CHAPTER

9

System Simulation and Simulink

9.1 GOALS OF THIS CHAPTER

We create models for a variety of reasons and with a range of complexity, from a general qualitative model on which to hang our thoughts to a detailed quantitative model that reflects our deepest understanding of a system. In this chapter, we are interested in quantitative systems models with elements defined by Laplace transfer functions. One of the great advantages of quantitative models is that they can be used to predict a system's response to any input. In Chapter 6, we developed frequency domain methods that could be implemented on a computer, but with the limitations that signals are periodic in steady state (or aperiodic) and system elements have no initial conditions. Considering that many real-world signals are transient (i.e., steplike) and zero initial conditions are rare, these are serious limitations. In Chapter 7, we introduced Laplace techniques that can deal with both these limitations, but the system responses were solved analytically and were not open to computer solution. Here we introduce a method that lets us have it all: a computer-based analysis of systems with nonzero initial conditions, which applies to any class of signals. This approach even lets us relax the linear and time invariant requirement. Selected nonlinear elements and elements that change their properties over time can be included in the system model. In this most useful approach, a continuous system operating in the continuous domain is simulated on a digital computer.

Digital simulation is an outgrowth of an early computation device known as an "electrical analog computer." Such computers used continuous variations in voltage to represent signal levels in a system. Electrical components were used to represent system processes and were capable of performing linear operations such as summation, scaling, and integration.[1] As they operated in the continuous domain, they did not suffer from quantization error, but did suffer from electronic noise (see Section 1.3.2.1). Programming these computers required manually connecting the electronic components together with patch cables (flexible wires with plugs at both ends), a time-consuming but rewarding effort.

Digital simulation is sort of a mix between the discrete and continuous domains. It is implemented on a computer, so the fundamental calculations are digital, but it mimics a

[1]A circuit that does summations of signals is described in Chapter 15 (Section 15.8). Integrator circuits use capacitors that produce voltages that are time integrals of their current (Chapter 12, Section 12.4.1.2).

continuous system. Like the Laplace transfer function, it is used to describe the behavior of a continuous system, but with much less effort. Simulation programs such as MATLAB's Simulink are basically digital computer representations of the now-extinct analog computer.

In this chapter, we explore the capabilities of MATLAB's simulation program, specifically we perform the following:

- Explain the principles of simulating continuous systems on a digital computer.
- Learn the basics of Simulink including some of its more useful options.
- Explore some of Simulink's extensive features that are particularly relevant to biomedical engineers.
- Apply Simulink to several biological and nonlinear systems.

9.2 DIGITAL SIMULATION OF CONTINUOUS SYSTEMS

Conceptually, digital simulation of a continuous system is straightforward. Information flows through the simulated system one instant (i.e., time step) at a time. Both time and amplitude are still quantized, but the time and amplitudes steps are made small enough to appear continuous (for all practical purposes). The basic idea is similar to that used in convolution: calculate responses to a number of small time slices of the input signal. The main difference is that the response of each element in the system is determined individually. Given the element's input(s) for a specific time slice and the operation performed by the element, its output is calculated. Then, for that same time slice, the element's response becomes the input to any element(s) connected to it. This proceeds until the outputs of all the elements, and the system itself, have been determined, and then begins again for the next time slice.

Figure 9.1 illustrates the digital simulation process for two elements assumed to be a subsection of a larger model. Five time slices are shown. The process begins at the first time slice

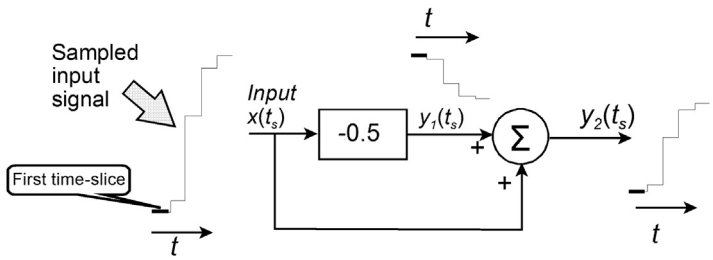


FIGURE 9.1 Two elements that might be part of a larger system showing hypothetical inputs and resultant outputs. The first time slice is shown as darker than the subsequent four time slices. Simulation software takes the value of the input to this subsystem, $x(t_s)$ during the first time slice and applies the specified scaling operation of the first element, in this case multiplying by -0.5. This produces an output $y_1(t_s)$, which feeds the summation element. The summation element adds $y_1(t_s)$ to $x(t_s)$ to produce the output $y_2(t_s)$, again, all during the first time slice. These steps are then repeated for all the elements of the system. For the next time slice, a new input value, $x(t_s)$, arrives at the input and the calculations proceed through the system. This process repeats until the last input time slice at the end of the input signal. If this were part of a larger system, $y_2(t_s)$ would feed additional elements.

(Figure 9.1, shown in bold) when a signal value is taken from the subsystem input and becomes the input to the first element. It is also the input to the summation element. This input comes from the output of another subsystem or a simulated signal source. The signal value at the first time slice is $x(t_s)$, at $t_s = 0$. One of the elements receiving this signal is a scaling element that multiples the input by -0.5 and produces $y_1(t_s{=}0)$, also shown in bold. The signal $x(t_s{=}0)$ is also sent to a summation element, which adds it to the output of the scalor. This summation element produces an algebraic sum of the two values ($x(t_s{=}0) + y_1(t_s{=}0)$) as its output, $y_2(t_s{=}\ 0)$.

The progression of the input to output of the signal continues through other system elements (not shown) until it reaches the last element in the system. The response of this last element is the output of the system and is usually displayed or recorded. This whole process is then repeated for the second time slice, the one immediately following the bold slice. After the second time slice propagates through all the elements, the cycle is repeated for successive time slices until the input signal ends or some other stopping criterion is met. During simulation, the output signal and the response of each element evolve over time just as in a real continuous system (admittedly in discrete steps, but these should be small enough that the operation appears continuous). Generally, it is possible to view the response of any element in the system, as this evolution takes place.

At the heart of continuous simulation, analog or digital, is the integrator. In the Laplace transfer function, there is integration behind every occurrence of $1/s$ (or $1/j\omega$ for a frequency domain transfer function). For example, a $1/j\omega$ by itself is just an integrator (Equation 6.29) and a $1/(1{+}j\omega)$ is an integrator in a feedback loop (e.g., Example 6.3). A second-order transfer function represents a system with two integrators (e.g., Example 6.5). Thus to simulate the behavior of a system, you need to perform integration, and on a computer, you need to perform integration digitally.

Previously, when we transformed a continuous equation to a discrete version, if it contained an integral we replaced it with summation. Summation is the digital equivalent of integration. But simulation programs want to mimic continuous systems, so they use sophisticated algorithms that behave more like true integrators. Simulink has a number of different integrator algorithms. These algorithms trade off speed, stability, and accuracy, but, as is often the case, the default integrator works well for most problems. Nonetheless, we will write a simple simulation routine using summation as an integrator to simulate the response of a first-order system to a step function.

EXAMPLE 9.1

Write a program to simulate the response of the continuous first-order system shown in Figure 9.2 to a step input.

In Chapter 7 we showed that this system has the transfer function:

$$TF(s) = \frac{1}{s + \omega_1} = \frac{1/\tau}{s + 1/\tau} \tag{9.1}$$

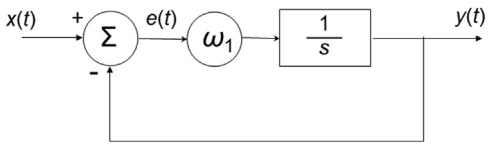


FIGURE 9.2 Simple first-order system whose output to a step function is found in Example 9.1 and later in Example 9.2.

where ω_1 is the cutoff frequency and equals $^1/_\tau$ (Equation 7.26). Make the cutoff frequency 5 rad/s ($\tau = 0.2$ s) and show the first 1.0 s of the response. Assume that the initial condition of the integrator is 0.0.

Solution

We use summation for the integration process. We divide the simulation time of 1.0 s into 100 time slices, so each time slice represents 10 ms. For the first step, the output of the integrator is the initial condition (which in this case is 0.0, but it need not be). After that, it produces the output of the next step, which is the summation of its last input with its last output.

```
% Example 9.1 Crude digital simulation of a first-order system

%
Ts = 0.010;                         % Step size 10 ms
N = 100;                            % Number of steps (Tt = 1 s)
w1 = 5*Ts;                          % Cutoff frequency of first-order system
out(1) = 0;                         % Initial condition for integrator

x = [0, ones(1,N-1)];               % Input (step function)

y = zeros(N); % Output array

for k = 1:N-1

  e(k) = x(k) - out(k);             % Error signal
  out(k+1) = w1*e(k) + out(k);      % Integrator summation
end
t = (1:N)*Ts;                       % Time vector for plotting
    ........plotting and labels........
```

Results

Both the system output, $y(t)$, and the integrator input, $e(t)$, are shown in Figure 9.3. As expected from our analytical solutions (see Example 7.3), the output is a smooth exponential with a time constant of 0.2 s (i.e., the response is 0.63 of its final value at 0.2 s). Note that it is easy to add in a nonzero initial condition for the integrator: just make y(1) equal to the desired initial condition. Similarly, you can think of ways to embed nonlinear elements into this program. An example is found in the problems.

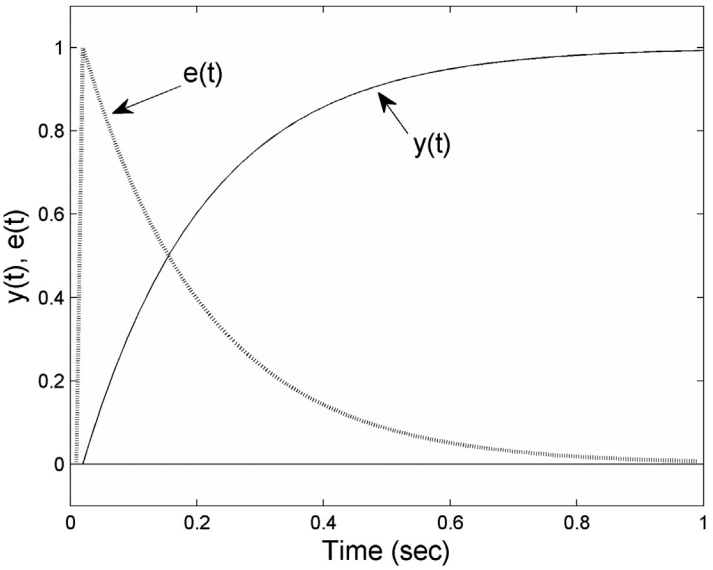


FIGURE 9.3 Output from the simulation of the first-order system. Both system output, $y(t)$, and the difference signal, $e(t)$ (for "error" signal), are shown.

MATLAB's simulation program, Simulink, is bit more sophisticated than the approach taken in Example 9.1. For starters, the system is defined with the aid of a Graphical User Interface (GUI) that not only simplifies model specification but also provides a nice diagram of the systems (suitable for publication). Unless instructed otherwise, Simulink uses time slices that vary in width depending on the dynamics of the system. Unlike signal sampling, there is no reason to keep step size constant, and Simulink varies the size of the time slice to match the dynamic properties of system elements. When the signals are changing quickly, the step size decreases to improve accuracy, but when signals change slowly the step size increases. Not only does this decrease computation time but by reducing the number of steps required during the simulation also reduces error propagation.[2] Finally, as mentioned above, Simulink uses sophisticated integration algorithms that behave more like real-world continuous integrators.

9.3 INTRODUCTION TO SIMULINK

All linear systems can be decomposed (at least in theory) into elements that perform addition or subtraction, scaling (i.e. gain elements), differentiation, and integration. In fact, it is

[2]Roundoff errors, although small for any given calculation, build up over the many steps involved in a simulation that requires short time slices over an extended time. It was this problem of error propagation that occurred in so-called "wide bandwidth" simulations (i.e., fast dynamics with long simulation times) that kept analog computers alive long after digital simulation became popular.

possible to rearrange systems to eliminate derivative elements, but this is not necessary when you use Simulink. Any simulation program must perform a number of tasks: (1) it must allow you to define the system you want to simulate including initial conditions, (2) generate the input(s) to the system and set up signals for display, and (3) calculate and track the responses of the individual elements as the signals flow through the system and display and/or record the system's output. These steps are detailed below:

Step (1) Setting up the model and initial conditions. A systems model is a collection of interconnected elements, so to set up a model you need to specify the elements and their interconnections. In systems models, interactions are unidirectional and explicitly indicated, usually by an arrow. In Simulink, model setup is done graphically: elements are selected from a number of libraries and the connections between them specified by dragging lines between the various elements. Those elements that have initial conditions can be adjusted after they have been specified. A step-by-step procedure is given in the next example.

Step (2) Generating input waveforms and setting up the output display. In Simulink, the input signal usually comes from a waveform-generating element, which is chosen from the library just like all the other elements. Simulink has a wide range of waveform generators stored in a library called Sources. It is even possible to generate a waveform in a standard MATLAB program and pass it to your model as an input signal. Outputs are also just elements selected from the library called Sinks. [3] Output elements can be attached to any element in the system and include time displays, outputs to a MATLAB routine, or outputs to a file.

Step (3) Tracking the responses of the individual elements and displaying the output. This is handled by Simulink so that once the simulation is initiated, it runs automatically for the requested duration. Simulink goes to great lengths to optimize step size by using special integration algorithms. In most simulations, including all of those covered here, we need not be concerned with integration techniques and step size manipulation. If needed, we can alter the simulation parameters, including the algorithm used for integration, and sometimes these alterations are useful to create smoother response curves. In addition we may want to request evenly spaced step sizes if we want to compare simulation results directly with time sampled data.

9.3.1 Model Specification and Simulation

The best way to learn Simulink is by doing, or through example. While the examples below provide a rather basic introduction to Simulink's extensive simulation capabilities, they cover the range of applications that most bioengineers need. As always, detailed instructions can be found in MATLAB's Help.[4] A simulation starts by entering simulink in MATLAB's main window. This opens a window that shows the libraries of possible elements (Figure 9.4). A list of the most important element libraries (for us) and the types of elements they include are given in Table 9.1. The opening window File tab allows saved

[3]Output and display devices are called "sinks" because the output signal flows into, and ends at, these elements just as water flows into a sink and disappears.

[4]Click on "Help" in the main MATLAB window, then select "Product Help" in the drop-down menu, and then scroll down the list on the left side and select "Simulink." The main Simulink help page will open and a large variety of helpful topics will be available.

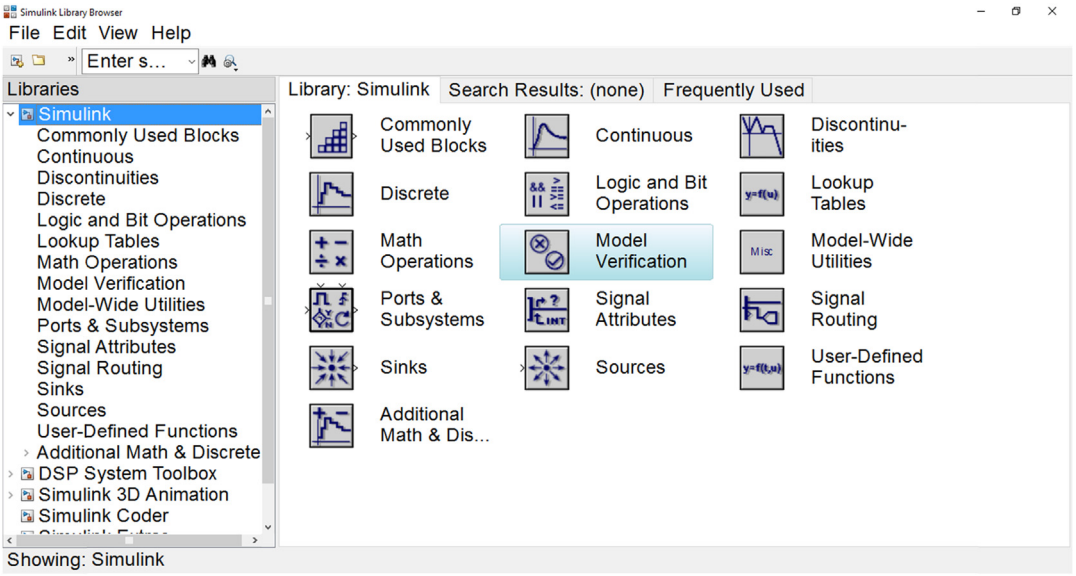


FIGURE 9.4 The initial Simulink window showing a list and icons for the various element libraries. The libraries most important to bioengineers are presented in Table 9.1 along with their most useful elements.

TABLE 9.1 Useful Simulink Libraries

Library	Element Name	Function
Sources (23 elements)	Constant	Constant output with adjustable value.
	Step	Step output with adjustable onset time, initial and final values.
	Ramp	Ramp output with adjustable onset time, slope, and initial value.
	Sine	Sine wave with adjustable frequency, phase, and amplitude.
	Pulse Generator	Pulse generator with adjustable pulse rate, width, and amplitude.
	Chirp	Chirp with adjustable start and stop frequencies and duration.
	Random	Gaussian random number with adjustable variance and sample time.
	Repeating Sequence	Outputs an adjustable repeating sequence of levels set by table.
Sinks (9 elements)	Scope[a]	Display signal with axes set interactively.
	To File	Sends signal to a specified file as a time series
	Out1[a]	or other.
	To Workspace	Sends signal as to a MATLAB program.
		Sends signal as time signal or other to workspace as specified variable.
Continuous (13 elements)	Integrator[a]	Output is integral of input.
	Transfer Fcn	General transfer function with any number of adjustable numerator
	Delay	and denominator coefficients.
		Adjustable time delay.
Math operations (37 elements)	Add	Adds or subtracts any number of signals.
	Sum[a]	Same as add, but with circular symbol
	Gain[a]	Multiplies signal by adjustable constant.
	Product[a]	Takes the produce of two signals.
	Math Function	Performs various math operations, including log, power, and exponential.
Discontinuous (12 elements)	Saturation[a]	Limits signal amplitude to adjustable minimum and maximum.
	Rate Limiter	Limits signal rate of change to adjustable minimum and maximum.
	Dead Zone	Outputs zero for signals within adjustable dead zone.

[a]*Also found in the "Commonly Used Blocks" library*

models to be loaded. Models can be saved at any time from the model window as explained below.

Example 9.2 provides a step-by-step demonstration of its use to determine the response of a second-order system to a step input.[5] Similar instructions can be found in MATLAB's help documentation following the links *Simulink*, *Getting Started*, *Creating a Simulink Model*, and ending up at *Creating a Simple Model*.

EXAMPLE 9.2

Use Simulink find the step response of the simple first-order system used in Example 7.1.

$$TF(s) = \frac{1}{s+\omega_1} = \frac{1/\tau}{s+1/\tau}$$

where $\omega_1 = 5$ rad/s.

Solution

Although Simulink provides a way to represent this first-order system as a single element, here we use an integrator and gain element in a feedback loop and shown in Figure 9.2. The Simulink graphical interface is used to set up the system model using mouse manipulations similar to those of other graphic programs such as PowerPoint. A model window is opened first, and then elements are installed in the window and moved by dragging. They can also be duplicated using Copy and Paste.

Step (1) Set up the model window. To create a system model, open Simulink by typing simulink in the MATLAB command window. The window that is opened by this command is shown in Figure 9.4, which also contains the Simulink Library Browser window. Use the File drop-down menu and select New and then Model. This opens a blank window called "`untitled1`." If you save this window as a model file (`name.slx`)[6], the name you choose appears in the title. The model window can be saved, reopened, and resaved just as with any other MATLAB program file.

Step (2) The next step is to populate the model window with the desired elements. We have a diagram of the system in Figure 9.2, which shows it contains three elements: a summation, a gain term, and an integrator. We also need the step input and some sort of output. Normally, we would add a Scope element for displaying the output. (For publication purposes, we actually output the data to MATLAB workspace then plot, but the Scope element is easier to use and gives us the signal immediately.) Table 9.1 indicates that all of these can be found in the Commonly Used Blocks library except for the step input, which is in the Sources library. To open a library, double-click on the library icon. Double-clicking on the Commonly Used Blocks library opens the window shown in Figure 9.5.

Next we drag the desired element(s) into the model window creating a duplicate of that element in the model window. Dragging the Sum, Gain, Integrator, and Scope elements onto our model window gives Figure 9.6. The elements could be placed anywhere, but it is reasonable to put them in approximately the same general position as in Figure 9.2.

For the step input signal, we turn to the Sources library, Figure 9.7, and drag out the Step icon found in the lower left corner.

[5]The Simulink examples shown here use MATLAB release R2013a, version 8.1.

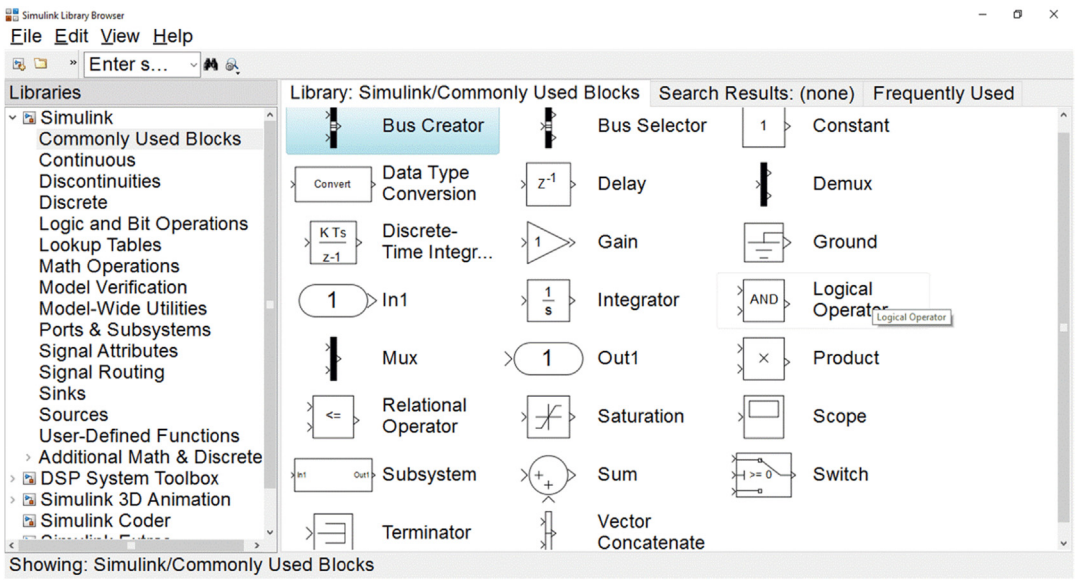


FIGURE 9.5 The Commonly Used Blocks library. All of the elements required in Example 9.2 can be found here except the step input.

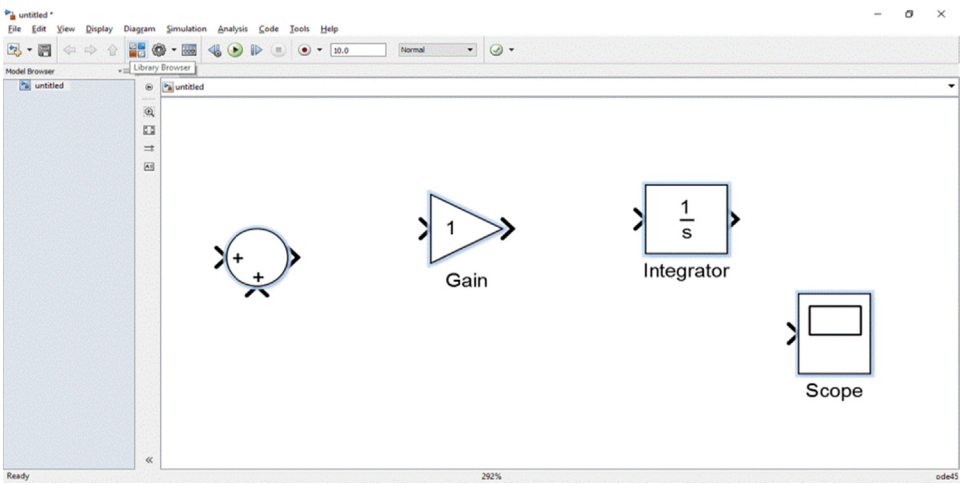


FIGURE 9.6 Three of the elements needed for the model used in Example 9.2 placed in approximately the same position as in the system diagram of Figure 9.2.

Step (3) Adjust element parameters. Most elements have parameters. In some case the default values are fine, but often they need to be modified. In our model, the Integrator element has an initial value parameter, the Gain element has a gain value, the Step element has initial and final values, and the Sum element parameters specify the signs and number of inputs. For the Integrator element, the default initial condition value is zero, which is appropriate for our system. As seen in

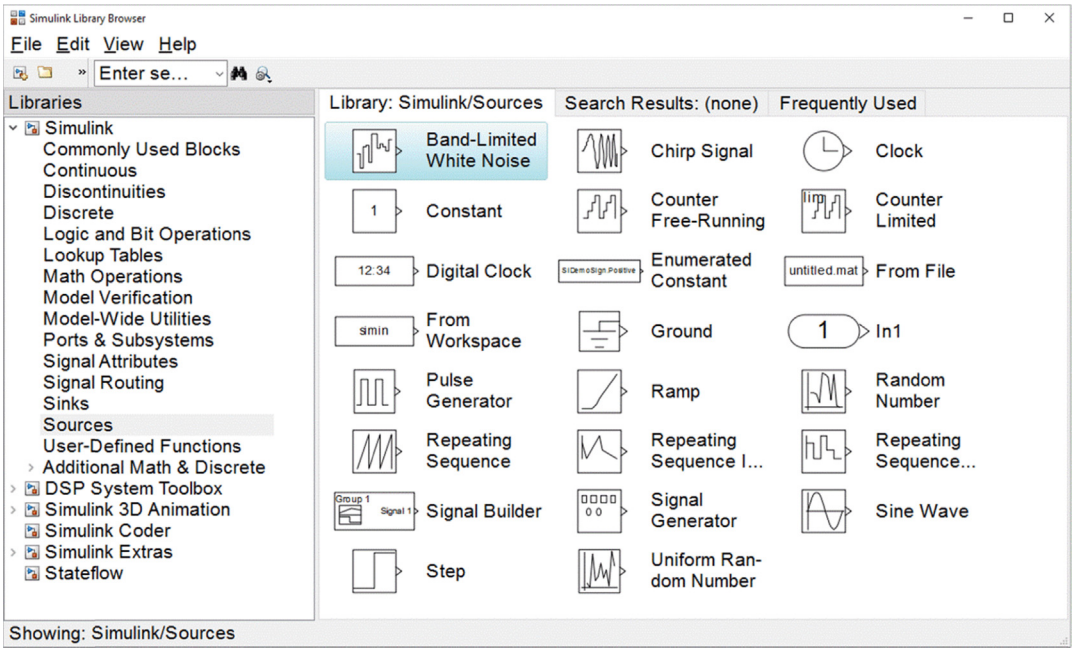


FIGURE 9.7 The Sources library window. The Step element is in the lower left corner.

Figure 9.6, the default signs for the Sum element are two positive inputs as indicated by the two "+" signs on the Sum icon. We need an element that takes one positive and one negative input, Figure 9.2. To change the "+" to "−" we double-click on the Sum icon, which opens the element's parameter window as shown in Figure 9.8A. The List of signs: line shows two "+" symbols. We simply backspace over the second "+" symbol and add a "−" symbol (Figure 9.8B). This makes the second (i.e., lower) input negative as shown in the final model in Figure 9.10.

The Step element parameter window is shown in Figure 9.9A. The default initial value is 0.0 and the final value is 1.0, which is fine for our simulation. However, the default Step time is 1.0 and we change it to 0.01 to match the small onset delay of our homegrown simulation program used in Example 9.1 The Gain element also needs adjustment as the desired gain value is 5.0 and the default value is 1.0 as seen in Figure 9.6. The Gain element parameter window is shown in Figure 9.9B where the Gain value has already been changed to 5.0.

Step (4) Connect the elements. This is done graphically by clicking on an element output and dragging the line to the desired element input, or vice versa. The complete, connected system is now in the model window as shown in Figure 9.10. A second Scope has been added to display the error signal. (Elements can be duplicated using copy-paste as in any graphics program.) The model can be saved any time using Save As… command under the File pull-down menu. (See Footnote 6 regarding computability issues for saved models.)

Step (5) Run the simulation. You can start the simulation simply by clicking the run icon found on the second bar of the model window shown in Figure 9.11. You can also use the Simulation pull-down menu found on the top bar of the model window and which displays a similar run icon.

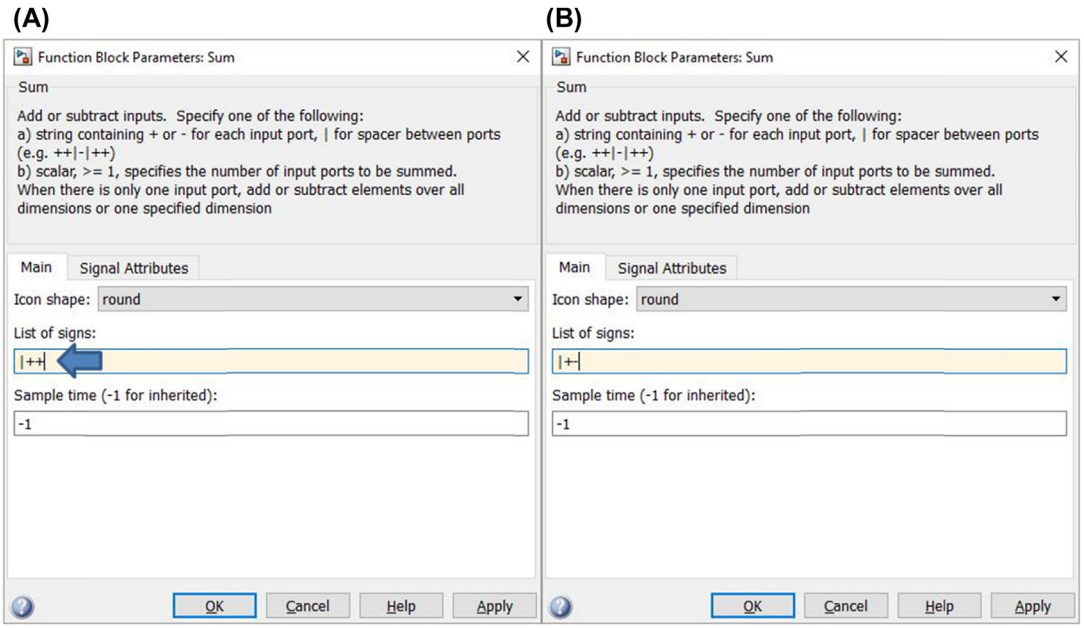


FIGURE 9.8 The parameter window of the Sum element. A) The List of signs: line (arrow) shows two '+' symbols, the default for this element. B) The second sign has been changed to a '−' sign which will make the second input a negative.

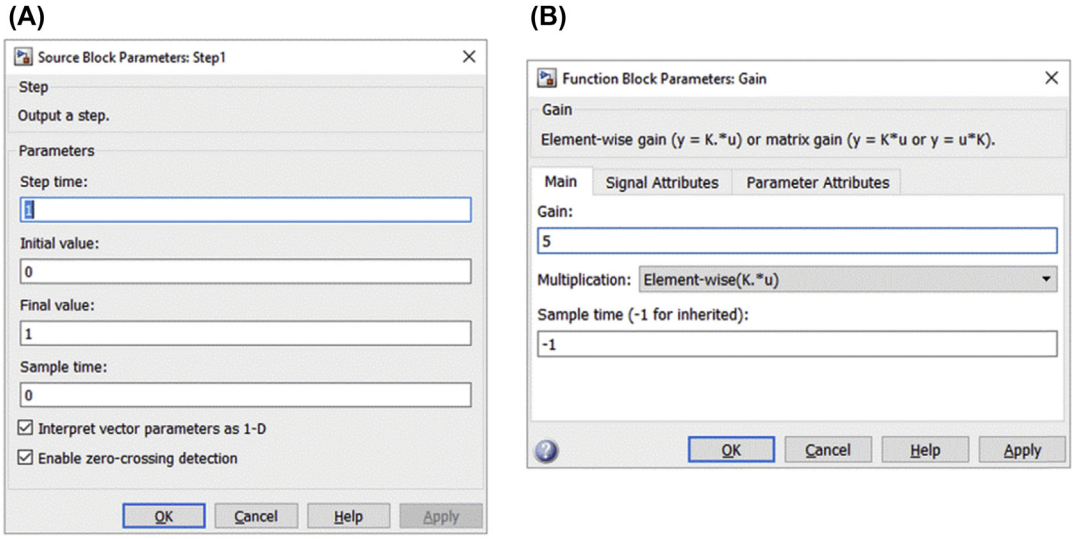


FIGURE 9.9 A) The Step element parameter window shows the default values for initial and final step valued to be 0.0 and 1.0, respectively. These are okay for our simulation; however, the Step time needs to be changed to 0.1. B) The Gain element parameter window where the Gain value has already been changed to the desired value of 5.0.

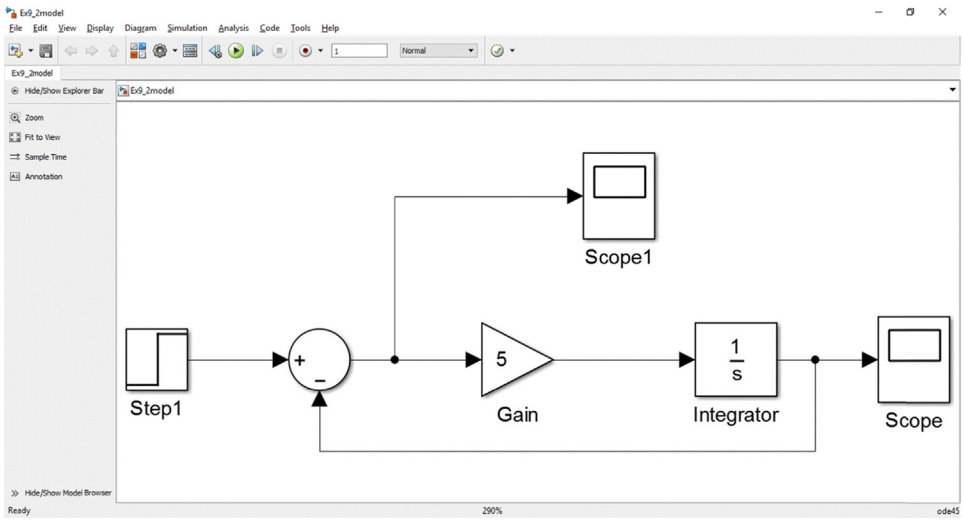


FIGURE 9.10 The completed model used in Example 9.2. A second Scope has been added to display the error signal. Note that the model layout is similar to that of the original system presented in Figure 9.2.

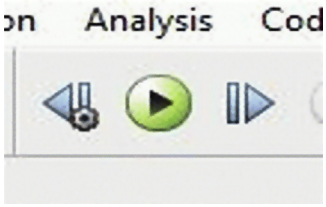


FIGURE 9.11 The Run icon found on the second bar of the model window shown in Figure 9.10. Clicking this button initiates the simulation.

In older versions of MATLAB there is no run icon, but the Simulation pull-down menu displays the word "Run."

The default time of the simulation is 10 (seconds).[7] To change the simulation run time along with many other simulation options, you can open the Model Configuration Parameters window also found on the Simulation pull-down menu (Figure 9.12). The first line of this window is labeled Simulation Time and has entries for start and stop time. In Figure 9.13, the stop time has been changed to 1.0 s. Alternatively, you can change the number that appears in the window on the second line of the simulation window (barely visible in Figure 9.10).

Results

The output of the model can be viewed by double-clicking the Scope element. The resulting window is a plot of the signal to which the scope is attached. The display features a light colored line against a dark background. As this does not reproduce well, the graph shown in Figure 9.13 is constructed from variables placed in MATLAB workspace using the simout element. This approach produces standard MATLAB plots and is fully described in the next example.

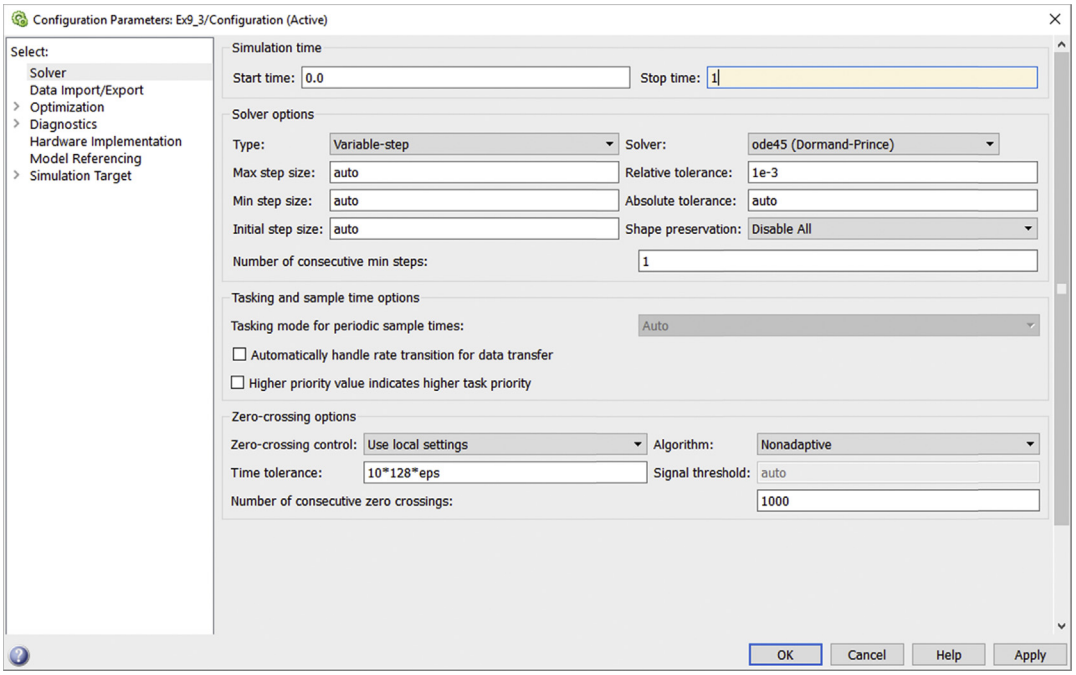


FIGURE 9.12 The Model Configuration Parameters window found on the Simulation pull-down menu can be used to set the start and stop time of the simulation along with other simulation parameters.

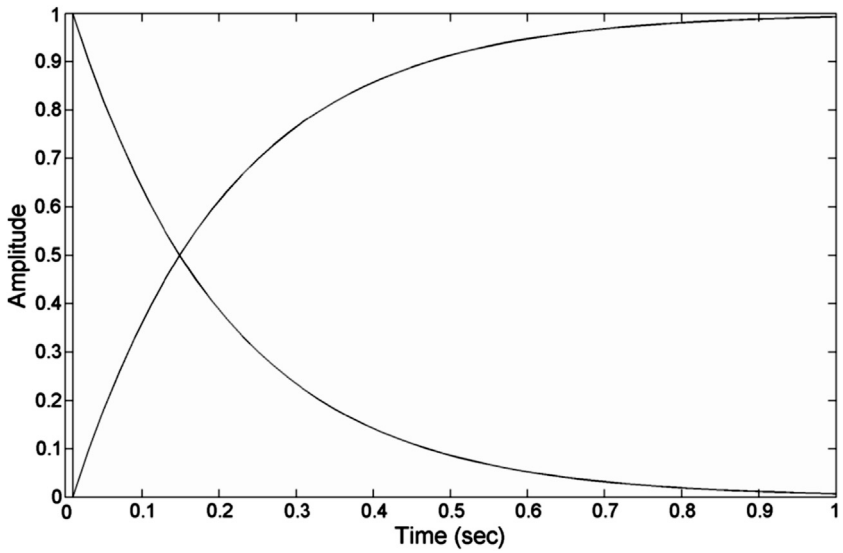


FIGURE 9.13 The output and error signal produced by simulation in Example 9.2. The two signals are very similar to those of Figure 9.3 that were produced by our homebrewed simulation program in Example 9.1.

[6]For MATLAB versions 2012b and earlier, models were stored differently under the .mdl extension. Newer versions of Simulink can load .mdl files and save them in this older format using Export Model in the File pull-down menu. However, earlier versions of Simulink cannot read the newer .slx files.

[7]Simulink time units are generic: they reflect the units used in the model parameters. We generally work in seconds, so Simulink time units can usually be taken as seconds. Some very slow biological processes such as the glucose response simulated in Example 7.5 use hours as the basic time unit.

The next example shows how to use the powerful Transfer Fcn element to simulate any transfer function and the useful simout element for displaying the results.

EXAMPLE 9.3

Simulate the impulse response of the fourth-order transfer function analyzed in Example 7.9 and repeated here:

$$TF(s) = \frac{s^3 + 5s^2 + 3s + 10}{s^4 + 12s^3 + 20s^2 + 14s + 10} \tag{9.2}$$

Solution

Follow the same steps used in the last example.

Step (1) Set up the model window. Enter simulink in the main MATLAB window and use the File pull-down menu to open a new model.

Step (2) Populate the model. We need only three elements to simulate and display this system: a Source that generates an impulse, the Transfer Fcn element to represent the transfer function, and simout to place the output signal in workspace. For an impulse we drag out the Pulse Generator element from the Sources library. We set the pulse frequency to be greater than the simulation time so we get only one pulse. We set the pulse width to be very short, as we want to mimic an impulse. The Transfer Fcn element is taken from the Continuous library,[8] and the simout element from the Sinks library. The resultant model is shown with connections in Figure 9.14

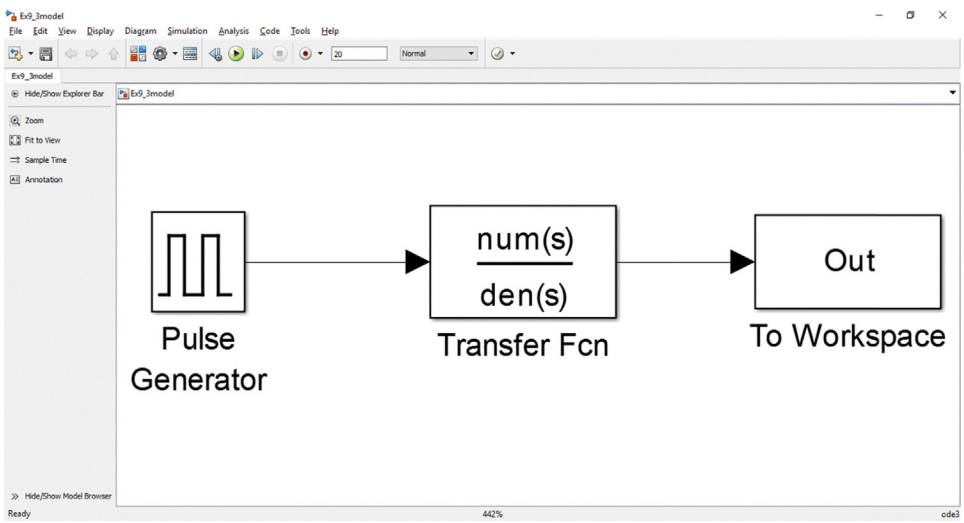


FIGURE 9.14 The model used in Example 9.3 to simulate the impulse response of a fourth-order transfer function.

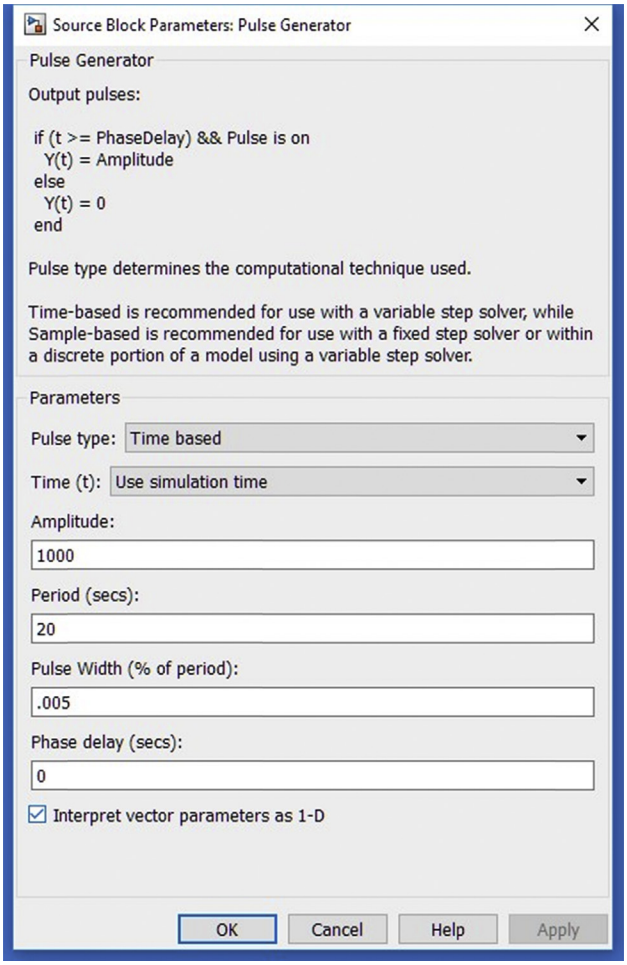


FIGURE 9.15 The Pulse Generator parameter window showing the new values; Pulse Width (% of period) is .005%, Amplitude is 1000, and Period (seconds) is 20.

Step (3) Adjust element parameters. The Pulse Generator window is shown in Figure 9.15. Initially, the default simulation time of 10 s was used, but a longer time period was needed to show the complete response. The simulation time was changed to 20 s and the Period (seconds) is set to 20, so only one pulse would be produced. Using the default Phase delay (seconds) of 0 ensures that the single pulse occurs at $t = 0$. The Pulse Width (% of period) is set to 0.005% to produce a short pulse. As the period is 20 s, this produces a 1.0 ms pulse. To determine if the pulse input realistically represents an impulse, we used the trick described in Example 5.1. We shortened the impulse and noted minimal change in the shape of the response, indicating that our 1.0 ms pulse is acting like an impulse to this system. For a unit impulse the area under the pulse should be 1.0 so we set the amplitude to 1000.

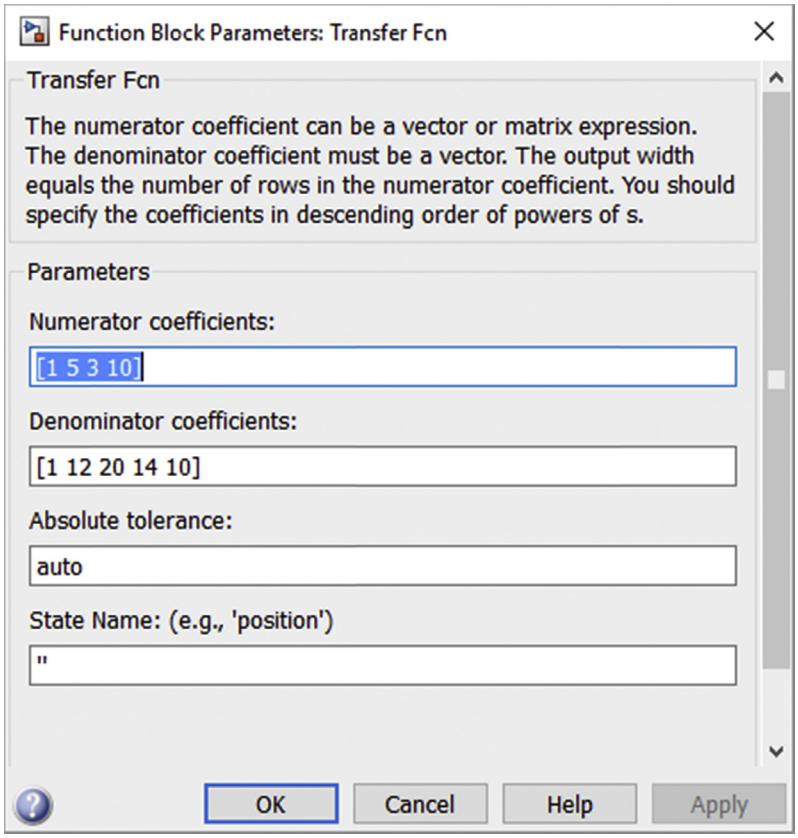


FIGURE 9.16 The Transfer Fcn element parameter window. The numerator and denominator coefficients of Equation 9.2 have been entered on the first two lines.

The Transfer Fcn element parameter window must be set to represent the transfer function in Equation 9.2. This is accomplished by entering the numerator and denominator coefficients as vectors in the parameter window. The numerator coefficients array is [1 5 3 10] and the denominator array is [1 12 20 14 10]. These coefficients have been entered in the Transfer Fcn parameter window shown in Figure 9.16.

The To Workspace element parameter window must also be modified to place the data in the workspace in the appropriate format. Specifically, we need to modify the Save format: pull-down menu near the bottom of the parameter window (Figure 9.17). This menu is used to change the format from the default Timeseries to Array. This element then produces two standard MATLAB arrays: tout, which is the time array, and Out, which is the response array. To plot the response, we simply enter plot(tout,Out); in the main MATLAB window.

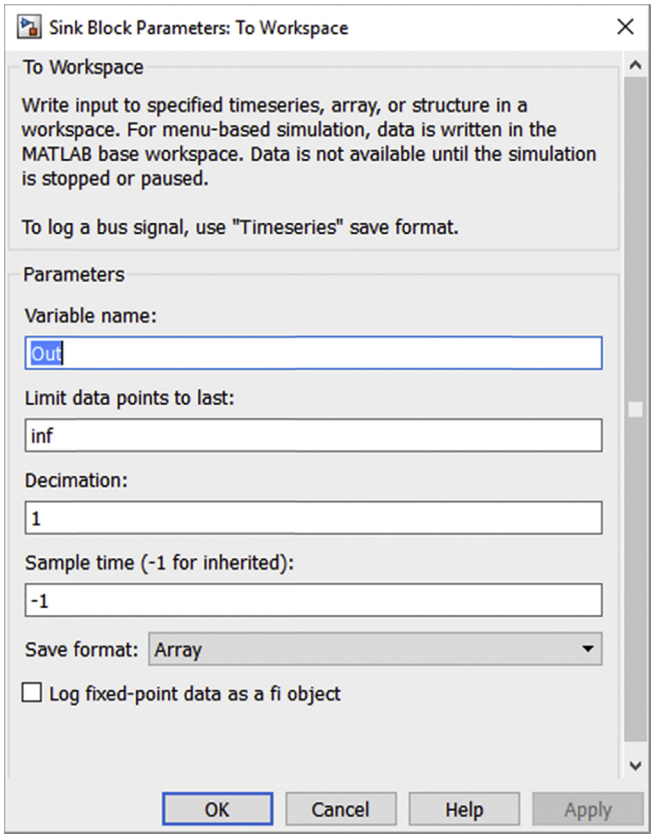


FIGURE 9.17 The To Workspace parameter window. The Save format has been changed from the default Timeseries to the Array format. The name given to the output variable in the workspace is "Out" as indicated by the Variable name: parameter. (Often we just use the default name of "simout.")

Step (4) Connect the elements. Because the model consists of only three elements, this is performed in Step 2 when the model window is populated.

Step (5) Run the simulation. After setting the simulation time to 20 s, we use the run icon to start the simulation.

Results

After the simulation is run, the output can be found as vector Out in the workspace along with the time vector tout. As mentioned above, the response can now be plotted using the command plot(tout,Out);. Axis labels were added resulting in Figure 9.18.

[8]As linear systems deal with signals that are theoretically continuous in time, simulation of LTI systems is sometimes referred to as "continuous simulation" although the simulation approach is still based on discrete time slices. This continuous simulation terminology is the basis of the name MATLAB used for this library.

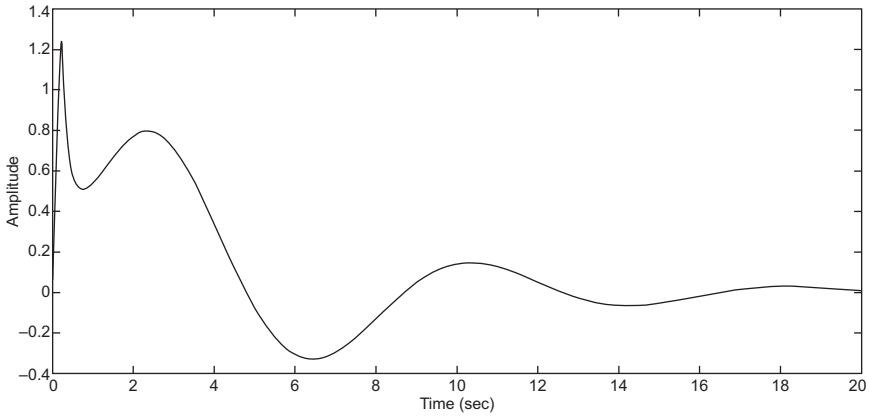


FIGURE 9.18 The impulse response of the system defined by Equation 9.2 obtained through simulation.

As we have the impulse response in the workspace, we can find the spectrum of this system by taking Fourier transform of this response. This is done in the next example.

EXAMPLE 9.4

Find the magnitude and phase spectrum of the system represented by the transfer function given in Equation 9.2. To conform to the spectral plots generated in Example 7.9 and shown in Figure 7.12, plot the magnitude spectrum in dB versus log and the phase spectrum in deg, both versus log frequency. Also, limit the frequency axis to between 0.1 and 100 Hz.

Solution

We have a simulation of the impulse response, so we should, in principle, be able to determine the spectrum from the Fourier transform of this response. However, in the default mode, Simulink does not use equal time spacing: the time spacing is optimized based on the model dynamics. To set Simulink to produce signals with a fixed time interval, we use the Model Configuration Parameters window found in the Simulation pull-down menu on the model window. This window is shown in its default configuration in Figure 9.12 (except for Stop Time, which was set to 1). The first entry below Type shows Variable-step, the default step mode. Use this pull-down menu to select Fixed-step (the only other option) as shown in Figure 9.19. This opens a line titled Fixed-step size (fundamental sample time): which we set to 0.001 s making $f_s = 1$ kHz.

Because we have changed to a fixed-step time, we also need to change the Pulse Width (% of period) value. We want to ensure that we have a true digital impulse; that the Pulse Generator has a nonzero output for only the first step. The duration of one sample is 0.001 s, so the pulse width should be 0.00005 (20) = 0.001. Because the pulse width variable is in percent, we should make Pulse Width (% of period) entry 100 (0.00005) = 0.0050. (Note: to check this, use simout to put the output of the Pulse Generator in the workspace and make sure only the first term is nonzero. Also

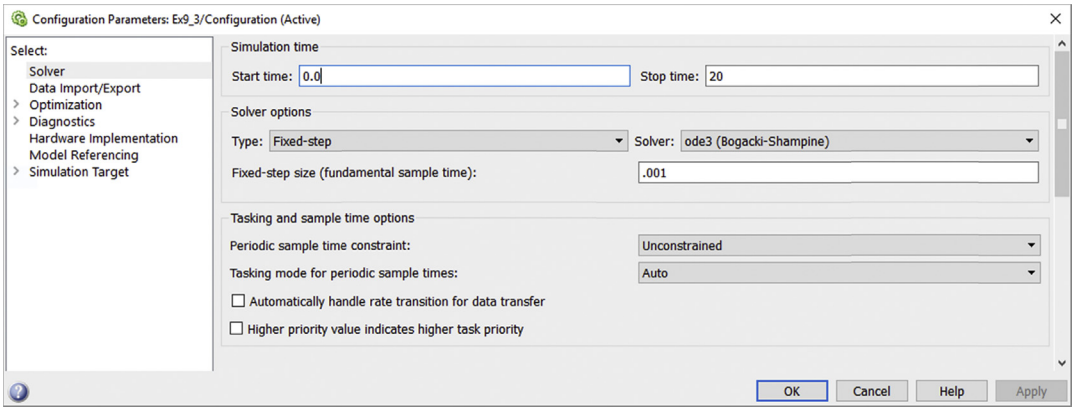


FIGURE 9.19 The Model Configuration Parameters window after modification to produce a simulation with a fixed-step size of 0.001 s ($f_s = 1$ kHz).

when using the fixed-step mode, Simulink requires that the pulse width should be an integer value of the step size: in this case the integer is 1.0.)

After these modifications, we run the model using the Run button (Figure 9.11), as before. This produces an impulse response as variable Out in the workspace. We then write a short MATLAB program to take the Fourier transform of this variable and calculate the magnitude and phase spectra, and plot.

```
% Example 9.3 Find the magnitude and phase spectrum of the transfer function
% given in Equation 9.2. Assumes data in vector Out

%
fs = 1000;                      % Sample frequency
N = length(Out);                % Get data length
N_2 = round(N/2);               % N/2 (valid spectral points)
f = (1:N)*fs/N;                 % Frequency vector for plotting
X = fft(Out);
Mag = 20*log10(abs(X));         % Magnitude spectrum in dB
Phase = angle(X)*320/(2*pi);    % Phase spectrum
subplot(2,1,1);
semilogx(f(1:N_2),Mag(1:N_2),'k','LineWidth',1); % Plot magnitude spectrum
.......label and repeat for phase spectrum.......
```

Result

This program produces the plots shown in Figure 9.20. Comparing these plots to those of Figure 7.12, we see the same general shape with dips in both magnitude and phase curves around 1.5 rad/s. The curves found by simulation are not as smooth as those found in Example 7.9. This is to be expected because the former are developed indirectly from a simulation of the time-domain response, whereas the latter were found directly from the frequency domain transfer function. However, as shown later, when a system contains nonlinear elements, simulation can often be used to determine both the time and frequency domain behavior where all our linear methods fail.

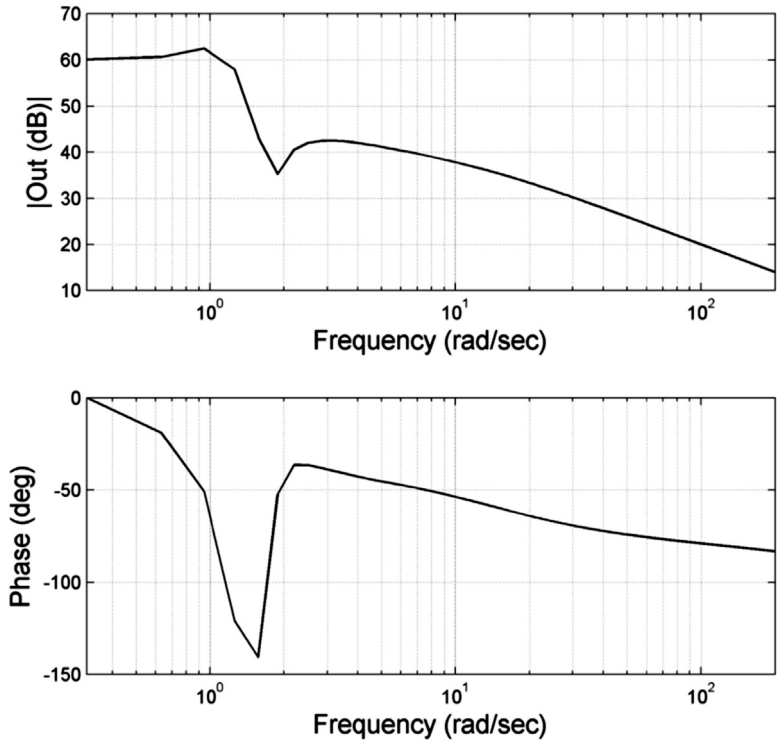


FIGURE 9.20 Spectra of the system defined in Equation 9.2 found by applying the Fourier transform to an impulse response generated by simulation.

9.3.2 Complex System Simulations

Even very complex systems are easy to analyze using Simulink. The next example applies the same methodology described in Example 9.2 and 9.3 to a somewhat more complicated system.

EXAMPLE 9.5

Simulate the step response of the system shown in Figure 9.2. Also simulate the response to a 200-ms pulse. Then determine the pulse width that represents an impulse input. Display not only the output signal but also the contributions of the upper and lower pathways.

Solution

Construct the model shown in Figure 9.21, adding a step input and displays. After simulating the step response, replace the step function element with a pulse generator element. Adjust the pulse width so that it appears as an impulse to the system using the strategy used in Example 9.3.

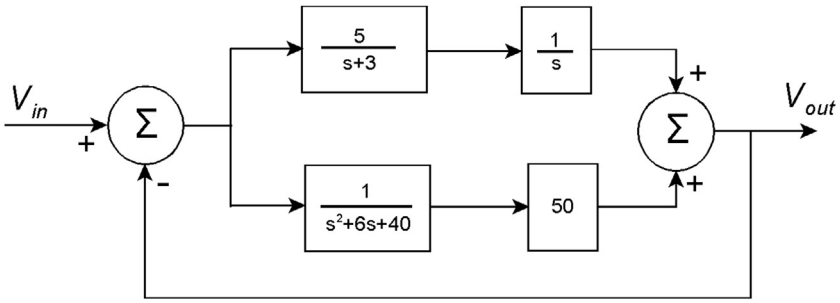


FIGURE 9.21 System simulated in Example 9.5

Step (1,2) From the Continuous library, we drag out three elements: two Transfer Fcn elements and an Integrator. From the Commonly Used Blocks library, we drag out two Sum elements (one will be changed to a subtraction element), a Gain element, and three display elements from the Sinks library. Again, we use simout to place the output signals in the workspace. Then we go to the Sources library to pick up the Step element. The resulting model window appears as shown in Figure 9.22, where the elements have been dragged into the window more or less randomly. The element parameters have already been modified in this figure.

Step (3) Next we modify the element parameters. As in Example 9.3, the transfer function elements must be set with the coefficients of the numerator and denominator equations. The element in the upper path in Figure 9.21 has a transfer function of $5/(s+3)$, giving rise to a numerator term of 5 (this is a scalar, so no brackets are needed) and a denominator term of [1, 3]. The transfer function in the lower branch, $1/(s^2+6s+40)$, has a numerator term of 1 and a denominator term of [1, 6, 40]. The gain element parameter, Gain:, should be set to 50. The Sum elements have a textbox labeled List of signs: in one of the two elements the '+' symbols should be changed to a '−' sign, so the

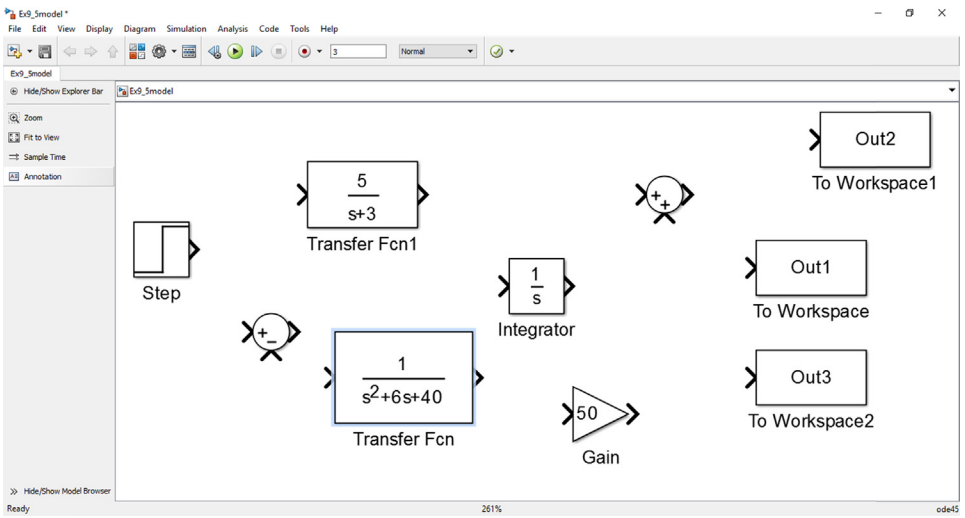


FIGURE 9.22 Elements used to simulate the model in Figure 9.21. The element parameters have already been modified as reflected by the text in each element.

element performs subtraction. The element's icon shows which input is negative. The Integrator block does not need to be changed. The Step element was modified, so the step begins 90 ms after the start of the simulation. This provides us with a better view of the initial response.

Step (4) The elements are then arranged in a configuration with the signals going from left to right (except for the feedback signal) and connected. There are shortcuts to connecting these elements, but the technique of going for input back to output is effective. The second connection does not have to be an element output terminal; it can be just another line which allows for 'T' connections. Right angles are made by momentarily lifting up on the mouse button as with other graphics programs. The three simout elements are connected to the output and upper and lower pathways. The resulting model window is shown in Figure 9.23.

Step (5) We run the simulation setting the Stop time to 5 s. With Simulink, we can monitor any element in the system; in this simulation we use three simout elements to monitor the upper, lower, and output signal paths. The three signals are plotted from the workspace and are shown in Figure 9.24. The upper pathway contributes a smooth exponential-like signal (actually it is a double exponential that continues to increase). The lower pathway contributes a transient component that not only serves to bring the response more quickly toward its final value but also produces the inflection in the combined response.

To determine the pulse response of the system, replace the Step element with the Pulse Generator element from the Sources library. Setting the Period textbox of the Pulse Generator to 10 s will ensure that the pulse occurs only once during the 5-s simulation period. Then setting the Pulse Width (% of Period) to 2% produces a 0.2-s pulse as shown in Figure 9.25 along with other pulse responses.

To estimate the pulse width that approximates an impulse, simulate the system using a range of pulse widths. Shorten the pulse to 0.1 s (i.e., 1%), then to 0.05, 0.25, and finally 0.01 s to produce the responses shown in Figure 9.25. These responses are normalized to the same amplitude to aid comparison. The responses produced by either the 10-ms or 25-ms pulse widths are very similar, so

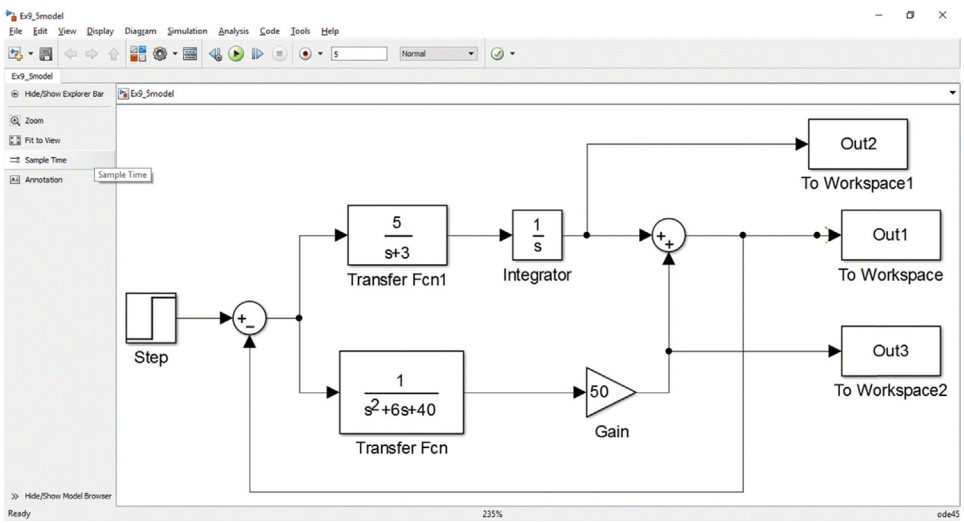


FIGURE 9.23 Assembled model used in Example 9.5 to simulate the system shown in Figure 9.21.

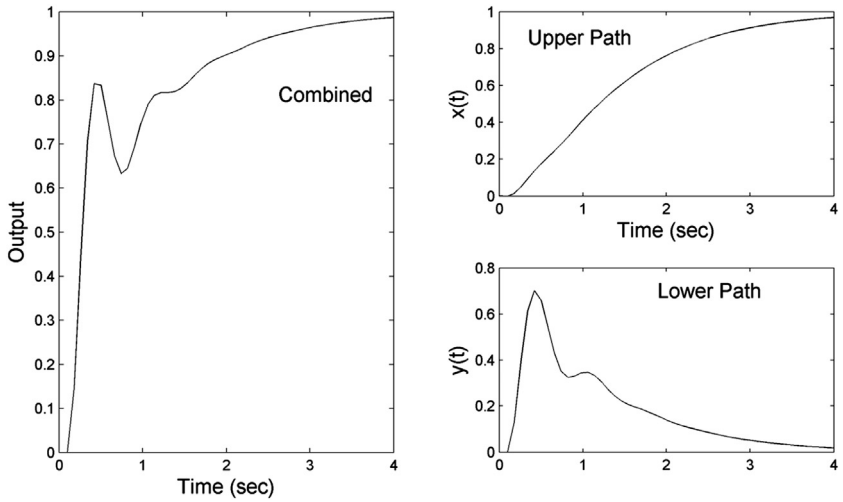


FIGURE 9.24 Three signals from the system shown in Figure 9.23.

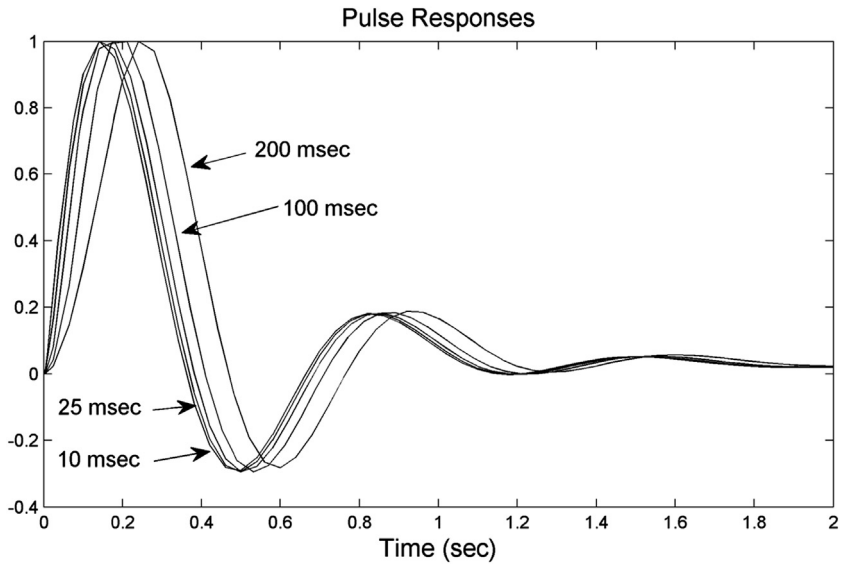


FIGURE 9.25 The response of the system shown in Figure 9.21 to pulses having a range of widths. Responses to pulses having widths of 10 and 25 ms produce nearly the same response indicating that pulses less than 25 ms can be considered impulse inputs with respect to the dynamics of this system.

a pulse having a width of 25 ms or less can be taken as an impulse input with respect to the dynamics of this system. A unit impulse function should have an area of 1.0, so the amplitude of the 0.025-s pulse should be set to $1/0.025 = 40$, whereas the amplitude of the 0.01-s pulse should be 100 to produce the correct response amplitude.

9.4 IMPROVING CONTROL SYSTEM PERFORMANCE: THE PID CONTROLLER

Simulink can be used to show how the performance of real system can be improved. The classic control system consists of a controller subsystem, which directs the actions of an effector subsystem (Figure 1.26). As stated in Chapter 1 (Section 1.4.5.2), the effector subsystem is sometimes referred to as the "plant." Often the controller uses feedback from the plant to modify its control signals to the plant. For example, in the human body, various components of the central nervous system serve as controllers for muscle subsystems. Vision and proprioception provide feedback to the neural controllers. In this section, we use Simulink to study a classic feedback control system and show how to improve its performance using different control strategies. Again we introduce these control strategies through an example.

Our system consists of controller and plant as shown in Figure 1.26. As a model plant, we use a second-order overdamped system, but the controller strategies we investigate are general and work to improve the performance of a wide range of plants. The transfer function of our model plant is

$$TF(s) = \frac{50}{s^2 + 20s + 50} \tag{9.3}$$

In this subsystem, ${\omega_n}^2 = 50$, so $\omega_n = 7.07$, and $2\delta\omega_n = 20$, so $\delta = 1.4$. The plant is an overdamped second-order system. The roots of the denominator are -17.07 and 2.9, so we could factor Equation 9.3 into two first-order elements, but as Simulink can represent elements of any order, why bother.

Our first controller will be just a gain term that operates on the difference between the input signal and the feedback system from the plant (Figure 9.26). We will investigate the response of the controller–plant system to a step input with the goal of improving the speed of response and the response accuracy.

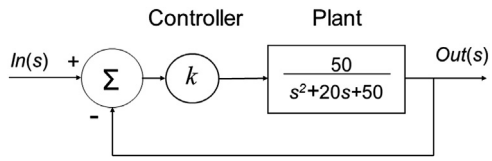


FIGURE 9.26 Classic feedback control system with a second-order, overdamped plant and a constant gain controller. This system is simulated in Example 9.6.

EXAMPLE 9.6

Find the step response of the system in Figure 9.26 to various values of controller gain k. Find the gain that results in the fastest step response without causing overshoot.

Solution

Simulate the step response of this system to a unit step input. Plot the response to several values of gain and find the largest gain that does not produce overshoot. Because we have no idea what the controller gain, k, should be, we start with a value of 1.0 and work our way up (or down if that produces overshoot). The model is laid out with a Step element, Sum element, Gain element, and Transfer Fcn element. The Step element parameter, Step time, is set to 0.1 (the default is 1.0) to clearly show the onset of the movement. In addition, one of the signs on the sum element is made negative, and the Transfer Fcn numerator and denominator coefficients are set to represent the transfer function of Equation 9.3. The output is sent to a To Workspace element, so the results can be plotted. This element's Save format: parameter was set to Array so the output would be a standard MATLAB array. (The default format is a structure.) This results in the model shown in Figure 9.27. The simulation time is reduced to 2.0 s, and the elements are renamed to reflect their role in the model.

Results

Initial simulations used gains of 0.1, 1.0, and 10.0. With a gain of 10, a noticeable overshoot was observed in the response. The gain was reduced to 2.0 and a slight overshoot was still observed. The system's step responses to gains of 0.8, 1.2, and 2.0 are shown in Figure 9.28. A gain of 1 produces a response with very little overshoot, but a gain of 2.0 produces noticeable overshoot. Increasing the controller gain, k, provides two benefits: the step response becomes faster and the final value of the

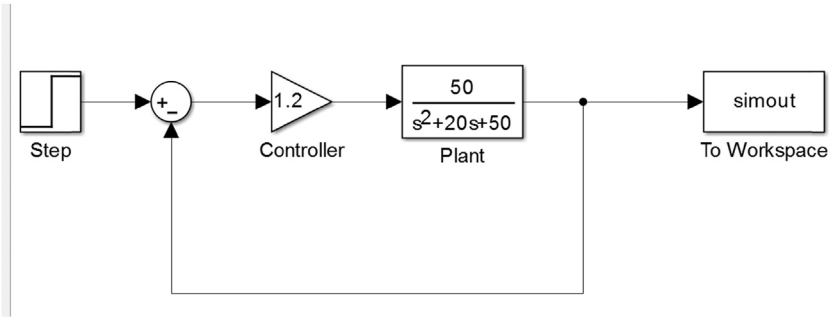


FIGURE 9.27 The Simulink representation of the system shown in Figure 9.26. The elements have been modified to reflect the model parameters. In this model version, the gain, k, was set to 1.2.

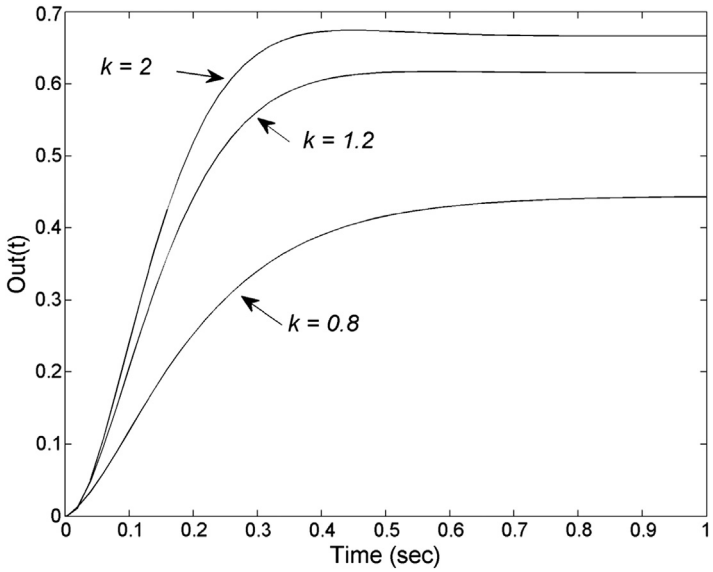


FIGURE 9.28 Simulation of the system in Figure 9.26 to three different values of controller gain, k. As k increases, the response becomes faster and the final value becomes closer to the input final value of 1.0; however, higher values of k also lead to overshoot. A steady-state error between the input and output is inherent in this type of controller as discussed below. More advanced controllers can eliminate this error as shown in the next example.

response comes closer to the step input amplitude of 1.0. However, even at a gain of 2.0 there is still a large error between the output final value of approximately 0.66 and the input final value which is 1.0.

The controller used in the feedback control system shown in Figure 9.26 is known as a "proportional controller" because its output is proportional to the error signal where the constant of proportionality is k. As the plant is driven by error, the error of such a system can never be zero. In fact, we do not need Simulink to tell us there will be an error, or what that error will be. Using the tools of Chapters 6 and 7, we can calculate this final error as a function of k analytically. Applying the feedback equation from Chapter 6 to the system in Figure 9.26 gives the overall transfer function:

$$TF(s) = \frac{Out(s)}{In(s)} = \frac{G(s)}{1 + G(s)H(s)}$$

where $G(s) = (k)\dfrac{50}{s^2 + 20s + 50}$ and $H(s) = 1$. So the overall transfer function is

$$TF(s) = \frac{\dfrac{50k}{s^2 + 20s + 50}}{1 + \dfrac{50k}{s^2 + 20s + 50}} = \frac{50k}{s^2 + 20s + 50 + 50k} \tag{9.4}$$

TABLE 9.2 Output Error as a Function of Controller Gain

Controller Gain, k	Final Output Value	Percent Error
1.2	0.546	45%
1.6	0.615	39%
2.0	0.667	33%

The Laplace domain output to a unit step, $1/s$, is

$$Out(s) = TF(s)\left(\frac{1}{s}\right) = \frac{50k}{s^2 + 20s + 50 + 50k}\left(\frac{1}{s}\right)$$

To find the final value, we turn to the Final Value Theorem (Equation 7.51, Section 7.4.2), which states that $x(t \to \infty) = \lim_{s \to 0} sX(s)$. Thus the final output value as a function of time is

$$Out(t \to \infty) = \lim_{s \to 0}\left[\frac{50ks}{s^2 + 20s + 50 + 50k}\left(\frac{1}{s}\right)\right] = \frac{50k}{50 + 50k} = \frac{k}{1 + k} \tag{9.5}$$

As the final value of the input is 1.0 (the input is a unit step), the final value given by Equation 9.5 shows us the error. This is summarized for three values of k in Table 9.2. The final values appear to be the same as those found in the simulation results shown in Figure 9.28. Our analytical approach (*sans* MATLAB) gives us more information regarding final values than simulation because it shows exactly how error changes with controller gain.

We could also use Equation 9.4 to determine the maximum value of k before response oscillation occurs by finding the value of k that makes the damping factor, δ, equal to 1.0. Recall δ can be determined from the characteristic equation (the denominator equation) of Equation 9.4. As Equation 9.5 shows, as the controller gain, k, increases the error is reduced, but the error can never be zero unless the gain becomes infinite.

The problem with this controller is that its drive is produced solely by the error; without some error there can be no signal to drive to the plant. Perhaps we could reduce this error with a more intelligent controller. For example, if the controller included an integrator, some of the controller's drive signal would come from integration of the error signal. This signal component should continue to increase until the error is finally zero. Such a combined proportional and integral controller is investigated in the next example.

EXAMPLE 9.7

Add an integrator into the feedback controller of the system shown in Figure 9.26. Adjust the gain of the integrator pathway to decrease the final error between input and output values, but still produce a response with no overshoot.

Solution

Modify the model shown in Figures 9.26 and 9.27 to include a second controller pathway containing an integrator and gain term. Such a modification is shown in Figure 9.29. We leave the

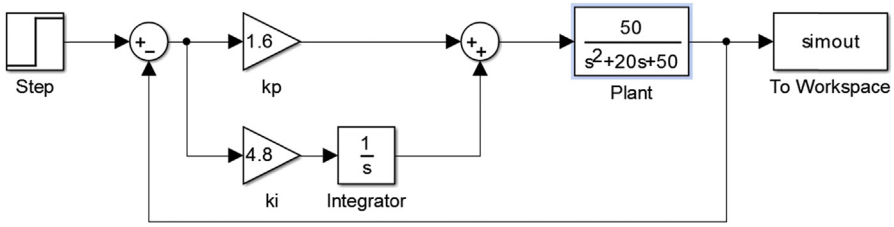


FIGURE 9.29 The system shown in Figure 9.26 with an improved controller that includes an additional pathway containing an integrator. One component of the controller signal should continue to increase (or decrease) as long as the error is nonzero. This system is simulated in Example 9.7.

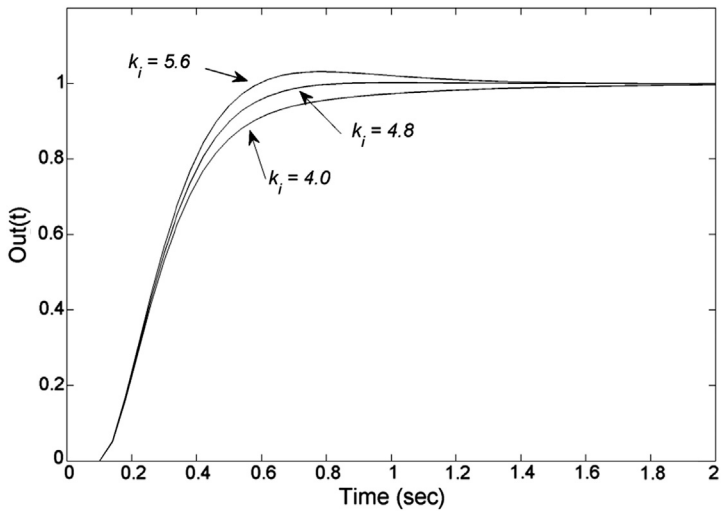


FIGURE 9.30 The response of a system containing both integral and proportional control for three different values of integrator gain, k_i. An integrator gain of 4.8 produces the fastest response without an overshoot. The proportional gain was set at 1.6, which was found to be the largest possible without overshoot in the last example. The timescale of this figure is twice as long as that of Figure 9.28, so these responses are actually a little slower, but they go to the correct final value.

proportional gain set at 1.6, the largest value possible before an overshoot occurs as found in the previous example. We then increase the gain of the integrator pathway until we get response oscillation as in the previous example.

Results

The simulated output of the system with this modified controller is shown in Figure 9.30. Our modified controller was quite successful. The response now goes to 1.0 indicating zero error between the final values of input and output. An integrator gain, k_i, of 4.8 produces a rapid response without overshoot. Note that the time frame of this plot is 2.0 s, twice as long as that of Figure 9.28, so these responses are actually a little slower, but they do go to the correct final value.

Modifying the controller to include an integrator was very successful in improving performance. But why stop there? We have eliminated the final error in the response, but maybe we could make it still faster without inducing overshoot. To boost the response speed, we might try adding a derivative component to the controller signal. This would produce an initial signal in response to the step input, but this signal would fade away as the response progressed and would be zero in the late steady-state section of the response. We will try this added component in the next example.

EXAMPLE 9.8

Modify the controller of the system simulated in the last example to include a derivative component. Increase the gain of this signal component to produce the fastest signal possible without inducing response overshoot.

Solution

Add a third pathway to the controller that includes a derivative operator and a gain term. Although Simulink has a derivative element, it does not work well in simulations with step changes, as the element produces an extremely large, spikelike output. It is better to use a realistic derivative operator consisting of a derivative and a first-order element that acts as a low-pass filter.

$$TF(s) = \frac{\omega_1 s}{s + \omega_1} \tag{9.6}$$

The low-pass element functions to limit the derivative output to more reasonable levels. Equation 9.6 is also the transfer function of a realistic derivative operator, one you would get if you used analog hardware to build a derivative operator. It is only necessary that the low-pass cutoff frequency, ω_1, be much greater than that of the plant, so the derivative controller is still faster than the plant. From the plant's transfer function, Equation 9.3, the undamped natural frequency, $\omega_n = \sqrt{50} = 7.07$ rad/s, so we should make the derivative controller much faster, say, with a cutoff frequency, ω_1, of 50 rad/s. This results in the model shown in Figure 9.31.

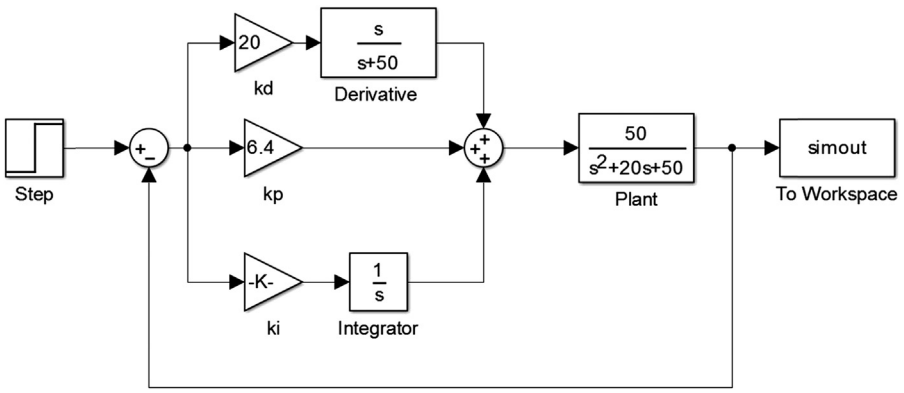


FIGURE 9.31 The system of Figure 9.26 with a more sophisticated controller containing proportional, integral, and derivative components. This system is simulated in Example 9.8.

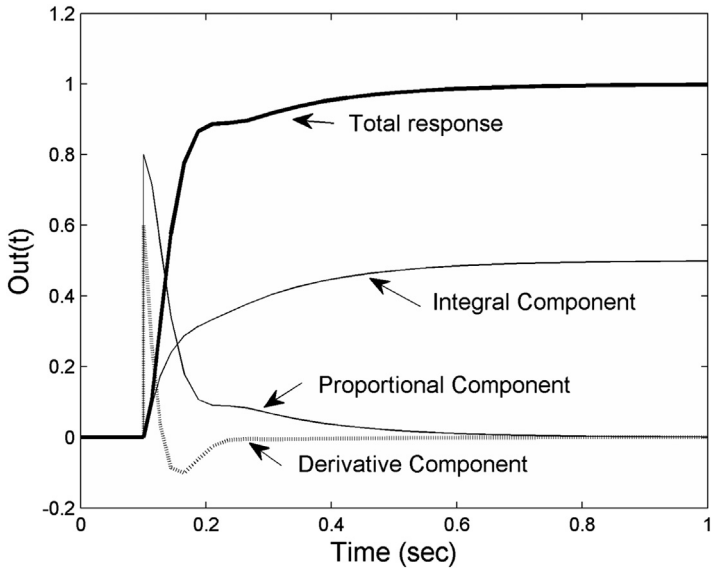


FIGURE 9.32 The response of a feedback control system having a plant defined by Equation 9.3 and a three-component controller. The three component signals are shown along with the total response. The component gains that produced the fastest response with no overshoot and no steady-state error are $k_p = 6.4$; $k_i = 14.4$; and $k_d = 20$.

Adjusting the gains for an optimal response is much more difficult when all three control components are present. Various strategies exist, including commercial software packages that guide parameter adjustment. The approach here is to increase the gains of both proportional and integral elements by a factor of 2 or 3, and then increase the derivative gain until the overshoot is eliminated. This requires multiple simulations and the simultaneous adjustment of all three parameters. A similar empirical approach is often used when actual hardware is involved.

Results

The approximately optimal step response of this system with a three-component controller is shown in Figure 9.32. The proportional, integral, and derivative component gains that produce this response are $k_p = 6.4$; $k_i = 14.4$; and $k_d = 20$. The three signal components that combine to drive the plant are also shown scaled to have about the same amplitude.

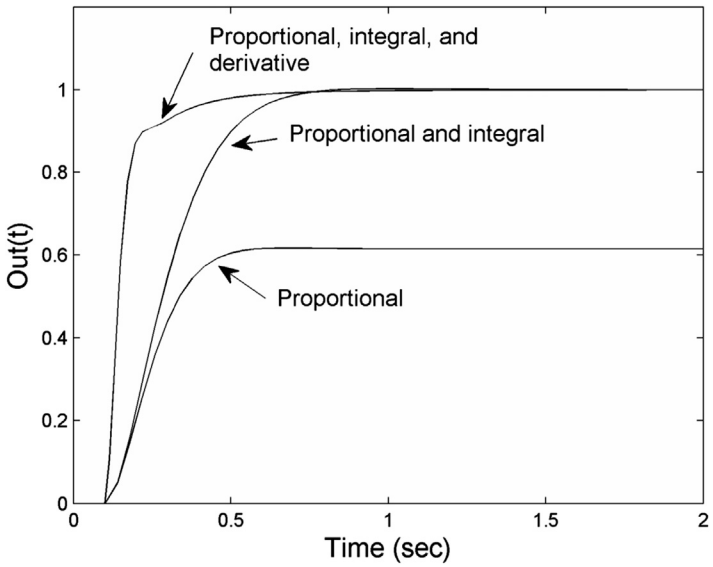


FIGURE 9.33 A comparison of the response of the same plant to three different controllers. The improvement in performance in terms of speed and accuracy of controllers that include proportional, integral, and derivative signals is obvious. Such controllers are called PID controllers, and a range of these controllers is offered commercially

Figure 9.33 compares the responses of the same plant (Equation 9.3) when driven by increasingly sophisticated controllers. The improvement in output error achieved by adding an integral component is striking as is the improvement in speed due to the derivative component. In many practical situations the dynamics of the plant are difficult to modify, but the controller (which is usually a computer) is easy to extend and to change.[9] Because of the obvious performance enhancement, three-component controllers are very common in feedback control systems. Such controllers are called "PID" controllers. They are found in many systems where the system output must track a desired level or simply maintain a desired level. A variety of commercial hardware PID controllers are available that will drive plants ranging from small motors to very large machines.

9.5 BIOLOGICAL EXAMPLES

Many biomedical models use Simulink, and some are quite detailed involving more than 50 elements. For a comprehensive Simulink model of the cardiovascular system, check out the model by Zhe Hu, downloadable from MATLAB Central at http://www.mathworks.com/matlabcentral/fileexchange/818-hsp. This model, with over 90 elements, is far too large to be used here.

[9]Think of the cruise control in an automobile. The plant dynamics (the way the car responds to increased throttle) depend on the weight of the car, the size of the engine, and unknown variables such as road conditions. In such cases, a control system that can respond quickly and accurately to keep speed at the requested level is essential.

Example 9.9 presents a realistic physiological model on a much smaller scale: a model of glucose and insulin balance in extracellular tissue. This model demonstrates the ability of Simulink to handle nonlinear elements.

9.5.1 Stolwijk–Hardy Model of Glucose–Insulin Concentrations

EXAMPLE 9.9

Simulate a simplified version of the Stolwijk–Hardy model for glucose–insulin concentrations in extracellular space.

Solution

The Stolwijk–Hardy model consists of two integrators involved in a complex feedback system. The concentration of glucose or insulin in the extracellular fluid is determined by integration of the net flow into each fluid compartment divided by the effective volumes of the fluid compartment. The Simulink version of the Stolwijk–Hardy model is shown in Figure 9.34. Concentrations of glucose and insulin are represented by the outputs of integrators labeled "Glucose" and "Insulin."

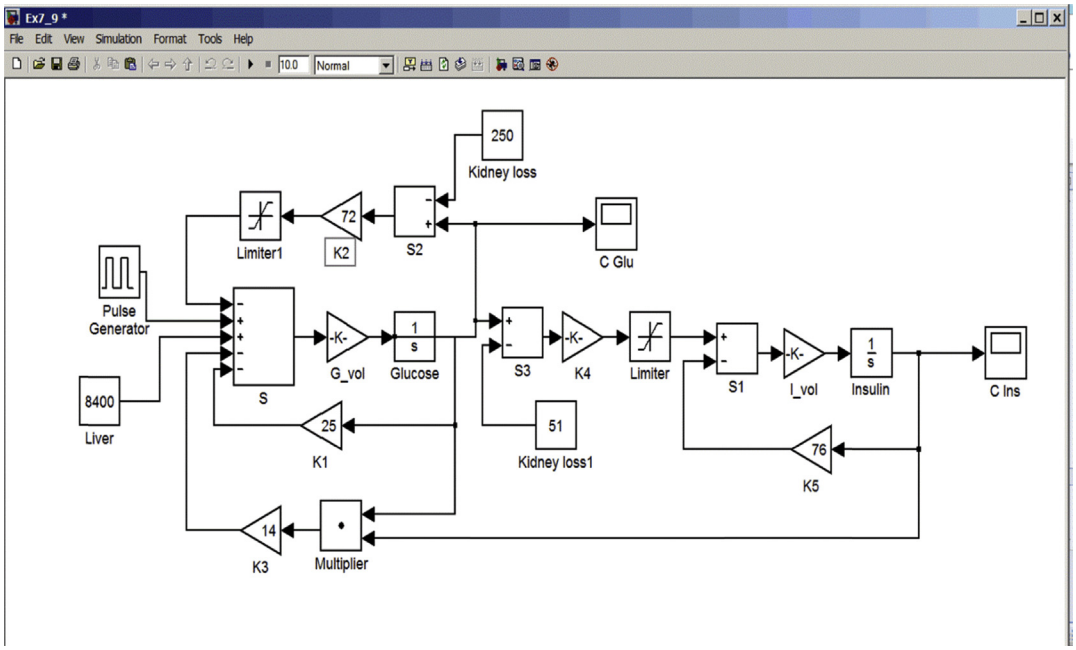


FIGURE 9.34 Simulink representation of the Stolwijk–Hardy model, which represents the concentration of glucose and insulin in the extracellular space.

The insulin integrator receives a negative feedback signal from its output modified by gain K5 and also receives a contribution from the glucose concentration signal. The glucose signal is modified by gain term, K4, and loss through the kidney is represented by a constant negative signal from Kidney loss1. This glucose signal also has an upper limit or saturation imposed by element Limiter.

The glucose integrator receives two feedback signals from its output: one direct through gain K1 and the other modified by Kidney loss and gain K2. This kidney feedback signal saturates: it has a maximum output, imposed by Limiter1. The glucose integrator also receives feedback from the insulin signal modified by its own output signal through the multiplier and gain K4. The glucose integrator receives two input signals: a constant input from the liver and one representing glucose intake. The latter is implemented by a pulse generator set to produce a single 0.5-h pulse after a 1-h delay. (Owing to the slow response of this system, the assumed time unit is in hours for this simulation.) Both integrators are preceded by gain terms that reduce the signal in proportion to the effective volumes of the extracellular fluid for glucose and insulin. This converts the glucose and insulin signals to concentrations. Both integrators also have initial values incorporated into the integrator. Alternatively, a longer delay between the start of the simulation and the onset of the pulse can be used to allow the integrators to obtain appropriate initial values. A problem at the end of this chapter demonstrates this approach.

The limiters and multiplier elements make the equations for this system nonlinear so that they cannot be solved using the linear techniques presented in Chapters 6 and 7. However, Simulink has no difficulty incorporating many different nonlinear elements in a model.

The various model parameters, from a description of the Stolwijk–Hardy model in Rideout (1991), are shown in Table 9.3.

TABLE 9.3 Parameter Values for the Stolwijk–Hardy Model of Glucose and Insulin Concentrations

Parameter	Value	Description	Parameter	Value	Description
C_{G_init}	81 (%mg/mg)	Glucose integrator initial value	G_vol	1/150 (% mL)	Inverse effective volume of glucose extracellular
C_{I_init}	5.65 (%mg/mg)	Insulin isntegrator initial value	I_vol	1/150 (% mL)	Inverse effective volume of insulin extracellular
Liver	8400 (mg/h)	Glucose inflow from liver	Limiter	10000	Glucose saturation driving insulin
K1	25	Gain of glucose feedback	Limiter1	10000	Glucose saturation in feedback signal
K2	72	Gain of glucose feedback from kidney loss	Kidney loss	200 (mg/h)	Renal spill in feedback signal
K3	14	Gain of insulin feedback to glucose	Kidney loss1	51 (mg/h)	Kidney loss in glucose feedback
K4	14	Gain of glucose drive to insulin	K5	76	Gain of insulin feedback

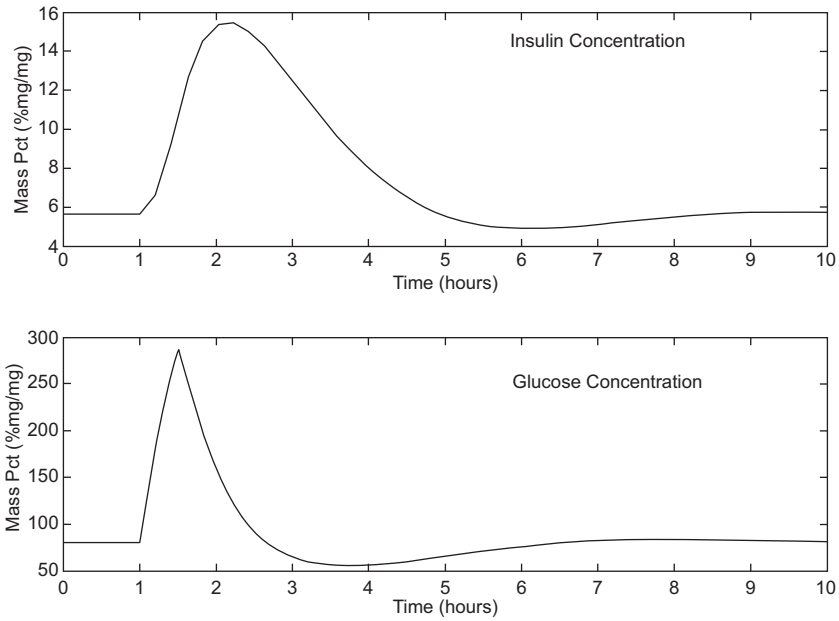


FIGURE 9.35 Simulated responses of glucose and insulin concentrations in the extracellular fluid after infusion of glucose. The 10-h simulation shows that, following a 1/2-h pulse infusion, glucose rises sharply, then falls off undershooting the baseline level. Insulin also increases, but more slowly, and undershoots only slightly before returning to baseline levels.

Results

The results of a 10-h simulation of this model are shown in Figure 9.35. The influx of glucose, modeled as a half-hour pulse of glucose, results in a sharp rise in glucose concentration and a slightly slower rise of insulin. When the glucose input stops, glucose concentration immediately begins to fall and undershoots before returning to the baseline level several hours later. Insulin concentration falls much more slowly and shows only a slight undershoot but does not return to baseline levels until about 7 h after the stimulus.

9.5.2 Model of the Neuromuscular Motor Reflex

Example 9.10 describes a model of the neuromuscular motor reflex. In this model, introduced by Khoo (2000), the mechanical properties of the limb and its muscles are modeled as a second-order system. The model addresses an experiment in which a sudden additional weight is used to exert torque on the forearm joint. The muscle spindles detect the stretching of arm muscles and generate a counterforce though the spinal reflex arc. The degree to which this counterforce compensates for the added load on the muscle can be determined by

measuring the change in the angle of the elbow joint. The model consists of a representation of the forearm muscle mechanism in conjunction with the spindle feedback system. The input to the model is a mechanical torque applied by a steplike increase in the load on the arm; the output is forearm position measured at the elbow in degrees.

EXAMPLE 9.10

Model the neuromuscular reflex of the forearm to a suddenly applied load.

Solution

To generate a model of the forearm reflex, we need a representation of the mechanics of the forearm at the elbow joint and a representation of the stretch reflex. The transfer function of mechanical models is covered in Chapter 13, where the function for the forearm joint is derived from basic mechanical elements. For now, we accept that the transfer function of the forearm muscle mechanism is given by a second-order equation. Using typical limb parameters, the Laplace domain transfer function is

$$TF(s) = \frac{V(s)}{M(s)} = \frac{250}{s2 + 25s + 500} \tag{9.7}$$

where $V(s)$ is the velocity of the change in angle of the elbow joint and $M(s)$ is the applied torque. As position is the integral of velocity, Equation 9.7 should be integrated to find position. In the Laplace domain, integration is just divided by s:

$$TF(s) = \frac{\theta(s)}{M(s)} = \frac{1}{s}\frac{V(s)}{M(s)} = \frac{250}{s(s2 + 25s + 500)} \tag{9.8}$$

The transfer function of Equation 9.8 represents the effector element in this system, i.e., the plant. The muscle spindles make up the controller of this reflex but are located in the feedback pathway. Khoo (2000) shows that the muscle spindle system can be approximated by the transfer function:

$$TF(s) = \frac{M(s)}{\theta(s)} = \frac{s + 60}{s + 300} \tag{9.9}$$

where $M(s)$ is the effective counter torque produced by the spindle reflex and $\theta(s)$ is the angle of the elbow joint. In addition, there is a Gain term in the spindle reflex nominally set to 50 and a time delay (i.e., e^{-sT}) of 20 ms. The spindles provide feedback, so these elements are placed in the feedback pathway. The overall model is shown in Figure 9.36.

Results

Simulation of this model in response to added load of 5 kg applied as step input is shown in Figure 9.37. Simulations are performed for two values of feedback gain, 50 and 100. At the higher feedback gain, the reflex response shows some oscillation, but the net displacement produced by the added weight is reduced. If the goal of the stretch reflex is to compensate for the change in load on a limb, the increase of feedback gain improves the performance of the reflex, as it reduces the change

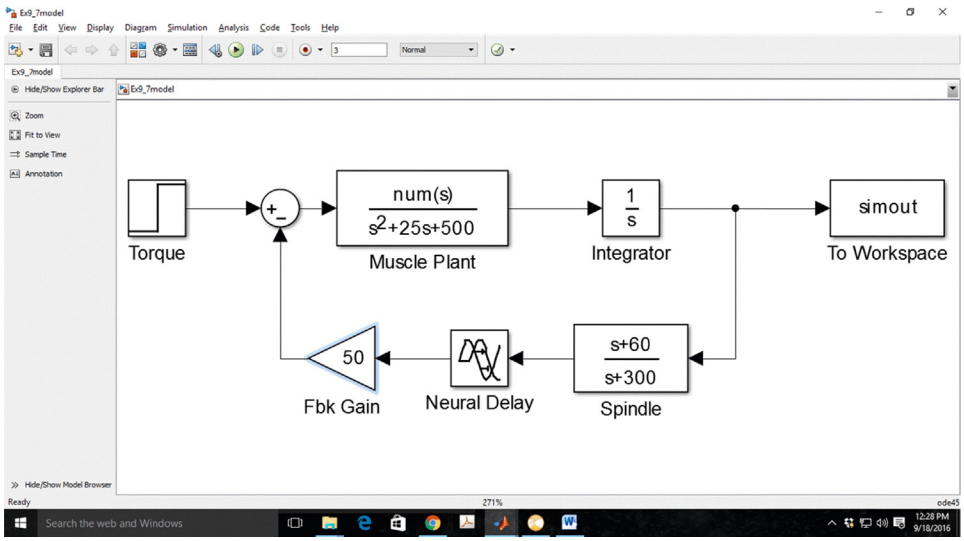


FIGURE 9.36 Model of the neuromuscular reflex developed by Khoo (2000).

produced by the added load. However, feedback systems with high loop gains tend to oscillate, a behavior that is exacerbated when there is a time delay in the loop. The effects of gain and delay on this oscillatory behavior are examined in a couple of problems based on this model.

Example 9.11, an extension of Example 9.10, finds the magnitude and phase spectra of the neuromuscular model used in that example. The approach is the same as in Example 9.4.

EXAMPLE 9.11

Find the magnitude and phase spectrum of the neuromuscular reflex model used in Example 9.10. Use a feedback gain of 90.

Solution

As in Example 9.4, we simulate the impulse response, then take the Fourier transform, and plot the magnitude and phase spectra. Again, to generate the impulse response, we shorten the width of a pulse until the responses generated by a pair of short pulses are the same. To find this pulse width, we replace the step generator in Figure 9.36 with a pulse generator. Simulations to pulses having widths less than 2 ms produce the same response dynamics, indicating that a 2-ms pulse width can be taken as an impulse. The response to a 2-ms pulse with an amplitude of 5 kg is shown in Figure 9.38. For all practical purposes, this is the impulse response.

The Model Configuration Parameters window was modified, so the simulator used a fixed step of 1.0 ms as in Example 9.4 ($f_s = 1$kHz). Because a 2-ms pulse acts as an impulse to this system, any sample interval $\leq$2 ms would work. After running the simulation, the same code developed in Example 9.4 was used to calculate and plot the magnitude and phase spectra

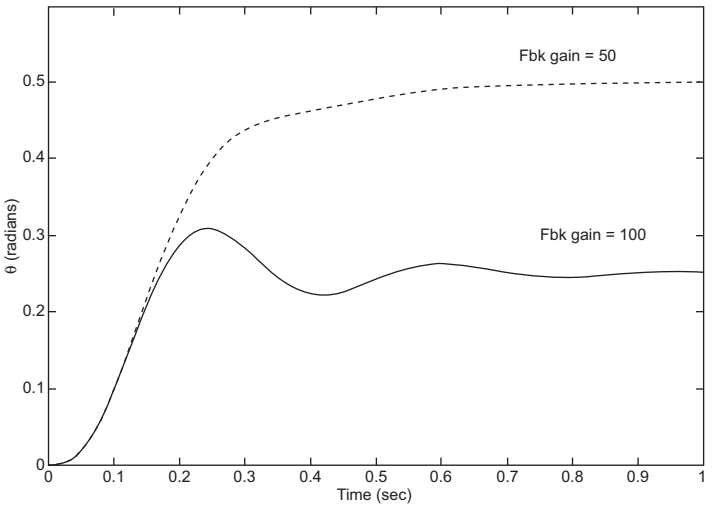


FIGURE 9.37 Model responses of the neuromuscular reflex to a steplike increase on load. Simulated responses are shown for two values of feedback gain.

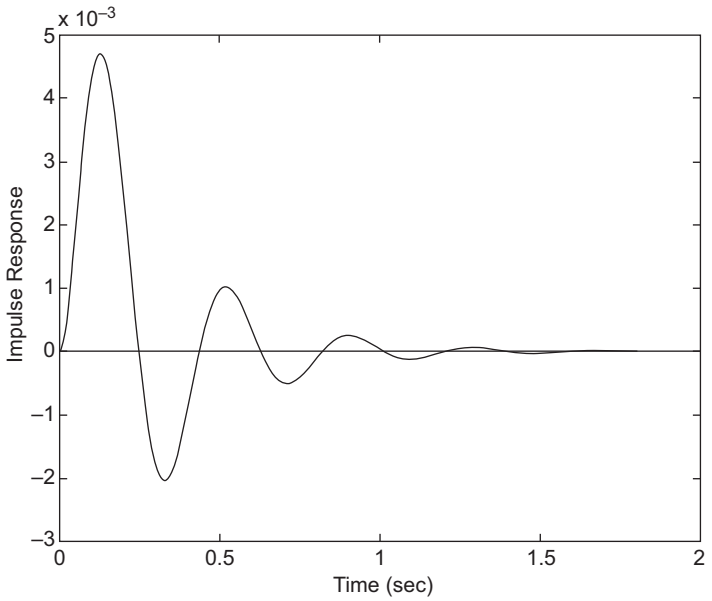


FIGURE 9.38 The response of the neuromuscular system shown in Figure 9.28 to a 2-ms pulse. Reducing pulse width does not change the shape of the response indicating that the 2-ms pulse is acting as an impulse for this system.

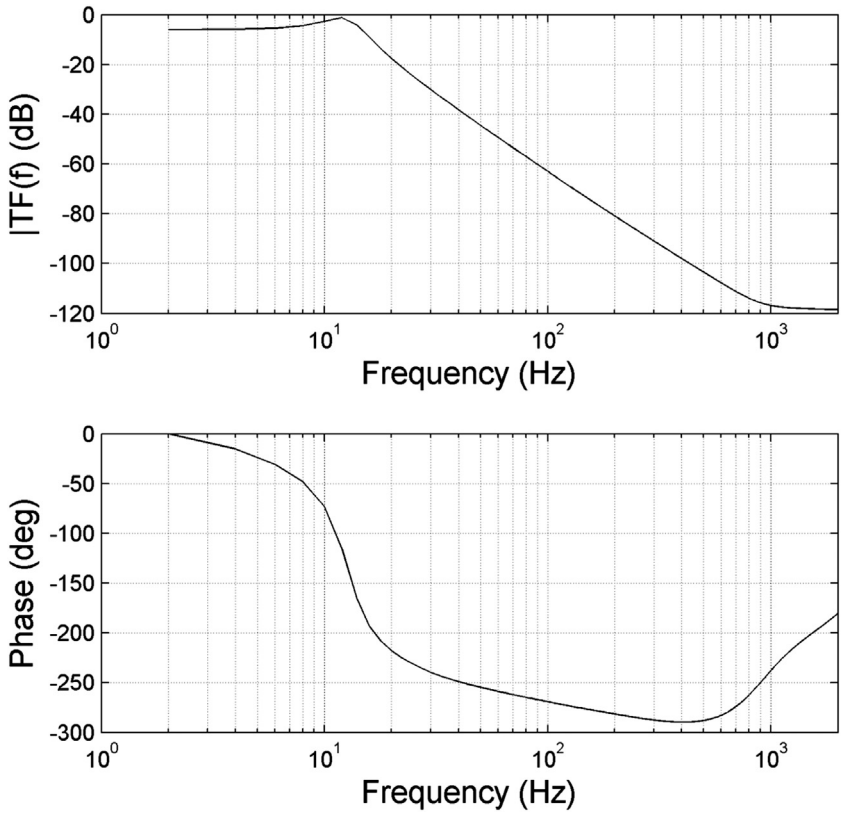


FIGURE 9.39 The spectral characteristics of the neuromuscular reflex shown in Figure 9.38. These spectra were determined by taking the Fourier transform of the impulse response.

Results

These two frequency components are plotted against log frequency in Hz in Figure 9.39. The magnitude curve shows a low pass characteristic with a cutoff frequency of approximately 15 Hz and an attenuation of 60 dB/decade as expected from a third-order system. Determining the spectral characteristics of a system using simulation is explored in a couple of the problems.

9.5.3 The Makay and Glass Model of Neutrophil Density

The last example features a highly nonlinear model and shows how to construct a model, given a differential equation for the system. This model represents white blood cell counts, or neutrophils, in patients with chronic myeloid leukemia. It was originally developed as a differential equation by Mackay and Glass in 1977 and is described in Khoo (2000). The equation defining neutrophil density, x, is given as

$$\frac{dx}{dt} = (\beta\theta^n)\left(\frac{x(t-T_D)}{\theta^n + x(t-T_D)^n}\right) - \gamma x(t) \tag{9.10}$$

where t is time in days (the time frame of this system is quite long), γ is the neutrophil extinction rate, θ and n describe the relationship between the neutrophil production rate and the past neutrophil density, T_D is the delay in neutrophil production, and β is a scale factor. Values of γ, θ, and n suggested by Khoo are 0.1, 1.0, and 10.0, and the value for β is around 0.3. The delay T_D varies between 2 and 20 days. In addition, we need an initial value for neutrophil density $x(0)$ and use 0.1. Equation 9.10 is highly nonlinear and would be difficult to solve analytically, but simulating the solution in MATLAB is straightforward.

EXAMPLE 9.12

Generate a Simulink model based on Equation 9.10 and simulate the response for neutrophil maturation delays of 2, 8, and 20 days.

Solution

The major hurdle in this example is generating the model from the equation. In general, it is easiest to start with the derivative dx/dt (or the highest derivative) and use an integrator to generate x. Then we work from x to produce the right side of the equation.

The $-\gamma x$ term is generated by applying a gain term gamma to x, then feeding this back negatively to the derivative input (Figure 9.40). Sending signal x through a transport delay produces the

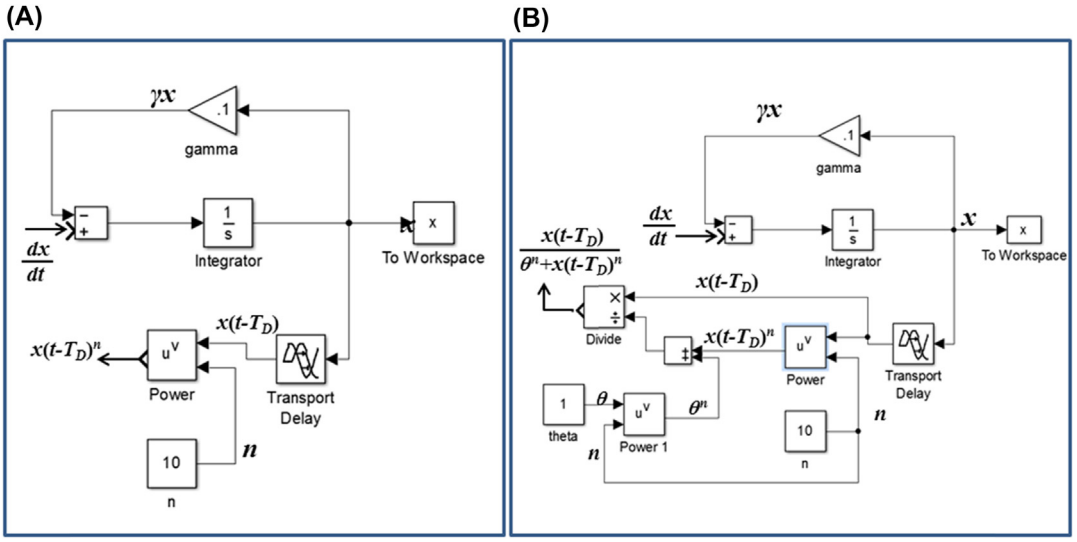


FIGURE 9.40 A) Partial model representation of differential equation (Equation 9.10). The second term in that equation, γx, is produced by applying a gain term to signal x. This is fedback negatively to be one component of the derivative signal. Two component signals of the first term, $x(t - T_D)$ and $x(t - T_D)^n$, are also generated. B) The constant θ^n is generated and with other signals is used to construct a signal representing the first term of Equation 9.10, missing only the two multiplier terms, θ^n and β.

signal $x(t - T_D)$ (Figure 9.40A). To raise this signal to power n, use the Math Function block found in the Math Operations library. This block supports a wide variety of mathematical operation, including exponentiation, natural and common logs, squaring, and raising to a power. We use the Function Block Parameter window to set Function to pow. This raises the signal in the upper input to the power of the signal (or constant, in this case) in the lower input. The output of this block, labeled Power in Figure 9.40A, is $x(t - T_D)^n$.

The Math Function block set to pow is also used to generate the θ^n term as seen in Figure 9.40B. Adding this constant term to signal $x(t - T_D)^n$ gives $\theta^n + x(t - T_D)^n$, the denominator of the fraction in Equation 9.10. As we already have signal $x(t - T_D)$, we use a divider block, also from the Math Operations library to produce a signal equivalent to $\left(\frac{x(t-T_D)}{\theta^n+x(t-T_D)^n}\right)$ as shown in Figure 9.40B. To finish the first component, all we need to do is to multiply this signal by θ^n, then scale by β. The resultant signal is added to the $-\gamma x$ term to complete the derivative, $\frac{dx}{dt}$. The final model is shown in Figure 9.41.

In essence, the system shown in Figure 9.41 can be thought of as an integrator that received two signals from two feedback pathways. One is a negative feedback pathway with linear gain of γ. The other is a positive feedback pathway that is highly nonlinear. The default Simulink solver showed slight instabilities, so an alternative solver was selected from the Model Configuration Parameters window. Under the Solver: tab, ode15 was selected on a trial and error basis simply because it produced stable simulations with short run times.

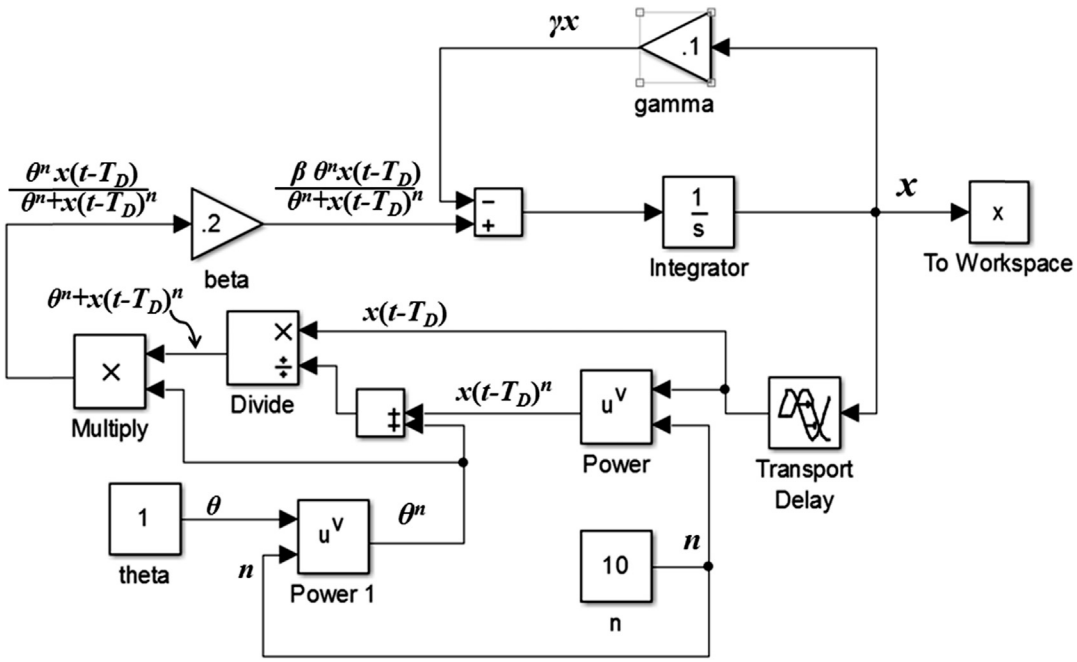


FIGURE 9.41 The completed model used in simulate Equation 9.10. This model is highly nonlinear. Adapted from Khoo (2000).

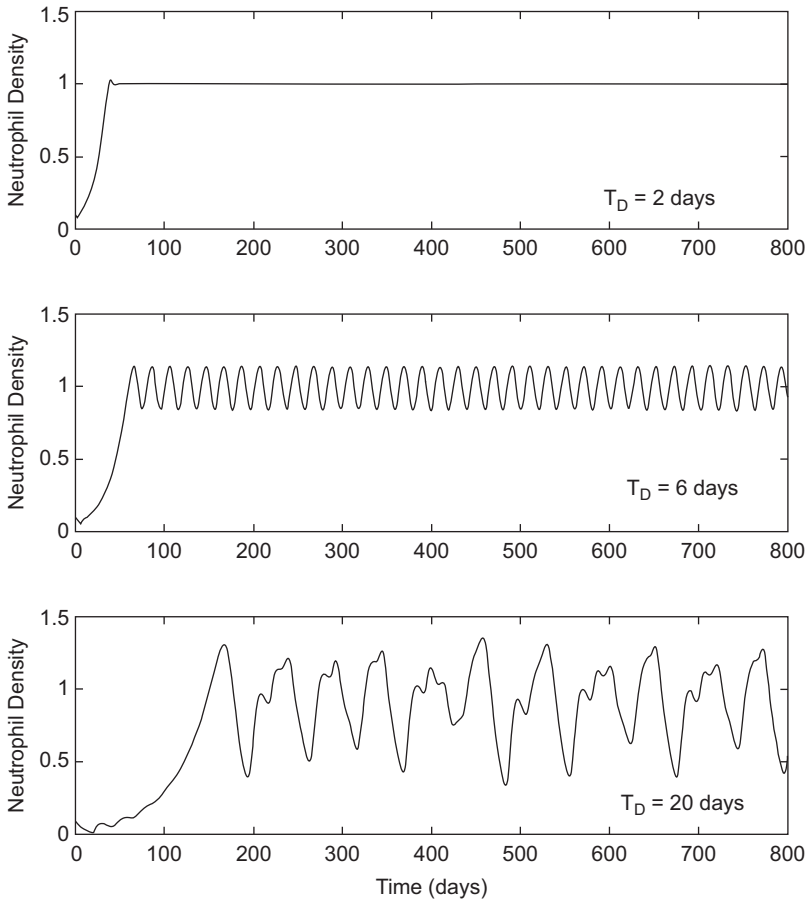


FIGURE 9.42 The modification in neutrophil density found for three maturation times: 2, 6, and 20 days. The 6-day delay produces a sustained oscillation of approximately 20 days, whereas the 20-day delay produces a response that oscillates randomly. The 20-day delay signal shows chaotic behavior: an apparent randomness in what is a completely deterministic system.

Results

Simulations were performed using the parameter specified above for three values of *TD*: 2, 8, and 20 days, and the results are shown in Figure 4.42.

Recall that the transport delay T_D represents the maturation time, the time required to produce mature neutrophils in the bone marrow. The two day period is normal and the simulation shows an initial rise to a constant neutrophil density that is sustained. The 6-day delay leads to an oscillatory response and is known as "cyclical neutropenia." When maturation is extended to 20 days, cyclical behavior now has a random component, a pattern sometimes found in "chronic myeloid leukemia." This random component is surprising, as the system is completely deterministic; there is no added noise in the system. This is an example of a chaotic system, a deterministic system governed by

fixed, well-defined equations, which shows apparently random behavior. It occurs in some nonlinear systems with positive feedback. (Linear systems with positive feedback may show sustained oscillations, but not chaotic behavior.)

One distinguishing feature of a chaotic system is high sensitivity to changes in initial conditions. Thus we might expect that if the initial condition, which is 0.1 for all the responses shown in Figure 9.42, were changed slightly, the response might be quite different. This assessment is found in one of the problems.

9.6 SUMMARY

Simulation is a powerful time-domain tool for exploring and designing complex systems. Simulation operates by automatically time-slicing the input signal and determining the response of each element in turn until the final output element is reached. This analysis is repeated time slice by time slice until a set time limit is reached. As the response of each element is analyzed for each time slice, it is possible to examine the behavior of each element, as it processes its local input as well as overall system behavior. Simulation also allows for incorporation of nonlinear elements such as the divider and exponential elements found in the model used in Example 9.12. MATLAB's Simulink is a powerful simulation tool that provides a wide range of elements and is easy to use.

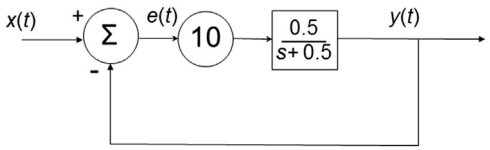
In classic feedback control models, the overall system is divided into two subsystems: an effector subsystem, often called the plant, which implements the system's output, and a controller that combines information from the input and feedback from the output to produce signals that control the plant. For instance, in a home heating system the plant contains the furnaces, radiators, and home thermal properties, whereas the controller is the thermostat. Often the dynamic characteristics of the plant are difficult to modify, or may change over time, or even be unknown, whereas those of the controller, which is usually computer based, can be easily changed. Improvements in the computer algorithm can often lead to significant improvements in performance of the overall system. One such control subsystem is the PID controller, which uses the error between the actual and desired response, along with the derivative and integral of that error, to drive the plant. Adjusting the three signal gains can be tricky but can significantly enhance system performance. Some advanced controllers are adaptive and can change their parameters, for example, the gains of PID control, in response to changes in the plant.

Simulation techniques are well-suited to analyzing biological systems, as they can represent the nonlinear properties found in these systems. Examples of glucose–insulin balance and the stretch reflex are given, but many more can be found on the Web.

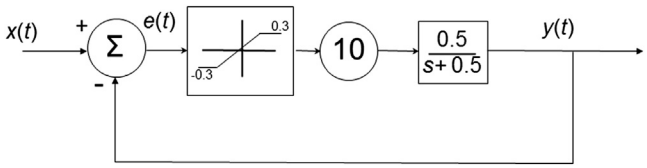
PROBLEMS

1. Write your own program following the approach used in Example 9.1 to simulate the system shown below to a unit step input. Note the first-order element can be simulated using an integrator in a feedback loop. (Using the feedback equation, Equation 7.6,

putting $\frac{0.5}{s}$ in a feedback loop gives $TF(s) = \frac{G}{1+GH} = \frac{0.5/s}{1+0.5/s} = \frac{0.5}{s+0.5}$. Thus you need to add another, external, feedback loop to the one in Example 9.1.)

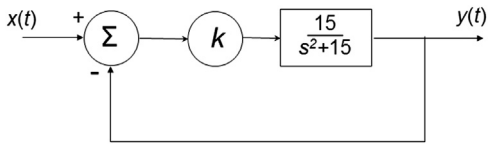


2. It is easy to add nonlinearity to a simulation, even when coding it yourself as in Problem 1. Write a program to simulate the response of the system below to a unit step. The system is the same as the one in Problem 1 except for saturation nonlinearity in the controller. The saturation nonlinearity restricts the output signal to fall within some maximum and minimum limits, in this case ± 0.3. (Hint. If you use MATLAB's min and max routines you only need one additional line of code.)



3. Simulate the system in Problem 2 using Simulink. Simulate the response to a unit step and to a 10 rad/s sine wave with an amplitude of 1.0. For the latter, increase the simulation time from 2.0 to 5.0 s. (To submit the MATLAB plot, you can use the To Workspace block in the Sinks library. Remember to set the Save format option to array and select a variable name or just use the default "simout".)
4. Use Simulink to find the step response of the system shown in Problem 7 of Chapter 7. Use values of 0.5 and 1 for k. Simulate a 10-s response. Use To Workspace to get the data in the workspace and plot the two responses on a single graph.
5. Use the approach shown in Example 9.3 to find the maximum pulse width that can still be considered an impulse. Use Simulink to generate the pulse response of the system used in Problem 4 with the value of $k = 0.5$. To determine when responses are essentially the same dynamically, use the scope to measure carefully some dynamic characteristic of the response such as the time of the first peak. To get an accurate estimate of the time when the first peak occurs, use the magnifying feature on the scope. When two different pulse widths produce responses with the same peak time, the input pulse can be taken effectively as an impulse. Use To Workspace to plot this impulse response.

6. Simulate 10-s of the response of the system below to a 0.1 s (i.e., 1%) pulse. Make $k = 1$ and 10 and plot the responses on the same graph. Can you explain the response?

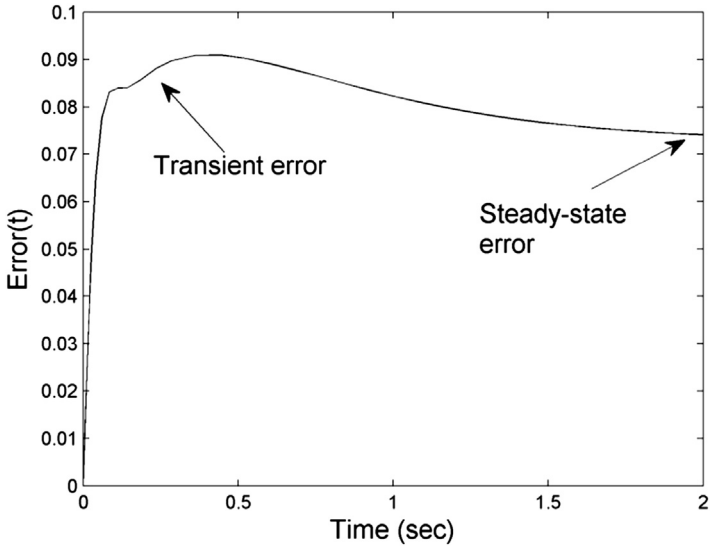


7. Find the magnitude spectrum of a system using white noise. Use Simulink to simulate the fourth-order transfer function of Equation 9.2 used in Example 9.4. Replace the Pulse Generator with a Band-Limited White Nosie generator also found in the Sources library. Theoretically, white noise contains energy at all frequencies, so the Fourier transform of the white noise response of a system can be used to find the magnitude spectrum (but not the phase spectrum because the phase of white noise is undefined). As in Example 9.4, open the Model Configuration Parameters window (under the pull-down Simulation tab) to set the Solver options/Type to Fixed-step and set the Fixed-step size to 0.001 s (1.0 ms). As we are dealing with noise input, it is better to have more data, so increase the simulation time to 200 s. Plot the magnitude in dB against log frequency. The resulting magnitude spectrum will be noisy, particularly at high frequencies, but should have the same shape as the magnitude curve in Figure 9.20. A filter can be applied to the spectrum to smooth the curve.
8. Develop a PID control to improve the step response of a feedback control system having a plant defined by the transfer function. Use a 2.0-s simulation time.

$$TF(s) = \frac{4}{s^2 + 8s + 8}$$

First set the proportional gain so as to get no overshoot in the response. Plot this response. As in Example 9.6, use the feedback equation in conjunction with the final value of the plant to calculate the steady-state error. Compare this error with the final value from the simulation. Next add integral control to eliminate the steady-state error. Plot this response. Finally, add derivative control and try to get the fastest response. You will find you can set the derivative gain very high and increase the other gains by four or five times to get a very rapid response. Plot your best response and include the PID parameter values in the title.

9. Use the optimized model from Problem 8 and replace the step input with a Ramp element. Set the Slope parameter of this input element to 2.0 (units/s). Although there was no steady-state error in this system's step response, there will be a steady-state error to the ramp input signal. This error can best be observed by plotting the error signal that feeds the three PID controller elements as shown in the figure.



Although all three of the PID controller signals affect the transient error, only one has an effect on the steady-state error (see figure). Which PID signal influences steady-state error? Scale up this parameter until the transient error just begins to show undershoot (in addition to the initial overshoot.) Record the steady-state error at this parameter value and at the values used in Problem 8. Plot the final error signal as in the figure and include the original and final steady-state errors in the title.

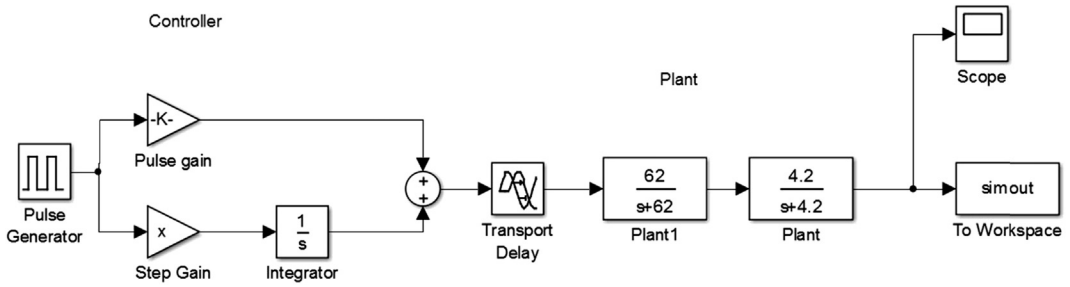
10. Can you do better than the human central nervous system? The variable verg_movement in file vergence_resp.mat is the response of the eyes to an inward moving target. The response, called a vergence eye movement, shows the angle between the two lines-of-sight in deg. The response covers the first 2 s after a step change in target distance and is sampled at 200 Hz. Your task, given the transfer function of the eye movement plant below, is to develop a PID controller that does as well. Use a fixed-step simulation with a Fixed-step size... of 0.005 s, so you can plot the actual and simulated response superimposed.

Several highly accurate models of the neuroocular plant have been developed based on the mechanics of the eye and its muscles. These sophisticated models feature fourth-order dynamics and nonlinear elements, but a reasonable second-order approximation is given by the transfer function:

$$TF(s) = \left(\frac{62}{s+62}\right)\left(\frac{4.2}{s+4.2}\right)$$

You should be able to generate a simulated response that comes quite close to the actual response. Plot the simulation to a 4.0 (deg) step input and actual response superimposed with the correct timescale. Note the vertical axis will be in deg. Provide the PID parameter values in the title of the plot. (You may want to store the simulation output for use in the next problem.)

11. You can actually make the response much faster without oscillation, provided you are willing to give up feedback. This means that there is some error in the final position; nonetheless the nervous system actually adapts this "open-loop" strategy to produce an ultrafast movement called "saccadic eye movements." (Adaptive processes are used by the nervous system to reduce systematic movement errors over the long term.) To generate this fast movement (the fastest in the human body), the nervous system generates a pulse that moves the eyes quickly to the final position. A step signal is also generated to sustain the eyes at the desired final position. There is neurophysiological evidence that the step is generated by neural integration of the pulse. This leads to the control model shown below.

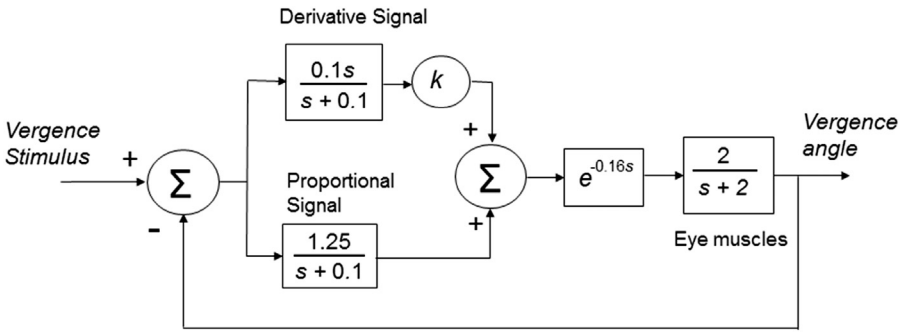


Adjust the step gain to produce a 4-deg response and then adjust the pulse gain to give the fastest response without overshooting. The result should be a response that is many times faster than that achievable with the feedback control system used in Problem 10. Plot the results of your saccadic simulation superimposed with the results of the vergence simulation found in Problem 10. Note the pulse and step gains in the title of the graph.

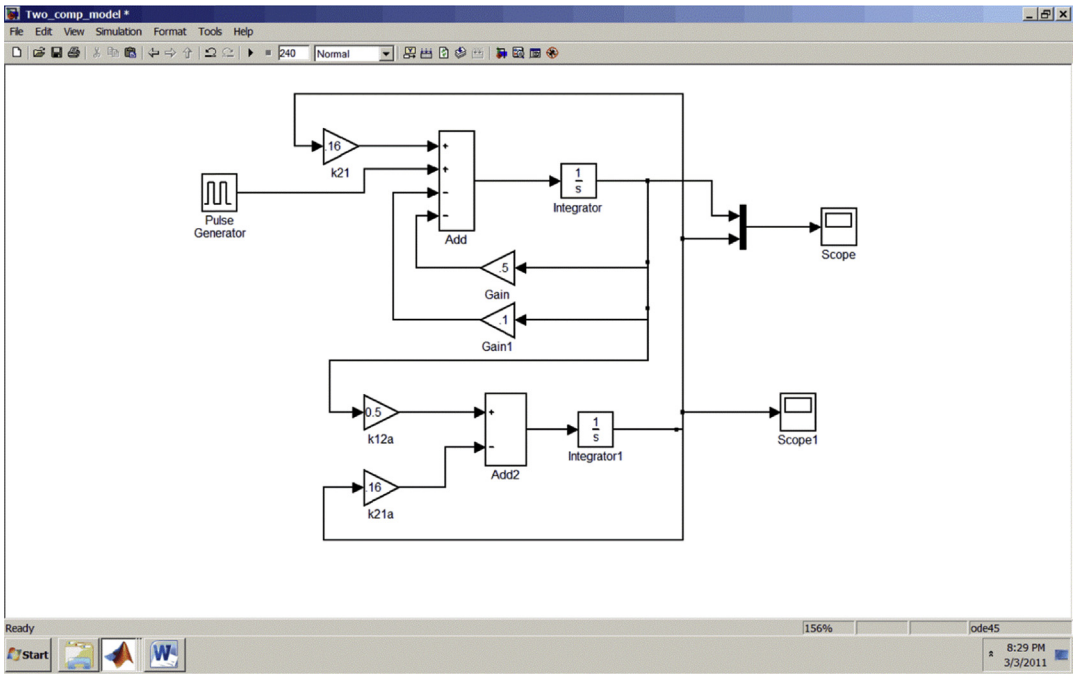
12. Use Simulink and the neuromuscular reflex model developed by Khoo (2000) and presented in Example 9.10 to investigate the effect of changing feedback gain on final value. Set the feedback gain to values: 9, 20, 40, 60, 90, 130, and 160. (Recall that the final value is a measure of the position error induced by the added force on the arm.) Measure the final error induced by the steplike addition of the 5 kg force: the same force used in Example 9.10. Lengthen the simulation time to 3 s and use the scope output to find the value of the angle at the end of the simulation period. Plot the final angle (i.e., force-induced error) versus the feedback gain using standard MATLAB. Label the axes correctly.

13. Use Simulink and the neuromuscular reflex model developed by Khoo (2000) and presented in Example 9.10 to investigate the effect of changing feedback gain on response overshoot. Set the feedback gain to values: 40, 60, 90, 130, and 160. Use the scope to measure the response overshoot in percent as the maximum overshoot divided by the final angle for three values of delay: 0, 20, and 30 ms. If there is no overshoot, record it as 0%. Plot percent overshoot versus feedback gain for the various delays using standard MATLAB. Label the axes correctly.

14. The figure below shows a physiologically realistic model of the vergence eye movement control system developed by Krishnan and Start in the 1960s. (Vergence eye movements are used and defined in Problem 10.) The controller of this model contains two elements of a PID controller: the upper path is the derivative control and the lower path the proportional control. (Note that the plant is a first-order approximation of that used in Problem 10.) Transcribe the model into Simulink and use simulation to find the first 2 s of the response to a 4.0-deg step input. (Use a fixed step of 0.005 s so you can compare the simulated response with actual data found as verg_movement in file vergence_resp.mat.) Increase the value of k until the response just begins to overshoot and note the improved speed of response. Plot the responses to your optimal value of derivative gain, k, and without derivative control; i.e., $k = 0$. On the same graph, plot the actual data. You will note the simulated data falls short of the actual response in two ways: there is a fairly large steady-state error, and the dynamics are slightly slower. An added integral controller could solve both these problems as shown in Problem 10.



15. The model shown below is a two-compartment circulatory pharmacokinetic model developed by Richard Upton of the University of Adelaide in Australia. This model has interacting compartments and can become unstable if the rate constants become imbalanced. Simulate the model using Simulink for the following parameter values: $k21 = k21a = 0.2$; $k12a = 0.5$; Gain = 0.5; Gain1 = 0.1. Then reduce the parameter $k21a$ to 0.15 and note that the model becomes unstable with an exponential increase in compartment concentrations. Find the lowest value of $k21a$ for which the model is stable. (Note: The vertical black bar combines the two input signal into an array so that the scope displays both signals. If you use a To Workspace output element with this signal, plot(tout,simout) plots both signals superimposed.)



16. Evaluate the Mackey and Glass model of neutrophils shown in Figure 9.41 for sensitivity to initial conditions. Use the parameters specified in Table 9.4 with a maturation delay, T_D, of 20 days. Plot the response when $IC = 0.10$ and superimpose the response when the $IC = 0.095$. Note the differences in the later response caused by this minor change in initial condition.
17. Construct a model based on the nonlinear van der Pol oscillator equation:

$$\frac{d^2x}{dt^2} = c\left(1 - x^2\right)\frac{dx}{dt} - x$$

TABLE 9.4 Parameters used in the Mackey and Glass model of Neutrophil Density

Parameter	Value	Description
γ	0.1	Neutrophil extinction
θ	1.0	Neutrophil production as function of past density
n	10	
β	0.2	Scale factor
T_D	20	Maturation time
IC	0.1	Initial condition

Plot the response of the system with an initial condition for x of 0.1 and values of c of 2.0 and 5.0. Make the simulation time 100 s. Note the sustained oscillations and the change in frequency with c. (Hint. Start with d^2x/dt^2 and integrate to get dx/dt and integrate again to get signal $x(t)$. Put the initial condition into the second integrator. Use x^2, gain c, and a constant 1.0 to construct $c(1 - x^2)$. You can use the Math Function with the option "square" to generate x^2. Then multiply this signal by dx/dt. Sum the resultant signal with $-x$ to get d^2x/dt^2, the input to the first integrator.) Be sure to save this model for possible use in Problem 9 of Chapter 10.