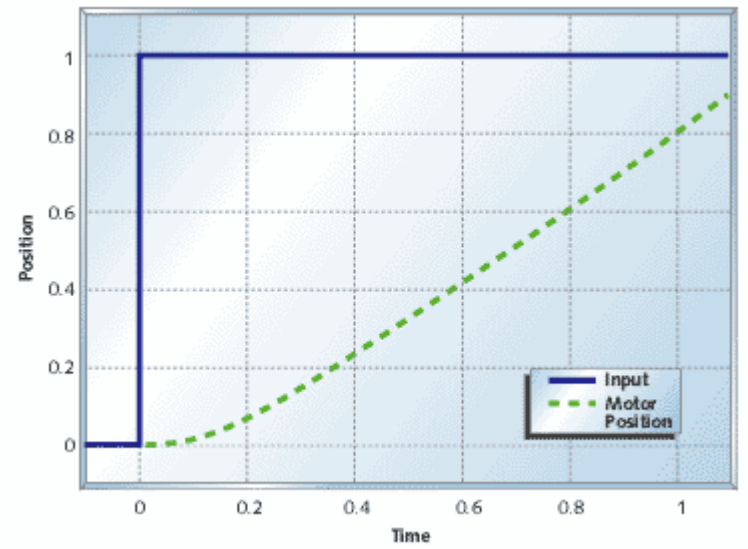


Figure 2: A voltage driven motor and gear train.

A DC motor driven by a voltage wants to go at a constant speed that is proportional to the applied voltage. Usually the motor armature has some resistance that limits its ability to accelerate, so the motor will have some delay between the change in input voltage and the resulting change in speed. The gear train takes the movement of the motor and multiplies it by a constant. Finally, the potentiometer measures the position of the output shaft.

Figure 3 shows the step response of the motor and gear combination. I'm using a time constant value of t0 = 0.2s. The step response of a system is just the behavior of the output in response to an input that goes from zero to some constant value at time t = 0. Since we're dealing with fairly generic examples here I've shown the step response as a fraction of full scale, so it goes to 1. Figure 3 shows the step input and the motor response. The response of the motor starts out slowly due to the time constant, but once that is out of the way the motor position ramps at a constant velocity.

Figure 3: Motor and gear position vs. time.



Precision actuator

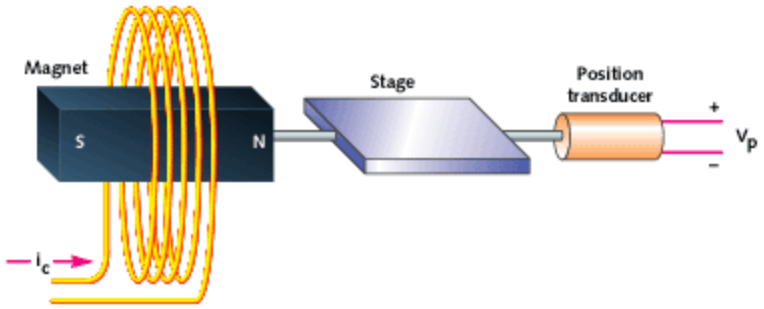
It is sometimes necessary to control the position of something very precisely. A precise positioning system can be built using a freely moving mechanical stage, a speaker coil (a coil and magnet arrangement), and a non-contact position transducer.

You might expect to see this sort of mechanism stabilizing an element of an optical system, or locating some other piece of equipment or sensor. Figure 4 shows such a system. Software commands the current in the coil. This current sets up a magnetic field that exerts a force on the magnet. The magnet is attached to the stage, which moves with an acceleration proportional to the coil current. Finally, the stage position is monitored by a non-contact position transducer.

With this arrangement, the force on the magnet is independent of the stage motion. Fortunately this isolates the stage from external effects. Unfortunately the resulting system is very "slippery," and can be a challenge to control. In addition, the

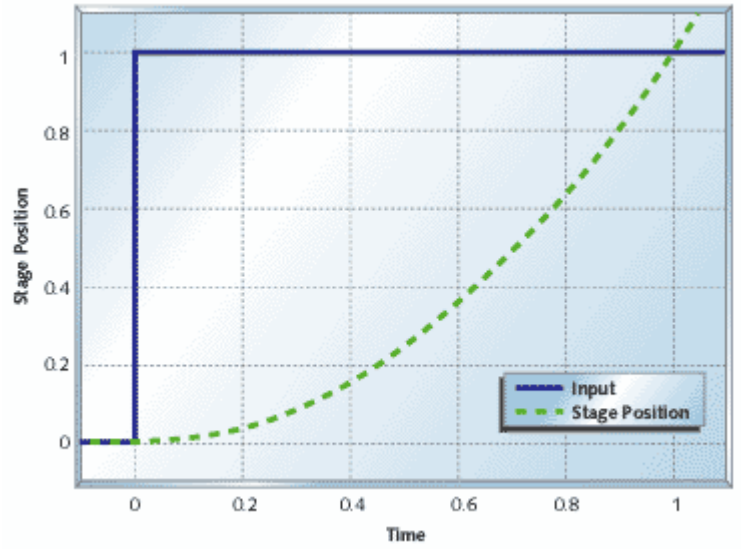
electrical requirements to build a good current-output amplifier and non-contact transducer interface can be challenging. You can expect that if you are doing a project like this you are a member of a fairly talented team (or you're working on a short-lived project).

Figure 4: A precision actuator.



The equations of motion for this system are fairly simple. The force on the stage is proportional to the drive command alone, so the acceleration of the system is exactly proportional to the drive. The step response of this system by itself is a parabola, as shown in Figure 5. As we will see later this makes the control problem more challenging because of the sluggishness with which the stage starts moving, and its enthusiasm to keep moving once it gets going.

Figure 5: Precision actuator position vs. time.



Temperature control

The third example plant I'll use is a heater. Figure 6 shows a diagram of an example system. The vessel is heated by an electric heater, and the temperature of its contents is sensed by a temperature-sensing device.

Thermal systems tend to have very complex responses. I'm going to ignore quite a bit of detail and give a very approximate model. Unless your performance requirements are severe, an accurate model isn't necessary.

Figure 7 shows the step response of the system to a change in Vd. I've used time constants of t1 = 0.1s and t2 = 0.3s. The response tends to settle out to a constant

temperature for a given drive, but it can take a great deal of time doing it. Also, without lots of insulation, thermal systems tend to be very sensitive to outside effects. This effect is not shown in the figure, but we'll be investigating it later in the article.

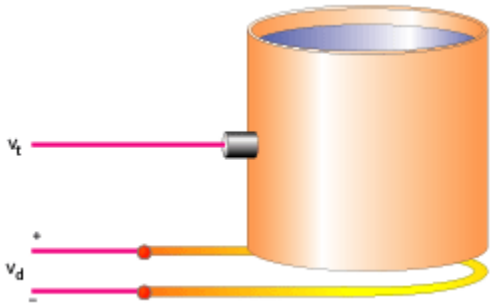


Figure 6: A heater.

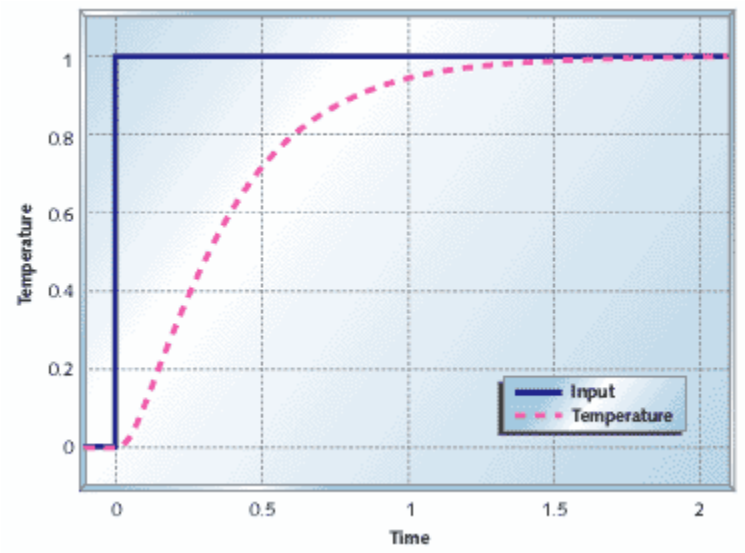


Figure 7: Heater temperature vs. time.

Controllers

The elements of a PID controller presented here either take their input from the measured plant output or from the error signal, which is the difference between the plant output and the system command. I'm going to write the control code using floating point to keep implementation details out of the discussion. It's up to you to adapt this if you are going to implement your controller with integer or other fixed-point arithmetic.

I'm going to assume a function call as shown below. As the discussion evolves, you'll see how the data structure and the internals of the function shapes up.

```
double UpdatePID(SPid * pid,
double error, double position)
{
.
.
.
}
```

The reason I pass the error to the PID update routine instead of passing the command is that sometimes you want to play tricks with the error. Leaving out the error calculation in the main code makes the application of the PID more universal. This function will get used like this:

```
.
.
position = ReadPlantADC();
drive = UpdatePID(&plantPID,
plantCommand - position,
position);
DrivePlantDAC(drive);
.
.
```

Proportional

Proportional control is the easiest feedback control to implement, and simple proportional control is probably the most common kind of control loop. A proportional controller is just the error signal multiplied by a constant and fed out to the drive. The proportional term gets calculated with the following code:

```
double pTerm;
.
.
.
pTerm = pid->pGain * error;
.
.
.
return pTerm;
```

Figure 8 shows what happens when you add proportional feedback to the motor and gear system. For small gains (kp = 1) the motor goes to the correct target, but it does so quite slowly. Increasing the gain (kp = 2) speeds up the response to a point. Beyond that point (kp = 5, kp = 10) the motor starts out faster, but it overshoots the target. In the end the system doesn't settle out any quicker than it would have with lower gain, but there is more overshoot. If we kept increasing the gain we would eventually reach a point where the system just oscillated around the target and never settled out-the system would be unstable.

The motor and gear start to overshoot with high gains because of the delay in the motor response. If you look back at Figure 2, you can see that the motor position doesn't start ramping up immediately. This delay, plus high feedback gain, is what causes the overshoot seen in Figure 8. Figure 9 shows the response of the precision actuator with proportional feedback only. Proportional control alone obviously doesn't help this system. There is so much delay in the plant that no matter how low the

gain is, the system will oscillate. As the gain is increased, the frequency of the output will increase but the system just won't settle.

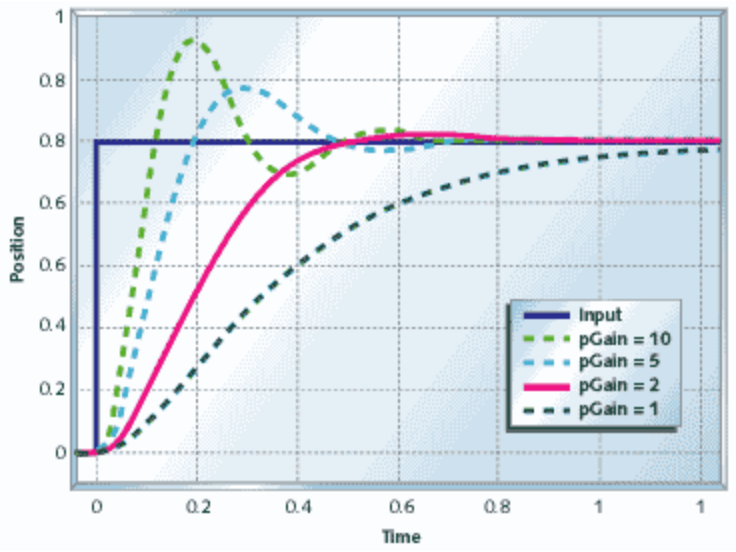


Figure 8: Motor and gear with proportional feedback position vs. time.

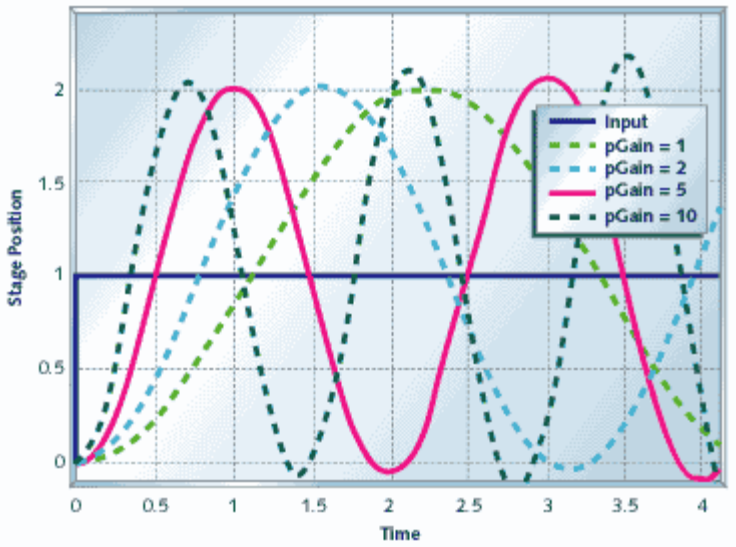
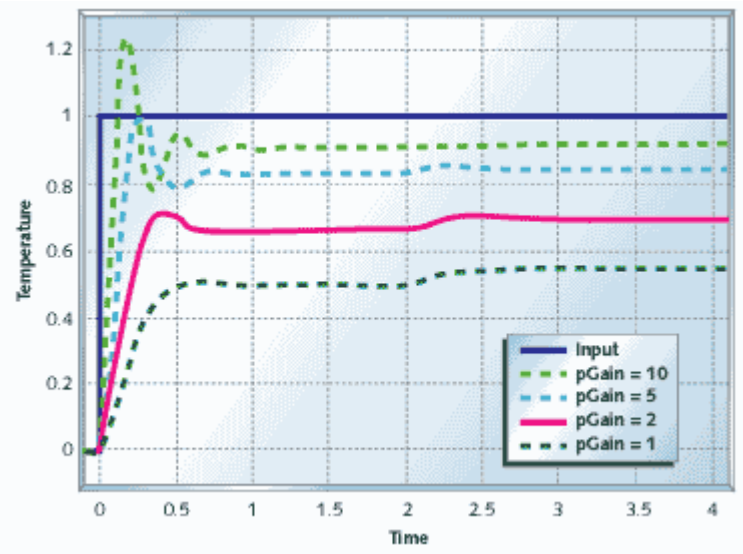


Figure 9: Precision actuator with proportional feedback vs. time.

Figure 10 shows what happens when you use pure proportional feedback with the temperature controller. I'm showing the system response with a disturbance due to a change in ambient temperature at t = 2s. Even without the disturbance you can see that proportional control doesn't get the temperature to the desired setting. Increasing the gain helps, but even with kp = 10 the output is still below target, and you are starting to see a strong overshoot that continues to travel back and forth (this is called ringing).

As the previous examples show, a proportional controller alone can be useful for some things, but it doesn't always help. Plants that have too much delay, like the precision actuator, can't be stabilized with proportional control. Some plants, like the temperature controller, cannot be brought to the desired set point. Plants like the motor and gear combination may work, but they may need to be driven faster than is possible with proportional control alone. To solve these control problems you need to add integral or differential control or both.

Figure 10: Temperature controller with proportional feedback. The kink at time = 2 is from the external disturbance.



Integral

Integral control is used to add long-term precision to a control loop. It is almost always used in conjunction with proportional control.

The code to implement an integrator is shown below. The integrator state, iState is the sum of all the preceding inputs. The parameters iMin and iMax are the minimum and maximum allowable integrator state values.

```
double iTerm;
.
.
.
// calculate the integral state
// with appropriate limiting
pid->iState += error;
if (pid->iState > pid->iMax)
pid->iState =
pid->iMax;
else if (pid->iState
<
pid->
iMin)
pid->iState = pid->iMin;
iTerm = pid->iGain * iState;
// calculate the integral term
.
.
.
```

Integral control by itself usually decreases stability, or destroys it altogether. Figure 11 shows the motor and gear with pure integral control (pGain = 0). The system doesn't settle. Like the precision actuator with proportional control, the motor and

gear system with integral control alone will oscillate with bigger and bigger swings until something hits a limit. (Hopefully the limit isn't breakable.)

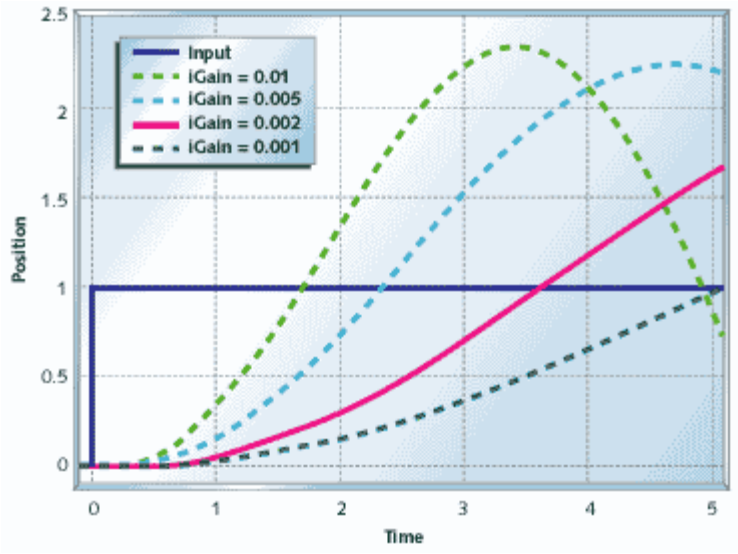


Figure 11: Motor and gear with pure integral control.

Figure 12 shows the temperature control system with pure integral control. This system takes a lot longer to settle out than the same plant with proportional control (see Figure 10), but notice that when it does settle out, it settles out to the target value-even with the disturbance added in. If your problem at hand doesn't require fast settling, this might be a workable system.

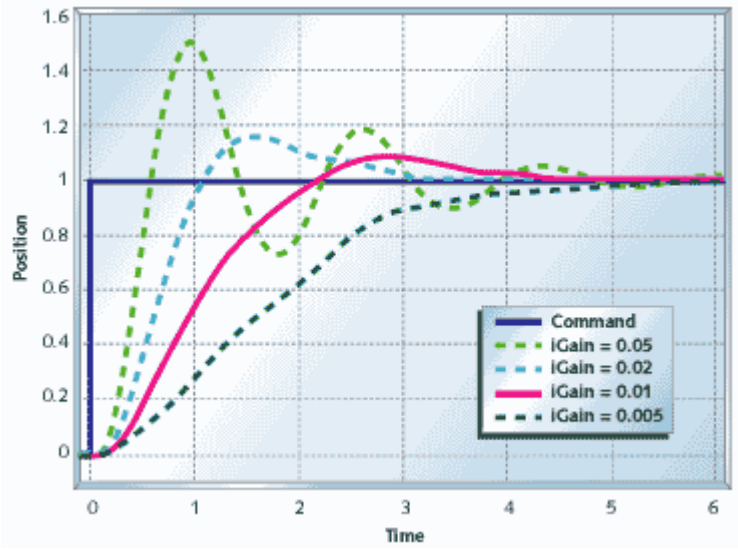


Figure 12: Temperature control system with integral control. The kink at time = 2 is from the external disturbance.

Figure 12 shows why we use an integral term. The integrator state "remembers" all that has gone on before, which is what allows the controller to cancel out any long term errors in the output. This same memory also contributes to instability-the controller is always responding too late, after the plant has gotten up speed. To stabilize the two previous systems, you need a little bit of their present value, which you get from a proportional term.

Figure 13 shows the motor and gear with proportional and integral (PI) control. Compare this with Figures 8 and 11. The position takes longer to settle out than the system with pure proportional control, but it will not settle to the wrong spot.

Figure 13: Motor and gear with PI control.

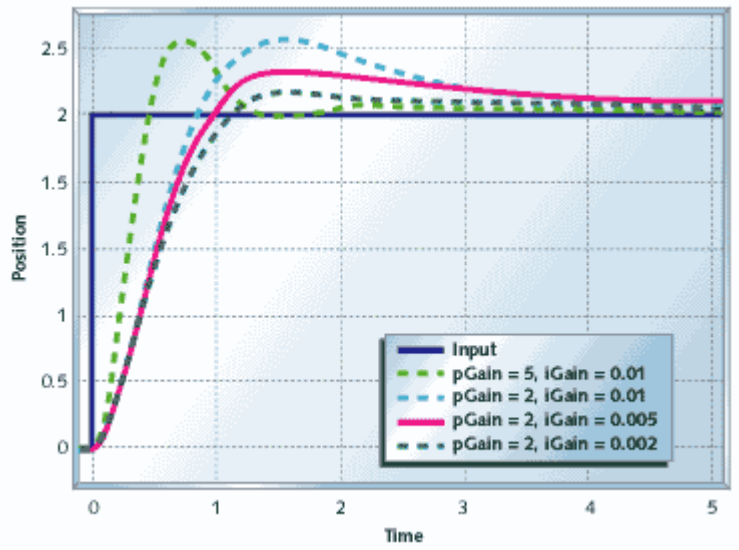
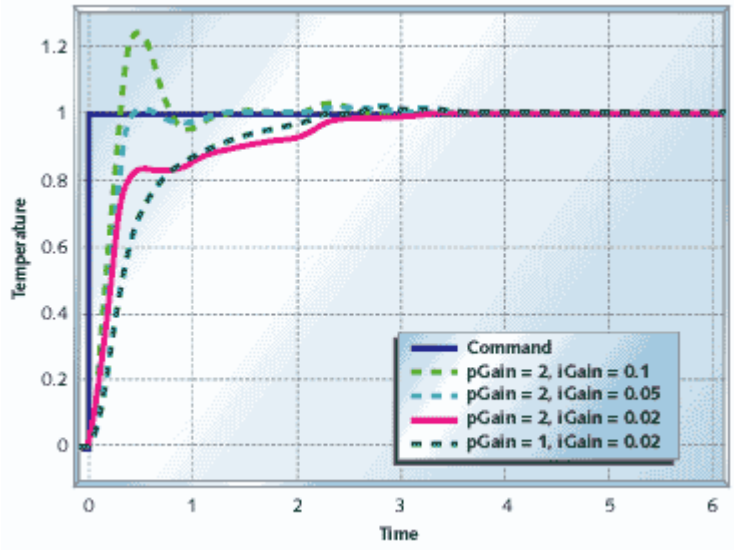


Figure 14 shows what happens when you use PI control on the heater system. The heater still settles out to the exact target temperature, as with pure integral control (see Figure 12), but with PI control, it settles out two to three times faster. This figure shows operation pretty close to the limit of the speed attainable using PI control with this plant.

Figure 14: Heater with PI control. The kink at time = 2 is from the external disturbance.



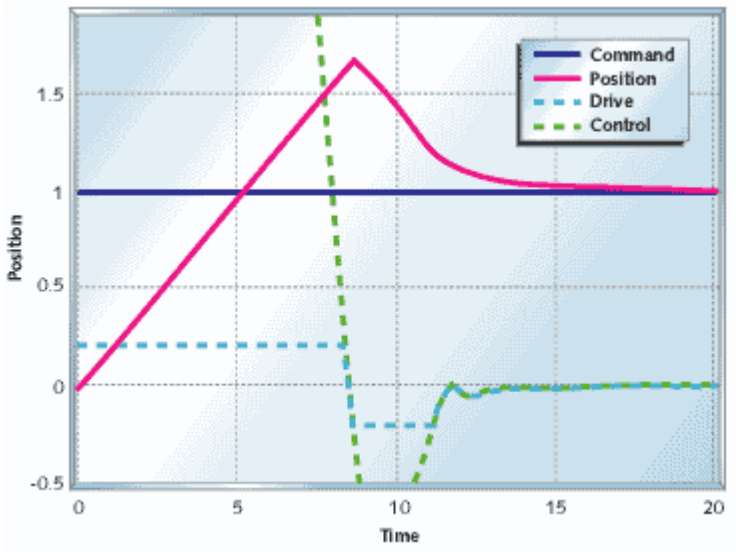
Before we leave the discussion of integrators, there are two more things I need to point out. First, since you are adding up the error over time, the sampling time that you are running becomes important. Second, you need to pay attention to the range of your integrator to avoid windup.

The rate that the integrator state changes is equal to the average error multiplied by the integrator gain multiplied by the sampling rate. Because the integrator tends to smooth things out over the long term you can get away with a somewhat uneven sampling rate, but it needs to average out to a constant value. At worst, your sampling rate should vary by no more than ý20% over any 10-sample interval. You can even get away with missing a few samples as long as your average sample rate stays within bounds. Nonetheless, for a PI controller I prefer to have a system where each sample falls within ý1% to ý5% of the correct sample time, and a long-term average rate that is right on the button.

If you have a controller that needs to push the plant hard, your controller output will spend significant amounts of time outside the bounds of what your drive can actually accept. This condition is called saturation. If you use a PI controller, then all the time spent in saturation can cause the integrator state to grow (wind up) to very large values. When the plant reaches the target, the integrator value is still very large, so the plant drives beyond the target while the integrator unwinds and the process reverses. This situation can get so bad that the system never settles out, but just slowly oscillates around the target position.

Figure 15 illustrates the effect of integrator windup. I used the motor/controller of Figure 13, and limited the motor drive to ý0.2. Not only is controller output much greater than the drive available to the motor, but the motor shows severe overshoot. The motor actually reaches its target at around five seconds, but it doesn't reverse direction until eight seconds, and doesn't settle out until 15 seconds have gone by.

Figure 15: Motor and gear with PI control and windup.



The easiest and most direct way to deal with integrator windup is to limit the integrator state, as I showed in my previous code example. Figure 16 shows what happens when you take the system in Figure 15 and limit the integrator term to the available drive output. The controller output is still large (because of the proportional term), but the integrator doesn't wind up very far and the system starts settling out at five seconds, and finishes at around six seconds.

Note that with the code example above you must scale iMin and iMax whenever you change the integrator gain. Usually you can just set the integrator minimum and

maximum so that the integrator output matches the drive minimum and maximum. If you know your disturbances will be small and you want quicker settling, you can limit the integrator further.

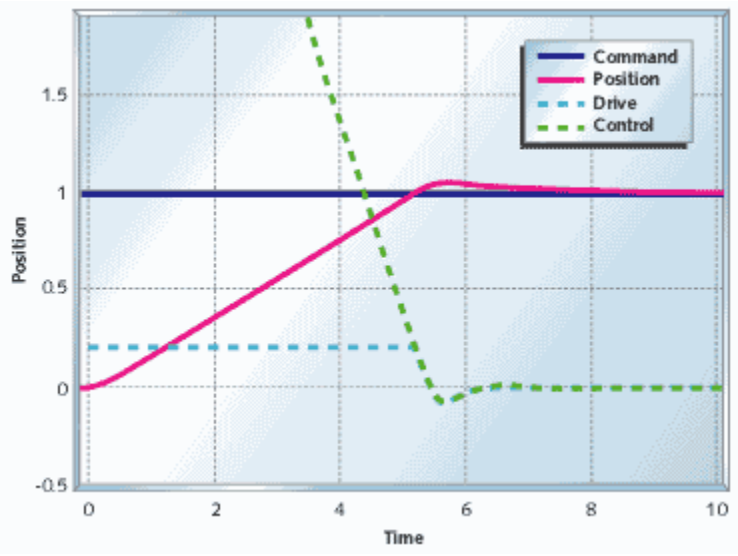


Figure 16: Motor and gear with PI control and integrator limiting.

Differential

I didn't even show the precision actuator in the previous section. This is because the precision actuator cannot be stabilized with PI control. In general, if you can't stabilize a plant with proportional control, you can't stabilize it with PI control. We know that proportional control deals with the present behavior of the plant, and that integral control deals with the past behavior of the plant. If we had some element that predicts the plant behavior then this might be used to stabilize the plant. A differentiator will do the trick.

The code below shows the differential term of a PID controller. I prefer to use the actual plant position rather than the error because this makes for smoother transitions when the command value changes. The differential term itself is the last value of the position minus the current value of the position. This gives you a rough estimate of the velocity (delta position/sample time), which predicts where the position will be in a while.

```
double dTerm;
.
.
.
dTerm = pid->dGain * (position - pid->dState);
pid->dState = position;
.
.
.
```

With differential control you can stabilize the precision actuator system. Figure 17 shows the response of the precision actuator system with proportional and derivative (PD) control. This system settles in less than 1/2 of a second, compared to multiple seconds for the other systems. Figure 18 shows the heating system with PID control.

You can see the performance improvement to be had by using full PID control with this plant.

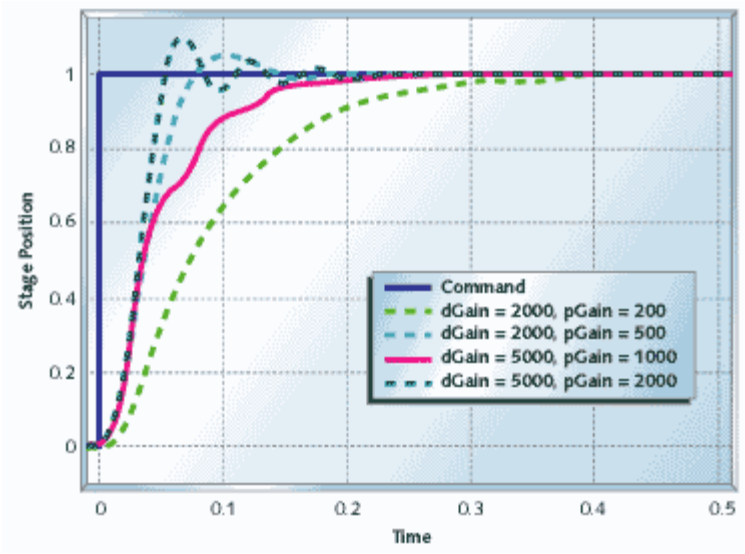


Figure 17: Precision actuator with PID control.

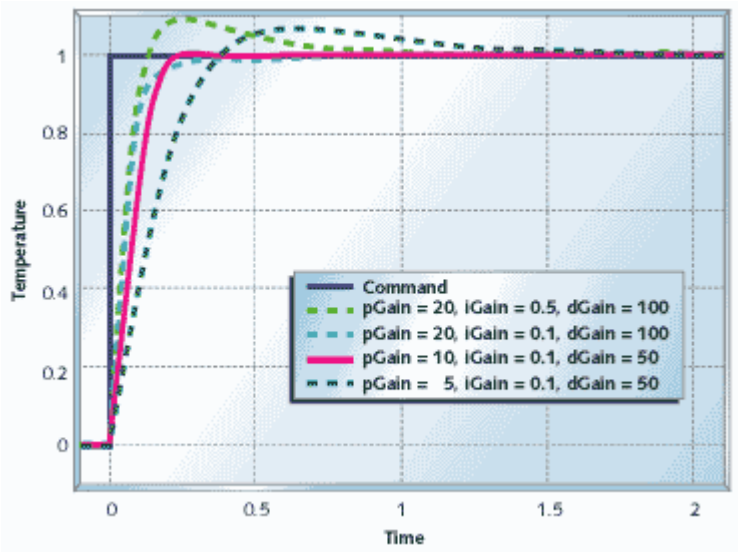


Figure 18: Heater with PID control.

Differential control is very powerful, but it is also the most problematic of the control types presented here. The three problems that you are most likely going to experience are sampling irregularities, noise, and high frequency oscillations. When I presented the code for a differential element I mentioned that the output is proportional to the position change divided by the sample time. If the position is changing at a constant rate but your sample time varies from sample to sample, you will get noise on your differential term. Since the differential gain is usually high, this noise will be amplified a great deal.

When you use differential control you need to pay close attention to even sampling. I'd say that you want the sampling interval to be consistent to within 1% of the total at all times-the closer the better. If you can't set the hardware up to enforce the sampling interval, design your software to sample with very high priority. You don't

have to actually execute the controller with such rigid precision-just make sure the actual ADC conversion happens at the right time. It may be best to put all your sampling in an ISR or very high-priority task, then execute the control code in a more relaxed manner.

Differential control suffers from noise problems because noise is usually spread relatively evenly across the frequency spectrum. Control commands and plant outputs, however, usually have most of their content at lower frequencies. Proportional control passes noise through unmolested. Integral control averages its input signal, which tends to kill noise. Differential control enhances high frequency signals, so it enhances noise. Look at the differential gains that I've set on the plants above, and think of what will happen if you have noise that makes each sample a little bit different. Multiply that little bit by a differential gain of 2,000 and think of what it means.

You can low-pass filter your differential output to reduce the noise, but this can severely affect its usefulness. The theory behind how to do this and how to determine if it will work is beyond the scope of this article. Probably the best that you can do about this problem is to look at how likely you are to see any noise, how much it will cost to get quiet inputs, and how badly you need the high performance that you get from differential control. Once you've worked this out, you can avoid differential control altogether, talk your hardware folks into getting you a lower noise input, or look for a control systems expert.

The full text of the PID controller code is shown in Listing 1 and is available at www.embedded.com/code.html.

Listing 1: PID controller code

```
typedef struct
{
double dState; // Last position input
double iState; // Integrator state
double iMax, iMin;
// Maximum and minimum allowable integrator state
double   iGain, // integral gain
pGain, // proportional gain
dGain; // derivative gain
} SPid;
double UpdatePID(SPid * pid, double error, double
position)
{
double pTerm,
dTerm, iTerm;
pTerm = pid->pGain * error;
// calculate the proportional term
// calculate the integral state with appropriate limiting
pid->iState += error;
if (pid->iState > pid->iMax) pid->iState = pid->iMax;
else if (pid->iState
<
```

```
pid->iMin) pid->iState = pid->iMin;
iTerm = pid->iGain * iState; // calculate the integral term
dTerm = pid->dGain * (position - pid->dState);
pid->dState = position;
return pTerm + iTerm - dTerm;
}
```

Tuning

The nice thing about tuning a PID controller is that you don't need to have a good understanding of formal control theory to do a fairly good job of it. About 90% of the closed-loop controller applications in the world do very well indeed with a controller that is only tuned fairly well.

If you can, hook your system up to some test equipment, or write in some debug code to allow you to look at the appropriate variables. If your system is slow enough you can spit the appropriate variables out on a serial port and graph them with a spreadsheet. You want to be able to look at the drive output and the plant output. In addition, you want to be able to apply some sort of a square-wave signal to the command input of your system. It is fairly easy to write some test code that will generate a suitable test command. Once you get the setup ready, set all gains to zero. If you suspect that you will not need differential control (like the motor and gear example or the thermal system) then skip down to the section that discusses tuning the proportional gain. Otherwise start by adjusting your differential gain.

The way the controller is coded you cannot use differential control alone. Set your proportional gain to some small value (one or less). Check to see how the system works. If it oscillates with proportional gain you should be able to cure it with differential gain. Start with about 100 times more differential gain than proportional gain. Watch your drive signal. Now start increasing the differential gain until you see oscillation, excessive noise, or excessive (more than 50%) overshoot on the drive or plant output. Note that the oscillation from too much differential gain is much faster than the oscillation from not enough. I like to push the gain up until the system is on the verge of oscillation then back the gain off by a factor of two or four. Make sure the drive signal still looks good. At this point your system will probably be responding very sluggishly, so it's time to tune the proportional and integral gains.

If it isn't set already, set the proportional gain to a starting value between 1 and 100. Your system will probably either show terribly slow performance or it will oscillate. If you see oscillation, drop the proportional gain by factors of eight or 10 until the oscillation stops. If you don't see oscillation, increase the proportional gain by factors of eight or 10 until you start seeing oscillation or excessive overshoot. As with the differential controller, I usually tune right up to the point of too much overshoot then reduce the gain by a factor of two or four. Once you are close, fine tune the proportional gain by factors of two until you like what you see.

Once you have your proportional gain set, start increasing integral gain. Your starting values will probably be from 0.0001 to 0.01. Here again, you want to find the range of integral gain that gives you reasonably fast performance without too much overshoot and without being too close to oscillation.

Other issues

Unless you are working on a project with very critical performance parameters you can often get by with control gains that are within a factor of two of the "correct" value. This means that you can do all your "multiplies" with shifts. This can be very handy when you're working with a slow processor.

Sampling rate

So far I've only talked about sample rates in terms of how consistent they need to be, but I haven't told you how to decide ahead of time what the sample rate needs to be. If your sampling rate is too low you may not be able to achieve the performance you want, because of the added delay of the sampling. If your sampling rate is too high you will create problems with noise in your differentiator and overflow in your integrator.

The rule of thumb for digital control systems is that the sample time should be between 1/10th and 1/100th of the desired system settling time. System settling time is the amount of time from the moment the drive comes out of saturation until the control system has effectively settled out. If you look at Figure 16, the controller comes out of saturation at about 5.2s, and has settled out at around 6.2s. If you can live with the one second settling time you could get away with a sampling rate as low as 10Hz.

You should treat the sampling rate as a flexible quantity. Anything that might make the control problem more difficult would indicate that you should raise the sampling rate. Factors such as having a difficult plant to control, or needing differential control, or needing very precise control would all indicate raising the sampling rate. If you have a very easy control problem you could get away with lowering the sampling rate somewhat (I would hesitate to lengthen the sample time to more than one-fifth of the desired settling time). If you aren't using a differentiator and you are careful about using enough bits in your integrator you can get away with sampling rates 1,000 times faster than the intended settling time.

Exert control

This covers the basics of implementing and tuning PID controllers. With this information, you should be able to attack the next control problem that comes your way and get it under control.

Tim Wescott has a master's degree in electrical engineering and has been working in industry for more than a decade. His experience has included a number of control loops closed in software using 8- to 32-bit microprocessors, DSPs, assembly language, C, and C++. He is currently involved in control systems design at FLIR Systems where he specifies mechanical, electrical, and software requirements and does electrical and software design. You can contact him at tim@wescottdesign.com.