8.18 References and Further Reading

- Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Reading, MA: Addison-Wesley, 1995. Describes concepts in operating systems.
- Jean Bacon. *Concurrent Systems*. New York: Addison-Wesley, 1993. Describes concurrent systems including real-time, database, and distributed systems.
- Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. New York: McGraw-Hill, 1994. Describes advanced concepts in operating systems and multitasking environments.
- Gary Cornell and Cay S. Horstmann. *Core Java*. Englewood Cliffs, NJ: Prentice Hall, 1997. This book describes the Java programming language, including the multithreaded programming.

8.19 Exercises

8.1 Define the following terms: finite-state machines, concurrent processes, real-time systems, and real-time operating system.

8.2 Briefly describe three computation models commonly used to describe embedded systems and/or their peripherals. For each model list two languages that can be used to capture it.

8.3 Describe the elevator UnitControl state machine in Figure 8.4 using the FSMD model definition <S, I, O, V, F, H, s0> given in this chapter. In other words, list the set of states (S), set of inputs (I), and so on.

8.4 Show how using the process create and join semantics one can emulate the procedure call semantics of a sequential programming model.

8.5 List three requirements of real-time systems and briefly describe each. Give examples of actual real-time systems to support your arguments.

8.6 Show why, in addition to ordered locking, two-phase locking is necessary in order to avoid deadlocks. Give an example and execution trace that results in deadlock if two-phase locking is not used.

8.7 Give pseudo-code for a pair of functions implementing the send and receive communication constructs. You may assume that mutex and condition variables are provided.

8.8 Show that the buffer size in the consumer-producer problem implemented in Figure 8.9 will never exceed one. Re-implement this problem, using monitors, to allow the size of the buffer to reach its maximum.

8.9 Given the processes A through F in Figure 8.21(a) where their deadline equals their period, determine whether they can be scheduled on time using a non-preemptive scheduler. Assume all processes begin at the same time and their execution times are as follows: A: 8 ms, B: 25 ms, C: 6 ms, D: 25 ms, E: 10 ms, F: 25 ms. Explain your answer by showing that each process either meets or misses its deadline.

CHAPTER 9: *Control Systems*[3]

9.1 Introduction

Control systems represent a very common class of embedded systems. A control system seeks to make a physical system's output track a desired reference input, by setting physical system inputs. Perhaps the best-known example is an automobile cruise controller, which seeks to make a car's speed track a desired speed, by setting the car's throttle and brake inputs. Another example is a thermostat controller, which seeks to force a building's temperature to a desired temperature, by turning on the heater or air conditioner and adjusting the fan speed. More examples include controlling the speed of a spinning disk drive by varying the applied motor voltage, and maintaining the altitude of an aircraft by adjustment of the aileron and elevator positions. In contrast, digital cameras, video games, and cell phones are not examples of control systems, as they do not seek to track a reference input. Figure 9.1 illustrates the idea of tracking in a control system.

Designing control systems is not easy. Think of a car's cruise controller. It should never let the car speed deviate significantly from the reference speed specified by the driver. It must adjust to external factors like wind speed, road grade, tire pressure, brake conditions, and

[3] This chapter was contributed mainly by Jay Farrell of the University of California, Riverside.

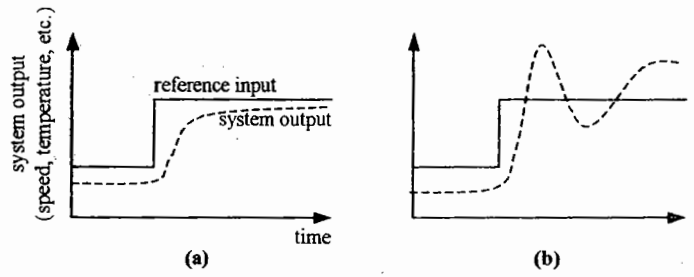


Figure 9.1: The goal of a control system is to force a physical system's output to track a reference input: (a) good tracking, (b) not-as-good tracking.

engine performance. It must correctly handle any situation presented to it, like accelerating from 20 mph to 50 mph while going down a steep hill. It should control the car in a way that is comfortable to the car's passengers, avoiding extremely fast acceleration or deceleration, and avoiding speed oscillations.

Control systems have been widely studied, and a rich theory for control system design exists. This chapter does not describe that theory in detail, since that requires a book in itself as well as a strong background in differential equations. Instead, we will introduce the basic concepts of control systems using a greatly simplified example. This introduction will lead up to PID controllers, which are extremely common. One of the goals of the chapter is to enable the reader to detect when an embedded system is an instance of a control system, so that the reader knows to turn to control theory (or to someone trained in control theory), rather than using ad hoc techniques, in those cases. However, in some cases, PID controllers can be used without extensive knowledge of control theory, and thus we will introduce some commonly used PID tuning techniques.

9.2 Open-Loop and Closed-Loop Control Systems

Overview

Control systems minimally consist of several parts, illustrated in Figure 9.2:

1. The *plant*, also known as the process, is the physical system to be controlled. An automobile is an example of a plant, as in Figure 9.2(a).
2. The *output* is the particular physical system aspect that we are interested in controlling. The speed of an automobile is an example of an output.
3. The *reference* input is the desired value that we want to see for the output. The desired speed set by an automobile's driver is an example of a reference input.

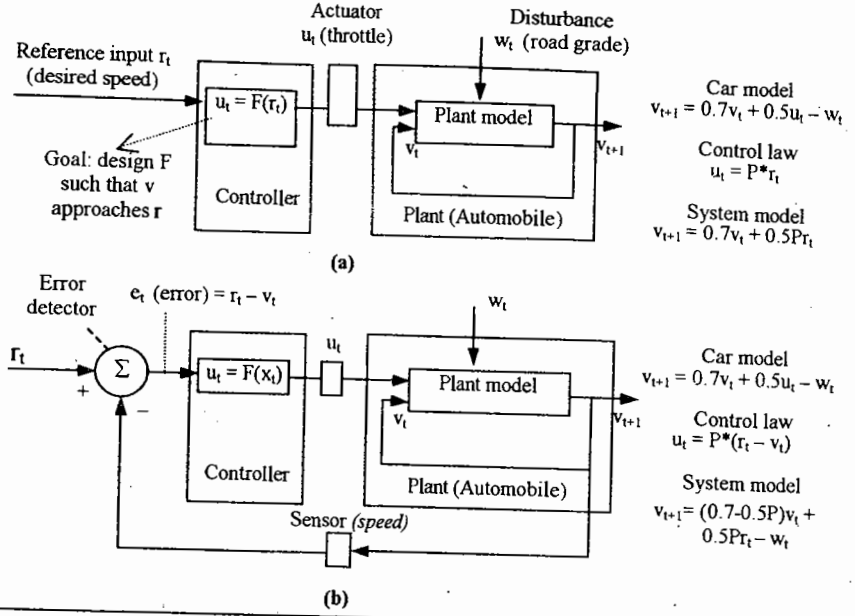


Figure 9.2: Control systems and automobile cruise controller example: (a) open-loop control, (b) closed-loop control.

4. The *actuator* is the device that we use to control the input to the plant. A stepper motor controlling a car's throttle position is an example of an actuator.
5. The *controller* is the system that we use to compute the input to the plant such that we achieve the desired output from the plant.
6. A *disturbance* is an additional undesirable input to the plant imposed by the environment that may cause the plant output to differ from what we would have expected based on the plant input. Wind and road grade are examples of disturbances that can alter the speed of an automobile.

A control system with these components, configured as in Figure 9.2(a), is referred to as an *open-loop*, or *feed-forward*, control system. The controller reads the reference input, and then computes a setting for the actuator. The actuator modifies the input to the plant, which, along with any disturbance, results some time later in a change in the plant output. In an open-loop system, the controller does not measure how well the plant output matches the reference input. Thus, open-loop control is best suited to situations where the plant output responds very predictably to the plant input (i.e., the model is accurate and disturbance effects are minimal).

Many control systems possess some additional parts, as illustrated in Figure 9.2(b):

1. A *sensor* measures the plant output.

2. An *error detector* determines the difference between the plant output and the reference input.

A control system with these parts, configured as in Figure 9.2(b), is known as a *closed-loop*, or *feedback*, control system. A closed-loop system monitors the error between the plant output and the reference input. The controller adjusts the plant input in response to this error. The goal is typically to minimize this tracking error given the physical constraints of the system.

A First Example: An Open-Loop Automobile Cruise Controller

We are primarily interested in closed-loop control in this chapter. However, let us begin by providing a simple example of an open-loop automobile cruise controller, illustrated in Figure 9.2(a). As you probably already know, the objective of a cruise-control system is to match the car's speed to the desired speed set by the driver.

Developing a Model: In many cases of controller design, our first task is to develop a *model* of how the plant behaves. A model describes how the plant output reacts as a function of the plant inputs and current state. For our cruise controller, the model should describe how the car reacts to the throttle position and the current speed of the car. As we will see later in this chapter, we don't always have to model the plant, and instead could design a controller through somewhat ad hoc experimenting. We could see how a particular controller works and iteratively modify the controller until the desired tracking is achieved. However, for many plants, like a car, such experimenting is dangerous, so using a model for the experimenting is preferable. Furthermore, with a model, we can even design the controller using quantitative techniques, thus avoiding the need for experimentation while creating a better controller.

The car has a throttle input whose position u can vary from 0 to 45 degrees. We decide to begin by test-driving the car on a flat road and taking measurements. Suppose that with the car traveling steadily at 50 mph and the throttle set at 30 degrees, we quickly change the throttle to 40 degrees, and measure the car's speed every second thereafter, until the car's speed finally becomes constant. Based on the measured speed data, suppose we determine that the following equation describes the car's speed as a function of the current speed and throttle position:

$$v_{t+1} = 0.7v_t + 0.5u_t$$

Here, v_t is the car's current speed, u_t is the throttle position, and v_{t+1} is the car's speed one second later. For example, $v_2 = 0.7v_1 + 0.5u_1 = 0.7 * 50 + 0.5 * 40 = 55$. Suppose further that we try a variety of other speeds and throttle positions, and we find that the above equation holds for all those other situations. Therefore, we decide that the above equation is a suitable first model for the car over the range of speed that is of interest. Please note that this is not actually a reasonable model of a car, and is instead used for illustrative purposes only.

Developing a Controller: Now let's turn our attention away from modeling the car and toward designing the cruise controller for the car. Suppose the only input to the controller is the desired speed r_t, as shown in Figure 9.2(a). The controller's behavior is a function F of the commanded speed, so that the throttle position is $u_t = F(r_t)$. The control designer can choose

F to be as simple or as complex a function as desired. Let's start by assuming that F is a simple linear function of the form:

$$u_t = F(r_t)$$

$$u_t = P * r_t$$

Here, P is a constant that the designer must specify. This linear proportional controller makes intuitive sense since it increases the throttle angle as the desired speed increases. In other words, the throttle angle is proportional to the desired speed.

Given this proportional control function, we can now write an equation that models the combined controller and plant, which will help us determine what value to use for P:

$$v_{t+1} = 0.7v_t + 0.5u_t$$

$$u_t = P * r_t$$

$$v_{t+1} = 0.7v_t + 0.5P * r_t$$

The design goal for the cruise controller is to keep the actual speed of the car v equal to the desired speed r at all times. Of course, it is impossible to keep these two values equal *at all times*, since the car will require some time to react to any changes the controller makes to the throttle angle. For example, the car cannot accelerate from 0 to 50 mph instantaneously. Rather, from the moment the controller sets the throttle, a car will take several seconds to accelerate to its final speed. Therefore, the design goal can be relaxed to that of forcing the car's actual speed v to be equal to the desired speed r in *steady state*. Steady state means that if the controller sets the throttle to a constant value, and nothing else changes, then at some time in the future, v will also not change. So in steady state, $v_{t+1} = v_t$. Let's refer to this steady-state velocity as v_{ss}. Substituting v_{ss} for both v_{t+1} and v_t above, we get:

$$v_{t+1} = 0.7v_t + 0.5P * r_t$$

$$let,\ v_{t+1} = v_t = v_{ss}$$

$$v_{ss} = 0.7v_{ss} + 0.5P * r_t$$

$$v_{ss} - 0.7v_{ss} = 0.5P * r$$

$$v_{ss} = 1.67P * r_t$$

So, if we want $v_{ss} = r_t$, we merely need to set $P = 1/1.67 = 0.6$. We have now designed our first controller:

$$u_t = F(r_t)$$

$$u_t = P * r_t$$

$$u_t = 0.6r_t$$

The controller merely multiplies the desired speed r_t by $0.6P$ to determine the desired throttle angle.

Time (t)	v_t	v_t for w = +5	v_t for w = –5
0	20.00	20.00	20.00
1	29.00	24.00	34.00
2	35.30	26.80	43.80
3	39.71	28.76	50.66
4	42.80	30.13	55.46
5	44.96	31.09	58.82
6	46.47	31.76	61.18
7	47.53	32.24	62.82
8	48.27	32.56	63.98
9	48.79	32.80	64.78
10	49.15	32.96	65.35
11	49.41	33.07	65.74
12	49.58	33.15	66.02
	(a)	(b)	(c)

Figure 9.3: Open-loop cruise controller trying to accelerate the car from 20 mph to 50 mph, when the grade is: (a) 0%, (b) +5%, (c) –5%.

Analyzing Our First Controller: Let's analyze how well this controller achieves its goal. Two issues are of interest: (1) what is the transient behavior when r changes; and (2) what effects do disturbances have on the system? The equation representing the entire system is:

$$v_{t+1} = 0.7v_t + 0.5*0.6r_t$$

$$v_{t+1} = 0.7v_t + 0.3r_t$$

To see how the system behaves, suppose a car is traveling steadily at 20 mph at time t = 0, at which time the desired speed r_0 is set to 50. Given the form of our controller above, we see that the controller will set the throttle position to 0.6 * 50 = 30 degrees, and hold it there until r_t changes again. We can "simulate" the system by evaluating the above equation for various time values (a spreadsheet program makes this task easy).[4] Figure 9.3(a) illustrates the car's speed over time. We see that (in the absence of disturbances) the controller does well, approaching the desired speed of 50 mph to within 0.3% in 10 seconds.

Considering Disturbances: Suppose now that additional testing of the car is performed on roads with grades w varying from –5 degrees, corresponding to downhill roads, to +5 degrees, corresponding to uphill roads. The car goes faster downhill and slower uphill. Suppose road grade is incorporated into the earlier model for the car alone as follows:

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

Since the open-loop controller has no means of sensing the road grade or its effect on the speed, this disturbance will obviously result in speed error when driving downhill or uphill. Figure 9.3(b) displays the behavior of the car with the open loop controller when driving up a +5% grade, and Figure 9.3(c) when driving down a –5% grade. The speed error at time t = 12 is about 50 – 33 = 17 mph in the uphill case, and about 50 – 66 = –16 mph in the downhill case. This error is quite bad! Closed-loop control systems, which will be discussed shortly, can help reduce errors caused by disturbances.

Determining Performance Parameters: Using the model of the system created earlier, a designer can quickly determine various important performance parameters. Assume that the initial speed is v_0, the desired speed is r_0, and the disturbance is w_0, then we can develop an equation for v_t as follows:

$$v_1 = 0.7v_0 + 0.5P * r_0 - w_0$$

$$v_2 = 0.7 * (0.7v_0 + 0.5P * r_0 - w_0) + 0.5P * r_0 - w_0$$

$$v_2 = 0.7 * 0.7v_0 + (0.7 + 1.0) * 0.5P * r_0 - (0.7 + 1.0) * w_0$$

$$v_t = 0.7^t * v_0 + (0.7^{t-1} + 0.7^{t-2} + \ldots + 0.7 + 1.0)(0.5P * r_0 - w_0)$$

The last equation shows three important points. First, in the model $v_{t+1} = 0.7v_t + 0.5u_t - w_t$, let's refer to the coefficient of v_t as α; in this case α = 0.7. Looking at the last equation, we see that α determines the *rate of decay of the effect of the initial speed.* In other words, a bigger α will result in the car taking longer to reach its desired speed. Notice that, in open loop control, the controller gain P has no effect on this rate of decay. In closed-loop control, it will.

Also note that if $|\alpha| > 1$, then v_t would grow without bound as time increased, since α is being raised to the power of t. Furthermore, note that a negative α will result in an oscillating speed. Again, in closed-loop control, we will be able to change α.

Second, the sensitivity of the speed to the *disturbance* is not altered by the open loop controller.

Third, if our assumed model were not correct, then this model error would cause the steady state speed, that results from the open loop controller $u = P_r$, to not equal the desired speed.

A Second Example: A Closed-Loop Automobile Cruise Controller

We can reduce the speed error caused by disturbances, like grade or wind, by enabling the controller to detect speed errors and correct for them. To detect speed errors, we introduce a speed sensor into the system, as shown in Figure 9.2(b), to measure the car's speed. We also introduce a device that outputs the difference between the desired speed r_t and the actual speed v_t. This difference is the speed error $e_t = r_t - v_t$. Note that the penalties for this

[4] Note that the simulation evaluates the controller performance relative to a model. For the simulation results to accurately predict the results of the future control experiments with the actual hardware, the model must be accurate. However, since there is expense involved with developing the model, there is always a trade off and art to determining when the model is sufficiently accurate to complete the analytic portion of the design. Tuning of the design typically occurs during the initial hardware experiments to accommodate differences between the model and hardware.

closed-loop approach are the cost of the sensor, added controller complexity, and the addition of sensor noise. The benefits will be the ability to change the rate of response, reduction of sensitivity to disturbances, and reduction of sensitivity to model error. If we select the form of the controller to be linear and proportional as before, namely $u_t = P * (r_t - v_t)$, then:

$$v_{t+1} = 0.7v_t + 0.5u_t - w,$$

$$v_{t+1} = 0.7v_t + 0.5P * (r_t - v_t) - w_t$$

$$v_{t+1} = (0.7 - 0.5P) * v_t + 0.5P * r_t - w_t$$

Note that the closed-loop controller results in $\alpha = 0.7 - 0.5P$, and remember that α determines the rate of decay of the effect of the initial speed. Therefore, by choice of the parameter P, the control system designer can alter the rate of convergence of the closed-loop system. However, we cannot make P arbitrarily large, because if the designer selects a value of P such that $|0.7 - 0.5P| > 1.0$, then the speed will not converge to the commanded speed, but instead grow without bound. The constraint $|0.7 - 0.5P| < 1.0$ is necessary for the system to be stable. This stability constraint translates to the following:

$$0.7 - 0.5P < 1.0$$

$$0.7 - 0.5P > -1.0$$

$$-0.5P < 0.3$$

$$-0.5P > -1.7$$

$$P > -0.6$$

$$P < 3.4$$

$$\text{so, } -0.6 < P < 3.4$$

We could set P close to 3.4 to obtain the fastest decay of the initial condition. However, remember that a negative α will cause oscillation, which is something we'd usually like to avoid. To keep α positive, we need:

$$0.7 - 0.5P >= 0$$

$$-0.5P >= -0.7$$

$$P <= 1.4$$

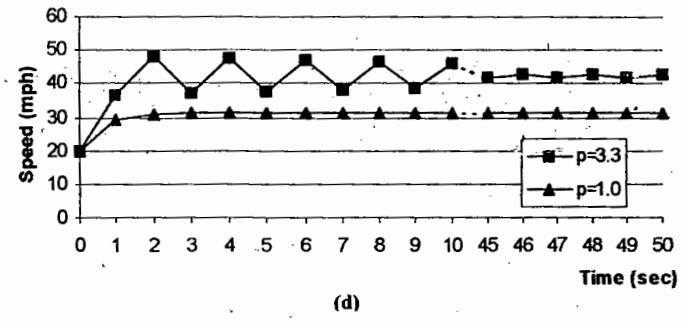
So the fastest rate of convergence to steady state without oscillation, known as deadbeat control, occurs when $P = 1.4$.

A control design goal is to achieve v equal to r in steady state, meaning $v_{t+1} = v_t = v_{ss} = r$. To analyze the steady-state response, we again assume the commanded speed and disturbance have the constant values r_0 and w_0. Substituting into the earlier system equation yields:

$$v_{ss} = (0.7 - 0.5P) * v_{ss} + 0.5P * r_0 - w_0$$

$$(1 - 0.7 + 0.5P) * v_{ss} = 0.5P * r_0 - w_0$$

Time	v_t	u_t	v_t	u_t	v_t	u_t
0	20.00	99.00	20.00	45.00	20.00	30.00
1	63.50	-44.55	36.50	44.55	29.00	21.00
2	22.18	91.82	47.83	7.18	30.80	19.20
3	61.43	-37.73	37.07	42.68	31.16	18.84
4	24.14	85.34	47.29	8.95	31.23	18.77
5	59.57	-31.58	37.58	40.99	31.25	18.75
6	25.91	79.50	46.80	10.55	31.25	18.75
7	57.89	-26.02	38.04	39.47	31.25	18.75
8	27.51	74.22	46.36	12.00	31.25	18.75
9	56.37	-21.01	38.45	38.10	31.25	18.75
10	28.95	69.46	45.97	13.31	31.25	18.75
...						
45	44.53	18.06	41.70	27.39	31.25	18.75
46	40.20	32.34	42.89	23.48	31.25	18.75
47	44.31	18.78	41.76	27.20	31.25	18.75
48	40.41	31.66	42.83	23.66	31.25	18.75
49	44.11	19.42	41.81	27.02	31.25	18.75
50	40.59	31.05	42.78	23.83	31.25	18.75
...						
ss	42.31	25.38	42.31	25.38	31.25	18.75
	(a)		(b)		(c)	



(d)

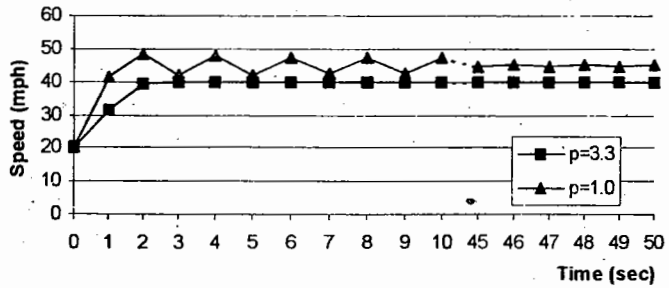
Figure 9.4: Closed-loop cruise controller trying to accelerate from 20 to 50 mph, ignoring disturbance, where v is car speed and u is throttle position: (a) invalid data when throttle saturation is ignored, (b) valid data for $P = 3.3$, (c) valid data for $P = 1.0$, (d) plot for $P = 3.3$ and $P = 1.0$.

$$v_{ss} = (0.5P / (0.3 + 0.5P)) * r_0 - (1.0 / (0.3 + 0.5P)) * w_0$$

From this equation, we see that we can reduce the effect of the disturbance w_0 by making the coefficient $(1.0 / (0.3 + 0.5P))$ less than 1, meaning that $P > 1.4$. But, remember that $P > 1.4$ will cause oscillation!

Time	v_t	u_t	v_t	u_t
0	20.00	45.00	20.00	45.00
1	31.50	45.00	41.50	28.05
2	39.55	34.49	48.08	6.35
3	39.93	33.24	41.83	26.97
4	39.57	34.42	47.76	7.38
5	39.91	33.30	42.13	25.99
6	39.59	34.37	47.48	8.31
7	39.89	33.35	42.39	25.10
8	39.60	34.32	47.23	9.15
9	39.88	33.40	42.63	24.30
10	39.62	34.27	47.00	9.91
...				
45	39.76	33.78	44.52	18.09
46	39.72	33.91	45.21	15.82
47	39.76	33.78	44.55	17.97
48	39.73	33.91	45.17	15.92
49	39.76	33.79	44.58	17.87
50	39.73	33.90	45.14	16.02
...				
ss	39.74	33.85	44.87	16.92

(a) (b)



(c)

Figure 9.5: The same closed-loop cruise controller, trying to accelerate from 20 to 50 mph, this time in the presence of a disturbance of a grade equal to: (a) +5%, (b) –5%, (c) graphical illustration of (a) and (b).

Also, if we want v_{ss} to be approximately equal to r_0, then we need to select P as large as possible, so that the term multiplying r_0, namely, $(0.5P / (0.3 + 0.5P))$, is approximately equal to 1. There is no value of P in the range $-0.6 < P < 3.4$ for which this coefficient equals 1, so perfect steady state tracking is not achievable by proportional control for this example. The best we can do to minimize steady-state error, therefore, is to set P reasonably close to 3.4. Note that the designer must select P to balance the trade-offs between the conflicting

constraints of convergence rate, disturbance rejection, and steady-state tracking accuracy. To continue, assuming that steady-state tracking is of primary performance, let $P = 3.3$.

We have now designed our second controller. The controller sets its output, the throttle angle u_t, to 3.3 times its input, as follows:

$$u_t = 3.3 * (r_t - v_t)$$

This controller will result in oscillation, but that's the price we pay to achieve the smallest steady-state error. Notice that the input to our second controller is the speed error, in contrast to our first controller, whose input was the desired speed. Let's analyze how well this second controller achieves its input-tracking goal by "simulating" the system, namely, by iterating the closed-loop equation over the time range of interest (again, a spreadsheet helps with such simulation). Initially, assume an initial speed of 20 mph, a grade w of 0, and then a desired speed setting of 50 mph. Figure 9.4(a) shows the speed v_t and the throttle position u_t from time 0 to 50 seconds. Notice that the controller generates throttle angle commands outside of the range of possible throttle positions of 0 to 45 degrees. Thus, the data in Figure 9.4(a) is not valid. Instead, we must treat any value less then 0 as 0, and greater than 45 as 45. The throttle is said to saturate at 0 and 45.

Figure 9.4(b) shows the speed and throttle position when we include this saturation in the model. Figure 9.4(d) shows the speed versus time graphically. Notice that, for $P = 3.3$ ($a = 0.7$ $-0.5P = -0.95$), the speed oscillates for many seconds until it finally reaches a steady-state speed of 42.31 mph. Recall that a negative a causes such oscillation. Intuitively, this oscillation means the cruise controller is accelerating too hard when the current speed is less than the desired speed, thus overshooting the desired speed. Also notice that the steady-state speed is not 50 mph, but rather 42.31 mph, representing an error of about 8 mph. The simulated responses in Figure 9.4(b) and (d) thus confirm our earlier analysis of oscillation and steady-state error.

Since oscillation of the car speed could be uncomfortable to the car's passengers, we would like to reduce or eliminate the oscillation. We can reduce the oscillation by decreasing the constant P in the controller. For example, Figure 9.4(c) shows the speed and throttle positions for $P = 1.0$. Figure 9.4(b) shows the speed graphically. The result of the smaller P is that oscillation is eliminated and convergence time is reduced; however, the steady-state speed is only 31.25 mph, representing a large error of nearly 19 mph.

We have learned an important lesson of control, namely, that system-performance objectives, such as reducing oscillation, obtaining fast convergence, and reducing steady-state error, often compete with one another.

Recall that a motivation for using closed-loop control was to reduce the speed error caused by disturbances like grade. Figure 9.5(a) and (b) show the effects of +5% and –5% grades, respectively, using $P = 3.3$ in the controller. Figure 9.5(c) shows the results for both situations graphically. Notice that steady-state errors of about 10 mph and 5 mph are not too much different than the 7 mph error with a 0% grade, but are much improved over the 17 mph and –16 mph errors that resulted from the open-loop controller in Figure 9.3(b) and (c). Thus, the goal of reducing the sensitivity to disturbances has been achieved, involving a trade-off of

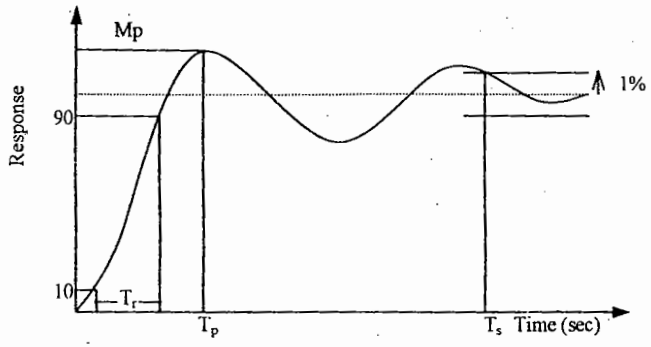


Figure 9.6: Control system response performance metrics.

having introduced additional steady-state error when there is no disturbance, and of having introduced oscillation.

To allow more control objectives to be satisfied with fewer trade-offs, the complexity of the controller will have to increase, as will be described subsequently.

9.3 General Control Systems and PID Controllers

Having seen the above examples, we can now discuss control systems more generally. This section discusses objectives of control design, modeling real physical systems, and the PID approach to controller design.

Control Objectives

The objective of control system design is to make a physical system behave in a useful fashion, in particular, by causing its output to track a desired reference input even in the presence of measurement noise, model error, and disturbances. Satisfaction of this objective can be evaluated through several metrics specified relative to a step change in the control systems input:

1. *Stability*: The main idea of stability is that all variables in the control system remain bounded. Preferably, the error variables, like desired output minus plant output, would converge to zero. Stability is of primary importance, since without stability, all of the other objectives are immaterial.
2. *Performance*: Assuming stability, performance describes how well the output tracks a change in the reference input. Performance has several aspects, illustrated in Figure 9.6.

   a) *Rise time* T_r is the time required for the response to change from 10% to 90% of the distance from the initial value to the final value, for the first time. Different percentages may be of interest in different applications.
   b) *Peak time* T_p is the time required to reach the first peak of the response.
   c) *Overshoot* M_p is the percentage amount by which the peak of the response exceeds the final value.
   d) *Settling time* T_s is the time required for the system to settle down to within 1% of its final value. A different percentage may be of interest in different applications.
3. *Disturbance rejection*: Disturbances are undesired effects on the system behavior caused by the environment. A designer cannot eliminate disturbances, but can reduce their impact on system behavior.
4. *Robustness*: The plant model is a simplification of a physical system, and is never perfect. Robustness requires that the stability and performance of the controlled system should not be significantly affected by the presence of model errors.

Modeling Real Physical Systems

An essential prelude to control system design is accurate modeling of the behavior of the plant. The controller will be designed based on this plant model. If the plant model is inaccurate, then the controller will be controlling the wrong plant. There are two key features that real systems display that our earlier example did not consider.

The first feature of real physical systems is that they typically respond as continuous variables and as continuous functions of time. In the cruise-controller example, we assumed that the car's speed would change exactly one second after a change in the throttle. Obviously, cars do not synchronize their reactions to the discrete time intervals, but instead they are continuously reacting. Therefore, the plant dynamic model is usually a differential equation. There are methods for determining a discrete time model that is equivalent — only at the sampling instants — to the plant differential equation. Between the sampling instants, the discrete time model tells the designer nothing about the continuous time response. Therefore, the sampling period must be selected much smaller than the system's reaction time so that the system cannot change significantly between sampling instants. The 1 second sample time used in the earlier examples of this chapter is not meant to be realistic. See also the subsequent discussion of aliasing.

The second feature of real physical systems is that they are typically much more complex than any model we create. The model will not include all nonlinear effects, all system states, or all state interactions. For example, the response of the speed of a car to a change in throttle depends on spark advance, manifold pressure, engine speed, and additional variables. Therefore, any model is a simplified abstraction. Modeling and control design is an iterative process, where the model of the actual plant is improved at each iteration to include key features identified during the prior iteration. Then the controller is improved to properly address the improved model. Linear models usually suffice when the variables of the model have a small operating range.

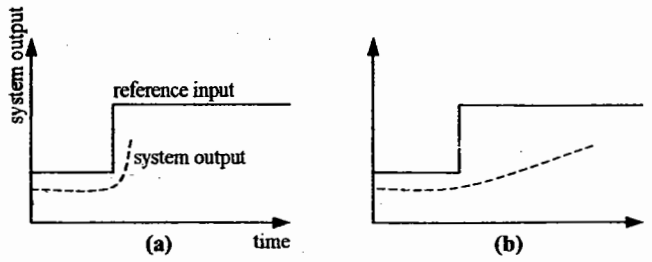


Figure 9.7: A better controller could be designed if it could predict the future. Both controllers have forced the output halfway to the desired value. But the controller for (a) should start to reduce the plant input, while the controller for (b) should have increased the input earlier. Derivative control seeks to satisfy this prediction goal.

Controller Design

The earlier closed loop example showed that increasing P caused the steady state speed v_{ss} to better match the desired speed r, and to resist tracking error caused by disturbances. A controller that multiplies the tracking error by a constant is known as using *proportional control*. To summarize, when proportional control is applied to a first order plant, the resulting closed loop model is similar to our particular cruise-controller model of:

$$v_{t+1} = (0.7 - 0.5P) * v_t + 0.5P * r_t - w_t$$

Therefore, the controller parameter P affects transient response, steady state tracking error, and disturbance rejection. However, we saw that adjusting P resulted in trade-offs among these control objectives. We could reduce oscillation and improve convergence, but at the expense of worse steady-state error, and vice versa.

PD Control: More degrees of freedom must be introduced into the controller design to allow greater flexibility in the optimization of the trade-offs involved in the closed loop performance. We can achieve this by using a proportional plus derivative controller. In proportional plus derivative control (*PD control*), the form of the control law is:

$$u_t = P * e_t + D * (e_t - e_{t-1})$$

Here, $e_t = r_t - v_t$ is the measured speed error, and $e_t - e_{t-1}$ is the derivative of the error (meaning the change in error over time). P is the proportional constant, and D is the derivative constant.

Intuitively, the derivative term is being used to predict the future. Consider Figure 9.7. The two plots show two different responses. In (a), we as humans can see that the system output is approaching the reference input quickly, and so we should probably reduce the plant input to prevent overshoot. In (b), we can see that the output is increasing very slowly, so we

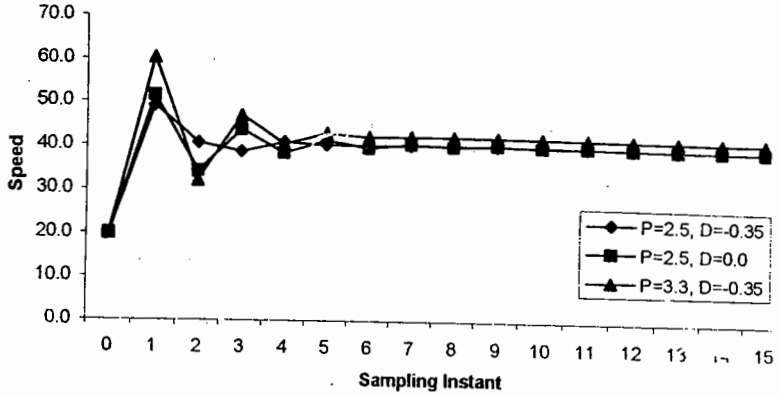


Figure 9.8: PD step response.

probably should increase the plant input, and actually should have increased it earlier. We see these things because we predict the system's future behavior will be similar to its past behavior, a good assumption when dealing with physical systems. The derivative term, which looks at the difference in the output between two successive time instances, can be used to achieve similar prediction, and thus can cause the controller to react accordingly. In the language of control systems, this is referred to as *adding lead*.

PD control implies a more complex controller, since the controller must keep track of the error derivative. However, PD control will give us more flexibility in achieving our control objectives. We can see this by deriving the equation for the complete cruise-controller system using PD control, just as we did for the simpler P controller:

$$v_{t+1} = 0.7v_t + 0.5u_t - w_t$$

$$let\ u_t = P * e_t + D * (e_t - e_{t-1})$$

$$and\ e_t = r_t - v_t$$

$$v_{t+1} = 0.7v_t + 0.5 * (P * (r_t - v_t) + D * ((r_t - v_t) - (r_{t-1} - v_{t-1}))) - w_t$$

...

$$v_{t+1} = (0.7 - 0.5 * (P + D)) * v_t + 0.5D * v_{t-1} + 0.5 * (P + D) * r_t - 0.5D * r_{t-1} - w_t$$

When the reference input and disturbance are constant, the steady-state speed is again:

$$v_{ss} = (0.5P / (1 - 0.7 + 0.5P)) * r$$

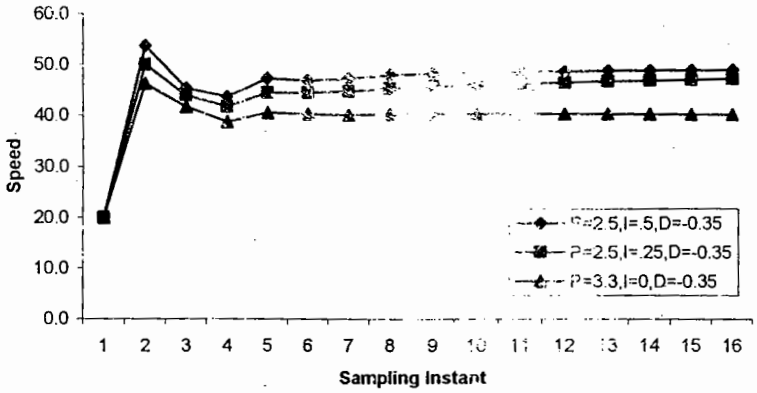


Figure 9.9: PID step response.

This is the same as for proportional control, since in steady state the effect of the derivative term is zero.

The characteristics of convergence of the tracking error e to its steady-state value is determined by the roots of the polynomial: $z^2 - (0.7 - 0.5 * (P + D)) * z - 0.5D = 0$, under the assumption that the magnitude of the roots (they may be complex) is less than 1. Therefore, adding the derivative term allows the transient response to be modified without affecting the steady state tracking or disturbance rejection characteristics. Figure 9.8 plots step responses for various values of P and D. Note that the steady state value of the response is affected by P, not D. The parameter D does significantly affect the character of the transient response, in other words, the rate of convergence and the oscillation. The dashed-dotted line should be compared with the response in Figure 9.4, for which $P = 3.3$ and for which we can treat $D = 0$. In summary, by building a slightly more complex controller, namely, a PD controller, which considers not just the error input but also the derivative of the error input, we can adjust the transient response and the steady-state error independently by adjusting D and P.

PI and PID Control: In proportional plus integral control (*PI control*), the form of the control law is:

$$u_t = P * e_t + I * (e_0 + e_1 + \ldots + e_t)$$

The integral term sums up the error over time. Let's consider this term intuitively. Look at Figure 9.4(d) again. Notice that both controllers achieve a steady-state value that is below the desired value of 50 mph. As humans, we can see that we should just increase the plant input again until this error goes to zero. In other words, as long as there's error, we shouldn't rest! The integral term achieves this goal by summing the error over time, we ensure that the

error will eventually go to zero, since otherwise the controller output would increase forever. In other words, with integral control, the steady state will not even exist unless $e_{ss} = 0$, since otherwise the integral term would be increasing. Therefore, if values of P and I are found such that the system is stable, then for a constant input the steady state tracking error is zero.

We can combine proportional, integral, and derivative control as follows:

$$u_t = P * e_t + I * (e_0 + e_1 + \ldots + e_t) + D * ((e_1 - e_0) + (e_2 - e_1) + \ldots + (e_t - e_{t-1}))$$

This is known as *PID control*. The controller design goal is then to select the PID gains to achieve the desired stable transient behavior. Figure 9.9 plots step responses for three different values of PID. The main effect of varying I is that as I is increased, the rate at which the response converges to its desired value increases; however, the I term does also affect the nature of the transient. If I is increased too much, then the response can become oscillatory or even unstable.

PID controllers are extremely common in embedded control systems. Several tools exist to help a designer choose the appropriate PID values for a given plant model. Off-the-shelf IC and cores with settable P, I, and D values, called PID controllers, are available to accomplish PID control.

9.4 Software Coding of a PID Controller

A PID controller can be implemented quite easily in software. Consider writing a program in C to implement a PID controller. It might consist of a main function with the following loop:

```
void main()
{
    double sensor_value, actuator_value, error_current;
    PID_DATA pid_data;
    PidInitialize(&pid_data);
    while (1) {
        sensor_value = SensorGetValue();
        reference_value = ReferenceGetValue();
        actuator_value =
         PidUpdate(&pid_data,sensor_value,reference_value);
        ActuatorSetValue(actuator_value);
    }
}
```

We create the main function to loop forever. During each iteration, we first read the plant output sensor, read the current desired reference input value, and pass this information to function *PidUpdate*. *PidUpdate* determines the value of the plant actuator, which we then use to set the actuator. Note that reading the sensor will typically involve an analog-to-digital converter, and setting the actuator will involve a digital-to-analog converter; the details of these functions are omitted.

Our *PID_DATA* data structure has the following form:

```
typedef struct PID_DATA {
    double Pgain, Dgain, Igain;
    double sensor_value_previous; // find the derivative
    double error_sum; // cumulative error
}
```

So *PID_DATA* holds the three gain constants, which we assume are set in the *PidInitialize* function. It also holds the previous sensor value, which will be used for the der vative term. Finally, it holds the cumulative sum of error values, used for the integral term.

We can now define our *PidUpdate* function as follows:

```
double PidUpdate(PID_DATA *pid_data, double sensor_value,
                 double reference_value)

{
    double Pterm, Iterm, Dterm;
    double error, difference;

    error = referen  _value - sensor_value;
    Pterm = pid_data->Pgain * error; /* proportional term*/
    pid_data->error_sum += error; /* current + cumulative*/
    // the integral term
    Iterm = pid_data->Igain * pid_data->error_sum;
    difference = pid_data->sensor_value_previous -
                 sensor_value;
    // update for next iteration
    pid_data->sensor_value_previous = sehsor_value;
    // the derivative term
    Dterm = pid_data->Dgain * difference;
    return (Pterm + Iterm + Dterm);
}
```

There are some modifications that are typically made to the basic code above to improve PID controller performance. For example, the *error_sum* is typically constrained to stay within a particular range, to reduce oscillation, and to avoid having the variable reach its upper limit and hence roll over to 0. Also, the integration of the error is typically stopped both when the tracking error is large and during periods of actuator saturation.

9.5 PID Tuning

Until now, we have discussed controller design based on a model of the plant. P, I, and D values could therefore be determined through quantitative analysis. In many cases, however, quantitatively determining the P, I, and D values is not necessary. In particular, in cases where

safety is not a concern, and the cost of using the plant is not a major concern either, we can select the PID values through a somewhat ad hoc tuning process. This has two advantages. First, our model of the plant may be too complex for us to work with quantitatively. Second, we may not even have a model of the plant, perhaps because we don't have the time or knowledge to create such a model. The tuning process we'll discuss has been shown to result in PID values that are reasonably close to the values that would have been obtained through quantitative analysis.

One tuning approach is to start by setting the P gain to some small value, and the D and I gains to 0. We then increase the D gain, usually starting about 100 times greater than P, until we see oscillation, at which point we reduce D by a factor of 2 to 4. At this point, the system will probably be responding slowly. Next, we begin increasing the P gain until we see oscillation or excessive overshoot, and then we reduce P by a factor of 2 to 4. Finally, we begin increasing the I gain, starting perhaps between 0.0001 and 0.01, and again backing off when we see oscillation or excessive overshoot. These three steps can be repeated until either satisfactory performance is achieved or performance cannot be further improved. There are many more detailed tuning approaches, but the one introduced here should give an idea of the basic approach.

9.6 Practical Issues Related to Computer-Based Control

Quantization and Overflow Effects

Quantization occurs when a signal or machine number must be altered to fit the constraints of the computer memory. For example, if the number 0.36 were to be stored as a 4-bit fraction, then it would have to quantized to one of the following machine numbers: 0.75, 0.50, 0.25, 0.00, –0.25, –0.50, –0.75, –1.00. The closed machine number is 0.25, which would result in a quantization error of 0.11. Quantization occurs for two reasons.

First, machine arithmetic can generate results requiring more precision that the original values. A simple example is the product of two 4-bit machine numbers:

$$0.50 * 0.25 = 0.125$$

This product cannot be stored as a signed four bit machine number. To limit the effects of quantization effects due to machine arithmetic, many digital processors will store intermediate results with higher precision than the final result. In such applications, arithmetic quanitzation only occurs when the final result of an operation is stored in a memory location. It is up to the designer to optimize the software implementation to take full advantage of such processor design features.

Second, the analog signals available from the sensors are real valued. These analog signals are quantized into machine numbers through the analog-to-digital conversion process. Accuracy and expense increase as the number of bits in the digital representation increase.

Overflow results when a machine operation outputs a number with magnitude too large to be represented by the numeric processor. In the 4-bit example above, 0.75 + 0.50 = 1.25 is too

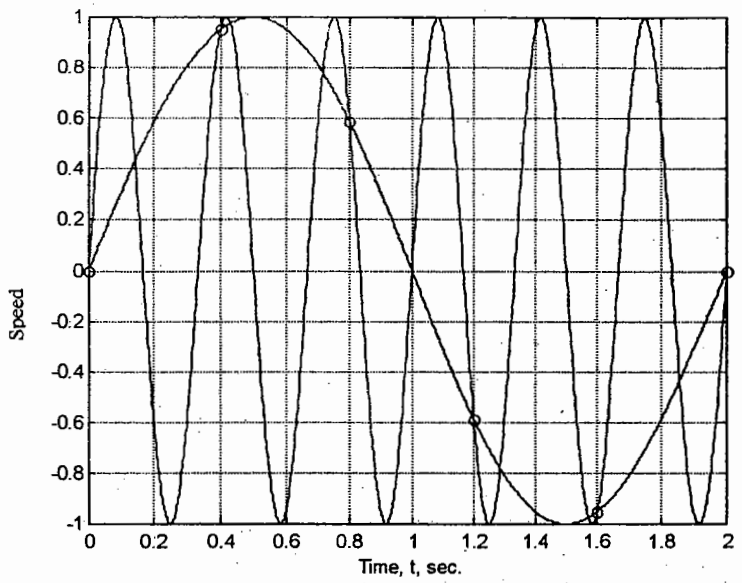


Figure 9.10: Illustration of aliasing. The sampling frequency is 2.5 Hz.

large to be represented as a machine number. The results of overflow are dependent on the method that the numeric processor uses to implement the binary representation of machine numbers.

The designer has a few design choices to address the affects of quantization and overflow. The first is fixed versus floating point representation of machine numbers. Fixed-point implementations are less expensive from the hardware perspective, but require the designer to carefully analyze and design the system to accommodate both quantization and overflow effects. The hardware for floating point implementations is typically more expensive, but the floating-point implementation may result in a faster time to market.

Aliasing

Physical systems typically evolve in continuous time, but discrete time control signals operate on samples of the evolving process. These simple facts can lead to counter intuitive behavior, if the sampling processes are not properly designed.

Figure 9.10 illustrates a process referred to as aliasing. The circles every 0.4 s represent the discrete time samples. The sampling frequency is 2.5 Hz. The actual signal is $y(t) = 1.0 * \sin(6\pi t)$, which is periodic with frequency 3.0 Hz. The figure shows that based on the samples

available to it from the sampling process, the actual signal is indistinguishable from $v(t) = 1.0 * \sin(\pi t)$. In fact, any of the sinusoids given by $z(t) = 1.0 * \sin((2\pi)(0.5 + 2.5n) * t)$ for any integer n would result in these same samples.

Aliasing is an artifact of the sampling process. When the sampling frequency is f_s based on the sample record, the computer can only resolve frequencies in a range of f_s Hz. In most applications, the system is designed so that this range is $f \varepsilon [-f_N, f_N]$, where $f_N = f_s / 2$ is the *Nyquist* frequency. When the actual signal has frequency content above the *Nyquist* frequency, it will appear to the computer as being within $[-f_N, f_N]$. For example, in Figure 9.10, the actual signal had a frequency of 3 Hz, which is above the *Nyquist* frequency of $f_N = 1.25$ Hz for that example. The computer treats the measured signal as if it has a frequency of $3.0 - f_S = 0.5$ Hz. Therefore, due to aliasing, the computer control system would be trying to compensate for a signal at the wrong frequency. The consequences of aliasing can be significant. For example, based on the sampling process above, the signal $\cos((2\pi / 0.4) * t)$ would be interpreted by the numeric processor as a unit amplitude constant signal.

This section has not addressed the theory of the aliasing phenomenon (see one of the references on discrete time control or signal processing). Instead, the objective of this section is to ensure that the reader understands that this phenomenon exists. Aliasing places two constraints on the design. First, the designer must understand the application well enough to judiciously specify the sampling frequency. Too large of a sampling frequency will result in unnecessary increases in the cost of the final product. Too small of a sampling frequency will either result in aliasing effects that can be very difficult to debug or too low of a system bandwidth. Second, analog anti-aliasing filters must be designed and used at the interface of each analog signal to the analog to digital converter. The purpose of the anti-aliasing filter is to attenuate frequency content above f_N so that the effect of aliasing is negligible.

Computation Delay

Time lags are of critical importance in control systems. Intuitively, delay results in the control signal being applied later than desired. Obviously, too much delay will result in performance degradation. The effect of delay can be accurately analyzed. For the designer of embedded control systems there are two important conclusions. First, analyze at an early stage in the design process, the hardware platform and processor speed relative to the phase lag that can be tolerated. Second, organize the software so that only the necessary computations occur between the time the sensor signals are sampled and the time that the control signal is output. Move all possible computations to outside of this time critical path.

9.7 Benefits of Computer-Based Control Implementations

Control systems can be implemented by either continuous time (analog component) or digital time (computer based) approaches. Since most processes that we are interested in controlling evolve as continuous variables in continuous time, and computer-based control approaches add additional complications such as quantization, overflow, aliasing, and computation delay, it is important to consider briefly the benefits obtained through embedded computer control.

Repeatability, Reproducability, and Stability

The analog components in a control system are affected by aging, temperature, and manufacturing tolerance effects. Alternatively, digital systems are inherently repeatable. If two processors are loaded with the same program and data, they will compute identical results. They are also more stable than analog implementations in the presence of aging.

Programmability

Programmability allows advanced features to be easily included in computer implementations that would be very complex in analog implementations. Examples of such advanced features include: control mode and gain switching, on-line performance evaluation, data storage, performance parameter estimation, and adaptive behavior. In addition to being programmable, computer-based control systems are easily reprogrammable. Therefore, it is straightforward to periodically upgrade and enhance the system characteristics.

9.8 Summary

This chapter introduced control systems. A control system has several components and signals, including the actuator, controller, plant, sensor, output, reference input, and disturbance. We developed increasingly complex controllers, specifically a proportional open-loop controller, a proportional closed-loop controller, a proportional-derivative (PD) closed-loop controller, and a proportional-integral-derivative (PID) closed-loop controller. There are numerous control objectives, including stability, and performance objectives such as rise time, peak time, overshoot, and settling time. These objectives may compete with one another. The more complex controllers assist us to achieve various objectives with less restrictive trade-offs between objectives. Several additional issues must be considered when using computers to implement a controller, including quantization and overflow effects aliasing, and computation delay.

9.9 References and Further Reading

- Astrom, Karl J. and Bjorn Wittermark. *Computer Controlled Systems: Theory and Design*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
- Franklin, Gene F., J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems, 3rd Ed.*, Reading, MA: Addison-Wesley, 1994.
- Marven, Craig and Gillian Ewers. *A Simple Approach to Digital Signal Processing*, Texas Instruments, 1994.
- Wescott, Tim. *PID without a PhD*, Embedded Systems Programming, Vol. 13, No. 11, October 2000.

9.10 Exercises

9.1 Explain the difference between open-looped and closed-looped control systems. Why are we more concerned with closed-looped systems?

9.2 List and describe the eight parts of the closed-loop system. Give a real-life example of each (other than those mentioned in the book).

9.3 Using a spreadsheet program, create a simulation of the cruise-control systems given in this chapter, using PI control only. Show simulations (graphs and data tables) for the following P and I values. Remember to include throttle saturation in your equations. You can ignore disturbance. (a) $P = 3.3$, $I = 0$. (b) $P = 4.0$, $I = 0$ (How do the results differ from part (a)? Explain!) (c) $P = 3.3$, $I = X$. (d) $P = 3.3$, $I = Y$. (Choose X and Y to achieve a meaningful trade-off. Explain that trade-off.

9.4 Write a generic PID controller in C.