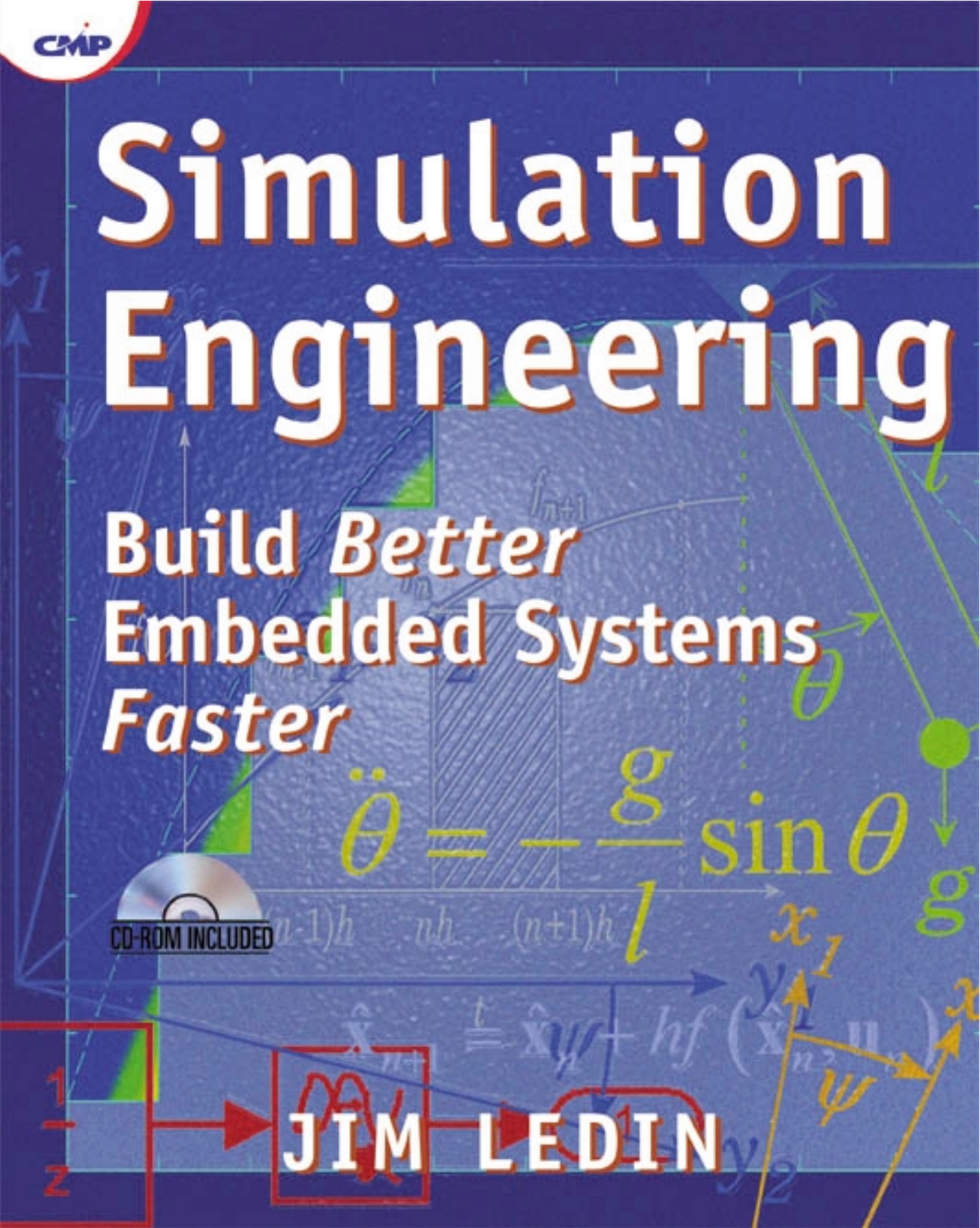
CMP

Simulation Engineering

Build *Better* Embedded Systems *Faster*



JIM LEDIN

Simulation Engineering

Jim Ledin

CMP Books
Lawrence, Kansas 66046

CMP Books
CMP Media LLC
1601 W. 23rd Street, Suite 200
Lawrence, KS 66046
USA
www.cmpbooks.com

Acquisitions Editor: Berney Williams
Editors: Michelle O'Neal, Rita Sooby, and Robert Ward
Layout Production: Kristi McAlister
Cover Art Design: John Freeman

Distributed in the U.S. and Canada by:
Publishers Group West
1700 Fourth Street
Berkeley, CA 94710
1-800-788-3123
www.pgw.com

ISBN: 1-57820-080-6

CMP*Books*

Dedicated to the memory of my father,
John Ronald Ledin.

Table of Contents

Chapter 2

Modeling Dynamic Systems

2.1 Introduction

To develop a simulation of an interesting and complex dynamic system, one must begin by developing mathematical models of the system components and the interactions between the system and its operational environment. A *mathematical model* is an algorithm or a set of equations and a set of related data values that together represent the significant behavior of a system, process, or phenomenon.

Depending on the system to be modeled, the development of a representative set of mathematical models may be an easy task or it may require a great deal of work. In cases where the system's dynamics are not well understood, it will be necessary for the developer to design and execute a series of experiments to collect data that can be used for model development.

This chapter introduces concepts involved in the mathematical modeling of dynamic systems and presents some of the basic techniques used in their development. The relevant properties of dynamic systems will be examined and I will provide examples of engineering techniques for deriving mathematical representations of their behavior. The positive and negative attributes of commonly used modeling approaches will also be discussed.

2.2 Dynamic Systems

A *dynamic system* has behavior that evolves over time. This behavior is commonly represented by *differential equations* if the system is of the continuous-time type or by *difference equations* if the system is of the discrete-time type. I will use the phrase *dynamic equations* to indicate the set of differential or difference equations that describe a system's behavior.

A system is *continuous-time* if its dynamic equations are valid at all points in time. A *discrete-time* system has dynamic equations that are updated or used only at discrete points in time. A dynamic system that is modeled using both difference equations and differential equations is called a combined discrete-continuous system, or simply a *combined system*.

2.2.1 Continuous-Time Systems

Some examples of continuous-time dynamic behaviors include:

- the translational and rotational motion of an aircraft,
- the orbital motion of a satellite,
- the response of a robotic arm to the motion of its actuators, and
- the response of an op-amp bandpass filter to its input signal.

The behavior of these systems can be represented mathematically by *differential equations*. A differential equation contains an unknown function and one or more of the function's derivatives. The goal is to find the unknown function, which will determine the system behavior over time.

Systems containing distributed parameters are described using *partial differential equations*, which contain partial derivatives. An example of a distributed parameter system is an electrical transmission line, which has resistance and inductance distributed continuously along its length. It is possible to approximate a distributed parameter system with a lumped parameter model, which contains a finite number of discrete locations where energy can be stored and dissipated. Lumped parameter models can be represented by *ordinary differential equations*, which contain ordinary derivatives rather than partial derivatives.

Note: This book will consider the modeling of dynamic systems using ordinary differential equations rather than partial differential equations. Therefore, distributed parameter systems will always be represented by lumped parameter models. When the phrase "differential equation" appears, it will refer to an ordinary (rather than partial) differential equation.

It is often necessary to begin the simulation implementation process with a mathematical model that is in a format *other* than differential equations and transform it so that it becomes a set of differential equations. In engineering analysis and design processes, continuous-time dynamic systems are usually studied using one of three formats [1]. These are the *s*-plane, the frequency response, and state–space representations. These formats are usually used to represent systems as *linear time-invariant models*. Linear time-invariant models and the three model formats are discussed in the following sections.

Linear Time-Invariant Models

Mathematically, a model is *linear* if it satisfies the following condition. Start with two model input signals $x_1(t)$ and $x_2(t)$, and their corresponding output signals $y_1(t)$ and $y_2(t)$. Create a new input signal that is the sum $x_1(t) + x_2(t)$ and apply it as an input to the model. If the model output equals $y_1(t) + y_2(t)$ for any arbitrarily selected $x_1(t)$ and $x_2(t)$, the model is linear. A model is *time-invariant* if the dynamic equations that define its behavior do not change as a function of time. A linear time-invariant model combines both of these properties.

The s-plane Format

The *s*-plane format is based on the Laplace transformation [2]. This technique is widely applied in the areas of classical control system analysis and design, though it is used less often in the simulation of nonlinear systems. *s*-plane models are typically used to represent linear time-invariant models. These models possess desirable properties for system analysis and control system design.

Models in the *s*-plane format are often represented as *transfer functions* consisting of a ratio of polynomials in the complex variable *s*. A transfer function represents the ratio of output to input in the *s*-domain. An example linear differential equation is shown in Equation 2.1 and its equivalent transfer function is given in Equation 2.2. Lowercase *x* and *y* indicate the signals in the time domain and uppercase *X* and *Y* indicate the *s*-domain representation. A single prime over a variable indicates a first derivative with respect to time. Two primes indicate a second time derivative, and so on.

2.1 $$y'' + 1.8y' + 100y = 100x$$

2.2 $$H(s) = \frac{Y}{X} = \frac{100}{s^2 + 1.8s + 100}$$

In many simulation environments, it is possible to model systems using differential equations but not transfer functions. For these environments, it is necessary to convert transfer functions to differential equations. It is a straightforward procedure to convert from an *s*-plane transfer function to an equivalent differential equation, assuming zero initial conditions. The steps are:

1. Given a transfer function in the format of Equation 2.2, multiply through by both denominators.
2. Wherever the variable *s* multiplies a variable, replace that term with the derivative of the same order as the power of *s* that is multiplying it.
3. Change the uppercase *X* and *Y* variables to lowercase.

Equation 2.3 shows the intermediate step in converting Equation 2.2 to Equation 2.1 where the multiplication through by the denominators in Equation 2.2 has taken place. The next step is to replace the terms s^2Y and $1.8sY$ by $\ddot{y}$ and $1.8\dot{y}$. Finally, the remaining uppercase *X* and *Y* variables are changed to lowercase variables and the result differential equation is as shown in Equation 2.1.

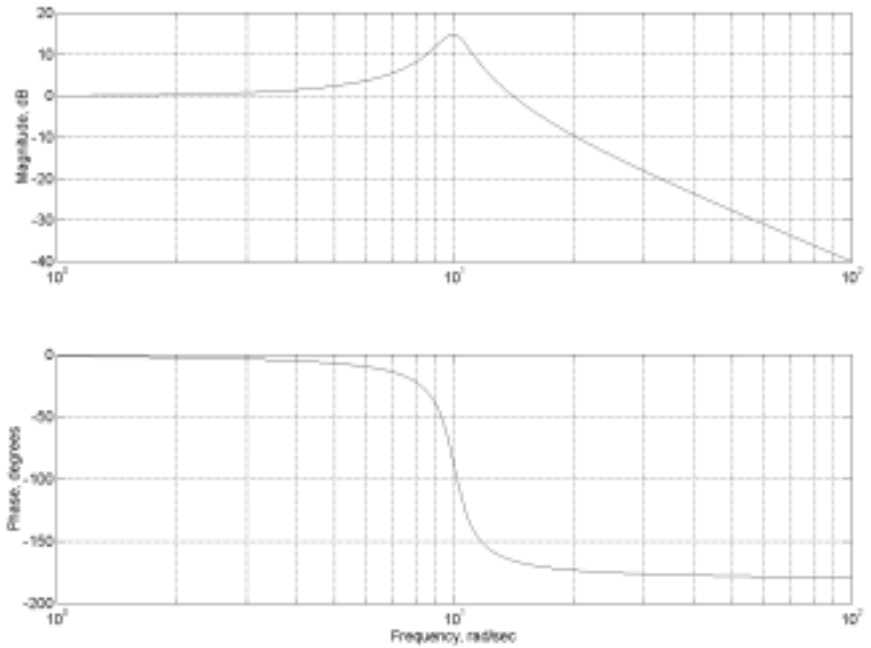
2.3 $$s^2Y + 1.8sY + 100Y = 100X$$

The Frequency Response Format

The frequency response format describes a system's behavior as its response in magnitude and phase to a sinusoidal input signal at various frequencies. The frequency response of a system can be determined empirically, which makes it useful for systems that are not understood well enough to model using the equations of physics.

Figure 2.1 shows a frequency response representation of the transfer function in Equation 2.2. The magnitude and phase of the transfer function are displayed in the form of *Bode plots*. Bode plots show the ratio of the magnitude of the output signal to the magnitude of the input signal in *decibels* and the phase lag of the output signal relative to the input signal in degrees. The horizontal axis in both plots is the frequency of the input signal in radians per second displayed on a logarithmic scale.

Figure 2.1 Bode plots of the response of Equation 2.2.



The decibel (or dB for short) is a way of expressing a ratio of two quantities. dBs are used instead of simple ratios because very large and very small ratios can be described with numbers of reasonable size. Another reason is that two ratios can be multiplied or divided by adding or subtracting their values in dB, which simplifies calculations. The mathematical definition of gain in dB appears in Equation 2.4, where z is the ratio of y to x in dB.

2.4 $$z = 20\log_{10}\left|\frac{y}{x}\right| \text{ dB}$$

The vertical bars around the quantity y/x in Equation 2.4 represent the absolute value of the ratio. z will be negative if the magnitude of y is smaller than that of x. Using this formula, if y is 1/1000 of x, z will equal –60 dB. Some other examples:

- If z is –1 dB, the ratio y/x is 0.89.
- If z is –6 dB, the ratio y/x is 0.50.
- If z is +6 dB, the ratio y/x is 2.0.

A frequency response cannot be used directly in a simulation. First, it must be transformed into a format suitable for implementing a simulation model. Assuming the system represented in the frequency response data is approximately linear, it is possible to develop an *s*-domain model that has a frequency response approximating the measured data. One (tedious) way to develop this model is by manually adjusting the coefficients of an *s*-domain model until its frequency response matches the system's frequency response to some degree.

Alternatively, the techniques of *system identification* [3] can be applied to develop a model from experimental data. System identification uses a computer program to process the sampled input signal and output signal from a test of the dynamic system. The system identification algorithms adjust the model parameters until the output of the model matches the output of the system as closely as possible. The model that results from system identification of a continuous system will typically be linear and time-invariant, and will usually be in the *s*-domain format. This approach will be discussed further in the section "System Identification" on page 52.

The State–Space Format

The state–space representation models a system as a set of first-order linear differential equations using matrix methods. As an example, Newton's law for a mass M moving in one dimension x under the influence of a force F is shown in Equation 2.5. A state–space representation of this second-order linear differential equation is shown in Equation 2.6. In this representation, the variable x_1 is the position of the mass and x_2 is its velocity.

2.5 $$Mx'' = F$$

2.6 $$\begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \frac{F}{M}$$

When working with linear time-invariant systems it is possible to transform state–space models to equivalent transfer functions and vice versa. It is also possible to convert a state–space model containing N first-order equations into a single Nth-order differential equation. However, the state–space representation is ideal for simulation because the numerical algorithms that we use for solving differential equations apply only to first-order equations.

The three system representation formats described rely on the assumption that the system being modeled is linear and (often) time-invariant. These assumptions are only reasonable

under specific conditions. For example, an aircraft can be represented as a linear time-invariant system when it is in a steady-state cruise condition, where the velocity, altitude, and pitch orientation are approximately constant over some period of time. Under these circumstances, it is reasonable (and accurate) for many purposes to assume that the response of the aircraft to the small control inputs used to maintain altitude, heading, and airspeed is linear.

It is not reasonable to expect a linear time-invariant model to be useful for simulating the flight of the aircraft from the start of the takeoff run until it reaches cruising altitude because the flight conditions change drastically in the transition from a low speed, low altitude takeoff environment to a high speed, high altitude cruise condition. A high fidelity simulation model of this aircraft is necessarily nonlinear and time-varying to account for the changes in dynamic behavior as different flight regimes are encountered. Nonlinear and time-varying dynamic behavior can be modeled in a straightforward manner using differential equations, as the next section will show.

The Differential Equation Format

D*ifferential equations* are the general format for representing dynamic systems. They contain an unknown function and one or more of its derivatives. The *order* of a differential equation is the order of the highest derivative appearing in it. The solution of a differential equation is a function that satisfies the equation at all points and that also satisfies any associated boundary conditions.

High fidelity dynamic equations representing real-world systems tend to be nonlinear and time-varying. The formats for modeling dynamic systems discussed previously (*s*-plane, frequency response, and state–space) are usually limited to linear time-invariant models. This is because many engineering analysis and design techniques are available only for linear time-invariant models and are not applicable to nonlinear models.

A standard technique for developing a linear system model is to perform a *Taylor series expansion* of the nonlinear, time-varying dynamic equations about a stable operating point. An example of a stable operating point is the aircraft in a steady-state cruise condition as discussed in the previous section. A variety of analysis techniques can be used with the linearized system model.

Simulation uses a different approach for examining system behavior. A simulation can model the behavior of a nonlinear, time-varying system just as easily as it can model a linear, time-invariant system. The basic method used in simulation is to numerically compute estimates of the solutions of the system's dynamic equations. Although techniques exist for finding exact analytic solutions of some categories of dynamic equations, this is usually not possible unless the equations are linear and have mathematically simple input signals such as a step function or a sine wave. Dynamic systems often have input signals that are not simple in a mathematical sense. An example of a mathematically complex input function is the rotational position of an automobile steering wheel as the driver travels along a road. For these reasons, the approach most commonly used in dynamic system simulation is to perform approximate numerical integration of the dynamic equations. The details of several algorithms for numerical integration will be examined in Chapter 3.

Numerical integration algorithms operate on first-order differential equations only. This means that if a dynamic equation contains second (or higher) derivatives, it must be transformed into an equivalent set of first-order differential equations. This is a simple procedure.

1. Solve for the highest order derivative. This places the equation into the form $x^{(n)} = f(t, x, x', ..., x^{(n-1)})$ where $x^{(n)}$ is the nth-order time derivative.
2. Make the following substitutions for the function and its derivatives: $x_1 = x$, $x_2 = x'$, $x_3 = x''$, and so on. This changes the equation into the form $x'_n = f(t, x_1, x_2, ..., x_n)$, a first-order differential equation.
3. Write first-order differential equations for each of the variables x_1 through x_{n-1} as follows: $x'_1 = x_2$, $x'_2 = x_3 \ldots$, $x'_{n-1} = x_n$.

Here's an example. Equation 2.7 is a second-order nonlinear differential equation. The result of solving Equation 2.7 for the highest derivative appears in Equation 2.8. Equation 2.9 is a set of two first-order differential equations that are equivalent to Equation 2.7. In Equation 2.9, x_1 is equal to the solution function x of Equation 2.7 and x_2 is equal to the derivative x' in Equation 2.7. The variables x_1 and x_2 are referred to as *state variables*.

2.7 $$x'' + 3x'^2 + 5x = 1$$

2.8 $$x'' = -3x'^2 - 5x + 1$$

2.9 $$x'_2 = -3x^2_2 - 5x_1 + 1$$
$$x'_1 = x_2$$

In general, many different solution functions can satisfy a differential equation such as Equation 2.7. Additional information in the form of boundary conditions must be provided to identify the solution of interest. In dynamic system simulation, boundary conditions are specified as *initial conditions* on the state variables. The initial conditions are the values of the state variables at the start of simulation execution.

Equation 2.10 shows an example set of initial conditions at time zero that, combined with Equation 2.9, uniquely specify the solution to the dynamic equation represented by Equation 2.7.

2.10 $$x_2(0) = 0$$
$$x'_1(0) = 0$$

2.2.2 Discrete-Time Systems

If the output of a system is updated or used only at discrete points in time, the system can be represented as a *discrete-time system*. A discrete-time system is described by a set of difference equations. One reason that this kind of modeling is of interest in the development of dynamic embedded systems is that the behavior of an embedded computer control system is well represented by a discrete-time model. This concept is applicable to embedded control systems that sample their inputs at discrete points in time, perform processing, and then update their outputs, with this cycle repeating at fixed time intervals.

In a general discrete-time system, the output at any time is some function of the current system input, previous input values, and previous output values. As with continuous systems described by differential equations in the previous section, difference equations can be linear or nonlinear and time-invariant or time-varying.

In engineering analysis and design, linear time-invariant models of discrete-time system are used in ways similar to the linear time-invariant models of continuous-time systems. For simulation purposes, it does not matter if the difference equation is linear and time-invariant or if it is nonlinear and time-varying. An example nonlinear difference equation is shown in Equation 2.11. The subscripts in Equation 2.11 represent the sample number in the discrete system's input and output sequences. y_{n+1} is the system output value at the next time step, x_n is the current input value, x_{n-1} is the input value of the previous step, y_n is the current output value, and y_{n-1} is the output value of the previous step.

$$y_{n+1} = \frac{1}{2}x_n + \frac{1}{4}x_{n-1} + \frac{1}{8}y_n^2 + \frac{1}{16}y_{n-1}^2 \tag{2.11}$$

The order of a difference equation is determined by the oldest previous output value that appears in the equation. Equation 2.11 is a second-order difference equation because y_{n-1} appears on the righthand side, which is two steps older than the equation output y_{n+1}. If no previous output values appear in the equation, the order of the difference equation is determined by the oldest previous input value instead.

Unlike the case of high order differential equations, there is no need to represent a high order difference equation as a set of first-order equations for simulation purposes. However, difference equations are similar to differential equations in that it is necessary to provide initial conditions to uniquely specify a solution. In Equation 2.11 — assuming that the system begins operating at $n = 0$ — the initial conditions would be y_0, y_{-1}, and x_{-1}.

2.3 Mathematical Modeling

A mathematical model is an algorithm or a set of equations that represents the interesting behavior of a system. Experts with thorough knowledge of the system and its interaction with the environment typically perform model creation tasks in development projects for complex dynamic systems. The development of a model for a complex dynamic system is an iterative process that involves significant effort to verify the correctness and accuracy of the resulting implementation.

The process of model development begins with a specification of the requirements the model must meet. The following are issues that must be addressed in developing a model of a complex system.

What effects should be included in the model? A system may exhibit many different kinds of behavior (for example, the motion of motors, vibration, wear of moving parts, etc.), but not all of these behaviors need to be modeled to produce an effective simulation. Limiting the effects modeled to only those that are truly necessary will make the model less complex and easier to build, test, and maintain — as well as requiring less computational resources to execute.

How detailed must the model be? In many cases, a simple model is all that is needed, but if precise determination of system behavior is required, the model may need to be very elaborate.

What interactions between the system and the outside environment must be modeled? For example, a communication satellite motion model must operate in conjunction with a model of the earth's gravitational field, as well as with models of other relevant phenomena such as solar pressure.

What techniques will be used to develop the model? A fundamental choice is whether to use physics-based equations or measured data as the basis for the model. The answer to this question is often obvious to those with expert knowledge of the system.

What data must be gathered to perform the modeling? For example, an aerodynamic model of an aircraft may require extensive wind tunnel testing.

How much time and how many people are available to develop and test the model? As model complexity increases, the development and test hours will increase as well.

What computing resources are available for the model? A large model may consume significant amounts of memory, disk space, and CPU time. However, given the capabilities of current computers, this may not be a critical issue.

Will the model eventually be used in a hardware-in-the-loop (HIL) simulation? This may place severe constraints on the execution time allowed for the model. Alternatively, a complex model may require high performance computing hardware for use in an HIL simulation, perhaps involving the use of multiple processors.

How can verification and validation be performed for the model implementation? There must be reasonable ways of confirming that the model has been implemented correctly and that its behavior matches the system being modeled to an acceptable degree.

These issues should be addressed as part of planning for the simulation effort. The questions listed can be applied at the highest level of the entire system being simulated initially and again as the system is broken down into subsystems and individual components to be modeled. These questions are also useful in the development of additional models needed for a complete simulation, such as the gravitational field and solar pressure models in the communication satellite example above.

Although our focus is on the mathematical modeling of dynamic systems, note that the models of system components and the operational environment will not always contain dynamic behavior. For example, the motion of an aircraft in response to pilot control inputs is represented by dynamic equations. However, the atmospheric properties (air temperature, pressure, and density) at the aircraft's location are often modeled as a set of equations that depend only on the aircraft's altitude. No dynamic behavior is involved in this atmosphere model, only the determination of atmospheric attributes at a given aircraft altitude. The point here is that not all models that go into a simulation will necessarily include dynamic behavior.

2.3.1 Level of Model Complexity

The required level of complexity in a mathematical model can be determined by finding answers to the first two questions in the previous section, i.e., which effects to model and the level of modeling detail required. For most systems intricate enough to be worthy of simulation, a large number of effects can be identified that potentially have some bearing on system performance. The model developer must determine which of these effects are truly significant and which can be ignored. This is partially an economic decision because as more effects are added to a model, the amount of time and money needed to develop and validate the model will increase.

An example of limiting the number of effects modeled occurs in modeling the orbit of an earth satellite. In theory, the motion of the satellite will be perturbed by all of the massive bodies in the solar system and beyond. This set of bodies includes the earth, moon, sun, all the other planets, asteroids, distant stars, etc. In reality, the satellite motion is primarily influenced by a limited number of bodies, perhaps just the earth, moon, and sun. The developer can ignore the effects of the other bodies or may wish to treat them as a random disturbance, depending on the goals for the simulation. Selecting which bodies to model and which to ignore provides an answer to the first question.

Now to address the second question: the issue of model detail. A simple model for the gravitational field of the earth assumes that it is a perfect sphere and the gravitational field is uniform in all directions. The earth is actually nonspherical (it is slightly oblate) and this affects the gravitational field. Furthermore, the strength of the gravitational field varies at different locations due to local differences in the density of the earth. Thus, there are at least three levels of modeling detail for the earth gravitational field that could be selected: a perfect uniform sphere, an oblate sphere, and locally varying gravity. Each of these levels of model detail requires a different level of effort to implement and test, and each requires a different quantity and type of data that must be included in the model. Selecting which level of detail to use answers the second question.

One helpful approach when dealing with these issues is to begin with a relatively simple model containing a limited set of effects and a coarse level of model detail. Then, as the developer gains experience with the simulation, more effects and model details can be added as the need for them becomes clear. Often in the early stages of simulation development, it is not obvious which effects and model details are truly significant. If a large number of effects and model details are included in the initial design for the model, it may turn out that much effort has been wasted modeling things that turn out to be trivial in determining system performance.

If the software interfaces to each model are clearly and completely defined, it should be possible to replace these low fidelity models with higher fidelity versions without any significant changes in the rest of the simulation. Instead of replacing the original models, however, it may be more useful to maintain multiple levels of fidelity for particular models in the simulation simultaneously. This will allow the simulation user to select the desired level of fidelity of individual models as part of the simulation input data set. The availability of multiple model fidelity levels in a simulation allows the user to perform detailed modeling of particular effects or model details when needed. When this level of detail is not required, lower fidelity models can be used instead, which may reduce simulation execution time. Sometimes lower fidelity models execute at speeds orders of magnitude faster than higher fidelity versions. The

ability to trade model detail for execution speed can help make the simulation a valuable tool for a variety of applications.

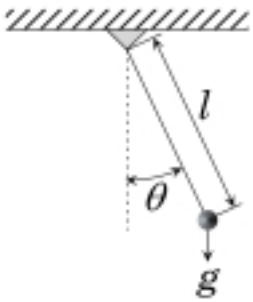
2.4 Modeling Methods

This section will discuss some techniques for developing the equations and data sets for a mathematical model. A model is "physics-based" if it is based on the equations of generally accepted physical laws. A spacecraft orbital model based on Newton's law of motion is an example of a physics-based model. Many systems have behavior that is too complex to represent in terms of the laws of physics. An example of this situation is the aerodynamic performance of a supersonic aircraft, which tends to be very nonlinear and difficult to represent using the equations of physics. In this case, the only reasonable approach for model development may be to measure the behavior of the system with a sub-scale model in a wind tunnel and use this data to create a set of interpolation tables. This approach leads to an "empirical" model.

2.4.1 Physics-Based Modeling: A Simple Pendulum Example

Figure 2.2 shows a pendulum suspended from a string of length l under the influence of gravitational acceleration g. The pendulum angular deflection with respect to the vertical is θ, given in radians. The mass of the pendulum bob is defined to be m. The goal for this model is to determine the period of oscillation of the pendulum as a function of the initial deflection angle θ_0, assuming that the initial velocity $\dot{\theta}_0$ is zero.

Figure 2.2 Simple pendulum.



To determine the oscillation period, begin by considering which effects are significant. Look at the relevant physical effects and determine which ones to include in the model:

- Gravity must be modeled because the pendulum would not move without it.
- The mass of the pendulum bob must be modeled for the same reason.
- If we assume that the size of the bob is small in comparison to the length of the string, the bob can be modeled as a point mass. This simplifies the model significantly.

- If the mass of the string is much less than that of the bob, the mass of the string can be ignored.
- Friction within the string will be assumed to be a small effect over short time periods and will be ignored.
- The pendulum will be assumed to move slowly so that air resistance is not a significant factor over a short time period.

We know that a real pendulum will eventually slow down and stop due to friction in the string and air resistance. This is not the kind of behavior we are interested in, so we will modify our goal to be the determination of the oscillation period at the time motion is started. This assumes that the pendulum slows gradually and the oscillation period changes slowly.

We have made several simplifying assumptions that will ease the model development task. Next, apply the laws of physics to develop dynamic equations for the system. Only attempt to model the effects that were determined to be relevant in the previous analysis.

The component of gravitational force that affects the motion of the pendulum bob is in the direction perpendicular to the string. This force is defined in Equation 2.12. The gravitational force component parallel to the string will create tension in the string, but it will not affect the motion of the bob, so ignore it.

2.12 $$F = -mg\sin\theta$$

Note that the force F will always be acting to move the bob back towards the center position. Applying Newton's law $F = ma$ to the problem leads to Equation 2.13, where a is the tangential acceleration of the bob.

2.13 $$a = -g\sin\theta$$

The acceleration a is related to the angle θ by the equation $a = l\theta''$. This leads to the final dynamic equation of Equation 2.14.

2.14 $$\theta'' = -\frac{g}{l}\sin\theta$$

Note that this equation does not depend on the mass of the bob m, however it does depend on the assumptions listed previously. It is also a nonlinear differential equation because it contains the term $\sin\theta$. To completely determine a solution for this equation, the initial conditions of the system must be specified as shown in Equation 2.15. The parameter θ_0 in Equation 2.15 is the angle from which the bob is released at time zero with an initial velocity of zero. For this system, the possible values for θ_0 are assumed to lie in the range

$$[-\frac{\pi}{2},\frac{\pi}{2}].$$

2.15 $$\theta(0) = \theta_0$$

$$\theta'(0) = 0$$

To make Equation 2.14 suitable for simulation, it must be transformed into a set of first-order differential equations using the procedure discussed in The Differential Equation Format on page 26. The resulting first-order equations and corresponding initial conditions are shown in Equation 2.16.

$$2.16 \quad \theta'_2 = -\frac{g}{l}\sin\theta_1$$

$$\theta'_1 = \theta_2$$

$$\theta_1(0) = \theta_0$$

$$\theta_2(0) = 0$$

Using the techniques of numerical integration (discussed in Chapter 3), we can solve these equations for various values of θ_0 and the oscillation period can be determined from examining the solutions.

Pendulum Simulation with the DSSL

Equation 2.16 represents a model of this dynamic system in differential equation format. With the use of the DSSL C++ routines, a complete simulation of this system is shown in Listing 2.1.

Listing 2.1 Pendulum.cpp

```
// Pendulum simulation
#include <dssl.h>

#include <cstdio>
#include <cmath>

int main()
{
    // Define the state variables
    StateList state_list;
    State<> theta(&state_list), theta_dot(&state_list);

    // Integration step size and simulation end time
    const double step_time = 0.01, end_time = 10.0;

    // Set the initial conditions
    theta.ic = 0.5;
    theta_dot.ic = 0.0;
```

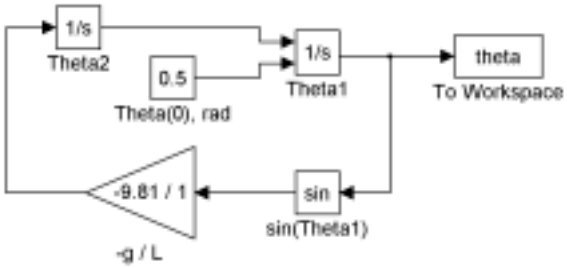
```
    state_list.Initialize(step_time);

    // Pendulum model parameters
    const double g = 9.81;
    const double L = 1.0;

    // Open an output file
    FILE* iov = fopen("pendulum.csv", "w");
    assert(iov);

    fprintf(iov, "Time, Theta\n");
    for(;;)
    {
        // Pendulum dynamic equations
        theta_dot.der = -(g/L) * sin(theta);
        theta.der = theta_dot;

        fprintf(iov, "%6.2lf, %9.6lf\n", state_list.Time(), double(theta));

        if (state_list.Time() >= end_time)
            break;

        state_list.Integrate();
    }

    fclose(iov);
    return 0;
}
```

This program must be compiled using the header files in the DSSL directory. You must also include the file `StateList.cpp` from that directory in the compilation to produce an executable image. After the program has finished executing, the file `pendulum.csv` will be available for analysis using a spreadsheet program such as Microsoft Excel.

Pendulum Simulation in Simulink

An equivalent model of the pendulum can be implemented in Simulink as shown in Figure 2.3. Parameters such as the simulation stop time must be set from a dialog box prior to starting a run. At the end of the run, the `theta` variable in the MATLAB workspace will contain the time history of the `Theta1` Simulink block output. MATLAB data analysis and plotting commands can then be used to process and display the data.

Figure 2.3 Simulink pendulum model.



2.4.2 Linearization of Nonlinear Models

The technique of *linearization* is so common in engineering that it is worthwhile to examine some of the effects that can occur when it is used. The approach used in linearization is to identify a stable point or trajectory for a nonlinear system and model small variations about that point or trajectory using linear equations. It is possible to analyze the resulting model using a variety of mathematical methods suitable for use only with linear systems. To demonstrate the technique, this section will linearize the pendulum model from the previous section about the stable point at which the pendulum hangs straight down with zero velocity.

The nonlinear term in the pendulum model is the $\sin\theta$ term. If we make an assumption that the value of θ_0 is "small," Equation 2.14 can be modified with the approximation $\sin\theta \approx \theta$ (in radians). The limit for this approximation depends on the tolerable amount of error in the solution. This approximation results in Equation 2.17, which is now a linear differential equation that is solvable with standard calculus techniques. The solution to this equation, incorporating the initial conditions of Equation 2.15, is shown in Equation 2.18.

$$\ddot{\theta} = -\frac{g}{l}\theta \tag{2.17}$$

$$\theta(t) = \theta_0 \cos\sqrt{\frac{g}{l}}t \tag{2.18}$$

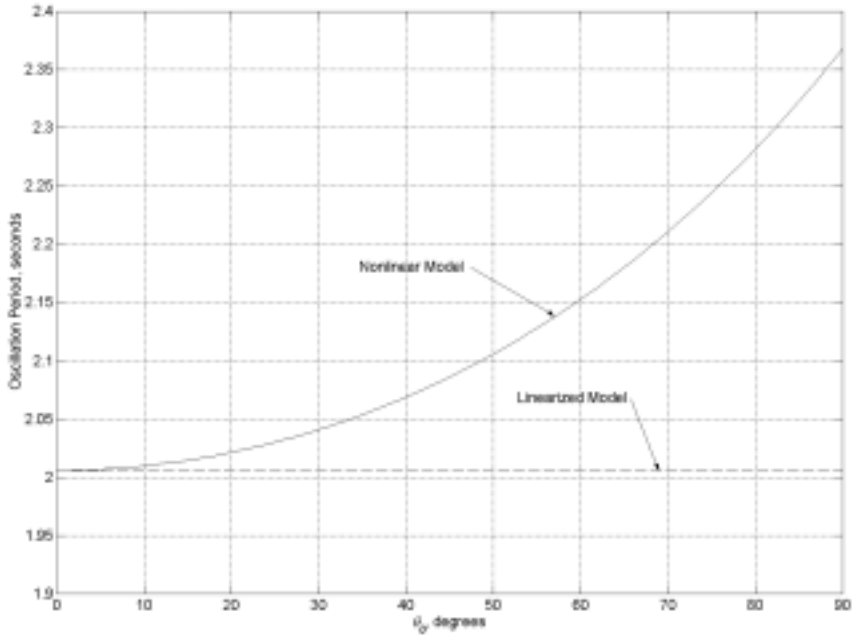
Equation 2.18 has an oscillation period of $2\pi\sqrt{\frac{l}{g}}$ seconds. Note that this period is independent of θ_0.

Our goal in solving this problem does not include the assumption of a small angle however, so we cannot employ this approximation. This example will show how the use of simplified, linear models in a simulation can produce unexpected and incorrect results when used inappropriately.

Figure 2.4 shows the results of numerically solving the nonlinear model of Equation 2.16 for values of θ_0 ranging from zero to 90 degrees and determining the oscillation period of each solution from the simulation output data. It also shows the oscillation period derived from the linear approximation to the solution, which is a constant for all θ_0. It is clear that as θ_0 approaches zero, the linear approximation becomes a good match to the nonlinear

model. It is also clear that using the linearized model will result in significant errors if θ_0 is large.

Figure 2.4 Comparison of nonlinear and linear pendulum model oscillation periods.



This example demonstrates the basic approach for developing a physics-based mathematical model of a dynamic system. Similar model development techniques are useful in other disciplines such as electronics and chemistry. These modeling techniques are applicable as long as the dynamic equations describing the system are well defined and the data values used in the equations are known with sufficient precision. In the pendulum example, the data values required were the gravitational acceleration *g*, the string length *l*, and the initial displacement θ_0. In addition, it is necessary to examine other data to verify that the assumptions used in the model development are reasonable — such as the assumption that the mass of the string is small relative to the mass of the bob.

Linear approximations of dynamic systems are used frequently in engineering, but their limitations should be well understood. Nonlinear system models are appropriate for use in simulation, and will result in more accurate results as compared to simplified linear models.

In situations where the dynamic equations or data values for a mathematical model are not known to a sufficient degree of precision, it is necessary to use alternative methods for model development. These techniques are discussed in the next section.

2.4.3 Empirical Modeling

Empirical modeling techniques use measured data from various types of experiments to develop a mathematical model of a system. In reality, all mathematical models are empirical to some degree. For example, the pendulum model in the previous section includes some experimentally determined constants. However, our interest in this section is on the development of models for systems with substantial dynamic behavior that is not readily modeled by known dynamic equations. The following sections present three empirical modeling techniques: table interpolation, system identification, and neural networks.

Table Interpolation

Table interpolation is a static modeling technique used to evaluate functions of the form shown in Equation 2.19. It is a static method because it does not permit the direct implementation of dynamic equations. However, table interpolation functions are useful in the construction of dynamic equations. For example, it is common to compute coefficients appearing in dynamic equations using table interpolation.

2.19 $y = f(x_1, x_2, x_3,...)$

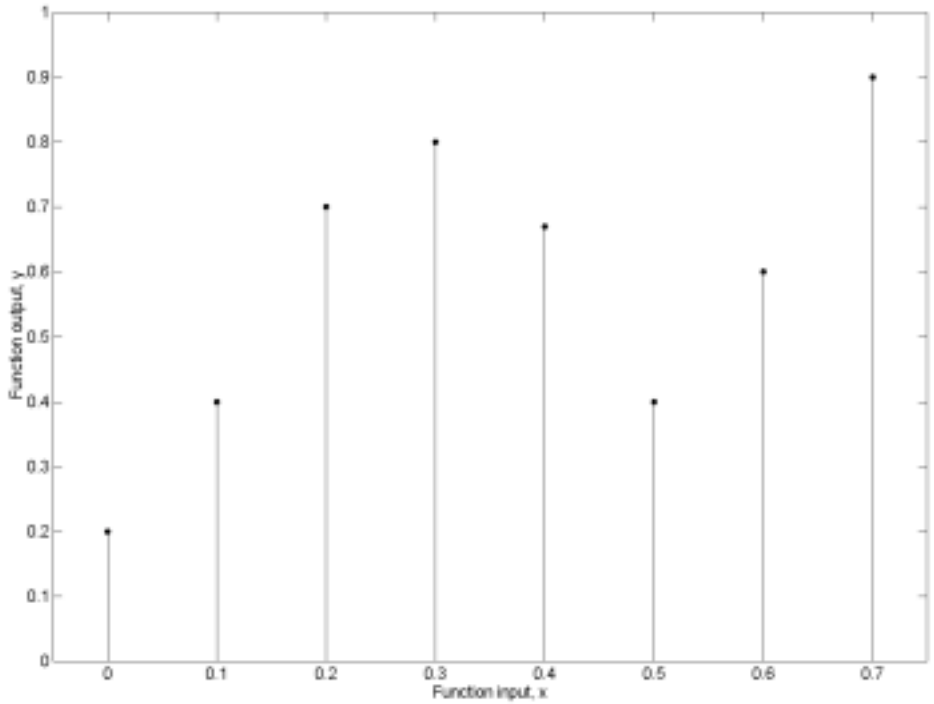
This approach is used when the function output must be determined experimentally. It is also applicable as a speed optimization technique if a lengthy computation (perhaps an iterative procedure) is required to evaluate the function. In that case, a table interpolation to estimate the result of the computation may execute many times faster than a direct computation.

The function inputs x_1, x_2, etc. can be any variable in the simulation — such as time, a state variable, or a constant. The number of function inputs is arbitrary, but in practical applications, it is usually five or less. As more inputs are added to the function, its memory requirements and execution time will increase. The output y depends only on the values of the function inputs at the time of evaluation.

An interpolation function with N inputs is evaluated with the use of an N-dimensional lookup table. Each input variable spans one dimension of the lookup table. For each table dimension, it is necessary to define a set of interpolation breakpoints which span the permissible range of the corresponding input variable. Each input variable can have a different number of interpolation breakpoints, and the breakpoints may be spaced equally across the span of the dimension or placed at arbitrary intervals.

A one-dimensional example of the data for a lookup table with eight equally-spaced breakpoints appears in Figure 2.5. The span of the input variable x is [0, 0.7]. If the input variable precisely matches the x location of one of the breakpoints, it is a simple matter to return the corresponding y value as the result of the function evaluation. If the input value falls between the breakpoints, an interpolation must be performed.

Figure 2.5 Example one-dimensional lookup table.

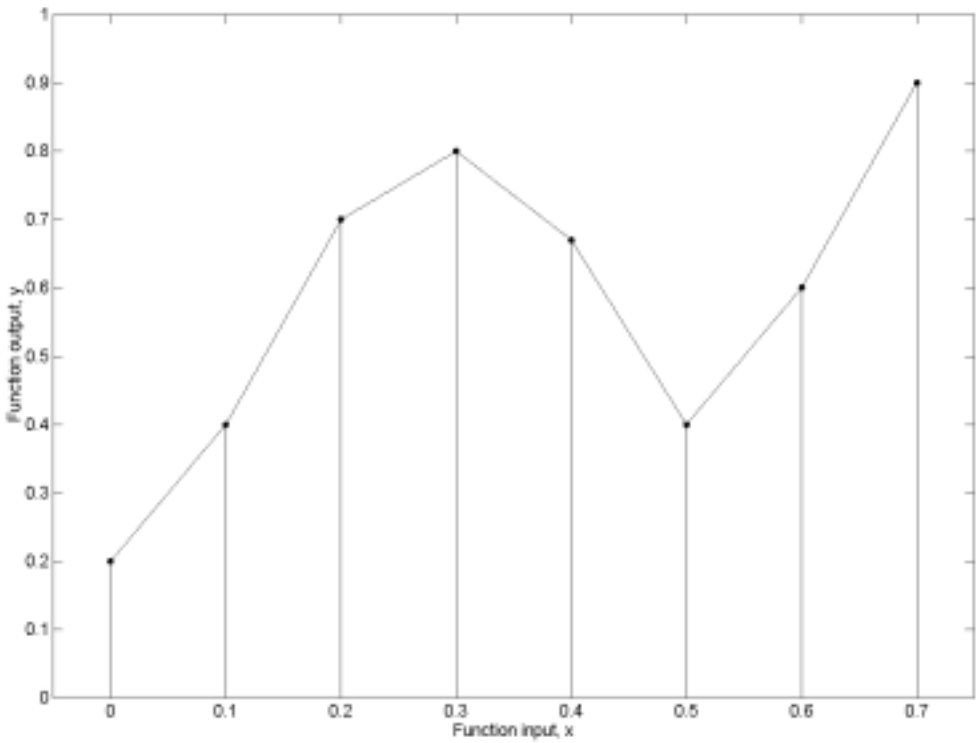


Many different techniques for performing interpolation exist that vary in computational complexity and smoothness of the interpolated function. Two methods that should satisfy most needs are: linear interpolation and cubic spline interpolation.

Linear Interpolation with Equally-Spaced Breakpoints

We can perform one-dimensional linear interpolation graphically by drawing straight lines between adjacent breakpoints as shown in Figure 2.6. The interpolated function is continuous and its derivative is discontinuous at the breakpoints.

Figure 2.6 Linear breakpoint interpolation.



One-dimensional linear interpolation using equally-spaced breakpoints is performed with the following steps. Assume that there are N breakpoints with y coordinates stored in an array with indexes that begin at zero. The value of $x(0)$, the leftmost x coordinate, and Δx, the interval between x coordinates, must also be provided.

1. Ensure that the input variable x_{in} has a value greater than or equal to $x(0)$ and less than or equal to $x(0) + (N - 1)\Delta x$. A limit function can be applied, if that is appropriate. It is also possible to linearly extrapolate outside the table using the first (or last) two data points in the table to define a straight line. However, this approach may introduce significant errors if the extrapolation does not accurately model the behavior of the function outside the range of the table — do not consider it here. It may make more sense to issue an error message and abort the simulation run if the input variable is outside the valid input range of the table.
2. Determine the array index of the closest breakpoint with an x coordinate that is less than or equal to the function input value. For equally-spaced breakpoints, the lower breakpoint index is computed as shown in Equation 2.20. Note that L is truncated to an integer.

$$2.20 \quad L = \left\lfloor \frac{x_{in} - x(0)}{\Delta x} \right\rfloor$$

In Equation 2.20, L is the index of the lower of the two breakpoints that surround the input value x_{in}, $x(0)$ is the x coordinate of the first breakpoint in the array, and Δx is the x interval between breakpoints. Based on the range limits placed on x_{in} in step 1, L will be in the range $0 \le L < N - 1$.

3. Perform linear interpolation between the breakpoints with indices L and $L + 1$ as shown in Equation 2.21. A special case occurs when $L = N - 1$, which is when x_{in} is located at the last breakpoint in the array and the correct interpolation result is $y = y(N - 1)$. When this happens, $y(L + 1)$ is undefined, although it ends up being multiplied by zero. It is important to handle this case properly to avoid potential memory access faults and floating point problems.

$$2.21 \quad y = y(L) + [y(L + 1) - y(L)]\frac{x_{in} - x(L)}{\Delta x}$$

Linear Interpolation with Unequally-Spaced Breakpoints

If the x coordinates of the breakpoints are not equally spaced, it takes more work to determine which breakpoint interval contains the function input value. A general approach for locating the correct interval is the technique of bisection — an algorithm for performing an efficient search of an ordered list.

The bisection algorithm locates the breakpoint pair surrounding the function input value. Assume that the x and y breakpoint coordinates are stored in arrays of length N that are indexed starting at zero.

1. Ensure that the input variable x_{in} has a value greater than or equal to the first breakpoint in the table and less than or equal to the last breakpoint.
2. Define an index variable L and initialize it to zero. Define an index variable U and initialize it to $N - 1$. These lower and upper indexes bracket the entire list initially.
3. Repeat the following steps until the quantity $(U - L)$ is equal to one:

   (a) Set the current index i to be $\frac{U + L}{2}$, truncated to an integer.

   (b) If the breakpoint at index i is greater than the input x_{in}, set $U = i$. Otherwise, set $L = i$.
4. Upon exiting the loop in the previous step, L will contain the index of the lower breakpoint of the correct breakpoint interval.
5. Perform linear interpolation between the breakpoints at indices L and $L + 1$ as shown in Equation 2.22.

$$2.22 \quad y = y(L) + [y(L + 1) - y(L)]\frac{x_{in} - x(L)}{x(L + 1) - x(L)}$$

Although the bisection method is the most general technique for locating the correct breakpoint interval, it may be possible to eliminate this search much of the time. If the input

value x changes slowly between evaluations of the function, the simple step of checking to see if the input is contained in the same breakpoint interval as the previous function evaluation will often eliminate the need for bisection. If the input is not in the same interval, bisection can then be performed.

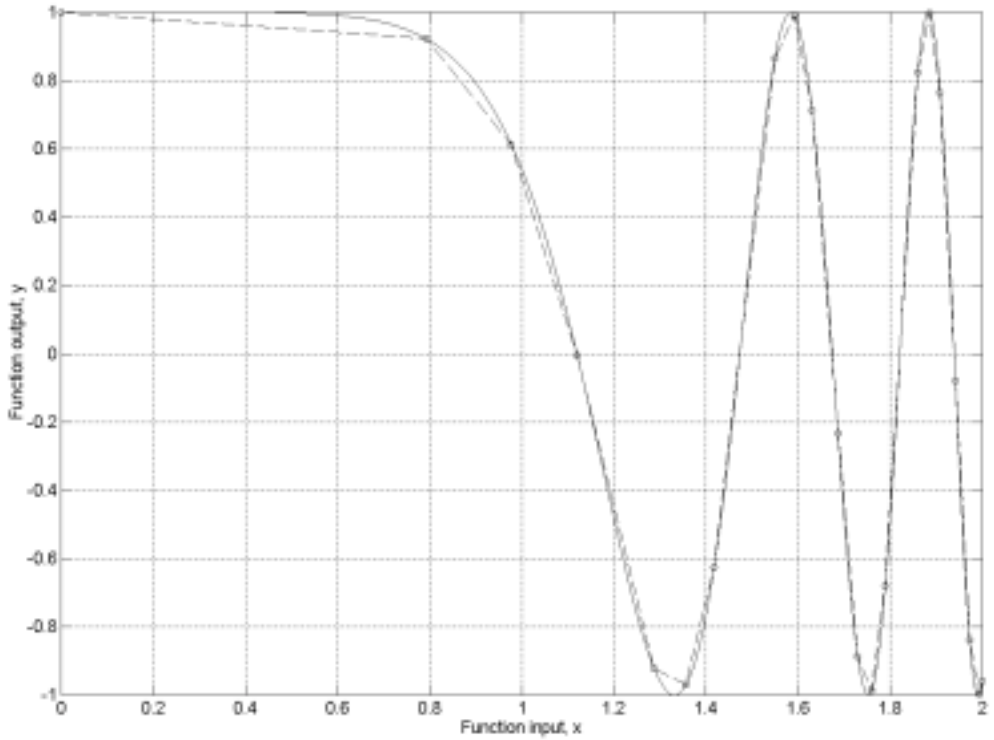
Alternatively, as a next step, the breakpoint intervals immediately above and below the previously-used interval can be checked, and bisection performed if the input does not lie in those intervals. The technique of checking the previously-used breakpoint interval followed by checking the adjacent intervals (if necessary) can sometimes eliminate the use of bisection completely, except for the very first function evaluation. This efficiency is realized when the input variable does not change quickly enough to jump over a breakpoint interval between function evaluations. The drawback of this technique is that, when the assumption of a slowly changing input turns out to be incorrect, the function evaluation process will be a bit slower due to the additional checking that precedes bisection.

On average, the bisection algorithm requires approximately $\log_2 N$ iterations of the loop in step 3 of the algorithm, which is considerably more time consuming than the direct computation used to locate the breakpoint interval when equally-spaced breakpoints are used. The advantage of using unequally-spaced breakpoints is that it may be possible to adequately model a function with a much smaller table than would be required with equally-spaced breakpoints. The points can be closely spaced in regions where the function has rapid fluctuations and they can be more widely spaced in regions where the function is relatively smooth. When using equally-spaced breakpoints, the points must be spaced closely enough to accommodate the most rapid fluctuations in the function even if these fluctuations only occurs over a small part of the input variable's span.

An example will clarify this point. We will use table interpolation to evaluate the function $y = \cos(x^4)$ over the input span [0,2] and compare the required table size for equally-spaced and unequally-spaced breakpoints. Require that the maximum interpolation error magnitude be no greater than 0.05 at any location along the curve.

Figure 2.7 shows the result of selecting unequally-spaced breakpoints to evaluate this function. The breakpoints were carefully selected to limit the error magnitude to the required 0.05 at any location between them. The breakpoints at both ends of the input range must always be included. Note how the breakpoints are widely spaced in the lower x values and are closely spaced as the function varies more rapidly in the higher x values. Twenty-one breakpoints are required in this case.

Figure 2.7 Unequally-spaced breakpoint interpolation of $y = \cos(x^4)$.



If equally-spaced breakpoints are used instead, it is necessary to use a breakpoint interval of 0.02 in order to limit the approximation error between breakpoints to 0.05 as shown in Figure 2.8. This requires 101 total breakpoints — a factor of 4.8 more points than are required in the unequally-spaced breakpoint implementation of this function.

Figure 2.8 Equally-spaced breakpoint interpolation of $y = \cos(x^4)$.

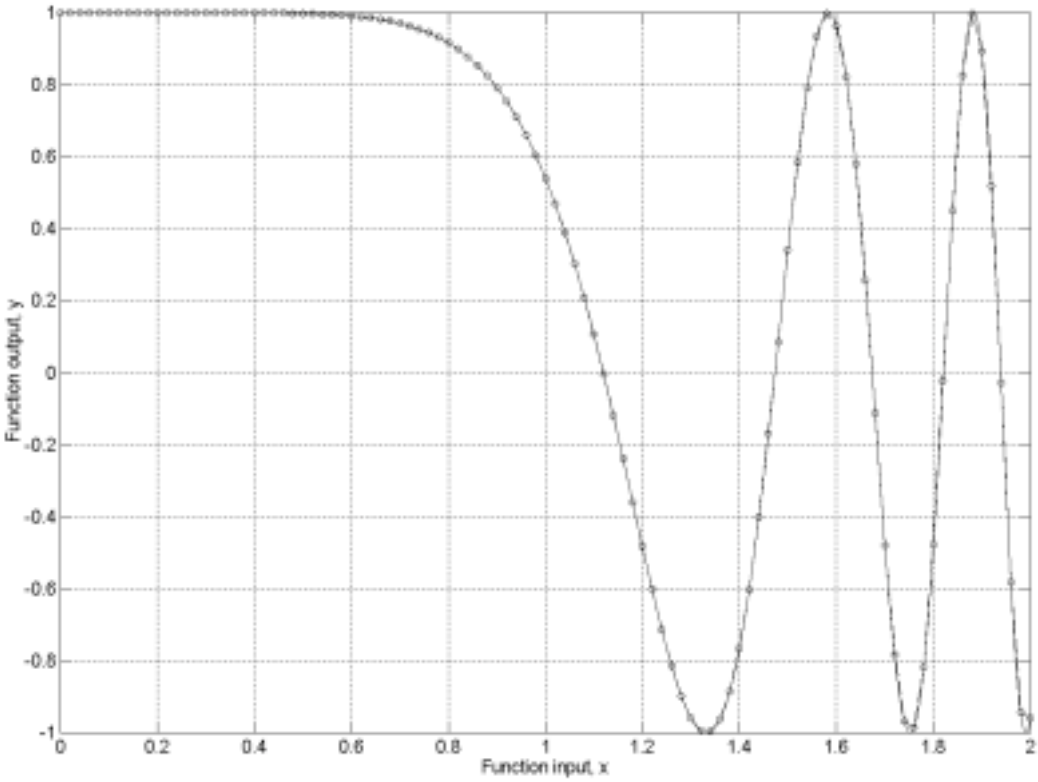
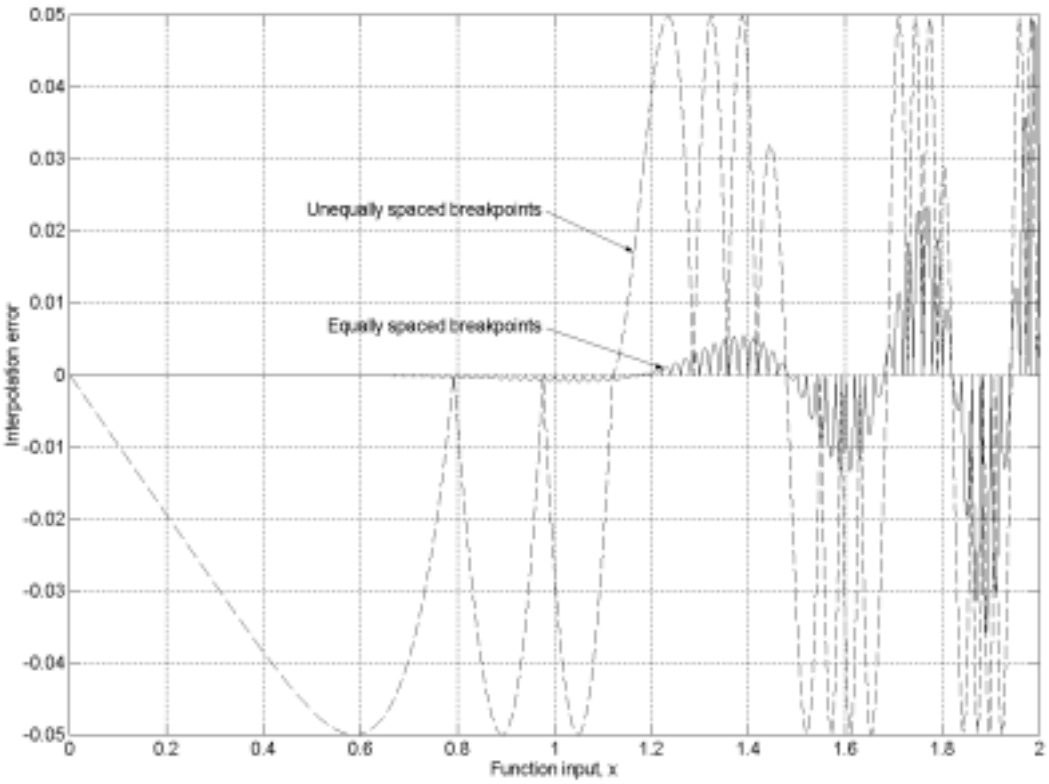


Figure 2.9 shows the interpolation error for both the unequally-spaced breakpoints of Figure 2.7 and the equally-spaced breakpoints of Figure 2.8. The error is distributed relatively evenly along the *x* axis for the case of unequal breakpoint spacing. In the equally-spaced breakpoint case, the error is very small for the lower part of the *x* range and grows larger as the function fluctuates more rapidly.

Figure 2.9 Interpolation error in Figure 2.7 and 2.8.



This example shows that, when using table interpolation, we must give some consideration to the choice between equally-spaced breakpoints versus unequally-spaced breakpoints. The selection of the appropriate type of breakpoint spacing depends on

- the characteristics of the function or data to be interpolated,
- the time available for function evaluation, and
- the memory available for table storage.

If unequally-spaced breakpoints are used, the locations of the breakpoints must be selected carefully to minimize the number of breakpoints required while simultaneously limiting the magnitude of the interpolation error. When equally-spaced breakpoints are used, one must choose the breakpoint interval to limit the maximum interpolation error to an acceptable value.

Cubic Spline Interpolation

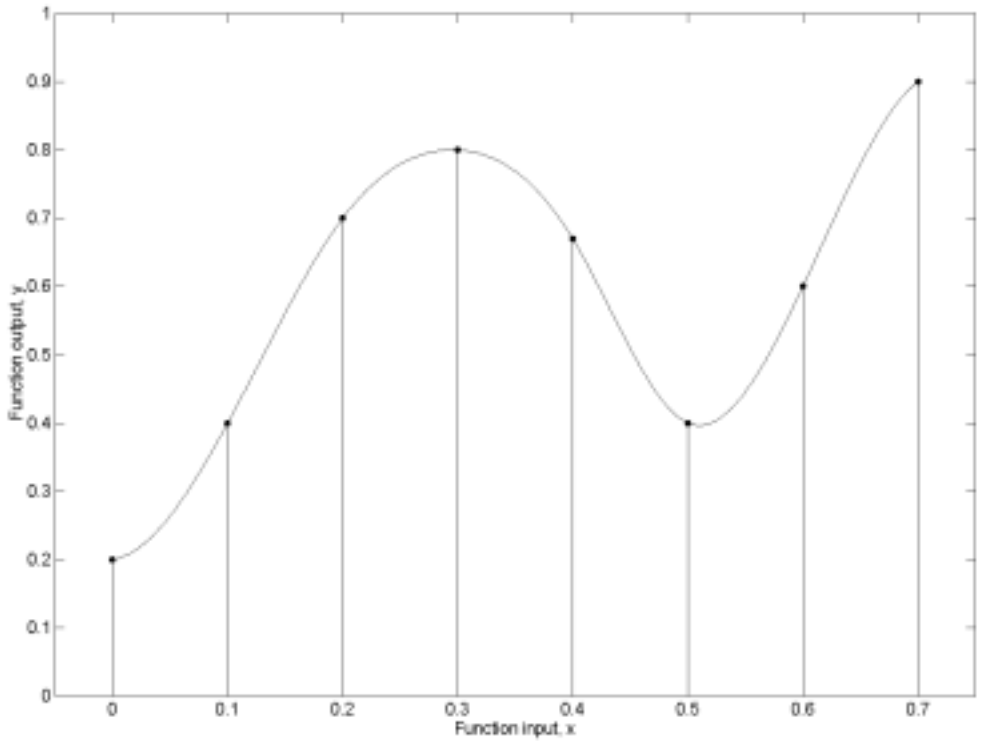
A drawback of the linear interpolation method is that the derivative of the evaluated function is discontinuous at the breakpoints, as can be seen in Figure 2.6 (page 39). If a smoother interpolation function is required, cubic spline interpolation is an appropriate choice. Cubic

spline interpolation uses a third-order polynomial to estimate the function value between breakpoints. Using this method, the estimating function, as well is its first and second derivatives, are continuous both between and at the breakpoints. In a sense, the cubic spline gives the smoothest interpolation possible through the breakpoints defining the function.

The costs of using cubic spline interpolation rather than linear interpolation are an increase in execution time and an increase in data memory required for a given number of breakpoints — as well as a significantly more complex algorithm. Each breakpoint interval requires the determination of four coefficients for the third-order interpolating polynomial, and this polynomial must be evaluated to determine the function output.

If the function to be approximated is somewhat smooth, it may be possible to use fewer breakpoints with cubic spline interpolation than would be needed with linear interpolation. This may mitigate the additional storage space required for the polynomial coefficients. Figure 2.10 shows the same set of interpolation breakpoints as Figure 2.5 with cubic spline interpolation used to estimate the function value between the breakpoints.

Figure 2.10 Cubic spline breakpoint interpolation.



A simulation application of this algorithm should be broken into two steps. A preprocessing step during initialization determines the polynomial coefficients for each breakpoint interval. The resulting coefficient values are stored for use during simulation execution. During the simulation run, the function evaluation is carried out by first locating the breakpoint interval containing the input value (either equally-spaced or unequally-spaced breakpoints can be

used) and evaluating the polynomial using the stored coefficients. An example of an algorithm for efficiently performing cubic spline interpolation with unequally-spaced breakpoints appears in [5].

Multidimensional Table Interpolation

The table interpolation examples presented in the previous sections were for functions that had only one input variable. The discussion now turns to interpolation methods for functions that have multiple inputs.

To define a multiple-input interpolation function, each input variable must have a set of breakpoints (either equally-spaced or unequally-spaced) associated with it. A function may have equally-spaced breakpoints for some input variables and unequally-spaced breakpoints for others. For each input, the breakpoint interval containing the current input must be located using the appropriate technique as described in "Linear Interpolation with Equally-Spaced Breakpoints" on page 38 and the following section "Linear Interpolation with Unequally-Spaced Breakpoints" on page 40. Using these breakpoints, a multidimensional interpolation must then be performed. It is possible to use linear interpolation for some input variables and cubic spline or other interpolation methods for other inputs, if that is appropriate.

Here's an example of a two-input function with equally-spaced breakpoints and linear interpolation for both input variables. It is straightforward to extend this example to three or more dimensions. Equations 2.23, 2.24, and 2.25 list the x and y axis breakpoint locations and the function values z at those locations. The table in Equation 2.25 is defined so that increasing values of x appear in the columns from left to right and increasing values of y are listed in the rows from top to bottom.

2.23 $x=[3\ 4\ 5\ 6]$

2.24 $y=[1.2\ 1.4\ 1.6\ 1.8]$

2.25 $$z = \begin{bmatrix} 0 & 0.1 & 0.3 & 0.3 \\ 0.1 & 0.3 & 0.5 & 0.4 \\ 0.1 & 0.5 & 0.6 & 0.7 \\ 0.2 & 0.5 & 0.6 & 0.9 \end{bmatrix}$$

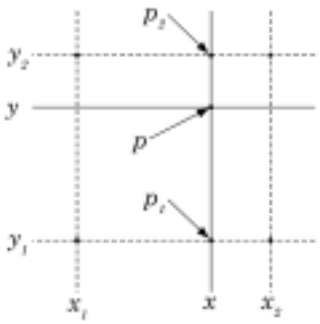
We will evaluate this function using linear interpolation. Assume that the correct breakpoint interval for each function input has already been located using the techniques mentioned previously. This example uses equally-spaced breakpoints, but the steps are identical for unequally-spaced breakpoints once the correct breakpoint interval has been determined for each input.

In Figure 2.11, the function inputs x and y define the point p, where we wish to evaluate the function output z. The four points defined by the intersection of the lines $x = x_1$, $x = x_2$, $y = y_1$, and $y = y_2$ represent the breakpoints that surround p (as defined in Equations 2.23, 2.24, and 2.25).

Two-dimensional interpolation must be performed in two steps.

1. The function z value is computed at the points p_1 and p_2 by performing linear interpolation along the x dimension.
2. The function z value at the point p is computed by performing linear interpolation between p_1 and p_2 along the y dimension.

Figure 2.11 Two-dimensional linear interpolation.



As a numerical example, let the input (x, y) pair be (4.8, 1.55). Consulting Equations 2.23, 2.24, and 2.25, observe that $x_1 = 4$, $x_2 = 5$, $y_1 = 1.4$, $y_2 = 1.6$, $z(x_1, y_1) = 0.3$, $z(x_2, y_1) = 0.5$, $z(x_1, y_2) = 0.5$, and $z(x_2, y_2) = 0.6$.

First, compute the interpolated function values at p_1 and p_2 using the formulas shown in Equation 2.26. Numerical results for this example are shown in Equation 2.27.

$$2.26 \quad z(x,y_1) = z(x_1,y_1) + [z(x_2,y_1) - z(x_1,y_1)]\frac{x - x_1}{x_2 - x_2}$$

$$z(x,y_2) = z(x_1,y_2) + [z(x_2,y_2) - z(x_1,y_2)]\frac{x - x_1}{x_2 - x_1}$$

$$2.27 \quad z(x,y_1) = z(4.8,1.4) = 0.3 + [0.5 - 0.3]\frac{4.8 - 4}{5 - 4} = 0.46$$

$$z(x,y_2) = z(4.8,1.6) = 0.5 + [0.6 - 0.5]\frac{4.8 - 4}{5 - 4} = 0.58$$

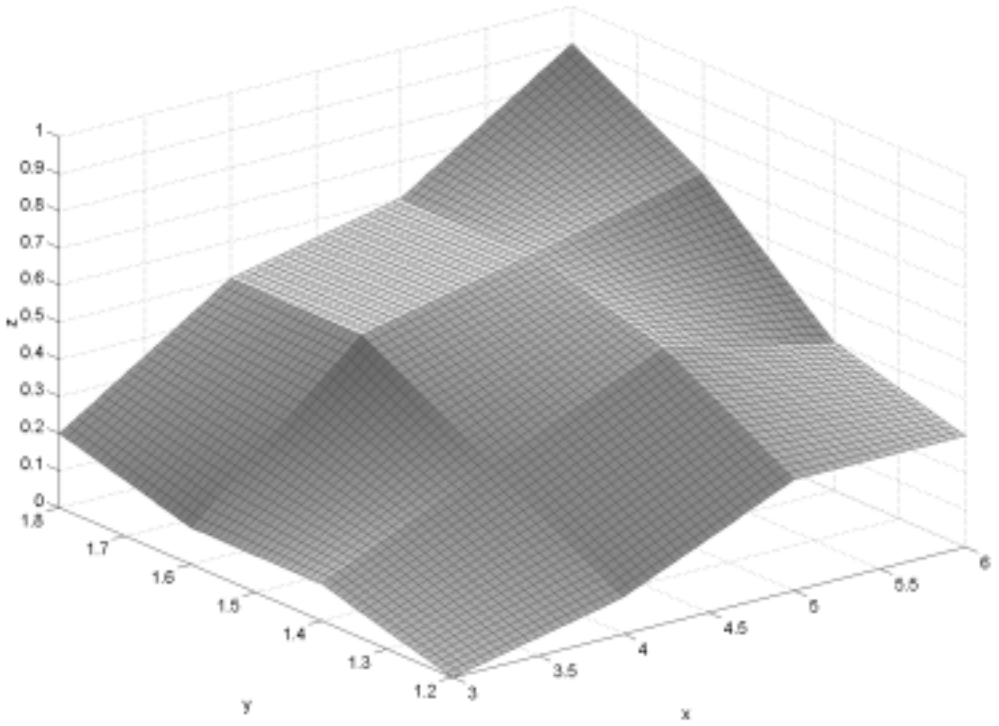
Finally, perform linear interpolation between the two function values computed in Equation 2.26 along the y dimension as shown in Equation 2.28. The result is the interpolated value of the function output at the point p, with the numerical result for this example shown in Equation 2.29.

$$2.28 \quad z(x,y) = z(x,y_1) + [z(x,y_2) - z(x,y_1)]\frac{y - y_1}{y_2 - y_1}$$

$$2.29 \quad z(x,y) = 0.46 + [0.58 - 0.46]\frac{1.55 - 1.4}{1.6 - 1.4} = 0.55$$

A plot of the function defined by Equations 2.23, 2.24, and 2.25 using linear interpolation appears in Figure 2.12. Note that the between-breakpoint surfaces resulting from the linear interpolation will not generally be flat. The only time an interpolation surface will be flat is when the four points at the corners of the surface all happen to lie in the same plane.

Figure 2.12 Two-dimensional function evaluation using linear interpolation.

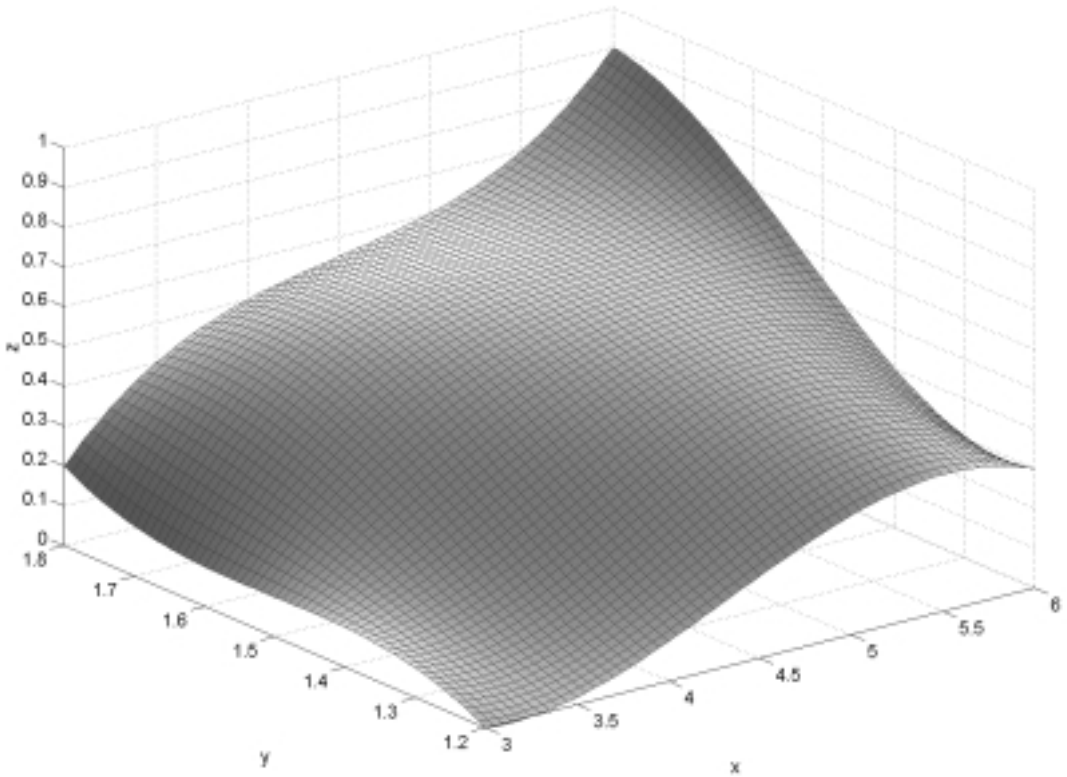


The function of Equations 2.23, 2.24, and 2.25 has been plotted using cubic spline interpolation in Figure 2.13. The resulting interpolated function is very smooth and passes through all the *z* values defined by the table of breakpoints. However, substantially more computation is required for multidimensional cubic spline interpolation. As was noted in the one-dimensional case, it may be possible to use fewer breakpoints if the interpolated function is smooth and maps well to the approximating polynomials used in cubic spline interpolation. The selection of the appropriate interpolation method depends on the requirements of the application.

It is possible to extend multidimensional interpolation to use any number of input variables. For example, with three inputs, linear interpolation is performed across a three dimensional box using the eight breakpoint values located at the box corners. In general, an

N-dimensional linear interpolation will require $2^N - 1$ one-dimensional interpolations to compute an output.

Figure 2.13 Two-dimensional function evaluation using cubic spline interpolation.



Linear function interpolation is a common tool in the simulation of complex dynamic systems. The selection of equally-spaced versus unequally-spaced breakpoints is a tradeoff between data table size and speed of function evaluation that depends on the characteristics of the data that represent the function. Linear interpolation is the standard method used with these tables due to the speed and simplicity of function evaluation. More complex interpolation techniques, such as the cubic spline method discussed here, are applicable when the requirements of a particular application dictate the use of a smoother approximating function.

Linear Interpolation with the DSSL

The DSSL library can perform linear interpolation for functions with any number of dimensions. Each function input can use either equally-spaced or unequally-spaced breakpoints. Equally-spaced breakpoints are the most computationally efficient. The unequally-spaced

breakpoint algorithm employs the technique of first checking to see if the function input falls within the previous breakpoint interval, followed by testing the two surrounding intervals before resorting to bisection. Because of this, the use of unequally-spaced breakpoints may *not* result in significantly worse performance if the input function changes in sufficiently small steps between function interpolations.

Listing 2.2 shows a program that performs two-dimensional table interpolation with equally-spaced breakpoints along both axes. The output file created by this program was plotted in MATLAB to produce Figure 2.12.

Listing 2.2 InterpTest.cpp

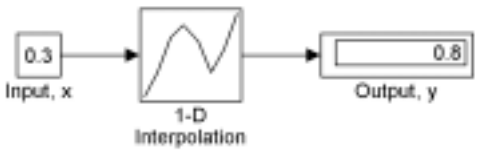
```
// Two dimensional interpolation

#include <dssl.h>

#include <cstdio>
#include <cassert>

int main()
{
    // Define the equally-spaced breakpoints. The first argument is the
    // first breakpoint value; the second is the breakpoint separation.
    EqSpacedBkpt<4> x_bkpt(3, 1), y_bkpt(1.2, 0.2);

    // Define a two-dimensional interpolation function with the above bkpts
    LinearInterp<2> interp;
    interp.SetDimension(0, &x_bkpt);
    interp.SetDimension(1, &y_bkpt);

    const double data[] =
    {
        0.0, 0.1, 0.3, 0.3,
        0.1, 0.3, 0.5, 0.4,
        0.1, 0.5, 0.6, 0.7,
        0.2, 0.5, 0.6, 0.9
    };

    interp.SetupData(data);

    // Open a file for the output data
    FILE* iov = fopen("z.txt", "w");
```

```
    assert(iov);

    // Interpolate the function across the ranges of the X and Y breakpoints
    // and write the results to the output file
    for (int ix=0; ix<=60; ix++)
    {
        double x = 3 + 0.05 * ix; // Function X input
        x_bkpt(x);
        for (int iy=0; iy<=60; iy++)
        {
            double y = 1.2 + 0.01 * iy; // Function Y input
            y_bkpt(y);

            double z = interp(); // Function interpolation result
            fprintf(iov, " %lf", z);
        }

        fprintf(iov, "\n");
    }

    fclose(iov);
    return 0;
}
```

Linear Interpolation in Simulink

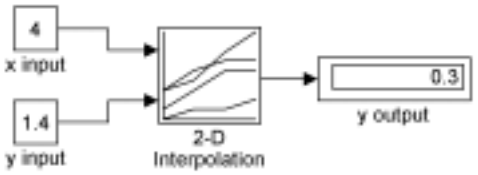
Simulink provides a one-dimensional table interpolation block as shown in Figure 2.14. This block takes two equal-length one-dimensional arrays as parameters: a monotonically increasing vector of *x* axis coordinates (which may be equally or unequally-spaced) and a vector of corresponding *y* values. If the block input is outside the range of the *x* array, the block performs extrapolation using the first or last two breakpoints. This block displays a graph of the function (compare to Figure 2.6).

Figure 2.14 Simulink one-dimensional interpolation.



Simulink also has a two-dimensional interpolation block, shown in Figure 2.15. The parameters for this block are monotonically increasing arrays for the *x* and *y* axis breakpoints and the two-dimensional table of *z* values. The table must have the same number of columns as the number of *x* axis breakpoints and the same number of rows as the number of *y* axis breakpoints. If either input is outside the range of the corresponding set of breakpoints, the block extrapolates along that dimension. This block displays a graph of the table, with each table column drawn as a line.

Figure 2.15 Simulink two-dimensional interpolation.



Simulink v4 provides additional blocks for performing interpolation along more than two dimensions. These newly-added blocks permit any number of function inputs and they support linear interpolation as well as cubic spline interpolation. The blocks allow the developer to select whether an out-of-range function input should result in limiting it to the valid range, extrapolating it, or halting the simulation with an error message.

System Identification

Another technique for developing models of dynamic systems is *system identification* [3]. To perform system identification, one or more test input data sequences and the measured output data sequences are required for the system being modeled. Typically, tests of a real world system must be designed and executed to generate this data. By applying a variety of system identification algorithms, it is possible to derive an estimate of the system transfer function from input to output. The resulting model is typically linear and time-invariant, so the developer must verify that this is an adequate representation of the system.

This book will not examine the details of algorithms for performing system identification. Simply note that, as was shown in “The s-plane Format” on page 23, it is a straightforward procedure to convert an *s*-domain transfer function resulting from system identification into an equivalent set of first-order differential equations suitable for implementation in a simulation.

The model resulting from system identification is a dynamic model, while table interpolation methods are static function evaluation techniques. System identification and table interpolation methods are similar in that they are based on the use of measured data rather than an assumed set of mathematical relations as was the case in physics-based modeling.

Neural Networks

Neural networks [6] provide a method for developing models from data using a method that is conceptually similar to system identification, but with a fundamentally different mathematical approach. A neural network is based on simple mathematical models of biological neurons — the fundamental cognitive units of the brain. This technique can be used to model highly nonlinear systems and phenomena that do not have an associated physics-based model.

A neural network functions as a static model, meaning that it processes a set of inputs to produce an output value during each evaluation. The actual processing occurs within the neurons. The neurons are interconnected through links, each of which has a weight associated with it. A weight controls the strength of the signal transmitted over its link. Each neuron has a set of these weighted signals connected to its input, which are summed and applied to the neuron activation function. The activation function determines the output signal of the neuron.

The "programming" of a neural network is contained in the connection pattern of the neurons, the type of activation function used within each neuron, and the learning algorithm used to set the connection weights between the neurons. The connection weights are set by an iterative procedure called *training* which applies a set of inputs to the network, evaluates the network output using the current weights, and compares the network output to the expected output. This comparison generates an error signal, which the learning algorithm uses to adjust the network connection weights to reduce the error. Training a neural network to perform a useful task may require applying thousands of training examples until the error for each example has been reduced to an acceptable level.

The training of a neural network is often a lengthy procedure, but once that step has been completed, the resulting model is fast and efficient. A simulation developer should consider the use of neural networks in simulation modeling when a complex, nonlinear system is to be modeled and the techniques presented earlier in this chapter cannot be used to develop a suitable model. One requirement for neural network model development is that a sufficiently large set of input-output training examples must be available to perform the training procedure.

2.5 Rigid Body Motion in Three-Dimensional Space

A model that describes the translational and rotational motion of one or more objects must express the motion of those objects with respect to a set of coordinate systems. It must be possible to determine the position, velocity, angular orientation, and rotational rate of each object in the different coordinate systems to perform system simulation. For example, when Newton's Law of motion is applied to determine the translational motion of an object, you must determine the direction of forces and moments acting on the object with respect to inertial (nonrotating and nonaccelerating) space. This may be difficult, because the forces and moments acting on the body are frequently defined in terms of a body-fixed coordinate system that accelerates and rotates with the object.

In an aircraft flight simulation, there are typically two primary coordinate systems [7]. One coordinate system is fixed to the body of the aircraft and moves with it in both translation and rotation. This coordinate system has its origin at the aircraft center of mass and is called the body-fixed coordinate system. The other originates on the surface of the earth at a

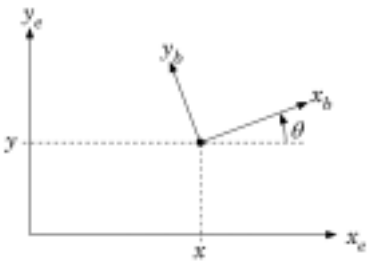
defined location and is called the earth-fixed coordinate system. The relations between these coordinate systems define the aircraft position, velocity, *Euler angles* (roll, pitch, and yaw angles), and rotation rates.

In many cases, the effects of the rotation of the earth and its curvature are negligible, which allows the earth to be modeled as flat and nonrotating. Under these assumptions, the earth-fixed coordinate system is an inertial system, which allows the direct application of Newton's Law of motion. If the rotation of the earth is not negligible for a particular application, a more complex model defines an inertial coordinate system with its origin at the center of the earth. In this model, the earth-fixed coordinate system also has its origin at the earth center and rotates with respect to the inertial coordinate system. The motion of the aircraft is defined relative to the rotating earth coordinate system, and the equations of motion must be applied with reference to the inertial coordinate system. Although this level of modeling detail is sometimes necessary, we will not consider such complex cases.

2.5.1 Two-Dimensional Motion

Let's first examine the relatively simple case of a body that moves in a two-dimensional plane and rotates about the axis perpendicular to the plane. Figure 2.16 shows the relationship between the body-fixed coordinate system (x_b, y_b) and the (assumed inertial) earth-fixed coordinate system (x_e, y_e).

Figure 2.16 Coordinate systems for two dimensional motion.



If we assume that the body mass m and its rotational inertia I are constant, the differential equations that describe the translational and rotational motion of the body in terms of the forces and the rotational moment in the body-fixed axes are shown in Equation 2.30. In these equations, F_x and F_y are the instantaneous forces acting on the body along the x_b and y_b axes, and M is the instantaneous rotational moment acting in the direction of the angle θ as shown in Figure 2.16. Equation 2.30 describes the position and orientation of the body in inertial space given the forces and moment acting on it in body-fixed coordinates at each instant in time. To completely specify the motion of the body, Equation 2.30 requires the forces F_x and F_y and the moment M as functions of time over the integration interval, as well as six initial conditions specifying $x(0)$, $x'(0)$, $y(0)$, $y'(0)$, $\theta(0)$, and $\theta'(0)$.

2.30
$$x'' = \frac{[F_x \cos\theta - F_y \sin\theta]}{m}$$
$$y'' = \frac{[F_x \sin\theta + F_y \cos\theta]}{m}$$
$$\theta'' = \frac{M}{I}$$

These equations of motion can be represented more clearly with the use of vector-matrix notation. The differential equations in Equation 2.30 can be rewritten as shown in Equation 2.31, where the position vector $\mathbf{P} = [x\ y]^T$ and the force vector $\mathbf{F} = [F_x\ F_y]^T$. The superscript T indicates matrix or vector transposition.

2.31
$$P'' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \frac{\mathbf{F}}{m} = \mathbf{C}(\theta)^T \frac{\mathbf{F}}{m}$$

The matrix $\mathbf{C}(\theta)$ in Equation 2.31 is called a *direction cosine matrix*. It is an orthonormal matrix, which means that

$$\mathbf{C}(\theta)^{-1} = \mathbf{C}(\theta)^T$$

for any value of θ. In other words, the matrix inverse of $\mathbf{C}(\theta)$ is equal to its transpose.

When a vector in earth-fixed coordinates is premultiplied by $\mathbf{C}(\theta)$, the result will be a vector in body-fixed coordinates, except for the difference in origin location. Using the orthonormality of $\mathbf{C}(\theta)$, you can transform a vector in body-fixed coordinates to earth-fixed coordinates (except for the difference in origin location) by premultiplying it by

$$\mathbf{C}(\theta)^T.$$

These relationships are shown in Equation 2.32, where $\mathbf{X}_b$ is an arbitrary vector in body-fixed coordinates and $\mathbf{X}_e$ is the equivalent (except for the difference in origin location) vector in earth-fixed coordinates.

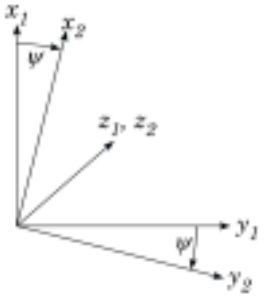
2.32
$$\mathbf{X}_b = \mathbf{C}(\theta)\mathbf{X}_e$$
$$\mathbf{X}_e = \mathbf{C}(\theta)^T \mathbf{X}_b$$

2.5.2 Three-Dimensional Motion

When the motion of a body extends to three degrees of freedom in both translation and rotation, the dynamic equations become somewhat more complicated. However, the additional effort is worthwhile because this method of simulating the motion of a body accurately models the translational and rotational motion of rigid bodies in three-dimensional space. This approach is called "six degrees of freedom motion simulation," often shortened to "6DOF" simulation. I will present two methods for solving the 6DOF rotational equations of motion in a simulation, but first I'll clarify some issues regarding the coordinate systems.

The coordinate systems used here will always be orthogonal, meaning that the three coordinate axes are at right angles to each other. Our coordinate systems will also be right-handed, which means that an angular rotation has a positive sign when it occurs in a clockwise direction as viewed along the positive direction of the axis of rotation. In addition, in a right-handed coordinate system the *z* axis will be in the direction of the vector cross product between the *x* and *y* axes. In terms of unit-length vectors along each axis, $\mathbf{u}_z = \mathbf{u}_x \times \mathbf{u}_y$ in a right-handed coordinate system. Figure 2.17 shows a positive rotation through the angle ψ from the (x_1, y_1, z_1) coordinate system to the (x_2, y_2, z_2) coordinate system, where the z_1 and z_2 axes are identical. Both of these coordinate systems are right-handed and orthogonal.

Figure 2.17 Positive rotation about the *z* axis.



Next, we will examine the transformation of a vector in a given initial coordinate system through angular rotations about the three axes in a particular sequence. By performing these three rotations, we can place the transformed coordinate system in any desired orientation. The steps are: rotate about the z_1 axis, then about the y_2 axis, and finish with a rotation about the x_3 axis.

In matrix-vector form, the transformation of a vector from the (x_1, y_1, z_1) coordinate system to the (x_2, y_2, z_2) coordinate system is as shown in Equation 2.33.

$$2.33 \qquad \mathbf{X}_2 = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{X}_1 = \mathbf{C}_z(\psi)\mathbf{X}_1$$

In Equation 2.33, the vector X_1 is an arbitrary vector in the (x_1, y_1, z_1)coordinate system, X_2 is the same vector in the (x_2, y_2, z_2) coordinate system, and $\mathbf{C}_z(\psi)$ is the direction cosine matrix that performs this coordinate transformation. The subscript *z* indicates the axis of rotation and the parameter ψ indicates the angle of rotation. Similarly, the next rotation through the angle θ about the y_2 axis is shown in Equation 2.34. The result of this transformation is the vector $\mathbf{X}_3$.

$$2.34 \qquad \mathbf{X}_3 = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \mathbf{X}_2 = \mathbf{C}_y(\theta)\mathbf{X}_2$$

The final rotation through the angle ϕ about the x_3 axis is shown in Equation 2.35. The vector $\mathbf{X}_4$ is the result after the full three axis coordinate transformation.

$$2.35 \quad \mathbf{X}_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \mathbf{X}_3 = \mathbf{C}_x(\phi)\mathbf{X}_3$$

The complete transformation from the (x_1, y_1, z_1) coordinate system to the (x_4, y_4, z_4) coordinate system appears in Equation 2.36, where the three single axis direction cosine matrices are applied in sequence using matrix multiplication.

$$2.36 \quad \mathbf{X}_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{X}_1 = \mathbf{C}_x(\phi)\mathbf{C}_y(\theta)\mathbf{C}_z(\psi)\mathbf{X}_1$$

Equation 2.37 shows the result of multiplying out the matrices in Equation 2.36. We will call this matrix the $\mathbf{C}_{zyx}(\psi, \theta, \phi)$ matrix to indicate the sequence of axes used for rotations and the angle of rotation about each axis. Note that matrix multiplication is not generally commutative, so changing the order of the axes for the rotations produces a different (and incorrect) result. Because of this, it is critical to perform the axis rotations in the correct order.

$$2.37 \quad \mathbf{X}_4 = \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \sin\phi\cos\theta \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\cos\theta \end{bmatrix} \mathbf{X}_1$$

$$= \mathbf{C}_{zyx}(\psi, \theta, \phi)\mathbf{X}_1$$

The orthonormality of the direction cosine matrix is maintained through the complete three axis rotation. Because of this, the direction cosine matrix for performing the reverse transform from the $\mathbf{X}_4$ vector to the $\mathbf{X}_1$ vector is the transpose of the $\mathbf{C}_{zyx}(\psi, \theta, \phi)$ matrix, which produces the $\mathbf{C}_{xyz}(-\phi, -\theta, -\psi)$ matrix. This matrix performs angular rotations of the opposite sign in the reverse axis order in comparison to the $\mathbf{C}_{zyx}(\psi, \theta, \phi)$ matrix.

When simulating the motion of a rigid body, we sometimes wish to transform vectors from earth-fixed coordinates to body-fixed coordinates. Define the vector $\mathbf{X}_1$ to be in earth-fixed coordinates and identify it as $\mathbf{X}_e$. The equivalent vector $\mathbf{X}_4$ in body-fixed coordinates will be called $\mathbf{X}_b$. The relations between these vectors are shown in Equation 2.38.

$$2.38 \quad \mathbf{X}_b = \mathbf{C}_{zyx}(\psi, \theta, \phi)\mathbf{X}_e$$

$$\mathbf{X}_e = \mathbf{C}_{zyx}(\psi, \theta, \phi)^T \mathbf{X}_b = \mathbf{C}_{xyz}(-\phi, -\theta, -\psi)\mathbf{X}_b$$

Equation 2.38 demonstrates how to transform vectors from earth-fixed coordinates to body-fixed coordinates and vice versa. We can use these relations to develop the 6DOF translational equation of motion. Define the instantaneous force acting on the body center of mass in body-fixed coordinates to be the vector

$$\mathbf{F}_b = \left[F_{b_x}\ F_{b_y}\ F_{b_z}\right]^T.$$

Equation 2.39 is the translational equation of motion, where $\mathbf{P}_e$ is the position of the body in earth-fixed coordinates and m is the instantaneous mass of the body. Using the methods described in Chapter 3, we integrate Equation 2.39 numerically to determine the body position and velocity over time.

$$2.39 \quad \mathbf{P}''_e = \mathbf{C}_{zyx}(\psi, \theta, \phi)^T$$

Equation 2.39 requires the three angles ϕ, θ, and ψ which are the roll, pitch, and yaw Euler angles. To determine these angles for a body that is undergoing angular accelerations, we must solve the rotational equation of motion. Let's now define this equation and examine approaches for solving it numerically.

The first step in the solution of the rotational equation of motion is to determine the angular rates of the body given the moments acting on it about its center of mass. This solution depends on the moments of inertia and the products of inertia for the body, which are defined in Equation 2.40 — where dm represents a differential element of the body mass at the location (x, y, z) in body-fixed coordinates. These integrations must be performed over the entire body mass. Here, as is often the case, one may assume that the moments and products of inertia of the body are constant — in other words, that the body is rigid.

$$2.40 \quad I_{xx} = \int_m (y^2 + z^2)\,dm$$

$$I_{yy} = \int_m (x^2 + z^2)\,dm$$

$$I_{zz} = \int_m (x^2 + y^2)\,dm$$

$$I_{xy} = \int_m xy\,dm$$

$$I_{xz} = \int_m xz\,dm$$

$$I_{yz} = \int_m yz\,dm$$

Next, place the moments and products of inertia into a matrix called the inertia tensor shown in Equation 2.41.

$$2.41 \quad \mathbf{I} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}$$

The differential equation that defines the angular rates in body-fixed coordinates is shown in Equation 2.42, where $\Omega = [p\ q\ r]^T$ is the rotation rate vector and $\mathbf{M} = [m_x\ m_y\ m_z]^T$ is the moment vector acting about the body's center of mass, all in body-fixed coordinates.

$$2.42 \quad \dot{\Omega} = \mathbf{I}^{-1}(\mathbf{M} - \Omega \times (\mathbf{I}\Omega))$$

Equation 2.42 is integrated numerically to compute the body rotation rate vector over time. The initial condition associated with this equation is the body rotation rate at the start of the simulation run.

Now that we know the body rotation rate, the next step is to determine the angular orientation resulting from these rotation rates, usually as the *Euler angles* ϕ, θ, and ψ. There are two commonly used techniques for performing this computation in 6DOF simulations: Euler angle integration and quaternion integration. Euler angle integration is conceptually simpler, but it will run into numerical problems if the pitch angle θ approaches 90 degrees in magnitude where a singularity occurs in the equations. *Quaternion* integration is more mathematically complex, but this approach does not have any difficulty when θ passes through 90 degrees in magnitude.

Euler Angle Integration

Assume that the body rotation rate vector Ω is available from the solution of Equation 2.42. Then, relate the time derivatives of the Euler angles to the body-fixed rotation rates using Equation 2.43 [7]. The initial conditions associated with these equations are the initial Euler angles of the body.

$$2.43 \quad \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{bmatrix} \Omega$$

We can integrate Equation 2.43 numerically during simulation which — along with Equation 2.42 — gives the complete solution for the rotational motion of the body given the moments acting about its center of gravity in body-fixed coordinates. Using these results, we compute the $\mathbf{C}_{zyx}(\psi, \theta, \phi)^T$ matrix as shown in Equation 2.37 (page 57) and use it in the solution of the translational equation of motion shown in Equation 2.39.

The numerical difficulty in Equation 2.43 occurs when the angle θ approaches 90 degrees in magnitude because the magnitude of the $\sec\theta$ terms in the last row of the matrix approaches infinity. If the simulation application will never have θ approach ±90 degrees, Equation 2.43 is appropriate for the solution of the equations of rotational motion. An example where this assumption may be valid is in the flight simulation of a transport aircraft, where θ would never be expected to exceed, say, 40 degrees in magnitude.

Quaternion Integration

On the other hand, if any arbitrary value of θ must be accommodated in the simulation, it is necessary to use an alternative approach to determine the body orientation relative to earth-fixed axes. The use of quaternion integration is the preferred technique in this situation. A quaternion is a four-element vector

$$\mathbf{b} = \left[b_1\ b_2\ b_3\ b_4\right]^T$$

that can be thought of as a four-component complex number. Use a quaternion to maintain the relationship between the body-fixed and earth-fixed coordinate systems.

The values of the quaternion elements must be initialized from the initial values of the body Euler angles. The initial earth-fixed to body-fixed direction cosine matrix $\mathbf{C}_{zyx}(\psi,\theta,\phi)$ is computed as shown in Equation 2.37. In this section, call this initial direction cosine matrix $\mathbf{C}$ and select individual elements from it with the notation $\mathbf{C}_{rc}$ where r and c identify the row and column of a particular matrix element. Using this notation, $\mathbf{C}_{11}$ is the first element in the first row.

Initialize the elements of the quaternion. First, initialize the last quaternion element as shown in Equation 2.44.

$$2.44 \quad b_4 = \frac{1}{2}\sqrt{1 + \mathbf{C}_{11} + \mathbf{C}_{22} + \mathbf{C}_{33}}$$

Then, initialize the remaining quaternion elements as shown in Equation 2.45.

$$2.45 \quad b_1 = \frac{1}{4b_4}(\mathbf{C}_{12} - \mathbf{C}_{21})$$

$$b_2 = \frac{1}{4b_4}(\mathbf{C}_{31} - \mathbf{C}_{13})$$

$$b_3 = \frac{1}{4b_4}(\mathbf{C}_{23} - \mathbf{C}_{32})$$

There is a potential problem if the computed value of b_4 happens to be zero, which results in division by zero in Equation 2.45. The file `RigidBody.cpp` on the companion disk contains some variations on Equations 2.44 and 2.45 that accommodate this situation and correctly initialize the quaternion from any arbitrary initial direction cosine matrix.

The differential equation that relates the change in the quaternion parameters to the rotation rate in body-fixed axes is shown in Equation 2.46, where

$$\Omega = [pqr]^T$$

as before.

$$2.46 \quad \dot{\mathbf{b}} = \frac{1}{2}\begin{bmatrix} -b_4 & -b_3 & -b_2 \\ -b_3 & b_4 & b_1 \\ b_2 & b_1 & -b_4 \\ b_1 & -b_2 & b_3 \end{bmatrix}\Omega$$

To correctly model the relationship between the two coordinate systems, the magnitude of the quaternion vector must equal one. Floating point roundoff errors and integration errors accumulate over time and cause the magnitude of the vector to change slowly. To ensure accurate results, we must correct this error. As a first step in performing this correction, compute an error term as shown in Equation 2.47.

$$2.47 \quad e_b = 1 - \|b\|^2 = 1 - (b_1{}^2 + b_2{}^2 + b_3{}^2 + b_4{}^2)$$

Then use the error term as a correction to modify Equation 2.47 as shown in Equation 2.48. This equation is solved numerically to determine the quaternion vector during the simulation run.

$$2.48 \quad \dot{\mathbf{b}} = \frac{1}{2}\left\{\begin{bmatrix} -b_4 & -b_3 & -b_2 \\ -b_3 & b_4 & b_1 \\ b_2 & b_1 & -b_4 \\ b_1 & -b_2 & b_3 \end{bmatrix}\Omega + e_b\mathbf{b}\right\}$$

Given the current quaternion vector $\mathbf{b}$, compute the direction cosine matrix

$$\mathbf{C}_{zyx}(\psi, \theta, \phi)$$

using the following method [8]. Here, the identifier $\mathbf{C}$ will represent $\mathbf{C}_{zyx}(\psi, \theta, \phi)$ and we will use the same row and column subscript notation as before. The equations for computing the direction cosine matrix elements are shown in Equation 2.49.

$$2.49 \quad \mathbf{C}_{11} = b_1b_1 - b_2b_2 - b_3b_3 + b_4b_4$$

$$\mathbf{C}_{12} = 2(b_1b_2 + b_3b_4)$$

$$\mathbf{C}_{13} = 2(b_2b_4 - b_1b_3)$$

$$\mathbf{C}_{21} = 2(b_3b_4 - b_1b_2)$$

$$\mathbf{C}_{22} = b_1b_1 - b_2b_2 + b_3b_3 - b_4b_4$$

$$\mathbf{C}_{23} = 2(b_2b_3 + b_1b_4)$$

$$\mathbf{C}_{31} = 2(b_1b_3 + b_2b_4)$$

$$\mathbf{C}_{32} = 2(b_2 b_3 - b_1 b_4)$$

$$\mathbf{C}_{33} = b_1 b_1 + b_2 b_2 - b_3 b_3 - b_4 b_4$$

Compute the Euler angles from the elements of $\mathbf{C}$ as shown in Equation 2.50.

2.50 $$\phi = \arctan2[\mathbf{C}_{12}, \mathbf{C}_{11}]$$

$$\theta = \arctan2[-\mathbf{C}_{13}, \sqrt{\mathbf{C}^2_{23} + \mathbf{C}^2_{33}}]$$

$$\psi = \arctan2[\mathbf{C}_{23}, \mathbf{C}_{33}]$$

In summary, to determine the body orientation relative to earth-fixed axes using quaternions, first initialize the quaternion vector $\mathbf{b}$ from the initial direction cosine matrix using Equation 2.44 and Equation 2.45. Update the quaternion during simulation execution by numerically integrating Equation 2.48. Use the current state of the quaternion to compute the direction cosine matrix $\mathbf{C}_{zyx}(\psi, \theta, \phi)$ via Equation 2.49 and the three Euler angles using Equation 2.50. Finally, use the direction cosine matrix to integrate the translational equation of motion in Equation 2.39 (page 58).

Although the quaternion computation is more complicated than Euler angle integration, it is the preferred method for solving the rotational equations of motion if the numerical difficulty of Euler angle integration is a potential problem.

Three-Dimensional Motion Simulation with the DSSL

The DSSL C++ library provides routines that implement the equations given previously for simulating three-dimensional motion of rigid bodies. Listing 2.3 is a simulation of the motion of a projectile fired from a gun with a rifled barrel. For a short time while in the barrel, the projectile accelerates along the *x* body-fixed axis while simultaneously experiencing an angular acceleration that spins it about the *x* body-fixed axis. After the projectile leaves the barrel, no further acceleration is modeled.

Listing 2.3 RigidBodyTest.cpp

```
#include "RigidBody.h"

StateList state_list;
RigidBody body(&state_list);

int main()
{
    // All states will be initialized to zero
    Vector<3> pos_ic, vel_ic, euler_ic, body_rate_ic;
```

```
    body.Initialize(pos_ic, vel_ic, euler_ic, body_rate_ic);

    const double step_time = 0.001, end_time = 1.0;

    state_list.Initialize(step_time);

    printf("Time, Px, Py, Pz, Vx, Vy, Vz, Phi, Theta, Psi, P, Q, R\n");
    for(;;)
    {
        // Set all accelerations to zero for now
        Vector<3> translational_accel, angular_accel;

        if (state_list.Time() <= 0.01) // If still in the barrel, accelerate
        {
            translational_accel[0] = 10000.0;
            angular_accel[0] = 1000.0;
        }

        body.Compute(translational_accel, angular_accel);

        // Print state information
        printf("%lf", state_list.Time());
        for (int i=0; i<3; i++) printf(",%lf", body.GetPos()[i]);
        for (i=0; i<3; i++) printf(",%lf", body.GetVel()[i]);
        for (i=0; i<3; i++) printf(",%lf", body.GetEuler()[i]);
        for (i=0; i<3; i++) printf(",%lf", body.GetBodyRate()[i]);
        printf("\n");

        if (state_list.Time() >= end_time)
            break;

        state_list.Integrate();
    }

    return 0;
}
```

Three-Dimensional Motion Simulation in Simulink

Simulink v4 provides several new blocks for solving the rotational equation of motion using the quaternion integration and Euler angle integration techniques. Additional blocks perform conversions between a direction cosine matrix, a set of Euler angles, and a quaternion vector. These new blocks enable the implementation of a full 6DOF simulation with a reasonable amount of effort.

2.6 Stochastic Systems

A system that always responds identically to the same set of initial conditions and control input signals during multiple test runs is deterministic. If there is some variation in the system behavior from run to run — even though all initial conditions and control inputs remain the same — the system behavior is described as stochastic, or random. An example of stochastic behavior is an aircraft flying with a predefined sequence of control inputs. Wind gusts affect the flight path differently on each attempt, so although we model the aircraft deterministically, the wind model in this case is stochastic. If any part of a mathematical model is stochastic, we describe the model itself as stochastic.

In addition to external random disturbances such as wind gusts, there may be randomness associated with the system itself. Manufacturing tolerances may produce variations from unit to unit in parameters that affect the performance of the system. We model these random variations in the parameters of the system and its operational environment in a stochastic manner.

First, consider the case where a single number is required to specify a random effect that we wish to model. The example we will use is the misalignment of a system component due to manufacturing tolerances. The alignment is specified as an angle, α. To meet specifications, this angle must lie in the range from α_{min} to α_{max} and is equally likely to have any value in this range. Absolute limits on α are possible because we will assume that the system would not pass manufacturing tests if the angular limits were exceeded.

We use a *probability density function* (PDF) [9] to describe the *random variable* α. Figure 2.18 shows the PDF for α as described, where the angles $\alpha_{min} = 1.9$ degrees and $\alpha_{max} = 2.1$ degrees. This random variable is typically described in terms of a tolerance such as 2.0±0.1 degrees.

It is a property of any PDF that the function $f(\alpha)$ is nonnegative for all α. In this example, $f(\alpha) = 0$ for $\alpha > 1.9$ and for $\alpha > 2.1$ In addition, the integral of the PDF over the entire *x* axis always equals one as shown in Equation 2.51.

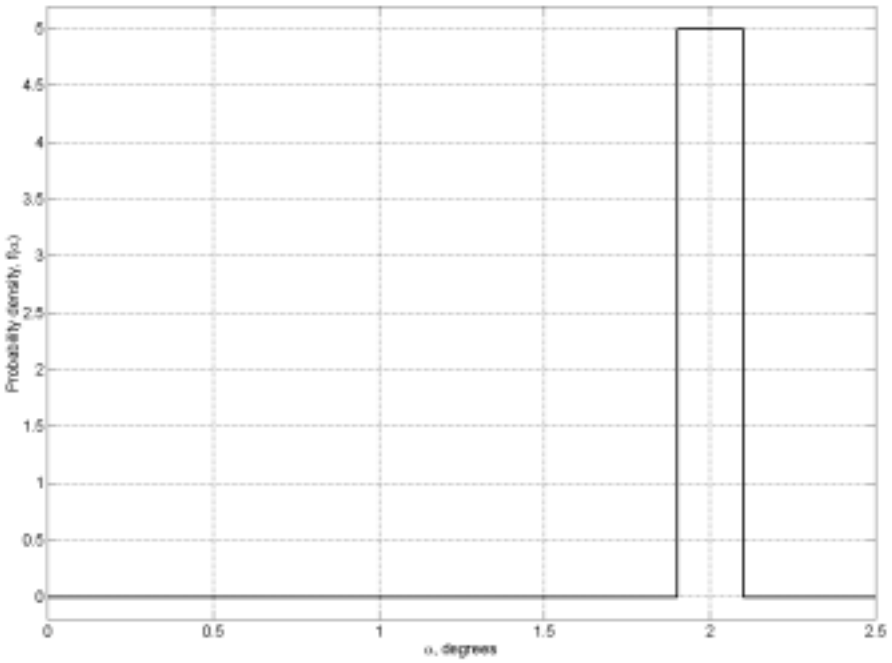
$$2.51 \quad \int_{-\infty}^{\infty} f(\alpha)d\alpha = 1$$

For any PDF, the probability that the angle α lies between two angles α_1 and α_2 (where $\alpha_2 \geq \alpha_1$) is shown in Equation 2.52.

$$2.52 \quad P(\alpha_1 \leq \alpha \leq \alpha_2) = \int_{\alpha_1}^{\alpha_2} f(\alpha)d\alpha$$

The uniform PDF is appropriate when all points over the possible range of values for a random variable appear to be equally likely. Programming languages and simulation development tools usually provide a uniform pseudorandom number generator that produces numbers over some range. A pseudorandom number generator that produces outputs over the range (0, 1) can be used as shown in Equation 2.53 to generate values from the distribution shown in Figure 2.18. In Equation 2.53, random() represents a call to the system random number generator routine that returns a value between 0 and 1.

Figure 2.18 Uniform probability density function.



2.53 $x = \alpha_{min} + (\alpha_{max} - \alpha_{min})\,\text{random}()$

One should always be cautious using system-supplied pseudorandom numbers because they are frequently of poor quality. One potential problem with these generators is that the first output value after the generator is seeded may not appear very random, i.e., it might always be a very small number. Another problem that sometimes occurs is that individual bits in the output may exhibit non-random behavior such as toggling between 0 and 1 with each call to the generator. It is a good idea to test a system pseudorandom number generator thoroughly before using it in a critical simulation application.

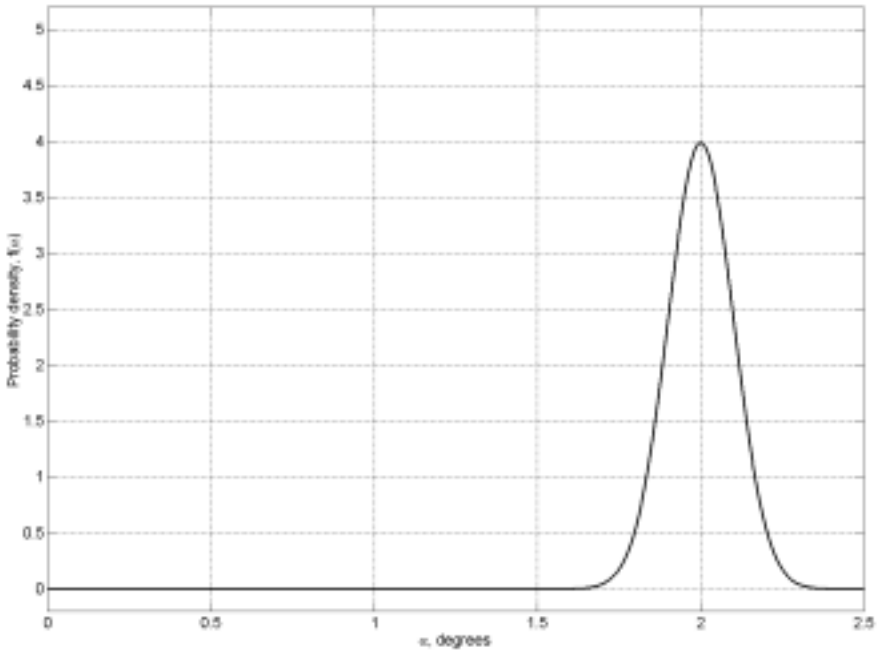
If the system-supplied pseudorandom number generator fails to satisfy your requirements, or if you have other needs such as cross-platform portability, you may want to develop your own uniform pseudorandom number generator routine. Some examples are provided in [10].

Another PDF that is commonly used is the normal (also known as Gaussian) distribution. This PDF is defined by Equation 2.54 where μ is the mean and σ is the standard deviation of the distribution. The normal PDF is used in situations where the total error is assumed to be the sum of a large number of independent errors.

$$2.54 \quad f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left[\frac{(x-\mu)^2}{2\sigma^2}\right]}$$

Figure 2.19 shows a normal distribution of a random variable with a mean μ of 2 and standard deviation σ of 0.1. Compare this distribution to the uniform distribution shown in Figure 2.18.

Figure 2.19 Normal probability density function.



Pseudorandom number generators with non-uniform PDFs such as the normal distribution are not available in some simulation development environments. If this is the case, it will be necessary to transform the output of a uniform pseudorandom number generator into the desired PDF. An efficient technique for performing the transformation from a uniform PDF to a normal PDF is the Box-Muller method [11].

So far, we have looked at techniques for generating single pseudorandom samples from a given PDF. A *random process* is a sequence of random variables over time. An example of a random process is the sequence of additive noise values that appear in the output of an analog-to-digital converter that performs conversions at regular time intervals.

A simple type of random process is an uncorrelated sequence that generates a new sample from the appropriate PDF at each time step. This model is useful for simulating the additive noise of the analog-to-digital converter.

A more complex random process involves the use of filtered noise, where, for example, the power spectrum of the noise is assumed to be uniform up to a cutoff frequency and zero at all higher frequencies. An example application of filtered noise is a model of the noise in the output of a communication receiver. This random process could be simulated by constructing a lowpass digital filter (perhaps designed with the Remez technique [12]) and using an uncorrelated random sequence as the filter input. The resulting filter output approximates the desired bandlimited noise spectrum.

When simulating a stochastic system, each simulation output affected by one or more random inputs will generate a probability distribution over a number of runs. The technique of *Monte Carlo simulation* is used to determine the probability distributions of simulation outputs. A Monte Carlo simulation consists of a large number of simulation runs performed under identical conditions, except that each run uses a *different* pseudorandom sequence for each random parameter or process in each run. The various sequences of pseudorandom numbers generate performance variations that involve combinations of random behaviors. The more simulation runs performed in a Monte Carlo set, the more accurate the probability distribution of the simulation outputs will be. However, the time available for performing simulation runs often limits the number of runs in Monte Carlo sets to less than a statistically ideal amount. The results of Monte Carlo testing that contain limited numbers of runs should be examined critically to ferret out anomalies resulting from the limited data set size.

Random Numbers in the DSSL

The example program in Listing 2.4 generates one million uniformly-distributed random numbers over the range (0,1) and counts how many of the numbers fall into each of one thousand equal-width bins. It then generates one hundred thousand normally-distributed random numbers with zero mean and unit variance and computes their sample mean and variance.

Listing 2.4 `RandomTest.cpp`

```
// Program for testing random numbers

#include <dssl.h>

int main()
{
    Random r;
    const int n_bin = 1000;
    int bin[n_bin];
    for (int i=0; i<n_bin; i++)
        bin[i] = 0;
```

```
    for (i=0; i<1000000; i++)
    {
        double val = r.Uniform();
        int j = int(val*n_bin);
        assert(0 <= j && j < n_bin);
        bin[j]++;
    }

    printf("Bin, Count\n");
    for (i=0; i<n_bin; i++)
        printf("%4d, %d\n", i, bin[i]);

    const int n_gauss = 100000;
    double g[n_gauss], sum = 0;
    for (i=0; i<n_gauss; i++)
    {
        g[i] = r.Normal();
        sum += g[i];
    }

    double mean = sum / n_gauss;

    double dev_sq = 0;
    for (i=0; i<n_gauss; i++)
        dev_sq += pow(g[i]-mean, 2);

    double sigma = sqrt(dev_sq/n_gauss);

    printf("Mean: %lf; Sigma: %lf\n", mean, sigma);

    return 0;
}
```

Random Numbers in Simulink

Simulink provides blocks for generating both uniformly- and normally-distributed pseudo-random numbers. When using the uniformly-distributed pseudorandom generator, the user must specify the minimum and maximum values of the output interval as well as the seed to use for the generator. When using the normally-distributed pseudorandom generator, the user must specify the distribution mean, variance, and the generator seed. For a given seed, each

generator will produce an identical sequence of pseudorandom values during each simulation run.

Figure 2.20 Simulink uniform random number generator block.

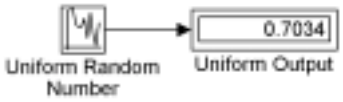


Figure 2.21 Simulink normal random number generator block.

Exercises[1]

*1. Indicate if each of the following systems is a dynamic system:

(a) A spacecraft coasting through deep space (gravitational effects are not significant).

(b) A logic circuit consisting of ideal boolean AND, OR, and NOT gates.

(c) A filter circuit consisting of resistors and capacitors.

(d) An automotive suspension system consisting of the frame, wheels, springs, and shock absorbers.

*2. Write the following differential equation as a system of first-order equations:

$$x''' = ax'' + bx' + cx + d$$

*3. Show how Equation 2.14 (page 32) is changed if wind resistance is included. The angular acceleration due to wind resistance is modeled as a constant C multiplied by the square of the bob velocity, acting in the direction opposite to the velocity.

4. Using the results of Exercise 3, add the effect of a steady wind to the model. The wind is modeled as moving with constant velocity v_w from left to right in the system shown in Figure 2.2 (page 31).

5. Verify that Equation 2.18 (page 35) is the solution of Equation 2.17.

6. Given a function $y = f(x)$ with x breakpoints {0, 0.2, 0.5, 0.7, 0.9} and y values at the breakpoints {0, 1.2, 2.1, 4.3, 3.9}, estimate the value of the function using linear interpolation at x values of 0.1, 0.2, and 0.55.

1. Answers are provided for those exercises with an asterisk in Appendix A, page 293.

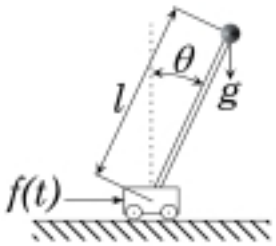
*7. Given a function $z = f(x, y)$ with x breakpoints {0.5, 0.8, 1.5, 1.7}, y breakpoints {0, 1.4, 2.5, 4.7}, and z values at the breakpoints as shown below, where the x breakpoints are in increasing order across the columns to the right and the y breakpoints are in increasing order down the rows, estimate the value of the function using linear interpolation at (x, y) input value pairs of (0.8, 2.6), (0.7, 2) and (0.7, 0.9).

$$z = \begin{bmatrix} 0.2 & 0.3 & 0.6 & 0.5 \\ 0.3 & 0.4 & 0.3 & 0.5 \\ 0.5 & 0.7 & 0.4 & 0.7 \\ 0.6 & 0.9 & 0.7 & 0.9 \end{bmatrix}$$

8. Given the function $y = \cos(x^4)$, develop an algorithm for selecting linear interpolation breakpoints over the interval $0 \le x \le 2$ so that the maximum interpolation error is minimized. Use the minimum number of breakpoints possible and limit the interpolation to 0.05 across given range of x. Use your algorithm to select a set of breakpoints and compare your results with Figure 2.7 (page 42).

9. Derive the equations of motion for the inverted pendulum shown in ExerciseFigure 2.1 in terms of the cart position x relative to a fixed point on the ground and the pendulum angle from the vertical θ. The cart has mass m_c, the pendulum bob has mass m_b and is supported by a stiff shaft of length l, and the gravitational acceleration is g. The masses of the pendulum shaft and cart wheels are negligible. Wind resistance and friction in the pendulum pivot and in the wheel motion can also be ignored. $f(t)$ is an arbitrary external force applied to the cart in the x direction.

ExerciseFigure 2.1 Inverted pendulum.



References

[1] Franklin, Gene F., J. David Powell, and Abbas Emami-Naeini, *Feedback Control of Dynamic Systems*. Reading, MA: Addison Wesley, 1986.

[2] Churchill, Ruel V., *Operational Mathematics*, Boston, MA: McGraw-Hill, 1972.

[3] Juang, Jer-Nan, *Applied System Identification*. Upper Saddle River, NJ: Prentice-Hall, 1993.

[4] Oppenheim, Alan V., and Ronald W. Schafer, *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[5] Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge, England: Cambridge University Press, 1992, §3.3.

[6] Beale, R., and T. Jackson, *Neural Computing: An Introduction*. Bristol, England: Adam Hilger, 1990.

[7] Roskam, Jan, *Airplane Flight Dynamics and Automatic Flight Controls*. Lawrence, KS: Roskam Aviation and Engineering Corporation, 1979.

[8] Farrell, Jay A., and Matthew Barth, *The Global Positioning System & Inertial Navigation*. New York, NY: McGraw-Hill, 1999, §2.4.2.

[9] Papoulis, Athanasios, *Probability, Random Variables, and Stochastic Process*. New York, NY: McGraw-Hill, 1991.

[10] Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge, England: Cambridge University Press, 1992, §7.1.

[11] Ibid., §7.2.

[12] Ledin, Jim, *Digital Filtering and Oversampling*. Dr. Dobb's Journal, April, 2000.