A high-angle photograph of a person bungee jumping into a pool of water in a canyon. The person is suspended in the air, arms outstretched, with their legs and bungee cord visible. The water below is dark and reflects the sky and the surrounding green, rocky cliffs. The scene is captured from a high vantage point, looking down into the canyon.

Steven C. Chapra

Applied Numerical Methods
with MATLAB for Engineers and Scientists

PREFACE

Today's engineering and science students routinely confront problems involving numerical solution techniques. Sometimes these solutions are "automatically" generated by a software package. In other cases, students must use programming skills to devise their own solutions. In either case, knowledge of numerical methods is absolutely necessary for developing and interpreting such solutions with wisdom and insight.

This book is written to support a one-semester course in numerical methods. The book's primary audience are students who want to learn numerical methods to solve problems in engineering and science. As such, the methods are motivated by problems rather than by mathematics. That said, sufficient theory is provided so that students come away with insight into the techniques and their shortcomings.

MATLAB[®] provides a great environment for such a course. Although other environments (e.g., Excel/VBA, Mathcad) or languages (e.g., Fortran 90, C++) could have been chosen, MATLAB presently offers a nice combination of handy programming features with powerful built-in numerical capabilities. On the one hand, its M-file programming environment allows students to implement moderately complicated algorithms in a structured and coherent fashion. On the other hand, its built-in numerical capabilities empower students to solve more difficult problems without trying to "reinvent the wheel."

The first chapters provide introductory material including background on mathematical modeling, MATLAB fundamentals, and error analysis. This is followed by chapters dealing with several major areas of numerical methods: root location, linear algebraic equations, least-squares regression, interpolation, integration, ordinary differential equations, and eigenvalues.

I have made a concerted effort to make this book as student-friendly as possible. Thus, I've tried to keep my explanations straightforward and oriented practically. Although my primary intent is to empower students by providing them with a sound introduction to numerical problem solving, I have the ancillary objective of making this introduction exciting and pleasurable. I believe that motivated students who enjoy engineering and science, problem solving, and—yes—programming, will ultimately make better professionals. If my book fosters enthusiasm and appreciation for this subject, I will consider the effort a success.

Acknowledgments Several members of the McGraw-Hill team have contributed to this project. Special thanks are due to Amanda Green, Suzanne Jeans, and Peggy Selle for their encouragement, support, and direction. Beatrice Sussman did a masterful job of copyediting the manuscript, and Rick Noel developed a clean, clear, and aesthetically pleasing design.

Valuable suggestions were also given by a number of reviewers and colleagues including: Malcolm J. Andrews (Texas A&M University), Larry Caretto (California State University–Northridge), Colm-Cille P. Caulfield (University of California–San Diego), Marc Compere (University of Texas–Austin), Marie Dahleh (University of California–Santa Barbara), Kurt Gurley (University of Florida), Larry Hoffman (Purdue University), Fred Kettleborough (Texas A&M University), Michael R. King (University of Rochester), Joe Koebbe (Utah State University), Isaac Edward Leonard (University of Alberta), Doug Meegan (University of Texas–Austin), Robert R. Meyer (University of Wisconsin–Madison), George Novacky (University of Pittsburgh), Siva Parameswaran (Texas Tech University), Leonardo Pérez y Pérez (California State University–Long Beach), David Rosen (Georgia Institute of Technology), Ralph G. Selfridge (University of Florida), Avi Singhal (Arizona State University), Dipendra K. Sinha (San Francisco State University), John Strikwerda (University of Wisconsin–Madison), Vishwanath V. Subramaniam (The Ohio State University), Brian Vick (Virginia Polytechnic Institute and State University), Kimberly F. Williams (Virginia Polytechnic Institute and State University), William R. Wise (University of Florida), J. B. (Barry) Wiskel (University of Alberta), Hsin-i Wu (Texas A&M University), and Wenjing Ye (Georgia Institute of Technology).

It should be stressed that although I received useful advice from the aforementioned individuals, I am responsible for any inaccuracies or mistakes you may find in this book. Please contact me via e-mail if you should detect any errors.

Finally, I want to thank my family—and in particular my wife, Cynthia—for their love, patience, and support through the time I’ve spent on this project.

Steve Chapra
Tufts University
Medford, Massachusetts
steven.chapra@tufts.edu

GUIDED TOUR

Chapter Objectives *Chapter Objectives* begin each chapter. The objectives provide students with the function of each chapter as well as the specific topics covered in each chapter. The objectives enable students to set tangible goals before they begin each chapter.

**Curve Fitting:
Fitting a Straight Line**

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to how least squares regression can be used to fit a straight line to measured data. Specific objectives and topics covered are:

- Understanding the difference between regression and interpolation.
- Fitting (regression) with some basic descriptive statistics and the normal distribution.
- Knowing how to compute the slope and intercept of a least squares line with linear regression.
- Knowing how to compute and understand the meaning of the coefficient of determination and the standard error of the estimate.
- Understanding how to use transformations to linearize nonlinear equations so that they can be fit with linear regression.
- Knowing how to implement least squares regression with MATLAB.

YOU'VE GOT A PROBLEM

In Chap. 1, we noted that a free-falling object works as a linear system in subject to the acceleration of an air resistance. An air resistance is the amount that the force that is proportional to the square of velocity as in:

$$F_D = -k v^2 \quad (1.1)$$

where F_D is the upward force of air resistance (N), k is a drag coefficient (kg/m), and v is velocity (m/s).

Equation (1.1) is a nonlinear differential equation. Although such equations are difficult to solve, they can be solved by using numerical methods. One such equation is depicted in Fig. 1.1. An individual is suspended in a wind

You've Got a Problem A section entitled *You've Got a Problem* can be found on the first page of most chapters. Here Chapra poses a real-life problem that requires the type of numerical solution technique that is the subject of the chapter. The intent is to introduce the student to the topic via a tangible example rather than through abstract mathematics. After an exposition of the numerical methods, the problem is then revisited in order to demonstrate how the learned material provides the means to solve the problem.

FIGURE 4.3

The relationship of $f(x)$ to $f'(x)$ and $f''(x)$ for a function $f(x)$ and its first and second derivatives.

FIGURE 4.3
The relationship of $f(x)$ to $f'(x)$ and $f''(x)$ for a function $f(x)$ and its first and second derivatives.

A useful way to approximate the Taylor series is to build it term by term. A good problem solver for this series is to produce a function value at one point in terms of the function value and its derivatives at another point.

Suppose that you are standing on a platform that is located on the side of a hill that descends (Fig. 4.3). We'll call your horizontal position x , and your vertical distance with respect to the base of the hill $f(x)$. You are given the task of predicting the height at a position x_1 , which is a distance h away from you.

All this you are placed on a platform that is completely horizontal so that you have no idea that the hill is sloping down away from you. As the police, what would be your best guess at the height at x_1 ? If you think about it, you should have no idea whatsoever what the height is. The best guess would be to use the height at where you are standing now! You could express this prediction mathematically as:

$$f(x_1) \approx f(x_0) \quad (4.1)$$

This relationship, which is called the zero-order approximation, indicates that the value of f at the new point is the same as the value at the old point. This result seems rather strange because if x_1 and x_0 are close to each other, it is likely that the new value is probably also close to the old value.

Equation (4.1) provides a poorer estimate of the function being approximated. It is less accurate. Therefore, you would be right only if the function were to be constant or a perfectly flat surface. However, if the function changes at all over the interval, additional terms of the Taylor series are required to provide a better estimate.

So now you are allowed to go off the platform and stand on the hill surface with only a position in front of you and the other behind. You immediately sense that the first

Theory Presented as it Informs Key Concepts The text is intended for Numerical Methods users, not developers. Therefore, theory is not included for "theory's sake," for example no proofs. Theory is included as it informs key concepts such as the Taylor Series, convergence, condition, etc. Hence, the student is shown how the theory connects with practical issues in problem solving.

GUIDED TOUR

xiii

Illustrations and Tables

Illustrations and tables are clear and accurate in order to help students better visualize the important concepts presented in the text.

7.1 WHAT ARE LINEAR ALGEBRAIC EQUATIONS?

125

7.1 WHAT ARE LINEAR ALGEBRAIC EQUATIONS?

Linear algebraic equations are of the general form,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \quad (7.2)$$

where the a 's are constant coefficients, the b 's are constants, the x 's are unknowns, and n is the number of equations. All other algebraic equations are nonlinear.

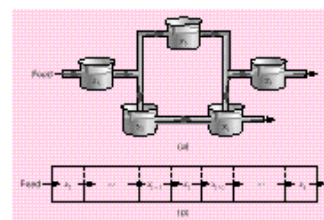
7.1.1 Linear Algebraic Equations and Engineering Practice

Many of the fundamental equations of engineering and science are based on conservation laws. Some familiar quantities that conform to such laws are mass, energy, and momentum. In mathematical terms, these principles lead to balance or continuity equations that relate system behavior as represented by the levels or response of the quantity being modeled to the properties or characteristics of the system and the external stimuli or forcing functions acting on the system.

As an example, the principle of mass conservation can be used to formulate a model for a series of chemical reactors (Fig. 7.3a). For this case, the quantity being modeled is the mass of the chemical in each reactor. The system properties are the reaction characteristics

FIGURE 7.3

Two types of system flow can be modeled using linear algebraic equations: (1) lumped-variable systems that involve coupled finite components and (2) distributed-variable systems that involve a continuum.



28 MATLAB FUNDAMENTALS

TABLE 2.1 Specifiers for colors, symbols, and line types.

Colors		Symbols		Line Types
Blue	b	Point	.	Solid
Cyan	c	Circle	o	Dashed
Red	r	Xmark	x	Dash-dot
Dark	k	Star	*	Dotted
Magenta	m	Star	*	Grayed
Yellow	y	Square	s	
Black	k	Diamond	d	
		Upward	^	
		Downward	v	
		Leftward	<	
		Rightward	>	
		Triangle	^	
		Triangle	v	
		Triangle	<	
		Triangle	>	

Table 2.1 lists the available specifiers. For example, if you want to use open circles with

`plot(t1, y1, 'o')`

MATLAB allows you to display more than one data set on the same plot. For example, if you want to connect each data marker with a straight line you could type

`plot(t1, y1, 'o', 'b')`

There are other features of graphics that are useful—for example, plotting objects instead of lines, families of curves, plots on the complex plane, multiple graph windows, log-log or scaling plots, three-dimensional mesh plots, and contour plots. As described next, a variety of resources are available to learn about these as well as other MATLAB capabilities.

2.6 OTHER RESOURCES

The *Integrating* was designed to focus on those features of MATLAB that we will be using in the remainder of this book. As such, it is obviously not a comprehensive overview of all of MATLAB's capabilities. If you are interested in learning more, you should consult one of the excellent books devoted to MATLAB (e.g., Palm, 2004).

Further, the package itself includes an extensive help facility that can be accessed by clicking on the *Help* menu in the command window. This will provide you with a number of different options for exploring and searching through MATLAB's Help system. In addition, it provides access to a number of interactive demos.

As described in this chapter, help is also available in interactive mode by typing the `help` command followed by the name of a command or function.

If you do not know the name, you can use the `lookfor` command to search the MATLAB Help files for occurrences of text. For example, suppose that you want to find all the commands and functions that relate to logarithms; you could enter

`>> lookfor log*`

and MATLAB will display all references that include the word *logarithm*.

Introductory MATLAB Material The text includes two introductory chapters on how to use MATLAB. Chapter 2 shows students how to perform computations and create graphs in MATLAB's standard command mode. Chapter 3 provides a primer on developing numerical programs via MATLAB M-file functions. Thus, the text provides students with the means to develop their own numerical algorithms as well as to tap into MATLAB's powerful built-in routines.

Algorithms Presented Using MATLAB M-files Instead of using pseudocode, this book presents algorithms as well-structured MATLAB M-files. Aside from being useful computer programs, these provide students with models for their own M-files that they will develop as homework exercises.

specify an absolute error. For these cases, bisection along with Eq. (5.6) can provide a useful root location algorithm.

5.4.1 MATLAB M-file

An M-file to implement bisection is displayed in Fig. 5.7. It is named the function `bisect` along with lower (`xl`) and upper (`xu`) guesses. In addition an optional stopping criterion

FIGURE 5.7
An M-file to implement the bisection method.

```
function root = bisect(fcn,xl,xu,ee,maxit)
% bisection(xl,xu,ee,maxit)
% uses bisection method to find the root of a function
% input:
% fcn = name of function
% xl, xu = lower and upper guesses
% ee = (optional) stopping criterion (R)
% maxit = (optional) maximum allowable iterations
% output:
% root = real root

if fcn(xl)*fcn(xu)>0 %if guesses do not bracket a sign
    error('no bracket') %change, display an error message
    return
end
% if necessary, assign default values
if nargin<2, maxit = 50; end %if maxit blank set to 50
if nargin<3, ee = 0.001; end %if ee blank set to 0.001

% bisection
iter = 0;
xt = xl;
while (1)
    xtold = xt;
    xt = (xl + xu)/2;
    iter = iter + 1;
    if xt == 0, ee = abs((xt - xtold)/xt) * 100; end
    root = fcn(xt)/fcn(xt);
    if root < 0
        xu = xt;
    elseif root > 0
        xl = xt;
    else
        ee = 0;
    end
    if ee <= ee || iter == maxit, break, end
end
root = xt;
```

EXAMPLE 5.5 The False-Position Method

Problem Statement: Use false position to solve the same problem approached graphically and with Newton in Examples 5.1 and 5.3.

Solution: As in Example 5.3, initiate the computation with guesses of $x_l = 50$ and $x_u = 200$.

First iteration:

$$\begin{aligned} x_l &= 50 & f(x_l) &= 4.579387 \\ x_u &= 200 & f(x_u) &= 0.860291 \\ x_r &= 200 - \frac{0.860291(50 - 200)}{4.579387 - 0.860291} = 176.2773 \end{aligned}$$

which has a true relative error of 23.5%.

Second iteration:

$$f(x_l)f(x_r) = 2.592752$$

Therefore, the root lies in the first subinterval, and x_l becomes the upper limit for the next iteration, $x_u = 176.2773$.

$$\begin{aligned} x_l &= 50 & f(x_l) &= 4.579387 \\ x_u &= 176.2773 & f(x_u) &= 0.566174 \\ x_r &= 176.2773 - \frac{0.566174(50 - 176.2773)}{4.579387 - 0.566174} = 162.3828 \end{aligned}$$

which has true and approximate relative errors of 13.76% and 8.59%, respectively. Additional iterations can be performed to refine the estimates of the root.

Although false position often performs better than bisection, there are other cases where it does not. As in the following example, there are certain cases where bisection yields superior results.

EXAMPLE 5.6 A Case Where Bisection Is Preferable to False Position

Problem Statement: Use bisection and false position to locate the root of

$$f(x) = y^{10} - 1$$

between $x = 0$ and 1.3.

Solution: Using bisection, the results can be summarized as

Iteration				e (%)	e (%)
1	0	1.3	0.65	100.0	35
2	0.65	1.3	0.975	93.8	2.5
3	0.975	1.3	1.1375	14.3	13.8
4	0.975	1.1375	1.10625	2.7	5.6
5	0.975	1.08625	1.015625	4.0	1.6

Worked Examples Each example begins with a problem statement and ends with a solution. The solution is laid out in detail so that students can clearly follow the steps in the numerical computation.

GUIDED TOUR

xv

122 ROOTS OF EQUATIONS: OPEN METHODS

Conservation of energy can be used to show that

$$\frac{1}{2}mv^2 + \frac{1}{2}kx^2 = \text{const}$$

Solve for x given the following parameter values: $k = 40,000 \text{ N/m}$, $v = 40 \text{ m/s}$, $x = 10 \text{ m}$, $g = 9.81 \text{ m/s}^2$, and $A = 0.1 \text{ m}$.

6.20 Aerospace engineers sometimes compute the trajectory of projectiles such as rockets. A related problem deals with the trajectory of a thrown ball. The trajectory of a ball thrown by a right fielder is defined by the x, y coordinates as displayed in Fig. P6.20. The trajectory can be modeled as

$$y = -0.005x^2 + 0.01x + 1.5$$

Find the appropriate initial angle θ , if $v = 30 \text{ m/s}$, and the distance to home plate is 90 m . Note that the throw leaves the right fielder's hand at an elevation of 1.5 m and the catcher receives it at 1 m .

6.21 You are designing a spherical tank (Fig. P6.21) to hold water for a small village in a developing country. The volume of liquid it can hold can be computed as

$$V = \pi h^2 \left(\frac{3R}{2} - \frac{h}{2} \right)$$

where V = volume (ft^3), R = depth of water in tank (ft), and h = the tank radius (ft).

If $R = 10 \text{ ft}$, what depth must the tank be filled to contain a liquid 1000 ft^3 ? Use three iterations of the most efficient numerical method possible to determine your answer. Determine

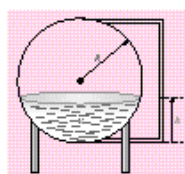


FIGURE P6.21

the appropriate relative error after each iteration. Also, provide justification for your choice of method. Extra information: (a) For bracketing methods, initial guesses of h and R will bracket a single root for this example. (b) For open methods, an initial guess of R will always converge.

6.22 Perform the standard MATLAB operations as those in Example 6.7 to manipulate and find all the roots of the polynomial

$$p(s) = s^4 - 2s^3 + 6s^2 - 10s + 5$$

6.23 In control system analysis, transfer functions are developed that mathematically relate the dynamics of a system's input to its output. A transfer function for a robotic positioning system is given by

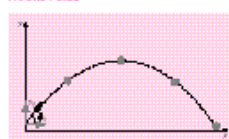
$$G(s) = \frac{C(s)}{D(s)} = \frac{s^2 + 6s^2 + 25s + 24}{s^4 + 10s^3 + 35s^2 + 50s + 24}$$

where $G(s)$ = system gain, $C(s)$ = system output, $D(s)$ = system input, and s = Laplace transform complex frequency. Use MATLAB to find the roots of the numerator and denominator and factor these into the form

$$G(s) = \frac{(s - a_1)(s - a_2)(s - a_3)(s - a_4)}{(s - b_1)(s - b_2)(s - b_3)(s - b_4)}$$

where a_i and b_i = the roots of the numerator and denominator, respectively.

FIGURE P6.20



Useful Indexes Appendix A contains MATLAB commands and Appendix B contains M-file functions.

APPENDIX A MATLAB BUILT-IN FUNCTIONS

abs, 25	linspace, 50, 227	poly, 275
acos, 25	log, 24	plot, 18
acosd, 256	log10, 24, 227	plot, 26-29
chol, 180-170	log2, 135, 138-139, 176	poly, 118, 300
cond, 180-181, 236	length, 26	polyfit, 217, 230, 259, 252, 255
conv, 319	linspace, 21	polyval, 118, 218, 239, 252-255
delquad, 320	log, 24	quad, 318-319
det, 118	log10, 24	quad, 318
diag, 15	log2, 25, 94	realmin, 94
exp, 272-273	log10, 30	realmax, 95
expm, 25	log2, 25	round, 317-319, 364
expm, 65	logspace, 21	round, 36
expm, 320	logspace, 21	sign, 41, 222
expm, 36	logspace, 21	sin, 25
exp, 25	logspace, 21	sin, 134
exp, 113	logspace, 21	split, 272-274
factor, 13, 34	logspace, 21	sqrt, 25-35
factor, 40-50, 227	logspace, 21	sqrt, 25
factor, 231	logspace, 21	sum, 169
factor, 100, 17	logspace, 21	sum, 25-26
factor, 16-17	logspace, 21	tan, 27
factor, 16-17	logspace, 21	tan, 300
factor, 113-116	logspace, 21	tan, 36
find, 27	logspace, 21	tan, 19
find, 32	logspace, 21	tan, 27
find, 27	logspace, 21	tan, 27
find, 27	logspace, 21	tan, 27
find, 27	logspace, 21	tan, 27

375



Supplements A text Web site is available at <http://www.mhhe.com/chapra>. Resources include PowerPoint slides of text figures and chapter objectives, M-files and additional MATLAB resources. Available to instructors only, the detailed solutions for all text problems will be delivered via CD-Rom, in our new, electronic, Complete Online Solution Manual Organization System. COSMOS is a database management tool geared toward assembling homework assignments, tests and quizzes.

1

Mathematical Modeling, Numerical Methods, and Problem Solving

CHAPTER OBJECTIVES

The primary objective of this chapter is to provide you with a concrete idea of what numerical methods are and how they relate to engineering and scientific problem solving. Specific objectives and topics covered are

- Learning how mathematical models can be formulated on the basis of scientific principles to simulate the behavior of a simple physical system.
- Understanding how numerical methods afford a means to generate solutions in a manner that can be implemented on a digital computer.
- Understanding the different types of conservation laws that lie beneath the models used in the various engineering disciplines and appreciating the difference between steady-state and dynamic solutions of these models.
- Learning about the different types of numerical methods we will be covering in this book.

YOU'VE GOT A PROBLEM

Suppose that a bungee-jumping company hires you. You're given the task of predicting the velocity of a jumper (Fig. 1.1) as a function of time during the free-fall part of the jump. This information will be used as part of a larger analysis to determine the length and required strength of the bungee cord for jumpers of different mass.

You know from your studies of physics that the acceleration should be equal to the ratio of the force to the mass (Newton's second law). Based on this insight and your knowledge



FIGURE 1.1
Forces acting on a
free-falling bungee
jumper.

of fluid mechanics, you develop the following mathematical model for the rate of change of velocity with respect to time,

$$\frac{dv}{dt} = g - \frac{c_d}{m}v^2$$

where v = vertical velocity (m/s), t = time (s), g = the acceleration due to gravity ($\approx 9.81 \text{ m/s}^2$), c_d = a second-order drag coefficient (kg/m), and m = the jumper's mass (kg).

Because this is a differential equation, you know that calculus might be used to obtain an analytical or exact solution for v as a function of t . However, in the following pages, we will illustrate an alternative solution approach. This will involve developing a computer-oriented numerical or approximate solution.

Aside from showing you how the computer can be used to solve this particular problem, our more general objective will be to illustrate (a) what numerical methods are and (b) how they figure in engineering and scientific problem solving. In so doing, we will also show how mathematical models figure prominently in the way engineers and scientists use numerical methods in their work.

1.1 A SIMPLE MATHEMATICAL MODEL

A *mathematical model* can be broadly defined as a formulation or equation that expresses the essential features of a physical system or process in mathematical terms. In a very general sense, it can be represented as a functional relationship of the form

$$\text{Dependent variable} = f\left(\begin{array}{c} \text{independent} \\ \text{variables} \end{array}, \begin{array}{c} \text{parameters,} \\ \text{forcing} \\ \text{functions} \end{array}\right) \quad (1.1)$$

where the *dependent variable* is a characteristic that usually reflects the behavior or state of the system; the *independent variables* are usually dimensions, such as time and space, along which the system's behavior is being determined; the *parameters* are reflective of the system's properties or composition; and the *forcing functions* are external influences acting upon it.

The actual mathematical expression of Eq. (1.1) can range from a simple algebraic relationship to large complicated sets of differential equations. For example, on the basis of his observations, Newton formulated his second law of motion, which states that the time rate of change of momentum of a body is equal to the resultant force acting on it. The mathematical expression, or model, of the second law is the well-known equation

$$F = ma \quad (1.2)$$

where F is the net force acting on the body (N, or kg m/s^2), m is the mass of the object (kg), and a is its acceleration (m/s^2).

The second law can be recast in the format of Eq. (1.1) by merely dividing both sides by m to give

$$a = \frac{F}{m} \quad (1.3)$$

where a is the dependent variable reflecting the system's behavior, F is the forcing function, and m is a parameter. Note that for this simple case there is no independent variable because we are not yet predicting how acceleration varies in time or space.

Equation (1.3) has a number of characteristics that are typical of mathematical models of the physical world.

- It describes a natural process or system in mathematical terms.
- It represents an idealization and simplification of reality. That is, the model ignores negligible details of the natural process and focuses on its essential manifestations. Thus, the second law does not include the effects of relativity that are of minimal importance when applied to objects and forces that interact on or about the earth's surface at velocities and on scales visible to humans.
- Finally, it yields reproducible results and, consequently, can be used for predictive purposes. For example, if the force on an object and its mass are known, Eq. (1.3) can be used to compute acceleration.

Because of its simple algebraic form, the solution of Eq. (1.2) was obtained easily. However, other mathematical models of physical phenomena may be much more complex, and either cannot be solved exactly or require more sophisticated mathematical techniques than simple algebra for their solution. To illustrate a more complex model of this kind, Newton's second law can be used to determine the terminal velocity of a free-falling body near the earth's surface. Our falling body will be a bungee jumper (Fig. 1.1). For this case, a model can be derived by expressing the acceleration as the time rate of change of the velocity (dv/dt) and substituting it into Eq. (1.3) to yield

$$\frac{dv}{dt} = \frac{F}{m} \quad (1.4)$$

where v is velocity (in meters per second). Thus, the rate of change of the velocity is equal to the net force acting on the body normalized to its mass. If the net force is positive, the object will accelerate. If it is negative, the object will decelerate. If the net force is zero, the object's velocity will remain at a constant level.

Next, we will express the net force in terms of measurable variables and parameters. For a body falling within the vicinity of the earth, the net force is composed of two opposing forces: the downward pull of gravity F_D and the upward force of air resistance F_U (Fig. 1.1):

$$F = F_D - F_U \quad (1.5)$$

If force in the downward direction is assigned a positive sign, the second law can be used to formulate the force due to gravity as

$$F_D = mg \quad (1.6)$$

where g is the acceleration due to gravity (9.81 m/s^2).

Air resistance can be formulated in a variety of ways. Knowledge from the science of fluid mechanics suggests that a good first approximation would be to assume that it is proportional to the square of the velocity,

$$F_U = -c_d v^2 \quad (1.7)$$

where c_d is a proportionality constant called the *drag coefficient* (kg/m). Thus, the greater the fall velocity, the greater the upward force due to air resistance. The parameter c_d accounts for properties of the falling object, such as shape or surface roughness, that affect air resistance. For the present case, c_d might be a function of the type of clothing or the orientation used by the jumper during free fall.

The net force is the difference between the downward and upward force. Therefore, Eqs. (1.4) through (1.7) can be combined to yield

$$\frac{dv}{dt} = g - \frac{c_d}{m} v^2 \quad (1.8)$$

Equation (1.8) is a model that relates the acceleration of a falling object to the forces acting on it. It is a *differential equation* because it is written in terms of the differential rate of change (dv/dt) of the variable that we are interested in predicting. However, in contrast to the solution of Newton's second law in Eq. (1.3), the exact solution of Eq. (1.8) for the velocity of the jumper cannot be obtained using simple algebraic manipulation. Rather, more advanced techniques such as those of calculus must be applied to obtain an exact or analytical solution. For example, if the jumper is initially at rest ($v = 0$ at $t = 0$), calculus can be used to solve Eq. (1.8) for

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) \quad (1.9)$$

where \tanh is the hyperbolic tangent that can be either computed directly¹ or via the more elementary exponential function as in

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (1.10)$$

Note that Eq. (1.9) is cast in the general form of Eq. (1.1) where $v(t)$ is the dependent variable, t is the independent variable, c_d and m are parameters, and g is the forcing function.

EXAMPLE 1.1 Analytical Solution to the Bungee Jumper Problem

Problem Statement. A bungee jumper with a mass of 68.1 kg leaps from a stationary hot air balloon. Use Eq. (1.9) to compute velocity for the first 12 s of free fall. Also determine the terminal velocity that will be attained for an infinitely long cord (or alternatively, the jumpmaster is having a particularly bad day!). Use a drag coefficient of 0.25 kg/m.

¹MATLAB® allows direct calculation of the hyperbolic tangent via the built-in function `tanh(x)`.

Solution. Inserting the parameters into Eq. (1.9) yields

$$v(t) = \sqrt{\frac{9.81(68.1)}{0.25}} \tanh\left(\sqrt{\frac{9.81(0.25)}{68.1}} t\right) = 51.6938 \tanh(0.18977t)$$

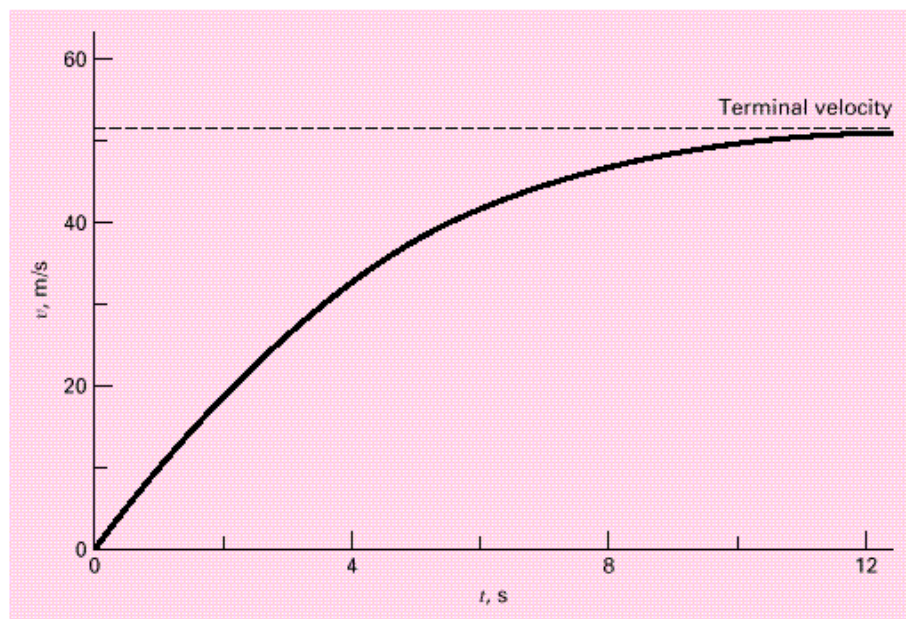
which can be used to compute

$t, \text{ s}$	$v, \text{ m/s}$
0	0
2	18.7292
4	33.1118
6	42.0762
8	46.9575
10	49.4214
12	50.6175
∞	51.6938

According to the model, the jumper accelerates rapidly (Fig. 1.2). A velocity of 49.4214 m/s (about 110 mi/h) is attained after 10 s. Note also that after a sufficiently long

FIGURE 1.2

The analytical solution for the bungee jumper problem as computed in Example 1.1. Velocity increases with time and asymptotically approaches a terminal velocity.



time, a constant velocity, called the *terminal velocity*, of 51.6983 m/s (115.6 mi/h) is reached. This velocity is constant because, eventually, the force of gravity will be in balance with the air resistance. Thus, the net force is zero and acceleration has ceased.

Equation (1.9) is called an *analytical* or *closed-form solution* because it exactly satisfies the original differential equation. Unfortunately, there are many mathematical models that cannot be solved exactly. In many of these cases, the only alternative is to develop a numerical solution that approximates the exact solution.

Numerical methods are those in which the mathematical problem is reformulated so it can be solved by arithmetic operations. This can be illustrated for Eq. (1.8) by realizing that the time rate of change of velocity can be approximated by (Fig. 1.3):

$$\frac{dv}{dt} \approx \frac{\Delta v}{\Delta t} = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i} \quad (1.11)$$

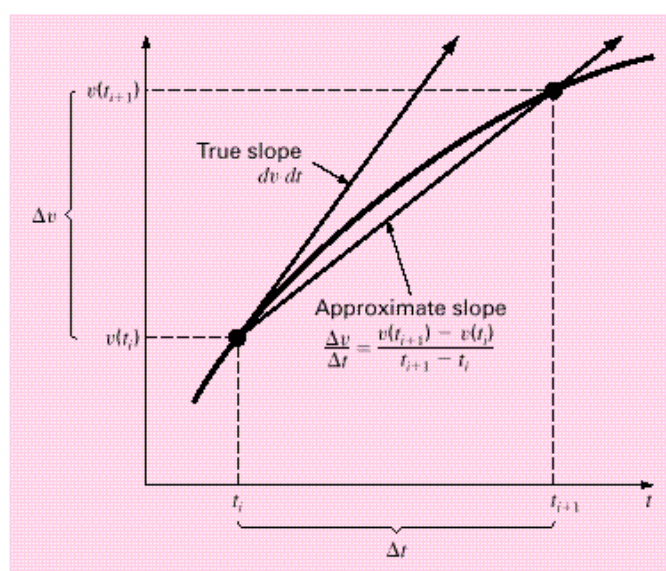
where Δv and Δt are differences in velocity and time computed over finite intervals, $v(t_i)$ is velocity at an initial time t_i , and $v(t_{i+1})$ is velocity at some later time t_{i+1} . Note that $dv/dt \approx \Delta v/\Delta t$ is approximate because Δt is finite. Remember from calculus that

$$\frac{dv}{dt} = \lim_{\Delta t \rightarrow 0} \frac{\Delta v}{\Delta t}$$

Equation (1.11) represents the reverse process.

FIGURE 1.3

The use of a finite difference to approximate the first derivative of v with respect to t .



1.1 A SIMPLE MATHEMATICAL MODEL

7

Equation (1.11) is called a *finite divided difference* approximation of the derivative at time t_i . It can be substituted into Eq. (1.8) to give

$$\frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i} = g + \frac{c_d}{m}v(t_i)^2$$

This equation can then be rearranged to yield

$$v(t_{i+1}) - v(t_i) = \left[g + \frac{c_d}{m}v(t_i)^2 \right] (t_{i+1} - t_i) \quad (1.12)$$

Notice that the term in brackets is the right-hand side of the differential equation itself [Eq. (1.8)]. That is, it provides a means to compute the rate of change or slope of v . Thus, the equation can be rewritten as

$$v_{i+1} - v_i = \frac{dv_i}{dt} \Delta t \quad (1.13)$$

where the nomenclature v_i designates velocity at time t_i and $\Delta t = t_{i+1} - t_i$.

Thus, we can see that the differential equation has been transformed into an equation that can be used to determine the velocity algebraically at t_{i+1} using the slope and previous values of v and t . If you are given an initial value for velocity at some time t_i , you can easily compute velocity at a later time t_{i+1} . This new value of velocity at t_{i+1} can in turn be employed to extend the computation to velocity at t_{i+2} and so on. Thus at any time along the way,

$$\text{New value} = \text{old value} + \text{slope} \cdot \text{step size}$$

This approach is formally called *Euler's method*. We'll discuss it in more detail when we turn to differential equations later in this book.

EXAMPLE 1.2 Numerical Solution to the Bungee Jumper Problem

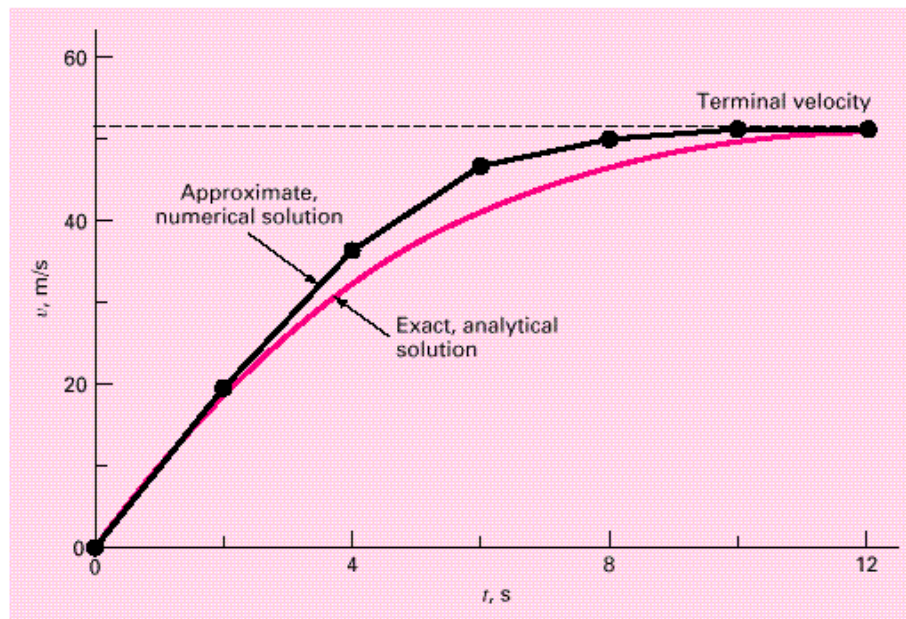
Problem Statement. Perform the same computation as in Example 1.1 but use Eq. (1.13) to compute velocity with Euler's method. Employ a step size of 2 s for the calculation.

Solution. At the start of the computation ($t_i = 0$), the velocity of the jumper is zero. Using this information and the parameter values from Example 1.1, Eq. (1.13) can be used to compute velocity at $t_{i+1} = 2$ s:

$$v = 0 + \left[9.81 + \frac{0.25}{68.1}(0)^2 \right] \cdot 2 = 19.62 \text{ m/s}$$

For the next interval (from $t = 2$ to 4 s), the computation is repeated, with the result

$$v = 19.62 + \left[9.81 + \frac{0.25}{68.1}(19.62)^2 \right] \cdot 2 = 36.4137 \text{ m/s}$$

**FIGURE 1.4**

Comparison of the numerical and analytical solutions for the bungee jumper problem.

The calculation is continued in a similar fashion to obtain additional values:

t, s	$v, m/s$
0	0
2	19.6200
4	36.4137
6	46.2983
8	50.1802
10	51.3123
12	51.6008
∞	51.6938

The results are plotted in Fig. 1.4 along with the exact solution. We can see that the numerical method captures the essential features of the exact solution. However, because we have employed straight-line segments to approximate a continuously curving function, there is some discrepancy between the two results. One way to minimize such discrepancies is to use a smaller step size. For example, applying Eq. (1.13) at 1-s intervals results in a smaller error, as the straight-line segments track closer to the true solution. Using hand calculations, the effort associated with using smaller and smaller step sizes would make such numerical solutions impractical. However, with the aid of the computer, large numbers of calculations can be performed easily. Thus, you can accurately model the velocity of the jumper without having to solve the differential equation exactly.

As in Example 1.2, a computational price must be paid for a more accurate numerical result. Each halving of the step size to attain more accuracy leads to a doubling of the number of computations. Thus, we see that there is a trade-off between accuracy and computational effort. Such trade-offs figure prominently in numerical methods and constitute an important theme of this book.

1.2 CONSERVATION LAWS IN ENGINEERING AND SCIENCE

Aside from Newton's second law, there are other major organizing principles in science and engineering. Among the most important of these are the *conservation laws*. Although they form the basis for a variety of complicated and powerful mathematical models, the great conservation laws of science and engineering are conceptually easy to understand. They all boil down to

$$\text{Change} \cdot \text{increases} \cdot \text{decreases} \quad (1.14)$$

This is precisely the format that we employed when using Newton's law to develop a force balance for the bungee jumper [Eq. (1.8)].

Although simple, Eq. (1.14) embodies one of the most fundamental ways in which conservation laws are used in engineering and science—that is, to predict changes with respect to time. We will give it a special name—the *time-variable* (or *transient*) computation.

Aside from predicting changes, another way in which conservation laws are applied is for cases where change is nonexistent. If change is zero, Eq. (1.14) becomes

$$\text{Change} \cdot 0 \cdot \text{increases} \cdot \text{decreases}$$

or

$$\text{Increases} \cdot \text{decreases} \quad (1.15)$$

Thus, if no change occurs, the increases and decreases must be in balance. This case, which is also given a special name—the *steady-state* calculation—has many applications in engineering and science. For example, for steady-state incompressible fluid flow in pipes, the flow into a junction must be balanced by flow going out, as in

$$\text{Flow in} \cdot \text{flow out}$$

For the junction in Fig. 1.5, the balance can be used to compute that the flow out of the fourth pipe must be 60.

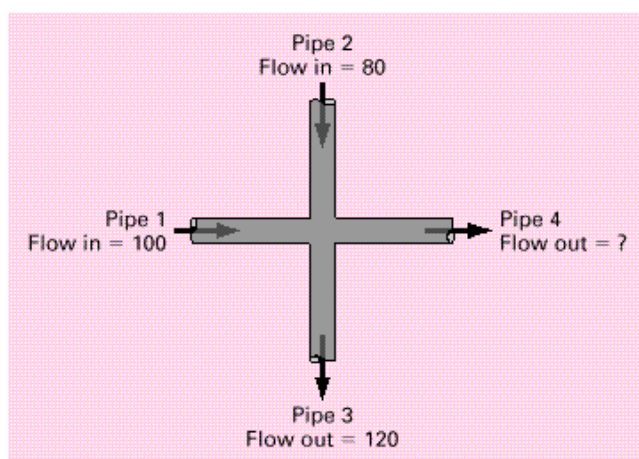
For the bungee jumper, the steady-state condition would correspond to the case where the net force was zero or [Eq. (1.8) with $dv/dt = 0$]

$$mg \cdot c_d v^2 \quad (1.16)$$

Thus, at steady state, the downward and upward forces are in balance and Eq. (1.16) can be solved for the terminal velocity

$$v = \sqrt{\frac{gm}{c_d}}$$

Although Eqs. (1.14) and (1.15) might appear trivially simple, they embody the two fundamental ways that conservation laws are employed in engineering and science. As such, they will form an important part of our efforts in subsequent chapters to illustrate the connection between numerical methods and engineering and science.

**FIGURE 1.5**

A flow balance for steady incompressible fluid flow at the junction of pipes.

Table 1.1 summarizes some models and associated conservation laws that figure prominently in engineering. Many chemical engineering problems involve mass balances for reactors. The mass balance is derived from the conservation of mass. It specifies that the change of mass of a chemical in the reactor depends on the amount of mass flowing in minus the mass flowing out.

Civil and mechanical engineers often focus on models developed from the conservation of momentum. For civil engineering, force balances are utilized to analyze structures such as the simple truss in Table 1.1. The same principles are employed for the mechanical engineering case studies to analyze the transient up-and-down motion or vibrations of an automobile.

Finally, electrical engineering studies employ both current and energy balances to model electric circuits. The current balance, which results from the conservation of charge, is similar in spirit to the flow balance depicted in Fig. 1.5. Just as flow must balance at the junction of pipes, electric current must balance at the junction of electric wires. The energy balance specifies that the changes of voltage around any loop of the circuit must add up to zero.

We should note that there are many other branches of engineering beyond chemical, civil, electrical, and mechanical. Many of these are related to the Big Four. For example, chemical engineering skills are used extensively in areas such as environmental, petroleum, and biomedical engineering. Similarly, aerospace engineering has much in common with mechanical engineering. We will endeavor to include examples from these areas in the coming pages.

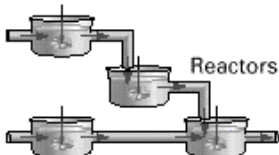

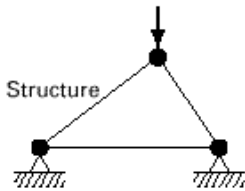
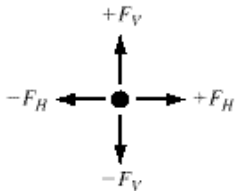
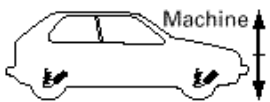
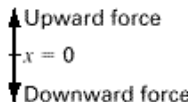
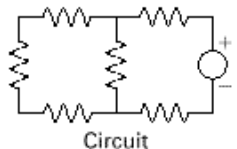
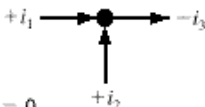
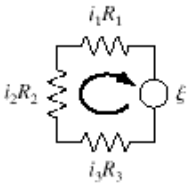
1.3 NUMERICAL METHODS COVERED IN THIS BOOK

We chose Euler's method for this introductory chapter because it is typical of many other classes of numerical methods. In essence, most consist of recasting mathematical operations into the simple kind of algebraic and logical operations compatible with digital computers. Figure 1.6 summarizes the major areas covered in this text.

1.3 NUMERICAL METHODS COVERED IN THIS BOOK

11

TABLE 1.1 Devices and types of balances that are commonly used in the four major areas of engineering. For each case, the conservation law on which the balance is based is specified.

Field	Device	Organizing Principle	Mathematical Expression
Chemical engineering		Conservation of mass	<p>Mass balance: </p> <p>Over a unit of time period $\Delta \text{mass} = \text{inputs} - \text{outputs}$</p>
Civil engineering		Conservation of momentum	<p>Force balance: </p> <p>At each node $\Sigma \text{horizontal forces } (F_H) = 0$ $\Sigma \text{vertical forces } (F_V) = 0$</p>
Mechanical engineering		Conservation of momentum	<p>Force balance: </p> <p>$m \frac{d^2x}{dt^2} = \text{downward force} - \text{upward force}$</p>
Electrical engineering		Conservation of charge	<p>Current balance: </p> <p>For each node $\Sigma \text{current } (i) = 0$</p>
		Conservation of energy	<p>Voltage balance: </p> <p>Around each loop $\Sigma \text{emf's} - \Sigma \text{voltage drops for resistors} = 0$ $\Sigma \xi - \Sigma iR = 0$</p>

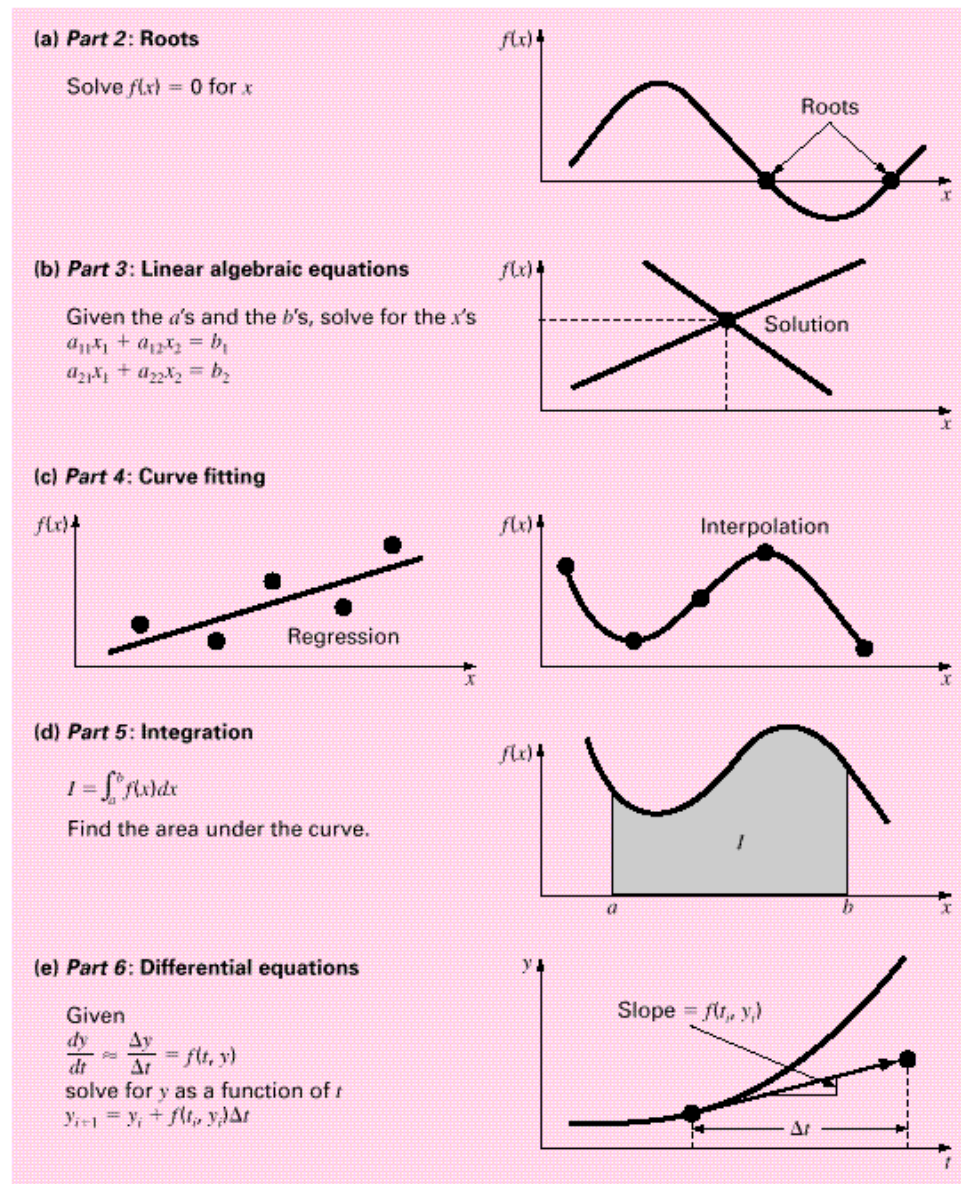


FIGURE 1.6

Summary of the numerical methods covered in this book.

PROBLEMS

1.1 Use calculus to verify that Eq. (1.9) is a solution of Eq. (1.8).

1.2 The following information is available for a bank account:

Date	Deposits	Withdrawals	Balance
5/1			1512.33
6/1	220.13	327.26	
7/1	216.80	378.61	
8/1	450.25	106.80	
9/1	127.31	350.61	

Use the conservation of cash to compute the balance on 6/1, 7/1, 8/1, and 9/1. Show each step in the computation. Is this a steady-state or a transient computation?

1.3 Repeat Example 1.2. Compute the velocity to $t = 12$ s, with a step size of (a) 1 and (b) 0.5 s. Can you make any statement regarding the errors of the calculation based on the results?

1.4 Rather than the nonlinear relationship of Eq. (1.7), you might choose to model the upward force on the bungee jumper as a linear relationship:

$$F_U = -c'v$$

where c' is a first-order drag coefficient (kg/s).

(a) Using calculus, obtain the closed-form solution for the case where the jumper is initially at rest ($v = 0$ at $t = 0$).

(b) Repeat the numerical calculation in Example 1.2 with the same initial condition and parameter values. Use a value of 12.5 kg/s for c' .

1.5 The amount of a uniformly distributed radioactive contaminant contained in a closed reactor is measured by its concentration c (becquerel/liter or Bq/L). The contaminant decreases at a decay rate proportional to its concentration; that is,

$$\text{Decay rate} = -kc$$

where k is a constant with units of day^{-1} . Therefore, according to Eq. (1.14), a mass balance for the reactor can be

written as

$$\frac{dc}{dt} = -kc$$

$$\left(\begin{array}{c} \text{change} \\ \text{in mass} \end{array} \right) = \left(\begin{array}{c} \text{decrease} \\ \text{by decay} \end{array} \right)$$

(a) Use Euler's method to solve this equation from $t = 0$ to 1 d with $k = 0.2 \text{ d}^{-1}$. Employ a step size of $\Delta t = 0.1$ d. The concentration at $t = 0$ is 10 Bq/L.

(b) Plot the solution on a semilog graph (i.e., $\ln c$ versus t) and determine the slope. Interpret your results.

1.6 A storage tank contains a liquid at depth y where $y = 0$ when the tank is half full. Liquid is withdrawn at a constant flow rate Q to meet demands. The contents are resupplied at a sinusoidal rate $3Q \sin^2(t)$. Equation (1.14) can be written for this system as

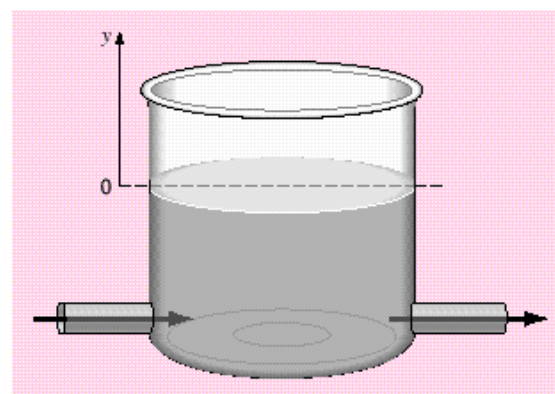
$$\frac{d(Ay)}{dx} = 3Q \sin^2(t) - Q$$

$$\left(\begin{array}{c} \text{change in} \\ \text{volume} \end{array} \right) = (\text{inflow}) - (\text{outflow})$$

or, since the surface area A is constant

$$\frac{dy}{dx} = 3 \frac{Q}{A} \sin^2(t) - \frac{Q}{A}$$

FIGURE P1.6



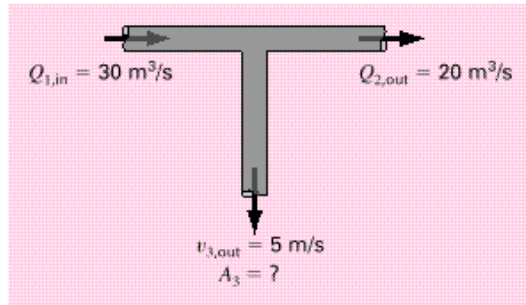


FIGURE P1.8

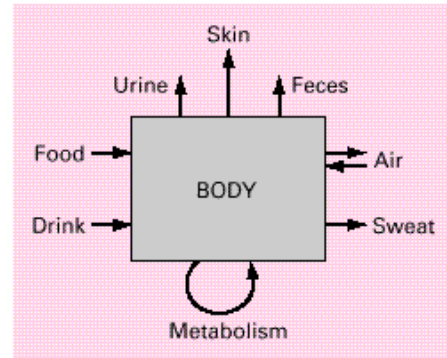


FIGURE P1.9

Use Euler's method to solve for the height y from $t = 0$ to 5 d with a step size of 0.5 d. The parameter values are $A = 1200 \text{ m}^2$ and $Q = 400 \text{ m}^3/\text{d}$.

1.7 For the free-falling bungee jumper with linear drag (Prob. 1.4), assume a first jumper is 68.1 kg and has a drag coefficient of 12.5 kg/s . If a second jumper has a drag coefficient of 14 kg/s and a mass of 75 kg , how long will it take her to reach the same velocity jumper 1 reached in 10 s ?

1.8 The volume flow rate through a pipe is given by $Q = vA$, where v is the average velocity and A is the cross-sectional area. Use volume-continuity to solve for the required area in pipe 3 of Fig. P1.8.

1.9 The following is a steady-state H_2O mass balance for an average man in one day (Fig. P1.9). One thousand grams are ingested as food, 1200 g are ingested as liquid, and the body will produce metabolically some additional water. In breathing air, the exchange is 50 g while inhaling, and 400 g while exhaling over a one-day period. The body will also lose H_2O through sweat, urine, feces, and in the skin in measurements

of 200 g , 1400 g , 200 g , and 350 g , respectively. To maintain steady state, where

$$\sum \text{mass in} = \sum \text{mass out} = 0$$

what amount of H_2O must be metabolically produced?

1.10 Water accounts for roughly 60% of total body weight. Assuming it can be categorized into six regions, the percentages go as follows. Plasma claims 4.5% of the body weight and is 7.5% of the total body water. Dense connective tissue and cartilage occupies 4.5% of the total body weight and 7.5% of the total body water. Interstitial lymph is 12% of the body weight, which is 20% of the total body water. Inaccessible bone water is also roughly 7.5% of the total body weight and 4.5% total body weight. If transcellular water is 1.5% of the total body weight and total intracellular water is 55% of the total body weight, what percent of total body weight must the intracellular water be, and what percent of total body water must the transcellular water be?

MATLAB Fundamentals

CHAPTER OBJECTIVES

The primary objective of this chapter is to provide an introduction and overview of how MATLAB's command mode is used to implement interactive computations. Specific objectives and topics covered are

- Learning how real and complex numbers are assigned to variables
- Learning how vectors and matrices are assigned values using simple assignment, the colon operator, and the `linspace` and `logspace` functions.
- Understanding the priority rules for constructing mathematical expressions.
- Gaining a general understanding of built-in functions and how you can learn more about them with MATLAB's Help facilities.
- Learning how to use vectors to create a simple line plot based on an equation.

MATLAB is a computer program that provides the user with a convenient environment for performing many types of calculations. In particular, it provides a very nice tool to implement numerical methods.

The most common way to operate MATLAB is by entering commands one at a time in the command window. In this chapter, we use this interactive mode to introduce you to common operations such as performing calculations and creating plots. In Chap. 3, we show how such commands can be used to create MATLAB programs.

One further note. This chapter has been written as a hands-on exercise. That is, you should read it while sitting in front of your computer. The most efficient way to become proficient is to actually implement the commands on MATLAB as you proceed through the following material.

2.1 THE MATLAB ENVIRONMENT

MATLAB uses three primary windows:

- Command window. Used to enter commands and data.
- Graphics window. Used to display plots and graphs.
- Edit window. Used to create and edit M-files.

In this chapter, we will make use of the command and graphics windows. In Chap. 3 we will use the edit window to create M-files.

After starting MATLAB, the command window will open with the command prompt being displayed

```
>>
```

The command mode of MATLAB operates in a sequential fashion as you type in commands line by line. For each command, you get a result. Thus, you can think of it as operating like a very fancy calculator. For example, if you type in

```
>> 55 - 16
```

MATLAB will display the result¹

```
ans =  
    39
```

Notice that MATLAB has automatically assigned the answer to a variable, `ans`. Thus, you could now use `ans` in a subsequent calculation:

```
>> ans + 11
```

with the result

```
ans =  
    50
```

MATLAB assigns the result to `ans` whenever you do not explicitly assign the calculation to a variable of your own choosing.

2.2 ASSIGNMENT

Assignment refers to assigning values to variable names. This results in the storage of the values in the memory location corresponding to the variable name.

2.2.1 Scalars

The assignment of values to scalar variables is similar to other computer languages. Try typing

```
>> a = 4
```

¹Note that MATLAB skips a line between the label (`ans =`) and the number (`39`). Here, we omit such blank lines for conciseness.

2.2 ASSIGNMENT

17

Note how the assignment echo prints to confirm what you have done:

```
a =  
4
```

Echo printing is a characteristic of MATLAB. It can be suppressed by terminating the command line with the semicolon (;) character. Try typing

```
>> A = 6;
```

You can type several commands on the same line by separating them with commas or semicolons. If you separate them with commas, they will be displayed, and if you use the semicolon, they will not. For example,

```
>> a = 4, A = 6; x = 1;  
  
a =  
4
```

MATLAB treats names in a case-sensitive manner—that is, the name *a* is not the same as the name *A*. To illustrate this, enter

```
>> a
```

and then enter

```
>> A
```

See how their values are distinct. They are distinct names.

We can assign complex values to variables, since MATLAB handles complex arithmetic automatically. The unit imaginary number $\sqrt{-1}$ is preassigned to the variable *i*. Consequently, a complex value can be assigned simply as in

```
>> x = 2+i*4  
  
x =  
2.0000 + 4.0000i
```

It should be noted that MATLAB allows the symbol *j* to be used to represent the unit imaginary number for input. However, it always uses an *i* for display. For example,

```
>> x = 2+j*4  
  
x =  
2.0000 + 4.0000i
```

There are several predefined variables, for example, *pi*.

```
>> pi  
  
ans =  
3.1416
```

Notice how MATLAB displays four decimal places. If you desire additional precision, enter the following:

```
>> format long
```


Now when `pi` is entered the result is displayed to 15 significant figures:

```
>> pi  
ans =  
3.14159265358979
```

To return to the four decimal version, type

```
>> format short
```

2.2.2 Arrays, Vectors and Matrices

An array is a collection of values that are represented by a single variable name. One-dimensional arrays are called *vectors* and two-dimensional arrays are called *matrices*. The scalars used in Section 2.2.1 are actually a matrix with one row and one column.

Brackets are used to enter arrays in the command mode. For example, a row vector can be assigned as follows:

```
>> a = [ 1 2 3 4 5 ]  
a =  
1      2      3      4      5
```

Note that this assignment overrides the previous assignment of `a = 4`.

In practice, row vectors are rarely used to solve mathematical problems. When we speak of vectors, we usually refer to column vectors, which are more commonly used. A column vector can be entered in several ways. Try them.

```
>> b = [2;4;6;8;10]
```

or

```
>> b = [ 2;  
4;  
6;  
8;  
10 ]
```

or, by transposing a row vector with the `'` operator,

```
>> b = [ 2 4 6 8 10 ]'
```

The result in all three cases will be

```
b =  
2  
4  
6  
8  
10
```

2.2 ASSIGNMENT

19

A matrix of values can be assigned as follows:

```
>> A = [1 2 3 ; 4 5 6 ; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9
```

In addition, the Enter key (carriage return) can be used to separate the rows. For example, in the following case, the Enter key would be struck after the 3, the 6 and the] to assign the matrix:

```
>> A = [1 2 3
        4 5 6
        7 8 9]
```

At any point in a session, a list of all current variables can be obtained by entering the `who` command:

```
>> who

Your variables are:
A    a    ans  b    x
```

or, with more detail, enter the `whos` command:

```
>> whos

Name      Size      Bytes  Class
A          3x3         72  double array
a          1x5         40  double array
ans        1x1          8  double array
b          5x1         40  double array
x          1x1         16  double array (complex)

Grand total is 21 elements using 176 bytes
```

Note that subscript notation can be used to access an individual element of an array. For example, the fourth element of the column vector `b` can be displayed as

```
>> b(4)

ans =

     8
```

For an array, `A(m,n)` selects the element in `m`th row and the `n`th column. For example,

```
>> A(2,3)

ans =

     6
```

There are several built-in functions that can be used to create matrices. For example, the `ones` and `zeros` functions create vectors or matrices filled with ones and zeros,

respectively. Both have two arguments, the first for the number of rows and the second for the number of columns. For example, to create a 2 × 3 matrix of zeros:

```
>> E = zeros(2,3)

E =
     0     0     0
     0     0     0
```

Similarly, the `ones` function can be used to create a row vector of ones:

```
>> u = ones(1,3)

u =
     1     1     1
```

2.2.3 The Colon Operator

The colon operator is a powerful tool for creating and manipulating arrays. If a colon is used to separate two numbers, MATLAB generates the numbers between them using an increment of one:

```
>> t = 1:5

t =
     1     2     3     4     5
```

If colons are used to separate three numbers, MATLAB generates the numbers between the first and third numbers using an increment equal to the second number:

```
>> t = 1:0.5:3

t =
     1.0000     1.5000     2.0000     2.5000     3.0000
```

Note that negative increments can also be used

```
>> t = 10:-1:5

t =
    10     9     8     7     6     5
```

Aside from creating series of numbers, the colon can also be used as a wildcard to select the individual rows and columns of a matrix. When a colon is used in place of a specific subscript, the colon represents the entire row or column. For example, the second row of the matrix `A` can be selected as in

```
>> A(2,:)

ans =
     4     5     6
```

2.3 MATHEMATICAL OPERATIONS

21

We can also use the colon notation to selectively extract a series of elements from within an array. For example, based on the previous definition of the vector t :

```
>> t(2:4)

ans =
     9     8     7
```

2.2.4 The `linspace` and `logspace` Functions

The `linspace` and `logspace` functions provide other handy tools to generate vectors of spaced points. The `linspace` function generates a row vector of equally spaced points. It has the form

```
linspace(·, ·, ·)
```

which generates \cdot points between \cdot and \cdot . For example

```
>> linspace(0,1,6)

ans =
     0     0.2000     0.4000     0.6000     0.8000     1.0000
```

If the \cdot is omitted, the function automatically generates 100 points.

The `logspace` function generates a row vector that is logarithmically equally spaced. It has the form

```
logspace(·, ·, ·)
```

which generates \cdot logarithmically equally spaced points between decades 10^{\cdot} and 10^{\cdot} . For example,

```
>> logspace(-1,2,4)

ans =
     0.1000     1.0000    10.0000   100.0000
```

If \cdot is omitted, it automatically generates 50 points.

2.3 MATHEMATICAL OPERATIONS

Operations with scalar quantities are handled in a straightforward manner, similar to other computer languages. The common operators, in order of priority, are

\wedge	Exponentiation
$-$	Negation
$*$ /	Multiplication and division
\backslash	Left division ²
$+$ $-$	Addition and subtraction

²Left division applies to matrix algebra. It will be discussed in detail later in this book.

These operators will work in calculator fashion. Try

```
>> 2*pi  
ans =  
6.2832
```

Also, scalar real variables can be included:

```
>> y = pi /4;  
>> y ^ 2.45  
ans =  
0.5533
```

Results of calculations can be assigned to a variable, as in the next-to-last example, or simply displayed, as in the last example.

As with other computer calculation, the priority order can be overridden with parentheses. For example, because exponentiation has higher priority than negation, the following result would be obtained:

```
>> y = -4 ^ 2  
y =  
-16
```

Thus, 4 is first squared and then negated. Parentheses can be used to override the priorities as in

```
>> y = (-4) ^ 2  
y =  
16
```

Calculations can also involve complex quantities. Here are some examples that use the values of $x(2 + 4i)$ and $y(16)$ defined previously:

```
>> 3 * x  
ans =  
6.0000 +12.0000i  
>> 1 / x  
ans =  
0.1000 - 0.2000i  
>> x ^ 2  
ans =  
-12.0000 +16.0000i  
>> x + y  
ans =  
18.0000 + 4.0000i
```

The real power of MATLAB is illustrated in its ability to carry out vector-matrix calculations. Although we will describe such calculations in Chap. 7, it is worth introducing some of those manipulations here.

2.3 MATHEMATICAL OPERATIONS

23

The *inner product* of two vectors (dot product) can be calculated using the `*` operator,

```
>> a * b  
  
ans =  
    110
```

and likewise, the *outer product*

```
>> b * a  
  
ans =  
     2     4     6     8    10  
     4     8    12    16    20  
     6    12    18    24    30  
     8    16    24    32    40  
    10    20    30    40    50
```

To further illustrate vector-matrix multiplication, first redefine `a` and `b`:

```
>> a = [1 2 3];
```

and

```
>> b = [4 5 6]';
```

Now, try

```
>> a * A  
  
ans =  
    30    36    42
```

or

```
>> A * b  
  
ans =  
    32  
    77  
   122
```

What happens when the dimensions are not those required by the operations? Try

```
>> A * a
```

MATLAB automatically displays the error message:

```
??? Error using ==> mtimes  
Inner matrix dimensions must agree.
```

Matrix-matrix multiplication is carried out in likewise fashion:

```
>> A * A  
  
ans =  
    30    36    42  
    66    81    96  
   102   126   150
```

Mixed operations with scalars are also possible:

```
>> A/pi  
  
ans =  
    0.3183    0.6366    0.9549  
    1.2732    1.5915    1.9099  
    2.2282    2.5465    2.8648
```

We must always remember that MATLAB will apply the simple arithmetic operators in vector-matrix fashion if possible. At times, you will want to carry out calculations item by item in a matrix or vector. MATLAB provides for that too. For example,

```
>> A ^ 2  
  
ans =  
    30    36    42  
    66    81    96  
   102   126   150
```

results in matrix multiplication of A with itself.

What if you want to square each element of A ? That can be done with

```
>> A .^ 2  
  
ans =  
     1     4     9  
    16    25    36  
    49    64    81
```

The `.` preceding the `^` operator signifies that the operation is to be carried out element by element. The MATLAB manual calls these *array operations*. They are also often referred to as *element-by-element operations*.

MATLAB contains a helpful shortcut for performing calculations that you've already done. Press the up-arrow key. You should get back the last line you typed in.

```
>> A .^ 2
```

Pressing Enter will perform the calculation again. But you can also edit this line. For example, change it to the line below and then press Enter.

```
>> A .^ 3  
  
ans =  
     1     8    27  
    64   125   216  
   343   512   729
```

Using the up-arrow key, you can go back to any command that you entered. Press the up-arrow until you get back the line

```
b * a
```

Alternatively, you can type `b` and press the up-arrow once and it will automatically bring up the last command beginning with the letter `b`. The up-arrow shortcut is a quick way to fix errors without having to retype the entire line.

2.4 USE OF BUILT-IN FUNCTIONS

MATLAB and its Toolboxes have a rich collection of built-in functions. You can use online help to find out more about them. For example, if you want to learn about the `log` function, type in

```
>> help log

LOG      Natural logarithm.
LOG(X) is the natural logarithm of the elements of X.
Complex results are produced if X is not positive.

See also LOG2, LOG10, EXP, LOGM.
```

For a list of all the elementary functions, type

```
>> help elfun
```

One of their important properties of MATLAB's built-in functions is that they will operate directly on vector and matrix quantities. For example, try

```
>> log(A)

ans =
      0      0.6931      1.0986
  1.3863      1.6094      1.7918
  1.9459      2.0794      2.1972
```

and you will see that the natural logarithm function is applied in array style, element by element, to the matrix `A`. Most functions, such as `sqrt`, `abs`, `sin`, `acos`, `tanh`, and `exp`, operate in array fashion. Certain functions, such as exponential and square root, have matrix definitions also. MATLAB will evaluate the matrix version when the letter `m` is appended to the function name. Try

```
>> sqrtm(A)

ans =
  0.4498 + 0.7623i   0.5526 + 0.2068i   0.6555 - 0.3487i
  1.0185 + 0.0842i   1.2515 + 0.0228i   1.4844 - 0.0385i
  1.5873 - 0.5940i   1.9503 - 0.1611i   2.3134 + 0.2717i
```

A common use of functions is to evaluate a formula for a series of arguments. Recall that the velocity of a free-falling bungee jumper can be computed with [Eq. (1.9)]:

$$v = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right)$$

where v is velocity (m/s), g is the acceleration due to gravity (9.81 m/s²), m is mass (kg), c_d is the drag coefficient (kg/m), and t is time (s).

Create a column vector t that contains values from 0 to 20 in steps of 2:

```
>> t = [0:2:20]';  
  
t =  
    0  
    2  
    4  
    6  
    8  
   10  
   12  
   14  
   16  
   18  
   20
```

Check the number of items in the t array with the `length` function:

```
>> length(t)  
  
ans =  
    11
```

Assign values to the parameters:

```
>> g = 9.81; m = 68.1; cd = 0.25;
```

MATLAB allows you to evaluate a formula such as $v = f(t)$, where the formula is computed for each value of the t array, and the result is assigned to a corresponding position in the v array. For our case,

```
>> v = sqrt(g*m/cd)*tanh(sqrt(g*cd/m)*t)  
  
v =  
    0  
  18.7292  
  33.1118  
  42.0762  
  46.9575  
  49.4214  
  50.6175  
  51.1871  
  51.4560  
  51.5823  
  51.6416
```

2.5 GRAPHICS

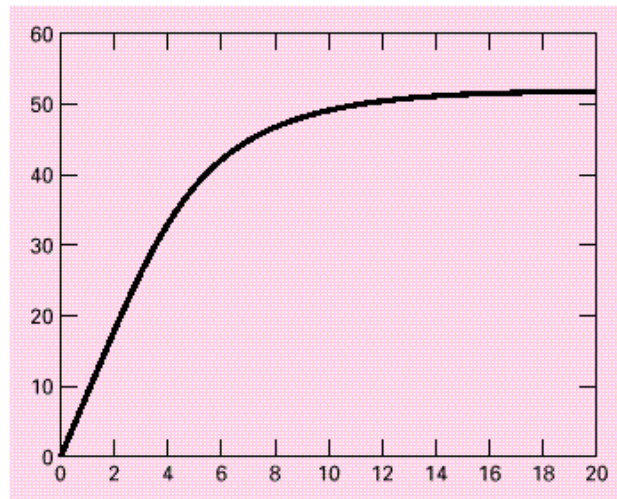
MATLAB allows graphs to be created quickly and conveniently. For example, to create a graph of the t and v arrays from the data above, enter

```
>> plot (t, v)
```

2.5 GRAPHICS

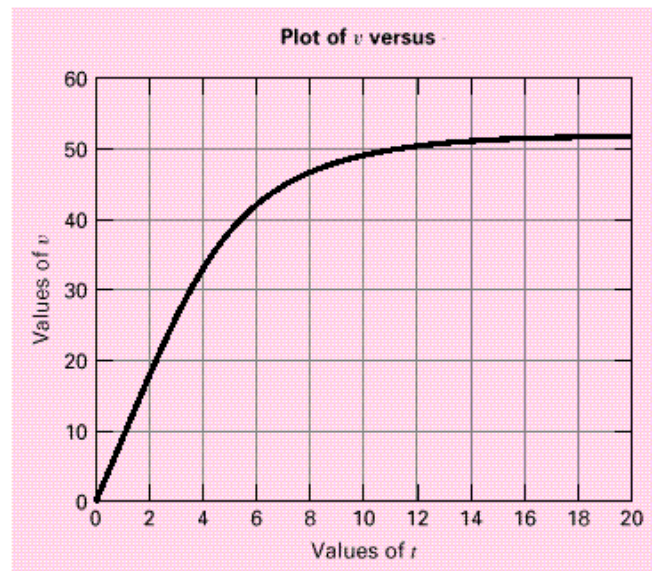
27

The graph appears in the graphics window and can be printed or transferred via the clipboard to other programs.



You can customize the graph a bit with commands such as the following:

```
>> title('Plot of v versus t')  
>> xlabel('Values of t')  
>> ylabel('Values of v')  
>> grid
```



The `plot` command displays a solid line by default. If you want to plot each point with a symbol, you can include a specifier enclosed in single quotes in the `plot` function.

TABLE 2.1 Specifiers for colors, symbols, and line types.

Colors		Symbols		Line Types	
Blue	b	Point	.	Solid	—
Green	g	Circle	o	Dotted	⋯
Red	r	X-mark	x	Dashdot	-.
Cyan	c	Plus	+	Dashed	--
Magenta	m	Star	*		
Yellow	y	Square	s		
Black	k	Diamond	d		
		Triangle down	v		
		Triangle up	^		
		Triangle left	<		
		Triangle right	>		
		Pentagram	p		
		Hexagram	h		

Table 2.1 lists the available specifiers. For example, if you want to use open circles enter

```
>> plot (t, v, 'o')
```

MATLAB allows you to display more than one data set on the same plot. For example, if you want to connect each data marker with a straight line you could type

```
>> plot (t, v, t, v, 'o')
```

There are other features of graphics that are useful—for example, plotting objects instead of lines, families of curves plots, plotting on the complex plane, multiple graphs windows, log-log or semilog plots, three-dimensional mesh plots, and contour plots. As described next, a variety of resources are available to learn about these as well as other MATLAB capabilities.

2.6 OTHER RESOURCES

The foregoing was designed to focus on those features of MATLAB that we will be using in the remainder of this book. As such, it is obviously not a comprehensive overview of all of MATLAB's capabilities. If you are interested in learning more, you should consult one of the excellent books devoted to MATLAB (e.g., Palm, 2004).

Further, the package itself includes an extensive Help facility that can be accessed by clicking on the Help menu in the command window. This will provide you with a number of different options for exploring and searching through MATLAB's Help material. In addition, it provides access to a number of instructive demos.

As described in this chapter, help is also available in interactive mode by typing the `help` command followed by the name of a command or function.

If you do not know the name, you can use the `lookfor` command to search the MATLAB Help files for occurrences of text. For example, suppose that you want to find all the commands and functions that relate to logarithms, you could enter

```
>> lookfor logarithm
```

and MATLAB will display all references that include the word `logarithm`.

PROBLEMS

29

Finally, you can obtain help from The MathWorks, Inc., website at www.mathworks.com. There you will find links to product information, newsgroups, books, and technical support as well as a variety of other useful resources.

PROBLEMS

2.1 A simple electric circuit consisting of a resistor, a capacitor, and an inductor is depicted in Fig. P2.1. The charge on the capacitor $q(t)$ as a function of time can be computed as

$$q(t) = q_0 e^{-Rt/(2L)} \cos \left[\sqrt{\frac{1}{LC}} \cdot \left(\frac{R}{2L} \right) t \right]$$

where t = time, q_0 = the initial charge, R = the resistance, L = inductance, and C = capacitance. Use MATLAB to generate a plot of this function from $t = 0$ to 0.5, given that $q_0 = 10$, $R = 50$, $L = 5$, and $C = 10^{-4}$.

2.2 The standard normal probability density function is a bell-shaped curve that can be represented as

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$$

Use MATLAB to generate a plot of this function from $z = -3$ to 3. Label the ordinate as frequency and the abscissa as z .

2.3 Use the `linspace` function to create vectors identical to the following created with colon notation:

(a) $t = 5:5:30$

(b) $x = -3:3$

2.4 Use colon notation to create vectors identical to the following created with the `linspace` function:

(a) $v = \text{linspace}(-2, 1, 5)$

(b) $r = \text{linspace}(6, 0, 7)$

2.5 If a force F (N) is applied to compress a spring, its displacement x (m) can often be modeled by Hooke's law:

$$F = kx$$

where k = the spring constant (N/m). The potential energy stored in the spring U (J) can then be computed as

$$U = \frac{1}{2} kx^2$$

Five springs are tested and the following data compiled:

k , N	10	12	15	9	12	16
x , m	0.013	0.020	0.009	0.010	0.012	0.010

Use MATLAB to store F and x as vectors and then compute vectors of the spring constants and the potential energies. Use the `max` function to determine the maximum potential energy.

2.6 The density of freshwater can be computed as a function of temperature with the following cubic equation:

$$\rho = 5.5289 \cdot 10^{-8} T_C^3 + 8.5016 \cdot 10^{-6} T_C^2 + 6.5622 \cdot 10^{-5} T_C + 0.99987$$

where ρ = density (g/cm^3) and T_C = temperature ($^{\circ}\text{C}$). Use MATLAB to generate a vector of temperatures ranging from 32°F to 93.2°F using increments of 3.6°F . Convert this vector to degrees Celsius and then compute a vector of densities based on the cubic formula. Create a plot of ρ versus T_C . Recall that $T_C = 5/9(T_F - 32)$.

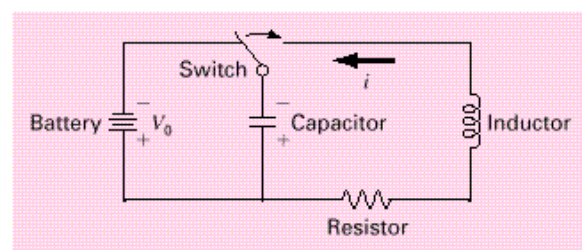
2.7 Manning's equation can be used to compute the velocity of water in a rectangular open channel:

$$U = \frac{\bar{S}}{n} \left(\frac{BH}{B + 2H} \right)^{2/3}$$

where U = velocity (m/s), \bar{S} = channel slope, n = roughness coefficient, B = width (m), and H = depth (m). The following data is available for five channels:

0.035	0.0001	10	2
0.020	0.0002	8	1
0.015	0.0010	20	1.5
0.030	0.0007	24	3
0.022	0.0003	15	2.5

FIGURE P2.1



Store these values in a matrix where each row represents one of the channels and each column represents one of the parameters. Write a single-line MATLAB statement to compute a column vector containing the velocities based on the values in the parameter matrix.

2.8 It is general practice in engineering and science that equations be plotted as lines and discrete data as symbols. Here is some data for concentration (c) versus time (t) for the photodegradation of aqueous bromine:

t , min	10	20	30	40	50	60
c , ppm	3.4	2.6	1.6	1.3	1.0	0.5

This data can be described by the following function:

$$c = 4.84e^{-0.034t}$$

Use MATLAB to create a plot displaying both the data (using square symbols) and the function (using a dashed line). Plot the function for $t = 0$ to 70 min.

2.9 The `semilogy` function operates in an identical fashion to the `plot` function except that a logarithmic (base-10) scale is used for the y axis. Use this function to plot the data and function as described in Prob. 2.8. Explain the results.

2.10 Here is some wind tunnel data for force (F) versus velocity (v):

v , m/s	10	20	30	40	50	60	70	80
F , N	25	70	380	550	610	1220	830	1450

This data can be described by the following function:

$$F = 0.2741v^{1.9842}$$

Use MATLAB to create a plot displaying both the data (using diamond symbols) and the function (using a dotted line). Plot the function for $v = 0$ to 100 m/s.

2.11 The `loglog` function operates in an identical fashion to the `plot` function except that logarithmic scales are used for both the x and y axes. Use this function to plot the data and function as described in Prob. 2.10. Explain the results.

2.12 The Maclaurin series expansion for the cosine is

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

Use MATLAB to create a plot of the cosine (solid line) along with a plot of the series expansion (dashed line) up to and including the term $x^6/6!$. Use the built-in function `factorial` in computing the series expansion. Make the range of the abscissa from $x = 0$ to $3\pi/2$.

3

Programming with MATLAB

CHAPTER OBJECTIVES

The primary objective of this chapter is to learn how to write M-file programs to implement numerical methods. Specific objectives and topics covered are

- Learning how to create well-documented M-files in the edit window and invoke them from the command window.
- Understanding how to set up M-files so that they interactively prompt users for information and display results in the command window.
- Learning how to write clear and well-documented M-files by employing structured programming constructs to implement logic and repetition.
- Understanding what is meant by vectorization and why it is beneficial.
- Understanding how functions can be passed to M-files.

YOU'VE GOT A PROBLEM

In Chap. 1, we used a force balance to develop a mathematical model to predict the fall velocity of a bungee jumper. This model took the form of the following differential equation:

$$\frac{dv}{dt} = g - \frac{c}{m}v^2$$

We also learned that a numerical solution of this equation could be obtained with Euler's method:

$$v_{i+1} = v_i + \frac{dv_i}{dt} \Delta t$$

This equation can be implemented repeatedly to compute velocity as a function of time. However, to obtain good accuracy, many small steps must be taken. This would be

extremely laborious and time consuming to implement by hand. However, with the aid of MATLAB, such calculations can be performed easily.

So our problem now is to figure out how to do this. This chapter will introduce you to how MATLAB M-files can be used to obtain such solutions.

3.1 M-FILES

The most common way to operate MATLAB is by entering commands one at a time in the command window. M-files provide an alternative way of performing operations that greatly expand MATLAB's problem-solving capabilities. An *M-file* contains a series of statements that can be run all at once. Note that the nomenclature "M-file" comes from the fact that such files are stored with a `.m` extension. M-files come in two flavors: script files and function files:

3.1.1 Script Files

A *script file* is merely a series of MATLAB commands that are saved on a file. The script can be executed by typing the file name in the command window or by invoking the menu selections in the edit window: **Debug, Run**.

EXAMPLE 3.1 Script File

Problem Statement. Develop a script file to compute the velocity of the free-falling bungee jumper.

Solution. Open the editor with the menu selection: **File, New, M-file**. Type in the following statements to compute the velocity of the free-falling bungee jumper at a specific time [recall Eq. (1.9)]:

```
g = 9.81; m = 68.1; t = 12; cd = 0.25;  
v = sqrt(g * m / cd) * tanh(sqrt(g * cd / m) * t)
```

Save the file as `scriptdemo.m`. Return to the command window and type in

```
>>scriptdemo
```

The result will be displayed as

```
v =  
50.6175
```

3.1.2 Function Files

Function files are M-files that start with the word `function`. In contrast to script files, they can accept input arguments and return outputs. Hence they are analogous to user-defined functions in programming languages such as Fortran, Visual Basic or C.

The syntax for the function file can be represented generally as

```
function outvar = funcname(arglist)  
% helpcomments  
statements  
outvar = value
```


where *outvar* = the name of the output variable, *funcname* = the function's name, *arglist* = the function's argument list (i.e., comma-delimited values that are passed into the function), *helpcomments* = text that provides the user with information regarding the function (these can be invoked by typing `Help funcname` in the command window), and *statements* = MATLAB statements that compute the *value* that is assigned to *outvar*.

The M-file should be saved as *funcname.m*. The function can then be run by typing *funcname* in the command window as illustrated in the following example. Note that even though MATLAB is case-sensitive, your computer's operating system may not be. Whereas MATLAB would treat function names like `freefallvel` and `FreeFallVel` as two different variables, your operating system might not.

EXAMPLE 3.2 Function File

Problem Statement. As in Example 3.1, compute the velocity of the free-falling bungee jumper, but now we will use a function file for the task.

Solution. Type the following statements in the file editor:

```
function velocity = freefallvel(m, cd, t)
% freefallvel(m, cd, t) computes the free-fall velocity (mps)
%                               of an object with second-order drag.
% input:
%   m = mass (kg)
%   cd = second-order drag coefficient (kg/m)
%   t = time (s)
% output:
%   velocity = downward velocity (m/s)

g = 9.81; % acceleration of gravity
velocity = sqrt(g * m / cd) * tanh(sqrt(g * cd / m) * t);
```

Save the file as `freefallvel.m`. To invoke the function, return to the command window and type in

```
>> freefallvel(68.1,0.25,12)
```

The result will be displayed as

```
ans =
    50.6175
```

One advantage of a function M-file is that it can be invoked repeatedly for different argument values. Suppose that we wanted to compute the velocity of a 100-kg jumper after 8 s:

```
>> freefallvel(100,0.25,8)
ans =
    53.1878
```

To invoke the help comments type

```
>> help freefallvel
```

which results in the comments being displayed

```
freefallvel(m, cd, t) computes the free-fall velocity (mps)
                        of an object with second-order drag.
input:
  m = mass (kg)
  cd = second-order drag coefficient (kg/m)
  t = time (s)
output:
  velocity = downward velocity (m/s)
```

Function M-files can return more than one result. In such cases, the variables containing the results are comma-delimited and enclosed in brackets. For example, the following function, `stats.m`, computes the mean and the standard deviation of a vector:

```
function [mean, stdev] = stats(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/(n-1)));
```

Here is an example of how it can be applied:

```
>> y = [8 5 10 12 6 7.5 4];
>> [m,s] = stats(y)
m =
    7.5000
s =
    2.8137
```

Because script M-files have limited utility, function M-files will be our primary programming tool for the remainder of this book. Hence, we will often refer to function M-files as simply M-files.

3.2 INPUT-OUTPUT

As in Section 3.1, information is passed into the function via the argument list and is output via the function's name. Two other functions provide ways to enter and display information directly using the command window.

The `input` Function. This function allows you to prompt the user for values directly from the command window. Its syntax is

```
n = input('promptstring')
```

The function displays the *promptstring*, waits for keyboard input, and then returns the value from the keyboard. For example,

```
m = input('Mass (kg): ')
```

When this line is executed, the user is prompted with the message

```
Mass (kg):
```

If the user enters a value, it would then be assigned to the variable `m`.

3.2 INPUT-OUTPUT

35

The `input` function can also return user input as a string. To do this, an 's' is appended to the function's argument list. For example,

```
name = input('Enter your name: ','s')
```

The `disp` Function. This function provides a handy way to display a value. Its syntax is

```
disp(value)
```

where *value* = the value you would like to display. It can be a numeric constant or variable, or a string message enclosed in hyphens. Its application is illustrated in the following example.

EXAMPLE 3.3 An Interactive M-File Function

Problem Statement. As in Example 3.2, compute the velocity of the free-falling bungee jumper, but now use the `input` and `disp` functions for input/output.

Solution. Type the following statements in the file editor:

```
function velocity = freefallinteract
% freefallinteract()
%   computes the free-fall velocity of a bungee jumper
%   with second-order drag.
% input: interactive from command window
% output:
%   velocity = downward velocity (m/s)

g = 9.81;      % acceleration of gravity
m = input('Mass (kg): ');
cd = input('Drag coefficient (kg/m): ');
t = input('Time (s): ');
disp(' ')
disp('Velocity (m/s):')
disp(sqrt(g * m / cd) * tanh(sqrt(g * cd / m) * t))
```

Save the file as `freefallinteract.m`. To invoke the function, return to the command window and type in

```
>> freefallinteract

Mass (kg): 68.1
Drag coefficient (kg/m): 0.25
Time (s): 12

Velocity (m/s):
    50.6175
```

Note that functions can call functions. For example, the M-file in the preceding example could have been written as (without comments)

```
function velocity = freefallfunctfunc
g = 9.81;      % acceleration of gravity
```

```
m = input('Mass (kg): ');
cd = input('Drag coefficient (kg/m): ');
t = input('Time (s): ');
disp(' ')
disp('Velocity (m/s):')
disp(vel(g,m,cd,t))

function v = vel(g,m,cd,t)
v = sqrt(g * m / cd) * tanh(sqrt(g * cd / m) * t);
```

The fprintf Function. This function provides additional control over the display of information. A simple representation of its syntax is

```
fprintf('format', x, ...)
```

where *format* is a string specifying how you want the value of the variable *x* to be displayed. The operation of this function is best illustrated by examples.

A simple example would be to display a value along with a message. For instance, suppose that the variable *velocity* has a value of 50.6175. To display the value using eight digits with four digits to the right of the decimal point along with a message, the statement along with the resulting output would be

```
>> fprintf('The velocity is %8.4f m/s\n', velocity)
The velocity is 50.6175 m/s
```

This example should make it clear how the format string works. MATLAB starts at the left end of the string and displays the labels until it detects one of the symbols: % or \. In our example, it first encounters a % and recognizes that the following text is a format code. As in Table 3.1, the *format codes* allow you to specify whether numeric values are displayed in integer, decimal, or scientific format. After displaying the value of *velocity*, MATLAB continues displaying the character information (in our case the units: m/s) until it detects the symbol \. This tells MATLAB that the following text is a control code. As in Table 3.1, the *control codes* provide a means to perform actions such as skipping to the next line. If we had omitted the code \n in the previous example, the command prompt would appear at the end of the label m/s rather than on the next line as would typically be desired.

TABLE 3.1 Commonly used format and control codes employed with the fprintf function.

Format Code	Description
%d	Integer format
%e	Scientific format with lowercase e
%E	Scientific format with uppercase E
%f	Decimal format
%g	The more compact of %e or %f
Control Code	Description
\n	Start new line
\t	Tab

3.3 STRUCTURED PROGRAMMING

37

The `fprintf` function can also be used to display several values per line with different formats. For example,

```
>> fprintf('%5d %10.3f %8.5e\n',100,2*pi,pi);  
100      6.283 3.14159e+000
```

It can also be used to display vectors and matrices. Here is an M-file that enters two sets of values as vectors. These vectors are then combined into a matrix, which is then displayed as a table with headings:

```
function fprintfdemo  
x = [1 2 3 4 5];  
y = [20.4 12.6 17.8 88.7 120.4];  
z = [x;y];  
fprintf('      x      y\n');  
fprintf('%5d %10.3f\n',z);
```

The result of running this M-file is

```
>> fprintfdemo  
      x      y  
1      20.400  
2      12.600  
3      17.800  
4      88.700  
5     120.400
```

3.3 STRUCTURED PROGRAMMING

The simplest of all M-files perform instructions sequentially. That is, the program statements are executed line by line starting at the top of the function and moving down to the end. Because a strict sequence is highly limiting, all computer languages include statements allowing programs to take nonsequential paths. These can be classified as

- *Decisions* (or Selection). The branching of flow based on a decision.
- *Loops* (or Repetition). The looping of flow to allow statements to be repeated.

3.3.1 Decisions

The *if* Structure. This structure allows you to execute a set of statements if a logical condition is true. Its general syntax is

```
if condition  
    statements  
end
```

where *condition* is a logical expression that is either true or false. For example, here is a simple M-file to evaluate whether a grade is passing:

```
function gout = grader(grade)  
% grader(grade):  
% determines whether grade is passing  
% input:  
% grade = numerical value of grade (0-100)
```

```
% output:
%   displayed message
if grade >= 60
    disp('passing grade:')
    disp(grade)
end
```

The following illustrates the result

```
>> grader(95.6)

passing grade:
    95.6000
```

For cases where only one statement is executed, it is often convenient to implement the `if` structure as a single line,

```
if grade > 60, disp('passing grade:'), end
```

This structure is called a *single-line if*. For cases where more than one statement is implemented, the multiline `if` structure is usually preferable because it is easier to read.

Error Function. A nice example of the utility of a single-line `if` is to employ it for rudimentary error trapping. This involves using the `error` function which has the syntax,

```
error(msg)
```

When this function is encountered, it displays the text message `msg` and causes the M-file to terminate and return to the command window.

An example of its use would be where we might want to terminate an M-file to avoid a division by zero. The following M-file illustrates how this could be done:

```
function f = errortest(x)
if x == 0, error('zero value encountered'), end
f = 1/x;
```

If a nonzero argument is used, the division would be implemented successfully as in

```
>> errortest(10)
ans =
    0.1000
```

However, for a zero argument, the function would terminate prior to the division and the error message would be displayed in red typeface:

```
>> errortest(0)
??? Error using ==> errortest
zero value encountered
```

Logical Conditions. The simplest form of the *condition* is a single relational expression that compares two values as in

```
value1 relation value2
```

where the *values* can be constants, variables, or expressions and the *relation* is one of the relational operators listed in Table 3.2.

TABLE 3.2 Summary of relational operators in MATLAB.

Example	Operator	Relationship
<code>x == 0</code>	<code>==</code>	Equal
<code>unit ~= 'm'</code>	<code>~=</code>	Not equal
<code>a < 0</code>	<code><</code>	less than
<code>s > t</code>	<code>></code>	Greater than
<code>3.9 <= a/3</code>	<code><=</code>	Less than or equal to
<code>r >= 0</code>	<code>>=</code>	Greater than or equal to

MATLAB also allows testing of more than one logical condition by employing logical operators. We will emphasize the following:

- `~` (*Not*). Used to perform logical negation on an expression.

`~expression`

If the *expression* is true, the result is false. Conversely, if the *expression* is false, the result is true.

- `&` (*And*). Used to perform a logical conjunction on two expressions.

`expression1 & expression2`

If both *expressions* evaluate to true, the result is true. If either or both *expressions* evaluates to false, the result is false.

- `|` (*Or*). Used to perform a logical disjunction on two expressions.

`expression1 | expression2`

If either or both *expressions* evaluate to true, the result is true.

Table 3.3 summarizes all possible outcomes for each of these operators. Just as for arithmetic operations, there is a priority order for evaluating logical operations. These are from highest to lowest: `~`, `&` and `|`. In choosing between operators of equal priority, MATLAB evaluates them from left to right. Finally, as with arithmetic operators, parentheses can be used to override the priority order.

Let's investigate how the computer employs the priorities to evaluate a logical expression. If `a = -1`, `b = 2`, `x = 1`, and `y = 'b'`, evaluate whether the following is true or false:

`a * b > 0 & b == 2 & x > 7 | ~(y > 'd')`

TABLE 3.3 A truth table summarizing the possible outcomes for logical operators employed in MATLAB. The order of priority of the operators is shown at the top of the table.

x	y	Highest			Lowest
		<code>~x</code>	<code>x & y</code>	<code>x y</code>	
T	T	F	T	T	T
T	F	F	F	T	T
F	T	T	F	T	F
F	F	T	F	F	F

To make it easier to evaluate, substitute the values for the variables:

```
-1 * 2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
```

The first thing that MATLAB does is to evaluate any mathematical expressions. In this example, there is only one: $-1 * 2$,

```
-2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
```

Next, evaluate all the relational expressions

```
    F    &    T    &    F    |    ~    F
-2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
```

At this point, the logical operators are evaluated in priority order. Since the \sim has highest priority, the last expression ($\sim F$) is evaluated first to give

```
F & T & F | T
```

The $\&$ operator is evaluated next. Since there are two, the left-to-right rule is applied and the first expression ($F \& T$) is evaluated:

```
F & F | T
```

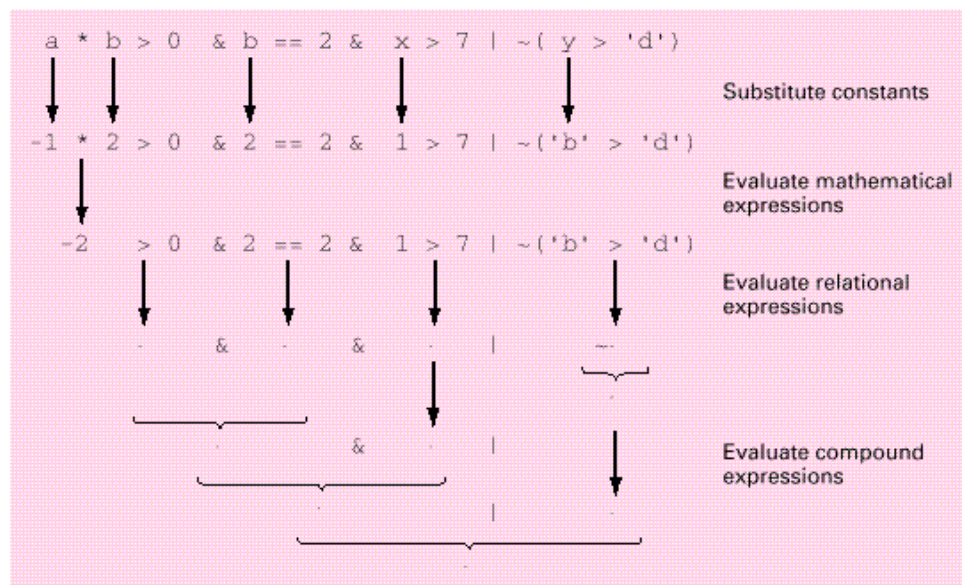
The $\&$ again has highest priority

```
F | T
```

Finally, the $|$ is evaluated as true. The entire process is depicted in Fig. 3.1.

FIGURE 3.1

A step-by-step evaluation of a complex decision.



3.3 STRUCTURED PROGRAMMING

41

The if...else Structure. This structure allows you to execute a set of statements if a logical condition is true and to execute a second set if the condition is false. Its general syntax is

```
if condition
    statements1
else
    statements2
end
```

The if...elseif Structure. It often happens that the false option of an if...else structure is another decision. This type of structure often occurs when we have more than two options for a particular problem setting. For such cases, a special form of decision structure, the if...elseif has been developed. It has the general syntax

```
if condition1
    statements1
elseif condition2
    statements2
elseif condition3
    statements3
.
.
.
else
    statementselse
end
```

EXAMPLE 3.4 if Structures

Problem Statement. For a scalar, the built-in MATLAB `sign` function returns the sign of its argument (-1, 0, 1). Here's a MATLAB session that illustrates how it works:

```
>> sign(25.6)

ans =
     1

>> sign(-0.776)

ans =
    -1

>> sign(0)

ans =
     0
```

Develop an M-file to perform the same function.

Solution. First, an `if` structure can be used to return 1 if the argument is positive:

```
function sgn = mysign(x)
% mysign(x) returns 1 if x is greater than zero.
if x > 0
    sgn = 1;
end
```

This function can be run as

```
>> mysign(25.6)

ans =
     1
```

Although the function handles positive numbers correctly, if it is run with a negative or zero argument, nothing is displayed. To partially remedy this shortcoming, an `if...else` structure can be used to display -1 if the condition is false:

```
function sgn = mysign(x)
% mysign(x) returns 1 if x is greater than zero.
%                               -1 if x is less than or equal to zero
if x > 0
    sgn = 1;
else
    sgn = -1;
end
```

This function can be run as

```
>> mysign(-0.776)

ans =
    -1
```

Although the positive and negative cases are now handled properly, -1 is erroneously returned if a zero argument is used. An `if...elseif` structure can be used to incorporate this final case:

```
function sgn = mysign(x)
% mysign(x) returns 1 if x is greater than zero.
%                               -1 if x is less than zero.
%                               0 if x is equal to zero.
if x > 0
    sgn = 1;
elseif x < 0
    sgn = -1;
else
    sgn = 0;
end
```

The function now handles all possible cases. For example,

```
>> mysign(0)

ans =
     0
```

3.3.2 Loops

As the name implies, loops perform operations repetitively. There are two types of loops, depending on how the repetitions are terminated. A *for loop* ends after a specified number of repetitions. A *while loop* ends on the basis of a logical condition.

The *for* Structure. A *for* loop repeats statements a specific number of times. Its general syntax is

```
for index = start:step:finish
    statements
end
```

The *for* loop operates as follows. The *index* is a variable that is set at an initial value, *start*. The program then compares the *index* with a desired final value, *finish*. If the *index* is less than or equal to the *finish*, the program executes the *statements*. When it reaches the *end* line that marks the end of the loop, the *index* variable is increased by the *step* and the program loops back up to the *for* statement. The process continues until the *index* becomes greater than the *finish* value. At this point, the loop terminates as the program skips down to the line immediately following the *end* statement.

Note that if an increment of 1 is desired (as is often the case), the *step* can be dropped. For example,

```
for i = 1:5
    disp(i)
end
```

When this executes, MATLAB would display in succession, 1, 2, 3, 4, 5. In other words, the default *step* is 1.

The size of the *step* can be changed from the default of 1 to any other numeric value. It does not have to be an integer, nor does it have to be positive. For example, step sizes of 0.2, -1, or -5, are all acceptable.

If a negative *step* is used, the loop will “countdown” in reverse. For such cases, the loop’s logic is reversed. Thus, the *finish* is less than the *start* and the loop terminates when the *index* is less than the *finish*. For example,

```
for j = 10:-1:1
    disp(j)
end
```

When this executes, MATLAB would display the classic “countdown” sequence: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

EXAMPLE 3.5 Using a `for` Loop to Compute the Factorial

Problem Statement. The factorial is computed as

$$\begin{aligned}0! &= 1 \\1! &= 1 \\2! &= 1 \cdot 2 = 2 \\3! &= 1 \cdot 2 \cdot 3 = 6 \\4! &= 1 \cdot 2 \cdot 3 \cdot 4 = 24 \\5! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \\&\vdots\end{aligned}$$

Develop an M-file to compute the factorial.¹

Solution. A simple function to implement this calculation can be developed as

```
function fout = factor(n)
% factor(n):
%   Computes the product of all the integers from 1 to n.
x = 1;
for i = 1:n
    x = x * i;
end
fout = x;
end
```

which can be run as

```
>> factor(5)

ans =
    120
```

This loop will execute 5 times (from 1 to 5). At the end of the process, `x` will hold a value of 5! (meaning 5 factorial or $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$).

Vectorization. The `for` loop is easy to implement and understand. However, for MATLAB, it is not necessarily the most efficient means to repeat statements a specific number of times. Because of MATLAB's ability to operate directly on arrays, *vectorization* provides a much more efficient option. For example, the following `for` structure:

```
i = 0;
for t = 0:0.02:50
    i = i + 1;
    y(i) = cos(t);
end
```

¹Note that MATLAB has a built-in function `factorial` that performs this computation.

3.3 STRUCTURED PROGRAMMING

45

can be represented in vectorized form as

```
t = 0:0.02:50;  
y = 5 * cos(t);
```

It should be noted that for more complex code, it may not be obvious how to vectorize the code. That said, wherever possible, vectorization is recommended.

Preallocation of Memory. MATLAB automatically increases the size of arrays every time you add a new element. This can become time consuming when you perform actions such as adding new values one at a time within a loop. For example, here is some code that sets value of elements of y depending on whether or not values of t are greater than one:

```
t = 0:.01:5;  
for i = 1:length(t)  
    if t(i)>1  
        y(i) = 1/t(i);  
    else  
        y(i) = 1;  
    end  
end
```

For this case, MATLAB must resize y every time a new value is determined. The following code preallocates the proper amount of memory by using a vectorized statement to assign ones to y prior to entering the loop.

```
t = 0:.01:5;  
y = ones(size(t));  
for i = 1:length(t)  
    if t(i)>1  
        y(i) = 1/t(i);  
    end  
end
```

Thus, the array is only sized once. In addition, preallocation helps reduce memory fragmentation, which also enhances efficiency.

The while Structure. A while loop repeats as long as a logical condition is true. Its general syntax is

```
while condition  
    statements  
end
```

The *statements* between the *while* and the *end* are repeated as long as the *condition* is true. A simple example is

```
function fout = whiledemo()  
x = 8  
while x > 0  
    x = x - 3;  
    disp(x)  
end
```

This function can be run as

```
>> whiledemo
x =
     8
     5
     2
    -1
```

The `while...break` Structure. Although the `while` structure is extremely useful, the fact that it always exits at the beginning of the structure on a false result is somewhat constraining. For this reason, languages such as Fortran 90 and Visual Basic have special structures that allow loop termination on a true condition anywhere in the loop. Although such structures are currently not available in MATLAB, their functionality can be mimicked by a special version of the `while` loop. The syntax of this version, called a *while...break structure*, can be written as

```
while (1)
    statements
    if condition, break, end
    statements
end
```

where the *condition* is a logical condition that tests true or false. Thus, a single line `if` is used to exit the loop if the condition tests true. Note that as shown, the `break` can be placed in the middle of the loop (i.e., with statements before and after it). Such a structure is called a *midtest loop*.

If the problem required it, we could place the `break` at the very beginning to create a *pretest loop*. An example is

```
while (1)
    If x < 0, break, end
    x = x - 5
end
```

Notice how 5 is subtracted from `x` on each iteration. This represents a mechanism so that the loop eventually terminates. Every decision loop must have such a mechanism. Otherwise it would repeat ad infinitum.

Alternatively, we could also place the `if...break` statement at the very end and create a *posttest loop*,

```
while (1)
    x = x - 5
    if x < 0, break, end
end
```

It should be clear that, in fact, all three structures are really the same. That is, depending on where we put the exit (beginning, middle, or end) dictates whether we have a pre-, mid- or posttest. It is this simplicity that led the computer scientists who developed Fortran 90 and Visual Basic to favor this structure over other forms of the decision loop such as the conventional `while` structure.

3.4 NESTING AND INDENTATION

We need to understand that structures can be “nested” within each other. *Nesting* refers to placing structures within other structures. As in the following example, a good example is to determine the roots of the quadratic equation.

EXAMPLE 3.6 Nesting Structures

Problem Statement. The roots of a quadratic equation

$$f(x) = ax^2 + bx + c$$

can be determined with the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Develop a function to implement this formula given values of the coefficients.

Solution. *Top-down design* provides a nice approach for designing an algorithm to compute the roots. This involves programming up the general structure without details and then refining the algorithm. To start, we first recognize that depending on whether the parameter a is zero, we will either have “weird” cases (e.g., single roots or trivial values) or conventional cases using the quadratic formula. This “big-picture” version can be programmed as

```
function quad = quadroots(a, b, c)
% quadroots(a, b, c):
%   computes real and complex roots of quadratic equation
% input:
%   a = second-order coefficient
%   b = first-order coefficient
%   c = zero-order coefficient
% output:
%   r1 = real part of first root
%   i1 = imaginary part of first root
%   r2 = real part of second root
%   i2 = imaginary part of second root

if a == 0
    %weird cases
else
    %quadratic formula
end
```

Next, we develop refined code to handle the “weird” cases:

```
%weird cases
if b ~= 0
    %single root
    r1 = -c / b
else
    %trivial solution
    error('Trivial solution. Reenter data')
end
```

And we can develop refined code to handle the quadratic formula cases:

```
%quadratic formula
d = b ^ 2 - 4 * a * c
if d >= 0
    %real roots
    r1 = (-b + sqrt(d)) / (2 * a)
    r2 = (-b - sqrt(d)) / (2 * a)
else
    %complex roots
    r1 = -b / (2 * a)
    r2 = r1
    i1 = sqrt(abs(d)) / (2 * a)
    i2 = -i1
end
```

We can then merely substitute these blocks back into the simple “big-picture” framework to give the final result:

```
function quad = quadroots(a, b, c)
% quadroots(a, b, c):
%   computes real and complex roots of quadratic equation
% input:
%   a = second-order coefficient
%   b = first-order coefficient
%   c = zero-order coefficient
% output:
%   r1 = real part of first root
%   i1 = imaginary part of first root
%   r2 = real part of second root
%   i2 = imaginary part of second root
if a == 0
    %weird cases
    if b ~= 0
        %single root
        r1 = -c / b
    else
        %trivial solution
        error('Trivial solution. Try again')
    end
else
    %quadratic formula
    d = b ^ 2 - 4 * a * c    %discriminant
    if d >= 0
        %real roots
        r1 = (-b + sqrt(d)) / (2 * a)
        r2 = (-b - sqrt(d)) / (2 * a)
    else
        %complex roots
        r1 = -b / (2 * a)
```

```
    r2 = r1
    i1 = sqrt(abs(d)) / (2 * a)
    i2 = -i1
end
```

end

As highlighted by the shading, notice how indentation helps to make the underlying logical structure clear. Also notice how “modular” the structures are. Here is a command window session illustrating how the function performs:

```
>> quadroots(1,1,1)
r1 =
    -0.5000
i1 =
     0.8660
r2 =
    -0.5000
i2 =
    -0.8660

>> quadroots(1,5,1)
r1 =
    -0.2087
r2 =
   -4.7913

>> quadroots(0,5,1)
r1 =
    -0.2000

>> quadroots(0,0,0)
??? Error using ==> quadroots
Trivial solution. Try again
```

3.5 PASSING FUNCTIONS TO M-FILES

There are many occasions when you would like a function to perform calculations using an arbitrary function. The built-in `feval` and `inline` functions can be used to accomplish this task.

The `feval` Function. The `feval` function provides an alternative means to evaluate a function. It has the syntax

```
outvar = feval('funcname', arg1, arg2, ...)
```

where the function `funcname` is evaluated at the values of the arguments, `arg`. Here is an example:

```
>> outvar = feval('cos', pi/6)
```

which evaluates to

```
outvar =
     0.8660
```

Note that this statement is equivalent to merely writing

```
>> outvar = cos(pi/6)
```

The inline Function. The inline function provides a way to create a one-line function that does not have to be stored in a separate M-file. It has the syntax

```
funcname = inline('expression', var1, var2, ... )
```

where *funcname* = the function's name, *expression* = the mathematical formula that is to be evaluated, and the *var*'s are names of the variables in the expression. If no *var*'s are included, a single variable *x* is assumed. Here's an example of its application:

```
>> fx = inline('cos(x)*sin(x)')  
fx =  
    Inline function:  
    fx(x) = cos(x)*sin(x)  
  
>> fx(pi/6)  
ans =  
    0.4330
```

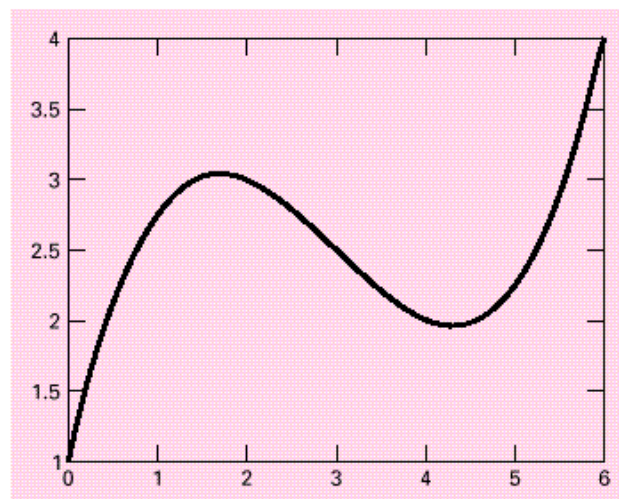
EXAMPLE 3.7 Passing Functions to M-files

Problem Statement. Develop an M-file to determine the average value of a function over a range. Illustrate its use for the simple cubic equation:

$$f(x) = 0.125x^3 - 1.125x^2 + 2.75x + 1$$

Solution. This function can be plotted in MATLAB over the range from $x = 0$ to 6 with the commands

```
>> x = [0:0.1:6];  
>> f = 0.125*x.^3 - 1.125*x.^2 + 2.75*x + 1;  
>> plot(x,f)
```



3.5 PASSING FUNCTIONS TO M-FILES

51

The average value of the function can be computed with standard MATLAB commands as

```
>> mean(f)
```

```
ans =  
    2.5000
```

Inspection of the plot shows that this result makes sense.

We can write an M-file to perform the same computation:

```
function favg = funcavg (a,b,n)  
% funcavg (a,b,n):  
%   average value of function  
% input:  
%   a = lower bound  
%   b = upper bound  
%   n = number of intervals  
% output:  
%   favg = average value of function  
x = linspace(a,b,n);  
y = feval('func',x);  
favg = mean(y);
```

where `func` is another function that is written to evaluate the cubic

```
function f = func(x)  
f = 0.125*x.^3 - 1.125*x.^2 + 2.75*x + 1;
```

The `funcavg` function can be run from the command window as

```
>> funcavg(0,6,60)
```

```
ans =  
    2.5000
```

Now let's rewrite the M-file so that rather than being specific to `func`, it evaluates a generic function name `f` that is passed in as an argument:

```
function favg = funcavg(f,a,b,n)  
% funcavg average value of function  
% funcavg (f,a,b,n):  
%   average value of function  
% input:  
%   f = function to be evaluated  
%   a = lower bound  
%   b = upper bound  
%   n = number of intervals  
% output:  
%   favg = average value of function
```

```
x = linspace(a,b,n);  
y = feval(f,x);  
favg = mean(y);
```

This can be run from the command window as

```
>> funcavg('func',0,6,60)  
  
ans =  
    2.5000
```

To demonstrate the generic nature of this version, we can use `funcavg` to determine the average value of the built-in `sin` function between 0 and 2π as

```
>> funcavg('sin',0,2*pi,180)  
  
ans =  
-3.2530e-015
```

Does this result make sense?

Finally, the real beauty of this approach is demonstrated by using the `inline` function to pass the cubic

```
>> funcavg(inline('0.125*x.^3-1.125*x.^2+2.75*x+1'),0,6,60)  
  
ans =  
    2.5000
```

Thus, we can see that `funcavg` can now evaluate any function using the `inline` command. We will do this on numerous occasions throughout the remainder of this text.

3.6 MATLAB M-FILE: BUNGEE JUMPER VELOCITY

In this section, we will use MATLAB to solve the free-falling bungee jumper problem we posed at the beginning of this chapter. This involves obtaining a solution of

$$\frac{dv}{dt} = g - \frac{c}{m}v^2$$

Recall that, given an initial condition for time and velocity, the problem involved iteratively solving the formula,

$$v_{i+1} = v_i + \frac{dv_i}{dt} \Delta t$$

Now also remember that if we desired to attain good accuracy, we would employ small steps. Therefore, we would probably want to apply the formula repeatedly to step out from our initial time to attain the value at the final time. Consequently, an algorithm to solve the problem would be based on a loop.

For example, suppose that we started the computation at $t = 0$ and wanted to predict velocity at $t = 12$ s using a time step of $\Delta t = 0.5$ s. We would therefore need to apply the

3.6 MATLAB M-FILE: BUNGEE JUMPER VELOCITY

53

iterative equation 24 times—that is,

$$n = \frac{12}{0.5} = 24$$

where n = the number of iterations of the loop. Because this result is exact (i.e., the ratio is an integer), we can use a `for` loop as the basis for the algorithm. Here's an M-file to do this:

```
function vend = velocity1(dt, ti, tf, vi, m, cd)
% velocity1(dt, ti, tf, vi, m, cd):
%   euler's method solution of freefalling bungee jumper
%   using a for loop
% input:
%   dt = delta t (s)
%   ti = initial time (s)
%   tf = final time (s)
%   vi = initial value of dependent variable (m/s)
%   m = mass(kg)
%   cd = second-order drag coefficient(kg/m)
% output:
%   vend = velocity at tf (m/s)
t = ti;
v = vi;

n = (tf - ti) / dt;
for i = 1:n
    dvdt = deriv(t, v, m, cd);
    v = v + dvdt * dt;
    t = t + dt;
end
vend = v;
```

We must also set up a function to compute the derivative:

```
function dv = deriv(t, v, m, cd)
g = 9.81;
dv = g - (cd / m) * v^2;
```

This function can be invoked from the command window with the result:

```
>> velocity1(0.5,0,12,0,68.1,0.25)

ans =
    50.9259
```

Note that the true value obtained from the analytical solution is 50.6175 (Example 3.1). We can then try a much smaller value of Δt to obtain a more accurate numerical result:

```
>> velocity1(0.001,0,12,0,68.1,0.25)

ans =
    50.6181
```

Although this function is certainly simple to program, it is not foolproof. In particular, it will not work if the computation interval is not evenly divisible by the time step. To cover such cases, a `while . . . break` loop can be substituted in place of the shaded area:

```
function vend = velocity2(dt, ti, tf, vi, m, cd)
% velocity2(dt, ti, tf, vi, m, cd):
%   euler's method solution of freefalling bungee jumper
%   using a while loop
% input:
%   dt = delta t (s)
%   ti = initial time (s)
%   tf = final time (s)
%   vi = initial value of dependent variable (m/s)
%   m = mass(kg)
%   cd = second-order drag coefficient(kg/m)
% output:
%   vend = velocity at tf (m/s)
t = ti;
v = vi;
h = dt;
while (1)
    if t + dt > tf, h = tf - t; end
    dvdt = deriv(t, v, m, cd);
    v = v + dvdt * h;
    t = t + h;
    if t >= tf, break, end
end
vend = v;
```

As soon as we enter the `while` loop, we use a single line `if` structure to test whether adding `t + dt` will take us beyond the end of the interval. If not (which would usually be the case at first), we do nothing. If so, we would shorten up the interval—that is, we set the variable step `h` to the interval remaining: `tf - t`. By doing this, we guarantee that the last step falls exactly on `tf`. After we implement this final step, the loop will terminate because the condition `t >= tf` will test true.

Notice that before entering the loop, we assign the value of the time step `dt` to another variable `h`. We create this *dummy variable* so that our routine does not change the given value of `dt` if and when we shorten the time step. We do this in anticipation that we might need to use the original value of `dt` somewhere else in the event that this code were integrated within a larger program.

If we run this new version, the result will be the same as for the version based on the `for` structure:

```
>> velocity2(0.5,0,12,0,68.1,0.25)

ans =
    50.9259
```

PROBLEMS

55

Further, we can use a Δt that is not evenly divisible into $t_f - t_i$:

```
>> velocity2(0.35,0,12,0.68,1,0.25)

ans =
    50.8348
```

We should note that the algorithm is still not foolproof. For example, the user could have mistakenly entered a step size greater than the calculation interval (e.g., $t_f - t_i = 5$ and $\Delta t = 20$). Thus, you might want to include error traps in your code to catch such errors and then allow the user to correct the mistake.

PROBLEMS

3.1 The sine function can be evaluated by the following infinite series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

Create an M-file to implement this formula so that it computes and displays the values of $\sin x$ as each term in the series is added. In other words, compute and display in sequence the values for

$$\begin{aligned} \sin x &= x \\ \sin x &= x - \frac{x^3}{3!} \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} \\ &\vdots \end{aligned}$$

up to the order term of your choosing. For each of the preceding, compute and display the percent relative error as

$$\%error = \frac{\text{true} - \text{series approximation}}{\text{true}} \cdot 100\%$$

As a test case, employ the program to compute $\sin(1.5)$ for up to and including eight terms—that is, up to the term $x^{15}/15!$.

3.2 An amount of money P is invested in an account where interest is compounded at the end of the period. The future worth F yielded at an interest rate i after n periods may be determined from the following formula:

$$F = P(1 + i)^n$$

Write an M-file that will calculate the future worth of an investment for each year from 1 through n . The input to the function should include the initial investment P , the interest

rate i (as a decimal), and the number of years n for which the future worth is to be calculated. The output should consist of a table with headings and columns for n and F . Run the program for $P = \$100,000$, $i = 0.08$, and $n = 8$ years.

3.3 Economic formulas are available to compute annual payments for loans. Suppose that you borrow an amount of money P and agree to repay it in n annual payments at an interest rate of i . The formula to compute the annual payment A is

$$A = P \frac{i(1 + i)^n}{(1 + i)^n - 1}$$

Write an M-file to compute A . Test it with $P = \$35,000$ and an interest rate of 7.6% ($i = 0.076$). Compute results for $n = 1, 2, 3, 4$, and 5 and display the results as a table with headings and columns for n and A .

3.4 The average daily temperature for an area can be approximated by the following function:

$$T = T_{\text{mean}} + (T_{\text{peak}} - T_{\text{mean}}) \cos(\omega(t - t_{\text{peak}}))$$

where T_{mean} = the average annual temperature, T_{peak} = the peak temperature, ω = the frequency of the annual variation ($= 2\pi/365$), and t_{peak} = day of the peak temperature ($= 205$ d). Parameters for some U.S. towns are listed here:

City	mean ($^{\circ}\text{C}$)	peak ($^{\circ}\text{C}$)
Miami, FL	22.1	28.3
Yuma, AZ	23.1	33.6
Bismarck, ND	5.2	22.1
Seattle, WA	10.6	17.6
Boston, MA	10.7	22.9

Develop an M-file that computes the average temperature between two days of the year for a particular city. Test it for

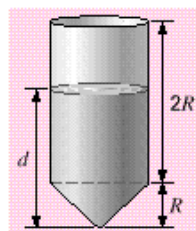


FIGURE P3.5

(a) January–February in Bismarck, ND ($t = 0$ to 59) and
(b) July–August temperature in Yuma, AZ ($t = 180$ to 242).
3.5 Figure P3.5 shows a cylindrical tank with a conical base. If the liquid level is quite low, in the conical part, the volume is simply the conical volume of liquid. If the liquid level is midrange in the cylindrical part, the total volume of liquid includes the filled conical part and the partially filled cylindrical part.

Use decisional structures to write an M-file to compute the tank's volume as a function of given values of R and d . Design the function so that it returns the volume for all cases where the depth is less than $3R$. Return an error message ("Overtop") if you overtop the tank—that is, $d > 3R$. Test it with the following data:

	1	1	1	1
	0.5	1.2	3.0	3.1

3.6 Two distances are required to specify the location of a point relative to an origin in two-dimensional space (Fig. P3.6):

- The horizontal and vertical distances (x, y) in Cartesian coordinates.
- The radius and angle (r, θ) in polar coordinates.

It is relatively straightforward to compute Cartesian coordinates (x, y) on the basis of polar coordinates (r, θ). The reverse process is not so simple. The radius can be computed by the following formula:

$$r = \sqrt{x^2 + y^2}$$

If the coordinates lie within the first and fourth quadrants (i.e., $x > 0$), then a simple formula can be used to compute θ :

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

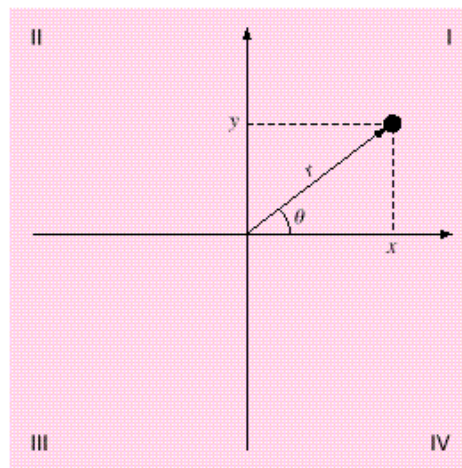


FIGURE P3.6

The difficulty arises for the other cases. The following table summarizes the possibilities:

		θ
<0	>0	$\tan^{-1}[y/x] + \pi$
<0	<0	$\tan^{-1}[y/x] - \pi$
<0	$=0$	π
$=0$	>0	$\pi/2$
$=0$	<0	$-\pi/2$
$=0$	$=0$	0

Write a well-structured M-file to calculate r and θ as a function of x and y . Express the final results for θ in degrees. Test your program by evaluating the following cases:

		θ
1	1	
1	-1	
1	0	
-1	1	
-1	-1	
-1	0	
0	1	
0	-1	
0	0	

3.7 Develop an M-file to determine polar coordinates as described in Prob. 3.6. However, rather than designing the

PROBLEMS

57

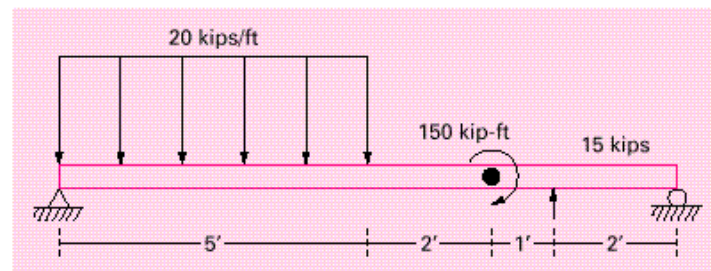


FIGURE P3.10

function to evaluate a single case, pass vectors of x and y . Have the function display the results as a table with columns for x , y , r , and θ . Test the program for the cases outlined in Prob. 3.6.

3.8 Develop an M-file function that is passed a numeric grade from 0 to 100 and returns a letter grade according to the scheme:

Letter	Criteria
A	$90 \leq \text{numeric grade} \leq 100$
B	$80 \leq \text{numeric grade} < 90$
C	$70 \leq \text{numeric grade} < 80$
D	$60 \leq \text{numeric grade} < 70$
F	$\text{numeric grade} < 60$

3.9 Manning's equation can be used to compute the velocity of water in a rectangular open channel:

$$U = \frac{S}{n} \left(\frac{BH}{B + 2H} \right)^{2/3}$$

where U = velocity (m/s), S = channel slope, n = roughness coefficient, B = width (m), and H = depth (m). The following data is available for five channels:

n	B (m)	H (m)	S
0.035	0.0001	10	2
0.020	0.0002	8	1
0.015	0.0010	20	1.5
0.030	0.0007	24	3
0.022	0.0003	15	2.5

Write an M-file that computes the velocity for each of these channels. Enter these values into a matrix where each column represents a parameter and each row represents a channel. Have the M-file display the input data along with the computed velocity in tabular form where velocity is the fifth column. Include headings on the table to label the columns.

3.10 A simply supported beam is loaded as shown in Fig. P3.10. Using singularity functions, the displacement along the beam can be expressed by the equation:

$$u_y(x) = \frac{5}{6} [x \cdot 0^4 - x \cdot 5^4] + \frac{15}{6} x \cdot 8^3 - 75 \cdot x \cdot 7^2 + \frac{57}{6} x^3 - 238.25x$$

By definition, the singularity function can be expressed as follows:

$$x \cdot a^n = \begin{cases} (x - a)^n & \text{when } x > a \\ 0 & \text{when } x \leq a \end{cases}$$

Develop an M-file that creates a plot of displacement versus distance along the beam, x . Note that $x = 0$ at the left end of the beam.

3.11 The volume V of liquid in a hollow horizontal cylinder of radius r and length L is related to the depth of the liquid h by

$$V = \left[r^2 \cos^{-1} \left(\frac{r-h}{r} \right) + (r-h) \sqrt{2rh-h^2} \right] L$$

Develop an M-file to create a plot of volume versus depth. Test the program for $r = 2$ m and $L = 5$ m.

Round-Off and Truncation Errors

CHAPTER OBJECTIVES

The primary objective of this chapter is to acquaint you with the major sources of errors involved in numerical methods. Specific objectives and topics covered are

- Understanding the distinction between accuracy and precision.
- Learning how to quantify error.
- Learning how error estimates can be used to decide when to terminate an iterative calculation.
- Understanding how round-off errors occur because digital computers have a limited ability to represent numbers.
- Recognizing that truncation errors occur when exact mathematical formulations are represented by approximations.
- Knowing how to use the Taylor series to estimate truncation errors.
- Understanding how to write forward, backward, and centered finite difference approximations of first and second derivatives.

YOU'VE GOT A PROBLEM

In Chap. 1, you developed a numerical model for the velocity of a bungee jumper. To solve the problem with a computer, you had to approximate the derivative of velocity with a divided difference:

$$\frac{dv}{dt} \approx \frac{\Delta v}{\Delta t} = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i}$$

Thus, the resulting solution is not perfect—that is, it has error.

In addition, the computer you use to obtain the solution is also an imperfect tool. Because it is a digital device, the computer is limited in its ability to represent the magnitudes and precision of numbers. Thus, the machine itself yields results that contain error.

So both your mathematical approximation and your digital computer cause your resulting model prediction to be uncertain. Your problem is: How do you deal with such uncertainty? This chapter introduces you to some approaches and ideas that engineers and scientists use to deal with this dilemma.

4.1 ERRORS

Engineers and scientists constantly find themselves having to accomplish objectives based on uncertain information. Although perfection is a laudable goal, it is rarely if ever attained. For example, despite the fact that the model developed from Newton's second law is an excellent approximation, it would never in practice exactly predict the jumper's fall. A variety of factors such as winds and slight variations in air resistance would result in deviations from the prediction. If these deviations are systematically high or low, then we might need to develop a new model. However, if they are randomly distributed and tightly grouped around the prediction, then the deviations might be considered negligible and the model deemed adequate. Numerical approximations also introduce similar discrepancies into the analysis.

This chapter covers basic topics related to the identification, quantification, and minimization of these errors. General information concerned with the quantification of error is reviewed in this section. This is followed by Sections 4.2 and 4.3, dealing with the two major forms of numerical error: round-off error (due to computer approximations) and truncation error (due to mathematical approximations). Finally, we briefly discuss errors not directly connected with the numerical methods themselves. These include blunders, model errors, and data uncertainty.

4.1.1 Accuracy and Precision

The errors associated with both calculations and measurements can be characterized with regard to their accuracy and precision. *Accuracy* refers to how closely a computed or measured value agrees with the true value. *Precision* refers to how closely individual computed or measured values agree with each other.

These concepts can be illustrated graphically using an analogy from target practice. The bullet holes on each target in Fig. 4.1 can be thought of as the predictions of a numerical technique, whereas the bull's-eye represents the truth. *Inaccuracy* (also called *bias*) is defined as systematic deviation from the truth. Thus, although the shots in Fig. 4.1c are more tightly grouped than in Fig. 4.1a, the two cases are equally biased because they are both centered on the upper left quadrant of the target. *Imprecision* (also called *uncertainty*), on the other hand, refers to the magnitude of the scatter. Therefore, although Fig. 4.1b and d are equally accurate (i.e., centered on the bull's-eye), the latter is more precise because the shots are tightly grouped.

Numerical methods should be sufficiently accurate or unbiased to meet the requirements of a particular problem. They also should be precise enough for adequate design. In this book, we will use the collective term *error* to represent both the inaccuracy and imprecision of our predictions.

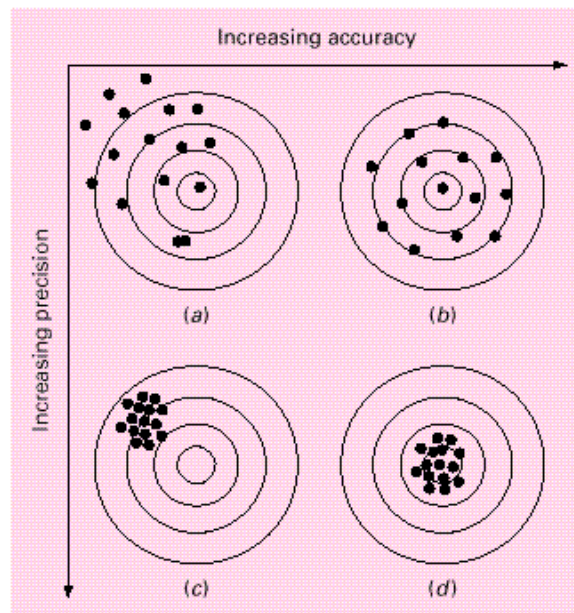


FIGURE 4.1

An example from marksmanship illustrating the concepts of accuracy and precision: (· ·) inaccurate and imprecise, (· ·) accurate and imprecise, (· ·) inaccurate and precise, and (· ·) accurate and precise.

4.1.2 Error Definitions

Numerical errors arise from the use of approximations to represent exact mathematical operations and quantities. For such errors, the relationship between the exact, or true, result and the approximation can be formulated as

$$\text{True value} - \text{approximation} = \text{error} \quad (4.1)$$

By rearranging Eq. (4.1), we find that the numerical error is equal to the discrepancy between the truth and the approximation, as in

$$E_t = \text{true value} - \text{approximation} \quad (4.2)$$

where E_t is used to designate the exact value of the error. The subscript t is included to designate that this is the “true” error. This is in contrast to other cases, as described shortly, where an “approximate” estimate of the error must be employed.

A shortcoming of this definition is that it takes no account of the order of magnitude of the value under examination. For example, an error of a centimeter is much more significant if we are measuring a rivet than a bridge. One way to account for the magnitudes of the quantities being evaluated is to normalize the error to the true value, as in

$$\text{True fractional relative error} = \frac{\text{true value} - \text{approximation}}{\text{true value}}$$

4.1 ERRORS

61

The relative error can also be multiplied by 100% to express it as

$$\varepsilon_t = \frac{\text{true value} - \text{approximation}}{\text{true value}} 100\% \quad (4.3)$$

where ε_t designates the true percent relative error.

For example suppose that you have the task of measuring the lengths of a bridge and a rivet and come up with 9999 and 9 cm, respectively. If the true values are 10,000 and 10 cm, respectively, the error in both cases is 1 cm. However, their percent relative errors can be computed using Eq. (4.3) as 0.01% and 10%, respectively. Thus, although both measurements have an absolute error of 1 cm, the relative error for the rivet is much greater. We would probably conclude that we have done an adequate job of measuring the bridge, whereas our estimate for the rivet leaves something to be desired.

Notice that for Eqs. (4.2) and (4.3), E and ε are subscripted with a t to signify that the error is based on the true value. For the example of the rivet and the bridge, we were provided with this value. However, in actual situations such information is rarely available. For numerical methods, the true value will only be known when we deal with functions that can be solved analytically. Such will typically be the case when we investigate the theoretical behavior of a particular technique for simple systems. However, in real-world applications, we will obviously not know the true answer *a priori*. For these situations, an alternative is to normalize the error using the best available estimate of the true value—that is, to the approximation itself, as in

$$\varepsilon_a = \frac{\text{approximate error}}{\text{approximation}} 100\% \quad (4.4)$$

where the subscript a signifies that the error is normalized to an approximate value. Note also that for real-world applications, Eq. (4.2) cannot be used to calculate the error term in the numerator of Eq. (4.4). One of the challenges of numerical methods is to determine error estimates in the absence of knowledge regarding the true value. For example, certain numerical methods use *iteration* to compute answers. In such an approach, a present approximation is made on the basis of a previous approximation. This process is performed repeatedly, or iteratively, to successively compute (hopefully) better and better approximations. For such cases, the error is often estimated as the difference between previous and present approximations. Thus, percent relative error is determined according to

$$\varepsilon_a = \frac{\text{present approximation} - \text{previous approximation}}{\text{present approximation}} 100\% \quad (4.5)$$

This and other approaches for expressing errors is elaborated on in subsequent chapters.

The signs of Eqs. (4.2) through (4.5) may be either positive or negative. If the approximation is greater than the true value (or the previous approximation is greater than the current approximation), the error is negative; if the approximation is less than the true value, the error is positive. Also, for Eqs. (4.3) to (4.5), the denominator may be less than zero, which can also lead to a negative error. Often, when performing computations, we may not be concerned with the sign of the error but are interested in whether the percent absolute value is lower than a prespecified percent tolerance ε_s . Therefore, it is often useful to employ the absolute value of Eq. (4.5). For such cases, the computation is repeated until

$$|\varepsilon_a| < \varepsilon_s \quad (4.6)$$

This relationship is referred to as a *stopping criterion*. If it is satisfied, our result is assumed to be within the prespecified acceptable level ε_s . Note that for the remainder of this text, we almost always employ absolute values when using relative errors.

It is also convenient to relate these errors to the number of significant figures in the approximation. It can be shown (Scarborough, 1966) that if the following criterion is met, we can be assured that the result is correct to *at least* n significant figures.

$$\varepsilon_s \leq (0.5 \cdot 10^{2-n})\% \quad (4.7)$$

EXAMPLE 4.1 Error Estimates for Iterative Methods

Problem Statement. In mathematics, functions can often be represented by infinite series. For example, the exponential function can be computed using

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \quad (\text{E4.1.1})$$

Thus, as more terms are added in sequence, the approximation becomes a better and better estimate of the true value of e^x . Equation (E4.1.1) is called a *Maclaurin series expansion*.

Starting with the simplest version, $e^x = 1$, add terms one at a time in order to estimate $e^{0.5}$. After each new term is added, compute the true and approximate percent relative errors with Eqs. (4.3) and (4.5), respectively. Note that the true value is $e^{0.5} = 1.648721 \dots$. Add terms until the absolute value of the approximate error estimate ε_a falls below a prespecified error criterion ε_s conforming to three significant figures.

Solution. First, Eq. (4.7) can be employed to determine the error criterion that ensures a result that is correct to at least three significant figures:

$$\varepsilon_s = (0.5 \cdot 10^{2-3})\% = 0.05\%$$

Thus, we will add terms to the series until ε_a falls below this level.

The first estimate is simply equal to Eq. (E4.1.1) with a single term. Thus, the first estimate is equal to 1. The second estimate is then generated by adding the second term as in

$$e^x = 1 + x$$

or for $x = 0.5$

$$e^{0.5} = 1 + 0.5 = 1.5$$

This represents a true percent relative error of [Eq. (4.3)]

$$\varepsilon_t = \left| \frac{1.648721 - 1.5}{1.648721} \right| \cdot 100\% = 9.02\%$$

Equation (4.5) can be used to determine an approximate estimate of the error, as in

$$\varepsilon_a = \left| \frac{1.5 - 1}{1.5} \right| \cdot 100\% = 33.3\%$$

4.2 ROUND-OFF ERRORS

63

Because ε_a is not less than the required value of ε_s , we would continue the computation by adding another term, $x^2/2!$, and repeating the error calculations. The process is continued until $|\varepsilon_a| < \varepsilon_s$. The entire computation can be summarized as

Terms	Result	ε_a , %	ε_s , %
1	1	39.3	
2	1.5	9.02	33.3
3	1.625	1.44	7.69
4	1.645833333	0.175	1.27
5	1.648437500	0.0172	0.158
6	1.648697917	0.00142	0.0158

Thus, after six terms are included, the approximate error falls below $\varepsilon_s = 0.05\%$, and the computation is terminated. However, notice that, rather than three significant figures, the result is accurate to five! This is because, for this case, both Eqs. (4.5) and (4.7) are conservative. That is, they ensure that the result is at least as good as they specify. Although, this is not always the case for Eq. (4.5), it is true most of the time.

4.2 ROUND-OFF ERRORS

Round-off errors arise because digital computers cannot represent some quantities exactly. They are important to engineering and scientific problem solving because they can lead to erroneous results. In certain cases, they can actually lead to a calculation going unstable and yielding obviously erroneous results. Worse still, they can lead to subtler discrepancies that are difficult to detect.

There are two major facets of round-off errors involved in numerical calculations:

1. Digital computers have size and precision limits on their ability to represent numbers.
2. Certain numerical manipulations are highly sensitive to round-off errors. This can result from both mathematical considerations as well as from the way in which computers perform arithmetic operations.

4.2.1 Computer Number Representation

Numerical round-off errors are directly related to the manner in which numbers are stored in a computer. The fundamental unit whereby information is represented is called a *word*. This is an entity that consists of a string of binary digits, or *bits*. Numbers are typically stored in one or more words. To understand how this is accomplished, we must first review some material related to number systems.

A *number system* is merely a convention for representing quantities. Because we have 10 fingers and 10 toes, the number system that we are most familiar with is the *decimal*, or *base-10*, number system. A base is the number used as the reference for constructing the system. The base-10 system uses the 10 digits—0, 1, 2, 3, 4, 5, 6, 7, 8, and 9—to represent numbers. By themselves, these digits are satisfactory for counting from 0 to 9.

For larger quantities, combinations of these basic digits are used, with the position or *place value* specifying the magnitude. The rightmost digit in a whole number represents a number from 0 to 9. The second digit from the right represents a multiple of 10. The third digit from the right represents a multiple of 100 and so on. For example, if we have the number 8642.9, then we have eight groups of 1000, six groups of 100, four groups of 10, two groups of 1, and nine groups of 0.1, or

$$(8 \cdot 10^3) + (6 \cdot 10^2) + (4 \cdot 10^1) + (2 \cdot 10^0) + (9 \cdot 10^{-1}) = 8642.9$$

This type of representation is called *positional notation*.

Now, because the decimal system is so familiar, it is not commonly realized that there are alternatives. For example, if human beings happened to have eight fingers and toes we would undoubtedly have developed an *octal*, or *base-8*, representation. In the same sense, our friend the computer is like a two-fingered animal who is limited to two states—either 0 or 1. This relates to the fact that the primary logic units of digital computers are on/off electronic components. Hence, numbers on the computer are represented with a *binary*, or *base-2*, system. Just as with the decimal system, quantities can be represented using positional notation. For example, the binary number 101.1 is equivalent to $(1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) + (1 \cdot 2^{-1}) = 4 + 0 + 1 + 0.5 = 5.5$ in the decimal system.

The fact that digital computers use a finite number of *binary digits* (or *bits*) to represent numbers has two major implications:

Precision. Some numbers cannot be represented exactly. For example, irrational numbers such as π , e , or $\sqrt{7}$ cannot be expressed by a finite number of significant figures. Therefore, they cannot be represented exactly by the computer. In addition, because computers use a binary, or base-2, representation they cannot precisely represent certain exact base-10 numbers. For example, the exact base-10 quantity 0.1 cannot be represented exactly in a base-2 system.

In general, for computer tools that use 16-bit word size, fractional numbers, called *floating point numbers* in computing jargon, can be expressed to about seven base-10 digits of precision. Thus, π can be expressed as 3.141593. For tools using 32-bit words, the precision increases to about 15 base-10 digits. Thus, π would be expressed as 3.14159265358979. Note that 32-bit precision is standard in MATLAB. This is sometimes called *double precision*.

Range. There are finite ranges of values that the computer can represent. For computer tools that use 16-bit word size, the range of integers is typically from $-32,768$ to $32,767$. For those using 32-bit word size, these increase to $-2,147,483,648$ to $2,147,483,647$. For floating-point numbers, the ranges are 10^{-38} to 10^{39} and 10^{-308} to 10^{308} for 16-bit and 32-bit word size, respectively.

MATLAB has a number of built-in functions related to its internal number representation. For example, the `realmax` function displays the largest positive real number:

```
>> format long
>> realmax

ans =
    1.797693134862316e+308
```

4.2 ROUND-OFF ERRORS

65

Numbers occurring in computations that exceed this value create an *overflow* and generally cause the calculation to terminate and an error message to be displayed.

The `realmin` function displays the smallest positive real number:

```
>> realmin  
  
ans =  
2.225073858507201e-308
```

Numbers that are smaller than this value create an *underflow* and are generally set to zero.

The `eps` function displays the smallest number that can be added to one that produces a number larger than one:

```
>> eps  
  
ans =  
2.220446049250313e-016
```

This value, which is sometimes referred to as the *machine epsilon*, represents the finest level of resolution that is possible for floating-point arithmetic. Thus, we can see that the value of `eps` supports our previous contention that tools such as MATLAB that use 32-bit word lengths can only represent about 15 base-10 digits.

4.2.2 Arithmetic Manipulations of Computer Numbers

Aside from the limitations of a computer's number system, the actual arithmetic manipulations involving these numbers can also result in round-off error. To understand how this occurs, let's look at how the computer performs simple addition and subtraction.

Because of their familiarity, normalized base-10 numbers will be employed to illustrate the effect of round-off errors on simple addition and subtraction. Other number bases would behave in a similar fashion. To simplify the discussion, we will employ a hypothetical decimal computer with a 4-digit mantissa and a 1-digit exponent.

When two floating-point numbers are added, the numbers are first expressed so that they have the same exponents. For example, if we want to add $1.557 \cdot 10^1 + 0.04341$, the computer would express the numbers as $0.1557 \cdot 10^1 + 0.004341 \cdot 10^1$. Then the mantissas are added to give $0.160041 \cdot 10^1$. Now, because this hypothetical computer only carries a four-digit mantissa, the excess number of digits get chopped off and the result is $0.1600 \cdot 10^1$. Notice how the last two digits of the second number (41) that were shifted to the right have essentially been lost from the computation.

Subtraction is performed identically to addition except that the sign of the subtrahend is reversed. For example, suppose that we are subtracting 26.86 from 36.41. That is,

$$\begin{array}{r} 0.3641 \cdot 10^2 \\ - 0.2686 \cdot 10^2 \\ \hline 0.0955 \cdot 10^2 \end{array}$$

For this case the result must be normalized because the leading zero is unnecessary. So we must shift the decimal one place to the right to give $0.9550 \cdot 10^1 = 9.550$. Notice that the zero added to the end of the mantissa is not significant but is merely appended to fill the empty space created by the shift. Even more dramatic results would be obtained when the

numbers are very close as in

$$\begin{array}{r} 0.7642 \cdot 10^3 \\ - 0.7641 \cdot 10^3 \\ \hline 0.0001 \cdot 10^3 \end{array}$$

which would be converted to $0.1000 \cdot 10^0 = 0.1000$. Thus, for this case, three nonsignificant zeros are appended.

The subtracting of two nearly equal numbers is called *subtractive cancellation*. It is the classic example of how the manner in which computers handle mathematics can lead to numerical problems. Other calculations that can cause problems include:

Large Computations. Certain methods require extremely large numbers of arithmetic manipulations to arrive at their final results. In addition, these computations are often interdependent. That is, the later calculations are dependent on the results of earlier ones. Consequently, even though an individual round-off error could be small, the cumulative effect over the course of a large computation can be significant. A very simple case involves summing a round base-10 number that is not round in base-2. Suppose that the following M-file is constructed:

```
function sout = sumdemo()
s = 0;
for i = 1:10000
    s = s + 0.0001;
end
sout = s;
```

When this function is executed, the result is

```
>> format long
>> sumdemo

ans =
    0.999999999999991
```

The `format long` command lets us see the 15 significant-digit representation used by MATLAB. You would expect that sum would be equal to 1. However, although 0.0001 is a nice round number in base-10, it cannot be expressed exactly in base-2. Thus, the sum comes out to be slightly different than 1. We should note that MATLAB has features that are designed to minimize such errors. For example, suppose that you form a vector as in

```
>> format long
>> s = [0:0.0001:1];
```

For this case, rather than being equal to 0.999999999999991, the last entry will be exactly one as verified by

```
>> s(10001)

ans =
    1
```

4.3 TRUNCATION ERRORS

67

Adding a Large and a Small Number. Suppose we add a small number, 0.0010, to a large number, 4000, using a hypothetical computer with the 4-digit mantissa and the 1-digit exponent. After modifying the smaller number so that its exponent matches the larger,

$$\begin{array}{r} 0.4000 \cdot 10^4 \\ 0.0000001 \cdot 10^4 \\ \hline 0.4000001 \cdot 10^4 \end{array}$$

which is chopped to $0.4000 \cdot 10^4$. Thus, we might as well have not performed the addition! This type of error can occur in the computation of an infinite series. The initial terms in such series are often relatively large in comparison with the later terms. Thus, after a few terms have been added, we are in the situation of adding a small quantity to a large quantity. One way to mitigate this type of error is to sum the series in reverse order. In this way, each new term will be of comparable magnitude to the accumulated sum.

Smearing. Smearing occurs whenever the individual terms in a summation are larger than the summation itself. One case where this occurs is in series of mixed signs.

Inner Products. As should be clear from the last sections, some infinite series are particularly prone to round-off error. Fortunately, the calculation of series is not one of the more common operations in numerical methods. A far more ubiquitous manipulation is the calculation of inner products as in

$$\sum_{i=1}^n x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

This operation is very common, particularly in the solution of simultaneous linear algebraic equations. Such summations are prone to round-off error. Consequently, it is often desirable to compute such summations in double precision as is done automatically in MATLAB.

4.3 TRUNCATION ERRORS

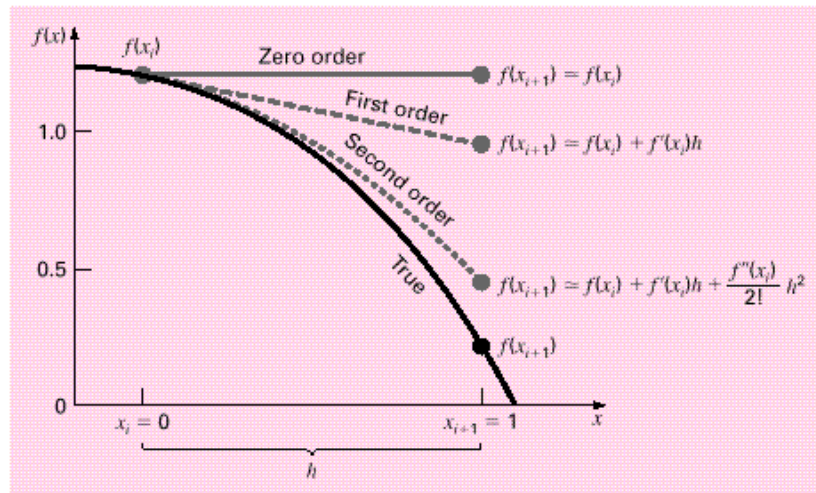
Truncation errors are those that result from using an approximation in place of an exact mathematical procedure. For example, in Chap. 1 we approximated the derivative of velocity of a bungee jumper by a finite-divided-difference equation of the form [Eq. (1.11)]

$$\frac{dv}{dt} \approx \frac{\Delta v}{\Delta t} = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i} \quad (4.8)$$

A truncation error was introduced into the numerical solution because the difference equation only approximates the true value of the derivative (recall Fig. 1.3). To gain insight into the properties of such errors, we now turn to a mathematical formulation that is used widely in numerical methods to express functions in an approximate fashion—the Taylor series.

4.3.1 The Taylor Series

Taylor's theorem and its associated formula, the Taylor series, is of great value in the study of numerical methods. In essence, the *Taylor theorem* states that any smooth function can be approximated as a polynomial. The *Taylor series* then provides a means to express this idea mathematically in a form that can be used to come up with practical results.

**FIGURE 4.2**

The approximation of $f(x) = -0.1x^4 + 0.15x^3 + 0.5x^2 + 0.25x + 1.2$ at $x = 1$ by zero-order, first-order, and second-order Taylor series expansions.

A useful way to gain insight into the Taylor series is to build it term by term. A good problem context for this exercise is to predict a function value at one point in terms of the function value and its derivatives at another point.

Suppose that you are blindfolded and taken to a location on the side of a hill facing downslope (Fig. 4.2). We'll call your horizontal location x_i and your vertical distance with respect to the base of the hill $f(x_i)$. You are given the task of predicting the height at a position x_{i+1} , which is a distance h away from you.

At first, you are placed on a platform that is completely horizontal so that you have no idea that the hill is sloping down away from you. At this point, what would be your best guess at the height at x_{i+1} ? If you think about it (remember you have no idea whatsoever what's in front of you), the best guess would be the same height as where you're standing now! You could express this prediction mathematically as

$$f(x_{i+1}) = f(x_i) \quad (4.9)$$

This relationship, which is called the *zero-order approximation*, indicates that the value of f at the new point is the same as the value at the old point. This result makes intuitive sense because if x_i and x_{i+1} are close to each other, it is likely that the new value is probably similar to the old value.

Equation (4.9) provides a perfect estimate if the function being approximated is, in fact, a constant. For our problem, you would be right only if you happened to be standing on a perfectly flat plateau. However, if the function changes at all over the interval, additional terms of the Taylor series are required to provide a better estimate.

So now you are allowed to get off the platform and stand on the hill surface with one leg positioned in front of you and the other behind. You immediately sense that the front

4.3 TRUNCATION ERRORS

69

foot is lower than the back foot. In fact, you're allowed to obtain a quantitative estimate of the slope by measuring the difference in elevation and dividing it by the distance between your feet.

With this additional information, you're clearly in a better position to predict the height at $f(x_{i+1})$. In essence, you use the slope estimate to project a straight line out to x_{i+1} . You can express this prediction mathematically by

$$f(x_{i+1}) \approx f(x_i) + f'(x_i)h \quad (4.10)$$

This is called a *first-order approximation* because the additional first-order term consists of a slope $f'(x_i)$ multiplied by h , the distance between x_i and x_{i+1} . Thus, the expression is now in the form of a straight line that is capable of predicting an increase or decrease of the function between x_i and x_{i+1} .

Although Eq. (4.10) can predict a change, it is only exact for a straight-line, or *linear*, trend. To get a better prediction, we need to add more terms to our equation. So now you are allowed to stand on the hill surface and take two measurements. First, you measure the slope behind you by keeping one foot planted at x_i and moving the other one back a distance Δx . Let's call this slope $f'_b(x_i)$. Then you measure the slope in front of you by keeping one foot planted at x_i and moving the other one forward Δx . Let's call this slope $f'_f(x_i)$. You immediately recognize that the slope behind is milder than the one in front. Clearly the drop in height is "accelerating" in front of you. Thus, the odds are that $f'(x_i)$ is even lower than your previous linear prediction.

As you might expect, you're now going to add a second-order term to your equation and make it into a parabola. The Taylor series provides the correct way to do this as in

$$f(x_{i+1}) \approx f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 \quad (4.11)$$

To make use of this formula, you need an estimate of the second derivative. You can use the last two slopes you determined to estimate it as

$$f''(x_i) \approx \frac{f'_f(x_i) - f'_b(x_i)}{\Delta x} \quad (4.12)$$

Thus, the second derivative is merely a derivative of a derivative; in this case, the rate of change of the slope.

Before proceeding, let's look carefully at Eq. (4.11). Recognize that all the values subscripted i represent values that you have estimated. That is, they are numbers. Consequently, the only unknowns are the values at the prediction position x_{i+1} . Consequently, it is a quadratic equation of the form

$$f(h) \approx a_2 h^2 + a_1 h + a_0$$

Thus, we can see that the second-order Taylor series approximates the function with a second-order polynomial.

Clearly, we could keep adding more derivatives to capture more of the function's curvature. Thus, we arrive at the complete Taylor series expansion

$$f(x_{i+1}) \approx f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f^{(3)}(x_i)}{3!}h^3 + \dots + \frac{f^{(n)}(x_i)}{n!}h^n + R_n \quad (4.13)$$

Note that because Eq. (4.13) is an infinite series, an equal sign replaces the approximate sign that was used in Eqs. (4.9) through (4.11). A remainder term is also included to account for all terms from $n + 1$ to infinity:

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1} \quad (4.14)$$

where the subscript n connotes that this is the remainder for the n th-order approximation and ξ is a value of x that lies somewhere between x_i and x_{i+1} .

Thus, we can now see why the Taylor theorem states that any smooth function can be approximated as a polynomial and that the Taylor series provides a means to express this idea mathematically.

In general, the n th-order Taylor series expansion will be exact for an n th-order polynomial. For other differentiable and continuous functions, such as exponentials and sinusoids, a finite number of terms will not yield an exact estimate. Each additional term will contribute some improvement, however slight, to the approximation. This behavior is demonstrated in Example 4.2. Only if an infinite number of terms are added will the series yield an exact result.

Although the foregoing is true, the practical value of Taylor series expansions is that, in most cases, the inclusion of only a few terms will result in an approximation that is close enough to the true value for practical purposes. The assessment of how many terms are required to get “close enough” is based on the remainder term of the expansion (Eq. 4.14). This relationship has two major drawbacks. First, ξ is not known exactly but merely lies somewhere between x_i and x_{i+1} . Second, to evaluate Eq. (4.14), we need to determine the $(n+1)$ th derivative of $f(x)$. To do this, we need to know $f(x)$. However, if we knew $f(x)$, there would be no need to perform the Taylor series expansion in the present context!

Despite this dilemma, Eq. (4.14) is still useful for gaining insight into truncation errors. This is because we *do* have control over the term h in the equation. In other words, we can choose how far away from x we want to evaluate $f(x)$, and we can control the number of terms we include in the expansion. Consequently, Eq. (4.14) is usually expressed as

$$R_n = O(h^{n+1})$$

where the nomenclature $O(h^{n+1})$ means that the truncation error is of the order of h^{n+1} . That is, the error is proportional to the step size h raised to the $(n+1)$ th power. Although this approximation implies nothing regarding the magnitude of the derivatives that multiply h^{n+1} , it is extremely useful in judging the comparative error of numerical methods based on Taylor series expansions. For example, if the error is $O(h)$, halving the step size will halve the error. On the other hand, if the error is $O(h^2)$, halving the step size will quarter the error.

In general, we can usually assume that the truncation error is decreased by the addition of terms to the Taylor series. In many cases, if h is sufficiently small, the first- and other lower-order terms usually account for a disproportionately high percent of the error. Thus, only a few terms are required to obtain an adequate approximation. This property is illustrated by the following example.

EXAMPLE 4.2 Approximation of a Function with a Taylor Series Expansion

Problem Statement. Use Taylor series expansions with $n = 0$ to 6 to approximate $f(x) = \cos x$ at $x_{i+1} = \pi/3$ on the basis of the value of $f(x)$ and its derivatives at $x_i = \pi/4$. Note that this means that $h = \pi/3 - \pi/4 = \pi/12$.

Solution. Our knowledge of the true function means that we can determine the correct value $f(\pi/3) = 0.5$. The zero-order approximation is [Eq. (4.9)]

$$f\left(\frac{\pi}{3}\right) \approx \cos\left(\frac{\pi}{4}\right) = 0.707106781$$

which represents a percent relative error of

$$\varepsilon_t = \left| \frac{0.5 - 0.707106781}{0.5} \right| 100\% = 41.4\%$$

For the first-order approximation, we add the first derivative term where $f'(x) = -\sin x$:

$$f\left(\frac{\pi}{3}\right) \approx \cos\left(\frac{\pi}{4}\right) - \sin\left(\frac{\pi}{4}\right)\left(\frac{\pi}{12}\right) = 0.521986659$$

which has $\varepsilon_t = 4.40\%$. For the second-order approximation, we add the second derivative term where $f''(x) = -\cos x$:

$$f\left(\frac{\pi}{3}\right) \approx \cos\left(\frac{\pi}{4}\right) - \sin\left(\frac{\pi}{4}\right)\left(\frac{\pi}{12}\right) - \frac{\cos(\pi/4)}{2}\left(\frac{\pi}{12}\right)^2 = 0.497754491$$

with $\varepsilon_t = 0.449\%$. Thus, the inclusion of additional terms results in an improved estimate. The process can be continued and the results listed as in

Order	$f^{(n)}(x)$	$f^{(n)}(\pi/4)$	ε_t
0	$\cos x$	0.707106781	41.4
1	$-\sin x$	0.521986659	4.40
2	$-\cos x$	0.497754491	0.449
3	$\sin x$	0.499869147	2.62×10^{-2}
4	$\cos x$	0.500007551	1.51×10^{-3}
5	$-\sin x$	0.500000304	6.08×10^{-5}
6	$-\cos x$	0.499999988	2.44×10^{-6}

Notice that the derivatives never go to zero as would be the case for a polynomial. Therefore, each additional term results in some improvement in the estimate. However, also notice how most of the improvement comes with the initial terms. For this case, by the time we have added the third-order term, the error is reduced to 0.026%, which means that we have attained 99.974% of the true value. Consequently, although the addition of more terms will reduce the error further, the improvement becomes negligible.

4.3.2 Using the Taylor Series to Estimate Truncation Errors

Although the Taylor series will be extremely useful in estimating truncation errors throughout this book, it may not be clear to you how the expansion can actually be applied to

numerical methods. In fact, we have already done so in our example of the bungee jumper. Recall that the objective of both Examples 1.1 and 1.2 was to predict velocity as a function of time. That is, we were interested in determining $v(t)$. As specified by Eq. (4.13), $v(t)$ can be expanded in a Taylor series:

$$v(t_{i+1}) = v(t_i) + v'(t_i)(t_{i+1} - t_i) + \frac{v''(t_i)}{2!}(t_{i+1} - t_i)^2 + \cdots + R_n \quad (4.15)$$

Now let us truncate the series after the first derivative term:

$$v(t_{i+1}) = v(t_i) + v'(t_i)(t_{i+1} - t_i) + R_1 \quad (4.16)$$

Equation (4.16) can be solved for

$$v'(t_i) = \underbrace{\frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i}}_{\text{First-order approximation}} + \underbrace{\frac{R_1}{t_{i+1} - t_i}}_{\text{Truncation error}} \quad (4.17)$$

The first part of Eq. (4.17) is exactly the same relationship that was used to approximate the derivative in Example 1.2 [Eq. (1.11)]. However, because of the Taylor series approach, we have now obtained an estimate of the truncation error associated with this approximation of the derivative. Using Eqs. (4.14) and (4.17) yields

$$\frac{R_1}{t_{i+1} - t_i} = \frac{v''(\xi)}{2!}(t_{i+1} - t_i) \quad (4.18)$$

or

$$\frac{R_1}{t_{i+1} - t_i} = O(t_{i+1} - t_i) \quad (4.19)$$

Thus, the estimate of the derivative [Eq. (1.11) or the first part of Eq. (4.17)] has a truncation error of order $t_{i+1} - t_i$. In other words, the error of our derivative approximation should be proportional to the step size. Consequently, if we halve the step size, we would expect to halve the error of the derivative.

4.3.3 Numerical Differentiation

Equation (4.17) is given a formal label in numerical methods—it is called a *finite difference*. It can be represented generally as

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} + O(x_{i+1} - x_i) \quad (4.20)$$

or

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h) \quad (4.21)$$

where h is called the step size—that is, the length of the interval over which the approximation is made, $x_{i+1} - x_i$. It is termed a “forward” difference because it utilizes data at i and $i + 1$ to estimate the derivative (Fig. 4.3a).

4.3 TRUNCATION ERRORS

73

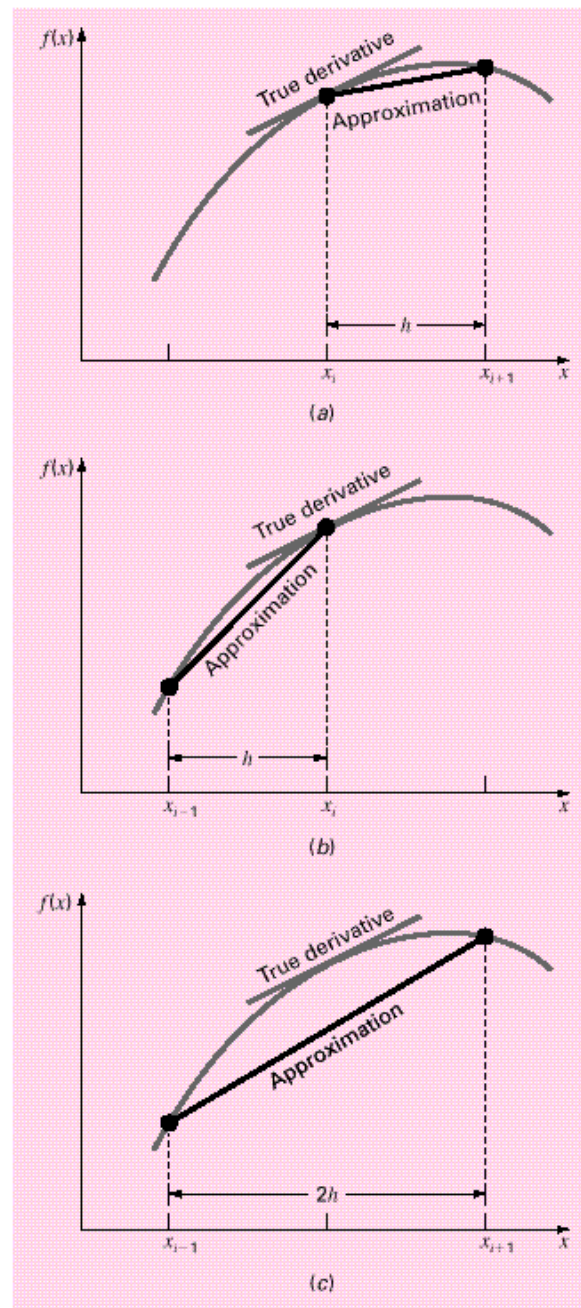


FIGURE 4.3

Graphical depiction of (·) forward, (·) backward, and (·) centered finite-difference approximations of the first derivative.

This forward divided difference is but one of many that can be developed from the Taylor series to approximate derivatives numerically. For example, backward and centered difference approximations of the first derivative can be developed in a fashion similar to the derivation of Eq. (4.17). The former utilizes values at x_{i-1} and x_i (Fig. 4.3b), whereas the latter uses values that are equally spaced around the point at which the derivative is estimated (Fig. 4.3c). More accurate approximations of the first derivative can be developed by including higher-order terms of the Taylor series. Finally, all the foregoing versions can also be developed for second, third, and higher derivatives. The following sections provide brief summaries illustrating how some of these cases are derived.

Backward Difference Approximation of the First Derivative. The Taylor series can be expanded backward to calculate a previous value on the basis of a present value, as in

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 - \dots \quad (4.22)$$

Truncating this equation after the first derivative and rearranging yields

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1}))}{h} \quad (4.23)$$

where the error is $O(h)$. See Fig. 4.3b for a graphical representation.

Centered Difference Approximation of the First Derivative. A third way to approximate the first derivative is to subtract Eq. (4.22) from the forward Taylor series expansion:

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 - \dots \quad (4.24)$$

to yield

$$f(x_{i-1}) - f(x_{i+1}) = -2f'(x_i)h + \frac{f^{(3)}(x_i)}{3!}h^3 - \dots$$

which can be solved for

$$f'(x_i) = \frac{f(x_{i-1}) - f(x_{i+1}))}{2h} - \frac{f^{(3)}(x_i)}{6}h^2 - \dots$$

or

$$f'(x_i) = \frac{f(x_{i-1}) - f(x_{i+1}))}{2h} + O(h^2) \quad (4.25)$$

Equation (4.25) is a *centered finite difference* representation of the first derivative. Notice that the truncation error is of the order of h^2 in contrast to the forward and backward approximations that were of the order of h . Consequently, the Taylor series analysis yields the practical information that the centered difference is a more accurate representation of the derivative (Fig. 4.3c). For example, if we halve the step size using a forward or backward difference, we would approximately halve the truncation error, whereas for the centered difference, the error would be quartered.

EXAMPLE 4.3 Finite-Divided-Difference Approximations of Derivatives

Problem Statement. Use forward and backward difference approximations of $O(h)$ and a centered difference approximation of $O(h^2)$ to estimate the first derivative of

$$f(x) = -0.1x^4 + 0.15x^3 + 0.5x^2 + 0.25x + 1.2$$

at $x = 0.5$ using a step size $h = 0.5$. Repeat the computation using $h = 0.25$. Note that the derivative can be calculated directly as

$$f'(x) = -0.4x^3 + 0.45x^2 + 1.0x + 0.25$$

and can be used to compute the true value as $f'(0.5) = 0.9125$.

Solution. For $h = 0.5$, the function can be employed to determine

$$x_{i-1} = 0 \quad f(x_{i-1}) = 1.2$$

$$x_i = 0.5 \quad f(x_i) = 0.925$$

$$x_{i+1} = 1.0 \quad f(x_{i+1}) = 0.2$$

These values can be used to compute the forward divided difference [Eq. (4.17)],

$$f'(0.5) \approx \frac{0.2 - 0.925}{0.5} = -1.45 \quad \epsilon_f = 58.9\%$$

the backward divided difference [Eq. (4.23)],

$$f'(0.5) \approx \frac{0.925 - 1.2}{0.5} = -0.55 \quad \epsilon_f = 39.7\%$$

and the centered divided difference [Eq. (4.25)],

$$f'(0.5) \approx \frac{0.2 - 1.2}{1.0} = -1.0 \quad \epsilon_f = 9.6\%$$

For $h = 0.25$,

$$x_{i-1} = 0.25 \quad f(x_{i-1}) = 1.10351563$$

$$x_i = 0.5 \quad f(x_i) = 0.925$$

$$x_{i+1} = 0.75 \quad f(x_{i+1}) = 0.63632813$$

which can be used to compute the forward divided difference,

$$f'(0.5) \approx \frac{0.63632813 - 0.925}{0.25} = -1.155 \quad \epsilon_f = 26.5\%$$

the backward divided difference,

$$f'(0.5) \approx \frac{0.925 - 1.10351563}{0.25} = -0.714 \quad \epsilon_f = 21.7\%$$

and the centered divided difference,

$$f'(0.5) \approx \frac{0.63632813 - 1.10351563}{0.5} = -0.934 \quad \epsilon_f = 2.4\%$$

For both step sizes, the centered difference approximation is more accurate than forward or backward differences. Also, as predicted by the Taylor series analysis, halving the step size approximately halves the error of the backward and forward differences and quarters the error of the centered difference.

Finite Difference Approximations of Higher Derivatives. Besides first derivatives, the Taylor series expansion can be used to derive numerical estimates of higher derivatives. To do this, we write a forward Taylor series expansion for $f(x_{i+2})$ in terms of $f(x_i)$:

$$f(x_{i+2}) = f(x_i) + f'(x_i)(2h) + \frac{f''(x_i)}{2!}(2h)^2 + \dots \quad (4.26)$$

Equation (4.24) can be multiplied by 2 and subtracted from Eq. (4.26) to give

$$f(x_{i+2}) - 2f(x_{i+1}) + f(x_i) = f''(x_i)h^2 + \dots$$

which can be solved for

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + O(h) \quad (4.27)$$

This relationship is called the *second forward finite difference*. Similar manipulations can be employed to derive a backward version

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2}))}{h^2} + O(h)$$

and a centered version

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1}))}{h^2} + O(h^2)$$

As was the case with the first-derivative approximations, the centered case is more accurate. Notice also that the centered version can be alternatively expressed as

$$f''(x_i) = \frac{\frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f(x_i) - f(x_{i-1}))}{h}}{h}$$

Thus, just as the second derivative is a derivative of a derivative, the second divided difference approximation is a difference of two first finite differences [recall Eq. (4.12)].

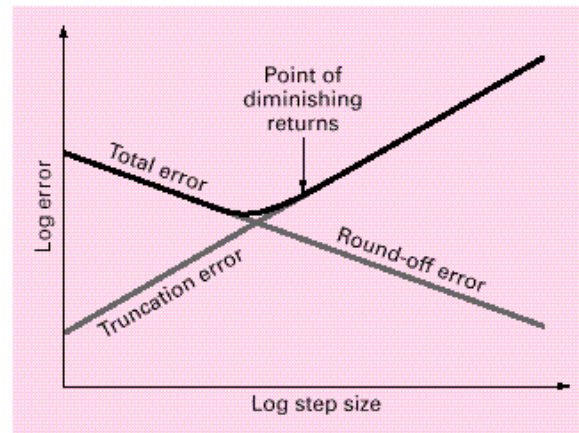
4.4 TOTAL NUMERICAL ERROR

The *total numerical error* is the summation of the truncation and round-off errors. In general, the only way to minimize round-off errors is to increase the number of significant figures of the computer. Further, we have noted that round-off error may *increase* due to subtractive cancellation or due to an increase in the number of computations in an analysis. In contrast, Example 4.3 demonstrated that the truncation error can be reduced by decreasing the step size. Because a decrease in step size can lead to subtractive cancellation or to an increase in computations, the truncation errors are *decreased* as the round-off errors are *increased*.

Therefore, we are faced by the following dilemma: The strategy for decreasing one component of the total error leads to an increase of the other component. In a computation, we could conceivably decrease the step size to minimize truncation errors only to discover that in doing so, the round-off error begins to dominate the solution and the total error

4.4 TOTAL NUMERICAL ERROR

77

**FIGURE 4.4**

A graphical depiction of the trade-off between round-off and truncation error that sometimes comes into play in the course of a numerical method. The point of diminishing returns is shown, where round-off error begins to negate the benefits of step-size reduction.

grows! Thus, our remedy becomes our problem (Fig. 4.4). One challenge that we face is to determine an appropriate step size for a particular computation. We would like to choose a large step size to decrease the amount of calculations and round-off errors without incurring the penalty of a large truncation error. If the total error is as shown in Fig. 4.4, the challenge is to identify the point of diminishing returns where round-off error begins to negate the benefits of step-size reduction.

When using MATLAB, such situations are relatively uncommon because of its 15-digit precision. Nevertheless, they sometimes do occur and suggest a sort of “numerical uncertainty principle” that places an absolute limit on the accuracy that may be obtained using certain computerized numerical methods.

4.4.1 Control of Numerical Errors

For most practical cases, we do not know the exact error associated with numerical methods. The exception, of course, is when we know the exact solution, which makes our numerical approximations unnecessary. Therefore, for most engineering and scientific applications we must settle for some estimate of the error in our calculations.

There are no systematic and general approaches to evaluating numerical errors for all problems. In many cases error estimates are based on the experience and judgment of the engineer or scientist.

Although error analysis is to a certain extent an art, there are several practical programming guidelines we can suggest. First and foremost, avoid subtracting two nearly equal numbers. Loss of significance almost always occurs when this is done. Sometimes you can rearrange or reformulate the problem to avoid subtractive cancellation. If this is not possible, you may want to use extended-precision arithmetic. Furthermore, when adding and subtracting numbers, it is best to sort the numbers and work with the smallest numbers first. This avoids loss of significance.

Beyond these computational hints, one can attempt to predict total numerical errors using theoretical formulations. The Taylor series is our primary tool for analysis of such errors. Prediction of total numerical error is very complicated for even moderately sized problems and tends to be pessimistic. Therefore, it is usually attempted for only small-scale tasks.

The tendency is to push forward with the numerical computations and try to estimate the accuracy of your results. This can sometimes be done by seeing if the results satisfy some condition or equation as a check. Or it may be possible to substitute the results back into the original equation to check that it is actually satisfied.

Finally you should be prepared to perform numerical experiments to increase your awareness of computational errors and possible ill-conditioned problems. Such experiments may involve repeating the computations with a different step size or method and comparing the results. We may employ sensitivity analysis to see how our solution changes when we change model parameters or input values. We may want to try different numerical algorithms that have different theoretical foundations, are based on different computational strategies, or have different convergence properties and stability characteristics.

When the results of numerical computations are extremely critical and may involve loss of human life or have severe economic ramifications, it is appropriate to take special precautions. This may involve the use of two or more independent groups to solve the same problem so that their results can be compared.

The roles of errors will be a topic of concern and analysis in all sections of this book. We will leave these investigations to specific sections.

4.5 BLUNDERS, MODEL ERRORS, AND DATA UNCERTAINTY

Although the following sources of error are not directly connected with most of the numerical methods in this book, they can sometimes have great impact on the success of a modeling effort. Thus, they must always be kept in mind when applying numerical techniques in the context of real-world problems.

4.5.1 Blunders

We are all familiar with gross errors, or blunders. In the early years of computers, erroneous numerical results could sometimes be attributed to malfunctions of the computer itself. Today, this source of error is highly unlikely, and most blunders must be attributed to human imperfection.

Blunders can occur at any stage of the mathematical modeling process and can contribute to all the other components of error. They can be avoided only by sound knowledge of fundamental principles and by the care with which you approach and design your solution to a problem.

Blunders are usually disregarded in discussions of numerical methods. This is no doubt due to the fact that, try as we may, mistakes are to a certain extent unavoidable. However, we believe that there are a number of ways in which their occurrence can be minimized. In particular, the good programming habits that were outlined in Chap. 3 are extremely useful for mitigating programming blunders. In addition, there are usually simple ways to check whether a particular numerical method is working properly. Throughout this book, we discuss ways to check the results of numerical calculations.

4.5.2 Model Errors

Model errors relate to bias that can be ascribed to incomplete mathematical models. An example of a negligible model error is the fact that Newton's second law does not account for relativistic effects. This does not detract from the adequacy of the solution in Example 1.1 because these errors are minimal on the time and space scales associated with the bungee jumper problem.

However, suppose that air resistance is not proportional to the square of the fall velocity, as in Eq. (1.7), but is related to velocity and other factors in a different way. If such were the case, both the analytical and numerical solutions obtained in Chap. 1 would be erroneous because of model error. You should be cognizant of this type of error and realize that, if you are working with a poorly conceived model, no numerical method will provide adequate results.

4.5.3 Data Uncertainty

Errors sometimes enter into an analysis because of uncertainty in the physical data on which a model is based. For instance, suppose we wanted to test the bungee jumper model by having an individual make repeated jumps and then measuring his or her velocity after a specified time interval. Uncertainty would undoubtedly be associated with these measurements, as the parachutist would fall faster during some jumps than during others. These errors can exhibit both inaccuracy and imprecision. If our instruments consistently underestimate or overestimate the velocity, we are dealing with an inaccurate, or biased, device. On the other hand, if the measurements are randomly high and low, we are dealing with a question of precision.

Measurement errors can be quantified by summarizing the data with one or more well-chosen statistics that convey as much information as possible regarding specific characteristics of the data. These descriptive statistics are most often selected to represent (1) the location of the center of the distribution of the data and (2) the degree of spread of the data. As such, they provide a measure of the bias and imprecision, respectively. We will return to the topic of characterizing data uncertainty when we discuss regression in Chaps. 12 and 13.

Although you must be cognizant of blunders, model errors, and uncertain data, the numerical methods used for building models can be studied, for the most part, independently of these errors. Therefore, for most of this book, we will assume that we have not made gross errors, we have a sound model, and we are dealing with error-free measurements. Under these conditions, we can study numerical errors without complicating factors.

PROBLEMS

4.1 The derivative of $f(x) = 1/(1 - 3x^2)$ is given by

$$\frac{6x}{(1 - 3x^2)^2}$$

Do you expect to have difficulties evaluating this function at $x = 0.577$? Try it using 3- and 4-digit arithmetic with chopping.

4.2 (a) Evaluate the polynomial

$$y = x^3 + 7x^2 + 8x + 0.35$$

at $x = 1.37$. Use 3-digit arithmetic with chopping. Evaluate the percent relative error.

(b) Repeat (a) but express y as

$$y = ((x - 7)x - 8)x - 0.35$$

Evaluate the error and compare with part (a).

4.3 The following infinite series can be used to approximate e^x :

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

(a) Prove that this Maclaurin series expansion is a special case of the Taylor series expansion (Eq. 4.13) with $x_i = 0$ and $h = x$.

(b) Use the Taylor series to estimate $f(x) = e^{-x}$ at $x_{i+1} = 1$ for $x_i = 0.25$. Employ the zero-, first-, second-, and third-order versions and compute the $|\varepsilon_t|$ for each case.

4.4 The Maclaurin series expansion for $\cos x$ is

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

Starting with the simplest version, $\cos x = 1$, add terms one at a time to estimate $\cos(\pi/4)$. After each new term is added, compute the true and approximate percent relative errors. Use your pocket calculator or MATLAB to determine the true value. Add terms until the absolute value of the approximate error estimate falls below an error criterion conforming to two significant figures.

4.5 Perform the same computation as in Prob. 4.4, but use the Maclaurin series expansion for the $\sin x$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

to estimate $\sin(\pi/4)$.

4.6 Use zero- through third-order Taylor series expansions to predict $f(2)$ for

$$f(x) = 25x^3 - 6x^2 - 7x + 88$$

using a base point at $x = 1$. Compute the true percent relative error ε_t for each approximation.

4.7 Use zero- through fourth-order Taylor series expansions to predict $f(3)$ for $f(x) = \ln x$ using a base point at $x = 1$.

Compute the true percent relative error ε_t for each approximation. Discuss the meaning of the results.

4.8 Use forward and backward difference approximations of $O(h)$ and a centered difference approximation of $O(h^2)$ to estimate the first derivative of the function examined in Prob. 4.6. Evaluate the derivative at $x = 2$ using a step size of $h = 0.25$. Compare your results with the true value of the derivative. Interpret your results on the basis of the remainder term of the Taylor series expansion.

4.9 Use a centered difference approximation of $O(h^2)$ to estimate the second derivative of the function examined in Prob. 4.6. Perform the evaluation at $x = 2$ using step sizes of $h = 0.2$ and 0.1 . Compare your estimates with the true value of the second derivative. Interpret your results on the basis of the remainder term of the Taylor series expansion.

4.10 If $|x| < 1$, it is known that

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + \cdots$$

Repeat Prob. 4.4 for this series for $x = 0.1$.

4.11 To calculate a planet's space coordinates, we have to solve the function

$$f(x) = x + 1 + 0.5 \sin x$$

Let the base point be $x_i = \pi/2$ on the interval $[0, \pi]$. Determine the highest-order Taylor series expansion resulting in a maximum error of 0.015 on the specified interval. The error is equal to the absolute value of the difference between the given function and the specific Taylor series expansion. (Hint: Solve graphically.)

4.12 Consider the function $f(x) = x^3 + 2x + 4$ on the interval $[-2, 2]$ with $h = 0.25$. Use the forward, backward, and centered finite difference approximations for the first and second derivatives so as to graphically illustrate which approximation is most accurate. Graph all three first-derivative finite difference approximations along with the theoretical, and do the same for the second derivative as well.

4.13 Develop your own M-file to compute the machine epsilon. Test it by comparing it to the result obtained with the built-in function `eps`.

Roots of Equations: Bracketing Methods

CHAPTER OBJECTIVES

The primary objective of this chapter is to acquaint you with bracketing methods for finding the root of a single nonlinear equation. Specific objectives and topics covered are

- Understanding what roots problems are and where they occur in engineering and science.
- Knowing how to determine a root graphically.
- Understanding the incremental search method and its shortcomings.
- Knowing how to solve a roots problem with the bisection method.
- Knowing how to estimate the error of bisection and why it differs from error estimates for other types of root location algorithms.
- Understanding false position and how it differs from bisection.

YOU'VE GOT A PROBLEM

Medical studies have established that a bungee jumper's chances of sustaining a significant vertebrae injury increase significantly if the free-fall velocity exceeds 36 m/s after 4 s of free fall. Your boss at the bungee-jumping company wants you to determine the mass at which this criterion is exceeded given a drag coefficient of 0.25 kg/m.

You know from your previous studies that the following analytical solution can be used to predict fall velocity as a function of time:

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) \quad (5.1)$$

Try as you might, you cannot manipulate this equation to explicitly solve for m —that is, you cannot isolate the mass on the left side of the equation.

An alternative way of looking at the problem involves subtracting $v(t)$ from both sides to give a new function:

$$f(m) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) - v(t) \quad (5.2)$$

Now we can see that the answer to the problem is the value of m that makes the function equal to zero. Hence, we call this a “roots” problem. This chapter will introduce you to how the computer is used as a tool to obtain such solutions.

5.1 INTRODUCTION AND BACKGROUND

5.1.1 What Are Roots?

Years ago, you learned to use the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (5.3)$$

to solve

$$f(x) = ax^2 + bx + c = 0 \quad (5.4)$$

The values calculated with Eq. (5.3) are called the “roots” of Eq. (5.4). They represent the values of x that make Eq. (5.4) equal to zero. For this reason, roots are sometimes called the *zeros* of the equation.

Although the quadratic formula is handy for solving Eq. (5.4), there are many other functions for which the root cannot be determined so easily. Before the advent of digital computers, there were a number of ways to solve for the roots of such equations. For some cases, the roots could be obtained by direct methods, as with Eq. (5.3). Although there were equations like this that could be solved directly, there were many more that could not. In such instances, the only alternative is an approximate solution technique.

One method to obtain an approximate solution is to plot the function and determine where it crosses the x axis. This point, which represents the x value for which $f(x) = 0$, is the root. Although graphical methods are useful for obtaining rough estimates of roots, they are limited because of their lack of precision. An alternative approach is to use *trial and error*. This “technique” consists of guessing a value of x and evaluating whether $f(x)$ is zero. If not (as is almost always the case), another guess is made, and $f(x)$ is again evaluated to determine whether the new value provides a better estimate of the root. The process is repeated until a guess results in an $f(x)$ that is close to zero.

Such haphazard methods are obviously inefficient and inadequate for the requirements of engineering practice. Numerical methods represent alternatives that are also approximate but employ systematic strategies to home in on the true root. As elaborated in the following pages, the combination of these systematic methods and computers makes the solution of most applied roots-of-equations problems a simple and efficient task.

5.1.2 Roots of Equations and Engineering Practice

Although they arise in other problem contexts, roots of equations frequently occur in the area of engineering design. Table 5.1 lists a number of fundamental principles that are routinely used in design work. As introduced in Chap. 1, mathematical equations or models

5.2 GRAPHICAL METHODS

83

TABLE 5.1 Fundamental principles used in engineering design problems.

Fundamental Principle	Dependent Variable	Independent Variable	Parameters
Heat balance	Temperature	Time and position	Thermal properties of material, system geometry
Mass balance	Concentration or quantity of mass	Time and position	Chemical behavior of material, mass transfer, system geometry
Force balance	Magnitude and direction of forces	Time and position	Strength of material, structural properties, system geometry
Energy balance	Changes in kinetic and potential energy	Time and position	Thermal properties, mass of material, system geometry
Newton's laws of motion	Acceleration, velocity, or location	Time and position	Mass of material, system geometry, dissipative parameters
Kirchhoff's laws	Currents and voltages	Time	Electrical properties [resistance, capacitance, inductance]

derived from these principles are employed to predict dependent variables as a function of independent variables, forcing functions, and parameters. Note that in each case, the dependent variables reflect the state or performance of the system, whereas the parameters represent its properties or composition.

An example of such a model is the equation for the bungee jumper's velocity. If the parameters are known, Eq. (5.1) can be used to predict the jumper's velocity. Such computations can be performed directly because v is expressed *explicitly* as a function of the model parameters. That is, it is isolated on one side of the equal sign.

However, as posed at the start of the chapter, suppose that we had to determine the mass for a jumper with a given drag coefficient to attain a prescribed velocity in a set time period. Although Eq. (5.1) provides a mathematical representation of the interrelationship among the model variables and parameters, it cannot be solved explicitly for mass. In such cases, m is said to be *implicit*.

This represents a real dilemma, because many engineering design problems involve specifying the properties or composition of a system (as represented by its parameters) to ensure that it performs in a desired manner (as represented by its variables). Thus, these problems often require the determination of implicit parameters.

The solution to the dilemma is provided by numerical methods for roots of equations. To solve the problem using numerical methods, it is conventional to reexpress Eq. (5.1) by subtracting the dependent variable v from both sides of the equation to give Eq. (5.2). The value of m that makes $f(m) = 0$ is, therefore, the root of the equation. This value also represents the mass that solves the design problem.

The following pages deal with a variety of numerical and graphical methods for determining roots of relationships such as Eq. (5.2). These techniques can be applied to many other problems confronted routinely in engineering and science.

5.2 GRAPHICAL METHODS

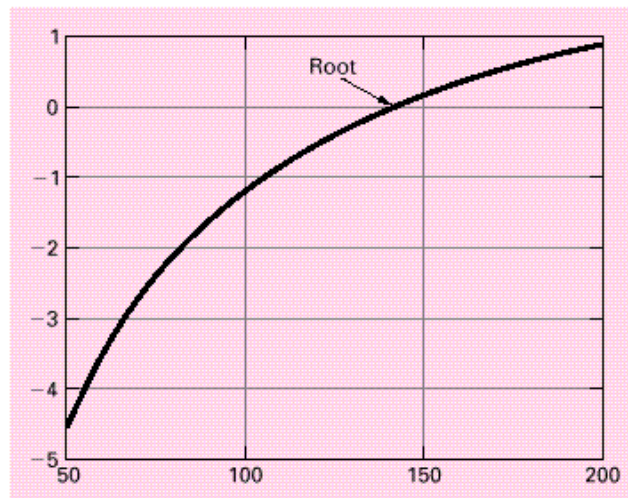
A simple method for obtaining an estimate of the root of the equation $f(x) = 0$ is to make a plot of the function and observe where it crosses the x axis. This point, which represents the x value for which $f(x) = 0$, provides a rough approximation of the root.

EXAMPLE 5.1 The Graphical Approach

Problem Statement. Use the graphical approach to determine the mass of the bungee jumper with a drag coefficient of 0.25 kg/m to have a velocity of 36 m/s after 4 s of free fall. Note: The acceleration of gravity is 9.81 m/s².

Solution. The following MATLAB session sets up a plot of Eq. (5.2) versus mass:

```
>> cd = 0.25; g = 9.81; v = 36; t = 4;  
>> mp = linspace(50,200);  
>> fp = sqrt(g*mp/cd).*tanh(sqrt(g*cd./mp)*t)-v;  
>> plot(mp,fp),grid
```



The function crosses the m axis between 140 and 150 kg. Visual inspection of the plot provides a rough estimate of the root of 145 kg (about 320 lb). The validity of the graphical estimate can be checked by substituting it into Eq. (5.2) to yield

```
>> sqrt(g*145/cd)*tanh(sqrt(g*cd/145)*t)-v  
ans =  
    0.0456
```

which is close to zero. It can also be checked by substituting it into Eq. (5.1) along with the parameter values from this example to give

```
>> sqrt(g*145/cd)*tanh(sqrt(g*cd/145)*t)  
ans =  
    36.0456
```

which is close to the desired fall velocity of 36 m/s.

Graphical techniques are of limited practical value because they are not very precise. However, graphical methods can be utilized to obtain rough estimates of roots. These

5.3 BRACKETING METHODS AND INITIAL GUESSES

85

estimates can be employed as starting guesses for numerical methods discussed in this chapter.

Aside from providing rough estimates of the root, graphical interpretations are useful for understanding the properties of the functions and anticipating the pitfalls of the numerical methods. For example, Fig. 5.1 shows a number of ways in which roots can occur (or be absent) in an interval prescribed by a lower bound x_l and an upper bound x_u . Figure 5.1*b* depicts the case where a single root is bracketed by negative and positive values of $f(x)$. However, Fig. 5.1*d*, where $f(x_l)$ and $f(x_u)$ are also on opposite sides of the x axis, shows three roots occurring within the interval. In general, if $f(x_l)$ and $f(x_u)$ have opposite signs, there are an odd number of roots in the interval. As indicated by Fig. 5.1*a* and *c*, if $f(x_l)$ and $f(x_u)$ have the same sign, there are either no roots or an even number of roots between the values.

Although these generalizations are usually true, there are cases where they do not hold. For example, functions that are tangential to the x axis (Fig. 5.2*a*) and discontinuous functions (Fig. 5.2*b*) can violate these principles. An example of a function that is tangential to the axis is the cubic equation $f(x) = (x - 2)(x - 2)(x - 4)$. Notice that $x - 2$ makes two terms in this polynomial equal to zero. Mathematically, $x - 2$ is called a *multiple root*. Although they are beyond the scope of this book, there are special techniques that are expressly designed to locate multiple roots (Chapra and Canale, 2002).

The existence of cases of the type depicted in Fig. 5.2 makes it difficult to develop foolproof computer algorithms guaranteed to locate all the roots in an interval. However, when used in conjunction with graphical approaches, the methods described in the following sections are extremely useful for solving many problems confronted routinely by engineers, scientists, and applied mathematicians.

5.3 BRACKETING METHODS AND INITIAL GUESSES

If you had a roots problem in the days before computing, you'd often be told to use "trial and error" to come up with the root. That is, you'd repeatedly make guesses until the function was sufficiently close to zero. The process was greatly facilitated by the advent of software tools such as spreadsheets. By allowing you to make many guesses rapidly, such tools can actually make the trial-and-error approach attractive for some problems.

But, for many other problems, it is preferable to have methods that come up with the correct answer automatically. Interestingly, as with trial and error, these approaches require an initial "guess" to get started. Then they systematically home in on the root in an iterative fashion.

The two major classes of methods available are distinguished by the type of initial guess. They are

- *Bracketing methods.* As the name implies, these are based on two initial guesses that "bracket" the root—that is, are on either side of the root.
- *Open methods.* These methods can involve one or more initial guesses, but there is no need for them to bracket the root.

For well-posed problems, the bracketing methods always work but converge slowly (i.e., they typically take more iterations to home in on the answer). In contrast, the open methods do not always work (i.e., they can diverge), but when they do they usually converge quicker.

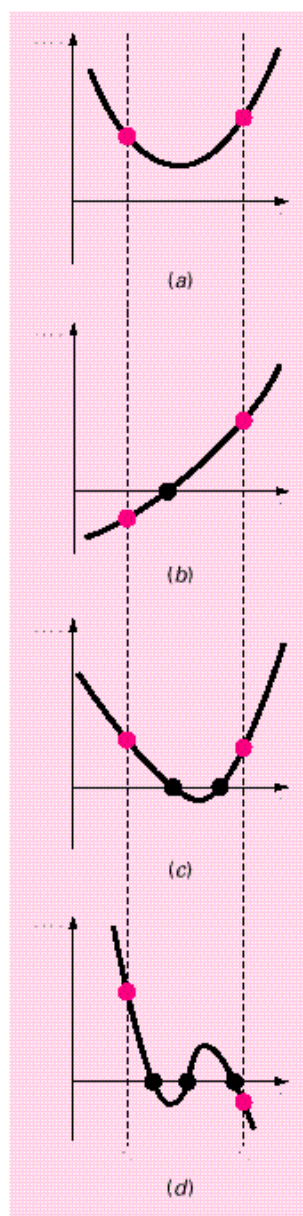


FIGURE 5.1

Illustration of a number of general ways that a root may occur in an interval prescribed by a lower bound x_l and an upper bound x_u . Parts (a) and (b) indicate that if both $f(x_l)$ and $f(x_u)$ have the same sign, either there will be no roots or there will be an even number of roots within the interval. Parts (c) and (d) indicate that if the function has different signs at the end points, there will be an odd number of roots in the interval.

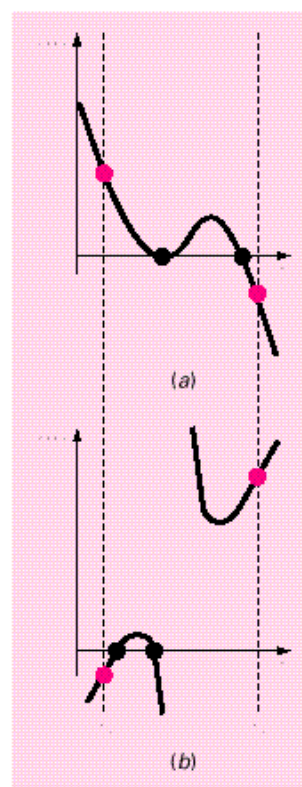
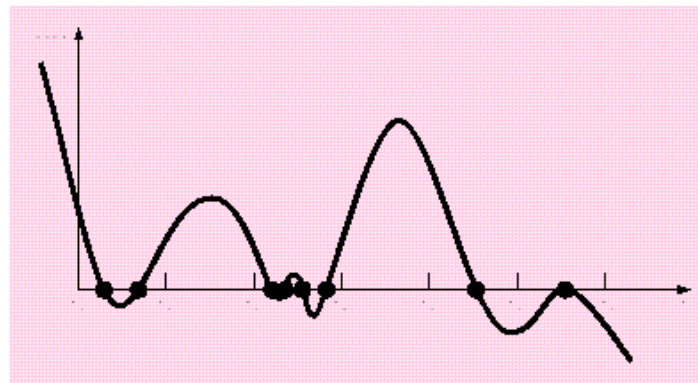


FIGURE 5.2

Illustration of some exceptions to the general cases depicted in Fig. 5.1. (a) Multiple roots that occur when the function is tangential to the x axis. For this case, although the end points are of opposite signs, there are an even number of axis interceptions for the interval. (b) Discontinuous functions where end points of opposite sign bracket an even number of roots. Special strategies are required for determining the roots for these cases.

**FIGURE 5.3**

Cases where roots could be missed because the incremental length of the search procedure is too large. Note that the last root on the right is multiple and would be missed regardless of the increment length.

In both cases, initial guesses are required. These may naturally arise from the physical context you are analyzing. However, in other cases, good initial guesses may not be obvious. In such cases, automated approaches to obtain guesses would be useful. The following section describes one such approach, the incremental search.

5.3.1 Incremental Search

When applying the graphical technique in Example 5.1, you observed that $f(x)$ changed sign on opposite sides of the root. In general, if $f(x)$ is real and continuous in the interval from x_l to x_u and $f(x_l)$ and $f(x_u)$ have opposite signs, that is,

$$f(x_l)f(x_u) < 0$$

then there is at least one real root between x_l and x_u .

Incremental search methods capitalize on this observation by locating an interval where the function changes sign. A potential problem with an incremental search is the choice of the increment length. If the length is too small, the search can be very time consuming. On the other hand, if the length is too great, there is a possibility that closely spaced roots might be missed (Fig. 5.3). The problem is compounded by the possible existence of multiple roots.

An M-file can be developed¹ that implements an incremental search to locate the roots of a function `func` within the range from `xmin` to `xmax` (Fig. 5.4). An optional argument `ns` allows the user to specify the number of intervals within the range. If `ns` is omitted, it is automatically set to 50. A `for` loop is used to step through each interval. In the event that a sign change occurs, the upper and low bounds are stored in an array `xb`.

¹ This function is a modified version of an M-file originally presented by Recktenwald (2000).

```
function xb = incsearch(func,xmin,xmax,ns)
% incsearch(func,xmin,xmax,ns):
%   finds brackets of x that contain sign changes of
%   a function on an interval
% input:
%   func = name of function
%   xmin, xmax = endpoints of interval
%   ns = (optional) number of subintervals along x
%         used to search for brackets
% output:
%   xb(k,1) is the lower bound of the kth sign change
%   xb(k,2) is the upper bound of the kth sign change
%   If no brackets found, xb = [].

if nargin < 4, ns = 50; end %if ns blank set to 50

% Incremental search
x = linspace(xmin,xmax,ns);
f = feval(func,x);
nb = 0; xb = []; %xb is null unless sign change detected
for k = 1:length(x)-1
    if sign(f(k)) ~= sign(f(k+1)) %check for sign change
        nb = nb + 1;
        xb(nb,1) = x(k);
        xb(nb,2) = x(k+1);
    end
end

if isempty(xb) %display that no brackets were found
    disp('no brackets found')
    disp('check interval or increase ns')
else
    disp('number of brackets:') %display number of brackets
    disp(nb)
end
```

FIGURE 5.4

An M-file to implement an incremental search.

EXAMPLE 5.2 Incremental Search

Problem Statement. Use the M-file `incsearch` (Fig. 5.4) to identify brackets within the interval $[3, 6]$ for the function:

$$f(x) = \sin(10x) \cdot \cos(3x)$$

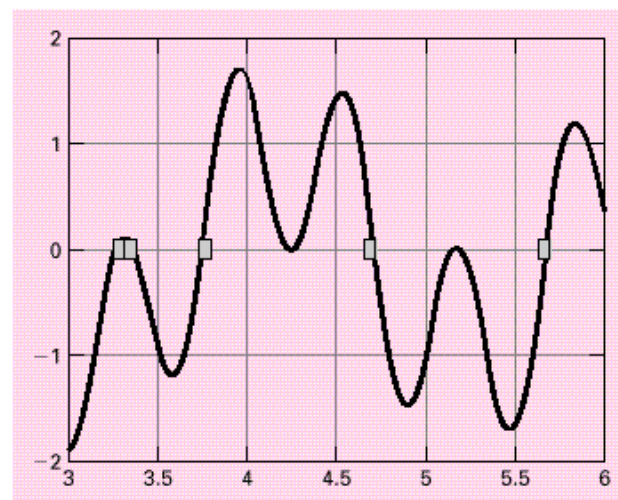
5.3 BRACKETING METHODS AND INITIAL GUESSES

89

Solution. The MATLAB session using the default number of intervals (50) is

```
>> incsearch(inline('sin(10*x)+cos(3*x)'),3,6)
number of possible roots:
    5
ans =
    3.2449    3.3061
    3.3061    3.3673
    3.7347    3.7959
    4.6531    4.7143
    5.6327    5.6939
```

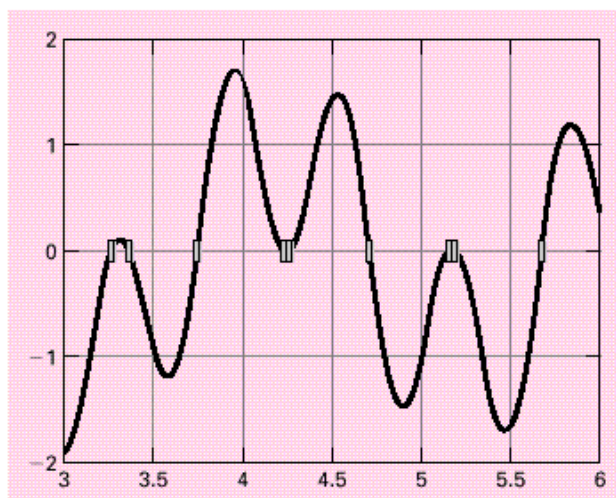
A plot of the function along with the root locations is shown here.



Although five sign changes are detected, because the subintervals are too wide, the function misses possible roots at $x \approx 4.25$ and 5.2 . These possible roots look like they might be double roots. However, by using the zoom in tool, it is clear that each represents two real roots that are very close together. The function can be run again with more subintervals with the result that all nine sign changes are located

```
>> incsearch(inline('sin(10*x)+cos(3*x)'),3,6,100)
number of possible roots:
    9
ans =
    3.2424    3.2727
    3.3636    3.3939
    3.7273    3.7576
    4.2121    4.2424
    4.2424    4.2727
    4.6970    4.7273
```

```
5.1515    5.1818
5.1818    5.2121
5.6667    5.6970
```



The foregoing example illustrates that brute-force methods such as incremental search are not foolproof. You would be wise to supplement such automatic techniques with any other information that provides insight into the location of the roots. Such information can be found by plotting the function and through understanding the physical problem from which the equation originated.

5.4 BISECTION

The *bisection method* is a variation of the incremental search method in which the interval is always divided in half. If a function changes sign over an interval, the function value at the midpoint is evaluated. The location of the root is then determined as lying within the subinterval where the sign change occurs. The subinterval then becomes the interval for the next iteration. The process is repeated until the root is known to the required precision. A graphical depiction of the method is provided in Fig. 5.5. The following example goes through the actual computations involved in the method.

EXAMPLE 5.3 The Bisection Method

Problem Statement. Use bisection to solve the same problem approached graphically in Example 5.1.

Solution. The first step in bisection is to guess two values of the unknown (in the present problem, m) that give values for $f(m)$ with different signs. From the graphical solution in Example 5.1, we can see that the function changes sign between values of 50 and 200. The plot obviously suggests better initial guesses, say 140 and 150, but for illustrative purposes let's assume we don't have the benefit of the plot and have made conservative guesses.

5.4 BISECTION

91

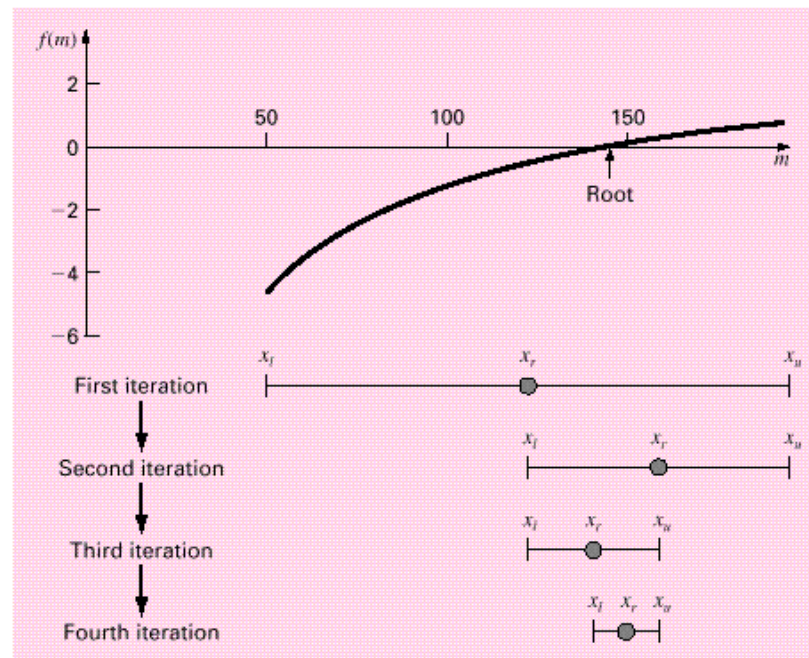


FIGURE 5.5

A graphical depiction of the bisection method. This plot corresponds to the first four iterations from Example 5.3.

Therefore, the initial estimate of the root x_r lies at the midpoint of the interval

$$x_r = \frac{50 + 200}{2} = 125$$

Note that the exact value of the root is 142.7376. This means that the value of 125 calculated here has a true percent relative error of

$$\epsilon_r = \left| \frac{142.7376 - 125}{142.7376} \right| \cdot 100\% = 12.43\%$$

Next we compute the product of the function value at the lower bound and at the midpoint:

$$f(50)f(125) = 4.579(-0.409) = -1.871$$

which is greater than zero, and hence no sign change occurs between the lower bound and the midpoint. Consequently, the root must be located in the upper interval between 125 and 200. Therefore, we create a new interval by redefining the lower bound as 125.

At this point, the new interval extends from $x_l = 125$ to $x_u = 200$. A revised root estimate can then be calculated as

$$x_r = \frac{125 + 200}{2} = 162.5$$

which represents a true percent error of $\epsilon_t = 13.85\%$. The process can be repeated to obtain refined estimates. For example,

$$f(125)f(162.5) = -0.409(0.359) = -0.147$$

Therefore, the root is now in the lower interval between 125 and 162.5. The upper bound is redefined as 162.5, and the root estimate for the third iteration is calculated as

$$x_r = \frac{125 + 162.5}{2} = 143.75$$

which represents a percent relative error of $\epsilon_t = 0.709\%$. The method can be repeated until the result is accurate enough to satisfy your needs.

We ended Example 5.3 with the statement that the method could be continued to obtain a refined estimate of the root. We must now develop an objective criterion for deciding when to terminate the method.

An initial suggestion might be to end the calculation when the error falls below some prespecified level. For instance, in Example 5.3, the true relative error dropped from 12.43 to 0.709% during the course of the computation. We might decide that we should terminate when the error drops below, say, 0.5%. This strategy is flawed because the error estimates in the example were based on knowledge of the true root of the function. This would not be the case in an actual situation because there would be no point in using the method if we already knew the root.

Therefore, we require an error estimate that is not contingent on foreknowledge of the root. One way to do this is by estimating an approximate percent relative error as in [recall Eq. (4.5)]

$$\epsilon_a = \left| \frac{x_r^{\text{new}} - x_r^{\text{old}}}{x_r^{\text{new}}} \right| 100\% \quad (5.5)$$

where x_r^{new} is the root for the present iteration and x_r^{old} is the root from the previous iteration. When ϵ_a becomes less than a prespecified stopping criterion ϵ_s , the computation is terminated.

EXAMPLE 5.4 Error Estimates for Bisection

Problem Statement. Continue Example 5.3 until the approximate error falls below a stopping criterion of $\epsilon_s = 0.5\%$. Use Eq. (5.5) to compute the errors.

Solution. The results of the first two iterations for Example 5.3 were 125 and 162.5. Substituting these values into Eq. (5.5) yields

$$\epsilon_a = \left| \frac{162.5 - 125}{162.5} \right| 100\% = 23.08\%$$

Recall that the true percent relative error for the root estimate of 162.5 was 13.85%. Therefore, ϵ_a is greater than ϵ_t . This behavior is manifested for the other iterations:

5.4 BISECTION

93

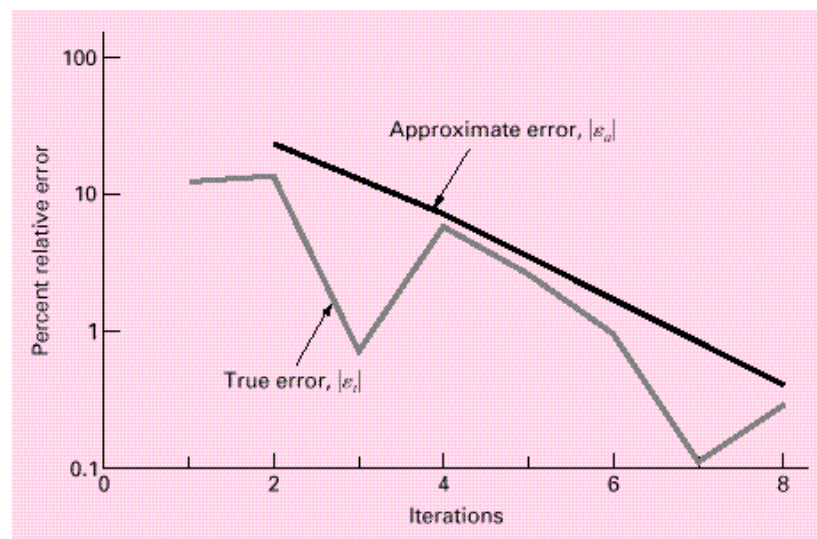
Iteration	x_1	x_2	x_s	ϵ_a (%)	ϵ_s (%)
1	50	200	125		12.43
2	125	200	162.5	23.08	13.85
3	125	162.5	143.75	13.04	0.71
4	125	143.75	134.375	6.98	5.86
5	134.375	143.75	139.0625	3.37	2.58
6	139.0625	143.75	141.4063	1.66	0.93
7	141.4063	143.75	142.5781	0.82	0.11
8	142.5781	143.75	143.1641	0.41	0.30

Thus after eight iterations ϵ_a finally falls below $\epsilon_s = 0.5\%$, and the computation can be terminated.

These results are summarized in Fig. 5.6. The “ragged” nature of the true error is due to the fact that, for bisection, the true root can lie anywhere within the bracketing interval. The true and approximate errors are far apart when the interval happens to be centered on the true root. They are close when the true root falls at either end of the interval.

FIGURE 5.6

Errors for the bisection method. True and approximate errors are plotted versus the number of iterations.



Although the approximate error does not provide an exact estimate of the true error, Fig. 5.6 suggests that ϵ_a captures the general downward trend of ϵ_t . In addition, the plot exhibits the extremely attractive characteristic that ϵ_a is always greater than ϵ_t . Thus, when ϵ_a falls below ϵ_s , the computation could be terminated with confidence that the root is known to be at least as accurate as the prespecified acceptable level.

While it is dangerous to draw general conclusions from a single example, it can be demonstrated that ϵ_a will always be greater than ϵ_t for bisection. This is due to the fact

that each time an approximate root is located using bisection as $x_r = (x_l + x_u)/2$, we know that the true root lies somewhere within an interval of $\Delta x = x_u - x_l$. Therefore, the root must lie within $\Delta x/2$ of our estimate. For instance, when Example 5.4 was terminated, we could make the definitive statement that

$$x_r = 143.1641 \pm \frac{143.7500 - 142.5781}{2} = 143.1641 \pm 0.5859$$

In essence, Eq. (5.5) provides an upper bound on the true error. For this bound to be exceeded, the true root would have to fall outside the bracketing interval, which by definition could never occur for bisection. Other root-locating techniques do not always behave as nicely. Although bisection is generally slower than other methods, the neatness of its error analysis is a positive feature that makes it attractive for certain engineering and scientific applications.

Another benefit of the bisection method is that the number of iterations required to attain an absolute error can be computed *a priori*—that is, before starting the computation. This can be seen by recognizing that before starting the technique, the absolute error is

$$E_a^0 = \frac{x_u^0 - x_l^0}{2} = \frac{\Delta x^0}{2}$$

where the superscript designates the iteration. Hence, before starting the method we are at the “zero iteration.” After the first iteration, the error becomes

$$E_a^1 = \frac{\Delta x^0}{4}$$

Because each succeeding iteration halves the error, a general formula relating the error and the number of iterations n is

$$E_a^n = \frac{\Delta x^0}{2^{n+1}}$$

If $E_{a,d}$ is the desired error, this equation can be solved for²

$$n = 1 + \frac{\log(\Delta x^0/E_{a,d})}{\log 2} = 1 + \log_2\left(\frac{\Delta x^0}{E_{a,d}}\right) \quad (5.6)$$

Let’s test the formula. For Example 5.4, the initial interval was $\Delta x_0 = 200 - 50 = 150$. After eight iterations, the absolute error was

$$E_a = \frac{143.7500 - 142.5781}{2} = 0.5859$$

We can substitute these values into Eq. (5.6) to give

$$n = 1 + \log_2\left(\frac{150/0.5859}{2}\right) = 8$$

Thus, if we knew beforehand that an error of less than 0.5859 was acceptable, the formula tells us that eight iterations would yield the desired result.

Although we have emphasized the use of relative errors for obvious reasons, there will be cases where (usually through knowledge of the problem context) you will be able to

² MATLAB provides the `log2` function to evaluate the base-2 logarithm directly. If the pocket calculator or computer language you are using does not include the base-2 logarithm as an intrinsic function, this equation shows a handy way to compute it. In general, $\log_b(x) = \log(x)/\log(b)$.

5.4 BISECTION

95

specify an absolute error. For these cases, bisection along with Eq. (5.6) can provide a useful root location algorithm.

5.4.1 MATLAB M-file:

An M-file to implement bisection is displayed in Fig. 5.7. It is passed the function (*func*) along with lower (*xl*) and upper (*xu*) guesses. In addition an optional stopping criterion

FIGURE 5.7

An M-file to implement the bisection method.

```
function root = bisection(func,xl,xu,es,maxit)
% bisection(func,xl,xu,es,maxit):
%   uses bisection method to find the root of a function
% input:
%   func = name of function
%   xl, xu = lower and upper guesses
%   es = (optional) stopping criterion (%)
%   maxit = (optional) maximum allowable iterations
% output:
%   root = real root

if func(xl)*func(xu)>0 %if guesses do not bracket a sign
    error('no bracket') %change, display an error message
    return %and terminate
end
% if necessary, assign default values
if nargin<5, maxit = 50; end %if maxit blank set to 50
if nargin<4, es = 0.001; end %if es blank set to 0.001

% bisection
iter = 0;
xr = xl;
while (1)
    xrold = xr;
    xr = (xl + xu)/2;
    iter = iter + 1;
    if xr ~= 0, ea = abs((xr - xrold)/xr) * 100; end
    test = func(xl)*func(xr);
    if test < 0
        xu = xr;
    elseif test > 0
        xl = xr;
    else
        ea = 0;
    end
    if ea <= es | iter >= maxit, break, end
end
root = xr;
```

(*es*) and maximum iterations (*maxit*) can be entered. The function first checks whether the initial guesses bracket a sign change. If not, an error message is displayed and the function is terminated. It also supplies default values if *maxit* and *es* are not supplied. Then a `while . . . break` loop is employed to implement the bisection algorithm until the approximate error falls below *es* or the iterations exceed *maxit*.

5.5 FALSE POSITION

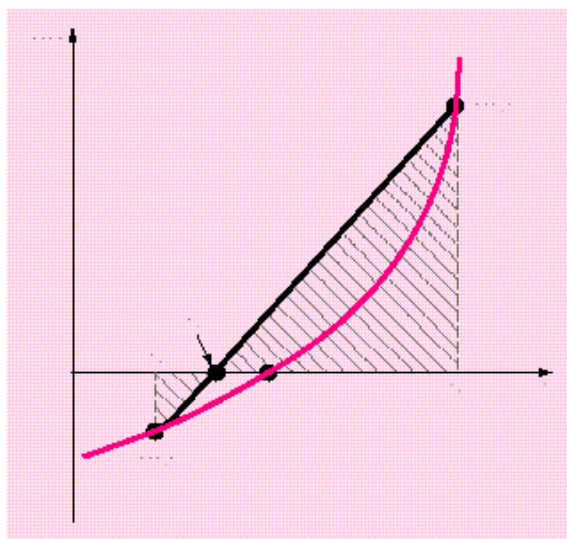
False position (also called the linear interpolation method) is another well-known bracketing method. It is very similar to bisection with the exception that it uses a different strategy to come up with its new root estimate. Rather than bisecting the interval, it locates the root by joining $f(x_l)$ and $f(x_u)$ with a straight line (Fig. 5.8). The intersection of this line with the x axis represents an improved estimate of the root. Thus, the shape of the function influences the new root estimate. Using similar triangles, the intersection of the straight line with the x axis can be estimated as (see Chapra and Canale, 2002, for details),

$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)} \quad (5.7)$$

This is the *false-position formula*. The value of x_r computed with Eq. (5.7) then replaces whichever of the two initial guesses, x_l or x_u , yields a function value with the same sign as $f(x_r)$. In this way the values of x_l and x_u always bracket the true root. The process is repeated until the root is estimated adequately. The algorithm is identical to the one for bisection (Fig. 5.7) with the exception that Eq. (5.7) is used.

FIGURE 5.8

False position.



5.5 FALSE POSITION

97

EXAMPLE 5.5 The False-Position Method

Problem Statement. Use false position to solve the same problem approached graphically and with bisection in Examples 5.1 and 5.3.

Solution. As in Example 5.3, initiate the computation with guesses of $x_l = 50$ and $x_u = 200$.

First iteration:

$$x_l = 50 \quad f(x_l) = -4.579387$$

$$x_u = 200 \quad f(x_u) = 0.860291$$

$$x_r = 200 \cdot \frac{0.860291(50 - 200)}{-4.579387 - 0.860291} = 176.2773$$

which has a true relative error of 23.5%.

Second iteration:

$$f(x_l)f(x_r) = -2.592732$$

Therefore, the root lies in the first subinterval, and x_r becomes the upper limit for the next iteration, $x_u = 176.2773$.

$$x_l = 50 \quad f(x_l) = -4.579387$$

$$x_u = 176.2773 \quad f(x_u) = 0.566174$$

$$x_r = 176.2773 \cdot \frac{0.566174(50 - 176.2773)}{-4.579387 - 0.566174} = 162.3828$$

which has true and approximate relative errors of 13.76% and 8.56%, respectively. Additional iterations can be performed to refine the estimates of the root.

Although false position often performs better than bisection, there are other cases where it does not. As in the following example, there are certain cases where bisection yields superior results.

EXAMPLE 5.6 A Case Where Bisection Is Preferable to False Position

Problem Statement. Use bisection and false position to locate the root of

$$f(x) = x^{10} - 1$$

between $x = 0$ and 1.3.

Solution. Using bisection, the results can be summarized as

Iteration	a	b	x_r	ϵ_r (%)	ϵ_a (%)
1	0	1.3	0.65	100.0	35
2	0.65	1.3	0.975	33.3	2.5
3	0.975	1.3	1.1375	14.3	13.8
4	0.975	1.1375	1.05625	7.7	5.6
5	0.975	1.05625	1.015625	4.0	1.6

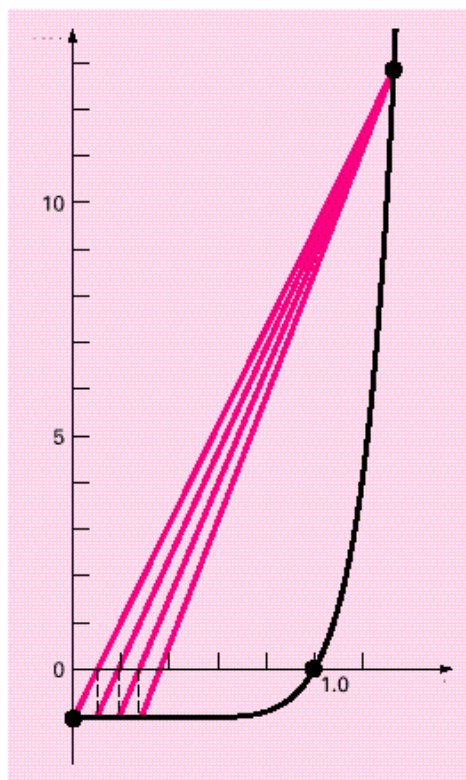
Thus, after five iterations, the true error is reduced to less than 2%. For false position, a very different outcome is obtained:

Iteration	x_L	x_U	x_r	ϵ (%)	ϵ (%)
1	0	1.3	0.09430		90.6
2	0.09430	1.3	0.18176	48.1	81.8
3	0.18176	1.3	0.26287	30.9	73.7
4	0.26287	1.3	0.33811	22.3	66.2
5	0.33811	1.3	0.40788	17.1	59.2

After five iterations, the true error has only been reduced to about 59%. Insight into these results can be gained by examining a plot of the function. As in Fig. 5.9, the curve violates the premise on which false position was based—that is, if $f(x_L)$ is much closer to zero than $f(x_U)$, then the root is closer to x_L than to x_U (recall Fig. 5.8). Because of the shape of the present function, the opposite is true.

FIGURE 5.9

Plot of $f(x) = x^{10} - 1$, illustrating slow convergence of the false-position method.



PROBLEMS

99

The forgoing example illustrates that blanket generalizations regarding root-location methods are usually not possible. Although a method such as false position is often superior to bisection, there are invariably cases that violate this general conclusion. Therefore, in addition to using Eq. (5.5), the results should always be checked by substituting the root estimate into the original equation and determining whether the result is close to zero.

The example also illustrates a major weakness of the false-position method: its one-sidedness. That is, as iterations are proceeding, one of the bracketing points will tend to stay fixed. This can lead to poor convergence, particularly for functions with significant curvature. Possible remedies for this shortcoming are available elsewhere (Chapra and Canale, 2002).

PROBLEMS

5.1 Use bisection to determine the drag coefficient needed so that an 80-kg bungee jumper has a velocity of 36 m/s after 4 s of free fall. Note: The acceleration of gravity is 9.81 m/s^2 . Start with initial guesses of $x_l = 0.1$ and $x_u = 0.2$ and iterate until the approximate relative error falls below 2%.

5.2 Develop your own M-file for bisection in a similar fashion to Fig. 5.7. However, rather than using the maximum iterations and Eq. (5.5), employ Eq. (5.6) as your stopping criterion. Make sure to round the result of Eq. (5.6) up to the next highest integer. Test your function by solving Prob. 5.1 using $E_{a,d} = 0.0001$.

5.3 Repeat Prob. 5.1, but use the false-position method to obtain your solution.

5.4 Develop an M-file for the false-position method. Test it by solving Prob. 5.1.

5.5 A beam is loaded as shown in Fig. P5.5. Use the bisection method to solve for the position inside the beam where there is no moment.

5.6 (a) Determine the roots of $f(x) = 12 + 21x + 18x^2 + 2.75x^3$ graphically. In addition, determine the first root of the function with **(b)** bisection and **(c)** false position.

For **(b)** and **(c)** use initial guesses of $x_l = -1$ and $x_u = 0$ and a stopping criterion of 1%.

5.7 Locate the first nontrivial root of $\sin(x) = x^2$ where x is in radians. Use a graphical technique and bisection with the initial interval from 0.5 to 1. Perform the computation until ε_a is less than $\varepsilon_s = 2\%$.

5.8 Determine the positive real root of $\ln(x^2) = 0.7$ **(a)** graphically, **(b)** using three iterations of the bisection method, with initial guesses of $x_l = 0.5$ and $x_u = 2$, and **(c)** using three iterations of the false-position method, with the same initial guesses as in **(b)**.

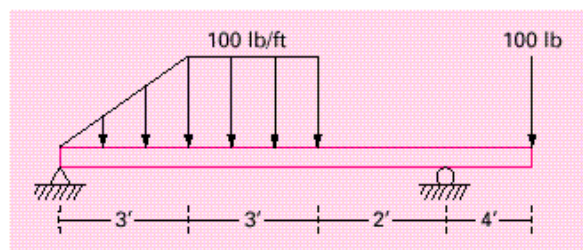
5.9 The saturation concentration of dissolved oxygen in freshwater can be calculated with the equation

$$\ln o_{sf} = 139.34411 - \frac{1.575701 \cdot 10^5}{T_a} + \frac{6.642308 \cdot 10^7}{T_a^2} - \frac{1.243800 \cdot 10^{10}}{T_a^3} + \frac{8.621949 \cdot 10^{11}}{T_a^4}$$

where o_{sf} = the saturation concentration of dissolved oxygen in freshwater at 1 atm (mg L^{-1}); and T_a = absolute temperature (K). Remember that $T_a = T + 273.15$, where T = temperature ($^{\circ}\text{C}$). According to this equation, saturation decreases with increasing temperature. For typical natural waters in temperate climates, the equation can be used to determine that oxygen concentration ranges from 14.621 mg/L at 0°C to 6.949 mg/L at 35°C . Given a value of oxygen concentration, this formula and the bisection method can be used to solve for temperature in $^{\circ}\text{C}$.

(a) If the initial guesses are set as 0 and 35°C , how many bisection iterations would be required to determine temperature to an absolute error of 0.05°C ?

FIGURE P5.5



- (b) Based on (a), develop and test a bisection M-file function to determine T as a function of a given oxygen concentration. Test your function for $\alpha_{sf} = 8, 10$ and 14 mg/L. Check your results.

5.10 Water is flowing in a trapezoidal channel at a rate of $Q = 20 \text{ m}^3/\text{s}$. The critical depth y for such a channel must satisfy the equation

$$0 = 1 - \frac{Q^2}{gA_c^3}B$$

where $g = 9.81 \text{ m/s}^2$, $A_c =$ the cross-sectional area (m^2), and $B =$ the width of the channel at the surface (m). For this

case, the width and the cross-sectional area can be related to depth y by

$$B = 3 + y$$

and

$$A_c = 3y + \frac{y^2}{2}$$

Solve for the critical depth using (a) the graphical method, (b) bisection, and (c) false position. For (b) and (c) use initial guesses of $x_l = 0.5$ and $x_u = 2.5$, and iterate until the approximate error falls below 1% or the number of iterations exceeds 10. Discuss your results.

6

Roots of Equations: Open Methods

CHAPTER OBJECTIVES

The primary objective of this chapter is to acquaint you with open methods for finding the root of a single nonlinear equation. Specific objectives and topics covered are

- Recognizing the difference between bracketing and open methods for root location.
- Understanding the fixed-point iteration method and how you can evaluate its convergence characteristics.
- Knowing how to solve a roots problem with the Newton-Raphson method and appreciating the concept of quadratic convergence.
- Knowing how to implement both the secant and the modified secant methods.
- Knowing how to use MATLAB's `fzero` function to estimate roots.
- Learning how to manipulate and determine the roots of polynomials with MATLAB.

For the bracketing methods in Chap. 5, the root is located within an interval prescribed by a lower and an upper bound. Repeated application of these methods always results in closer estimates of the true value of the root. Such methods are said to be *convergent* because they move closer to the truth as the computation progresses (Fig. 6.1a).

In contrast, the *open methods* described in this chapter require only a single starting value or two starting values that do not necessarily bracket the root. As such, they sometimes *diverge* or move away from the true root as the computation progresses (Fig. 6.1b). However, when the open methods converge (Fig. 6.1c) they usually do so much more quickly than the bracketing methods. We will begin our discussion of open techniques with a simple approach that is useful for illustrating their general form and also for demonstrating the concept of convergence.

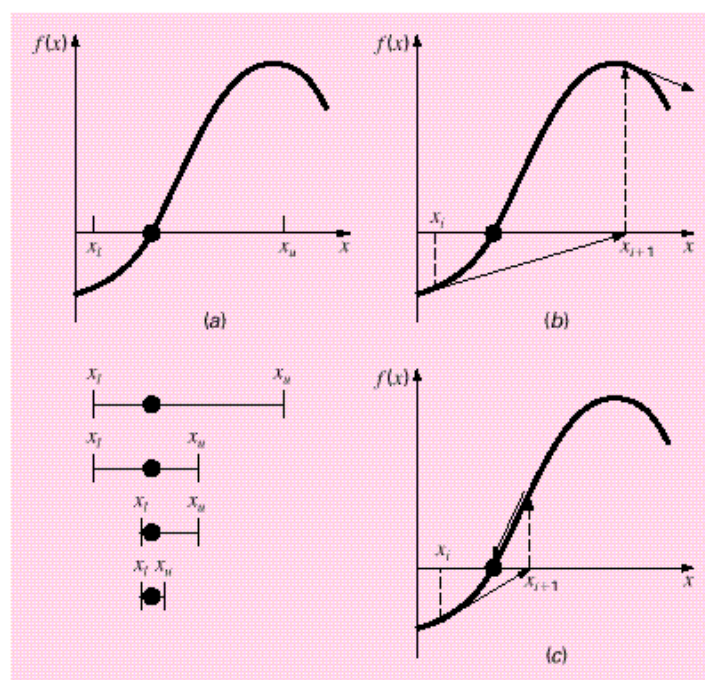


FIGURE 6.1

Graphical depiction of the fundamental difference between the $\{ \cdot \}$ bracketing and $[\cdot]$ and (\cdot) open methods for root location. In $\{ \cdot \}$, which is bisection, the root is constrained within the interval prescribed by x_l and x_u . In contrast, for the open method depicted in $[\cdot]$ and (\cdot) , which is Newton-Raphson, a formula is used to project from x_l to x_{l+1} in an iterative fashion. Thus the method can either $\{ \cdot \}$ diverge or $[\cdot]$ converge rapidly, depending on the shape of the function and the value of the initial guess.

6.1 SIMPLE FIXED-POINT ITERATION

As just mentioned, open methods employ a formula to predict the root. Such a formula can be developed for simple *fixed-point iteration* (or, as it is also called, one-point iteration or successive substitution) by rearranging the function $f(x) = 0$ so that x is on the left-hand side of the equation:

$$x = g(x) \quad (6.1)$$

This transformation can be accomplished either by algebraic manipulation or by simply adding x to both sides of the original equation.

The utility of Eq. (6.1) is that it provides a formula to predict a new value of x as a function of an old value of x . Thus, given an initial guess at the root x_l , Eq. (6.1) can be used to compute a new estimate x_{l+1} as expressed by the iterative formula

$$x_{l+1} = g(x_l) \quad (6.2)$$

6.1 SIMPLE FIXED-POINT ITERATION

103

As with many other iterative formulas in this book, the approximate error for this equation can be determined using the error estimator:

$$\varepsilon_a = \left| \frac{x_{i+1} - x_i}{x_{i+1}} \right| 100\% \quad (6.3)$$

EXAMPLE 6.1 Simple Fixed-Point Iteration

Problem Statement. Use simple fixed-point iteration to locate the root of $f(x) = e^{-x} - x$.

Solution. The function can be separated directly and expressed in the form of Eq. (6.2) as

$$x_{i+1} = e^{-x_i}$$

Starting with an initial guess of $x_0 = 0$, this iterative equation can be applied to compute:

i	x_i	error, %	error, %	error, %
0	0.0000		100.000	
1	1.0000	100.000	76.322	0.763
2	0.3679	171.828	35.135	0.460
3	0.6922	46.854	22.050	0.628
4	0.5005	38.309	11.755	0.533
5	0.6062	17.447	6.894	0.586
6	0.5454	11.157	3.835	0.556
7	0.5796	5.903	2.199	0.573
8	0.5601	3.481	1.239	0.564
9	0.5711	1.931	0.705	0.569
10	0.5649	1.109	0.399	0.566

Thus, each iteration brings the estimate closer to the true value of the root: 0.56714329.

Notice that the true percent relative error for each iteration of Example 6.1 is roughly proportional (by a factor of about 0.5 to 0.6) to the error from the previous iteration. This property, called *linear convergence*, is characteristic of fixed-point iteration.

Aside from the “rate” of convergence, we must comment at this point about the “possibility” of convergence. The concepts of convergence and divergence can be depicted graphically. Recall that in Section 5.2, we graphed a function to visualize its structure and behavior. Such an approach is employed in Fig. 6.2a for the function $f(x) = e^{-x} - x$. An alternative graphical approach is to separate the equation into two component parts, as in

$$f_1(x) = f_2(x)$$

Then the two equations

$$y_1 = f_1(x) \quad (6.4)$$

and

$$y_2 = f_2(x) \quad (6.5)$$

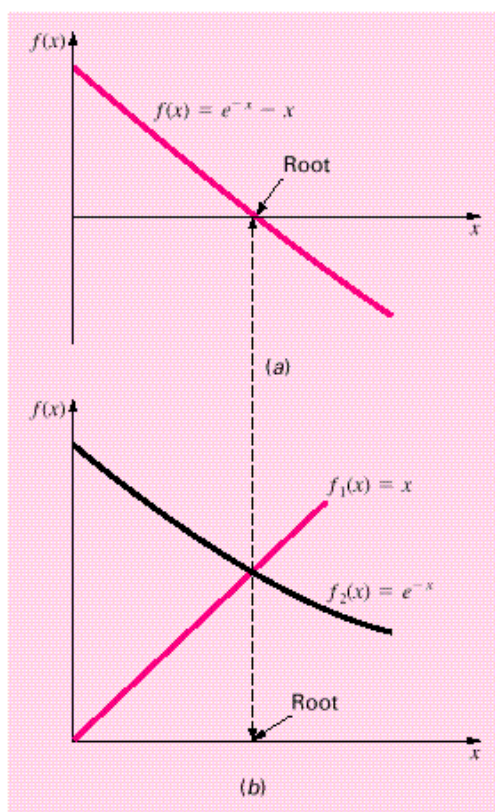


FIGURE 6.2

Two alternative graphical methods for determining the root of $f(x) = e^{-x} - x$. (·) Root at the point where it crosses the x axis; (·) root at the intersection of the component functions.

can be plotted separately (Fig. 6.2b). The x values corresponding to the intersections of these functions represent the roots of $f(x) = 0$.

The two-curve method can now be used to illustrate the convergence and divergence of fixed-point iteration. First, Eq. (6.1) can be reexpressed as a pair of equations $y_1 = x$ and $y_2 = g(x)$. These two equations can then be plotted separately. As was the case with Eqs. (6.4) and (6.5), the roots of $f(x) = 0$ correspond to the abscissa value at the intersection of the two curves. The function $y_1 = x$ and four different shapes for $y_2 = g(x)$ are plotted in Fig. 6.3.

For the first case (Fig. 6.3a), the initial guess of x_0 is used to determine the corresponding point on the y_2 curve $[x_0, g(x_0)]$. The point $[x_1, x_1]$ is located by moving left horizontally to the y_1 curve. These movements are equivalent to the first iteration of the fixed-point method:

$$x_1 = g(x_0)$$

Thus, in both the equation and in the plot, a starting value of x_0 is used to obtain an estimate of x_1 . The next iteration consists of moving to $[x_1, g(x_1)]$ and then to $[x_2, x_2]$. This

6.1 SIMPLE FIXED-POINT ITERATION

105

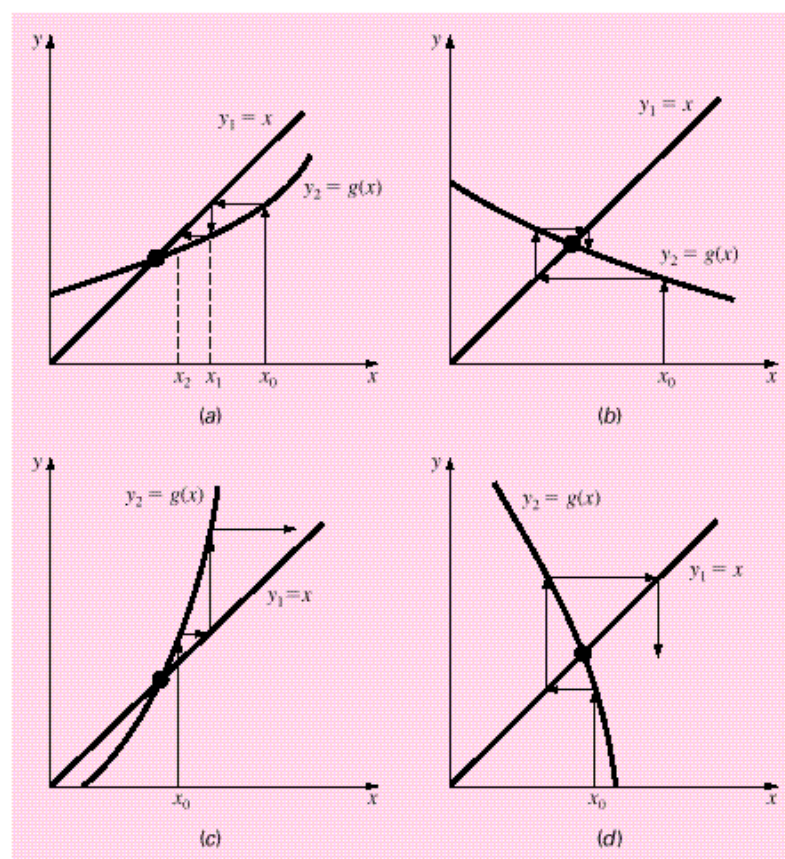


FIGURE 6.3

Graphical depiction of $\{ \cdot \}$ and $\{ \cdot \}$ convergence and $\{ \cdot \}$ and $\{ \cdot \}$ divergence of simple fixed-point iteration. Graphs $\{ \cdot \}$ and $\{ \cdot \}$ are called monotone patterns whereas $\{ \cdot \}$ and $\{ \cdot \}$ are called oscillating or spiral patterns. Note that convergence occurs when $|g'(x)| < 1$.

iteration is equivalent to the equation

$$x_2 = g(x_1)$$

The solution in Fig. 6.3a is *convergent* because the estimates of x move closer to the root with each iteration. The same is true for Fig. 6.3b. However, this is not the case for Fig. 6.3c and d, where the iterations diverge from the root.

A theoretical derivation can be used to gain insight into the process. As described in Chapra and Canale (2002), it can be shown that the error for any iteration is linearly proportional to the error from the previous iteration multiplied by the absolute value of the slope of g :

$$E_{i+1} = g'(\xi)E_i$$

Consequently, if $|g'| < 1$, the errors decrease with each iteration. For $|g'| > 1$ the errors grow. Notice also that if the derivative is positive, the errors will be positive, and hence the errors will have the same sign (Fig. 6.3a and c). If the derivative is negative, the errors will change sign on each iteration (Fig. 6.3b and d).

6.2 NEWTON-RAPHSON

Perhaps the most widely used of all root-locating formulas is the *Newton-Raphson method* (Fig. 6.4). If the initial guess at the root is x_i , a tangent can be extended from the point $[x_i, f(x_i)]$. The point where this tangent crosses the x axis usually represents an improved estimate of the root.

The Newton-Raphson method can be derived on the basis of this geometrical interpretation. As in Fig. 6.4, the first derivative at x is equivalent to the slope:

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

which can be rearranged to yield

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (6.6)$$

which is called the *Newton-Raphson formula*.

EXAMPLE 6.2 Newton-Raphson Method

Problem Statement. Use the Newton-Raphson method to estimate the root of $f(x) = e^{-x} - x$ employing an initial guess of $x_0 = 0$.

Solution. The first derivative of the function can be evaluated as

$$f'(x) = -e^{-x} - 1$$

which can be substituted along with the original function into Eq. (6.6) to give

$$x_{i+1} = x_i - \frac{e^{-x_i} - x_i}{-e^{-x_i} - 1}$$

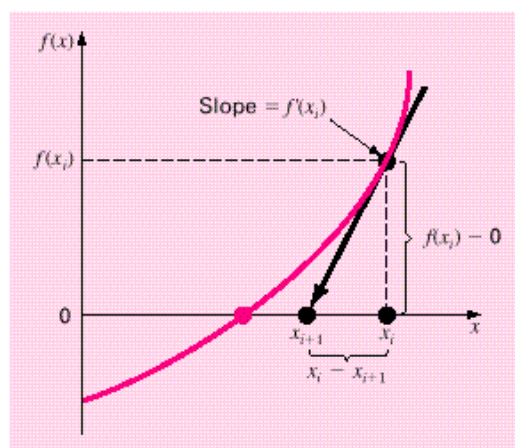
Starting with an initial guess of $x_0 = 0$, this iterative equation can be applied to compute

		Error, %
0	0	100
1	0.500000000	11.8
2	0.566311003	0.147
3	0.567143165	0.0000220
4	0.567143290	$< 10^{-8}$

Thus, the approach rapidly converges on the true root. Notice that the true percent relative error at each iteration decreases much faster than it does in simple fixed-point iteration (compare with Example 6.1).

6.2 NEWTON-RAPHSON

107

**FIGURE 6.4**

Graphical depiction of the Newton-Raphson method. A tangent to the function of x_i [that is, $f'(x)$] is extrapolated down to the x axis to provide an estimate of the root at x_{i+1} .

As with other root-location methods, Eq. (6.3) can be used as a termination criterion. In addition, a theoretical analysis (Chapra and Canale, 2002) provides insight regarding the rate of convergence as expressed by

$$E_{i,j+1} = \frac{-f''(x_r)}{2f'(x_r)} E_{i,j}^2 \quad (6.7)$$

Thus, the error should be roughly proportional to the square of the previous error. In other words, the number of significant figures of accuracy approximately doubles with each iteration. This behavior is called *quadratic convergence* and is one of the major reasons for the popularity of the method.

Although the Newton-Raphson method is often very efficient, there are situations where it performs poorly. A special case—multiple roots—is discussed elsewhere (Chapra and Canale, 2002). However, even when dealing with simple roots, difficulties can also arise, as in the following example.

EXAMPLE 6.3 A Slowly Converging Function with Newton-Raphson

Problem Statement. Determine the positive root of $f(x) = x^{10} - 1$ using the Newton-Raphson method and an initial guess of $x = 0.5$.

Solution. The Newton-Raphson formula for this case is

$$x_{i+1} = x_i - \frac{x_i^{10} - 1}{10x_i^9}$$

which can be used to compute

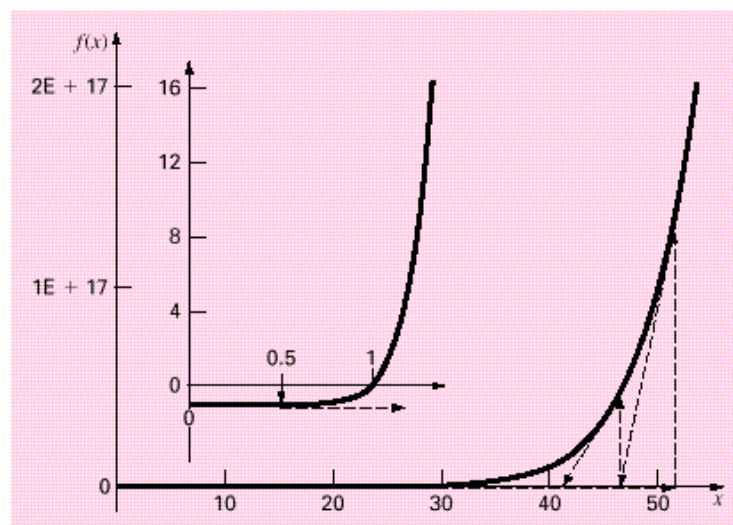
		error, %
0	0.5	
1	51.65	99.032
2	46.485	11.111
3	41.8365	11.111
4	37.65285	11.111
⋮		
40	1.002316	2.130
41	1.000024	0.229
42	1	0.002

Thus, after the first poor prediction, the technique is converging on the true root of 1, but at a very slow rate.

Why does this happen? As shown in Fig. 6.5, a simple plot of the first few iterations is helpful in providing insight. Notice how the first guess is in a region where the slope is near zero. Thus, the first iteration flings the solution far away from the initial guess to a new value ($x = 51.65$) where $f(x)$ has an extremely high value. The solution then plods along for over 40 iterations until converging on the root with adequate accuracy.

FIGURE 6.5

Graphical depiction of the Newton-Raphson method for a case with slow convergence. The inset shows how a near-zero slope initially shoots the solution far from the root. Thereafter, the solution very slowly converges on the root.



Aside from slow convergence due to the nature of the function, other difficulties can arise, as illustrated in Fig. 6.6. For example, Fig. 6.6a depicts the case where an inflection

6.2 NEWTON-RAPHSON

109

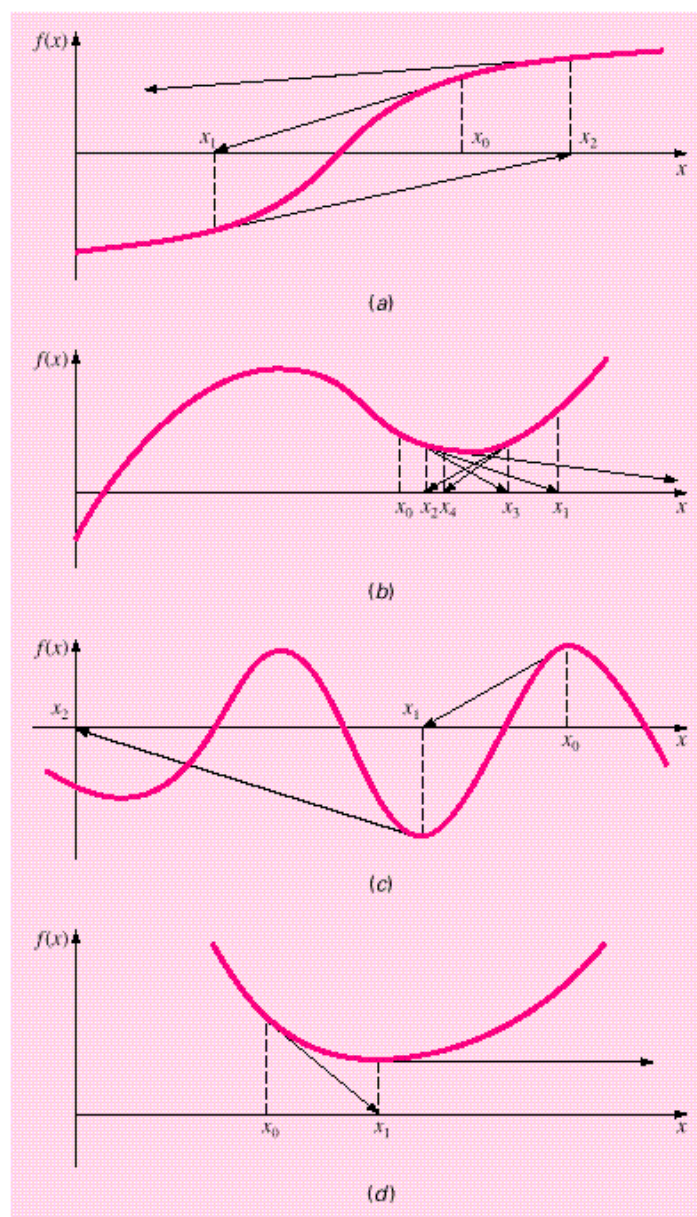


FIGURE 6.6

Four cases where the Newton-Raphson method exhibits poor convergence.

point (i.e., $f'(x) = 0$) occurs in the vicinity of a root. Notice that iterations beginning at x_0 progressively diverge from the root. Fig. 6.6*b* illustrates the tendency of the Newton-Raphson technique to oscillate around a local maximum or minimum. Such oscillations may persist, or, as in Fig. 6.6*b*, a near-zero slope is reached whereupon the solution is sent far from the area of interest. Figure 6.6*c* shows how an initial guess that is close to one root can jump to a location several roots away. This tendency to move away from the area of interest is due to the fact that near-zero slopes are encountered. Obviously, a zero slope [$f'(x) = 0$] is a real disaster because it causes division by zero in the Newton-Raphson formula [Eq. (6.6)]. As in Fig. 6.6*d*, it means that the solution shoots off horizontally and never hits the x axis.

Thus, there is no general convergence criterion for Newton-Raphson. Its convergence depends on the nature of the function and on the accuracy of the initial guess. The only remedy is to have an initial guess that is “sufficiently” close to the root. And for some functions, no guess will work! Good guesses are usually predicated on knowledge of the physical problem setting or on devices such as graphs that provide insight into the behavior of the solution. It also suggests that good computer software should be designed to recognize slow convergence or divergence.

6.2.1 MATLAB M-file:

An algorithm for the Newton-Raphson method can be easily developed (Fig. 6.7). Note that the program must have access to the function (`func`) and its first derivative (`dfunc`). These can be simply accomplished by the inclusion of user-defined functions to compute these quantities. Alternatively, as in the algorithm in Fig. 6.7, they can be passed to the function as arguments.

After the M-file is entered and saved, it can be invoked to solve for root. For example, for the simple function $x^2 - 9$, the root can be determined as in

```
>> newtraph(inline('x^2-9'),inline('2*x'),5)

ans =
     3
```

EXAMPLE 6.4 Newton-Raphson Bungee Jumper Problem

Problem Statement. Use the M-file function from Fig. 6.7 to determine the mass of the bungee jumper with a drag coefficient of 0.25 kg/m to have a velocity of 36 m/s after 4 s of free fall. The acceleration of gravity is 9.81 m/s².

Solution. The function to be evaluated is

$$f(m) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) - v(t) \quad (\text{E6.4.1})$$

To apply the Newton-Raphson method, the derivative of this function must be evaluated with respect to the unknown, m :

$$\frac{df(m)}{dm} = \frac{1}{2} \sqrt{\frac{g}{mc_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) - \frac{g}{2m} t \operatorname{sech}^2\left(\sqrt{\frac{gc_d}{m}} t\right) \quad (\text{E6.4.2})$$

We should mention that although this derivative is not difficult to evaluate in principle, it involves a bit of concentration and effort to arrive at the final result.

6.3 SECANT METHODS

111

```
function root = newtraph(func,dfunc,xr,es,maxit)
% newtraph(func,dfunc,xguess,es,maxit):
% uses Newton-Raphson method to find root of a function
% input:
% func = name of function
% dfunc = name of derivative of function
% xguess = initial guess
% es = (optional) stopping criterion (%)
% maxit = (optional) maximum allowable iterations
% output:
% root = real root

% if necessary, assign default values
if nargin<5, maxit = 50; end      %if maxit blank set to 50
if nargin<4, es = 0.001; end    %if es blank set to 0.001

% Newton-Raphson
iter = 0;
while (1)
    xrold = xr;
    xr = xr - func(xr)/dfunc(xr);
    iter = iter + 1;
    if xr ~= 0, ea = abs((xr - xrold)/xr) * 100; end
    if ea <= es | iter >= maxit, break, end
end
root = xr;
```

FIGURE 6.7

An Mfile to implement the Newton-Raphson method.

The two formulas can now be used in conjunction with the function `newtraph` to evaluate the root:

```
>> y = inline('sqrt(9.81*m/0.25)*tanh(sqrt(9.81*0.25/m)*4)-36','m');
>> dy = inline('1/2*sqrt(9.81/(m*0.25))*tanh((9.81*0.25/m)^(1/2)*4)
           -9.81/(2*m)*sech(sqrt(9.81*0.25/m)*4)^2','m');

>> newtraph(y,dy,140,0.00001)

ans =
    142.7376
```

6.3 SECANT METHODS

As in Example 6.4, a potential problem in implementing the Newton-Raphson method is the evaluation of the derivative. Although this is not inconvenient for polynomials and many other functions, there are certain functions whose derivatives may be difficult or

inconvenient to evaluate. For these cases, the derivative can be approximated by a backward finite divided difference:

$$f'(x_i) \cong \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}$$

This approximation can be substituted into Eq. (6.6) to yield the following iterative equation:

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)} \quad (6.8)$$

Equation (6.8) is the formula for the *secant method*. Notice that the approach requires two initial estimates of x . However, because $f(x)$ is not required to change signs between the estimates, it is not classified as a bracketing method.

Rather than using two arbitrary values to estimate the derivative, an alternative approach involves a fractional perturbation of the independent variable to estimate $f'(x)$,

$$f'(x_i) \cong \frac{f(x_i + \delta x_i) - f(x_i)}{\delta x_i}$$

where δ = a small perturbation fraction. This approximation can be substituted into Eq. (6.6) to yield the following iterative equation:

$$x_{i+1} = x_i - \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i)} \quad (6.9)$$

We call this the *modified secant method*. As in the following example, it provides a nice means to attain the efficiency of Newton-Raphson without having to compute derivatives.

EXAMPLE 6.5 Modified Secant Method

Problem Statement. Use the modified secant method to determine the mass of the bungee jumper with a drag coefficient of 0.25 kg/m to have a velocity of 36 m/s after 4 s of free fall. Note: The acceleration of gravity is 9.81 m/s². Use an initial guess of 50 kg and a value of 10⁻⁶ for the perturbation fraction.

Solution. Inserting the parameters into Eq. (6.9) yields

First iteration:

$$\begin{aligned} x_0 &= 50 & f(x_0) &= -4.57938708 \\ x_0 + \delta x_0 &= 50.000005 & f(x_0 + \delta x_0) &= -4.579381118 \\ x_1 &= 50 - \frac{10^{-6}(50)(-4.57938708)}{-4.579381118 - (-4.57938708)} \\ &= 88.39931 (|\varepsilon_t| = 38.1\%; |\varepsilon_a| = 43.4\%) \end{aligned}$$

6.4 MATLAB FUNCTION: `fzero`

113

Second iteration:

$$\begin{aligned}
 x_1 &= 88.39931 & f(x_1) &= -1.69220771 \\
 x_1 + \delta x_1 &= 88.39940 & f(x_1 + \delta x_1) &= -1.692203516 \\
 x_2 &= 88.39931 - \frac{10^{-6}(88.39931)(-1.69220771)}{-1.692203516 - (-1.69220771)} \\
 &= 124.08970 (|\epsilon_f| = 13.1\%; |\epsilon_d| = 28.76\%)
 \end{aligned}$$

The calculation can be continued to yield

		$\epsilon_f, \%$	$\epsilon_d, \%$
0	50.0000	64.971	
1	88.3993	38.069	43.438
2	124.0897	13.064	28.762
3	140.5417	1.538	11.706
4	142.7072	0.021	1.517
5	142.7376	4.1×10^{-6}	0.021
6	142.7376	3.4×10^{-12}	4.1×10^{-6}

The choice of a proper value for δ is not automatic. If δ is too small, the method can be swamped by round-off error caused by subtractive cancellation in the denominator of Eq. (6.9). If it is too big, the technique can become inefficient and even divergent. However, if chosen correctly, it provides a nice alternative for cases where evaluating the derivative is difficult and developing two initial guesses is inconvenient.

Further, in its most general sense, a univariate function is merely an entity that returns a single value in return for values sent to it. Perceived in this sense, functions are not always simple formulas like the one-line equations solved in the preceding examples in this chapter. For example, a function might consist of many lines of code that could take a significant amount of execution time to evaluate. In some cases, the function might even represent an independent computer program. For such cases, the secant and modified secant methods are valuable.

6.4 MATLAB FUNCTION: `fzero`

The methods we have described to this point are either reliable but slow (bracketing) or fast but possibly unreliable (open). The MATLAB `fzero` function provides the best qualities of both. The `fzero` function is designed to find the real root of a single equation. A simple representation of its syntax is

```
fzero(function, x0)
```

where *function* is the name of the function being evaluated, and *x0* is the initial guess. Note that two guesses that bracket the root can be passed as a vector:

```
fzero(function, [x0 x1])
```

where *x0* and *x1* are guesses that bracket a sign change.

Here is a simple MATLAB session that solves for the root of a simple quadratic: $x^2 - 9$. Clearly two roots exist at -3 and 3 . To find the negative root:

```
>> x = fzero(inline('x^2-9'),-4)

x =
   -3
```

If we want to find the positive root, use a guess that is near it:

```
>> x = fzero(inline('x^2-9'),4)

x =
    3
```

If we put in an initial guess of zero, it finds the negative root

```
>> x = fzero(inline('x^2-9'),0)

x =
   -3
```

If we wanted to ensure that we found the positive root, we could enter two guesses as in

```
>> x = fzero(inline('x^2-9'),[0 4])

x =
    3
```

Also, if a sign change does not occur between the two guesses, an error message is displayed

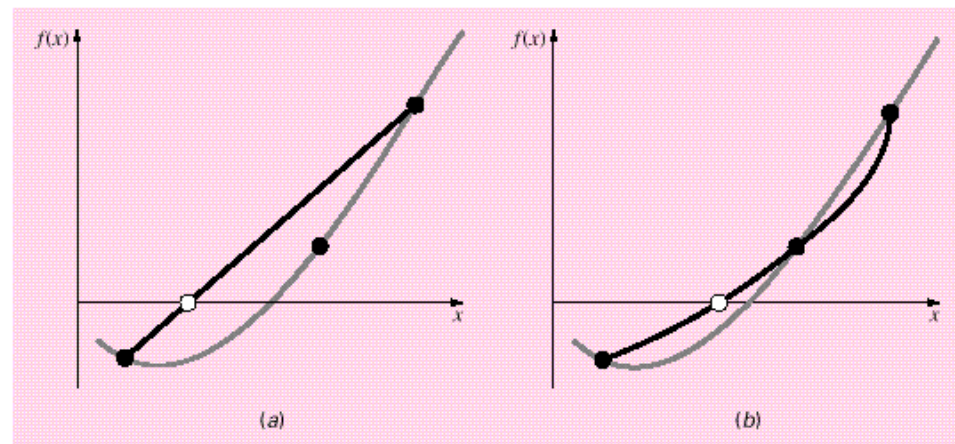
```
>> x = fzero(inline('x^2-9'),[-4 4])

??? Error using ==> fzero
The function values at the interval endpoints must differ in sign.
```

The `fzero` function is a combination of the reliable bisection method with two faster algorithms: the secant method and inverse quadratic interpolation. *Inverse quadratic interpolation* is similar in spirit to the secant method. As in Fig. 6.8a, the secant method is based on computing a straight line that goes through two guesses. The intersection of this straight line with the x axis represents the new root estimate. The inverse quadratic interpolation uses a similar strategy but is based on computing a quadratic equation (i.e., a parabola) that goes through three points (Fig. 6.8b).

The `fzero` function works as follows. If a single initial guess is passed, it first performs a search to identify a sign change. This search differs from the incremental search described in Section 5.3.1, in that the search starts at the single initial guess and then takes increasingly bigger steps in both the positive and negative directions until a sign change is detected.

Thereafter, the fast methods (secant and inverse quadratic interpolation) are used unless an unacceptable result occurs (e.g., the root estimate falls outside the bracket). If a bad result happens, bisection is implemented until an acceptable root is obtained with one of the fast methods. As might be expected, bisection typically dominates at first but as the root is approached, the technique shifts to the faster methods.

**FIGURE 6.8**

Comparison of (·) the secant method and (·) inverse quadratic interpolation. Note that the approach in (·) is called "inverse" because the quadratic function is written in y rather than in x .

A more complete representation of the `fzero` syntax can be written as

```
[x, fx] = fzero(function, x0, options, p1, p2, ...)
```

where $[x, fx]$ is a vector containing the root x and the function evaluated at the root fx , `options` is a data structure created by the `optimset` function, and $p1, p2, \dots$ are any parameters that the function requires. Note that if you desire to pass in parameters but not use the `options`, pass an empty vector `[]` in its place.

The `optimset` function has the syntax

```
options = optimset('par1', val1, 'par2', val2, ...)
```

where the parameter par_j has the value val_j . A complete listing of all the possible parameters can be obtained by merely entering `optimset` at the command prompt. The parameters that are commonly used with the `fzero` function are

`display`: When set to 'iter' displays a detailed record of all the iterations.
`tolx`: A positive scalar that sets a termination tolerance on x .

EXAMPLE 6.6 The `fzero` and `optimset` Functions

Problem Statement. Recall that in Example 6.3, we found the positive root of $f(x) = x^{10} - 1$ using the Newton-Raphson method with an initial guess of 0.5. Solve the same problem with `optimset` and `fzero`.

Solution. An interactive MATLAB session can be implemented as follows:

```
>> options = optimset('display','iter');
>> [x,fx] = fzero(inline('x^10-1'),0.5,options)
```

```
Func-count      x          f(x)        Procedure
1              0.5        -0.999023    initial
2             0.485858    -0.999267    search
3             0.514142    -0.998709    search
4              0.48        -0.999351    search
5              0.52        -0.998554    search
6             0.471716    -0.999454    search
.
.
.
23             0.952548    -0.385007    search
24             -0.14        -1          search
25              1.14        2.70722     search

Looking for a zero in the interval [-0.14, 1.14]

26             0.205272        -1          interpolation
27             0.672636    -0.981042    bisection
28             0.906318    -0.626056    bisection
29             1.02316      0.257278    bisection
30             0.989128    -0.103551    interpolation
31             0.998894    -0.0110017   interpolation
32             1.00001      7.68385e-005 interpolation
33              1          -3.83061e-007 interpolation
34              1          -1.3245e-011 interpolation
35              1              0          interpolation

Zero found in the interval: [-0.14, 1.14].

x =
    1

fx =
    0
```

Thus, after 25 iterations of searching, `fzero` finds a sign change. It then uses interpolation and bisection until it gets close enough to the root so that interpolation takes over and rapidly converges on the root.

Suppose that we would like to use a less stringent tolerance. We can use the `optimset` function to set a low maximum tolerance and a less accurate estimate of the root results:

```
>> options = optimset ('tolx', 1e-3);
>> [x,fx] = fzero(inline('x^10-1'),0.5,options)

x =
    1.0009

fx =
    0.0090
```


6.5 POLYNOMIALS

Polynomials are a special type of nonlinear algebraic equation of the general form

$$f_n(x) = a_1x^n + a_2x^{n-1} + \cdots + a_{n-1}x^2 + a_nx + a_{n+1} \quad (6.10)$$

where n is the order of the polynomial, and the a 's are constant coefficients. In many (but not all) cases, the coefficients will be real. For such cases, the roots can be real and/or complex. In general, an n th order polynomial will have n roots.

Polynomials have many applications in engineering and science. For example, they are used extensively in curve fitting. However, one of their most interesting and powerful applications is in characterizing dynamic systems—and, in particular, linear systems. Examples include reactors, mechanical devices, structures, and electrical circuits.

6.5.1 MATLAB Function:

If you are dealing with a problem where you must determine a single real root of a polynomial, the techniques such as bisection and the Newton-Raphson method can have utility. However, in many cases, engineers desire to determine all the roots, both real and complex. Unfortunately, simple techniques like bisection and Newton-Raphson are not available for determining all the roots of higher-order polynomials. However, MATLAB has an excellent built-in capability, the `roots` function, for this task.

The `roots` function has the syntax,

$$x = \text{roots}(c)$$

where x is a column vector containing the roots and c is a row vector containing the polynomial's coefficients.

So how does the `roots` function work? MATLAB is very good at finding the eigenvalues of a matrix. Consequently, the approach is to recast the root evaluation task as an eigenvalue problem. Because we will be describing eigenvalue problems later in the book, we will merely provide an overview here.

Suppose we have a polynomial

$$a_1x^5 + a_2x^4 + a_3x^3 + a_4x^2 + a_5x + a_6 = 0 \quad (6.11)$$

Dividing by a_1 and rearranging yields

$$x^5 = -\frac{a_2}{a_1}x^4 - \frac{a_3}{a_1}x^3 - \frac{a_4}{a_1}x^2 - \frac{a_5}{a_1}x - \frac{a_6}{a_1} \quad (6.12)$$

A special matrix can be constructed by using the coefficients from the right-hand side as the first row and with 1's and 0's written for the other rows as shown:

$$\begin{bmatrix} -a_2/a_1 & -a_3/a_1 & -a_4/a_1 & -a_5/a_1 & -a_6/a_1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (6.13)$$

Equation (6.13) is called the polynomial's *companion matrix*. It has the useful property that its eigenvalues are the roots of the polynomial. Thus, the algorithm underlying

the `roots` function consists of merely setting up the companion matrix and then using MATLAB's powerful eigenvalue evaluation function to determine the roots. Its application, along with some other related polynomial manipulation functions, are described in the following example.

We should note that `roots` has an inverse function called `poly`, which when passed the values of the roots, will return the polynomial's coefficients. Its syntax is

```
c = poly(r)
```

where r is a column vector containing the roots and c is a row vector containing the polynomial's coefficients.

EXAMPLE 6.7 Using MATLAB to Manipulate Polynomials and Determine Their Roots

Problem Statement. Use the following equation to explore how MATLAB can be employed to manipulate polynomials:

$$f_5(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25 \quad (\text{E6.7.1})$$

Note that this polynomial has three real roots: 0.5, -1.0 , and 2; and one pair of complex roots: $-1 \pm 0.5i$.

Solution. Polynomials are entered into MATLAB by storing the coefficients as a row vector. For example, entering the following line stores the coefficients in the vector a :

```
>> a = [1 -3.5 2.75 2.125 -3.875 1.25];
```

We can then proceed to manipulate the polynomial. For example we can evaluate it at $x = 1$, by typing

```
>> polyval(a,1)
```

with the result, $1(1)^5 - 3.5(1)^4 + 2.75(1)^3 + 2.125(1)^2 - 3.875(1) + 1.25 = -0.25$:

```
ans =  
-0.2500
```

We can create a quadratic polynomial that has roots corresponding to two of the original roots of Eq. (E6.7.1): 0.5 and -1 . This quadratic is $(x - 0.5)(x + 1) = x^2 + 0.5x - 0.5$. It can be entered into MATLAB as the vector b :

```
>> b = [1 .5 -.5]  
b =  
1.0000 0.5000 -0.5000
```

Note that the `poly` function can be used to perform the same task as in

```
>> b = poly([0.5 -1])  
b =  
1.0000 0.5000 -0.5000
```

We can divide this polynomial into the original polynomial by

```
>> [q,r] = deconv(a,b)
```

PROBLEMS

119

with the result being a quotient (a third-order polynomial, q) and a remainder (r)

```
q =  
    1.0000    -4.0000     5.2500    -2.5000  
r =  
     0     0     0     0     0     0
```

Because the polynomial is a perfect divisor, the remainder polynomial has zero coefficients. Now, the roots of the quotient polynomial can be determined as

```
>> x = roots(q)
```

with the expected result that the remaining roots of the original polynomial Eq. (E6.7.1) are found:

```
x =  
    2.0000  
    1.0000 + 0.5000i  
    1.0000 - 0.5000i
```

We can now multiply q by b to come up with the original polynomial:

```
>> a = conv(q,b)  
a =  
    1.0000   -3.5000     2.7500     2.1250   -3.8750     1.2500
```

We can then determine all the roots of the original polynomial by

```
>> x = roots(a)  
x =  
    2.0000  
   -1.0000  
    1.0000 + 0.5000i  
    1.0000 - 0.5000i  
    0.5000
```

Finally, we can return to the original polynomial again by using the `poly` function:

```
>> a = poly(x)  
a =  
    1.0000   -3.5000     2.7500     2.1250   -3.8750     1.2500
```

PROBLEMS

6.1 Employ fixed-point iteration to locate the root of

$$f(x) = \sin(\sqrt{x}) - x$$

Use an initial guess of $x_0 = 0.5$ and iterate until $\varepsilon_a \leq 0.01\%$.

6.2 Use (a) fixed-point iteration and (b) the Newton-Raphson method to determine a root of $f(x) = -0.9x^2 + 1.7x + 2.5$ using $x_0 = 5$. Perform the computation until ε_a is less than $\varepsilon_s = 0.01\%$. Also check your final answer.

6.3 Determine the highest real root of $f(x) = x^3 - 6x^2 + 11x - 6.1$:

- (a) Graphically.
- (b) Using the Newton-Raphson method (three iterations, $x_i = 3.5$).
- (c) Using the secant method (three iterations, $x_{i-1} = 2.5$ and $x_i = 3.5$).
- (d) Using the modified secant method (three iterations, $x_i = 3.5$, $\delta = 0.02$).
- (e) Determine all the roots with MATLAB.

6.4 Determine the lowest positive root of $f(x) = 7 \sin(x)e^{-x} - 1$:

- (a) Graphically.
- (b) Using the Newton-Raphson method (three iterations, $x_i = 0.3$).
- (c) Using the secant method (three iterations, $x_{i-1} = 0.4$ and $x_i = 0.3$).
- (d) Using the modified secant method (five iterations, $x_i = 0.3$, $\delta = 0.01$).

6.5 Use (a) the Newton-Raphson method and (b) the modified secant method ($\delta = 0.05$) to determine a root of $f(x) = x^5 - 16.05x^4 + 88.75x^3 - 192.0375x^2 + 116.35x + 31.6875$ using an initial guess of $x = 0.5825$ and $\varepsilon_s = 0.01\%$. Explain your results.

6.6 Develop an M-file for the secant method. Along with the two initial guesses, pass the function as an argument. Test it by solving Prob. 6.3.

6.7 Develop an M-file for the modified secant method. Along with the initial guess and the perturbation fraction, pass the function as an argument. Test it by solving Prob. 6.3.

6.8 Differentiate Eq. (E6.4.1) to get Eq. (E6.4.2).

6.9 Employ the Newton-Raphson method to determine a real root for $f(x) = -2 + 6x - 4x^2 + 0.5x^3$, using an initial guess of (a) 4.5, and (b) 4.43. Discuss and use graphical and analytical methods to explain any peculiarities in your results.

6.10 The “divide and average” method, an old-time method for approximating the square root of any positive number a , can be formulated as

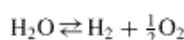
$$x_{i+1} = \frac{x_i + a/x_i}{2}$$

Prove that this formula is based on the Newton-Raphson algorithm.

6.11 (a) Apply the Newton-Raphson method to the function $f(x) = \tanh(x^2 - 9)$ to evaluate its known real root at $x = 3$. Use an initial guess of $x_0 = 3.2$ and take a minimum of three iterations. (b) Did the method exhibit convergence onto its real root? Sketch the plot with the results for each iteration labeled.

6.12 The polynomial $f(x) = 0.0074x^4 - 0.284x^3 + 3.355x^2 - 12.183x + 5$ has a real root between 15 and 20. Apply the Newton-Raphson method to this function using an initial guess of $x_0 = 16.15$. Explain your results.

6.13 In a chemical engineering process, water vapor (H_2O) is heated to sufficiently high temperatures that a significant portion of the water dissociates, or splits apart, to form oxygen (O_2) and hydrogen (H_2):



If it is assumed that this is the only reaction involved, the mole fraction x of H_2O that dissociates can be represented by

$$K = \frac{x}{1-x} \sqrt{\frac{2p_i}{2+x}} \quad (P6.13.1)$$

where K is the reaction's equilibrium constant and p_i is the total pressure of the mixture. If $p_i = 3$ atm and $K = 0.05$, determine the value of x that satisfies Eq. (P6.13.1).

6.14 The Redlich-Kwong equation of state is given by

$$p = \frac{RT}{v-b} - \frac{a}{v(v+b)\sqrt{T}}$$

where R = the universal gas constant [$=0.518$ kJ/(kg K)], T = absolute temperature (K), p = absolute pressure (kPa), and v = the volume of a kg of gas (m^3/kg). The parameters a and b are calculated by

$$a = 0.427 \frac{R^2 T_c^{2.5}}{p_c} \quad b = 0.0866 R \frac{T_c}{p_c}$$

where $p_c = 4600$ kPa and $T_c = 191$ K. As a chemical engineer, you are asked to determine the amount of methane fuel that can be held in a 3-m^3 tank at a temperature of -40°C with a pressure of $65,000$ kPa. Use a root locating method of your choice to calculate v and then determine the mass of methane contained in the tank.

6.15 The volume of liquid V in a hollow horizontal cylinder of radius r and length L is related to the depth of the liquid h by

$$V = \left[r^2 \cos^{-1} \left(\frac{r-h}{r} \right) - (r-h) \sqrt{2rh-h^2} \right] L$$

Determine h given $r = 2$ m, $L = 5$ m³, and $V = 8$ m³.

6.16 A catenary cable is one which is hung between two points not in the same vertical line. As depicted in Fig. P6.16a, it is subject to no loads other than its own weight. Thus, its weight acts as a uniform load per unit length along the cable w (N/m). A free-body diagram of a section AB is depicted in Fig. P6.16b, where T_A and T_B are

PROBLEMS

121

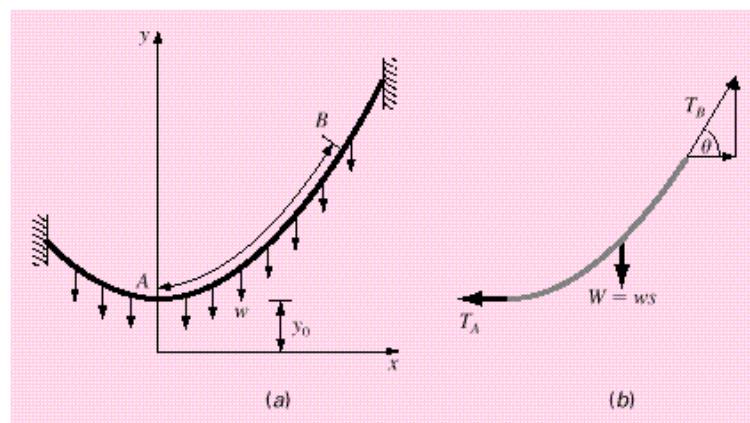


FIGURE P6.16

the tension forces at the end. Based on horizontal and vertical force balances, the following differential equation model of the cable can be derived:

$$\frac{d^2y}{dx^2} = \frac{w}{T_A} \sqrt{1 + \left(\frac{dy}{dx}\right)^2}$$

Calculus can be employed to solve this equation for the height of the cable y as a function of distance x :

$$y = \frac{T_A}{w} \cosh\left(\frac{w}{T_A}x\right) + y_0 - \frac{T_A}{w}$$

- (a) Use a numerical method to calculate a value for the parameter T_A given values for the parameters $w = 10$ and $y_0 = 5$, such that the cable has a height of $y = 15$ at $x = 50$.

- (b) Develop a plot of y versus x for $x = -50$ to 100.

6.17 An oscillating current in an electric circuit is described by $I = 9e^{-t} \sin(2\pi t)$, where t is in seconds. Determine all values of t such that $I = 3.5$.

6.18 Figure P6.18 shows a circuit with a resistor, an inductor, and a capacitor in parallel. Kirchhoff's rules can be used to express the impedance of the system as

$$\frac{1}{Z} = \sqrt{\frac{1}{R^2} + \left(\omega C - \frac{1}{\omega L}\right)^2}$$

where Z = impedance (Ω), and ω is the angular frequency. Find the ω that results in an impedance of 100Ω using the `fzero` function with initial guesses of 1 and 1000 for the following parameters: $R = 225 \Omega$, $C = 0.6 \times 10^{-6} \text{ F}$, and $L = 0.5 \text{ H}$.

6.19 Real mechanical systems may involve the deflection of nonlinear springs. In Fig. P6.19, a block of mass m is released a distance h above a nonlinear spring. The resistance force F of the spring is given by

$$F = -(k_1 d + k_2 d^{3/2})$$

FIGURE P6.18

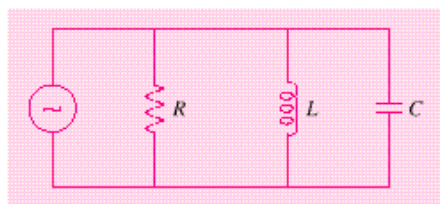
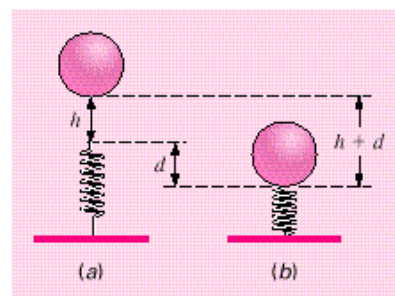


FIGURE P6.19



Conservation of energy can be used to show that

$$0 = \frac{2k_2 d^{5/2}}{5} + \frac{1}{2} k_1 d^2 - mgd - mgh$$

Solve for d , given the following parameter values: $k_1 = 40,000 \text{ g/s}^2$, $k_2 = 40 \text{ g/(s}^2 \text{ m}^5)$, $m = 95 \text{ g}$, $g = 9.81 \text{ m/s}^2$, and $h = 0.43 \text{ m}$.

6.20 Aerospace engineers sometimes compute the trajectories of projectiles such as rockets. A related problem deals with the trajectory of a thrown ball. The trajectory of a ball thrown by a right fielder is defined by the (x, y) coordinates as displayed in Fig. P6.20. The trajectory can be modeled as

$$y = (\tan \theta_0)x - \frac{g}{2v_0^2 \cos^2 \theta_0} x^2$$

Find the appropriate initial angle θ_0 , if $v_0 = 30 \text{ m/s}$, and the distance to home plate is 90 m . Note that the throw leaves the right fielder's hand at an elevation of 1.8 m and the catcher receives it at 1 m .

6.21 You are designing a spherical tank (Fig. P6.21) to hold water for a small village in a developing country. The volume of liquid it can hold can be computed as

$$V = \pi h^2 \frac{[3R - h]}{3}$$

where V = volume [ft^3], h = depth of water in tank [ft], and R = the tank radius [ft].

If $R = 10 \text{ ft}$, what depth must the tank be filled to so that it holds 1000 ft^3 ? Use three iterations of the most efficient numerical method possible to determine your answer. Determine

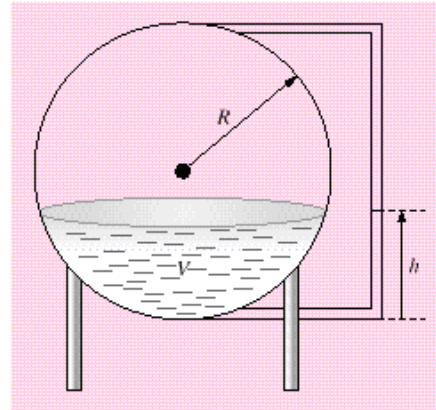


FIGURE P6.21

the approximate relative error after each iteration. Also, provide justification for your choice of method. Extra information: (a) For bracketing methods, initial guesses of 0 and R will bracket a single root for this example. (b) For open methods, an initial guess of R will always converge.

6.22 Perform the identical MATLAB operations as those in Example 6.7 to manipulate and find all the roots of the polynomial

$$f_5(x) = (x + 2)(x - 6)(x - 1)(x + 4)(x - 8)$$

6.23 In control systems analysis, transfer functions are developed that mathematically relate the dynamics of a system's input to its output. A transfer function for a robotic positioning system is given by

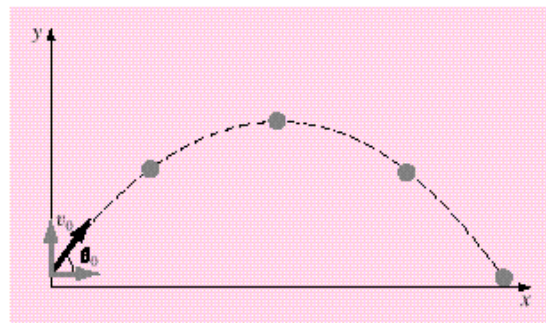
$$G(s) = \frac{C(s)}{N(s)} = \frac{s^3 + 9s^2 + 26s + 24}{s^4 + 15s^3 + 77s^2 + 153s + 90}$$

where $G(s)$ = system gain, $C(s)$ = system output, $N(s)$ = system input, and s = Laplace transform complex frequency. Use MATLAB to find the roots of the numerator and denominator and factor these into the form

$$G(s) = \frac{(s + a_1)(s + a_2)(s + a_3)}{(s + b_1)(s + b_2)(s + b_3)(s + b_4)}$$

where a_i and b_i = the roots of the numerator and denominator, respectively.

FIGURE P6.20



Linear Algebraic Equations and Matrices

CHAPTER OBJECTIVES

The primary objective of this chapter is to acquaint you with linear algebraic equations and their relationship to matrices and matrix algebra. Specific objectives and topics covered are

- Understanding what linear systems of equations are and where they occur in engineering and science.
- Understanding matrix notation.
- Being able to identify the following types of matrices: identity, diagonal, symmetric, triangular, and tridiagonal.
- Knowing how to perform matrix multiplication and being able to assess when it is feasible.
- Knowing how to represent a system of linear algebraic equations in matrix form.
- Knowing how to solve linear algebraic equations with left division and matrix inversion in MATLAB.

YOU'VE GOT A PROBLEM

Suppose that three jumpers are connected by bungee cords. Figure 7.1a shows them being held in place vertically so that each cord is fully extended but unstretched. We can define three distances, x_1 , x_2 , and x_3 , as measured downward from each of their unstretched positions. After they are released, gravity takes hold and the jumpers will eventually come to the equilibrium positions shown in Fig. 7.1b.

Suppose that you are asked to compute the displacement of each of the jumpers. If we assume that each cord behaves as a linear spring and follows Hooke's law, free-body diagrams can be developed for each jumper as depicted in Fig. 7.2.

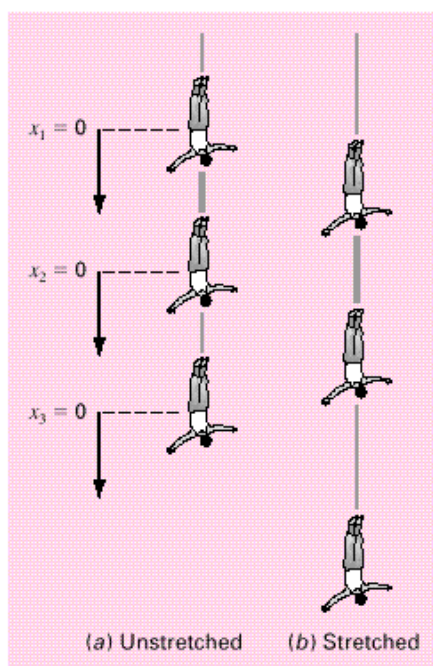


FIGURE 7.1
Three individuals connected by bungee cords.

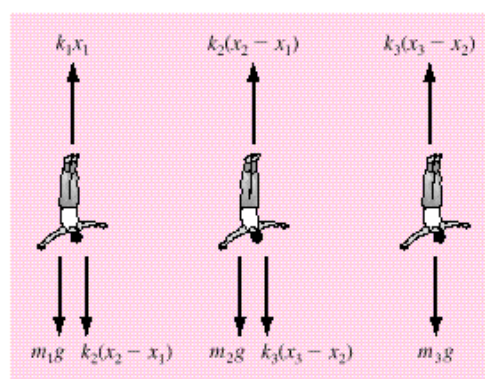


FIGURE 7.2
Free-body diagrams.

Using Newton's second law, a steady-state force balance can be written for each jumper:

$$\begin{aligned} m_1 g + k_2(x_2 - x_1) - k_1 x_1 &= 0 \\ m_2 g + k_3(x_3 - x_2) - k_2(x_2 - x_1) &= 0 \\ m_3 g - k_3(x_3 - x_2) &= 0 \end{aligned}$$

where m_i = the mass of jumper i (kg), k_j = the spring constant for cord j (N/m), x_i = the displacement of jumper i measured downward from the equilibrium position (m), and g = gravitational acceleration (9.81 m/s^2). Collecting terms gives

$$\begin{aligned} (k_1 + k_2)x_1 - k_2 x_2 &= m_1 g \\ -k_2 x_1 + (k_2 + k_3)x_2 - k_3 x_3 &= m_2 g \\ -k_3 x_2 + k_3 x_3 &= m_3 g \end{aligned} \quad (7.1)$$

Thus, the problem reduces to solving a system of three simultaneous equations for the three unknown displacements. Because we have used a linear law for the cords, these equations are linear algebraic equations. Chapters 7 through 11 will introduce you to how MATLAB is used to solve such systems of equations.

7.1 WHAT ARE LINEAR ALGEBRAIC EQUATIONS?

Linear algebraic equations are of the general form,

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\&\vdots \\a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n\end{aligned}\tag{7.2}$$

where the a 's are constant coefficients, the b 's are constants, the x 's are unknowns, and n is the number of equations. All other algebraic equations are nonlinear.

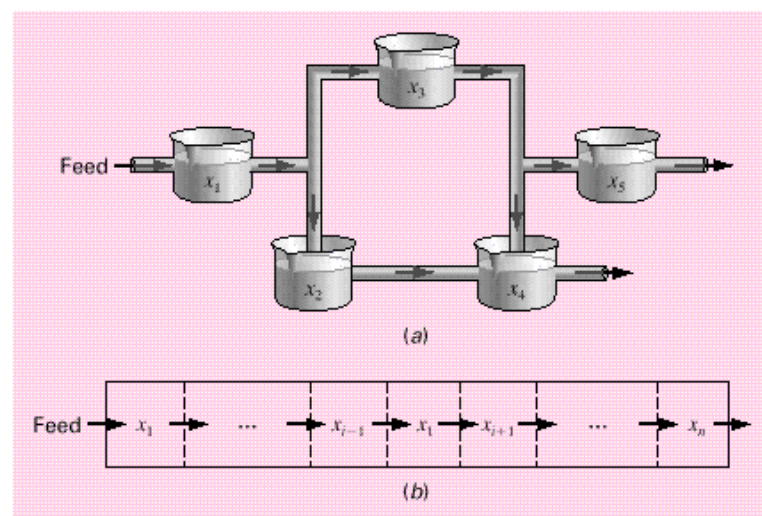
7.1.1 Linear Algebraic Equations and Engineering Practice

Many of the fundamental equations of engineering and science are based on conservation laws. Some familiar quantities that conform to such laws are mass, energy, and momentum. In mathematical terms, these principles lead to balance or continuity equations that relate system behavior as represented by the levels or response of the quantity being modeled to the properties or characteristics of the system and the external stimuli or forcing functions acting on the system.

As an example, the principle of mass conservation can be used to formulate a model for a series of chemical reactors (Fig. 7.3a). For this case, the quantity being modeled is the mass of the chemical in each reactor. The system properties are the reaction characteristics

FIGURE 7.3

Two types of systems that can be modeled using linear algebraic equations: (· · ·) lumped variable system that involves coupled finite components and (—) distributed variable system that involves a continuum.



of the chemical and the reactors' sizes and flow rates. The forcing functions are the feed rates of the chemical into the system.

When we studied roots of equations, you saw how single-component systems result in a single equation that can be solved using root-location techniques. Multicomponent systems result in a coupled set of mathematical equations that must be solved simultaneously. The equations are coupled because the individual parts of the system are influenced by other parts. For example, in Fig. 7.3a, reactor 4 receives chemical inputs from reactors 2 and 3. Consequently, its response is dependent on the quantity of chemical in these other reactors.

When these dependencies are expressed mathematically, the resulting equations are often of the linear algebraic form of Eq. (7.1). The x 's are usually measures of the magnitudes of the responses of the individual components. Using Fig. 7.3a as an example, x_1 might quantify the amount of chemical mass in the first reactor, x_2 might quantify the amount in the second, and so forth. The a 's typically represent the properties and characteristics that bear on the interactions between components. For instance, the a 's for Fig. 7.3a might be reflective of the flow rates of mass between the reactors. Finally, the b 's usually represent the forcing functions acting on the system, such as the feed rate in Fig. 7.3a.

Multicomponent problems of these types arise from both lumped (macro-) or distributed (micro-) variable mathematical models (Fig. 7.3). Lumped variable problems involve coupled finite components. Examples include trusses, reactors, and electric circuits. The three bungee jumpers at the beginning of this chapter are a lumped system.

Conversely, distributed variable problems attempt to describe the spatial detail of systems on a continuous or semicontinuous basis. The distribution of chemicals along the length of an elongated, rectangular reactor (Fig. 7.3b) is an example of a continuous variable model. Differential equations derived from conservation laws specify the distribution of the dependent variable for such systems. These differential equations can be solved numerically by converting them to an equivalent system of simultaneous algebraic equations.

The solution of such sets of equations represents a major engineering application area for the methods in the following chapters. These equations are coupled because the variables at one location are dependent on the variables in adjoining regions. For example, the concentration at the middle of the reactor in Fig. 7.3b is a function of the concentration in adjoining regions. Similar examples could be developed for the spatial distribution of temperature, momentum, or electricity.

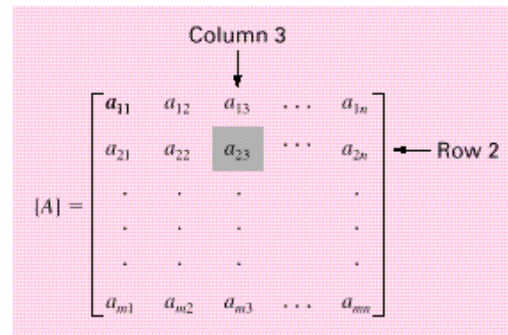
Aside from physical systems, simultaneous linear algebraic equations also arise in a variety of mathematical problem contexts. These result when mathematical functions are required to satisfy several conditions simultaneously. Each condition results in an equation that contains known coefficients and unknown variables. The techniques discussed in this part can be used to solve for the unknowns when the equations are linear and algebraic. Some widely used numerical techniques that employ simultaneous equations are regression analysis and spline interpolation.

7.2 MATRIX ALGEBRA OVERVIEW

Knowledge of matrices is essential for understanding the solution of linear algebraic equations. The following sections outline how matrices provide a concise way to represent and manipulate linear algebraic equations.

7.2 MATRIX ALGEBRA OVERVIEW

127

**FIGURE 7.4**

A matrix.

7.2.1 Matrix Notation

A *matrix* consists of a rectangular array of elements represented by a single symbol. As depicted in Fig. 7.4, $[A]$ is the shorthand notation for the matrix and a_{ij} designates an individual *element* of the matrix.

A horizontal set of elements is called a *row* and a vertical set is called a *column*. The first subscript i always designates the number of the row in which the element lies. The second subscript j designates the column. For example, element a_{23} is in row 2 and column 3.

The matrix in Fig. 7.4 has m rows and n columns and is said to have a dimension of m by n (or $m \times n$). It is referred to as an m by n matrix.

Matrices with row dimension $m = 1$, such as

$$[b] = [b_1 \quad b_2 \quad \cdots \quad b_n]$$

are called *row vectors*. Note that for simplicity, the first subscript of each element is dropped. Also, it should be mentioned that there are times when it is desirable to employ a special shorthand notation to distinguish a row matrix from other types of matrices. One way to accomplish this is to employ special open-topped brackets, as in $\cdot b \cdot$.¹

Matrices with column dimension $n = 1$, such as

$$[c] = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

are referred to as *column vectors*. For simplicity, the second subscript is dropped. As with the row vector, there are occasions when it is desirable to employ a special shorthand notation to distinguish a column matrix from other types of matrices. One way to accomplish this is to employ special brackets, as in $\cdot c \cdot$.

¹In addition to special brackets, we will use case to distinguish between vectors (lowercase) and matrices (uppercase).

Matrices where $m = n$ are called *square matrices*. For example, a 3×3 matrix is

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The diagonal consisting of the elements a_{11} , a_{22} , and a_{33} is termed the *principal* or *main diagonal* of the matrix.

Square matrices are particularly important when solving sets of simultaneous linear equations. For such systems, the number of equations (corresponding to rows) and the number of unknowns (corresponding to columns) must be equal for a unique solution to be possible. Consequently, square matrices of coefficients are encountered when dealing with such systems.

There are a number of special forms of square matrices that are important and should be noted:

A *symmetric matrix* is one where the rows equal the columns—that is, $a_{ij} = a_{ji}$ for all i 's and j 's. For example,

$$[A] = \begin{bmatrix} 5 & 1 & 2 \\ 1 & 3 & 7 \\ 2 & 7 & 8 \end{bmatrix}$$

is a 3×3 symmetric matrix.

A *diagonal matrix* is a square matrix where all elements off the main diagonal are equal to zero, as in

$$[A] = \begin{bmatrix} a_{11} & & \\ & a_{22} & \\ & & a_{33} \end{bmatrix}$$

Note that where large blocks of elements are zero, they are left blank.

An *identity matrix* is a diagonal matrix where all elements on the main diagonal are equal to 1, as in

$$[A] = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}$$

The symbol $[I]$ is used to denote the identity matrix. The identity matrix has properties similar to unity. That is,

$$[A][I] = [I][A] = [A]$$

An *upper triangular matrix* is one where all the elements below the main diagonal are zero, as in

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ & a_{22} & a_{23} \\ & & a_{33} \end{bmatrix}$$

7.2 MATRIX ALGEBRA OVERVIEW

129

A *lower triangular matrix* is one where all elements above the main diagonal are zero, as in

$$[A] = \begin{bmatrix} a_{11} & & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

A *banded matrix* has all elements equal to zero, with the exception of a band centered on the main diagonal:

$$[A] = \begin{bmatrix} a_{11} & a_{12} & & \\ a_{21} & a_{22} & a_{23} & \\ & a_{32} & a_{33} & a_{34} \\ & & a_{43} & a_{44} \end{bmatrix}$$

The preceding matrix has a bandwidth of 3 and is given a special name—the *tridiagonal matrix*.

7.2.2 Matrix Operating Rules

Now that we have specified what we mean by a matrix, we can define some operating rules that govern its use. Two m by n matrices are equal if, and only if, every element in the first is equal to every element in the second—that is, $[A] = [B]$ if $a_{ij} = b_{ij}$ for all i and j .

Addition of two matrices, say, $[A]$ and $[B]$, is accomplished by adding corresponding terms in each matrix. The elements of the resulting matrix $[C]$ are computed as

$$c_{ij} = a_{ij} + b_{ij}$$

for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$. Similarly, the subtraction of two matrices, say, $[E]$ minus $[F]$, is obtained by subtracting corresponding terms, as in

$$d_{ij} = e_{ij} - f_{ij}$$

for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$. It follows directly from the preceding definitions that addition and subtraction can be performed only between matrices having the same dimensions.

Both addition and subtraction are commutative:

$$[A] + [B] = [B] + [A]$$

and associative:

$$([A] + [B]) + [C] = [A] + ([B] + [C])$$

The multiplication of a matrix $[A]$ by a scalar g is obtained by multiplying every element of $[A]$ by g . For example, for a 3×3 matrix:

$$[D] = g[A] = \begin{bmatrix} ga_{11} & ga_{12} & ga_{13} \\ ga_{21} & ga_{22} & ga_{23} \\ ga_{31} & ga_{32} & ga_{33} \end{bmatrix}$$

The product of two matrices is represented as $[C] = [A][B]$, where the elements of $[C]$ are defined as

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad (7.3)$$

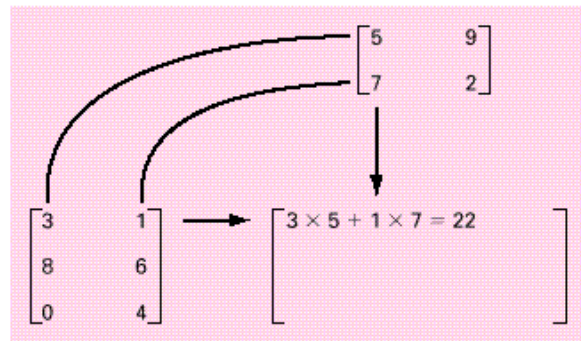


FIGURE 7.5

Visual depiction of how the rows and columns line up in matrix multiplication.

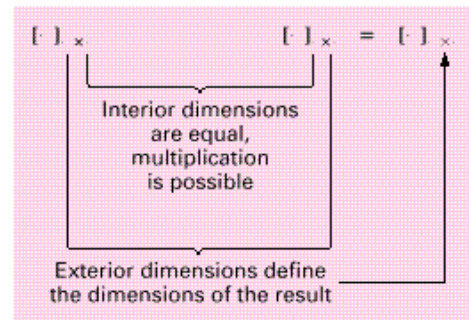


FIGURE 7.6

Matrix multiplication can be performed only if the inner dimensions are equal.

where n = the column dimension of $[A]$ and the row dimension of $[B]$. That is, the c_{ij} element is obtained by adding the product of individual elements from the i th row of the first matrix, in this case $[A]$, by the j th column of the second matrix $[B]$. Figure 7.5 depicts how the rows and columns line up in matrix multiplication.

According to this definition, matrix multiplication can be performed only if the first matrix has as many columns as the number of rows in the second matrix. Thus, if $[A]$ is an m by n matrix, $[B]$ could be an n by l matrix. For this case, the resulting $[C]$ matrix would have the dimension of m by l . However, if $[B]$ were an m by l matrix, the multiplication could not be performed. Figure 7.6 provides an easy way to check whether two matrices can be multiplied.

If the dimensions of the matrices are suitable, matrix multiplication is *associative*:

$$([A][B])[C] = [A]([B][C])$$

and *distributive*:

$$[A]([B] + [C]) = [A][B] + [A][C]$$

or

$$([A] + [B])[C] = [A][C] + [B][C]$$

However, multiplication is not generally *commutative*:

$$[A][B] \neq [B][A]$$

That is, the order of multiplication is important.

Although multiplication is possible, matrix division is not a defined operation. However, if a matrix $[A]$ is square and nonsingular, there is another matrix $[A]^{-1}$, called the *inverse* of $[A]$, for which

$$[A][A]^{-1} = [A]^{-1}[A] = [I]$$

Thus, the multiplication of a matrix by the inverse is analogous to division, in the sense that a number divided by itself is equal to 1. That is, multiplication of a matrix by its inverse leads to the identity matrix.

7.2 MATRIX ALGEBRA OVERVIEW

131

The inverse of a 2×2 matrix can be represented simply by

$$[A]^{-1} = \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

Similar formulas for higher-dimensional matrices are much more involved. Chapter 10 will deal with techniques for using numerical methods and the computer to calculate the inverse for such systems.

Two other matrix manipulations that will have utility in our discussion are the transpose and the augmentation of a matrix. The *transpose* of a matrix involves transforming its rows into columns and its columns into rows. For example, for the 3×3 matrix:

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

the transpose, designated $[A]^T$, is defined as

$$[A]^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

In other words, the element a_{ij} of the transpose is equal to the a_{ji} element of the original matrix.

The transpose has a variety of functions in matrix algebra. One simple advantage is that it allows a column vector to be written as a row, and vice versa. For example, if

$$c = \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \end{Bmatrix}$$

then

$$c^T = [c_1 \quad c_2 \quad c_3]$$

In addition, the transpose has numerous mathematical applications.

The final matrix manipulation that will have utility in our discussion is *augmentation*. A matrix is augmented by the addition of a column (or columns) to the original matrix. For example, suppose we have a 3×3 matrix of coefficients. We might wish to augment this matrix $[A]$ with a 3×3 identity matrix to yield a 3-by-6-dimensional matrix:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array} \right]$$

Such an expression has utility when we must perform a set of identical operations on the rows of two matrices. Thus, we can perform the operations on the single augmented matrix rather than on the two individual matrices.

EXAMPLE 7.1 MATLAB Matrix Manipulations

Problem Statement. The following example illustrates how a variety of matrix manipulations are implemented with MATLAB. It is best approached as a hands-on exercise on the computer.

Solution. Create a 3×3 matrix:

```
>> A = [1 5 6; 7 4 2; -3 6 7]
```

```
A =  
     1     5     6  
     7     4     2  
    -3     6     7
```

The transpose of $[A]$ can be obtained using the $'$ operator:

```
>> A'  
ans =  
     1     7    -3  
     5     4     6  
     6     2     7
```

Next we will create another 3×3 matrix on a row basis. First create three row vectors:

```
>> x = [8 6 9];  
>> y = [-5 8 1];  
>> z = [4 8 2];
```

Then we can combine these to form the matrix:

```
>> B = [x; y; z]  
B =  
     8     6     9  
    -5     8     1  
     4     8     2
```

We can add $[A]$ and $[B]$ together:

```
>> C = A+B  
C =  
     9    11    15  
     2    12     3  
     1    14     9
```

Further, we can subtract $[B]$ from $[C]$ to arrive back at $[A]$:

```
>> C = C-B  
C =  
     1     5     6  
     7     4     2  
    -3     6     7
```

Because their inner dimensions are equal, $[A]$ and $[B]$ can be multiplied

```
>> A*B  
ans =  
     7    94    26  
    44    90    71  
   -26    86   -7
```

7.2 MATRIX ALGEBRA OVERVIEW

133

Note that $[A]$ and $[B]$ can also be multiplied on an element-by-element basis by including a period with the multiplication operator as in

```
>> A.*B

ans =
     8     30     54
    -35     32     2
    -12     48    14
```

A 2×3 matrix can be set up

```
>> D = [1 4 3;5 8 1];
```

If $[A]$ is multiplied times $[D]$, an error message will occur

```
>> A*D

??? Error using ==> *
Inner matrix dimensions must agree.
```

However, if we reverse the order of multiplication so that the inner dimensions match, matrix multiplication works

```
>> D*A

ans =
    20    39    35
    58    63    53
```

The matrix inverse can be computed with the `inv` function:

```
>> AI = inv(A)

AI =
    0.2462    0.0154   -0.2154
   -0.8462    0.3846    0.6154
    0.8308   -0.3231   -0.4769
```

To test that this is the correct result, the inverse can be multiplied by the original matrix to give the identity matrix:

```
>> A*AI

ans =
    1.0000   -0.0000   -0.0000
    0.0000    1.0000   -0.0000
    0.0000   -0.0000    1.0000
```

The `eye` function can be used to generate an identity matrix:

```
>> I = eye(3)
```

```
I =
```

```
    1    0    0
    0    1    0
    0    0    1
```

Finally, matrices can be augmented simply as in

```
>> Aug = [A I]
```

```
Aug =
```

```
    1    5    6    1    0    0
    7    4    2    0    1    0
   -3    6    7    0    0    1
```

Note that the dimensions of a matrix can be determined by the `size` function:

```
>> [n,m] = size(Aug)
```

```
n =
```

```
    3
```

```
m =
```

```
    6
```

7.2.3 Representing Linear Algebraic Equations in Matrix Form

It should be clear that matrices provide a concise notation for representing simultaneous linear equations. For example, a 3×3 set of linear equations,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \quad (7.4)$$

can be expressed as

$$[A] \cdot x = b \quad (7.5)$$

where $[A]$ is the matrix of coefficients:

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

b is the column vector of constants:

$$b^T = [b_1 \quad b_2 \quad b_3]$$

and x is the column vector of unknowns:

$$x^T = [x_1 \quad x_2 \quad x_3]$$

Recall the definition of matrix multiplication [Eq. (7.3)] to convince yourself that Eqs. (7.4) and (7.5) are equivalent. Also, realize that Eq. (7.5) is a valid matrix multiplication because the number of columns n of the first matrix $[A]$ is equal to the number of rows n of the second matrix $\{x\}$.

This part of the book is devoted to solving Eq. (7.5) for $\{x\}$. A formal way to obtain a solution using matrix algebra is to multiply each side of the equation by the inverse of $[A]$ to yield

$$[A]^{-1}[A] \cdot x = [A]^{-1} \cdot b$$

Because $[A]^{-1}[A]$ equals the identity matrix, the equation becomes

$$x = [A]^{-1} \cdot b \quad (7.6)$$

Therefore, the equation has been solved for $\{x\}$. This is another example of how the inverse plays a role in matrix algebra that is similar to division. It should be noted that this is not a very efficient way to solve a system of equations. Thus, other approaches are employed in numerical algorithms. However, as discussed in Section 10.1.2, the matrix inverse itself has great value in the engineering analyses of such systems.

It should be noted that systems with more equations (rows) than unknowns (columns), $m > n$, are said to be *overdetermined*. A typical example is least-squares regression where an equation with n coefficients is fit to m data points (x, y) . Conversely, systems with less equations than unknowns, $m < n$, are said to be *underdetermined*. A typical example of underdetermined systems is numerical optimization.

7.3 SOLVING LINEAR ALGEBRAIC EQUATIONS WITH MATLAB

MATLAB provides two direct ways to solve systems of linear algebraic equations. The most efficient way is to employ the backslash, or “left-division,” operator as in

```
>> x = A\b
```

The second is to use matrix inversion:

```
>> x = inv(A)*b
```

As stated at the end of Section 7.2.3, the matrix inverse solution is less efficient than using the backslash. Both options are illustrated in the following example.

EXAMPLE 7.2 Solving the Bungee Jumper Problem with MATLAB

Problem Statement. Use MATLAB to solve the bungee jumper problem described at the beginning of this chapter. The parameters for the problem are

Jumper	Mass (kg)	Spring Constant (N/m)	Unstretched Cord Length (m)
Top (1)	60	50	20
Middle (2)	70	100	20
Bottom (3)	80	50	20

Solution. Substituting these parameter values into Eq. (7.1) gives

$$\begin{bmatrix} 150 & -100 & 0 \\ -100 & 150 & -50 \\ 0 & -50 & 50 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 588.6 \\ 686.7 \\ 784.8 \end{Bmatrix}$$

Start up MATLAB and enter the coefficient matrix and the right-hand-side vector:

```
>> K = [150 -100 0; -100 150 -50; 0 -50 50]
```

```
K =  
    150    -100     0  
   -100     150    -50  
     0     -50     50
```

```
>> mg = [588.6; 686.7; 784.8]
```

```
mg =  
    588.6000  
    686.7000  
    784.8000
```

Employing left division yields

```
>> x = K \ mg
```

```
x =  
    41.2020  
    55.9170  
    71.6130
```

Alternatively, multiplying the inverse of the coefficient matrix by the right-hand-side vector gives the same result:

```
>> x = inv(K) * mg
```

```
x =  
    41.2020  
    55.9170  
    71.6130
```

Because the jumpers were connected by 20-m cords, their initial positions relative to the platform is

```
>> xi = [20; 40; 60];
```

Thus, their final positions can be calculated as

```
>> xf = x + xi
```

```
xf =  
    61.2020  
    95.9170  
   131.6130
```

The results, which are displayed in Fig. 7.7, make sense. The first cord is extended the longest because it has a lower spring constant and is subject to the most weight (all three

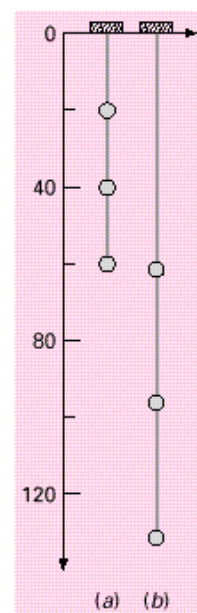


FIGURE 7.7
Positions of three
individuals
connected by
bungee cords.
[-] Unstretched
and [-] stretched.

PROBLEMS

137

jumpers). Notice that the second and third cords are extended about the same amount. Because it is subject to the weight of two jumpers, one might expect the second cord to be extended longer than the third. However, because it is stiffer (i.e., it has a higher spring constant), it stretches less than expected based on the weight it carries.

PROBLEMS

7.1 Given a square matrix $[A]$, write a single line MATLAB command that will create a new matrix $[Aug]$ that consists of the original matrix $[A]$ augmented by an identity matrix $[I]$.

7.2 A number of matrices are defined as

$$[A] = \begin{bmatrix} 4 & 5 \\ 1 & 2 \\ 5 & 6 \end{bmatrix} \quad [B] = \begin{bmatrix} 4 & 3 & 7 \\ 1 & 2 & 6 \\ 1 & 0 & 4 \end{bmatrix}$$

$$[C] = \begin{bmatrix} 2 \\ 6 \\ 1 \end{bmatrix} \quad [D] = \begin{bmatrix} 5 & 4 & 3 & 6 \\ 2 & 1 & 7 & 5 \end{bmatrix}$$

$$[E] = \begin{bmatrix} 1 & 5 & 6 \\ 7 & 1 & 3 \\ 4 & 0 & 5 \end{bmatrix} \quad [F] = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 6 & 3 \end{bmatrix}$$

$$[G] = \begin{bmatrix} 8 & 6 & 4 \end{bmatrix}$$

Answer the following questions regarding these matrices:

- What are the dimensions of the matrices?
- Identify the square, column, and row matrices.
- What are the values of these elements: a_{12} , b_{23} , d_{32} , e_{22} , f_{12} , g_{12} ?
- Perform the following operations:
 - $[E] \cdot [B]$
 - $[B] \cdot [E]$
 - $[A] \cdot [F]$
 - $5 \cdot [B]$
 - $[A] \cdot [B]$
 - $[B] \cdot [A]$
 - $[G] \cdot [C]$
 - $[C]^T$
 - $[D]^T$
 - $I \cdot [B]$

7.3 Write the following set of equations in matrix form:

$$\begin{aligned} 10 &= 3x_2 + 7x_1 \\ 4x_2 + 7x_3 &= 30 + 0 \\ x_1 + 7x_3 &= 40 + 3x_2 + 5x_1 \end{aligned}$$

Use MATLAB to solve for the unknowns. In addition, use it to compute the transpose and the inverse of the coefficient matrix.

7.4 Three matrices are defined as

$$[X] = \begin{bmatrix} 6 & 1 \\ 12 & 7 \\ 5 & 3 \end{bmatrix} \quad [Y] = \begin{bmatrix} 4 & 0 \\ 1 & 8 \end{bmatrix} \quad [Z] = \begin{bmatrix} 1 & 2 \\ 6 & 4 \end{bmatrix}$$

Perform all possible multiplications that can be computed between pairs of these matrices.

7.5 The position of three masses suspended vertically by a series of identical springs can be modeled by the following steady-state force balances:

$$\begin{aligned} 0 &= k(x_2 - x_1) - m_1g - kx_1 \\ 0 &= k(x_3 - x_2) - m_2g - k(x_2 - x_1) \\ 0 &= m_3g - k(x_3 - x_2) \end{aligned}$$

If $g = 9.81 \text{ m/s}^2$, $m_1 = 2 \text{ kg}$, $m_2 = 3 \text{ kg}$, $m_3 = 2.5 \text{ kg}$, and the k 's $= 10 \text{ N/m}$, use MATLAB to solve for the displacements x .

7.6 Five reactors linked by pipes are shown in Fig. P7.6. The rate of mass flow through each pipe is computed as the product of flow (Q) and concentration (c). At steady state, the mass flow into and out of each reactor must be equal. For example, for the first reactor, a *mass balance* can be written as

$$Q_{01}c_{01} = Q_{31}c_3 + Q_{15}c_1 + Q_{12}c_1$$

Write mass balances for the remaining reactors in Fig. P7.6 and express the equations in matrix form. Then use MATLAB to solve for the concentrations in each reactor.

7.7 An important problem in structural engineering is that of finding the forces in a statically determinate truss (Fig. P7.7). This type of structure can be described as a system of coupled linear algebraic equations derived from force balances. The sum of the forces in both horizontal and vertical directions must be zero at each node, because the system is at rest. Therefore, for node 1:

$$\begin{aligned} \sum F_H &= 0 = F_1 \cos 30^\circ + F_3 \cos 60^\circ - F_{1,h} \\ \sum F_V &= 0 = F_1 \sin 30^\circ + F_3 \sin 60^\circ - F_{1,v} \end{aligned}$$

for node 2:

$$\begin{aligned} \sum F_H &= 0 = F_2 + F_1 \cos 30^\circ - F_{2,h} - H_2 \\ \sum F_V &= 0 = F_1 \sin 30^\circ + F_{2,v} - V_2 \end{aligned}$$

for node 3:

$$\begin{aligned} \sum F_H &= 0 = F_2 + F_3 \cos 60^\circ - F_{3,h} \\ \sum F_V &= 0 = F_3 \sin 60^\circ + F_{3,v} - V_3 \end{aligned}$$

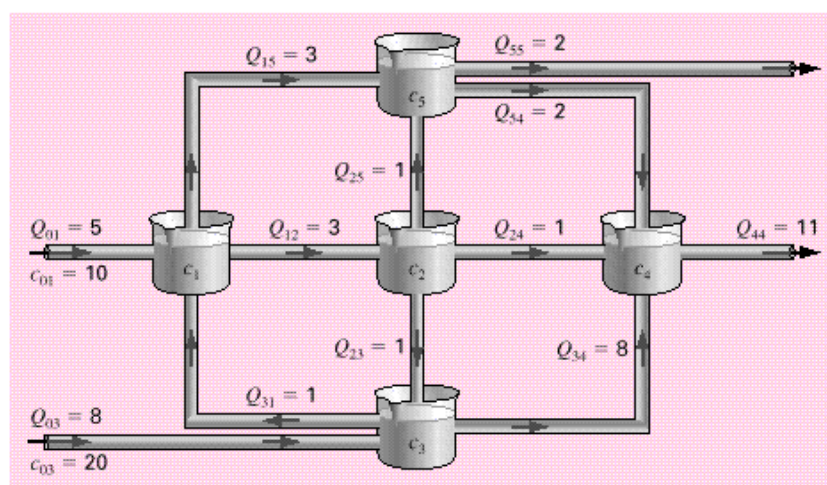


FIGURE P7.6

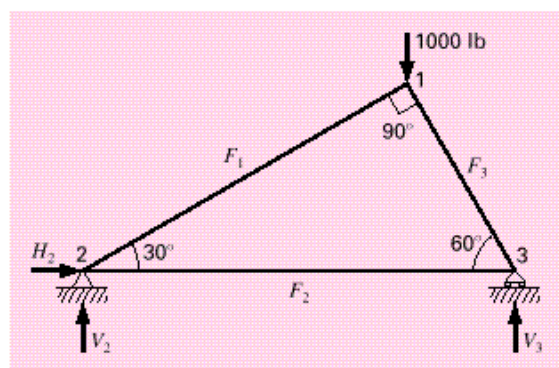


FIGURE P7.7

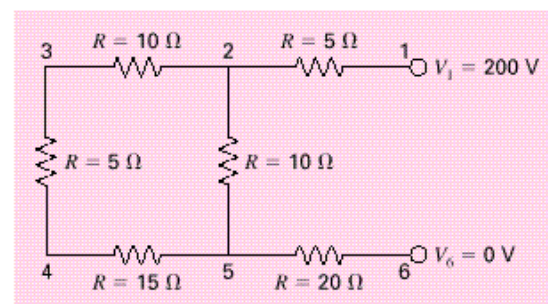
where $F_{i,h}$ is the external horizontal force applied to node i (where a positive force is from left to right) and $F_{i,v}$ is the external vertical force applied to node i (where a positive force is upward). Thus, in this problem, the 1000-lb downward force on node 1 corresponds to $F_{1,v} = -1000$. For this case all other $F_{i,v}$'s, and $F_{i,h}$'s are zero. Express this set of linear algebraic equations in matrix form and then use MATLAB to solve for the unknowns.

7.8 A common problem in electrical engineering involves determining the currents and voltages at various locations in resistor circuits. Such problems are solved using Kirchhoff's current and voltage rules. The current rule states that the

algebraic sum of all currents entering a node must be zero. The voltage rule specifies that the algebraic sum of the potential differences (i.e., voltage changes) in any loop must equal zero. Application of these rules results in systems of simultaneous linear algebraic equations because the various loops within a circuit are coupled. For example, for the circuit shown in Fig. P7.8, Kirchhoff's current rule is applied at each node to yield

$$\begin{aligned} i_{12} + i_{52} + i_{32} &= 0 \\ i_{65} + i_{52} + i_{54} &= 0 \\ i_{43} + i_{32} &= 0 \\ i_{54} + i_{43} &= 0 \end{aligned}$$

FIGURE P7.8



PROBLEMS

139

Application of the voltage rule together with Ohm's law to each of the two loops gives

$$\begin{aligned} i_{54}R_{54} + i_{43}R_{43} + i_{32}R_{32} + i_{52}R_{52} &= 0 \\ i_{65}R_{65} + i_{52}R_{52} + i_{12}R_{12} &= 200 = 0 \end{aligned}$$

where R_{ij} = the resistance between node i and j as shown in Fig. P7.8. Express this set of linear algebraic equations in matrix form and then use MATLAB to solve for the unknowns.

7.9 Consider the three mass-four spring system in Fig. P7.9. Determining the equations of motion from $\sum F_x = ma_x$ for each mass using its free-body diagram results in the following differential equations:

$$\begin{aligned} x_1 &= \left(\frac{k_1 + k_2}{m_1} \right) x_1 - \left(\frac{k_2}{m_1} \right) x_2 = 0 \\ x_2 &= \left(\frac{k_2}{m_2} \right) x_1 + \left(\frac{k_2 + k_3}{m_2} \right) x_2 - \left(\frac{k_3}{m_2} \right) x_3 = 0 \\ x_3 &= \left(\frac{k_3}{m_3} \right) x_2 + \left(\frac{k_3 + k_4}{m_3} \right) x_3 = 0 \end{aligned}$$

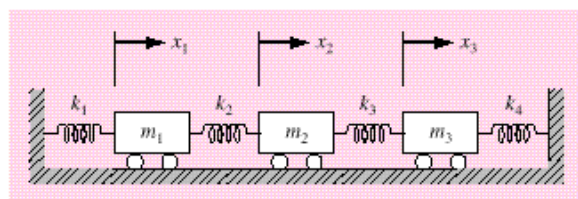


FIGURE P7.9

where $k_1 = k_4 = 10$ N/m, $k_2 = k_3 = 40$ N/m, and $m_1 = m_2 = m_3 = m_4 = 1$ kg. The three equations can be written in matrix form:

$$\begin{aligned} 0 &= \{\text{Acceleration vector}\} \\ &= [k/m \text{ matrix}]\{\text{displacement vector } x\} \end{aligned}$$

At a specific time where $x_1 = 0.05$ m, $x_2 = 0.04$ m, and $x_3 = 0.03$ m, this forms a tridiagonal matrix. Use MATLAB to solve for the acceleration of each mass.

Gauss Elimination

CHAPTER OBJECTIVES

The primary objective of this chapter is to describe the Gauss elimination algorithm for solving linear algebraic equations. Specific objectives and topics covered are

- Knowing how to solve small sets of linear equations with the graphical method and Cramer's rule.
- Understanding how to implement forward elimination and back substitution as in Gauss elimination.
- Understanding how to count flops to evaluate the efficiency of an algorithm.
- Understanding the concepts of singularity and ill-condition.
- Understanding how partial pivoting is implemented and how it differs from complete pivoting.
- Recognizing how the banded structure of a tridiagonal system can be exploited to obtain extremely efficient solutions.

At the end of Chap. 7, we stated that MATLAB provides two simple and direct methods for solving systems of linear algebraic equations: left-division,

```
>> x = A\b
```

and matrix inversion,

```
>> x = inv(A)*b
```

Chapters 8 and 9 provide background on how such solutions are obtained. This material is included to provide insight into how MATLAB operates. In addition, it is intended to show how you can build your own solution algorithms in computational environments that do not have MATLAB's built-in capabilities.

The technique described in this chapter is called Gauss elimination because it involves combining equations to eliminate unknowns. Although it is one of the earliest methods for solving simultaneous equations, it remains among the most important algorithms in use today and is the basis for linear equation solving on many popular software packages including MATLAB.

8.1 SOLVING SMALL NUMBERS OF EQUATIONS

Before proceeding to Gauss elimination, we will describe several methods that are appropriate for solving small ($n \leq 3$) sets of simultaneous equations and that do not require a computer. These are the graphical method, Cramer's rule, and the elimination of unknowns.

8.1.1 The Graphical Method

A graphical solution is obtainable for two linear equations by plotting them on Cartesian coordinates with one axis corresponding to x_1 and the other to x_2 . Because the equations are linear, each equation will plot as a straight line. For example, suppose that we have the following equations:

$$3x_1 + 2x_2 = 18$$

$$-x_1 + 2x_2 = 2$$

If we assume that x_1 is the abscissa, we can solve each of these equations for x_2 :

$$x_2 = -\frac{3}{2}x_1 + 9$$

$$x_2 = \frac{1}{2}x_1 + 1$$

The equations are now in the form of straight lines—that is, $x_2 = (\text{slope})x_1 + \text{intercept}$. When these equations are graphed, the values of x_1 and x_2 at the intersection of the lines represent the solution (Fig. 8.1). For this case, the solution is $x_1 = 4$ and $x_2 = 3$.

For three simultaneous equations, each equation would be represented by a plane in a three-dimensional coordinate system. The point where the three planes intersect would represent the solution. Beyond three equations, graphical methods break down and, consequently, have little practical value for solving simultaneous equations. However, they are useful in visualizing properties of the solutions.

For example, Fig. 8.2 depicts three cases that can pose problems when solving sets of linear equations. Fig. 8.2a shows the case where the two equations represent parallel lines. For such situations, there is no solution because the lines never cross. Figure 8.2b depicts the case where the two lines are coincident. For such situations there is an infinite number of solutions. Both types of systems are said to be *singular*.

In addition, systems that are very close to being singular (Fig. 8.2c) can also cause problems. These systems are said to be *ill-conditioned*. Graphically, this corresponds to the fact that it is difficult to identify the exact point at which the lines intersect. Ill-conditioned systems will also pose problems when they are encountered during the numerical solution of linear equations. This is because they will be extremely sensitive to round-off error.

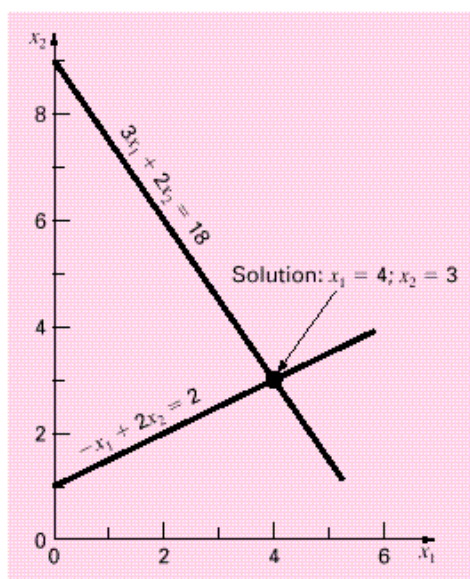


FIGURE 8.1

Graphical solution of a set of two simultaneous linear algebraic equations. The intersection of the lines represents the solution.

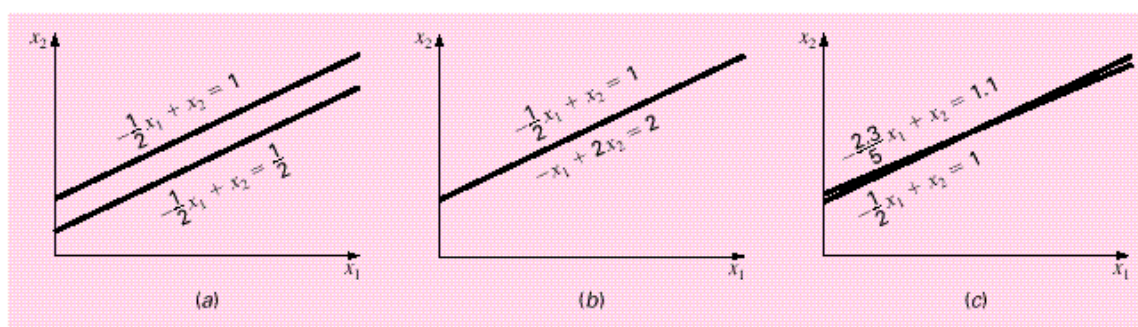


FIGURE 8.2

Graphical depiction of singular and ill-conditioned systems: (a) no solution, (b) infinite solutions, and (c) ill-conditioned system where the slopes are so close that the point of intersection is difficult to detect visually.

8.1.2 Determinants and Cramer's Rule

Cramer's rule is another solution technique that is best suited to small numbers of equations. Before describing this method, we will briefly review the concept of the determinant, which is used to implement Cramer's rule. In addition, the determinant has relevance to the evaluation of the ill-conditioning of a matrix.

Determinants. The determinant can be illustrated for a set of three equations:

$$[A]\{x\} = \{b\}$$

where $[A]$ is the coefficient matrix

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The *determinant* of this system is formed from the coefficients of $[A]$ and is represented as

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

Although the determinant D and the coefficient matrix $[A]$ are composed of the same elements, they are completely different mathematical concepts. That is why they are distinguished visually by using brackets to enclose the matrix and straight lines to enclose the determinant. In contrast to a matrix, the determinant is a single number. For example, the value of the determinant for two simultaneous equations

$$D = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

is calculated by

$$D = a_{11}a_{22} - a_{12}a_{21}$$

For the third-order case, the determinant can be computed as

$$D = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \quad (8.1)$$

where the 2 by 2 determinants are called *minors*.

EXAMPLE 8.1 Determinants

Problem Statement. Compute values for the determinants of the systems represented in Figs. 8.1 and 8.2.

Solution. For Fig. 8.1:

$$D = \begin{vmatrix} 3 & 2 \\ -1 & 2 \end{vmatrix} = 3(2) - 2(-1) = 8$$

For Fig. 8.2a:

$$D = \begin{vmatrix} -\frac{1}{2} & 1 \\ -\frac{1}{2} & 1 \end{vmatrix} = -\frac{1}{2}(1) - 1\left(\frac{-1}{2}\right) = 0$$

For Fig. 8.2b:

$$D = \begin{vmatrix} -\frac{1}{2} & 1 \\ -1 & 2 \end{vmatrix} = -\frac{1}{2}(2) - 1(-1) = 0$$

For Fig. 8.2c:

$$D = \begin{vmatrix} -\frac{1}{2} & 1 \\ -\frac{2.3}{5} & 1 \end{vmatrix} = -\frac{1}{2}(1) - 1\left(\frac{-2.3}{5}\right) = -0.04$$

In the foregoing example, the singular systems had zero determinants. Additionally, the results suggest that the system that is almost singular (Fig. 8.2c) has a determinant that is close to zero. These ideas will be pursued further in our subsequent discussion of ill-conditioning in Chap. 10.

Cramer's Rule. This rule states that each unknown in a system of linear algebraic equations may be expressed as a fraction of two determinants with denominator D and with the numerator obtained from D by replacing the column of coefficients of the unknown in question by the constants b_1, b_2, \dots, b_n . For example, for three equations, x_1 would be computed as

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}}{D}$$

EXAMPLE 8.2 Cramer's Rule

Problem Statement. Use Cramer's rule to solve

$$0.3x_1 + 0.52x_2 + x_3 = -0.01$$

$$0.5x_1 + x_2 + 1.9x_3 = 0.67$$

$$0.1x_1 + 0.3x_2 + 0.5x_3 = -0.44$$

Solution. The determinant D can be evaluated as [Eq. (8.1)]:

$$D = 0.3 \begin{vmatrix} 1 & 1.9 \\ 0.3 & 0.5 \end{vmatrix} - 0.52 \begin{vmatrix} 0.5 & 1.9 \\ 0.1 & 0.5 \end{vmatrix} + 1 \begin{vmatrix} 0.5 & 1 \\ 0.1 & 0.3 \end{vmatrix} = -0.0022$$

The solution can be calculated as

$$x_1 = \frac{\begin{vmatrix} -0.01 & 0.52 & 1 \\ 0.67 & 1 & 1.9 \\ -0.44 & 0.3 & 0.5 \end{vmatrix}}{-0.0022} = \frac{0.03278}{-0.0022} = -14.9$$

$$x_2 = \frac{\begin{vmatrix} 0.3 & -0.01 & 1 \\ 0.5 & 0.67 & 1.9 \\ 0.1 & -0.44 & 0.5 \end{vmatrix}}{-0.0022} = \frac{0.0649}{-0.0022} = -29.5$$

$$x_3 = \frac{\begin{vmatrix} 0.3 & 0.52 & -0.01 \\ 0.5 & 1 & 0.67 \\ 0.1 & 0.3 & -0.44 \end{vmatrix}}{-0.0022} = \frac{-0.04356}{-0.0022} = 19.8$$

For more than three equations, Cramer's rule becomes impractical because, as the number of equations increases, the determinants are time consuming to evaluate by hand (or by computer). Consequently, more efficient alternatives are used. Some of these alternatives are based on the last noncomputer solution technique covered in Section 8.1.3—the elimination of unknowns.

8.1.3 Elimination of Unknowns

The elimination of unknowns by combining equations is an algebraic approach that can be illustrated for a set of two equations:

$$a_{11}x_1 + a_{12}x_2 = b_1 \quad (8.2)$$

$$a_{21}x_1 + a_{22}x_2 = b_2 \quad (8.3)$$

The basic strategy is to multiply the equations by constants so that one of the unknowns will be eliminated when the two equations are combined. The result is a single equation that can be solved for the remaining unknown. This value can then be substituted into either of the original equations to compute the other variable.

For example, Eq. (8.2) might be multiplied by a_{21} and Eq. (8.3) by a_{11} to give

$$a_{21}a_{11}x_1 + a_{21}a_{12}x_2 = a_{21}b_1 \quad (8.4)$$

$$a_{11}a_{21}x_1 + a_{11}a_{22}x_2 = a_{11}b_2 \quad (8.5)$$

Subtracting Eq. (8.4) from Eq. (8.5) will, therefore, eliminate the x_1 term from the equations to yield

$$a_{11}a_{22}x_2 - a_{21}a_{12}x_2 = a_{11}b_2 - a_{21}b_1$$

which can be solved for

$$x_2 = \frac{a_{11}b_2 - a_{21}b_1}{a_{11}a_{22} - a_{21}a_{12}} \quad (8.6)$$

Equation (8.6) can then be substituted into Eq. (8.2), which can be solved for

$$x_1 = \frac{a_{22}b_1 - a_{12}b_2}{a_{11}a_{22} - a_{21}a_{12}} \quad (8.7)$$

Notice that Eqs. (8.6) and (8.7) follow directly from Cramer's rule:

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{12} \\ b_2 & a_{22} \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}} = \frac{a_{22}b_1 - a_{12}b_2}{a_{11}a_{22} - a_{21}a_{12}}$$
$$x_2 = \frac{\begin{vmatrix} a_{11} & b_1 \\ a_{21} & b_2 \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}} = \frac{a_{11}b_2 - a_{21}b_1}{a_{11}a_{22} - a_{21}a_{12}}$$

The elimination of unknowns can be extended to systems with more than two or three equations. However, the numerous calculations that are required for larger systems make the method extremely tedious to implement by hand. However, as described in Section 8.2, the technique can be formalized and readily programmed for the computer.

8.2 NAIVE GAUSS ELIMINATION

In Section 8.1.3, the elimination of unknowns was used to solve a pair of simultaneous equations. The procedure consisted of two steps (Fig. 8.3):

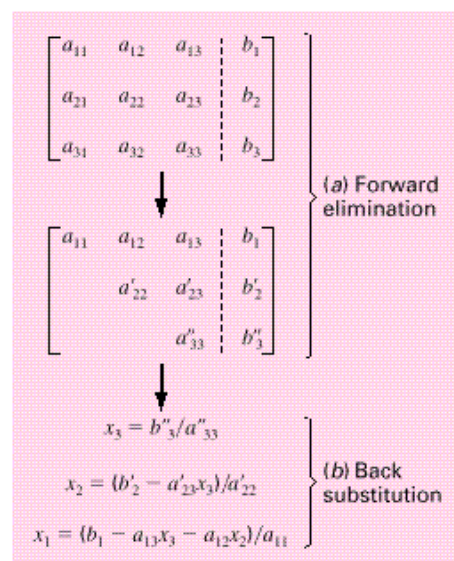
1. The equations were manipulated to eliminate one of the unknowns from the equations. The result of this elimination step was that we had one equation with one unknown.
2. Consequently, this equation could be solved directly and the result back-substituted into one of the original equations to solve for the remaining unknown.

This basic approach can be extended to large sets of equations by developing a systematic scheme or algorithm to eliminate unknowns and to back-substitute. Gauss elimination is the most basic of these schemes.

This section includes the systematic techniques for forward elimination and back substitution that comprise Gauss elimination. Although these techniques are ideally suited for implementation on computers, some modifications will be required to obtain a reliable algorithm. In particular, the computer program must avoid division by zero. The following method is called “naive” Gauss elimination because it does not avoid this problem. Section 8.3 will deal with the additional features required for an effective computer program.

FIGURE 8.3

The two phases of Gauss elimination: { - } forward elimination and { - } back substitution.



8.2 NAIVE GAUSS ELIMINATION

147

The approach is designed to solve a general set of n equations:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \quad (8.8a)$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \quad (8.8b)$$

$$\vdots \quad \vdots$$

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n \quad (8.8c)$$

As was the case with the solution of two equations, the technique for n equations consists of two phases: elimination of unknowns and solution through back substitution.

Forward Elimination of Unknowns. The first phase is designed to reduce the set of equations to an upper triangular system (Fig. 8.3a). The initial step will be to eliminate the first unknown x_1 from the second through the n th equations. To do this, multiply Eq. (8.8a) by a_{21}/a_{11} to give

$$a_{21}x_1 + \frac{a_{21}}{a_{11}}a_{12}x_2 + \frac{a_{21}}{a_{11}}a_{13}x_3 + \cdots + \frac{a_{21}}{a_{11}}a_{1n}x_n = \frac{a_{21}}{a_{11}}b_1 \quad (8.9)$$

Now this equation can be subtracted from Eq. (8.8b) to give

$$\left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)x_2 + \cdots + \left(a_{2n} - \frac{a_{21}}{a_{11}}a_{1n}\right)x_n = b_2 - \frac{a_{21}}{a_{11}}b_1$$

or

$$a'_{22}x_2 + \cdots + a'_{2n}x_n = b'_2$$

where the prime indicates that the elements have been changed from their original values.

The procedure is then repeated for the remaining equations. For instance, Eq. (8.8a) can be multiplied by a_{31}/a_{11} and the result subtracted from the third equation. Repeating the procedure for the remaining equations results in the following modified system:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \quad (8.10a)$$

$$a'_{22}x_2 + a'_{23}x_3 + \cdots + a'_{2n}x_n = b'_2 \quad (8.10b)$$

$$a'_{32}x_2 + a'_{33}x_3 + \cdots + a'_{3n}x_n = b'_3 \quad (8.10c)$$

$$\vdots \quad \vdots$$

$$a'_{n2}x_2 + a'_{n3}x_3 + \cdots + a'_{nn}x_n = b'_n \quad (8.10d)$$

For the foregoing steps, Eq. (8.8a) is called the *pivot equation* and a_{11} is called the *pivot element*. Note that the process of multiplying the first row by a_{21}/a_{11} is equivalent to dividing it by a_{11} and multiplying it by a_{21} . Sometimes the division operation is referred to as *normalization*. We make this distinction because a zero pivot element can interfere with normalization by causing a division by zero. We will return to this important issue after we complete our description of naive Gauss elimination.

The next step is to eliminate x_2 from Eq. (8.10c) through (8.10d). To do this multiply Eq. (8.10b) by a'_{32}/a'_{22} and subtract the result from Eq. (8.10c). Perform a similar

elimination for the remaining equations to yield

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 + \cdots + a'_{2n}x_n &= b'_2 \\ a''_{33}x_3 + \cdots + a''_{3n}x_n &= b''_3 \\ \vdots &\vdots \\ a''_{nn}x_n &= b''_n \end{aligned}$$

where the double prime indicates that the elements have been modified twice.

The procedure can be continued using the remaining pivot equations. The final manipulation in the sequence is to use the $(n - 1)$ th equation to eliminate the x_{n-1} term from the n th equation. At this point, the system will have been transformed to an upper triangular system:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \quad (8.11a)$$

$$a'_{22}x_2 + a'_{23}x_3 + \cdots + a'_{2n}x_n = b'_2 \quad (8.11b)$$

$$a''_{33}x_3 + \cdots + a''_{3n}x_n = b''_3 \quad (8.11c)$$

$$\begin{aligned} \ddots &\vdots \\ a^{(n-1)}_{nn}x_n &= b^{(n-1)}_n \end{aligned} \quad (8.11d)$$

Back Substitution. Equation (8.11d) can now be solved for x_n :

$$x_n = \frac{b^{(n-1)}_n}{a^{(n-1)}_{nn}} \quad (8.12)$$

This result can be back-substituted into the $(n - 1)$ th equation to solve for x_{n-1} . The procedure, which is repeated to evaluate the remaining x 's, can be represented by the following formula:

$$x_i = \frac{b^{(i-1)}_i - \sum_{j=i+1}^n a^{(i-1)}_{ij}x_j}{a^{(i-1)}_{ii}} \quad \text{for } i = n - 1, n - 2, \dots, 1 \quad (8.13)$$

EXAMPLE 8.3 Naive Gauss Elimination

Problem Statement. Use Gauss elimination to solve

$$3x_1 - 0.1x_2 - 0.2x_3 = 7.85 \quad (\text{E8.3.1})$$

$$0.1x_1 + 7x_2 - 0.3x_3 = -19.3 \quad (\text{E8.3.2})$$

$$0.3x_1 - 0.2x_2 + 10x_3 = 71.4 \quad (\text{E8.3.3})$$

Solution. The first part of the procedure is forward elimination. Multiply Eq. (E8.3.1) by 0.1/3 and subtract the result from Eq. (E8.3.2) to give

$$7.00333x_2 - 0.293333x_3 = -19.5617$$

8.2 NAIVE GAUSS ELIMINATION

149

Then multiply Eq. (E8.3.1) by 0.3/3 and subtract it from Eq. (E8.3.3). After these operations, the set of equations is

$$3x_1 - 0.1x_2 - 0.2x_3 = 7.85 \quad (\text{E8.3.4})$$

$$7.00333x_2 - 0.293333x_3 = -19.5617 \quad (\text{E8.3.5})$$

$$-0.190000x_2 + 10.0200x_3 = 70.6150 \quad (\text{E8.3.6})$$

To complete the forward elimination, x_2 must be removed from Eq. (E8.3.6). To accomplish this, multiply Eq. (E8.3.5) by $-0.190000/7.00333$ and subtract the result from Eq. (E8.3.6). This eliminates x_2 from the third equation and reduces the system to an upper triangular form, as in

$$3x_1 - 0.1x_2 - 0.2x_3 = 7.85 \quad (\text{E8.3.7})$$

$$7.00333x_2 - 0.293333x_3 = -19.5617 \quad (\text{E8.3.8})$$

$$10.0120x_3 = 70.0843 \quad (\text{E8.3.9})$$

We can now solve these equations by back substitution. First, Eq. (E8.3.9) can be solved for

$$x_3 = \frac{70.0843}{10.0120} = 7.00003$$

This result can be back-substituted into Eq. (E8.3.8), which can then be solved for

$$x_2 = \frac{-19.5617 + 0.293333(7.00003)}{7.00333} = -2.50000$$

Finally, $x_3 = 7.00003$ and $x_2 = -2.50000$ can be substituted back into Eq. (E8.3.7), which can be solved for

$$x_1 = \frac{7.85 + 0.1(-2.50000) + 0.2(7.00003)}{3} = 3.00000$$

Although there is a slight round-off error, the results are very close to the exact solution of $x_1 = 3$, $x_2 = -2.5$, and $x_3 = 7$. This can be verified by substituting the results into the original equation set:

$$3(3) - 0.1(-2.5) - 0.2(7.00003) = 7.84999 \cong 7.85$$

$$0.1(3) + 7(-2.5) - 0.3(7.00003) = -19.30000 = -19.3$$

$$0.3(3) - 0.2(-2.5) + 10(7.00003) = 71.4003 \cong 71.4$$

8.2.1 MATLAB M-file: GaussNaive

An M-file that implements naive Gauss elimination is listed in Fig. 8.4. Notice that the coefficient matrix **A** and the right-hand-side vector **b** are combined in the augmented matrix **Aug**. Thus, the operations are performed on **Aug** rather than separately on **A** and **b**.

Two nested loops provide a concise representation of the forward elimination step. An outer loop moves down the matrix from one pivot row to the next. The inner loop moves

```
function x = GaussNaive(A,b)
% GaussNaive(A,b):
%   Gauss elimination without pivoting.
% input:
%   A = coefficient matrix
%   b = right hand side vector
% output:
%   x = solution vector

[m,n] = size(A);
if m ~= n, error('Matrix A must be square'); end
nb = n+1;
Aug = [A b];
% forward elimination
for k = 1:n-1
    for i = k+1:n
        factor = Aug(i,k)/Aug(k,k);
        Aug(i,k:nb) = Aug(i,k:nb)-factor*Aug(k,k:nb);
    end
end
% back substitution
x = zeros(n,1);
x(n) = Aug(n,nb)/Aug(n,n);
for i = n-1:-1:1
    x(i) = (Aug(i,nb)-Aug(i,i+1:n)*x(i+1:n))/Aug(i,i);
end
```

FIGURE 8.4

An M-file to implement naive Gauss elimination.

below the pivot row to each of the subsequent rows where elimination is to take place. Finally, the actual elimination is represented by a single line that takes advantage of MATLAB's ability to perform matrix operations.

The back-substitution step follows directly from Eqs. (8.12) and (8.13). Again, MATLAB's ability to perform matrix operations allows Eq. (8.13) to be programmed as a single line.

8.2.2 Operation Counting

The execution time of Gauss elimination depends on the amount of *floating-point operations* (or *flops*) involved in the algorithm. On modern computers using math coprocessors, the time consumed to perform addition/subtraction and multiplication/division is about the same. Therefore, totaling up these operations provides insight into which parts of the algorithm are most time consuming and how computation time increases as the system gets larger.

8.2 NAIVE GAUSS ELIMINATION

151

Before analyzing naive Gauss elimination, we will first define some quantities that facilitate operation counting:

$$\sum_{i=1}^m cf(i) = c \sum_{i=1}^m f(i) \quad \sum_{i=1}^m f(i) + g(i) = \sum_{i=1}^m f(i) + \sum_{i=1}^m g(i) \quad (8.14a,b)$$

$$\sum_{i=1}^m 1 = 1 + 1 + 1 + \cdots + 1 = m \quad \sum_{i=k}^m 1 = m - k + 1 \quad (8.14c,d)$$

$$\sum_{i=1}^m i = 1 + 2 + 3 + \cdots + m = \frac{m(m+1)}{2} = \frac{m^2}{2} + O(m) \quad (8.14e)$$

$$\sum_{i=1}^m i^2 = 1^2 + 2^2 + 3^2 + \cdots + m^2 = \frac{m(m+1)(2m+1)}{6} = \frac{m^3}{3} + O(m^2) \quad (8.14f)$$

where $O(m^n)$ means “terms of order m^n and lower.”

Now let us examine the naive Gauss elimination algorithm (Fig. 8.4) in detail. We will first count the flops in the elimination stage. On the first pass through the outer loop, $k = 1$. Therefore, the limits on the inner loop are from $i = 2$ to n . According to Eq. (8.14d), this means that the number of iterations of the inner loop will be

$$\sum_{i=2}^n 1 = n - 2 + 1 = n - 1 \quad (8.15)$$

For every one of these iterations, there is one division to calculate the factor. The next line then performs a multiplication and a subtraction for each column element from 2 to nb . Because $nb = n + 1$, going from 2 to nb results in n multiplications and n subtractions. Together with the single division, this amounts to $n + 1$ multiplications/divisions and n addition/subtractions for every iteration of the inner loop. The total for the first pass through the outer loop is therefore $(n - 1)(n + 1)$ multiplication/divisions and $(n - 1)(n)$ addition/subtractions.

Similar reasoning can be used to estimate the flops for the subsequent iterations of the outer loop. These can be summarized as

Outer Loop	Inner Loop	Addition/Subtraction flops	Multiplication/Division flops
1	2, n	$(n - 1) n $	$(n - 1)(n + 1)$
2	3, n	$(n - 2) n - 1 $	$(n - 2) n $
\vdots	\vdots		
\cdot	$k + 1, n$	$(n - \cdot) n + 1 - k $	$(n - k)(n + 2 - \cdot)$
\vdots	\vdots		
$n - 1$	n, n	$(1) 2 $	$(1) 3 $

Therefore, the total addition/subtraction flops for elimination can be computed as

$$\sum_{k=1}^{n-1} (n - k)(n + 1 - k) = \sum_{k=1}^{n-1} [n(n + 1) - k(2n + 1) + k^2] \quad (8.16)$$

or

$$n(n+1) \sum_{k=1}^{n-1} 1 - (2n+1) \sum_{k=1}^{n-1} k + \sum_{k=1}^{n-1} k^2 \quad (8.17)$$

Applying some of the relationships from Eq. (8.14) yields

$$[n^3 + O(n)] - [n^3 + O(n^2)] + \left[\frac{1}{3}n^3 + O(n^2) \right] = \frac{n^3}{3} + O(n) \quad (8.18)$$

A similar analysis for the multiplication/division flops yields

$$[n^3 + O(n^2)] - [n^3 + O(n)] + \left[\frac{1}{3}n^3 + O(n^2) \right] = \frac{n^3}{3} + O(n^2) \quad (8.19)$$

Summing these results gives

$$\frac{2n^3}{3} + O(n^2) \quad (8.20)$$

Thus, the total number of flops is equal to $2n^3/3$ plus an additional component proportional to terms of order n^2 and lower. The result is written in this way because as n gets large, the $O(n^2)$ and lower terms become negligible. We are therefore justified in concluding that for large n , the effort involved in forward elimination converges on $2n^3/3$.

Because only a single loop is used, back substitution is much simpler to evaluate. The number of addition/subtraction flops is equal to $n(n-1)/2$. Because of the extra division prior to the loop, the number of multiplication/division flops is $n(n+1)/2$. These can be added to arrive at a total of

$$n^2 + O(n) \quad (8.21)$$

Thus, the total effort in naive Gauss elimination can be represented as

$$\underbrace{\frac{2n^3}{3} + O(n^2)}_{\text{Forward elimination}} + \underbrace{n^2 + O(n)}_{\text{Back substitution}} \xrightarrow{\text{as } n \text{ increases}} \frac{2n^3}{3} + O(n^2) \quad (8.22)$$

Two useful general conclusions can be drawn from this analysis:

1. As the system gets larger, the computation time increases greatly. As in Table 8.1, the amount of flops increases nearly three orders of magnitude for every order of magnitude increase in the number of equations.

TABLE 8.1 Number of flops for naive Gauss elimination.

	Elimination	Back Substitution	Total Flops	$2^3 \cdot 3$	Percent Due to Elimination
10	705	100	805	667	87.58%
100	671550	10000	681550	666667	98.53%
1000	6.67×10^8	1×10^6	6.68×10^8	6.67×10^8	99.85%

2. Most of the effort is incurred in the elimination step. Thus, efforts to make the method more efficient should probably focus on this step.

8.3 PIVOTING

The primary reason that the foregoing technique is called “naive” is that during both the elimination and the back-substitution phases, it is possible that a division by zero can occur. For example, if we use naive Gauss elimination to solve

$$\begin{aligned}2x_2 + 3x_3 &= 8 \\4x_1 + 6x_2 + 7x_3 &= -3 \\2x_1 - 3x_2 + 6x_3 &= 5\end{aligned}$$

the normalization of the first row would involve division by $a_{11} = 0$. Problems may also arise when the pivot element is close, rather than exactly equal, to zero because if the magnitude of the pivot element is small compared to the other elements, then round-off errors can be introduced.

Therefore, before each row is normalized, it is advantageous to determine the coefficient with the largest absolute value in the column below the pivot element. The rows can then be switched so that the largest element is the pivot element. This is called *partial pivoting*.

If columns as well as rows are searched for the largest element and then switched, the procedure is called *complete pivoting*. Complete pivoting is rarely used because switching columns changes the order of the x 's and, consequently, adds significant and usually unjustified complexity to the computer program.

The following example illustrates the advantages of partial pivoting. Aside from avoiding division by zero, pivoting also minimizes round-off error. As such, it also serves as a partial remedy for ill-conditioning.

EXAMPLE 8.4 Partial Pivoting

Problem Statement. Use Gauss elimination to solve

$$\begin{aligned}0.0003x_1 + 3.0000x_2 &= 2.0001 \\1.0000x_1 + 1.0000x_2 &= 1.0000\end{aligned}$$

Note that in this form the first pivot element, $a_{11} = 0.0003$, is very close to zero. Then repeat the computation, but partial pivot by reversing the order of the equations. The exact solution is $x_1 = 1/3$ and $x_2 = 2/3$.

Solution. Multiplying the first equation by $1/(0.0003)$ yields

$$x_1 + 10,000x_2 = 6667$$

which can be used to eliminate x_1 from the second equation:

$$-9999x_2 = -6666$$

which can be solved for $x_2 = 2/3$. This result can be substituted back into the first equation to evaluate x_1 :

$$x_1 = \frac{2.0001 - 3(2/3)}{0.0003} \quad (\text{E8.4.1})$$

Due to subtractive cancellation, the result is very sensitive to the number of significant figures carried in the computation:

Significant Figures	x_2	x_1	Absolute Value of Percent Relative Error for x_1
3	0.667	-3.33	1099
4	0.6667	0.0000	100
5	0.66667	0.30000	10
6	0.666667	0.330000	1
7	0.6666667	0.3330000	0.1

Note how the solution for x_1 is highly dependent on the number of significant figures. This is because in Eq. (E8.4.1), we are subtracting two almost-equal numbers.

On the other hand, if the equations are solved in reverse order, the row with the larger pivot element is normalized. The equations are

$$1.0000x_1 + 1.0000x_2 = 1.0000$$

$$0.0003x_1 + 3.0000x_2 = 2.0001$$

Elimination and substitution again yields $x_2 = 2/3$. For different numbers of significant figures, x_1 can be computed from the first equation, as in

$$x_1 = \frac{1 - (2/3)}{1}$$

This case is much less sensitive to the number of significant figures in the computation:

Significant Figures	x_2	x_1	Absolute Value of Percent Relative Error for x_1
3	0.667	0.333	0.1
4	0.6667	0.3333	0.01
5	0.66667	0.33333	0.001
6	0.666667	0.333333	0.0001
7	0.6666667	0.3333333	0.0000

Thus, a pivot strategy is much more satisfactory.

8.3.1 MATLAB M-file: GaussPivot

An M-file that implements Gauss elimination with partial pivoting is listed in Fig. 8.5. It is identical to the M-file for naive Gauss elimination presented previously in Section 8.2.1 with the exception of the bold portion that implements partial pivoting.

8.3 PIVOTING

155

```
function x = GaussPivot(A,b)
% GaussPivot(A,b):
%   Gauss elimination without pivoting.
% input:
%   A = coefficient matrix
%   b = right hand side vector
% output:
%   x = solution vector

[m,n] = size(A);
if m ~= n, error('Matrix A must be square'); end
nb = n+1;
Aug = [A b];
% forward elimination
for k = 1:n-1
    % partial pivoting
    [big,i] = max(abs(Aug(k:n,k)));
    ipr = i+k-1;
    if ipr ~= k
        % pivot the rows
        Aug([k,ipr],:) = Aug([ipr,k],:);
    end
    for i = k+1:n
        factor = Aug(i,k)/Aug(k,k);
        Aug(i,k:nb) = Aug(i,k:nb)-factor*Aug(k,k:nb);
    end
end
% back substitution
x = zeros(n,1);
x(n) = Aug(n,nb)/Aug(n,n);
for i = n-1:-1:1
    x(i) = (Aug(i,nb)-Aug(i,i+1:n)*x(i+1:n))/Aug(i,i);
end
```

FIGURE 8.5

An M-file to implement the Gauss elimination with partial pivoting.

Notice how the built-in MATLAB function `max` is used to determine the largest available coefficient in the column below the pivot element. The `max` function has the syntax

$$[y,i] = \max(x)$$

where y is the largest element in the vector x , and i is the index corresponding to that element.

8.4 TRIDIAGONAL SYSTEMS

Certain matrices have a particular structure that can be exploited to develop efficient solution schemes. For example, a banded matrix is a square matrix that has all elements equal to zero, with the exception of a band centered on the main diagonal.

A *tridiagonal* system has a bandwidth of 3 and can be expressed generally as

$$\begin{bmatrix} f_1 & g_1 & & & & \\ e_2 & f_2 & g_2 & & & \\ & e_3 & f_3 & g_3 & & \\ & & & \ddots & \ddots & \ddots \\ & & & & e_{n-1} & f_{n-1} & g_{n-1} \\ & & & & & e_n & f_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_{n-1} \\ r_n \end{bmatrix} \quad (8.23)$$

Notice that we have changed our notation for the coefficients from a 's and b 's to e 's, f 's, g 's, and r 's. This was done to avoid storing large numbers of useless zeros in the square matrix of a 's. This space-saving modification is advantageous because the resulting algorithm requires less computer memory.

An algorithm to solve such systems can be directly patterned after Gauss elimination—that is, using forward elimination and back substitution. However, because most of the matrix elements are already zero, much less effort is expended than for a full matrix. This efficiency is illustrated in the following example.

EXAMPLE 8.5 Solution of a Tridiagonal System

Problem Statement. Solve the following tridiagonal system:

$$\begin{bmatrix} 2.04 & -1 & & \\ -1 & 2.04 & -1 & \\ & -1 & 2.04 & -1 \\ & & -1 & 2.04 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 40.8 \\ 0.8 \\ 0.8 \\ 40.8 \end{bmatrix}$$

Solution. As with Gauss elimination, the first step involves transforming the matrix to upper triangular form. This is done by multiplying the first equation by the factor e_2/f_1 and subtracting the result from the second equation. This creates a zero in place of e_2 and transforms the other coefficients to new values,

$$f_2 = f_2 - \frac{e_2}{f_1}g_1 = 2.04 - \frac{-1}{2.04}(-1) = 1.550$$

$$r_2 = r_2 - \frac{e_2}{f_1}r_1 = 0.8 - \frac{-1}{2.04}(40.8) = 20.8$$

Notice that g_2 is unmodified because the element above it in the first row is zero.

After performing a similar calculation for the third and fourth rows, the system is transformed to the upper triangular form

$$\begin{bmatrix} 2.04 & -1 & & \\ & 1.550 & -1 & \\ & & 1.395 & -1 \\ & & & 1.323 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 40.8 \\ 20.8 \\ 14.221 \\ 210.996 \end{bmatrix}$$

8.4 TRIDIAGONAL SYSTEMS

157

Now back substitution can be applied to generate the final solution:

$$x_4 = \frac{r_4}{f_4} = \frac{210.996}{1.323} = 159.480$$

$$x_3 = \frac{r_3 - g_3 x_4}{f_3} = \frac{14.221 - (-1)159.480}{1.395} = 124.538$$

$$x_2 = \frac{r_2 - g_2 x_3}{f_2} = \frac{20.800 - (-1)124.538}{1.550} = 93.778$$

$$x_1 = \frac{r_1 - g_1 x_2}{f_1} = \frac{40.800 - (-1)93.778}{2.040} = 65.970$$

8.4.1 MATLAB M-file: Tridiag

An M-file that solves a tridiagonal system of equations is listed in Fig. 8.6. Note that the algorithm does not include partial pivoting. Although pivoting is sometimes required, most tridiagonal systems routinely solved in engineering and science do not require pivoting.

FIGURE 8.6

An M-file to solve a tridiagonal system.

```
function x = Tridiag(e,f,g,r)
% Tridiag(e,f,g,r):
%   Tridiagonal system solver.
% input:
%   e = subdiagonal vector
%   f = diagonal vector
%   g = superdiagonal vector
%   r = right hand side vector
% output:
%   x = solution vector

n = length(f);
% forward elimination
for k = 2:n
    factor = e(k)/f(k-1);
    f(k) = f(k) - factor*g(k-1);
    r(k) = r(k) - factor*r(k-1);
end
% back substitution
x(n) = r(n)/f(n);
for k = n-1:-1:1
    x(k) = (r(k)-g(k)*x(k+1))/f(k);
end
```


Recall that the computational effort for Gauss elimination was proportional to n^3 . Because of its sparseness, the effort involved in solving tridiagonal systems is proportional to n . Consequently, the algorithm in Fig. 8.6 executes much, much faster than Gauss elimination, particularly for large systems.

PROBLEMS

8.1 Determine the number of total flops as a function of the number of equations n for the tridiagonal algorithm (Fig. 8.6).

8.2 Use the graphical method to solve

$$\begin{aligned} 4x_1 - 8x_2 &= -24 \\ x_1 + 6x_2 &= 34 \end{aligned}$$

Check your results by substituting them back into the equations.

8.3 Given the system of equations

$$\begin{aligned} -1.1x_1 + 10x_2 &= 120 \\ -2x_1 + 17.4x_2 &= 174 \end{aligned}$$

- (a) Solve graphically and check your results by substituting them back into the equations.
- (b) On the basis of the graphical solution, what do you expect regarding the condition of the system?
- (c) Compute the determinant.

8.4 Given the system of equations

$$\begin{aligned} -3x_2 + 7x_3 &= 2 \\ x_1 + 2x_2 - x_3 &= 3 \\ 5x_1 - 2x_2 &= 2 \end{aligned}$$

- (a) Compute the determinant.
- (b) Use Cramer's rule to solve for the x 's.
- (c) Use Gauss elimination with partial pivoting to solve for the x 's.
- (d) Substitute your results back into the original equations to check your solution.

8.5 MATLAB has a built-in function that computes the determinant. It has the syntax,

$$d = \det(A)$$

where d is the determinant of the square matrix A . Use this function to compute the determinant for the systems from Probs. 8.3 and 8.4.

8.6 Given the equations

$$\begin{aligned} 0.5x_1 - x_2 &= -9.5 \\ 1.02x_1 - 2x_2 &= -18.8 \end{aligned}$$

- (a) Solve graphically.
- (b) Compute the determinant.
- (c) On the basis of (a) and (b), what would you expect regarding the system's condition?
- (d) Solve by the elimination of unknowns.
- (e) Solve again, but with a_{11} modified slightly to 0.52. Interpret your results.

8.7 Given the equations

$$\begin{aligned} 10x_1 + 2x_2 - x_3 &= 27 \\ -3x_1 - 6x_2 + 2x_3 &= -61.5 \\ x_1 + x_2 + 5x_3 &= -21.5 \end{aligned}$$

- (a) Solve by naive Gauss elimination. Show all steps of the computation.
- (b) Substitute your results into the original equations to check your answers.

8.8 Given the equations

$$\begin{aligned} 2x_1 - 6x_2 - x_3 &= -38 \\ -3x_1 - x_2 + 7x_3 &= -34 \\ -8x_1 + x_2 - 2x_3 &= -20 \end{aligned}$$

- (a) Solve by Gauss elimination with partial pivoting. Show all steps of the computation.
- (b) Substitute your results into the original equations to check your answers.

8.9 Perform the same calculations as in Example 8.5, but for the tridiagonal system:

$$\begin{bmatrix} 0.8 & -0.4 & 0 \\ -0.4 & 0.8 & -0.4 \\ 0 & -0.4 & 0.8 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 41 \\ 25 \\ 105 \end{Bmatrix}$$

8.10 Figure P8.10 shows three reactors linked by pipes. As indicated, the rate of transfer of chemicals through each pipe is equal to a flow rate (Q , with units of cubic meters per second) multiplied by the concentration of the reactor from which the flow originates (c , with units of milligrams per cubic meter). If the system is at a steady state, the transfer into each reactor will balance the transfer out. Develop mass-balance equations for the reactors and solve the three simultaneous linear algebraic equations for their concentrations.

PROBLEMS

159

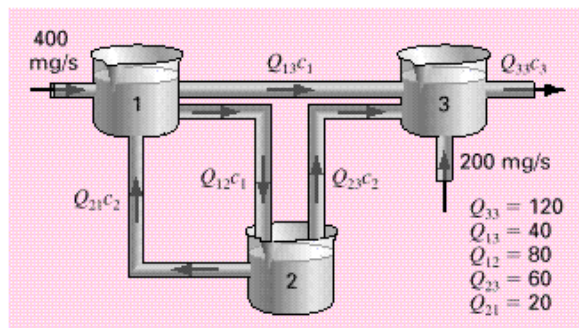


FIGURE P8.10

Three reactors linked by pipes. The rate of mass transfer through each pipe is equal to the product of flow and concentration of the reactor from which the flow originates.

8.11 A civil engineer involved in construction requires 6000, 5000, and 8000 m³ of sand, fine gravel, and coarse gravel, respectively, for a building project. There are three pits from which these materials can be obtained. The composition of these pits is

	Sand %	Fine Gravel %	Coarse Gravel %
Pit 1	32	30	38
Pit 2	25	40	35
Pit 3	35	15	50

How many cubic meters must be hauled from each pit in order to meet the engineer's needs?

8.12 Linear algebraic equations can arise in the solution of differential equations. For example, the following differential equation derives from a heat balance for a long, thin rod (Fig. P8.12):

$$\frac{d^2 T}{dx^2} + h'(T_a - T) = 0 \quad (\text{P8.12.1})$$

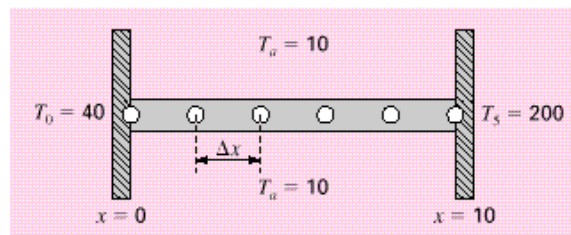


FIGURE P8.12

A noninsulated uniform rod positioned between two walls of constant but different temperature. The finite-difference representation employs four interior nodes.

where T = temperature (°C), x = distance along the rod (m), h' = a heat transfer coefficient between the rod and the ambient air (m⁻²), and T_a = the temperature of the surrounding air (°C). This equation can be transformed into a set of linear algebraic equations by using a finite-divided-difference approximation for the second derivative (recall Section 4.3.3):

$$\frac{d^2 T}{dx^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$

where T_i designates the temperature at node i . This approximation can be substituted into Eq. (P8.12.1) to give

$$-T_{i-1} + (2 + h'\Delta x^2)T_i - T_{i+1} = h'\Delta x^2 T_a$$

This equation can be written for each of the interior nodes of the rod resulting in a tridiagonal system of equations. The first and last nodes at the rods ends are fixed by boundary conditions.

Develop a numerical solution for Eq. (P8.12.1) for a 10-m rod with $T_a = 20$, $T(x = 0) = 40$, $T(x = 10) = 200$, and $h' = 0.05$. Use a finite-difference solution with four interior nodes as shown in Fig. P8.12 ($\Delta x = 2$ m).

.. Decomposition

CHAPTER OBJECTIVES

The primary objective of this chapter is to acquaint you with *LU* decomposition. Specific objectives and topics covered are

- Understanding that *LU* decomposition involves factoring the coefficient matrix into two triangular matrices that can then be used to efficiently evaluate different right-hand-side vectors.
- Knowing how to express Gauss elimination as an *LU* decomposition.
- Given an *LU* decomposition, knowing how to evaluate multiple right-hand-side vectors.
- Recognizing that Cholesky's method provides an efficient way to decompose a symmetric matrix and that the resulting triangular matrix and its transpose can be used to evaluate right-hand-side vectors efficiently.
- Understanding in general terms what happens when MATLAB's backslash operator is used to solve linear systems.

As described in Chap. 8, Gauss elimination is designed to solve systems of linear algebraic equations:

$$[A] \cdot x = b \quad (9.1)$$

Although it certainly represents a sound way to solve such systems, it becomes inefficient when solving equations with the same coefficients $[A]$, but with different right-hand-side constants b .

Recall that Gauss elimination involves two steps: forward elimination and back substitution (Fig. 8.3). As we learned in Section 8.2.2, the forward-elimination step comprises the bulk of the computational effort. This is particularly true for large systems of equations.

LU decomposition methods separate the time-consuming elimination of the matrix $[A]$ from the manipulations of the right-hand side $\cdot b \cdot$. Thus, once $[A]$ has been “decomposed,” multiple right-hand-side vectors can be evaluated in an efficient manner.

Interestingly, Gauss elimination itself can be expressed as an *LU* decomposition. Before showing how this can be done, let us first provide a mathematical overview of the decomposition strategy.

9.1 OVERVIEW OF LU DECOMPOSITION

Just as was the case with Gauss elimination, *LU* decomposition requires pivoting to avoid division by zero. However, to simplify the following description, we will omit pivoting. In addition, the following explanation is limited to a set of three simultaneous equations. The results can be directly extended to n -dimensional systems.

Equation (9.1) can be rearranged to give

$$[A] \cdot x \cdot = \cdot b \cdot = 0 \quad (9.2)$$

Suppose that Eq. (9.2) could be expressed as an upper triangular system. For example for a 3×3 system:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \end{Bmatrix} \quad (9.3)$$

Recognize that this is similar to the manipulation that occurs in the first step of Gauss elimination. That is, elimination is used to reduce the system to upper triangular form. Equation (9.3) can also be expressed in matrix notation and rearranged to give

$$[U] \cdot x \cdot = \cdot d \cdot = 0 \quad (9.4)$$

Now assume that there is a lower diagonal matrix with 1's on the diagonal,

$$[L] = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \quad (9.5)$$

that has the property that when Eq. (9.4) is premultiplied by it, Eq. (9.2) is the result. That is,

$$[L] \cdot [U] \cdot x \cdot = \cdot d \cdot = [A] \cdot x \cdot = \cdot b \cdot \quad (9.6)$$

If this equation holds, it follows from the rules for matrix multiplication that

$$[L][U] = [A] \quad (9.7)$$

and

$$[L] \cdot d \cdot = \cdot b \cdot \quad (9.8)$$

A two-step strategy (see Fig. 9.1) for obtaining solutions can be based on Eqs. (9.3), (9.7), and (9.8):

1. *LU* decomposition step. $[A]$ is factored or “decomposed” into lower $[L]$ and upper $[U]$ triangular matrices.

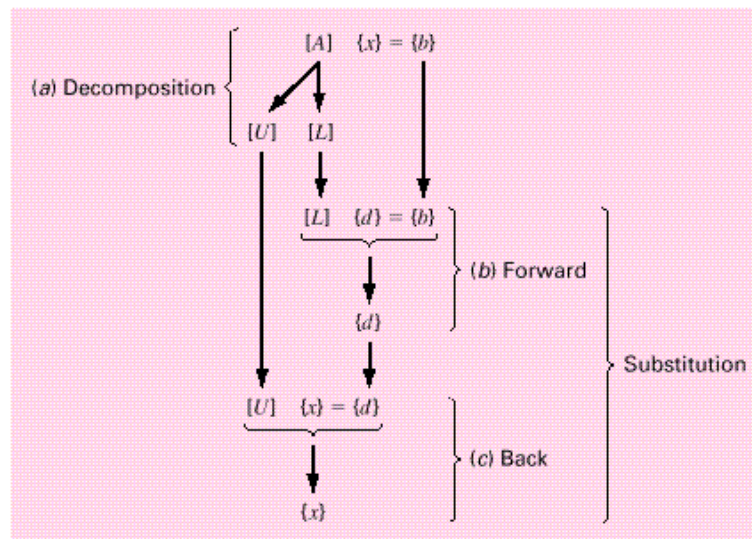


FIGURE 9.1

The steps in LU decomposition.

2. Substitution step. $[L]$ and $[U]$ are used to determine a solution $\{x\}$ for a right-hand side $\{b\}$. This step itself consists of two steps. First, Eq. (9.8) is used to generate an intermediate vector $\{d\}$ by forward substitution. Then, the result is substituted into Eq. (9.3) which can be solved by back substitution for $\{x\}$.

Now let us show how Gauss elimination can be implemented in this way.

9.2 GAUSS ELIMINATION AS LU DECOMPOSITION

Although it might appear at face value to be unrelated to LU decomposition, Gauss elimination can be used to decompose $[A]$ into $[L]$ and $[U]$. This can be easily seen for $[U]$, which is a direct product of the forward elimination. Recall that the forward-elimination step is intended to reduce the original coefficient matrix $[A]$ to the form

$$[U] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix} \quad (9.9)$$

which is in the desired upper triangular format.

Though it might not be as apparent, the matrix $[L]$ is also produced during the step. This can be readily illustrated for a three-equation system,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \end{Bmatrix}$$

9.2 GAUSS ELIMINATION AND LU DECOMPOSITION

163

The first step in Gauss elimination is to multiply row 1 by the factor [recall Eq. (8.9)]

$$f_{21} = \frac{a_{21}}{a_{11}}$$

and subtract the result from the second row to eliminate a_{21} . Similarly, row 1 is multiplied by

$$f_{31} = \frac{a_{31}}{a_{11}}$$

and the result subtracted from the third row to eliminate a_{31} . The final step is to multiply the modified second row by

$$f_{32} = \frac{a_{32}}{a_{22}}$$

and subtract the result from the third row to eliminate a_{32} .

Now suppose that we merely perform all these manipulations on the matrix $[A]$. Clearly, if we do not want to change the equations, we also have to do the same to the right-hand side $\cdot b \cdot$. But there is absolutely no reason that we have to perform the manipulations simultaneously. Thus, we could save the f 's and manipulate $\cdot b \cdot$ later.

Where do we store the factors f_{21} , f_{31} , and f_{32} ? Recall that the whole idea behind the elimination was to create zeros in a_{21} , a_{31} , and a_{32} . Thus, we can store f_{21} in a_{21} , f_{31} in a_{31} , and f_{32} in a_{32} . After elimination, the $[A]$ matrix can therefore be written as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ f_{21} & a_{22} & a_{23} \\ f_{31} & f_{32} & a_{33} \end{bmatrix} \quad (9.10)$$

This matrix, in fact, represents an efficient storage of the LU decomposition of $[A]$,

$$[A] = [L][U] \quad (9.11)$$

where

$$[U] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix} \quad (9.12)$$

and

$$[L] = \begin{bmatrix} 1 & 0 & 0 \\ f_{21} & 1 & 0 \\ f_{31} & f_{32} & 1 \end{bmatrix} \quad (9.13)$$

The following example confirms that $[A] = [L][U]$.

EXAMPLE 9.1 LU Decomposition with Gauss Elimination

Problem Statement. Derive an LU decomposition based on the Gauss elimination performed previously in Example 8.3.

Solution. In Example 8.3, we used Gauss elimination to solve a set of linear algebraic equations that had the following coefficient matrix:

$$[A] = \begin{bmatrix} 3 & 0.1 & 0.2 \\ 0.1 & 7 & 0.3 \\ 0.3 & 0.2 & 10 \end{bmatrix}$$

After forward elimination, the following upper triangular matrix was obtained:

$$[U] = \begin{bmatrix} 3 & 0.1 & 0.2 \\ 0 & 7.00333 & 0.293333 \\ 0 & 0 & 10.0120 \end{bmatrix}$$

The factors employed to obtain the upper triangular matrix can be assembled into a lower triangular matrix. The elements a_{21} and a_{31} were eliminated by using the factors

$$f_{21} = \frac{0.1}{3} = 0.0333333 \quad f_{31} = \frac{0.3}{3} = 0.100000$$

and the element a_{32} was eliminated by using the factor

$$f_{32} = \frac{0.19}{7.00333} = 0.0271300$$

Thus, the lower triangular matrix is

$$[L] = \begin{bmatrix} 1 & 0 & 0 \\ 0.0333333 & 1 & 0 \\ 0.100000 & 0.0271300 & 1 \end{bmatrix}$$

Consequently, the LU decomposition is

$$[A] = [L][U] = \begin{bmatrix} 1 & 0 & 0 \\ 0.0333333 & 1 & 0 \\ 0.100000 & 0.0271300 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0.1 & 0.2 \\ 0 & 7.00333 & 0.293333 \\ 0 & 0 & 10.0120 \end{bmatrix}$$

This result can be verified by performing the multiplication of $[L][U]$ to give

$$[L][U] = \begin{bmatrix} 3 & 0.1 & 0.2 \\ 0.0999999 & 7 & 0.3 \\ 0.3 & 0.2 & 9.99996 \end{bmatrix}$$

where the minor discrepancies are due to round-off.

After the matrix is decomposed, a solution can be generated for a particular right-hand-side vector b . This is done in two steps. First, a forward-substitution step is executed by solving Eq. (9.8) for d . It is important to recognize that this merely amounts to performing the elimination manipulations on b . Thus, at the end of this step, the right-hand side will be in the same state that it would have been had we performed forward manipulation on $[A]$ and b simultaneously.

9.2 GAUSS ELIMINATION AND DECOMPOSITION

165

The forward-substitution step can be represented concisely as

$$d_i = b_i - \sum_{j=1}^{i-1} l_{ij}b_j \quad \text{for } i = 1, 2, \dots, n$$

The second step then merely amounts to implementing back substitution to solve Eq. (9.3). Again, it is important to recognize that this is identical to the back-substitution phase of conventional Gauss elimination [compare with Eqs. (8.12) and (8.13)]:

$$x_n = d_n/a_{nn}$$

$$d_i = \sum_{j=i+1}^n u_{ij}x_j$$

$$x_i = \frac{d_i}{u_{ii}} \quad \text{for } i = n-1, n-2, \dots, 1$$

EXAMPLE 9.2 The Substitution Steps

Problem Statement. Complete the problem initiated in Example 9.1 by generating the final solution with forward and back substitution.

Solution. As just stated, the intent of forward substitution is to impose the elimination manipulations that we had formerly applied to $[A]$ on the right-hand-side vector b . Recall that the system being solved is

$$\begin{bmatrix} 3 & 0.1 & 0.2 \\ 0.1 & 7 & 0.3 \\ 0.3 & 0.2 & 10 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 7.85 \\ 19.3 \\ 71.4 \end{Bmatrix}$$

and that the forward-elimination phase of conventional Gauss elimination resulted in

$$\begin{bmatrix} 3 & 0.1 & 0.2 \\ 0 & 7.00333 & 0.293333 \\ 0 & 0 & 10.0120 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 7.85 \\ 19.5617 \\ 70.0843 \end{Bmatrix}$$

The forward-substitution phase is implemented by applying Eq. (9.8):

$$\begin{bmatrix} 1 & 0 & 0 \\ 0.0333333 & 1 & 0 \\ 0.100000 & 0.0271300 & 1 \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \end{Bmatrix} = \begin{Bmatrix} 7.85 \\ 19.3 \\ 71.4 \end{Bmatrix}$$

or multiplying out the left-hand side:

$$\begin{aligned} d_1 &= 7.85 \\ 0.0333333d_1 + d_2 &= 19.3 \\ 0.100000d_1 + 0.0271300d_2 + d_3 &= 71.4 \end{aligned}$$

We can solve the first equation for $d_1 = 7.85$, which can be substituted into the second equation to solve for

$$d_2 = 19.3 - 0.0333333(7.85) = 19.5617$$

Both d_1 and d_2 can be substituted into the third equation to give

$$d_3 = 71.4 - 0.1(7.85) - 0.02713(19.5617) = 70.0843$$

Thus,

$$d = \begin{Bmatrix} 7.85 \\ 19.5617 \\ 70.0843 \end{Bmatrix}$$

This result can then be substituted into Eq. (9.3), $[U]x = d$:

$$\begin{bmatrix} 3 & 0.1 & 0.2 \\ 0 & 7.00333 & 0.293333 \\ 0 & 0 & 10.0120 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 7.85 \\ 19.5617 \\ 70.0843 \end{Bmatrix}$$

which can be solved by back substitution (see Example 8.3 for details) for the final solution:

$$d = \begin{Bmatrix} 3 \\ 2.5 \\ 7.00003 \end{Bmatrix}$$

The LU decomposition algorithm requires the same total flops as for Gauss elimination. The only difference is that a little less effort is expended in the decomposition phase since the operations are not applied to the right-hand side. Conversely, the substitution phase takes a little more effort.

9.2.1 MATLAB Function: `lu`

MATLAB has a built-in function `lu` that generates the LU decomposition. It has the general syntax:

$$[L, U] = \text{lu}(A)$$

where L and U are the lower triangular and upper triangular matrices, respectively, derived from the LU decomposition of the matrix A . Note that this function uses partial pivoting to avoid division by zero. The following example shows how it can be employed to generate both the decomposition and a solution for the same problem that was solved in Examples 9.1 and 9.2.

EXAMPLE 9.3 9.2.1 Decomposition with MATLAB

Problem Statement. Use MATLAB to compute the LU decomposition and find the solution for the same linear system analyzed in Examples 9.1 and 9.2

$$\begin{bmatrix} 3 & 0.1 & 0.2 \\ 0.1 & 7 & 0.3 \\ 0.3 & 0.2 & 10 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 7.85 \\ 19.3 \\ 71.4 \end{Bmatrix}$$

Solution. The coefficient matrix and the right-hand-side vector can be entered in standard fashion as

```
>> A = [3 0.1 0.2; 0.1 7 0.3; 0.3 0.2 10];
>> b = [7.85; 19.3; 71.4];
```

9.3 CHOLESKY DECOMPOSITION

167

Next, the LU decomposition can be computed with

```
>> [L,U] = lu(A)

L =
    1.0000         0         0
    0.0333    1.0000         0
    0.1000   -0.0271    1.0000

U =
    3.0000   -0.1000   -0.2000
         0    7.0033   -0.2933
         0         0   10.0120
```

This is the same result that we obtained by hand in Example 9.1. We can test that it is correct by computing the original matrix as

```
>> L*U

ans =
    3.0000   -0.1000   -0.2000
    0.1000    7.0000   -0.3000
    0.3000   -0.2000   10.0000
```

To generate the solution, we first compute

```
>> d = L\b

d =
    7.8500
   -19.5617
    70.0843
```

And then use this result to compute the solution

```
>> x = U\d

x =
    3.0000
   -2.5000
    7.0000
```

These results conform to those obtained by hand in Example 9.2.

9.3 CHOLESKY DECOMPOSITION

Recall from Chap. 7 that a symmetric matrix is one where $a_{ij} = a_{ji}$ for all i and j . In other words, $[A] = [A]^T$. Such systems occur commonly in both mathematical and engineering/science problem contexts.

Special solution techniques are available for such systems. They offer computational advantages because only half the storage is needed and only half the computation time is required for their solution.

One of the most popular approaches involves *Cholesky decomposition* (also called Cholesky factorization). This algorithm is based on the fact that a symmetric matrix can be decomposed, as in

$$[A] = [U]^T [U] \quad (9.14)$$

That is, the resulting triangular factors are the transpose of each other.

The terms of Eq. (9.14) can be multiplied out and set equal to each other. The decomposition can be generated efficiently by recurrence relations. For the i th row:

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2} \quad (9.15)$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}}{u_{ii}} \quad \text{for } j = i + 1, \dots, n \quad (9.16)$$

EXAMPLE 9.4 Cholesky Decomposition

Problem Statement. Compute the Cholesky decomposition for the symmetric matrix

$$[A] = \begin{bmatrix} 6 & 15 & 55 \\ 15 & 55 & 225 \\ 55 & 225 & 979 \end{bmatrix}$$

Solution. For the first row ($i = 1$), Eq. (9.15) is employed to compute

$$u_{11} = \sqrt{a_{11}} = \sqrt{6} = 2.44949$$

Then, Eq. (9.16) can be used to determine

$$u_{12} = \frac{a_{12}}{u_{11}} = \frac{15}{2.44949} = 6.123724$$

$$u_{13} = \frac{a_{13}}{u_{11}} = \frac{55}{2.44949} = 22.45366$$

For the second row ($i = 2$):

$$u_{22} = \sqrt{a_{22} - u_{12}^2} = \sqrt{55 - (6.123724)^2} = 4.1833$$

$$u_{23} = \frac{a_{23} - u_{12}u_{13}}{u_{22}} = \frac{225 - 6.123724(22.45366)}{4.1833} = 20.9165$$

For the third row ($i = 3$):

$$u_{33} = \sqrt{a_{33} - u_{13}^2 - u_{23}^2} = \sqrt{979 - (22.45366)^2 - (20.9165)^2} = 6.110101$$

9.3 CHOLSKY DECOMPOSITION

169

Thus, the Cholesky decomposition yields

$$[U] = \begin{bmatrix} 2.44949 & 6.123724 & 22.45366 \\ & 4.1833 & 20.9165 \\ & & 6.110101 \end{bmatrix}$$

The validity of this decomposition can be verified by substituting it and its transpose into Eq. (9.14) to see if their product yields the original matrix $[A]$. This is left for an exercise.

After obtaining the decomposition, it can be used to determine a solution for a right-hand-side vector b in a manner similar to LU decomposition. First, an intermediate vector d is created by solving

$$[U]^T d = b \quad (9.17)$$

Then, the final solution can be obtained by solving

$$[U]x = d \quad (9.18)$$

9.3.1 MATLAB Function: `chol`

MATLAB has a built-in function `chol` that generates the Cholesky decomposition. It has the general syntax,

$$[U] = \text{chol}(A)$$

where $[U]$ is an upper triangular matrix so that $[U]^T [U] = A$. The following example shows how it can be employed to generate both the decomposition and a solution for the same matrix that we looked at in the previous example.

EXAMPLE 9.5 Cholesky Decomposition with MATLAB

Problem Statement. Use MATLAB to compute the Cholesky decomposition for the same matrix we analyzed in Example 9.4.

$$[A] = \begin{bmatrix} 6 & 15 & 55 \\ 15 & 55 & 225 \\ 55 & 225 & 979 \end{bmatrix}$$

Also obtain a solution for a right-hand-side vector that is the sum of the rows of $[A]$. Note that for this case, the answer will be a vector of ones.

Solution. The matrix is entered in standard fashion as

```
>> A = [6 15 55; 15 55 225; 55 225 979];
```

A right-hand-side vector that is the sum of the rows of $[A]$ can be generated as

```
>> b = [sum(A(1,:)); sum(A(2,:)); sum(A(3,:))]
```

```
b =  
    76  
   295  
  1259
```

Next, the Cholesky decomposition can be computed with

```
>> U = chol(A)

U =
    2.4495    6.1237   22.4537
         0    4.1833   20.9165
         0         0    6.1101
```

We can test that this is correct by computing the original matrix as

```
>> U'*U

ans =
    6.0000   15.0000   55.0000
   15.0000   55.0000  225.0000
   55.0000  225.0000  979.0000
```

To generate the solution, we first compute

```
>> d = A'\b

d =
   31.0269
   25.0998
    6.1101
```

And then use this result to compute the solution

```
>> x = A\y

x =
    1.0000
    1.0000
    1.0000
```

9.4 MATLAB LEFT DIVISION

We previously introduced left division without any explanation of how it works. Now that we have some background on matrix solution techniques, we can provide a simplified description of its operation.

When we implement left division with the backslash operator, MATLAB invokes a highly sophisticated algorithm to obtain a solution. In essence, MATLAB examines the structure of the coefficient matrix and then implements an optimal method to obtain the solution. Although the details of the algorithm are beyond our scope, a simplified overview can be outlined.

First, MATLAB checks to see whether $[A]$ is in a format where a solution can be obtained without full Gauss elimination. These include systems that are (a) sparse and banded, (b) triangular (or easily transformed into triangular form), or (c) symmetric. If any of these cases are detected, the solution is obtained with the efficient techniques that are available for such systems. Some of the techniques include banded solvers, back and forward substitution, and Cholesky decomposition.

PROBLEMS

171

If none of these simplified solutions are possible and the matrix is square,¹ a general triangular factorization is computed by Gauss elimination with partial pivoting and the solution obtained with substitution.

¹It should be noted that in the event that $[A]$ is not square, a least-squares solution is obtained with an approach called QR factorization.

PROBLEMS

9.1 Determine the total flops as a function of the number of equations n for the (a) decomposition, (b) forward substitution, and (c) back substitution phases of the LU decomposition version of Gauss elimination.

9.2 Use the rules of matrix multiplication to prove that Eqs. (9.7) and (9.8) follow from Eq. (9.6).

9.3 Use naive Gauss elimination to decompose the following system according to the description in Section 9.2:

$$\begin{aligned} 10x_1 + 2x_2 + x_3 &= 27 \\ 3x_1 + 6x_2 + 2x_3 &= 61.5 \\ x_1 + x_2 + 5x_3 &= 21.5 \end{aligned}$$

Then, multiply the resulting $[L]$ and $[U]$ matrices to determine that $[A]$ is produced.

9.4 Use LU decomposition to solve the system of equations in Prob. 9.3. Show all the steps in the computation. Also solve the system for an alternative right-hand-side vector

$$b^T = [12 \quad 18 \quad 6]$$

9.5 Solve the following system of equations using LU decomposition with partial pivoting:

$$\begin{aligned} 2x_1 + 6x_2 + x_3 &= 38 \\ 3x_1 + x_2 + 7x_3 &= 34 \\ 8x_1 + x_2 + 2x_3 &= 20 \end{aligned}$$

9.6 Develop your own M-file to determine the LU decomposition of a square matrix without partial pivoting. That is, develop a function that is passed the square matrix and returns the triangular matrices $[L]$ and $[U]$. Test your function by using it to solve the system in Prob. 9.3. Confirm that your function is working properly by verifying that $[L][U] = [A]$ and by using the built-in function `lu`.

9.7 Confirm the validity of the Cholesky decomposition of Example 9.4 by substituting the results into Eq. (9.14) to verify that the product of $[U]^T$ and $[U]$ yields $[A]$.

9.8 (a) Perform a Cholesky decomposition of the following symmetric system by hand:

$$\begin{bmatrix} 8 & 20 & 15 \\ 20 & 80 & 50 \\ 15 & 50 & 60 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 50 \\ 250 \\ 100 \end{bmatrix}$$

(b) Verify your hand calculation with the built-in `chol` function. (c) Employ the results of the decomposition $[U]$ to determine the solution for the right-hand-side vector.

9.9 Develop your own M-file to determine the Cholesky decomposition of a symmetric matrix without pivoting. That is, develop a function that is passed the symmetric matrix and returns the matrix $[U]$. Test your function by using it to solve the system in Prob. 9.8 and use the built-in function `chol` to confirm that your function is working properly.

Matrix Inverse and Condition

CHAPTER OBJECTIVES

The primary objective of this chapter is to show how to compute the matrix inverse and to illustrate how it can be used to analyze complex linear systems that occur in engineering and science. In addition, a method to assess a matrix solution's sensitivity to round-off error is described. Specific objectives and topics covered are

- Knowing how to determine the matrix inverse in an efficient manner based on *LU* decomposition.
- Understanding how the matrix inverse can be used to assess stimulus-response characteristics of engineering systems.
- Understanding the meaning of matrix and vector norms and how they are computed.
- Knowing how to use norms to compute the matrix condition number.
- Understanding how the magnitude of the condition number can be used to estimate the precision of solutions of linear algebraic equations.

10.1 THE MATRIX INVERSE

In our discussion of matrix operations (Section 7.2.2), we introduced the notion that if a matrix $[A]$ is square, there is another matrix $[A]^{-1}$, called the inverse of $[A]$, for which

$$[A][A]^{-1} = [A]^{-1}[A] = [I] \quad (10.1)$$

Now we will focus on how the inverse can be computed numerically. Then we will explore how it can be used for engineering analysis.

10.1.1 Calculating the Inverse

The inverse can be computed in a column-by-column fashion by generating solutions with unit vectors as the right-hand-side constants. For example, if the right-hand-side constant

10.1 THE MATRIX INVERSE

173

has a 1 in the first position and zeros elsewhere,

$$b = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix} \quad (10.2)$$

the resulting solution will be the first column of the matrix inverse. Similarly, if a unit vector with a 1 at the second row is used

$$b = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix} \quad (10.3)$$

the result will be the second column of the matrix inverse.

The best way to implement such a calculation is with LU decomposition. Recall that one of the great strengths of LU decomposition is that it provides a very efficient means to evaluate multiple right-hand-side vectors. Thus, it is ideal for evaluating the multiple unit vectors needed to compute the inverse.

EXAMPLE 10.1 Matrix Inversion

Problem Statement. Employ LU decomposition to determine the matrix inverse for the system from Example 9.1:

$$[A] = \begin{bmatrix} 3 & 0.1 & 0.2 \\ 0.1 & 7 & 0.3 \\ 0.3 & 0.2 & 10 \end{bmatrix}$$

Recall that the decomposition resulted in the following lower and upper triangular matrices:

$$[U] = \begin{bmatrix} 3 & 0.1 & 0.2 \\ 0 & 7.00333 & 0.293333 \\ 0 & 0 & 10.0120 \end{bmatrix} \quad [L] = \begin{bmatrix} 1 & 0 & 0 \\ 0.0333333 & 1 & 0 \\ 0.100000 & 0.0271300 & 1 \end{bmatrix}$$

Solution. The first column of the matrix inverse can be determined by performing the forward-substitution solution procedure with a unit vector (with 1 in the first row) as the right-hand-side vector. Thus, the lower triangular system can be set up as (recall Eq. [9.8])

$$\begin{bmatrix} 1 & 0 & 0 \\ 0.0333333 & 1 & 0 \\ 0.100000 & 0.0271300 & 1 \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix}$$

and solved with forward substitution for $d = {}^T [1 \quad 0.03333 \quad 0.1009]$. This vector can then be used as the right-hand side of the upper triangular system (recall Eq. [9.3]):

$$\begin{bmatrix} 3 & 0.1 & 0.2 \\ 0 & 7.00333 & 0.293333 \\ 0 & 0 & 10.0120 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 0.03333 \\ 0.1009 \end{Bmatrix}$$

which can be solved by back substitution for $x = {}^T [0.33249 \quad 0.00518 \quad 0.01008]$, which is the first column of the matrix inverse:

$$[A]^{-1} = \begin{bmatrix} 0.33249 & 0 & 0 \\ 0.00518 & 0 & 0 \\ 0.01008 & 0 & 0 \end{bmatrix}$$

To determine the second column, Eq. (9.8) is formulated as

$$\begin{bmatrix} 1 & 0 & 0 \\ 0.0333333 & 1 & 0 \\ 0.100000 & 0.0271300 & 1 \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix}$$

This can be solved for d , and the results are used with Eq. (9.3) to determine $x^T = [0.004944 \quad 0.142903 \quad 0.00271]$, which is the second column of the matrix inverse:

$$[A]^{-1} = \begin{bmatrix} 0.33249 & 0.004944 & 0 \\ 0.00518 & 0.142903 & 0 \\ 0.01008 & 0.002710 & 0 \end{bmatrix}$$

Finally, the same procedures can be implemented with $b^T = [0 \quad 0 \quad 1]$ to solve for $x^T = [0.006798 \quad 0.004183 \quad 0.09988]$, which is the final column of the matrix inverse:

$$[A]^{-1} = \begin{bmatrix} 0.33249 & 0.004944 & 0.006798 \\ 0.00518 & 0.142903 & 0.004183 \\ 0.01008 & 0.002710 & 0.099880 \end{bmatrix}$$

The validity of this result can be checked by verifying that $[A][A]^{-1} = [I]$.

10.1.2 Stimulus-Response Computations

As discussed in Section 7.1.1, many of the linear systems of equations arising in engineering and science are derived from conservation laws. The mathematical expression of these laws is some form of balance equation to ensure that a particular property—mass, force, heat, momentum, electrostatic potential—is conserved. For a force balance on a structure, the properties might be horizontal or vertical components of the forces acting on each node of the structure. For a mass balance, the properties might be the mass in each reactor of a chemical process. Other fields of engineering and science would yield similar examples.

A single balance equation can be written for each part of the system, resulting in a set of equations defining the behavior of the property for the entire system. These equations are interrelated, or coupled, in that each equation may include one or more of the variables from the other equations. For many cases, these systems are linear and, therefore, of the exact form dealt with in this chapter:

$$[A]x = b \quad (10.4)$$

Now, for balance equations, the terms of Eq. (10.4) have a definite physical interpretation. For example, the elements of x are the levels of the property being balanced for each part of the system. In a force balance of a structure, they represent the horizontal and vertical forces in each member. For the mass balance, they are the mass of chemical in each reactor. In either case, they represent the system's *state* or *response*, which we are trying to determine.

The right-hand-side vector b contains those elements of the balance that are independent of behavior of the system—that is, they are constants. In many problems, they represent the *forcing functions* or *external stimuli* that drive the system.

10.1 THE MATRIX INVERSE

175

Finally, the matrix of coefficients $[A]$ usually contains the *parameters* that express how the parts of the system *interact* or are coupled. Consequently, Eq. (10.4) might be reexpressed as

$$[\text{Interactions}] \cdot \text{response} = [\text{stimuli}]$$

As we know from previous chapters, there are a variety of ways to solve Eq. (10.4). However, using the matrix inverse yields a particularly interesting result. The formal solution can be expressed as

$$x = [A]^{-1} \cdot b$$

or (recalling our definition of matrix multiplication from Section 7.2.2)

$$x_1 = a_{11}^{-1}b_1 + a_{12}^{-1}b_2 + a_{13}^{-1}b_3$$

$$x_2 = a_{21}^{-1}b_1 + a_{22}^{-1}b_2 + a_{23}^{-1}b_3$$

$$x_3 = a_{31}^{-1}b_1 + a_{32}^{-1}b_2 + a_{33}^{-1}b_3$$

Thus, we find that the inverted matrix itself, aside from providing a solution, has extremely useful properties. That is, each of its elements represents the response of a single part of the system to a unit stimulus of any other part of the system.

Notice that these formulations are linear and, therefore, superposition and proportionality hold. *Superposition* means that if a system is subject to several different stimuli (the b 's), the responses can be computed individually and the results summed to obtain a total response. *Proportionality* means that multiplying the stimuli by a quantity results in the response to those stimuli being multiplied by the same quantity. Thus, the coefficient a_{11}^{-1} is a proportionality constant that gives the value of x_1 due to a unit level of b_1 . This result is independent of the effects of b_2 and b_3 on x_1 , which are reflected in the coefficients a_{12}^{-1} and a_{13}^{-1} , respectively. Therefore, we can draw the general conclusion that the element a_{ij}^{-1} of the inverted matrix represents the value of x_i due to a unit quantity of b_j .

Using the example of the structure, element a_{ij}^{-1} of the matrix inverse would represent the force in member i due to a unit external force at node j . Even for small systems, such behavior of individual stimulus-response interactions would not be intuitively obvious. As such, the matrix inverse provides a powerful technique for understanding the interrelationships of component parts of complicated systems.

EXAMPLE 10.2 Analyzing the Bungee Jumper Problem

Problem Statement. At the beginning of Chap. 7, we set up a problem involving three individuals suspended vertically connected by bungee cords. We derived a system of linear algebraic equations based on force balances for each jumper,

$$\begin{bmatrix} 150 & -100 & 0 \\ -100 & 150 & -50 \\ 0 & -50 & 50 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 588.6 \\ 686.7 \\ 784.8 \end{Bmatrix}$$

In Example 7.2, we used MATLAB to solve this system for the vertical positions of the jumpers (the x 's). In the present example, use MATLAB to compute the matrix inverse and interpret what it means.

Solution. Start up MATLAB and enter the coefficient matrix:

```
>> K = [150 -100 0; -100 150 -50; 0 -50 50];
```

The inverse can then be computed as

```
>> KI = inv(K)
```

```
KI =  
    0.0200    0.0200    0.0200  
    0.0200    0.0300    0.0300  
    0.0200    0.0300    0.0500
```

Each element of the inverse, k_{ij}^{-1} of the inverted matrix represents the vertical change in position (in meters) of jumper i due to a unit change in force (in Newtons) applied to jumper j .

First, observe that the numbers in the first column ($j = 1$) indicate that the position of all three jumpers would increase by 0.02 m if the force on the first jumper was increased by 1 N. This makes sense, because the additional force would only elongate the first cord by that amount.

In contrast, the numbers in the second column ($j = 2$) indicate that applying a force of 1 N to the second jumper would move the first jumper down by 0.02 m, but the second and third by 0.03 m. The 0.02-m elongation of the first jumper makes sense because the first cord is subject to an extra 1 N regardless of whether the force is applied to the first or second jumper. However, for the second jumper the elongation is now 0.03 m because along with the first cord, the second cord also elongates due to the additional force. And of course, the third jumper shows the identical translation as the second jumper as there is no additional force on the third cord that connects them.

As expected, the third column ($j = 3$) indicates that applying a force of 1 N to the third jumper results in the first and second jumpers moving the same distances as occurred when the force was applied to the second jumper. However, now because of the additional elongation of the third cord, the third jumper is moved farther downward.

Superposition and proportionality can be demonstrated by using the inverse to determine how much farther the third jumper would move downward if additional forces of 10, 50, and 20 N were applied to the first, second, and third jumpers, respectively. This can be done simply by using the appropriate elements of the third row of the inverse to compute,

$$\Delta x_3 = k_{31}^{-1} \Delta F_1 + k_{32}^{-1} \Delta F_2 + k_{33}^{-1} \Delta F_3 = 0.02(10) + 0.03(50) + 0.05(20) = 2.7 \text{ m}$$

10.2 ERROR ANALYSIS AND SYSTEM CONDITION

Aside from its engineering and scientific applications, the inverse also provides a means to discern whether systems are ill-conditioned. Three direct methods can be devised for this purpose:

1. Scale the matrix of coefficients $[A]$ so that the largest element in each row is 1. Invert the scaled matrix and if there are elements of $[A]^{-1}$ that are several orders of magnitude greater than one, it is likely that the system is ill-conditioned.

2. Multiply the inverse by the original coefficient matrix and assess whether the result is close to the identity matrix. If not, it indicates ill-conditioning.
3. Invert the inverted matrix and assess whether the result is sufficiently close to the original coefficient matrix. If not, it again indicates that the system is ill-conditioned.

Although these methods can indicate ill-conditioning, it would be preferable to obtain a single number that could serve as an indicator of the problem. Attempts to formulate such a matrix condition number are based on the mathematical concept of the norm.

10.2.1 Vector and Matrix Norms

A *norm* is a real-valued function that provides a measure of the size or “length” of multi-component mathematical entities such as vectors and matrices.

A simple example is a vector in three-dimensional Euclidean space (Fig. 10.1) that can be represented as

$$F = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

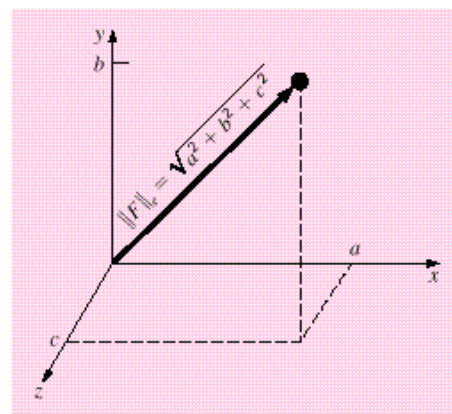
where a , b , and c are the distances along the x , y , and z axes, respectively. The length of this vector—that is, the distance from the coordinate $(0, 0, 0)$ to (a, b, c) —can be simply computed as

$$\|F\|_e = \sqrt{a^2 + b^2 + c^2}$$

where the nomenclature $\|F\|_e$ indicates that this length is referred to as the *Euclidean norm* of $[F]$.

FIGURE 10.1

Graphical depiction of a vector in Euclidean space.



Similarly, for an n -dimensional vector $X = [x_1 \ x_2 \ \cdots \ x_n]^T$, a Euclidean norm would be computed as

$$\|X\|_e = \sqrt{\sum_{i=1}^n x_i^2}$$

The concept can be extended further to a matrix $[A]$, as in

$$\|A\|_e = \sqrt{\sum_{i=1}^n \sum_{j=1}^n a_{i,j}^2} \quad (10.5)$$

which is given a special name—the *Frobenius norm*. As with the other vector norms, it provides a single value to quantify the “size” of $[A]$.

It should be noted that there are alternatives to the Euclidean and Frobenius norms. For vectors, there are alternatives called p norms that can be represented generally by

$$\|X\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}$$

We can see that the Euclidean norm and the 2 norm, $\|X\|_2$, are identical for vectors.

Other important examples are ($p = 1$)

$$\|X\|_1 = \sum_{i=1}^n |x_i|$$

which represents the norm as the sum of the absolute values of the elements. Another is the maximum-magnitude or uniform-vector norm ($p = \infty$),

$$\|X\|_\infty = \max_{1 \leq i \leq n} |x_i|$$

which defines the norm as the element with the largest absolute value.

Using a similar approach, norms can be developed for matrices. For example,

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^n |a_{ij}|$$

That is, a summation of the absolute values of the coefficients is performed for each column, and the largest of these summations is taken as the norm. This is called the *column-sum norm*.

A similar determination can be made for the rows, resulting in a uniform-matrix or *row-sum norm*:

$$\|A\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$$

It should be noted that, in contrast to vectors, the 2 norm and the Euclidean norm for a matrix are not the same. Whereas the Euclidean norm $\|A\|_e$ can be easily determined by Eq. (10.5), the matrix 2 norm $\|A\|_2$ is calculated as

$$\|A\|_2 = (\mu_{\max})^{1/2}$$

where μ_{\max} is the largest eigenvalue of $[A]^T[A]$. In a later chapter, we will learn more about eigenvalues. For the time being, the important point is that the $\|A\|_2$, or *spectral norm*, is the minimum norm and, therefore, provides the tightest measure of size (Ortega, 1972).

10.2.2 Matrix Condition Number

Now that we have introduced the concept of the norm, we can use it to define

$$\text{Cond}[A] = \|A\| \|A^{-1}\|$$

where $\text{Cond}[A]$ is called the *matrix condition number*. Note that for a matrix $[A]$, this number will be greater than or equal to 1. It can be shown (Ralston and Rabinowitz, 1978; Gerald and Wheatley, 1989) that

$$\frac{\| \Delta X \|}{\| X \|} = \text{Cond}[A] \frac{\| \Delta A \|}{\| A \|}$$

That is, the relative error of the norm of the computed solution can be as large as the relative error of the norm of the coefficients of $[A]$ multiplied by the condition number. For example, if the coefficients of $[A]$ are known to t -digit precision (i.e., rounding errors are on the order of 10^{-t}) and $\text{Cond}[A] = 10^c$, the solution $[X]$ may be valid to only $t - c$ digits (rounding errors $\sim 10^{c-t}$).

EXAMPLE 10.3 Matrix Condition Evaluation

Problem Statement. The Hilbert matrix, which is notoriously ill-conditioned, can be represented generally as

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n+1} \end{bmatrix}$$

Use the row-sum norm to estimate the matrix condition number for the 3×3 Hilbert matrix:

$$[A] = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

Solution. First, the matrix can be normalized so that the maximum element in each row is 1:

$$[A] = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 1 & \frac{2}{3} & \frac{1}{2} \\ 1 & \frac{3}{4} & \frac{3}{5} \end{bmatrix}$$

Summing each of the rows gives 1.833, 2.1667, and 2.35. Thus, the third row has the largest sum and the row-sum norm is

$$\|A\|_{\infty} = 1 + \frac{3}{4} + \frac{3}{5} = 2.35$$

The inverse of the scaled matrix can be computed as

$$[A]^{-1} = \begin{bmatrix} 9 & -18 & 10 \\ -36 & 96 & -60 \\ 30 & -90 & 60 \end{bmatrix}$$

Note that the elements of this matrix are larger than the original matrix. This is also reflected in its row-sum norm, which is computed as

$$\|A^{-1}\|_{\infty} = \max(36, 96, 60) = 96$$

Thus, the condition number can be calculated as

$$\text{Cond}[A] = 2.35(96) = 451.2$$

The fact that the condition number is much greater than unity suggests that the system is ill-conditioned. The extent of the ill-conditioning can be quantified by calculating $c = \log 451.2 = 2.65$. Hence, the last three significant digits of the solution could exhibit rounding errors. Note that such estimates almost always overpredict the actual error. However, they are useful in alerting you to the possibility that round-off errors may be significant.

10.2.3 Norms and Condition Number in MATLAB

MATLAB has built-in functions to compute both norms and condition numbers:

```
>> norm(x, p)
```

and

```
>> cond(x, p)
```

where x is the vector or matrix and p designates the type of norm or condition number (1, 2, inf, or 'fro'). Note that the `cond` function is equivalent to

```
>> norm(x, p) * norm(inv(x), p)
```

Also, note that if p is omitted, it is automatically set to 2.

EXAMPLE 10.4 Matrix Condition Evaluation with MATLAB

Problem Statement. Use MATLAB to evaluate both the norms and condition numbers for the scaled Hilbert matrix previously analyzed in Example 10.3:

$$[A] = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 1 & \frac{2}{3} & \frac{1}{2} \\ 1 & \frac{3}{4} & \frac{3}{5} \end{bmatrix}$$

(a) As in Example 10.3, first compute the row-sum versions ($p = \text{inf}$). (b) Also compute the Frobenius ($p = \text{'fro'}$) and the spectral ($p = 2$) condition numbers.

Solution: (a) First, enter the matrix:

```
>> A = [1 1/2 1/3; 1 2/3 1/2; 1 3/4 3/5];
```

PROBLEMS

181

Then, the row-sum norm and condition number can be computed as

```
>> norm(A,inf)

ans =
    2.3500

>> cond(A,inf)

ans =
   451.2000
```

These results correspond to those that were calculated by hand in Example 10.3.

(b) The condition number based on the Frobenius and spectral norms are

```
>> cond(A,'fro')

ans =
   368.0866

>> cond(A)

ans =
   366.3503
```

PROBLEMS

10.1 Determine the matrix inverse for the following system:

$$\begin{aligned} 10x_1 + 2x_2 + x_3 &= 27 \\ 3x_1 + 6x_2 + 2x_3 &= 61.5 \\ x_1 + x_2 + 5x_3 &= 21.5 \end{aligned}$$

Check your results by verifying that $[A][A]^{-1} = [I]$. Do not use a pivoting strategy.

10.2 Determine the matrix inverse for the following system:

$$\begin{aligned} 8x_1 + x_2 + 2x_3 &= 20 \\ 2x_1 + 6x_2 + x_3 &= 38 \\ 3x_1 + x_2 + 7x_3 &= 34 \end{aligned}$$

10.3 The following system of equations is designed to determine concentrations (the c 's in g/m^3) in a series of coupled reactors as a function of the amount of mass input to each reactor (the right-hand sides in g/day):

$$\begin{aligned} 15c_1 + 3c_2 + c_3 &= 3800 \\ 3c_1 + 18c_2 + 6c_3 &= 1200 \\ 4c_1 + c_2 + 12c_3 &= 2350 \end{aligned}$$

- (a) Determine the matrix inverse.
(b) Use the inverse to determine the solution.

(c) Determine how much the rate of mass input to reactor 3 must be increased to induce a 10 g/m^3 rise in the concentration of reactor 1.

(d) How much will the concentration in reactor 3 be reduced if the rate of mass input to reactors 1 and 2 is reduced by 500 and 250 g/day , respectively?

10.4 Determine the matrix inverse for the system described in Prob. 7.6. Use the matrix inverse to determine the concentration in reactor 5 if the inflow concentrations are changed to $c_{01} = 20$ and $c_{03} = 50$.

10.5 Determine the matrix inverse for the system described in Prob. 7.7. Use the matrix inverse to determine the force in the three members (F_1 , F_2 and F_3) if the vertical load at node 1 is doubled to $F_{1,v} = 2000 \text{ lb}$ and a horizontal load of $F_{3,h} = 500 \text{ lb}$ is applied to node 3.

10.6 Determine $A^{-1}_{:,e}$, $A^{-1}_{:,1}$, and $A^{-1}_{:,}$ for

$$[A] = \begin{bmatrix} 8 & 2 & 10 \\ 9 & 1 & 3 \\ 15 & 1 & 6 \end{bmatrix}$$

Before determining the norms, scale the matrix by making the maximum element in each row equal to one.

10.7 Determine the Frobenius and row-sum norms for the systems in Probs. 10.2 and 10.3.

10.8 Use MATLAB to determine the spectral condition number for the following system. Do not normalize the system:

$$\begin{bmatrix} 1 & 4 & 9 & 16 & 25 \\ 4 & 9 & 16 & 25 & 36 \\ 9 & 16 & 25 & 36 & 49 \\ 16 & 25 & 36 & 49 & 64 \\ 25 & 36 & 49 & 64 & 81 \end{bmatrix}$$

Compute the condition number based on the row-sum norm.

10.9 Besides the Hilbert matrix, there are other matrices that are inherently ill-conditioned. One such case is the *Vandermonde matrix*, which has the following form:

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix}$$

(a) Determine the condition number based on the row-sum norm for the case where $x_1 = 4$, $x_2 = 2$, and $x_3 = 7$.

(b) Use MATLAB to compute the spectral and Frobenius condition numbers.

10.10 Use MATLAB to determine the spectral condition number for a 10-dimensional Hilbert matrix. How many digits of precision are expected to be lost due to ill-conditioning? Determine the solution for this system for the case where each element of the right-hand-side vector b consists of the summation of the coefficients in its row. In other words, solve for the case where all the unknowns should be exactly one. Compare the resulting errors with those expected based on the condition number.

10.11 Repeat Prob. 10.10, but for the case of a six-dimensional Vandermonde matrix (see Prob. 10.9) where $x_1 = 4$, $x_2 = 2$, $x_3 = 7$, $x_4 = 10$, $x_5 = 3$, and $x_6 = 5$.

Iterative Methods for Systems of Equations

CHAPTER OBJECTIVES

The primary objective of this chapter is to acquaint you with iterative methods for solving simultaneous equations. Specific objectives and topics covered are

- Understanding the difference between the Gauss-Seidel and Jacobi methods.
- Knowing how to assess diagonal dominance and knowing what it means.
- Recognizing how relaxation can be used to improve the convergence of iterative methods.
- Understanding how to solve systems of nonlinear equations with successive substitution and Newton-Raphson.

Iterative or approximate methods provide an alternative to the elimination methods described to this point. Such approaches are similar to the techniques we developed to obtain the roots of a single equation in Chaps. 5 and 6. Those approaches consisted of guessing a value and then using a systematic method to obtain a refined estimate of the root. Because the present part of the book deals with a similar problem—obtaining the values that simultaneously satisfy a set of equations—we might suspect that such approximate methods could be useful in this context. In this chapter, we will present approaches for solving both linear and nonlinear simultaneous equations.

11.1 LINEAR SYSTEMS: GAUSS-SEIDEL

The *Gauss-Seidel method* is the most commonly used iterative method for solving linear algebraic equations. Assume that we are given a set of n equations:

$$[A]\{x\} = \{b\}$$

Suppose that for conciseness we limit ourselves to a 3×3 set of equations. If the diagonal elements are all nonzero, the first equation can be solved for x_1 , the second for x_2 , and the

third for x_3 to yield

$$x_1^j = \frac{b_1 - a_{12}x_2^{j-1} - a_{13}x_3^{j-1}}{a_{11}} \quad (11.1a)$$

$$x_2^j = \frac{b_2 - a_{21}x_1^j - a_{23}x_3^{j-1}}{a_{22}} \quad (11.1b)$$

$$x_3^j = \frac{b_3 - a_{31}x_1^j - a_{32}x_2^j}{a_{33}} \quad (11.1c)$$

where j and $j - 1$ are the present and previous iterations.

To start the solution process, initial guesses must be made for the x 's. A simple approach is to assume that they are all zero. These zeros can be substituted into Eq. (11.1a), which can be used to calculate a new value for $x_1 = b_1/a_{11}$. Then we substitute this new value of x_1 along with the previous guess of zero for x_3 into Eq. (11.1b) to compute a new value for x_2 . The process is repeated for Eq. (11.1c) to calculate a new estimate for x_3 . Then we return to the first equation and repeat the entire procedure until our solution converges closely enough to the true values. Convergence can be checked using the criterion that for all i ,

$$\varepsilon_{a,i} = \left| \frac{x_i^j - x_i^{j-1}}{x_i^j} \right| \times 100\% \leq \varepsilon_s \quad (11.2)$$

EXAMPLE 11.1 Gauss-Seidel Method

Problem Statement. Use the Gauss-Seidel method to obtain the solution for

$$3x_1 - 0.1x_2 - 0.2x_3 = 7.85$$

$$0.1x_1 + 7x_2 - 0.3x_3 = -19.3$$

$$0.3x_1 - 0.2x_2 + 10x_3 = 71.4$$

Note that the solution is $\{x\}^T = [3 \quad -2.5 \quad 7]$.

Solution. First, solve each of the equations for its unknown on the diagonal:

$$x_1 = \frac{7.85 + 0.1x_2 + 0.2x_3}{3} \quad (E11.1.1)$$

$$x_2 = \frac{-19.3 - 0.1x_1 + 0.3x_3}{7} \quad (E11.1.2)$$

$$x_3 = \frac{71.4 - 0.3x_1 + 0.2x_2}{10} \quad (E11.1.3)$$

By assuming that x_2 and x_3 are zero, Eq. (E11.1.1) can be used to compute

$$x_1 = \frac{7.85 + 0.1(0) + 0.2(0)}{3} = 2.616667$$

This value, along with the assumed value of $x_3 = 0$, can be substituted into Eq. (E11.1.2) to calculate

$$x_2 = \frac{-19.3 - 0.1(2.616667) + 0.3(0)}{7} = -2.794524$$

The first iteration is completed by substituting the calculated values for x_1 and x_2 into Eq. (E11.1.3) to yield

$$x_3 = \frac{71.4 - 0.3(2.616667) + 0.2(-2.794524)}{10} = 7.005610$$

For the second iteration, the same process is repeated to compute

$$x_1 = \frac{7.85 + 0.1(-2.794524) + 0.2(7.005610)}{3} = 2.990557$$

$$x_2 = \frac{-19.3 - 0.1(2.990557) + 0.3(7.005610)}{7} = -2.499625$$

$$x_3 = \frac{71.4 - 0.3(2.990557) + 0.2(-2.499625)}{10} = 7.000291$$

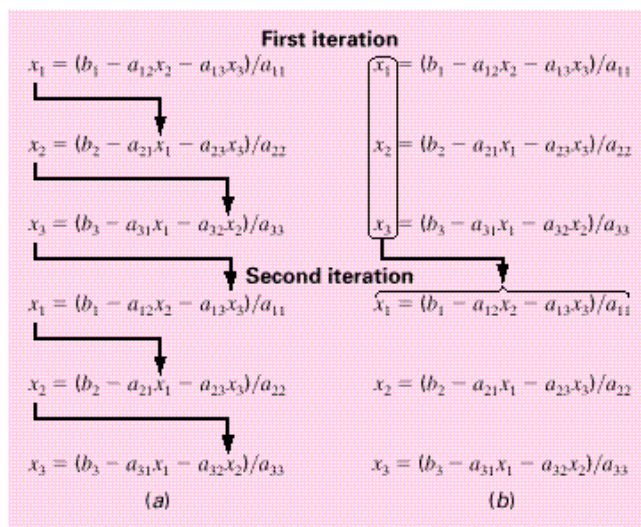
The method is, therefore, converging on the true solution. Additional iterations could be applied to improve the answers. However, in an actual problem, we would not know the true answer *a priori*. Consequently, Eq. (11.2) provides a means to estimate the error. For example, for x_1 :

$$\varepsilon_{a,1} = \left| \frac{2.990557 - 2.616667}{2.990557} \right| \times 100\% = 12.5\%$$

For x_2 and x_3 , the error estimates are $\varepsilon_{a,2} = 11.8\%$ and $\varepsilon_{a,3} = 0.076\%$. Note that, as was the case when determining roots of a single equation, formulations such as Eq. (11.2) usually provide a conservative appraisal of convergence. Thus, when they are met, they ensure that the result is known to at least the tolerance specified by ε_s .

As each new x value is computed for the Gauss-Seidel method, it is immediately used in the next equation to determine another x value. Thus, if the solution is converging, the best available estimates will be employed. An alternative approach, called *Jacobi iteration*, utilizes a somewhat different tactic. Rather than using the latest available x 's, this technique uses Eq. (11.1) to compute a set of new x 's on the basis of a set of old x 's. Thus, as new values are generated, they are not immediately used but rather are retained for the next iteration.

The difference between the Gauss-Seidel method and Jacobi iteration is depicted in Fig. 11.1. Although there are certain cases where the Jacobi method is useful, Gauss-Seidel's utilization of the best available estimates usually makes it the method of preference.

**FIGURE 11.1**

Graphical depiction of the difference between (a) the Gauss-Seidel and (b) the Jacobi iterative methods for solving simultaneous linear algebraic equations.

11.1.1 Convergence and Diagonal Dominance

Note that the Gauss-Seidel method is similar in spirit to the technique of simple fixed-point iteration that was used in Section 6.1 to solve for the roots of a single equation. Recall that simple fixed-point iteration was sometimes nonconvergent. That is, as the iterations progressed, the answer moved farther and farther from the correct result.

Although the Gauss-Seidel method can also diverge, because it is designed for linear systems, its ability to converge is much more predictable than for fixed-point iteration of nonlinear equations. It can be shown that if the following condition holds, Gauss-Seidel will converge:

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \quad (11.3)$$

That is, the absolute value of the diagonal coefficient in each of the equations must be larger than the sum of the absolute values of the other coefficients in the equation. Such systems are said to be *diagonally dominant*. This criterion is sufficient but not necessary for convergence. That is, although the method may sometimes work if Eq. (11.3) is not met, convergence is guaranteed if the condition is satisfied. Fortunately, many engineering and scientific problems of practical importance fulfill this requirement. Therefore, Gauss-Seidel represents a feasible approach to solve many problems in engineering and science.

11.1.2 MATLAB M-file:

Before developing an algorithm, let us first recast Gauss-Seidel in a form that is compatible with MATLAB's ability to perform matrix operations. This is done by expressing Eq. (11.1) as

$$\begin{aligned}x_1^{\text{new}} &= \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}}x_2^{\text{old}} - \frac{a_{13}}{a_{11}}x_3^{\text{old}} \\x_2^{\text{new}} &= \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1^{\text{new}} - \frac{a_{23}}{a_{22}}x_3^{\text{old}} \\x_3^{\text{new}} &= \frac{b_3}{a_{33}} - \frac{a_{31}}{a_{33}}x_1^{\text{new}} - \frac{a_{32}}{a_{33}}x_2^{\text{new}}\end{aligned}$$

Notice that the solution can be expressed concisely in matrix form as

$$\{x\} = \{d\} - [C]\{x\} \quad (11.4)$$

where

$$\{d\} = \begin{Bmatrix} b_1/a_{11} \\ b_2/a_{22} \\ b_3/a_{33} \end{Bmatrix}$$

and

$$[C] = \begin{bmatrix} 0 & a_{12}/a_{11} & a_{13}/a_{11} \\ a_{21}/a_{22} & 0 & a_{23}/a_{22} \\ a_{31}/a_{33} & a_{32}/a_{33} & 0 \end{bmatrix}$$

An M-file to implement Eq. (11.4) is listed in Fig. 11.2.

11.1.3 Relaxation

Relaxation represents a slight modification of the Gauss-Seidel method that is designed to enhance convergence. After each new value of x is computed using Eq. (11.1), that value is modified by a weighted average of the results of the previous and the present iterations:

$$x_i^{\text{new}} = \lambda x_i^{\text{new}} + (1 - \lambda)x_i^{\text{old}} \quad (11.5)$$

where λ is a weighting factor that is assigned a value between 0 and 2.

If $\lambda = 1$, $(1 - \lambda)$ is equal to 0 and the result is unmodified. However, if λ is set at a value between 0 and 1, the result is a weighted average of the present and the previous results. This type of modification is called *underrelaxation*. It is typically employed to make a nonconvergent system converge or to hasten convergence by dampening out oscillations.

For values of λ from 1 to 2, extra weight is placed on the present value. In this instance, there is an implicit assumption that the new value is moving in the correct direction toward the true solution but at too slow a rate. Thus, the added weight of λ is intended to improve the estimate by pushing it closer to the truth. Hence, this type of modification, which is called *overrelaxation*, is designed to accelerate the convergence of an already convergent system. The approach is also called *successive overrelaxation*, or SOR.


```
function x = GaussSeidel(A,b,es,maxit)
% GaussSeidel(A,b):
%   Gauss Seidel method.
% input:
%   A = coefficient matrix
%   b = right hand side vector
%   es = (optional) stop criterion (%) (default = 0.00001)
%   maxit = (optional) max iterations (default = 50)
% output:
%   x = solution vector

% default values
if nargin<4, maxit=50; end
if nargin<3, es=0.00001; end
[m,n] = size(A);
if m~=n, error('Matrix A must be square'); end
C = A;
for i = 1:n
    C(i,i) = 0;
    x(i) = 0;
end
x = x';
for i = 1:n
    C(i,1:n) = C(i,1:n)/A(i,i);
end
for i = 1:n
    d(i) = b(i)/A(i,i);
end
iter = 0;
while (1)
    xold = x;
    for i = 1:n
        x(i) = d(i)-C(i,:)*x;
        if x(i) ~= 0
            ea(i) = abs((x(i) - xold(i))/x(i)) * 100;
        end
    end
    iter = iter+1;
    if max(ea)<=es | iter >= maxit, break, end
end
```

FIGURE 11.2

MATLAB Mfile to implement Gauss-Seidel.

The choice of a proper value for λ is highly problem-specific and is often determined empirically. For a single solution of a set of equations it is often unnecessary. However, if the system under study is to be solved repeatedly, the efficiency introduced by a wise choice of λ can be extremely important. Good examples are the very large systems of linear algebraic equations that can occur when solving partial differential equations in a variety of engineering and scientific problem contexts.

11.2 NONLINEAR SYSTEMS

The following is a set of two simultaneous nonlinear equations with two unknowns:

$$x_1^2 + x_1x_2 = 10 \quad (11.6a)$$

$$x_2 + 3x_1x_2^2 = 57 \quad (11.6b)$$

In contrast to linear systems which plot as straight lines (recall Fig. 8.1), these equations plot as curves on an x_2 versus x_1 graph. As in Fig. 11.3, the solution is the intersection of the curves.

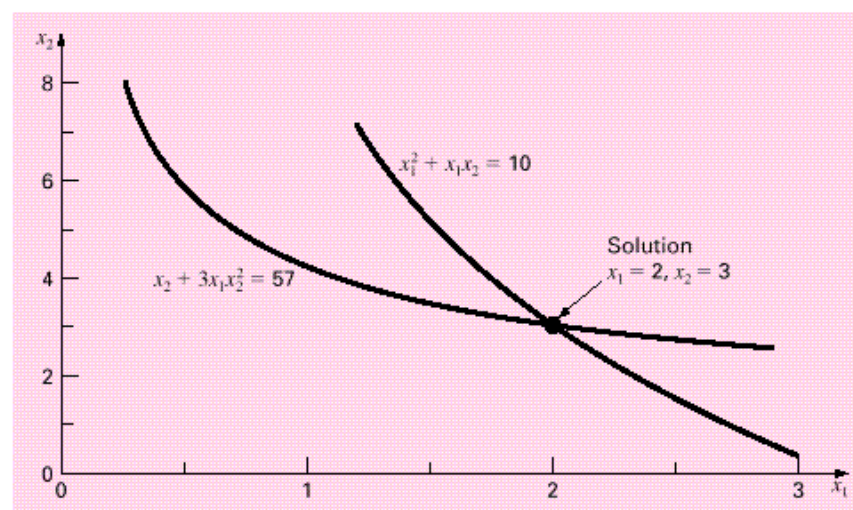
Just as we did when we determined roots for single nonlinear equations, such systems of equations can be expressed generally as

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= 0 \end{aligned} \quad (11.7)$$

Therefore, the solution are the values of the x 's that make the equations equal to zero.

FIGURE 11.3

Graphical depiction of the solution of two simultaneous nonlinear equations.



11.2.1 Successive Substitution

A simple approach for solving Eq. (11.7) is to use the same strategy that was employed for fixed-point iteration and the Gauss-Seidel method. That is, each one of the nonlinear equations can be solved for one of the unknowns. These equations can then be implemented iteratively to compute new values which (hopefully) will converge on the solutions. This approach, which is called *successive substitution*, is illustrated in the following example.

EXAMPLE 11.2 Successive Substitution for a Nonlinear System

Problem Statement. Use successive substitution to determine the roots of Eq. (11.6). Note that a correct pair of roots is $x_1 = 2$ and $x_2 = 3$. Initiate the computation with guesses of $x_1 = 1.5$ and $x_2 = 3.5$.

Solution. Equation (11.6a) can be solved for

$$x_1 = \frac{10 - x_1^2}{x_2} \quad (\text{E11.2.1})$$

and Eq. (11.6b) can be solved for

$$x_2 = 57 - 3x_1x_2^2 \quad (\text{E11.2.2})$$

On the basis of the initial guesses, Eq. (E11.2.1) can be used to determine a new value of x_1 :

$$x_1 = \frac{10 - (1.5)^2}{3.5} = 2.21429$$

This result and the initial value of $x_2 = 3.5$ can be substituted into Eq. (E11.2.2) to determine a new value of x_2 :

$$x_2 = 57 - 3(2.21429)(3.5)^2 = -24.37516$$

Thus, the approach seems to be diverging. This behavior is even more pronounced on the second iteration:

$$x_1 = \frac{10 - (2.21429)^2}{-24.37516} = -0.20910$$

$$x_2 = 57 - 3(-0.20910)(-24.37516)^2 = 429.709$$

Obviously, the approach is deteriorating.

Now we will repeat the computation but with the original equations set up in a different format. For example, an alternative solution of Eq. (11.6a) is

$$x_1 = \sqrt{10 - x_1x_2}$$

and of Eq. (11.6b) is

$$x_2 = \sqrt{\frac{57 - x_2}{3x_1}}$$

11.2 NONLINEAR SYSTEMS

191

Now the results are more satisfactory:

$$x_1 = \sqrt{10 - 1.5(3.5)} = 2.17945$$

$$x_2 = \sqrt{\frac{57 - 3.5}{3(2.17945)}} = 2.86051$$

$$x_1 = \sqrt{10 - 2.17945(2.86051)} = 1.94053$$

$$x_2 = \sqrt{\frac{57 - 2.86051}{3(1.94053)}} = 3.04955$$

Thus, the approach is converging on the true values of $x_1 = 2$ and $x_2 = 3$.

The previous example illustrates the most serious shortcoming of successive substitution—that is, convergence often depends on the manner in which the equations are formulated. Additionally, even in those instances where convergence is possible, divergence can occur if the initial guesses are insufficiently close to the true solution. These criteria are so restrictive that fixed-point iteration has limited utility for solving nonlinear systems.

11.2.2 Newton-Raphson

Just as fixed-point iteration can be used to solve systems of nonlinear equations, other open root location methods such as the Newton-Raphson method can be used for the same purpose. Recall that the Newton-Raphson method was predicated on employing the derivative (i.e., the slope) of a function to estimate its intercept with the axis of the independent variable—that is, the root. In Chap. 6, we used a graphical derivation to compute this estimate. An alternative is to derive it from a first-order Taylor series expansion:

$$f(x_{i+1}) = f(x_i) + (x_{i+1} - x_i)f'(x_i) \quad (11.8)$$

where x_i is the initial guess at the root and x_{i+1} is the point at which the slope intercepts the x axis. At this intercept, $f(x_{i+1})$ by definition equals zero and Eq. (11.8) can be rearranged to yield

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (11.9)$$

which is the single-equation form of the Newton-Raphson method.

The multiequation form is derived in an identical fashion. However, a multivariable Taylor series must be used to account for the fact that more than one independent variable contributes to the determination of the root. For the two-variable case, a first-order Taylor series can be written for each nonlinear equation as

$$f_{1,i+1} = f_{1,i} + (x_{1,i+1} - x_{1,i})\frac{\partial f_{1,i}}{\partial x_1} + (x_{2,i+1} - x_{2,i})\frac{\partial f_{1,i}}{\partial x_2} \quad (11.10a)$$

$$f_{2,i+1} = f_{2,i} + (x_{1,i+1} - x_{1,i})\frac{\partial f_{2,i}}{\partial x_1} + (x_{2,i+1} - x_{2,i})\frac{\partial f_{2,i}}{\partial x_2} \quad (11.10b)$$

Just as for the single-equation version, the root estimate corresponds to the values of x_1 and x_2 , where $f_{1,i+1}$ and $f_{2,i+1}$ equal zero. For this situation, Eq. (11.10) can be rearranged to give

$$\frac{\partial f_{1,i}}{\partial x_1} x_{1,i+1} + \frac{\partial f_{1,i}}{\partial x_2} x_{2,i+1} = -f_{1,i} + x_{1,i} \frac{\partial f_{1,i}}{\partial x_1} + x_{2,i} \frac{\partial f_{1,i}}{\partial x_2} \quad (11.11a)$$

$$\frac{\partial f_{2,i}}{\partial x_1} x_{1,i+1} + \frac{\partial f_{2,i}}{\partial x_2} x_{2,i+1} = -f_{2,i} + x_{1,i} \frac{\partial f_{2,i}}{\partial x_1} + x_{2,i} \frac{\partial f_{2,i}}{\partial x_2} \quad (11.11b)$$

Because all values subscripted with i 's are known (they correspond to the latest guess or approximation), the only unknowns are $x_{1,i+1}$ and $x_{2,i+1}$. Thus, Eq. (11.11) is a set of two linear equations with two unknowns. Consequently, algebraic manipulations (e.g., Cramer's rule) can be employed to solve for

$$x_{1,i+1} = x_{1,i} - \frac{f_{1,i} \frac{\partial f_{2,i}}{\partial x_2} - f_{2,i} \frac{\partial f_{1,i}}{\partial x_2}}{\frac{\partial f_{1,i}}{\partial x_1} \frac{\partial f_{2,i}}{\partial x_2} - \frac{\partial f_{1,i}}{\partial x_2} \frac{\partial f_{2,i}}{\partial x_1}} \quad (11.12a)$$

$$x_{2,i+1} = x_{2,i} - \frac{f_{2,i} \frac{\partial f_{1,i}}{\partial x_1} - f_{1,i} \frac{\partial f_{2,i}}{\partial x_1}}{\frac{\partial f_{1,i}}{\partial x_1} \frac{\partial f_{2,i}}{\partial x_2} - \frac{\partial f_{1,i}}{\partial x_2} \frac{\partial f_{2,i}}{\partial x_1}} \quad (11.12b)$$

The denominator of each of these equations is formally referred to as the determinant of the *Jacobian* of the system.

Equation (11.12) is the two-equation version of the Newton-Raphson method. As in the following example, it can be employed iteratively to home in on the roots of two simultaneous equations.

EXAMPLE 11.3 Newton-Raphson for a Nonlinear System

Problem Statement. Use the multiple-equation Newton-Raphson method to determine roots of Eq. (11.6). Initiate the computation with guesses of $x_1 = 1.5$ and $x_2 = 3.5$.

Solution. First compute the partial derivatives and evaluate them at the initial guesses of x and y :

$$\begin{aligned} \frac{\partial f_{1,0}}{\partial x_1} &= 2x_1 + x_2 = 2(1.5) + 3.5 = 6.5 & \frac{\partial f_{1,0}}{\partial x_2} &= x_1 = 1.5 \\ \frac{\partial f_{2,0}}{\partial x_1} &= 3x_2^2 = 3(3.5)^2 = 36.75 & \frac{\partial f_{2,0}}{\partial x_2} &= 1 + 6x_1x_2 = 1 + 6(1.5)(3.5) = 32.5 \end{aligned}$$

Thus, the determinant of the Jacobian for the first iteration is

$$6.5(32.5) - 1.5(36.75) = 156.125$$

The values of the functions can be evaluated at the initial guesses as

$$\begin{aligned} f_{1,0} &= (1.5)^2 + 1.5(3.5) - 10 = -2.5 \\ f_{2,0} &= 3.5 + 3(1.5)(3.5)^2 - 57 = 1.625 \end{aligned}$$

11.2 NONLINEAR SYSTEMS

193

These values can be substituted into Eq. (11.12) to give

$$x_1 = 1.5 - \frac{-2.5(32.5) - 1.625(1.5)}{156.125} = 2.03603$$

$$x_2 = 3.5 - \frac{1.625(6.5) - (-2.5)(36.75)}{156.125} = 2.84388$$

Thus, the results are converging to the true values of $x_1 = 2$ and $x_2 = 3$. The computation can be repeated until an acceptable accuracy is obtained.

When the multiequation Newton-Raphson works, it exhibits the same speedy quadratic convergence as the single-equation version. However, just as with successive substitution, it can diverge if the initial guesses are not sufficiently close to the true roots. Whereas graphical methods could be employed to derive good guesses for the single-equation case, no such simple procedure is available for the multiequation version. Although there are some advanced approaches for obtaining acceptable first estimates, often the initial guesses must be obtained on the basis of trial and error and knowledge of the physical system being modeled.

The two-equation Newton-Raphson approach can be generalized to solve n simultaneous equations. To do this, Eq. (11.11) can be written for the k th equation as

$$\frac{\partial f_{k,i}}{\partial x_1} x_{1,i+1} + \frac{\partial f_{k,i}}{\partial x_2} x_{2,i+1} + \cdots + \frac{\partial f_{k,i}}{\partial x_n} x_{n,i+1} = -f_{k,i} + x_{1,i} \frac{\partial f_{k,i}}{\partial x_1} + x_{2,i} \frac{\partial f_{k,i}}{\partial x_2} + \cdots + x_{n,i} \frac{\partial f_{k,i}}{\partial x_n} \quad (11.13)$$

where the first subscript k represents the equation or unknown and the second subscript denotes whether the value or function in question is at the present value (i) or at the next value ($i + 1$). Notice that the only unknowns in Eq. (11.13) are the $x_{k,i+1}$ terms on the left-hand side. All other quantities are located at the present value (i) and, thus, are known at any iteration. Consequently, the set of equations generally represented by Eq. (11.13) (i.e., with $k = 1, 2, \dots, n$) constitutes a set of linear simultaneous equations that can be solved numerically by the elimination methods elaborated in previous chapters.

Matrix notation can be employed to express Eq. (11.13) concisely as

$$[Z]\{x_{i+1}\} = -\{f\} + [Z]\{x_i\} \quad (11.14)$$

where the partial derivatives evaluated at i are written as the *Jacobian matrix*:

$$[Z] = \begin{bmatrix} \frac{\partial f_{1,i}}{\partial x_1} & \frac{\partial f_{1,i}}{\partial x_2} & \cdots & \frac{\partial f_{1,i}}{\partial x_n} \\ \frac{\partial f_{2,i}}{\partial x_1} & \frac{\partial f_{2,i}}{\partial x_2} & \cdots & \frac{\partial f_{2,i}}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_{n,i}}{\partial x_1} & \frac{\partial f_{n,i}}{\partial x_2} & \cdots & \frac{\partial f_{n,i}}{\partial x_n} \end{bmatrix} \quad (11.15)$$

The initial and final values are expressed in vector form as

$$\{x_i\}^T = [x_{1,i} \quad x_{2,i} \quad \cdots \quad x_{n,i}]$$

and

$$\{x_{i+1}\}^T = [x_{1,i+1} \quad x_{2,i+1} \quad \cdots \quad x_{n,i+1}]$$

Finally, the function values at i can be expressed as

$$\{f\}^T = [f_{1,i} \quad f_{2,i} \quad \cdots \quad f_{n,i}]$$

Equation (11.14) can be solved using a technique such as Gauss elimination. This process can be repeated iteratively to obtain refined estimates in a fashion similar to the two-equation case in Example 11.3.

We should note that there are two shortcomings to the foregoing approach. First, Eq. (11.15) is sometimes inconvenient to evaluate. Therefore, variations of the Newton-Raphson approach have been developed to circumvent this dilemma. As might be expected, most are based on using finite-difference approximations for the partial derivatives that comprise $[Z]$. The second shortcoming of the multiequation Newton-Raphson method is that excellent initial guesses are usually required to ensure convergence. Because these are sometimes difficult or inconvenient to obtain, alternative approaches that are slower than Newton-Raphson but which have better convergence behavior have been developed. One approach is to reformulate the nonlinear system as a single function:

$$F(x) = \sum_{i=1}^n [f_i(x_1, x_2, \dots, x_n)]^2$$

where $f_i(x_1, x_2, \dots, x_n)$ is the i th member of the original system of Eq. (11.7). The values of x that minimize this function also represent the solution of the nonlinear system. Therefore, nonlinear optimization techniques can be employed to obtain solutions.

PROBLEMS

11.1 (a) Use the Gauss-Seidel method to solve the following system until the percent relative error falls below $\varepsilon_s = 5\%$:

$$\begin{bmatrix} 0.8 & -0.4 & \\ -0.4 & 0.8 & -0.4 \\ & -0.4 & 0.8 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 41 \\ 25 \\ 105 \end{Bmatrix}$$

(b) Repeat **(a)** but use overrelaxation with $\lambda = 1.2$.

11.2 Use the Gauss-Seidel method to solve the following system until the percent relative error falls below $\varepsilon_s = 5\%$:

$$\begin{aligned} 10x_1 + 2x_2 - x_3 &= 27 \\ -3x_1 - 6x_2 + 2x_3 &= -61.5 \\ x_1 + x_2 + 5x_3 &= -21.5 \end{aligned}$$

11.3 Repeat Prob. 11.2 but use Jacobi iteration.

11.4 The following system of equations is designed to determine concentrations (the c 's in g/m^3) in a series of coupled

reactors as a function of the amount of mass input to each reactor (the right-hand sides in g/day):

$$\begin{aligned} 15c_1 - 3c_2 - c_3 &= 3800 \\ -3c_1 + 18c_2 - 6c_3 &= 1200 \\ -4c_1 - c_2 + 12c_3 &= 2350 \end{aligned}$$

Solve this problem with the Gauss-Seidel method to $\varepsilon_s = 5\%$.

11.5 Use the Gauss-Seidel method **(a)** without relaxation and **(b)** with relaxation ($\lambda = 1.2$) to solve the following system to a tolerance of $\varepsilon_s = 5\%$. If necessary, rearrange the equations to achieve convergence.

$$\begin{aligned} 2x_1 - 6x_2 - x_3 &= -38 \\ -3x_1 - x_2 + 7x_3 &= -34 \\ -8x_1 + x_2 - 2x_3 &= -20 \end{aligned}$$

PROBLEMS

195

11.6 Of the following three sets of linear equations, identify the set that cannot be solved using an iterative method like Gauss-Seidel. Rearrange the equations as necessary to foster convergence. Show using any number of iterations that is necessary that your solution does not converge.

Set One	Set Two	Set Three
$8x + 3y + z = 12$	$-x + 3y + 5z = 7$	$x + y + 5z = 7$
$-6x + 7z = 1$	$-2x + 4y - 5z = -3$	$x + 4y - z = 4$
$2x + 4y - z = 5$	$2y - z = 1$	$3x + y - z = 3$

11.7 Determine the solution of the simultaneous nonlinear equations:

$$y = -x^2 + x + 0.5$$

$$y + 5xy = x^2$$

Use the Newton-Raphson method and employ initial guesses of $x = y = 1.2$.

11.8 Determine the solution of the simultaneous nonlinear equations:

$$x^2 = 5 - y^2$$

$$y + 1 = x^2$$

- (a) Graphically.
 (b) Successive substitution using initial guesses of $x = y = 1.5$.
 (c) Newton-Raphson using initial guesses of $x = y = 1.5$.

Curve Fitting: Fitting a Straight Line

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to how least-squares regression can be used to fit a straight line to measured data. Specific objectives and topics covered are

- Understanding the difference between regression and interpolation.
- Familiarizing yourself with some basic descriptive statistics and the normal distribution.
- Knowing how to compute the slope and intercept of a best-fit straight line with linear regression.
- Knowing how to compute and understand the meaning of the coefficient of determination and the standard error of the estimate.
- Understanding how to use transformations to linearize nonlinear equations so that they can be fit with linear regression.
- Knowing how to implement linear regression with MATLAB.

YOU'VE GOT A PROBLEM

In Chap. 1, we noted that a free-falling object such as a bungee jumper is subject to the upward force of air resistance. As a first approximation, we assumed that this force was proportional to the square of velocity as in

$$F_U = c_d v^2 \quad (12.1)$$

where F_U = the upward force of air resistance [$\text{N} = \text{kg m/s}^2$], c_d = a drag coefficient (kg/m), and v = velocity [m/s].

Expressions such as Eq. (12.1) come from the field of fluid mechanics. Although such relationships derive in part from theory, experiments play a critical role in their formulation. One such experiment is depicted in Fig. 12.1. An individual is suspended in a wind

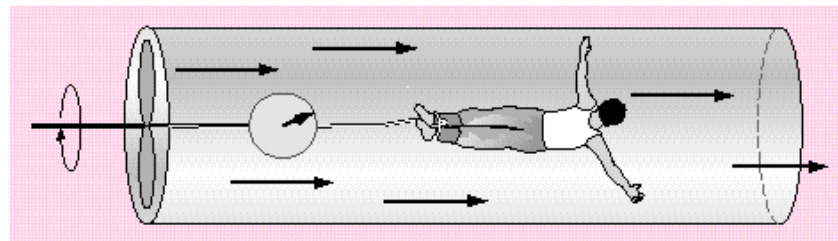


FIGURE 12.1

Wind tunnel experiment to measure how the force of air resistance depends on velocity.

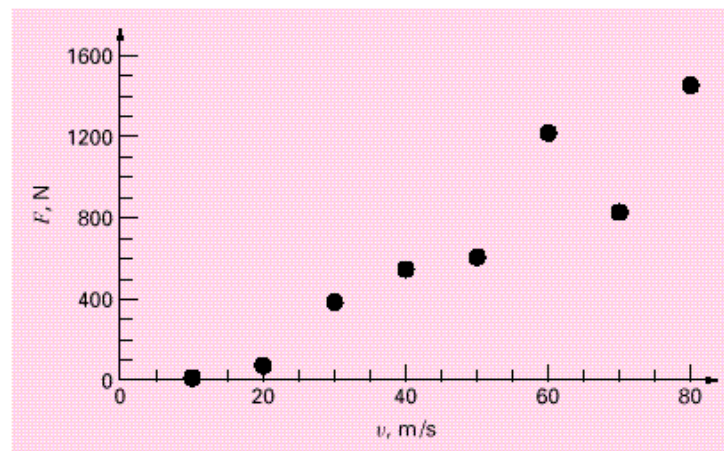


FIGURE 12.2

Plot of force versus wind velocity for an object suspended in a wind tunnel.

TABLE 12.1 Experimental data for force (N) and velocity (m/s) from a wind tunnel experiment.

v , m/s	10	20	30	40	50	60	70	80
F , N	25	70	380	550	610	1220	830	1450

tunnel (any volunteers?) and the force measured for various levels of wind velocity. The result might be as listed in Table 12.1.

The relationship can be visualized by plotting force versus velocity. As in Fig. 12.2, several features of the relationship bear mention. First, the points indicate that the force increases as velocity increases. Second, the points do not increase smoothly, but exhibit rather significant scatter, particularly at the higher velocities. Finally, although it may not be obvious, the relationship between force and velocity may not be linear. This conclusion becomes more apparent if we assume that force is zero for zero velocity.

In Chaps. 12 and 13, we will explore how to fit a “best” line or curve to such data. In so doing, we will illustrate how relationships like Eq. (12.1) arise from experimental data.

12.1 WHAT IS CURVE FITTING?

Data is often given for discrete values along a continuum. However, you may require estimates at points between the discrete values. Chapters 12 through 15 describe techniques to fit curves to such data to obtain intermediate estimates. In addition, you may require a simplified version of a complicated function. One way to do this is to compute values of the function at a number of discrete values along the range of interest. Then, a simpler function may be derived to fit these values. Both of these applications are known as *curve fitting*.

There are two general approaches for curve fitting that are distinguished from each other on the basis of the amount of error associated with the data. First, where the data exhibits a significant degree of error or “scatter,” the strategy is to derive a single curve that represents the general trend of the data. Because any individual data point may be incorrect, we make no effort to intersect every point. Rather, the curve is designed to follow the pattern of the points taken as a group. One approach of this nature is called *least-squares regression* (Fig. 12.3a).

Second, where the data is known to be very precise, the basic approach is to fit a curve or a series of curves that pass directly through each of the points. Such data usually originates from tables. Examples are values for the density of water or for the heat capacity of gases as a function of temperature. The estimation of values between well-known discrete points is called *interpolation* (Fig. 12.3b and c).

12.1.1 Curve Fitting and Engineering Practice

Your first exposure to curve fitting may have been to determine intermediate values from tabulated data—for instance, from interest tables for engineering economics or from steam tables for thermodynamics. Throughout the remainder of your career, you will have frequent occasion to estimate intermediate values from such tables.

Although many of the widely used engineering properties have been tabulated, there are a great many more that are not available in this convenient form. Special cases and new problem contexts often require that you measure your own data and develop your own predictive relationships. Two types of applications are generally encountered when fitting experimental data: trend analysis and hypothesis testing.

Trend analysis represents the process of using the pattern of the data to make predictions. For cases where the data is measured with high precision, you might utilize interpolating polynomials. Imprecise data is often analyzed with least-squares regression.

Trend analysis may be used to predict or forecast values of the dependent variable. This can involve extrapolation beyond the limits of the observed data or interpolation within the range of the data. All fields of engineering and science involve problems of this type.

A second application of experimental curve fitting is *hypothesis testing*. Here, an existing mathematical model is compared with measured data. If the model coefficients are unknown, it may be necessary to determine values that best fit the observed data. On the other hand, if estimates of the model coefficients are already available, it may be appropriate to compare predicted values of the model with observed values to test the adequacy of

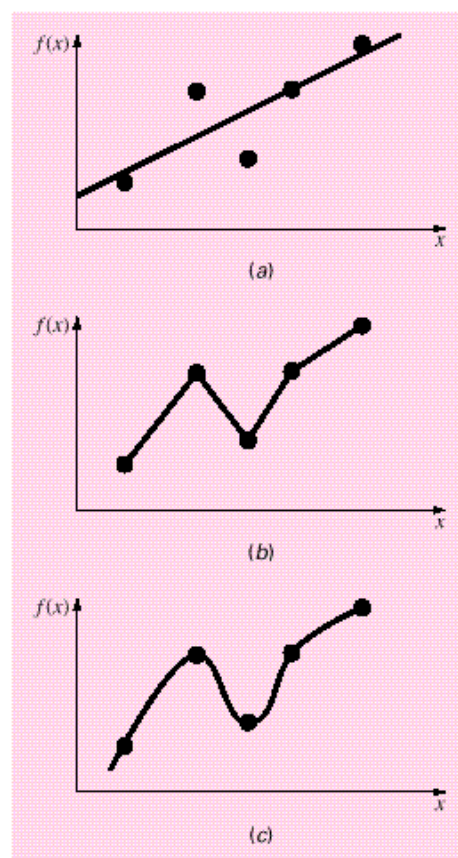


FIGURE 12.3

Three attempts to fit a “best” curve through five data points: (—) least-squares regression, (—) linear interpolation, and (—) curvilinear interpolation.

the model. Often, alternative models are compared and the “best” one is selected on the basis of empirical observations.

In addition to the foregoing engineering applications, curve fitting is important in other numerical methods such as integration and the approximate solution of differential equations. Finally, curve-fitting techniques can be used to derive simple functions to approximate complicated functions.

12.2 STATISTICS REVIEW

Before describing least-squares regression, we will first review some basic concepts from the field of statistics. If you are familiar with the concepts of the mean, standard deviation, residual sum of the squares, and the normal distribution, feel free to skip the following pages and proceed directly to Section 12.3. If you are unfamiliar with these concepts or are in need of a review, the following material is designed as a brief introduction to these topics.

TABLE 12.2 Measurements of the coefficient of thermal expansion of structural steel
[$\times 10^{-6}$] in/[in \cdot $^{\circ}$ F].

6.495	6.595	6.615	6.635	6.485	6.555
6.665	6.505	6.435	6.625	6.715	6.655
6.755	6.625	6.715	6.575	6.655	6.605
6.565	6.515	6.555	6.395	6.775	6.685

12.2.1 Simple Statistics

Suppose that in the course of an engineering study, several measurements were made of a particular quantity. For example, Table 12.2 contains 24 readings of the coefficient of thermal expansion of a structural steel. Taken at face value, the data provides a limited amount of information—that is, that the values range from a minimum of 6.395 to a maximum of 6.775. Additional insight can be gained by summarizing the data in one or more well-chosen statistics that convey as much information as possible about specific characteristics of the data set. These descriptive statistics are most often selected to represent (1) the location of the center of the distribution of the data and (2) the degree of spread of the data set.

The most common measure of central tendency is the arithmetic mean. The *arithmetic mean* (\bar{y}) of a sample is defined as the sum of the individual data points (y_i) divided by the number of points (n), or

$$\bar{y} = \frac{\sum y_i}{n} \quad (12.2)$$

where the summation (and all the succeeding summations in this section) is from $i = 1$ through n .

The most common measure of spread for a sample is the *standard deviation* (s_y) about the mean:

$$s_y = \sqrt{\frac{S_t}{n-1}} \quad (12.3)$$

where S_t is the total sum of the squares of the residuals between the data points and the mean, or

$$S_t = \sum (y_i - \bar{y})^2 \quad (12.4)$$

Thus, if the individual measurements are spread out widely around the mean, S_t (and, consequently, s_y) will be large. If they are grouped tightly, the standard deviation will be small. The spread can also be represented by the square of the standard deviation, which is called the *variance*:

$$s_y^2 = \frac{\sum (y_i - \bar{y})^2}{n-1} \quad (12.5)$$

Note that the denominator in both Eqs. (12.3) and (12.5) is $n - 1$. The quantity $n - 1$ is referred to as the *degrees of freedom*. Hence S_t and s_y are said to be based on $n - 1$ degrees of freedom. This nomenclature derives from the fact that the sum of the quantities

12.2 STATISTICS REVIEW

201

upon which S_y is based (i.e., $\bar{y} - y_1, \bar{y} - y_2, \dots, \bar{y} - y_n$) is zero. Consequently, if \bar{y} is known and $n - 1$ of the values are specified, the remaining value is fixed. Thus, only $n - 1$ of the values are said to be freely determined. Another justification for dividing by $n - 1$ is the fact that there is no such thing as the spread of a single data point. For the case where $n = 1$, Eqs. (12.3) and (12.5) yield a meaningless result of infinity.

We should note that an alternative, more convenient formula is available to compute the variance:

$$s_y^2 = \frac{\sum y_i^2 - (\sum y_i)^2 / n}{n - 1} \quad (12.6)$$

This version does not require precomputation of \bar{y} and yields an identical result as Eq. (12.5).

A final statistic that has utility in quantifying the spread of data is the coefficient of variation (c.v.). This statistic is the ratio of the standard deviation to the mean. As such, it provides a normalized measure of the spread. It is often multiplied by 100 so that it can be expressed in the form of a percent:

$$\text{c.v.} = \frac{s_y}{\bar{y}} \times 100\% \quad (12.7)$$

EXAMPLE 12.1 Simple Statistics of a Sample

Problem Statement. Compute the mean, variance, standard deviation, and coefficient of variation for the data in Table 12.2.

Solution. The data can be assembled in tabular form and the necessary sums computed as in Table 12.3.

The mean can be computed as [Eq. (12.2)],

$$\bar{y} = \frac{158.4}{24} = 6.6$$

As in Table 12.3, the sum of the squares of the residuals is 0.217000, which can be used to compute the standard deviation [Eq. (12.3)]:

$$s_y = \sqrt{\frac{0.217000}{24 - 1}} = 0.097133$$

the variance [Eq. (12.5)]:

$$s_y^2 = (0.097133)^2 = 0.009435$$

and the coefficient of variation [Eq. (12.7)]:

$$\text{c.v.} = \frac{0.097133}{6.6} \times 100\% = 1.47\%$$

The validity of Eq. (12.6) can also be verified by computing

$$s_y^2 = \frac{1045.657 - (158.400)^2 / 24}{24 - 1} = 0.009435$$

TABLE 12.3 Data and summations for computing simple descriptive statistics for the coefficients of thermal expansion from Table 12.2.

i	x_i	$(x_i - \bar{x})^2$	x_i^2
1	6.395	0.04203	40.896
2	6.435	0.02723	41.409
3	6.485	0.01323	42.055
4	6.495	0.01103	42.185
5	6.505	0.00903	42.315
6	6.515	0.00723	42.445
7	6.555	0.00203	42.968
8	6.555	0.00203	42.968
9	6.565	0.00123	43.099
10	6.575	0.00063	43.231
11	6.595	0.00003	43.494
12	6.605	0.00002	43.626
13	6.615	0.00022	43.758
14	6.625	0.00062	43.891
15	6.625	0.00062	43.891
16	6.635	0.00122	44.023
17	6.655	0.00302	44.289
18	6.655	0.00302	44.289
19	6.665	0.00422	44.422
20	6.685	0.00722	44.689
21	6.715	0.01322	45.091
22	6.715	0.01322	45.091
23	6.755	0.02402	45.630
24	6.775	0.03062	45.901
Σ	158.400	0.21700	1045.657

12.2.2 The Normal Distribution

Another characteristic that bears on the present discussion is the data distribution—that is, the shape with which the data is spread around the mean. A histogram provides a simple visual representation of the distribution. A histogram is constructed by sorting the measurements into intervals. The units of measurement are plotted on the abscissa and the frequency of occurrence of each interval is plotted on the ordinate.

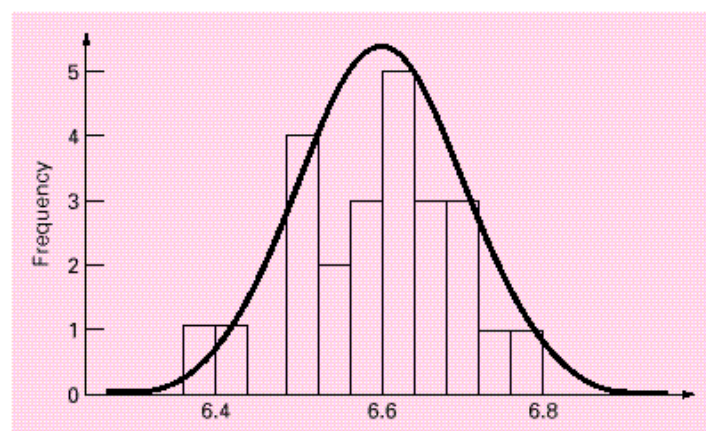
As an example, a histogram can be created for the data from Table 12.2. The result (Fig. 12.4) suggests that most of the data is grouped close to the mean value of 6.6.

If we have a very large set of data, the histogram often can be approximated by a smooth curve. The symmetric, bell-shaped curve superimposed on Fig. 12.4 is one such characteristic shape—the *normal distribution*. Given enough additional measurements, the histogram for this particular case could eventually approach the normal distribution.

The concepts of the mean, standard deviation, residual sum of the squares, and normal distribution all have great relevance to engineering practice. A very simple example is their use to quantify the confidence that can be ascribed to a particular measurement. If a quantity is normally distributed, the range defined by $\bar{y} - s_y$ to $\bar{y} + s_y$ will encompass

12.3 LINEAR LEAST-SQUARES REGRESSION

203

**FIGURE 12.4**

A histogram used to depict the distribution of data. As the number of data points increases, the histogram often approaches the smooth, bell-shaped curve called the normal distribution.

approximately 68% of the total measurements. Similarly, the range defined by $\bar{y} - 2s_y$ to $\bar{y} + 2s_y$ will encompass approximately 95%.

For example, for the data in Table 12.2, we calculated in Example 12.1 that $\bar{y} = 6.6$ and $s_y = 0.097133$. Based on our analysis, we can tentatively make the statement that approximately 95% of the readings should fall between 6.405734 and 6.794266. Because it is so far outside these bounds, if someone told us that they had measured a value of 7.35, we would suspect that the measurement might be erroneous.

12.3 LINEAR LEAST-SQUARES REGRESSION

Where substantial error is associated with data, the best curve-fitting strategy is to derive an approximating function that fits the shape or general trend of the data without necessarily matching the individual points. One way to do this is to derive the curve that minimizes the sum of the squares of the discrepancy between the data points and the curve. This technique is called *least-squares regression*.

The simplest example of a least-squares approximation is fitting a straight line to a set of paired observations: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. The mathematical expression for the straight line is

$$y = a_0 + a_1x + e \quad (12.8)$$

where a_0 and a_1 are coefficients representing the intercept and the slope, respectively, and e is the error, or residual, between the model and the observations, which can be represented by rearranging Eq. (12.8) as

$$e = y - a_0 - a_1x \quad (12.9)$$

Thus, the error, or residual, is the discrepancy between the true value of y and the approximate value, $a_0 + a_1x$, predicted by the linear equation.

12.3.1 Criteria for a “Best” Fit

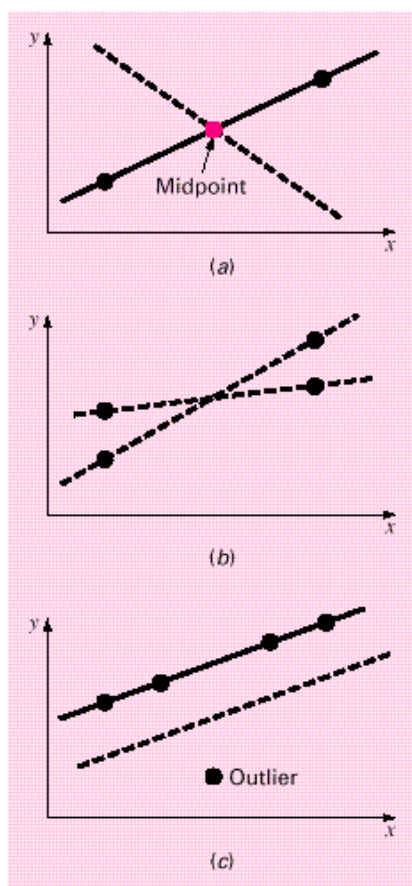
One strategy for fitting a “best” line through the data would be to minimize the sum of the residual errors for all the available data, as in

$$\sum_{i=1}^n e_i = \sum_{i=1}^n (y_i - a_0 - a_1 x_i) \quad (12.10)$$

where n = total number of points. However, this is an inadequate criterion, as illustrated by Fig. 12.5a, which depicts the fit of a straight line to two points. Obviously, the best fit is the line connecting the points. However, any straight line passing through the midpoint of the connecting line (except a perfectly vertical line) results in a minimum value of Eq. (12.10) equal to zero because positive and negative errors cancel.

FIGURE 12.5

Examples of some criteria for “best fit” that are inadequate for regression: (·) minimizes the sum of the residuals, (·) minimizes the sum of the absolute values of the residuals, and (·) minimizes the maximum error of any individual point.



12.3 LINEAR LEAST-SQUARES REGRESSION

205

Another logical criterion might be to minimize the sum of the absolute values of the discrepancies, as in

$$\sum_{i=1}^n |e_i| = \sum_{i=1}^n |y_i - a_0 - a_1 x_i| \quad (12.11)$$

Figure 12.5*b* demonstrates why this criterion is also inadequate. For the four points shown, any straight line falling within the dashed lines will minimize the sum of the absolute values of the residuals. Thus, this criterion also does not yield a unique best fit.

A third strategy for fitting a best line is the *minimax* criterion. In this technique, the line is chosen that minimizes the maximum distance that an individual point falls from the line. As depicted in Fig. 12.5*c*, this strategy is ill-suited for regression because it gives undue influence to an outlier—that is, a single point with a large error. It should be noted that the minimax principle is sometimes well-suited for fitting a simple function to a complicated function (Carnahan, Luther, and Wilkes, 1969).

A strategy that overcomes the shortcomings of the aforementioned approaches is to minimize the sum of the squares of the residuals:

$$S_r = \sum_{i=1}^n e_i^2 = \sum_{i=1}^n (y_i - a_0 - a_1 x_i)^2 \quad (12.12)$$

This criterion has a number of advantages, including the fact that it yields a unique line for a given set of data. Before discussing these properties, we will present a technique for determining the values of a_0 and a_1 that minimize Eq. (12.12).

12.3.2 Least-Squares Fit of a Straight Line

To determine values for a_0 and a_1 , Eq. (12.12) is differentiated with respect to each unknown coefficient:

$$\begin{aligned} \frac{\partial S_r}{\partial a_0} &= -2 \sum (y_i - a_0 - a_1 x_i) \\ \frac{\partial S_r}{\partial a_1} &= -2 \sum [(y_i - a_0 - a_1 x_i)x_i] \end{aligned}$$

Note that we have simplified the summation symbols; unless otherwise indicated, all summations are from $i = 1$ to n . Setting these derivatives equal to zero will result in a minimum S_r . If this is done, the equations can be expressed as

$$\begin{aligned} 0 &= \sum y_i - \sum a_0 - \sum a_1 x_i \\ 0 &= \sum x_i y_i - \sum a_0 x_i - \sum a_1 x_i^2 \end{aligned}$$

Now, realizing that $\sum a_0 = na_0$, we can express the equations as a set of two simultaneous linear equations with two unknowns (a_0 and a_1):

$$na_0 + \left(\sum x_i\right)a_1 = \sum y_i \quad (12.13)$$

$$\left(\sum x_i\right)a_0 + \left(\sum x_i^2\right)a_1 = \sum x_i y_i \quad (12.14)$$

These are called the *normal equations*. They can be solved simultaneously for

$$a_1 = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2} \quad (12.15)$$

This result can then be used in conjunction with Eq. (12.13) to solve for

$$a_0 = \bar{y} - a_1 \bar{x} \quad (12.16)$$

where \bar{y} and \bar{x} are the means of y and x , respectively.

EXAMPLE 12.2 Linear Regression

Problem Statement. Fit a straight line to the values in Table 12.1.

Solution. In this application, force is the dependent variable (y) and velocity is the independent variable (x). The data can be set up in tabular form and the necessary sums computed as in Table 12.4.

The means can be computed as

$$\bar{x} = \frac{360}{8} = 45 \quad \bar{y} = \frac{5,135}{8} = 641.875$$

The slope and the intercept can then be calculated with Eqs. (12.15) and (12.16) as

$$a_1 = \frac{8(312,850) - 360(5,135)}{8(20,400) - (360)^2} = 19.47024$$

$$a_0 = 641.875 - 19.47024(45) = -234.2857$$

Using force and velocity in place of y and x , the least-squares fit is

$$F = -234.2857 + 19.47024v$$

The line, along with the data, is shown in Fig. 12.6.

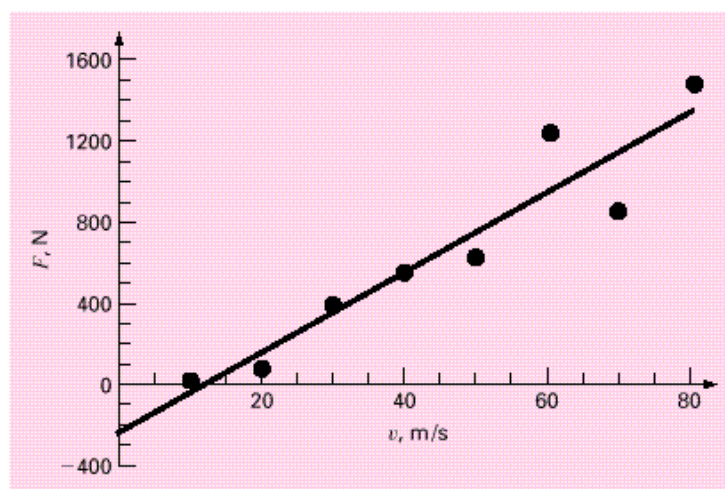
Notice that although the line fits the data well, the zero intercept means that the equation predicts physically unrealistic negative forces at low velocities. In Section 12.3.4, we

TABLE 12.4 Data and summations needed to compute the best-fit line for the data from Table 12.1.

i	x_i	y_i	x_i^2	$x_i y_i$
1	10	25	100	250
2	20	70	400	1,400
3	30	380	900	11,400
4	40	550	1,600	22,000
5	50	610	2,500	30,500
6	60	1,220	3,600	73,200
7	70	830	4,900	58,100
8	80	1,450	6,400	116,000
Σ	360	5,135	20,400	312,850

12.3 LINEAR LEAST-SQUARES REGRESSION

207

**FIGURE 12.6**

Least-squares fit of a straight line to the data from Table 12.1

will show how transformations can be employed to derive an alternative best-fit line that is more physically realistic.

12.3.3 Quantification of Error of Linear Regression

Any line other than the one computed in Example 12.2 results in a larger sum of the squares of the residuals. Thus, the line is unique and in terms of our chosen criterion is a “best” line through the points. A number of additional properties of this fit can be elucidated by examining more closely the way in which residuals were computed. Recall that the sum of the squares is defined as [Eq. (12.12)]

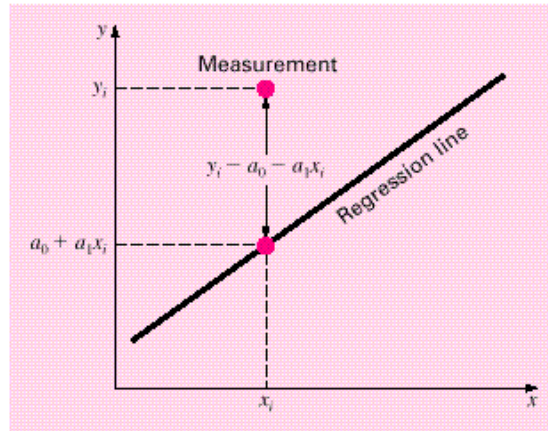
$$S_r = \sum_{i=1}^n (y_i - a_0 - a_1 x_i)^2 \quad (12.17)$$

Notice the similarity between this equation and Eq. (12.4)

$$S_t = \sum (y_i - \bar{y})^2 \quad (12.18)$$

In Eq. (12.18), the square of the residual represented the square of the discrepancy between the data and a single estimate of the measure of central tendency—the mean. In Eq. (12.17), the square of the residual represents the square of the vertical distance between the data and another measure of central tendency—the straight line (Fig. 12.7).

The analogy can be extended further for cases where (1) the spread of the points around the line is of similar magnitude along the entire range of the data and (2) the distribution of these points about the line is normal. It can be demonstrated that if these criteria are met, least-squares regression will provide the best (i.e., the most likely) estimates of a_0


FIGURE 12.7

The residual in linear regression represents the vertical distance between a data point and the straight line.

and a_1 (Draper and Smith, 1981). This is called the *maximum likelihood principle* in statistics. In addition, if these criteria are met, a “standard deviation” for the regression line can be determined as [compare with Eq. (12.3)]

$$s_{y/x} = \sqrt{\frac{S_r}{n-2}} \quad (12.19)$$

where $s_{y/x}$ is called the *standard error of the estimate*. The subscript notation “ y/x ” designates that the error is for a predicted value of y corresponding to a particular value of x . Also, notice that we now divide by $n-2$ because two data-derived estimates— a_0 and a_1 —were used to compute S_r ; thus, we have lost two degrees of freedom. As with our discussion of the standard deviation, another justification for dividing by $n-2$ is that there is no such thing as the “spread of data” around a straight line connecting two points. Thus, for the case where $n=2$, Eq. (12.19) yields a meaningless result of infinity.

Just as was the case with the standard deviation, the standard error of the estimate quantifies the spread of the data. However, $s_{y/x}$ quantifies the spread *around the regression line* as shown in Fig. 12.8b in contrast to the standard deviation s_y that quantified the spread *around the mean* (Fig. 12.8a).

These concepts can be used to quantify the “goodness” of our fit. This is particularly useful for comparison of several regressions (Fig. 12.9). To do this, we return to the original data and determine the total sum of the squares around the mean for the dependent variable (in our case, y). As was the case for Eq. (12.18), this quantity is designated S_t . This is the magnitude of the residual error associated with the dependent variable prior to regression. After performing the regression, we can compute S_r , the sum of the squares of the residuals around the regression line. This characterizes the residual error that remains after the regression. It is, therefore, sometimes called the unexplained sum of the squares. The difference between the two quantities, $S_t - S_r$, quantifies the improvement or error

12.3 LINEAR LEAST-SQUARES REGRESSION

209

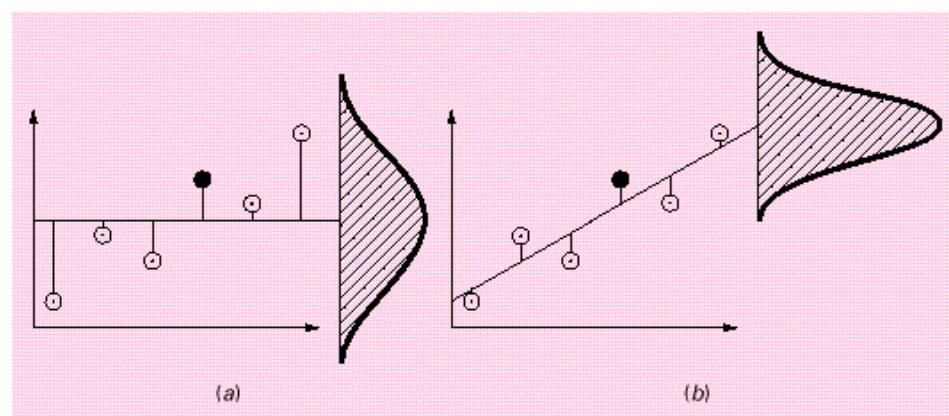


FIGURE 12.8

Regression data showing (·) the spread of the data around the mean of the dependent variable and (·) the spread of the data around the best-fit line. The reduction in the spread in going from (·) to (·), as indicated by the bell-shaped curves at the right, represents the improvement due to linear regression.

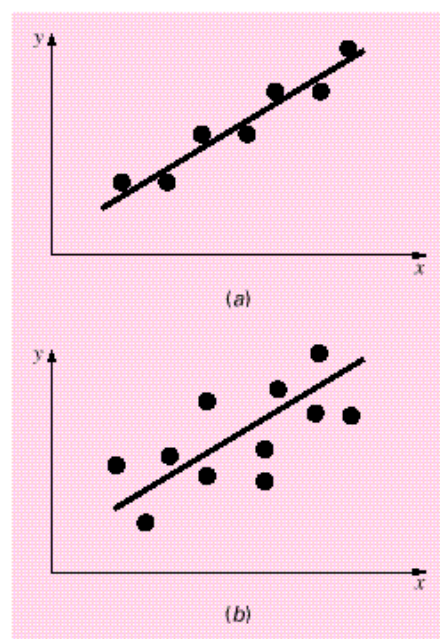


FIGURE 12.9

Examples of linear regression with (·) small and (·) large residual errors.

reduction due to describing the data in terms of a straight line rather than as an average value. Because the magnitude of this quantity is scale-dependent, the difference is normalized to S_t to yield

$$r^2 = \frac{S_t - S_r}{S_t} \quad (12.20)$$

where r^2 is called the *coefficient of determination* and r is the *correlation coefficient* ($= \sqrt{r^2}$). For a perfect fit, $S_r = 0$ and $r^2 = 1$, signifying that the line explains 100% of the variability of the data. For $r^2 = 0$, $S_r = S_t$ and the fit represents no improvement. An alternative formulation for r that is more convenient for computer implementation is

$$r = \frac{n \sum (x_i y_i) - (\sum x_i)(\sum y_i)}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}} \quad (12.21)$$

EXAMPLE 12.3 Estimation of Errors for the Linear Least-Squares Fit

Problem Statement. Compute the total standard deviation, the standard error of the estimate, and the correlation coefficient for the data in Example 12.2.

Solution. The data can be set up in tabular form and the necessary sums computed as in Table 12.5.

The standard deviation is [Eq. (12.3)]

$$s_y = \sqrt{\frac{1,808,297}{8-1}} = 508.26$$

and the standard error of the estimate is [Eq. (12.19)]

$$s_{y/x} = \sqrt{\frac{216,118}{8-2}} = 189.79$$

TABLE 12.5 Data and summations needed to compute the goodness-of-fit statistics for the data from Table 12.1.

i	x_i	y_i	$y_0 = 1$	$(y_i - y_0)^2$	$(y_i - y_0 - y_1 y_0)^2$
1	10	25	-39.58	380,535	4,171
2	20	70	-155.12	327,041	7,245
3	30	380	349.82	68,579	911
4	40	550	544.52	8,441	30
5	50	610	739.23	1,016	16,699
6	60	1,220	933.93	334,229	81,837
7	70	830	1,128.63	35,391	89,180
8	80	1,450	1,323.33	653,066	16,044
Σ	360	5,135		1,808,297	216,118

12.3 LINEAR LEAST-SQUARES REGRESSION

211

Thus, because $s_{y/x} < s_y$, the linear regression model has merit. The extent of the improvement is quantified by [Eq. (12.20)]

$$r^2 = \frac{1,808,297 - 216,118}{1,808,297} = 0.8805$$

or $r = \sqrt{0.8805} = 0.9383$. These results indicate that 88.05% of the original uncertainty has been explained by the linear model.

Before proceeding, a word of caution is in order. Although the coefficient of determination provides a handy measure of goodness-of-fit, you should be careful not to ascribe more meaning to it than is warranted. Just because r^2 is “close” to 1 does not mean that the fit is necessarily “good.” For example, it is possible to obtain a relatively high value of r^2 when the underlying relationship between y and x is not even linear. Draper and Smith (1981) provide guidance and additional material regarding assessment of results for linear regression. In addition, at the minimum, you should always inspect a plot of the data along with your regression curve.

12.3.4 Linearization of Nonlinear Relationships

Linear regression provides a powerful technique for fitting a best line to data. However, it is predicated on the fact that the relationship between the dependent and independent variables is linear. This is not always the case, and the first step in any regression analysis should be to plot and visually inspect the data to ascertain whether a linear model applies. In some cases, techniques such as polynomial regression, which is described in Chap. 13, are appropriate. For others, transformations can be used to express the data in a form that is compatible with linear regression.

One example is the *exponential model*:

$$y = \alpha_1 e^{\beta_1 x} \quad (12.22)$$

where α_1 and β_1 are constants. This model is used in many fields of engineering and science to characterize quantities that increase (positive β_1) or decrease (negative β_1) at a rate that is directly proportional to their own magnitude. For example, population growth or radioactive decay can exhibit such behavior. As depicted in Fig. 12.10a, the equation represents a nonlinear relationship (for $\beta_1 \neq 0$) between y and x .

Another example of a nonlinear model is the simple *power equation*:

$$y = \alpha_2 x^{\beta_2} \quad (12.23)$$

where α_2 and β_2 are constant coefficients. This model has wide applicability in all fields of engineering and science. As depicted in Fig. 12.10b, the equation (for $\beta_2 \neq 0$) is nonlinear.

A third example of a nonlinear model is the *saturation-growth-rate equation*:

$$y = \alpha_3 \frac{x}{\beta_3 + x} \quad (12.24)$$

where α_3 and β_3 are constant coefficients. This model, which is particularly well-suited for characterizing population growth rate under limiting conditions, also represents a

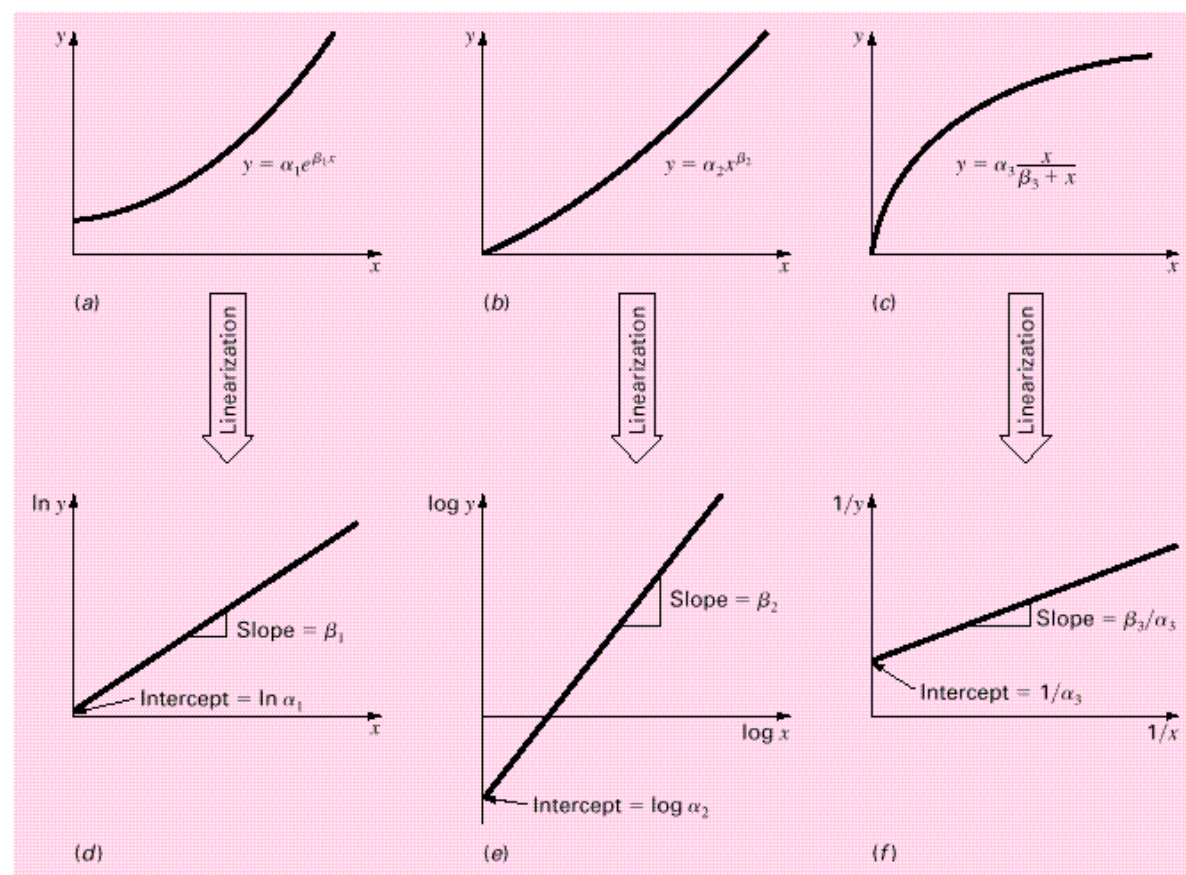


FIGURE 12.10

(·) The exponential equation, (·) the power equation, and (·) the saturation-growth-rate equation. Parts (·), (·), and (·) are linearized versions of these equations that result from simple transformations.

nonlinear relationship between y and x (Fig. 12.10c) that levels off, or “saturates,” as x increases.

Nonlinear regression techniques are available to fit these equations to experimental data directly. However, a simpler alternative is to use mathematical manipulations to transform the equations into a linear form. Then simple linear regression can be employed to fit the equations to data.

For example, Eq. (12.22) can be linearized by taking its natural logarithm to yield

$$\ln y = \ln \alpha_1 + \beta_1 x \quad (12.25)$$

Thus, a plot of $\ln y$ versus x will yield a straight line with a slope of β_1 and an intercept of $\ln \alpha_1$ (Fig. 12.10d).

Equation (12.23) is linearized by taking its base-10 logarithm to give

$$\log y = \log \alpha_2 + \beta_2 \log x \quad (12.26)$$

12.3 LINEAR LEAST-SQUARES REGRESSION

213

Thus, a plot of $\log y$ versus $\log x$ will yield a straight line with a slope of β_2 and an intercept of $\log \alpha_2$ (Fig. 12.10e). Note that any base logarithm can be used to linearize this model. However, as done here, the base-10 logarithm is most commonly employed.

Equation (12.24) is linearized by inverting it to give

$$\frac{1}{y} = \frac{1}{\alpha_3} + \frac{\beta_3}{\alpha_3} \frac{1}{x} \quad (12.27)$$

Thus, a plot of $1/y$ versus $1/x$ will be linear, with a slope of β_3/α_3 and an intercept of $1/\alpha_3$ (Fig. 12.10f).

In their transformed forms, these models can use linear regression to evaluate the constant coefficients. They could then be transformed back to their original state and used for predictive purposes. The following illustrates this procedure for the power model.

EXAMPLE 12.4 Fitting Data with the Power Equation

Problem Statement. Fit Eq. (12.23) to the data in Table 12.1 using a logarithmic transformation of the data.

Solution. The data can be set up in tabular form and the necessary sums computed as in Table 12.6.

The means can be computed as

$$\bar{x} = \frac{12.606}{8} = 1.5757 \quad \bar{y} = \frac{20.515}{8} = 2.5644$$

The slope and the intercept can then be calculated with Eqs. (12.15) and (12.16) as

$$a_1 = \frac{8(33.622) - 12.606(20.515)}{8(20.516) - (12.606)^2} = 1.9842$$

$$a_0 = 2.5644 - 1.9842(1.5757) = -0.5620$$

TABLE 12.6 Data and summations needed to fit the power model to the data from Table 12.1

	x_i	y_i	$\log x_i$	$\log y_i$	$(\log x_i)^2$	$\log x_i \log y_i$
1	10	25	1.000	1.398	1.000	1.398
2	20	70	1.301	1.845	1.693	2.401
3	30	380	1.477	2.580	2.182	3.811
4	40	550	1.602	2.740	2.567	4.390
5	50	610	1.699	2.785	2.886	4.732
6	60	1220	1.778	3.086	3.162	5.488
7	70	830	1.845	2.919	3.404	5.386
8	80	1450	1.903	3.161	3.622	6.016
Σ			12.606	20.515	20.516	33.622

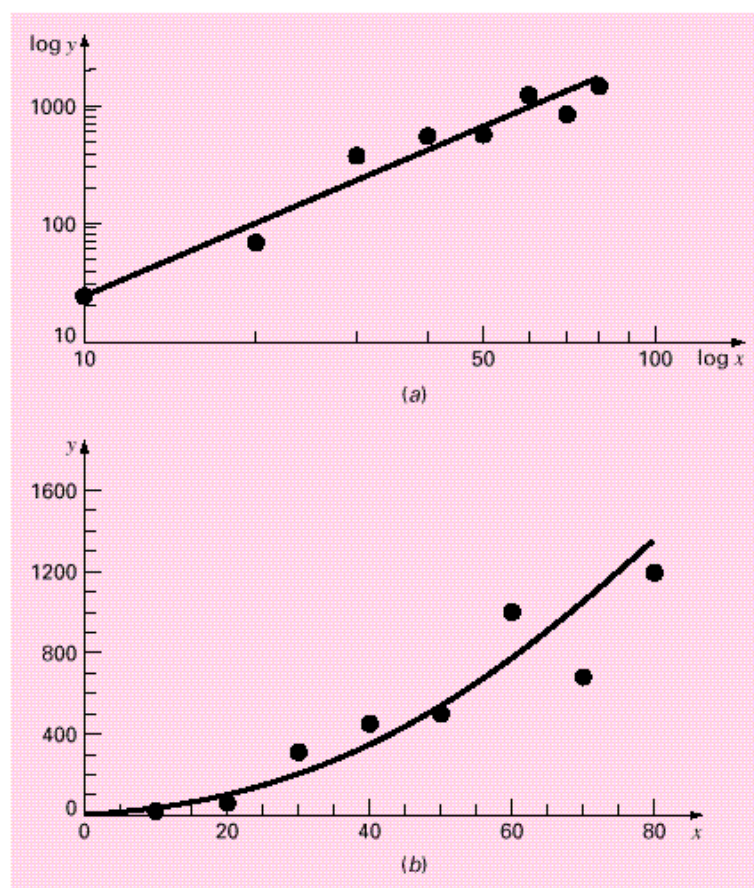


FIGURE 12.11

Least-squares fit of a power model to the data from Table 12.1. (· ·) The fit of the transformed data. (—) The power equation fit along with the data.

The least-squares fit is

$$\log y = -0.5620 + 1.9842 \log x$$

The fit along with the data is shown in Fig. 12.11a.

We can also display the fit using the untransformed coordinates. To do this, the coefficients of the power model are determined as $\alpha_2 = 10^{-0.5620} = 0.2741$ and $\beta_2 = 1.9842$. Using force and velocity in place of y and x , the least-squares fit is

$$F = 0.2741v^{1.9842}$$

This equation, along with the data, is shown in Fig. 12.11b.

The fits in Example 12.4 (Fig. 12.11) should be compared with the one obtained previously in Example 12.2 (Fig. 12.6) using linear regression on the untransformed data. Although both results would appear to be acceptable, the transformed result has the advantage that it does not yield negative force predictions at low velocities. Further, it is known from the discipline of fluid mechanics that the drag force on an object moving through a fluid is often well described by a model with velocity raised to a power. Thus, knowledge from the field you are studying often has a large bearing on the choice of the appropriate model equation you use for curve fitting.

12.3.5 General Comments on Linear Regression

Before proceeding to curvilinear and multiple linear regression, we must emphasize the introductory nature of the foregoing material on linear regression. We have focused on the simple derivation and practical use of equations to fit data. You should be cognizant of the fact that there are theoretical aspects of regression that are of practical importance but are beyond the scope of this book. For example, some statistical assumptions that are inherent in the linear least-squares procedures are

1. Each x has a fixed value; it is not random and is known without error.
2. The y values are independent random variables and all have the same variance.
3. The y values for a given x must be normally distributed.

Such assumptions are relevant to the proper derivation and use of regression. For example, the first assumption means that (1) the x values must be error-free and (2) the regression of y versus x is not the same as x versus y . You are urged to consult other references such as Draper and Smith (1981) to appreciate aspects and nuances of regression that are beyond the scope of this book.

12.4 COMPUTER APPLICATIONS

Linear regression is so commonplace that it can be implemented on most pocket calculators. In this section, we will show how a simple M-file can be developed to determine the slope and intercept as well as to create a plot of the data and the best-fit line. We will also show how linear regression can be implemented with the built-in `polyfit` function.

12.4.1 MATLAB M-file:

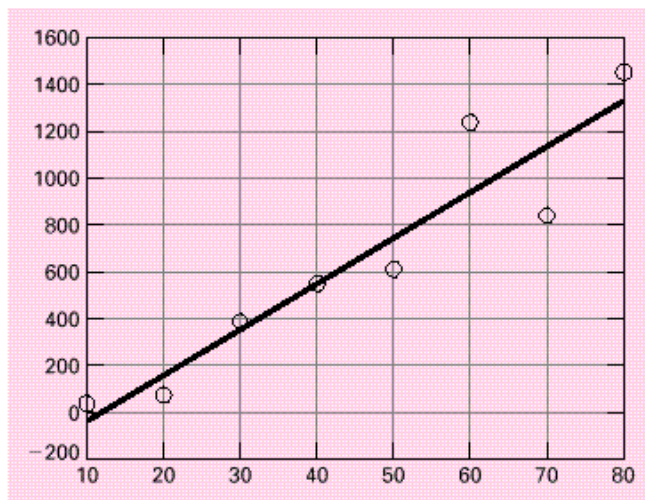
An algorithm for linear regression can be easily developed (Fig. 12.12). The required summations are readily computed with MATLAB's `sum` function. These are then used to compute the slope and the intercept with Eqs. (12.15) and (12.16). The routine displays the intercept and slope, the coefficient of determination, and a plot of the best-fit line along with the measurements.

A simple example of the use of this M-file would be to fit the force-velocity data that was analyzed in Example 12.2:

```
>> x = [10 20 30 40 50 60 70 80];  
>> y = [25 70 380 550 610 1220 830 1450];  
>> linregr(x,y)
```

```
r2 =
    0.8805

ans =
    19.4702 -234.2857
```

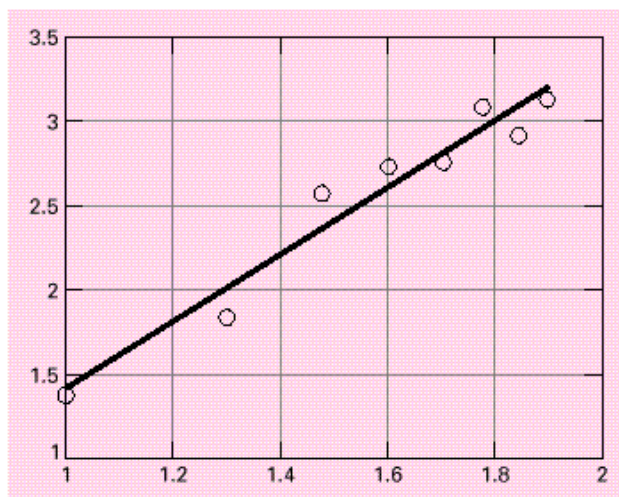


It can just as easily be used to fit the power model (Example 12.4) by applying the `log10` function to the data as in

```
>> linregr(log10(x),log10(y))

r2 =
    0.9481

ans =
    1.9842 -0.5620
```



12.4 COMPUTER APPLICATIONS

217

```
function [a, r2] = linregr(x,y)
% linregr(x,y):
%   Least squares fit of a straight line to data
%   by solving the normal equations.
% input:
%   x = independent variable
%   y = dependent variable
% output:
%   a = vector of slope, a(1), and intercept, a(2)
%   r2 = coefficient of determination

n = length(x);
if length(y)~=n, error('x and y must be same length'); end
x = x(:); y = y(:); % convert to column vectors
sx = sum(x); sy = sum(y);
sx2 = sum(x.*x); sxy = sum(x.*y); sy2 = sum(y.*y);
a(1) = (n*sxy-sx*sy)/(n*sx2-sx^2);
a(2) = sy/n-a(1)*sx/n;
r2 = ((n*sxy-sx*sy)/sqrt(n*sx2-sx^2)/sqrt(n*sy2-sy^2))^2
% create plot of data and best fit line
xp = linspace(min(x),max(x),2);
yp = a(1)*xp+a(2);
plot(x,y,'o',xp,yp)
grid on
```

FIGURE 12.12

An Mfile to implement linear regression.

12.4.2 MATLAB Functions: `polyfit` and `polyval`

MATLAB has a built-in function `polyfit` that fits a least-squares n th-order polynomial to data. It can be applied as in

```
>> p = polyfit(x, y, n)
```

where `x` and `y` are the vectors of the independent and the dependent variables, respectively, and `n` is the order of the polynomial. The function returns a vector `p` containing the polynomial's coefficients. We should note that it represents the polynomial using decreasing powers of x as in the following representation:

$$f(x) = p_1x^{n-1} + p_2x^{n-2} + \cdots + p_{n-1}x + p_n$$

Because a straight line is a first-order polynomial, `polyfit(x,y,1)` will return the slope and the intercept of the best-fit straight line.

```
>> x = [10 20 30 40 50 60 70 80];
>> y = [25 70 380 550 610 1220 830 1450];
>> a = polyfit(x,y,1)
```



```
a =
    19.4702 -234.2857
```

Thus, the slope is 19.4702 and the intercept is -234.2857 .

Another function, `polyval`, can then be used to compute a value using the coefficients. It has the general format:

```
>> y = polyval(a, x)
```

where `a` = the polynomial coefficients, and `x` = the best-fit value at `x`. For example,

```
>> y = polyval(a, 45)

y =
    641.8750
```

PROBLEMS

12.1 Given the data

8.8	9.5	9.8	9.4	10.0
9.4	10.1	9.2	11.3	9.4
10.0	10.4	7.9	10.4	9.8
9.8	9.5	8.9	8.8	10.6
10.1	9.5	9.6	10.2	8.9

Determine (a) the mean, (b) the standard deviation, (c) variance, and (d) the coefficient of variation.

12.2 Construct a histogram from the data from Prob. 12.1. Use a range from 7.5 to 11.5 with intervals of 0.5.

12.3 Given the data

28.65	26.55	26.65	27.65	27.35	28.35	26.85
28.65	29.65	27.85	27.05	28.25	28.85	26.75
27.65	28.45	28.65	28.45	31.65	26.35	27.75
29.25	27.65	28.65	27.65	28.55	27.65	27.25

Determine (a) the mean, (b) the standard deviation, (c) variance, and (d) the coefficient of variation. (e) Construct a histogram. Use a range from 26 to 32 with increments of 0.5. (f) Assuming that the distribution is normal and that your estimate of the standard deviation is valid, compute the range (i.e., the lower and the upper values) that encompasses 68% of the readings. Determine whether this is a valid estimate for the data in this problem.

12.4 Using the same approach as was employed to derive Eqs. (12.15) and (12.16), derive the least-squares fit of the following model:

$$y = a_1x + e$$

That is, determine the slope that results in the least-squares fit for a straight line with a zero intercept.

12.5 The acceleration due to gravity at an altitude y above the surface of the earth is given by

$y, \text{ m}$	0	20,000	40,000	60,000	80,000
$g, \text{ m/s}^2$	9.8100	9.7487	9.6879	9.6278	9.5682

Use linear regression to compute g at $y = 55,000$ m.

12.6 The following data was gathered to determine the relationship between pressure and temperature of a fixed volume of 1 kg of nitrogen. The volume is 10 m^3 .

$T, ^\circ\text{C}$	-20	0	20	40
$P, \text{ N/m}^2$	7500	8104	8700	9300
$T, ^\circ\text{C}$	50	70	100	120
$P, \text{ N/m}^2$	9620	10,200	11,200	11,700

Employ the ideal gas law $pV = nRT$ to determine R on the basis of this data. Note that for the law, T must be expressed in kelvins.

12.7 The data shown here was obtained from a creep test performed at room temperature on a wire composed of 40% tin, 60% lead, and solid core solder. This was done by measuring the increase in strain over time while a constant load was applied to a test specimen. Using a linear regression method, find (a) the equation of the line that best fits the data and (b) the r^2 value. Plot your results. Does the line pass through the origin? That is, at time zero, should there be any strain? If the line does not pass through the origin, force it to do so. Does this new line represent the data trend? Suggest a new equation that satisfies zero strain at zero time and also represents the data trend well.

PROBLEMS

219

Time min	Strain %	Time min	Strain %
0.083	0.099	5.589	0.365
0.584	0.130	6.089	0.387
1.084	0.160	6.590	0.409
1.585	0.184	7.090	0.431
2.085	0.204	7.590	0.452
2.586	0.229	8.091	0.474
3.086	0.252	8.591	0.497
3.587	0.262	9.092	0.517
4.087	0.295	9.592	0.540
4.588	0.319	10.095	0.562
5.088	0.342		

12.8 Beyond the examples in Fig. 12.10, there are other basic models that can be linearized using transformations. For example,

$$y = \alpha_4 x e^{\beta_4 x}$$

Linearize this model and use it to estimate α_4 and β_4 based on the following data. Develop a plot of your fit along with the data.

0.1	0.2	0.4	0.6	0.9	1.3	1.5	1.7	1.8
0.75	1.25	1.45	1.25	0.85	0.55	0.35	0.28	0.18

12.9 Fit a power model to the data from Table 12.1, but use natural logarithms to perform the transformations.

12.10 The concentration of *E. coli* bacteria in a swimming area is monitored after a storm:

(hr)	4	8	12	16	20	24
(CFU/100 mL)	1590	1320	1000	900	650	560

The time is measured in hours following the end of the storm, and the unit CFU is a "colony forming unit." Use this data to estimate (a) the concentration at the end of the storm ($t = 0$) and (b) the time at which the concentration will reach 200 CFU/100 mL. Note that your choice of model should be consistent with the fact that negative concentrations are impossible.

12.11 Rather than using the base- e exponential model (Eq. 12.22), a common alternative is to use a base-10 model:

$$y = \alpha_5 10^{\beta_5 x}$$

When used for curve fitting, this equation yields identical results to the base- e version, but the value of the exponent parameter (β_5) will differ from that estimated with Eq. 12.22

(β_1). Use the base-10 version to solve Prob. 12.10. In addition, develop a formulation to relate β_1 to β_5 .

12.12 On average the surface area A of human beings is related to weight W and height H . Measurements on a number of individuals of height 180 cm and different weights (kg) give values of A (m²) in the following table:

(kg)	70	75	77	80	82	84	87	90
(m ²)	2.10	2.12	2.15	2.20	2.22	2.23	2.26	2.30

Show that a power law $A = aW^b$ fits these data reasonably well. Evaluate the constants a and b , and predict what the surface area is for a 95-kg person.

12.13 Find a power law relationship between an animal's mass and its metabolism. The following table gives animals' masses in kg and metabolism rates in kcal/d:

Animal	Mass (kg)	Metabolism (kcal/d)
Cow	300	5600
Human	70	1700
Sheep	60	1100
Hen	2	100
Rat	0.3	30

12.14 The creep rate ($\dot{\epsilon}$, the time rate at which strain increases) and stress data (σ) shown here were also obtained from the testing procedure described in Prob. 12.7. Using a power law curve fit ($\dot{\epsilon} = B\sigma^m$), find the value of B and m . Plot your results using a log-log scale.

Creep rate, min ⁻¹	0.0004	0.0011	0.0031
Stress, MPa	5.775	8.577	12.555

12.15 It is a common practice when examining a fluid's viscous behavior to plot the shear rate (velocity gradient):

$$\frac{dv}{dy} \equiv \dot{\gamma}$$

on the abscissa versus shear stress (τ) on the ordinate. When a fluid has a straight line behavior between these two variables, it is called a *Newtonian fluid*, and the resulting relationship is

$$\tau = \mu \dot{\gamma}$$

where μ is the fluid viscosity. Many common fluids follow this behavior such as water, milk, and oil. Fluids that do not behave in this way are called *non-Newtonian*. Some examples of non-Newtonian fluids are (see Fig. P12.15):

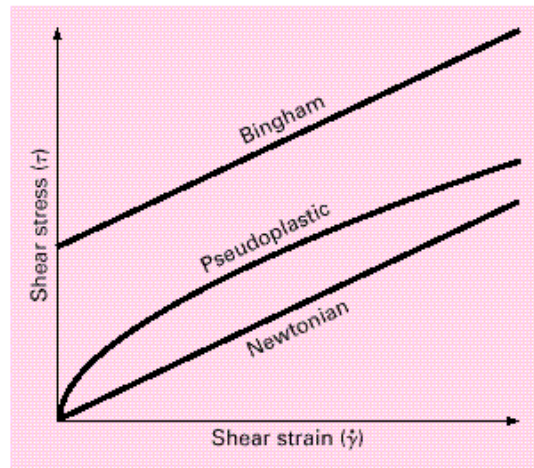


FIGURE P12.15

For *Bingham plastics*, there is a yield stress τ_y that must be overcome before flow will begin:

$$\tau = \tau_y + \mu \dot{\gamma}$$

A common example is toothpaste.

For *pseudoplastics*, the shear stress is raised to the power n :

$$\tau = \mu \dot{\gamma}^n$$

Common examples are yogurt and shampoo.

The following data shows the relationship between the shear stress τ and the shear strain rate $\dot{\gamma}$ for a Bingham plastic fluid. The yield stress τ_y is the amount of stress that must be exceeded before flow begins. Find the viscosity μ (slope), τ_y , and the r^2 using a regression method.

Stress, N/m ²	3.57	3.89	4.96	5.62	6.13
Shear strain rate $\dot{\gamma}$, 1/s	1	2	3	4	5

12.16 The relationship between stress τ and the shear strain rate $\dot{\gamma}$ for a pseudoplastic fluid (see Prob. 12.15) can be expressed by the equation $\tau = \mu \dot{\gamma}^n$. The following data comes from a 0.5% hydroxethylcellulose in water solution. Using a power law fit, find the values of μ and n :

Shear strain rate $\dot{\gamma}$, 1/s	50	70	90	110	130
Shear stress τ , N/m ²	5.99	7.45	8.56	9.09	10.25

General Linear Least-Squares and Nonlinear Regression

CHAPTER OBJECTIVES

This chapter takes the concept of fitting a straight line and extends it to (a) fitting a polynomial and (b) fitting a variable that is a linear function of two or more independent variables. We will then show how such applications can be generalized and applied to a broader group of problems. Finally, we will show how optimization techniques can be used to implement nonlinear regression. Specific objectives and topics covered are

- Knowing how to implement polynomial regression.
- Knowing how to implement multiple linear regression.
- Understanding the formulation of the general linear least-squares model.
- Understanding how the general linear least-squares model can be solved on MATLAB with either the normal equations or with left division.
- Understanding how to implement nonlinear regression with optimization techniques.

13.1 POLYNOMIAL REGRESSION

In Chap. 12, a procedure was developed to derive the equation of a straight line using the least-squares criterion. Some data, although exhibiting a marked pattern such as seen in Fig. 13.1, is poorly represented by a straight line. For these cases, a curve would be better suited to fit the data. As discussed in Chap. 12, one method to accomplish this objective is to use transformations. Another alternative is to fit polynomials to the data using *polynomial regression*.

The least-squares procedure can be readily extended to fit the data to a higher-order polynomial. For example, suppose that we fit a second-order polynomial or quadratic:

$$y = a_0 + a_1x + a_2x^2 + e \quad (13.1)$$

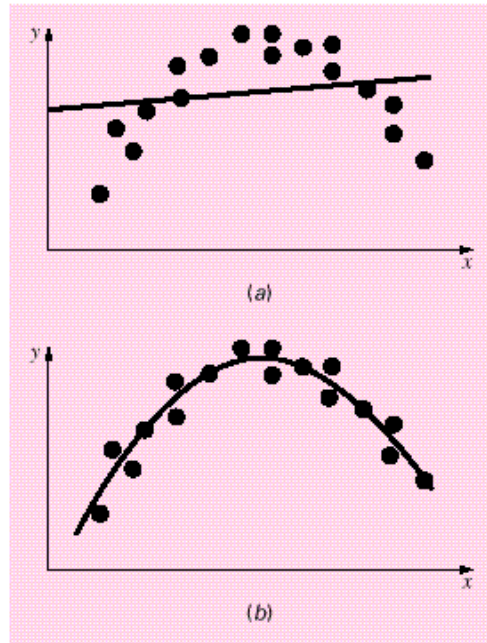


FIGURE 13.1

(· ·) Data that is ill-suited for linear least-squares regression. (—) Indication that a parabola is preferable.

For this case the sum of the squares of the residuals is

$$S_r = \sum_{i=1}^n (y_i - a_0 - a_1 x_i - a_2 x_i^2)^2 \quad (13.2)$$

To generate the least-squares fit, we take the derivative of Eq. (13.2) with respect to each of the unknown coefficients of the polynomial, as in

$$\begin{aligned} \frac{\partial S_r}{\partial a_0} &= -2 \sum (y_i - a_0 - a_1 x_i - a_2 x_i^2) \\ \frac{\partial S_r}{\partial a_1} &= -2 \sum x_i (y_i - a_0 - a_1 x_i - a_2 x_i^2) \\ \frac{\partial S_r}{\partial a_2} &= -2 \sum x_i^2 (y_i - a_0 - a_1 x_i - a_2 x_i^2) \end{aligned}$$

These equations can be set equal to zero and rearranged to develop the following set of normal equations:

$$\begin{aligned} (n)a_0 + (\sum x_i) a_1 + (\sum x_i^2) a_2 &= \sum y_i \\ (\sum x_i) a_0 + (\sum x_i^2) a_1 + (\sum x_i^3) a_2 &= \sum x_i y_i \\ (\sum x_i^2) a_0 + (\sum x_i^3) a_1 + (\sum x_i^4) a_2 &= \sum x_i^2 y_i \end{aligned}$$

13.1 POLYNOMIAL REGRESSION

223

where all summations are from $i = 1$ through n . Note that the preceding three equations are linear and have three unknowns: a_0 , a_1 , and a_2 . The coefficients of the unknowns can be calculated directly from the observed data.

For this case, we see that the problem of determining a least-squares second-order polynomial is equivalent to solving a system of three simultaneous linear equations. The two-dimensional case can be easily extended to an m th-order polynomial as in

$$y = a_0 + a_1x + a_2x^2 + \cdots + a_mx^m + e$$

The foregoing analysis can be easily extended to this more general case. Thus, we can recognize that determining the coefficients of an m th-order polynomial is equivalent to solving a system of $m + 1$ simultaneous linear equations. For this case, the standard error is formulated as

$$s_{y/x} = \sqrt{\frac{S_r}{n - (m + 1)}} \quad (13.3)$$

This quantity is divided by $n - (m + 1)$ because $(m + 1)$ data-derived coefficients— a_0, a_1, \dots, a_m —were used to compute S_r ; thus, we have lost $m + 1$ degrees of freedom. In addition to the standard error, a coefficient of determination can also be computed for polynomial regression with Eq. (12.20).

EXAMPLE 13.1 Polynomial Regression

Problem Statement. Fit a second-order polynomial to the data in the first two columns of Table 13.1.

TABLE 13.1 Computations for an error analysis of the quadratic least-squares fit.

i	x_i	$(x_i - \bar{x})^2$	$(y_i - \bar{y} - a_1(x_i - \bar{x}) - a_2(x_i - \bar{x})^2)^2$
0	2.1	544.44	0.14332
1	7.7	314.47	1.00286
2	13.6	140.03	1.08160
3	27.2	3.12	0.80487
4	40.9	239.22	0.61959
5	61.1	1272.11	0.09434
Σ	152.6	2513.39	3.74657

Solution. The following can be computed from the data:

$$\begin{array}{lll}
 m = 2 & \sum x_i = 15 & \sum x_i^4 = 979 \\
 n = 6 & \sum y_i = 152.6 & \sum x_i y_i = 585.6 \\
 \bar{x} = 2.5 & \sum x_i^2 = 55 & \sum x_i^2 y_i = 2488.8 \\
 \bar{y} = 25.433 & \sum x_i^3 = 225 &
 \end{array}$$

Therefore, the simultaneous linear equations are

$$\begin{bmatrix} 6 & 15 & 55 \\ 15 & 55 & 225 \\ 55 & 225 & 979 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 152.6 \\ 585.6 \\ 2488.8 \end{bmatrix}$$

These equations can be solved to evaluate the coefficients. For example, using MATLAB:

```
>> N = [6 15 55;15 55 225;55 225 979];
>> r = [152.6 585.6 2488.8];
>> a = N\r
a =
    2.4786
    2.3593
    1.8607
```

Therefore, the least-squares quadratic equation for this case is

$$y = 2.4786 + 2.3593x + 1.8607x^2$$

The standard error of the estimate based on the regression polynomial is [Eq. (13.3)]

$$s_{y/x} = \sqrt{\frac{3.74657}{6 - (2 + 1)}} = 1.1175$$

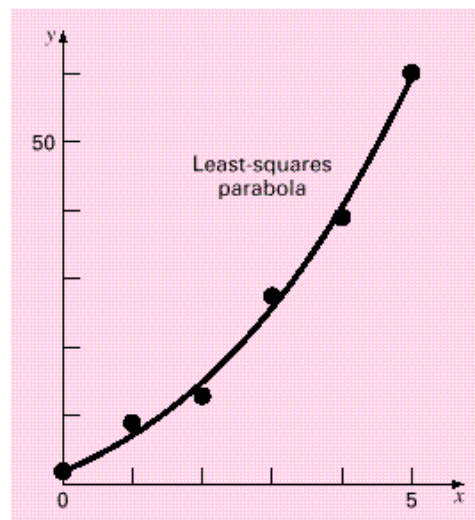
The coefficient of determination is

$$r^2 = \frac{2513.39 - 3.74657}{2513.39} = 0.99851$$

and the correlation coefficient is $r = 0.99925$.

FIGURE 13.2

Fit of a second-order polynomial.



These results indicate that 99.851 percent of the original uncertainty has been explained by the model. This result supports the conclusion that the quadratic equation represents an excellent fit, as is also evident from Fig. 13.2.

13.2 MULTIPLE LINEAR REGRESSION

Another useful extension of linear regression is the case where y is a linear function of two or more independent variables. For example, y might be a linear function of x_1 and x_2 , as in

$$y = a_0 + a_1x_1 + a_2x_2 + e$$

Such an equation is particularly useful when fitting experimental data where the variable being studied is often a function of two other variables. For this two-dimensional case, the regression “line” becomes a “plane” (Fig. 13.3).

As with the previous cases, the “best” values of the coefficients are determined by formulating the sum of the squares of the residuals:

$$S_r = \sum_{i=1}^n (y_i - a_0 - a_1x_{1,i} - a_2x_{2,i})^2 \quad (13.4)$$

and differentiating with respect to each of the unknown coefficients:

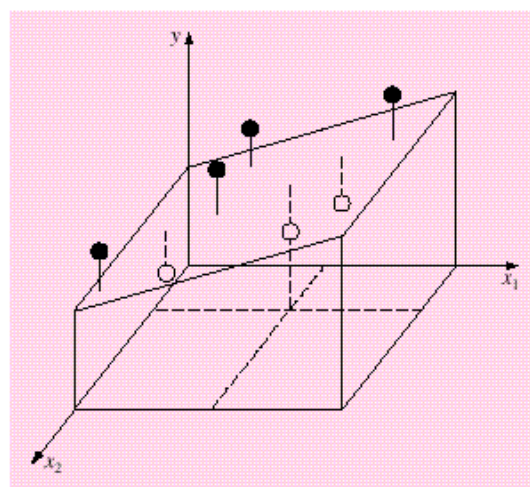
$$\frac{\partial S_r}{\partial a_0} = -2 \sum (y_i - a_0 - a_1x_{1,i} - a_2x_{2,i})$$

$$\frac{\partial S_r}{\partial a_1} = -2 \sum x_{1,i} (y_i - a_0 - a_1x_{1,i} - a_2x_{2,i})$$

$$\frac{\partial S_r}{\partial a_2} = -2 \sum x_{2,i} (y_i - a_0 - a_1x_{1,i} - a_2x_{2,i})$$

FIGURE 13.3

Graphical depiction of multiple linear regression where y is a linear function of x_1 and x_2 .



The coefficients yielding the minimum sum of the squares of the residuals are obtained by setting the partial derivatives equal to zero and expressing the result in matrix form as

$$\begin{bmatrix} n & \sum x_{1,i} & \sum x_{2,i} \\ \sum x_{1,i} & \sum x_{1,i}^2 & \sum x_{1,i}x_{2,i} \\ \sum x_{2,i} & \sum x_{1,i}x_{2,i} & \sum x_{2,i}^2 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} \sum y_i \\ \sum x_{1,i}y_i \\ \sum x_{2,i}y_i \end{Bmatrix} \quad (13.5)$$

EXAMPLE 13.2 Multiple Linear Regression

Problem Statement. The following data was calculated from the equation $y = 5 + 4x_1 - 3x_2$:

x_1	x_2	y
0	0	5
2	1	10
2.5	2	9
1	3	0
4	6	3
7	2	27

Use multiple linear regression to fit this data.

Solution. The summations required to develop Eq. (13.5) are computed in Table 13.2. Substituting them into Eq. (13.5) gives

$$\begin{bmatrix} 6 & 16.5 & 14 \\ 16.5 & 76.25 & 48 \\ 14 & 48 & 54 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} 54 \\ 243.5 \\ 100 \end{Bmatrix} \quad (13.6)$$

which can be solved for

$$a_0 = 5 \quad a_1 = 4 \quad a_2 = -3$$

which is consistent with the original equation from which the data was derived.

The foregoing two-dimensional case can be easily extended to m dimensions, as in

$$y = a_0 + a_1x_1 + a_2x_2 + \cdots + a_mx_m + e$$

TABLE 13.2 Computations required to develop the normal equations for Example 13.2.

	x_1	x_2	x_1^2	x_2^2	x_1x_2	y	xy
5	0	0	0	0	0	0	0
10	2	1	4	1	2	20	10
9	2.5	2	6.25	4	5	22.5	18
0	1	3	1	9	3	0	0
3	4	6	16	36	24	12	18
27	7	2	49	4	14	189	54
54	16.5	14	76.25	54	48	243.5	100

13.3 GENERAL LINEAR LEAST SQUARES

227

where the standard error is formulated as

$$s_{y/x} = \sqrt{\frac{S_r}{n - (m + 1)}}$$

and the coefficient of determination is computed as in Eq. (12.20).

Although there may be certain cases where a variable is linearly related to two or more other variables, multiple linear regression has additional utility in the derivation of power equations of the general form

$$y = a_0 x_1^{a_1} x_2^{a_2} \cdots x_m^{a_m}$$

Such equations are extremely useful when fitting experimental data. To use multiple linear regression, the equation is transformed by taking its logarithm to yield

$$\log y = \log a_0 + a_1 \log x_1 + a_2 \log x_2 + \cdots + a_m \log x_m$$

13.3 GENERAL LINEAR LEAST SQUARES

In the preceding pages, we have introduced three types of regression: simple linear, polynomial, and multiple linear. In fact, all three belong to the following general linear least-squares model:

$$y = a_0 z_0 + a_1 z_1 + a_2 z_2 + \cdots + a_m z_m + e \quad (13.7)$$

where z_0, z_1, \dots, z_m are $m + 1$ basis functions. It can easily be seen how simple linear and multiple linear regression fall within this model—that is, $z_0 = 1, z_1 = x_1, z_2 = x_2, \dots, z_m = x_m$. Further, polynomial regression is also included if the basis functions are simple monomials as in $z_0 = 1, z_1 = x, z_2 = x^2, \dots, z_m = x^m$.

Note that the terminology “linear” refers only to the model’s dependence on its parameters—that is, the a ’s. As in the case of polynomial regression, the functions themselves can be highly nonlinear. For example, the z ’s can be sinusoids, as in

$$y = a_0 + a_1 \cos(\omega x) + a_2 \sin(\omega x)$$

Such a format is the basis of Fourier analysis.

On the other hand, a simple-looking model such as

$$y = a_0(1 - e^{-a_1 x})$$

is truly nonlinear because it cannot be manipulated into the format of Eq. (13.7).

Equation (13.7) can be expressed in matrix notation as

$$\{y\} = [Z]\{a\} + \{e\} \quad (13.8)$$

where $[Z]$ is a matrix of the calculated values of the basis functions at the measured values of the independent variables:

$$[Z] = \begin{bmatrix} z_{01} & z_{11} & \cdots & z_{m1} \\ z_{02} & z_{12} & \cdots & z_{m2} \\ \vdots & \vdots & & \vdots \\ z_{0n} & z_{1n} & \cdots & z_{mn} \end{bmatrix}$$

where m is the number of variables in the model and n is the number of data points. Because $n \geq m + 1$, you should recognize that most of the time, $[Z]$ is not a square matrix.

The column vector $\{y\}$ contains the observed values of the dependent variable:

$$\{y\}^T = [y_1 \quad y_2 \quad \cdots \quad y_n]$$

The column vector $\{a\}$ contains the unknown coefficients:

$$\{a\}^T = [a_0 \quad a_1 \quad \cdots \quad a_m]$$

and the column vector $\{e\}$ contains the residuals:

$$\{e\}^T = [e_1 \quad e_2 \quad \cdots \quad e_n]$$

The sum of the squares of the residuals for this model can be defined as

$$S_r = \sum_{i=1}^n \left(y_i - \sum_{j=0}^m a_j z_{ji} \right)^2 \quad (13.9)$$

This quantity can be minimized by taking its partial derivative with respect to each of the coefficients and setting the resulting equation equal to zero. The outcome of this process is the normal equations that can be expressed concisely in matrix form as

$$[[Z]^T [Z]] \{a\} = \{[Z]^T \{y\}\} \quad (13.10)$$

It can be shown that Eq. (13.10) is, in fact, equivalent to the normal equations developed previously for simple linear, polynomial, and multiple linear regression.

The coefficient of determination and the standard error can also be formulated in terms of matrix algebra. Recall that r^2 is defined as

$$r^2 = \frac{S_t - S_r}{S_t} = 1 - \frac{S_r}{S_t}$$

Substituting the definitions of S_r and S_t gives

$$r^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

where \hat{y} = the prediction of the least-squares fit. The residuals between the best fit curve and the data, $y_i - \hat{y}$, can be expressed in vector form as

$$\{y\} - [Z]\{a\}$$

Matrix algebra can then be used to manipulate this vector to compute both the coefficient of determination and the standard error of the estimate as illustrated in the following example.

EXAMPLE 13.3 Polynomial Regression with MATLAB

Problem Statement. Repeat Example 13.1, but use matrix operations as described in this section.

Solution. First, enter the data to be fit

```
>> x = [0 1 2 3 4 5]';
>> y = [2.1 7.7 13.6 27.2 40.9 61.1]';
```

13.3 GENERAL LINEAR LEAST SQUARES

229

Next, create the $[Z]$ matrix:

```
>> Z = [ones(size(x)) x x.^2]
```

```
Z =  
     1     0     0  
     1     1     1  
     1     2     4  
     1     3     9  
     1     4    16  
     1     5    25
```

We can verify that $[Z]^T[Z]$ results in the coefficient matrix for the normal equations:

```
>> Z'*Z  
  
ans =  
     6     15     55  
    15     55    225  
    55    225    979
```

We can solve for the coefficients of the least-squares quadratic by implementing Eq. (13.10):

```
>> a = (Z'*Z)\(Z'*y)  
  
ans =  
    2.4786  
    2.3593  
    1.8607
```

In order to compute r^2 and $s_{y/x}$, first compute the sum of the squares of the residuals:

```
>> Sr = sum((y-Z*a).^2)  
  
Sr =  
    3.7466
```

Then r^2 can be computed as

```
>> r2 = 1-Sr/sum((y-mean(y)).^2)  
  
r2 =  
    0.9985
```

and $s_{y/x}$ can be computed as

```
>> syx = sqrt(Sr/(length(x)-length(a)))  
  
syx =  
    1.1175
```

Our primary motivation for the foregoing has been to illustrate the unity among the three approaches and to show how they can all be expressed simply in the same matrix notation. It also sets the stage for the next section where we will gain some insights into the preferred strategies for solving Eq. (13.10). The matrix notation will also have relevance when we turn to nonlinear regression in Section 13.5.

13.4 QR FACTORIZATION AND THE BACKSLASH OPERATOR

Generating a best fit by solving the normal equations is widely used and certainly adequate for many curve-fitting applications in engineering and science. It must be mentioned, however, that the normal equations can be ill-conditioned and hence sensitive to round-off errors.

Two more advanced methods, *QR factorization* and *singular value decomposition*, are more robust in this regard. Although the description of these methods is beyond the scope of this text, we mention them here because they can be implemented with MATLAB.

Further, QR factorization is automatically used in two simple ways within MATLAB. First, for cases where you want to fit a polynomial, the built-in `polyfit` function automatically uses QR factorization to obtain its results.

Second, the general linear least squares problem can be directly solved with the backslash operator. Recall that the general model is formulated as Eq. (13.8)

$$\{y\} = [Z]\{a\} \quad (13.11)$$

In Section 9.4, we used left division with the backslash operator to solve systems of linear algebraic equations where the number of equations equals the number of unknowns ($n = m$). For Eq. (13.8) as derived from general least squares, the number of equations is greater than the number of unknowns ($n > m$). Such systems are said to be *overdetermined*. When MATLAB senses that you want to solve such systems with left division, it automatically uses QR factorization to obtain the solution. The following example illustrates how this is done.

EXAMPLE 13.4 Implementing Polynomial Regression with `polyfit` and Left Division

Problem Statement. Repeat Example 13.3, but use the built-in `polyfit` function and left division to calculate the coefficients.

Solution. As in Example 13.3, the data can be entered and used to create the $[Z]$ matrix as in

```
>> x = [0 1 2 3 4 5]';  
>> y = [2.1 7.7 13.6 27.2 40.9 61.1]';  
>> Z = [ones(size(x)) x x.^2];
```

The `polyfit` function can be used to compute the coefficients:

```
>> a = polyfit(x,y,2)  
a =  
    1.8607    2.3593    2.4786
```

The same result can also be calculated using the backslash:

```
>> a = Z\y  
a =  
    2.4786  
    2.3593  
    1.8607
```

As just stated, both these results are obtained automatically with QR factorization.

13.5 NONLINEAR REGRESSION

There are many cases in engineering and science where nonlinear models must be fit to data. In the present context, these models are defined as those that have a nonlinear dependence on their parameters. For example,

$$y = a_0(1 - e^{-a_1 x}) + e \quad (13.12)$$

This equation cannot be manipulated so that it conforms to the general form of Eq. (13.7).

As with linear least squares, nonlinear regression is based on determining the values of the parameters that minimize the sum of the squares of the residuals. However, for the nonlinear case, the solution must proceed in an iterative fashion.

There are techniques expressly designed for nonlinear regression. For example, the Gauss-Newton method uses a Taylor series expansion to express the original nonlinear equation in an approximate, linear form. Then least-squares theory can be used to obtain new estimates of the parameters that move in the direction of minimizing the residual. Details on this approach are provided elsewhere (Chapra and Canale, 2002).

An alternative is to use optimization techniques to directly determine the least-squares fit. For example, Eq. (13.12) can be expressed as an objective function to compute the sum of the squares:

$$f(a_0, a_1) = \sum_{i=1}^n [y_i - a_0(1 - e^{-a_1 x_i})]^2 \quad (13.13)$$

An optimization routine can then be used to determine the values of a_0 and a_1 that minimize the function.

MATLAB's `fminsearch` function can be used for this purpose. It has the general syntax

$$[\cdot, \dots] = \text{fminsearch}(\dots, \dots, \dots, \dots, \dots)$$

where \cdot = a vector of the values of the parameters that minimize the function \dots, \dots = the value of the function at the minimum, \dots = a vector of the initial guesses for the parameters, \dots = a structure containing values of the optimization parameters as created with the `optimset` function (recall Section 6.4), and \dots, \dots , etc. = additional arguments that are passed to the objective function. Note that if \dots is omitted, MATLAB uses default values that are reasonable for most problems. If you would like to pass additional arguments (\dots, \dots, \dots), but do not want to set the \dots , use empty brackets `[]` as a place holder.

EXAMPLE 13.4 Nonlinear Regression with MATLAB

Problem Statement. Recall that in Example 12.4, we fit the power model to data from Table 12.1 by linearization using logarithms. This yielded the model:

$$F = 0.2741v^{1.9842}$$

Repeat this exercise, but use nonlinear regression. Employ initial guesses of 1 for the coefficients.

Solution. First, an M-file function must be created to compute the sum of the squares. The following file, called `fSSR.m`, is set up for the power equation:

```
function f = fSSR(a,xm,ym)
yp = a(1)*xm.^a(2);
f = sum((ym-yp).^2);
```

In command mode, the data can be entered as

```
>> x = [10 20 30 40 50 60 70 80];
>> y = [25 70 380 550 610 1220 830 1450];
```

The minimization of the function is then implemented by

```
>> fminsearch(@fSSR, [1, 1], [], x, y)
ans =
    2.5384    1.4359
```

The best-fit model is therefore

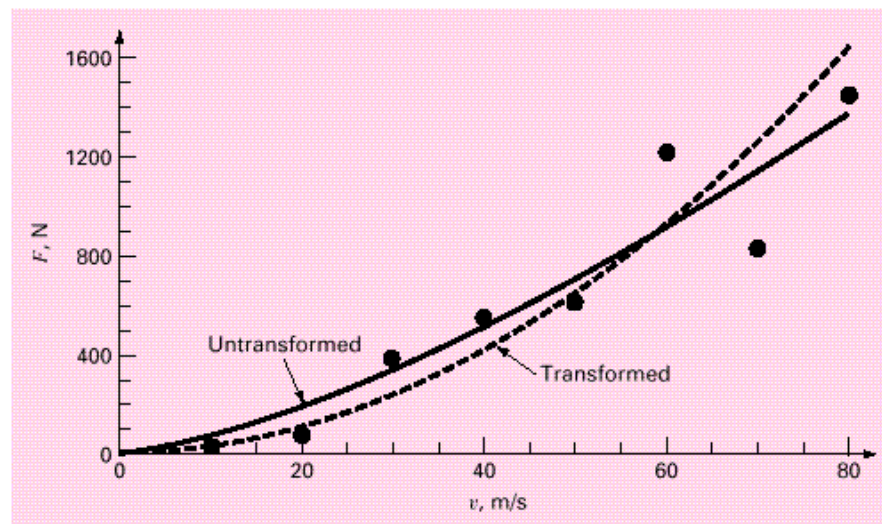
$$F = 2.5384v^{1.4359}$$

Both the original transformed fit and the present version are displayed in Fig. 13.4. Note that although the model coefficients are very different, it is difficult to judge which fit is superior based on inspection of the plot.

This example illustrates how different best-fit equations result when fitting the same model using nonlinear regression versus linear regression employing transformations. This is because the former minimizes the residuals of the original data whereas the latter minimizes the residuals of the transformed data.

FIGURE 13.4

Comparison of transformed and untransformed model fits for force versus velocity data from Table 12.1.



PROBLEMS

233

PROBLEMS

13.1 Fit a parabola to the data from Table 12.1. Determine the r^2 for the fit and comment on the efficacy of the result.

13.2 Using the same approach as was employed to derive Eqs. (12.15) and (12.16), derive the least-squares fit of the following model:

$$y = a_1x + a_2x^2 + e$$

That is, determine the coefficients that results in the least-squares fit for a second-order polynomial with a zero intercept. Test the approach by using it to fit the data from Table 12.1.

13.3 Fit a cubic polynomial to the following data:

3	4	5	7	8	9	11	12
1.6	3.6	4.4	3.4	2.2	2.8	3.8	4.6

Along with the coefficients, determine r^2 and $s_{y/x}$.

13.4 Develop an M-file to implement polynomial regression. Pass the M-file two vectors holding the x and y values along with the desired order m . Test it by solving Prob. 13.3.

13.5 For the data from Table P13.5, use polynomial regression to derive a predictive equation for dissolved oxygen concentration as a function of temperature for the case where the chloride concentration is equal to zero. Employ a polynomial that is of sufficiently high order that the predictions match the number of significant digits displayed in the table.

13.6 Use multiple linear regression to derive a predictive equation for dissolved oxygen concentration as a function of

temperature and chloride based on the data from Table P13.5. Use the equation to estimate the concentration of dissolved oxygen for a chloride concentration of 15 g/L at $T = 12^\circ\text{C}$. Note that the true value is 9.09 mg/L. Compute the percent relative error for your prediction. Explain possible causes for the discrepancy.

13.7 As compared with the models from Probs. 13.5 and 13.6, a somewhat more sophisticated model that accounts for the effect of both temperature and chloride on dissolved oxygen saturation can be hypothesized as being of the form

$$0 = f_3(T) + f_1(c)$$

That is, a third-order polynomial in temperature and a linear relationship in chloride is assumed to yield superior results. Use the general linear least-squares approach to fit this model to the data in Table P13.5. Use the resulting equation to estimate the dissolved oxygen concentration for a chloride concentration of 15 g/L at $T = 12^\circ\text{C}$. Note that the true value is 9.09 mg/L. Compute the percent relative error for your prediction.

13.8 Use multiple linear regression to fit

1	0	1	1	2	2	3	3	4	4
2	0	1	2	1	2	1	2	1	2
3	15.1	17.9	12.7	25.6	20.5	35.1	29.7	45.4	40.2

Compute the coefficients, the standard error of the estimate, and the correlation coefficient.

13.9 The following data was collected for the steady flow of water in a concrete circular pipe:

TABLE P13.5 Dissolved oxygen concentration in water as a function of temperature ($^\circ\text{C}$) and chloride concentration (g/L).

Dissolved Oxygen (mg/L) for Temperature ($^\circ\text{C}$) and Concentration of Chloride (g/L)			
$T, ^\circ\text{C}$	0 g/L	10 g/L	20 g/L
0	14.6	12.9	11.4
5	12.8	11.3	10.3
10	11.3	10.1	8.96
15	10.1	9.03	8.08
20	9.09	8.17	7.35
25	8.26	7.46	6.73
30	7.56	6.85	6.20

Experiment	Diameter, m	Slope, m/m	Flow, m^3/s
1	0.3	0.001	0.04
2	0.6	0.001	0.24
3	0.9	0.001	0.69
4	0.3	0.01	0.13
5	0.6	0.01	0.82
6	0.9	0.01	2.38
7	0.3	0.05	0.31
8	0.6	0.05	1.95
9	0.9	0.05	5.66

Use multiple linear regression to fit the following model to this data:

$$Q = \alpha_0 D^{\alpha_1} S^{\alpha_2}$$

where Q = flow, D = diameter, and S = slope.

13.10 Three disease-carrying organisms decay exponentially in seawater according to the following model:

$$p(t) = Ae^{-1.5t} + Be^{-0.3t} + Ce^{-0.05t}$$

Estimate the initial concentration of each organism (A , B , and C) given the following measurements:

t	0.5	1	2	3	4	5	6	7	8	9
$p(t)$	7	5.2	3.8	3.2	2.5	2.1	1.8	1.5	1.2	1.1

13.11 The following model is used to represent the effect of solar radiation on the photosynthesis rate of aquatic plants:

$$P = P_m \frac{I}{I_{sat}} e^{-\frac{I}{I_{sat}} + 1}$$

where P = the photosynthesis rate ($\text{mg m}^{-3}\text{d}^{-1}$), P_m = the maximum photosynthesis rate ($\text{mg m}^{-3}\text{d}^{-1}$), I = solar radiation ($\mu\text{E m}^{-2}\text{s}^{-1}$), and I_{sat} = optimal solar radiation ($\mu\text{E m}^{-2}\text{s}^{-1}$). Use nonlinear regression to evaluate P_m and I_{sat} based on the following data:

I	50	80	130	200	250	350	450	550	700
P	99	177	202	248	229	219	173	142	72

13.12 In Prob. 12.8 we used transformations to linearize and fit the following model:

$$y = \alpha_4 x e^{\beta_4 x}$$

Use nonlinear regression to estimate α_4 and β_4 based on the following data. Develop a plot of your fit along with the data.

x	0.1	0.2	0.4	0.6	0.9	1.3	1.5	1.7	1.8
y	0.75	1.25	1.45	1.25	0.85	0.55	0.35	0.28	0.18

13.13 Enzymatic reactions are used extensively to characterize biologically mediated reactions. The following is an example of a model that is used to fit such reactions:

$$v_0 = \frac{k_m [S]^3}{K + [S]^3}$$

where v_0 = the initial rate of the reaction (M/s), $[S]$ = the substrate concentration (M), and k_m and K are parameters. The following data can be fit with this model:

$[S], \text{M}$	$v_0, \text{M/s}$
0.01	6.078×10^{-11}
0.05	7.595×10^{-9}
0.1	6.063×10^{-8}
0.5	5.788×10^{-6}
1	1.737×10^{-5}
5	2.423×10^{-5}
10	2.430×10^{-5}
50	2.431×10^{-5}
100	2.431×10^{-5}

- Use a transformation to linearize the model and evaluate the parameters. Display the data and the model fit on a graph.
- Perform the same evaluation as in (a) but use nonlinear regression.

Curve Fitting: Polynomial Interpolation

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to polynomial interpolation. Specific objectives and topics covered are

- Recognizing that evaluating polynomial coefficients with simultaneous equations is an ill-conditioned problem.
- Knowing how to evaluate polynomial coefficients and interpolate with MATLAB's `polyfit` and `polyval` functions.
- Knowing how to perform an interpolation with Newton's polynomial.
- Knowing how to perform an interpolation with a Lagrange polynomial.
- Knowing how to solve an inverse interpolation problem by recasting it as a roots problem.
- Appreciating the dangers of extrapolation.
- Recognizing that higher-order polynomials can manifest large oscillations.

YOU'VE GOT A PROBLEM

If we want to improve the velocity prediction for the free-falling bungee jumper, we might expand our model to account for other factors beyond mass and the drag coefficient. For example, the drag coefficient can itself be formulated as a function of other factors such as the area of the jumper and characteristics such as the air's density and viscosity.

Air density and viscosity are commonly presented in tabular form as a function of temperature. For example, Table 14.1 is reprinted from a popular fluid mechanics textbook (White, 1999).

Suppose that you desired the density at a temperature not included in the table. In such a case, you would have to interpolate. That is, you would have to estimate the value at the

TABLE 14.1 Density (ρ), dynamic viscosity (μ), and kinematic viscosity (ν) as a function of temperature ($^{\circ}\text{C}$) at 1 atm as reported by White (1999).

$T, ^{\circ}\text{C}$	$\rho, \text{kg/m}^3$	$\mu, \text{N} \cdot \text{s/m}^2$	$\nu, \text{m}^2/\text{s}$
-40	1.52	1.51×10^{-5}	0.99×10^{-5}
0	1.29	1.71×10^{-5}	1.33×10^{-5}
20	1.20	1.80×10^{-5}	1.50×10^{-5}
50	1.09	1.95×10^{-5}	1.79×10^{-5}
100	0.946	2.17×10^{-5}	2.30×10^{-5}
150	0.835	2.38×10^{-5}	2.85×10^{-5}
200	0.746	2.57×10^{-5}	3.45×10^{-5}
250	0.675	2.75×10^{-5}	4.08×10^{-5}
300	0.616	2.93×10^{-5}	4.75×10^{-5}
400	0.525	3.25×10^{-5}	6.20×10^{-5}
500	0.457	3.55×10^{-5}	7.77×10^{-5}

desired temperature based on the densities that bracket it. The simplest approach is to determine the equation for the straight line connecting the two adjacent values and use this equation to estimate the density at the desired intermediate temperature. Although such *linear interpolation* is perfectly adequate in many cases, error can be introduced when the data exhibits significant curvature. In this chapter, we will explore a number of different approaches for obtaining adequate estimates for such situations.

14.1 INTRODUCTION TO INTERPOLATION

You will frequently have occasion to estimate intermediate values between precise data points. The most common method used for this purpose is polynomial interpolation. The general formula for an $(n - 1)$ th-order polynomial can be written as

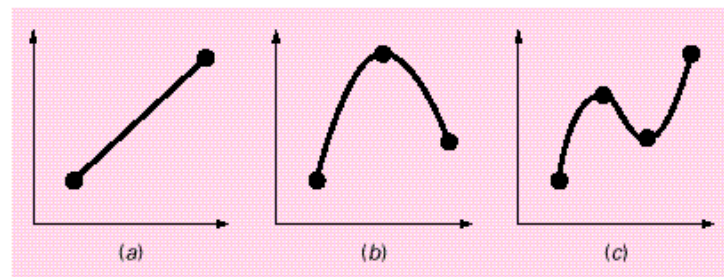
$$f(x) = a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1} \quad (14.1)$$

For n data points, there is one and only one polynomial of order $(n - 1)$ that passes through all the points. For example, there is only one straight line (i.e., a first-order polynomial) that connects two points (Fig. 14.1a). Similarly, only one parabola connects a set of three points (Fig. 14.1b). *Polynomial interpolation* consists of determining the unique $(n - 1)$ th-order polynomial that fits n data points. This polynomial then provides a formula to compute intermediate values.

Before proceeding, we should note that MATLAB represents polynomial coefficients in a different manner than Eq. (14.1). Rather than using increasing powers of x , it uses decreasing powers as in

$$f(x) = p_1x^{n-1} + p_2x^{n-2} + \cdots + p_{n-1}x + p_n \quad (14.2)$$

To be consistent with MATLAB, we will adopt this scheme in the following section.

**FIGURE 14.1**

Examples of interpolating polynomials: (·) first-order (linear) connecting two points, (·) second-order (quadratic or parabolic) connecting three points, and (·) third-order (cubic) connecting four points.

14.1.1 Determining Polynomial Coefficients

A straightforward way for computing the coefficients of Eq. (14.2) is based on the fact that n data points are required to determine the n coefficients. As in the following example, this allows us to generate n linear algebraic equations that we can solve simultaneously for the coefficients.

EXAMPLE 14.1 Determining Polynomial Coefficients with Simultaneous Equations

Problem Statement. Suppose that we want to determine the coefficients of the parabola, $f(x) = p_1x^2 + p_2x + p_3$, that passes through the last three density values from Table 14.1:

$$x_1 = 300 \quad f(x_1) = 0.616$$

$$x_2 = 400 \quad f(x_2) = 0.525$$

$$x_3 = 500 \quad f(x_3) = 0.457$$

Each of these pairs can be substituted into Eq. (14.2) to yield a system of three equations:

$$0.616 = p_1(300)^2 + p_2(300) + p_3$$

$$0.525 = p_1(400)^2 + p_2(400) + p_3$$

$$0.457 = p_1(500)^2 + p_2(500) + p_3$$

or in matrix form:

$$\begin{bmatrix} 90,000 & 300 & 1 \\ 160,000 & 400 & 1 \\ 250,000 & 500 & 1 \end{bmatrix} \begin{Bmatrix} p_1 \\ p_2 \\ p_3 \end{Bmatrix} = \begin{Bmatrix} 0.616 \\ 0.525 \\ 0.457 \end{Bmatrix}$$

Thus, the problem reduces to solving three simultaneous linear algebraic equations for the three unknown coefficients. A simple MATLAB session can be used to obtain the

solution:

```
>> format long  
>> A = [90000 300 1;160000 400 1;250000 500 1];  
>> b = [0.616 0.525 0.457]';  
>> p = A\b
```

```
p =  
 0.00000115000000  
-0.00171500000000  
 1.02700000000000
```

Thus, the parabola that passes exactly through the three points is

$$f(x) = 0.00000115x^2 - 0.001715x + 1.027$$

This polynomial then provides a means to determine intermediate points. For example, the value of density at a temperature of 350 °C can be calculated as

$$f(350) = 0.00000115(350)^2 - 0.001715(350) + 1.027 = 0.567625$$

Although the approach in Example 14.1 provides an easy way to perform interpolation, it has a serious deficiency. To understand this flaw, notice that the coefficient matrix in Example 14.1 has a decided structure. This can be seen clearly by expressing it in general terms:

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{Bmatrix} p_1 \\ p_2 \\ p_3 \end{Bmatrix} = \begin{Bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{Bmatrix} \quad (14.3)$$

Coefficient matrices of this form are referred to as *Vandermonde matrices*. Such matrices are very ill-conditioned. That is, their solutions are very sensitive to round-off errors. This can be illustrated by using MATLAB to compute the condition number for the coefficient matrix from Example 14.1 as

```
>> cond(A)  
  
ans =  
 5.8932e+006
```

This condition number, which is quite large for a 3×3 matrix, implies that about six digits of the solution would be questionable. The ill-conditioning becomes even worse as the number of simultaneous equations becomes larger.

As a consequence, there are alternative approaches that do not manifest this shortcoming. In this chapter, we will also describe two alternatives that are well-suited for computer implementation: the Newton and the Lagrange polynomials. Before doing this, however, we will first briefly review how the coefficients of the interpolating polynomial can be estimated directly with MATLAB's built-in functions.

14.1.2 MATLAB Functions: `polyfit` and `polyval`

Recall from Section 12.4.2, that the `polyfit` function can be used to perform polynomial regression. In such applications, the number of data points is greater than the number of coefficients being estimated. Consequently, the least-squares fit line does not necessarily pass through any of the points, but rather follows the general trend of the data.

For the case where the number of data points equals the number of coefficients, `polyfit` performs interpolation. That is, it returns the coefficients of the polynomial that pass directly through the data points. For example, it can be used to determine the coefficients of the parabola that passes through the last three density values from Table 14.1:

```
>> format long
>> T = [300 400 500];
>> density = [0.616 0.525 0.457];
>> p = polyfit(T,density,2)

p =
    0.00000115000000   -0.00171500000000    1.02700000000000
```

We can then use the `polyval` function to perform an interpolation as in

```
>> d = polyval(p,350)

d =
    0.56762500000000
```

These results are the same as those obtained previously in Example 14.1 with simultaneous equations.

14.2 NEWTON INTERPOLATING POLYNOMIAL

There are a variety of alternative forms for expressing an interpolating polynomial beyond the familiar format of Eq. (14.2). Newton's divided-difference interpolating polynomial is among the most popular and useful forms. Before presenting the general equation, we will introduce the first- and second-order versions because of their simple visual interpretation.

14.2.1 Linear Interpolation

The simplest form of interpolation is to connect two data points with a straight line. This technique, called *linear interpolation*, is depicted graphically in Fig. 14.2. Using similar triangles,

$$\frac{f_1(x) - f(x_1)}{x - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \quad (14.4)$$

which can be rearranged to yield

$$f_1(x) = f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1) \quad (14.5)$$

which is the *Newton linear-interpolation formula*. The notation $f_1(x)$ designates that this is a first-order interpolating polynomial. Notice that besides representing the slope of the

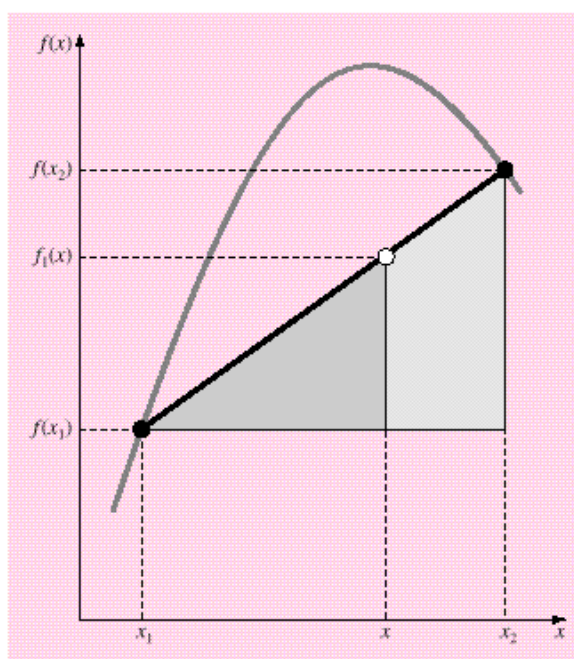


FIGURE 14.2

Graphical depiction of linear interpolation. The shaded areas indicate the similar triangles used to derive the Newton linear-interpolation formula [Eq. (14.5)].

line connecting the points, the term $[f(x_2) - f(x_1)]/(x_2 - x_1)$ is a finite-divided-difference approximation of the first derivative [recall Eq. (4.20)]. In general, the smaller the interval between the data points, the better the approximation. This is due to the fact that, as the interval decreases, a continuous function will be better approximated by a straight line. This characteristic is demonstrated in the following example.

EXAMPLE 14.2 Linear Interpolation

Problem Statement. Estimate the natural logarithm of 2 using linear interpolation. First, perform the computation by interpolating between $\ln 1 = 0$ and $\ln 6 = 1.791759$. Then, repeat the procedure, but use a smaller interval from $\ln 1$ to $\ln 4$ (1.386294). Note that the true value of $\ln 2$ is 0.6931472.

Solution. We use Eq. (14.5) from $x_1 = 1$ to $x_2 = 6$ to give

$$f_1(2) = 0 + \frac{1.791759 - 0}{6 - 1}(2 - 1) = 0.3583519$$

which represents an error of $\epsilon_f = 48.3\%$. Using the smaller interval from $x_1 = 1$ to $x_2 = 4$ yields

$$f_1(2) = 0 + \frac{1.386294 - 0}{4 - 1}(2 - 1) = 0.4620981$$

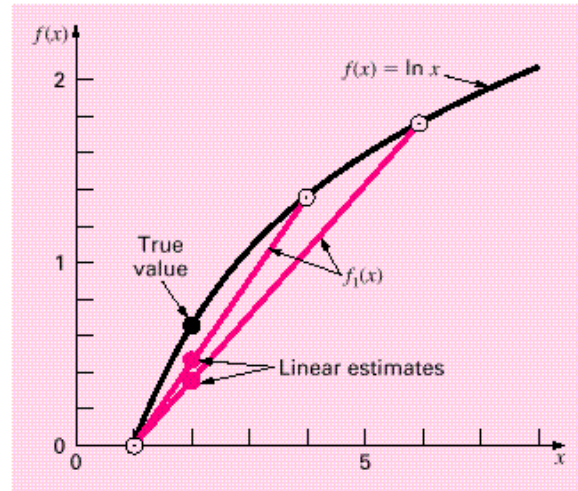


FIGURE 14.3

Two linear interpolations to estimate $\ln 2$. Note how the smaller interval provides a better estimate.

Thus, using the shorter interval reduces the percent relative error to $\varepsilon_t = 33.3\%$. Both interpolations are shown in Fig. 14.3, along with the true function.

14.2.2 Quadratic Interpolation

The error in Example 14.2 resulted from approximating a curve with a straight line. Consequently, a strategy for improving the estimate is to introduce some curvature into the line connecting the points. If three data points are available, this can be accomplished with a second-order polynomial (also called a quadratic polynomial or a parabola). A particularly convenient form for this purpose is

$$f_2(x) = b_1 + b_2(x - x_1) + b_3(x - x_1)(x - x_2) \quad (14.6)$$

A simple procedure can be used to determine the values of the coefficients. For b_1 , Eq. (14.6) with $x = x_1$ can be used to compute

$$b_1 = f(x_1) \quad (14.7)$$

Equation (14.7) can be substituted into Eq. (14.6), which can be evaluated at $x = x_2$ for

$$b_2 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \quad (14.8)$$

Finally, Eqs. (14.7) and (14.8) can be substituted into Eq. (14.6), which can be evaluated at $x = x_3$ and solved (after some algebraic manipulations) for

$$b_3 = \frac{\frac{f(x_3) - f(x_2)}{x_3 - x_2} - \frac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_1} \quad (14.9)$$

Notice that, as was the case with linear interpolation, b_2 still represents the slope of the line connecting points x_1 and x_2 . Thus, the first two terms of Eq. (14.6) are equivalent to linear interpolation between x_1 and x_2 , as specified previously in Eq. (14.5). The last term, $b_3(x - x_1)(x - x_2)$, introduces the second-order curvature into the formula.

Before illustrating how to use Eq. (14.6), we should examine the form of the coefficient b_3 . It is very similar to the finite-divided-difference approximation of the second derivative introduced previously in Eq. (4.27). Thus, Eq. (14.6) is beginning to manifest a structure that is very similar to the Taylor series expansion. That is, terms are added sequentially to capture increasingly higher-order curvature.

EXAMPLE 14.3 Quadratic Interpolation

Problem Statement. Employ a second-order Newton polynomial to estimate $\ln 2$ with the same three points used in Example 14.2:

$$\begin{aligned}x_1 &= 1 & f(x_1) &= 0 \\x_2 &= 4 & f(x_2) &= 1.386294 \\x_3 &= 6 & f(x_3) &= 1.791759\end{aligned}$$

Solution. Applying Eq. (14.7) yields

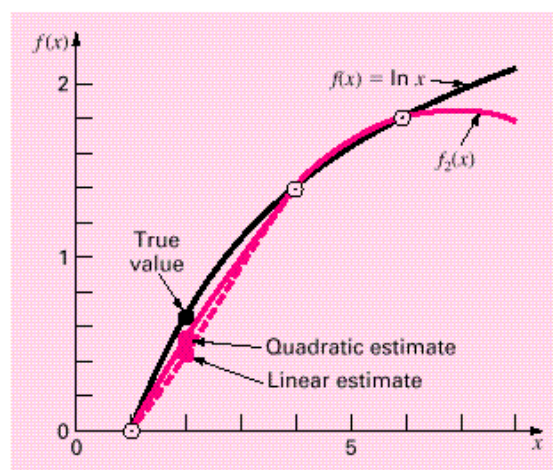
$$b_1 = 0$$

Equation (14.8) gives

$$b_2 = \frac{1.386294 - 0}{4 - 1} = 0.4620981$$

FIGURE 14.4

The use of quadratic interpolation to estimate $\ln 2$. The linear interpolation from $x = 1$ to 4 is also included for comparison.



and Eq. (14.9) yields

$$b_3 = \frac{\frac{1.791759 - 1.386294}{6 - 4} - 0.4620981}{6 - 1} = -0.0518731$$

Substituting these values into Eq. (14.6) yields the quadratic formula

$$f_2(x) = 0 + 0.4620981(x - 1) - 0.0518731(x - 1)(x - 4)$$

which can be evaluated at $x = 2$ for $f_2(2) = 0.5658444$, which represents a relative error of $\varepsilon_t = 18.4\%$. Thus, the curvature introduced by the quadratic formula (Fig. 14.4) improves the interpolation compared with the result obtained using straight lines in Example 14.2 and Fig. 14.3.

14.2.3 General Form of Newton's Interpolating Polynomials

The preceding analysis can be generalized to fit an $(n - 1)$ th-order polynomial to n data points. The $(n - 1)$ th-order polynomial is

$$f_{n-1}(x) = b_1 + b_2(x - x_1) + \cdots + b_n(x - x_1)(x - x_2) \cdots (x - x_{n-1}) \quad (14.10)$$

As was done previously with linear and quadratic interpolation, data points can be used to evaluate the coefficients b_1, b_2, \dots, b_n . For an $(n - 1)$ th-order polynomial, n data points are required: $[x_1, f(x_1)], [x_2, f(x_2)], \dots, [x_n, f(x_n)]$. We use these data points and the following equations to evaluate the coefficients:

$$b_1 = f(x_1) \quad (14.11)$$

$$b_2 = f[x_2, x_1] \quad (14.12)$$

$$b_3 = f[x_3, x_2, x_1] \quad (14.13)$$

$$\vdots$$

$$b_n = f[x_n, x_{n-1}, \dots, x_2, x_1] \quad (14.14)$$

where the bracketed function evaluations are finite divided differences. For example, the first finite divided difference is represented generally as

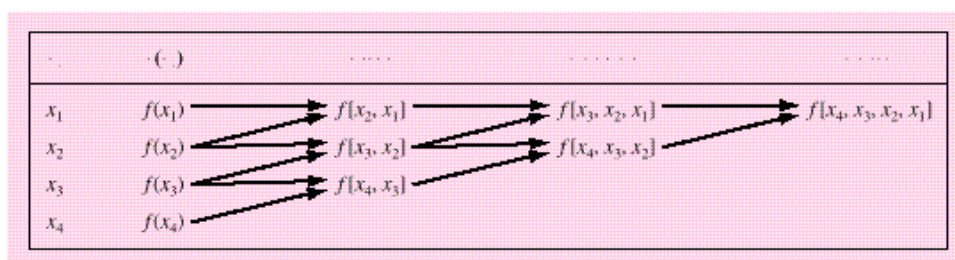
$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j} \quad (14.15)$$

The second finite divided difference, which represents the difference of two first divided differences, is expressed generally as

$$f[x_i, x_j, x_k] = \frac{f[x_i, x_j] - f[x_j, x_k]}{x_i - x_k} \quad (14.16)$$

Similarly, the n th finite divided difference is

$$f[x_n, x_{n-1}, \dots, x_2, x_1] = \frac{f[x_n, x_{n-1}, \dots, x_2] - f[x_{n-1}, x_{n-2}, \dots, x_1]}{x_n - x_1} \quad (14.17)$$

**FIGURE 14.5**

Graphical depiction of the recursive nature of finite divided differences. This representation is referred to as a divided difference table.

These differences can be used to evaluate the coefficients in Eqs. (14.11) through (14.14), which can then be substituted into Eq. (14.10) to yield the general form of Newton's interpolating polynomial:

$$f_{n-1}(x) = f(x_1) + (x - x_1)f[x_2, x_1] + (x - x_1)(x - x_2)f[x_3, x_2, x_1] \\ + \cdots + (x - x_1)(x - x_2) \cdots (x - x_{n-1})f[x_n, x_{n-1}, \dots, x_2, x_1] \quad (14.18)$$

We should note that it is not necessary that the data points used in Eq. (14.18) be equally spaced or that the abscissa values necessarily be in ascending order, as illustrated in the following example. Also, notice how Eqs. (14.15) through (14.17) are recursive—that is, higher-order differences are computed by taking differences of lower-order differences (Fig. 14.5). This property will be exploited when we develop an efficient M-file in Section 14.2.4 to implement the method.

EXAMPLE 14.4 Newton Interpolating Polynomial

Problem Statement. In Example 14.3, data points at $x_1 = 1$, $x_2 = 4$, and $x_3 = 6$ were used to estimate $\ln 2$ with a parabola. Now, adding a fourth point [$x_4 = 5$; $f(x_4) = 1.609438$], estimate $\ln 2$ with a third-order Newton's interpolating polynomial.

Solution. The third-order polynomial, Eq. (14.10) with $n = 4$, is

$$f_3(x) = b_1 + b_2(x - x_1) + b_3(x - x_1)(x - x_2) + b_4(x - x_1)(x - x_2)(x - x_3)$$

The first divided differences for the problem are [Eq. (14.15)]

$$f[x_2, x_1] = \frac{1.386294 - 0}{4 - 1} = 0.462098$$

$$f[x_3, x_2] = \frac{1.791759 - 1.386294}{6 - 4} = 0.2027326$$

$$f[x_4, x_3] = \frac{1.609438 - 1.791759}{5 - 6} = 0.1823216$$

14.2 NEWTON INTERPOLATING POLYNOMIAL

245

The second divided differences are [Eq. (14.16)]

$$f[x_3, x_2, x_1] = \frac{0.2027326 - 0.4620981}{6 - 1} = -0.05187311$$

$$f[x_4, x_3, x_2] = \frac{0.1823216 - 0.2027326}{5 - 4} = -0.02041100$$

The third divided difference is [Eq. (14.17) with $n = 4$]

$$f[x_4, x_3, x_2, x_1] = \frac{-0.02041100 - (-0.05187311)}{5 - 1} = 0.007865529$$

Thus, the divided difference table is

x_i	$f(x_i)$	First	Second	Third
1	0	0.4620981	-0.05187311	0.007865529
4	1.386294	0.2027326	-0.02041100	
6	1.791759	0.1823216		
5	1.609438			

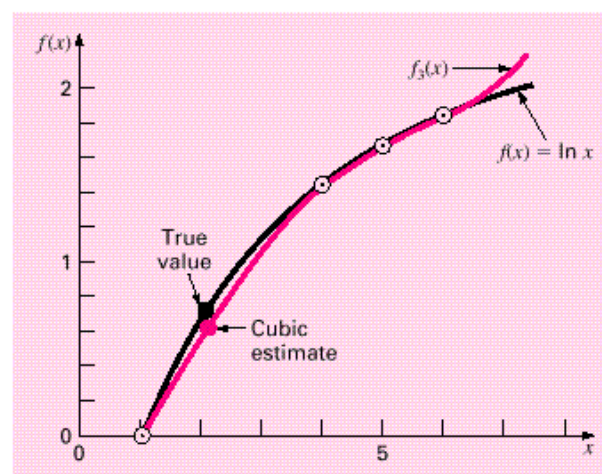
The results for $f(x_1)$, $f[x_2, x_1]$, $f[x_3, x_2, x_1]$, and $f[x_4, x_3, x_2, x_1]$ represent the coefficients b_1 , b_2 , b_3 , and b_4 , respectively, of Eq. (14.10). Thus, the interpolating cubic is

$$f_3(x) = 0 + 0.4620981(x - 1) - 0.05187311(x - 1)(x - 4) + 0.007865529(x - 1)(x - 4)(x - 6)$$

which can be used to evaluate $f_3(2) = 0.6287686$, which represents a relative error of $\epsilon_t = 9.3\%$. The complete cubic polynomial is shown in Fig. 14.6.

FIGURE 14.6

The use of cubic interpolation to estimate $\ln 2$.



14.2.4 MATLAB M-file:

It is straightforward to develop an M-file to implement Newton interpolation. As in Fig. 14.7, the first step is to compute the finite divided differences and store them in an array. The differences are then used in conjunction with Eq. (14.18) to perform the interpolation.

An example of a session using the function would be to duplicate the calculation we just performed in Example 14.3:

```
>> format long  
>> x = [1 4 6 5]';
```

FIGURE 14.7

An M-file to implement Newton interpolation.

```
function yint = Newtint(x,y,xx)  
% newtint(x,y,xx):  
%   Newton interpolation. Uses an (n - 1)-order Newton  
%   interpolating polynomial based on n data points (x, y)  
%   to determine a value of the dependent variable (yint)  
%   at a given value of the independent variable, xx.  
% input:  
%   x = independent variable  
%   y = dependent variable  
%   xx = value of independent variable at which  
%       interpolation is calculated  
% output:  
%   yint = interpolated value of dependent variable  
  
% compute the finite divided differences in the form of a  
% difference table  
n = length(x);  
if length(y)~=n, error('x and y must be same length'); end  
b = zeros(n,n);  
% assign dependent variables to the first column of b.  
b(:,1) = y(:); % the (:) ensures that y is a column vector.  
for j = 2:n  
    for i = 1:n-j+1  
        b(i,j) = (b(i+1,j-1)-b(i,j-1))/(x(i+j-1)-x(i));  
    end  
end  
% use the finite divided differences to interpolate  
xt = 1;  
yint = b(1,1);  
for j = 1:n-1  
    xt = xt*(xx-x(j));  
    yint = yint+b(1,j+1)*xt;  
end
```

```
>> y = log(x);
>> Newtint(x,y,2)

ans =
    0.62876857890841
```

14.3 LAGRANGE INTERPOLATING POLYNOMIAL

Suppose we formulate a linear interpolating polynomial as the weighted average of the two values that we are connecting by a straight line:

$$f(x) = L_1 f(x_1) + L_2 f(x_2) \quad (14.19)$$

where the L 's are the weighting coefficients. It is logical that the first weighting coefficient is the straight line that is equal to 1 at x_1 and 0 at x_2 :

$$L_1 = \frac{x - x_2}{x_1 - x_2}$$

Similarly, the second coefficient is the straight line that is equal to 1 at x_2 and 0 at x_1 :

$$L_2 = \frac{x - x_1}{x_2 - x_1}$$

Substituting these coefficients into Eq. 14.19 yields the straight line that connects the points (Fig. 14.8):

$$f_1(x) = \frac{x - x_2}{x_1 - x_2} f(x_1) + \frac{x - x_1}{x_2 - x_1} f(x_2) \quad (14.20)$$

where the nomenclature $f_1(x)$ designates that this is a first-order polynomial. Equation (14.20) is referred to as the *linear Lagrange interpolating polynomial*.

The same strategy can be employed to fit a parabola through three points. For this case three parabolas would be used with each one passing through one of the points and equaling zero at the other two. Their sum would then represent the unique parabola that connects the three points. Such a second-order Lagrange interpolating polynomial can be written as

$$\begin{aligned} f_2(x) = & \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} f(x_1) + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} f(x_2) \\ & + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} f(x_3) \end{aligned} \quad (14.21)$$

Notice how the first term is equal to $f(x_1)$ at x_1 and is equal to zero at x_2 and x_3 . The other terms work in a similar fashion.

Both the first- and second-order versions as well as higher-order Lagrange polynomials can be represented concisely as

$$f_{n-1}(x) = \sum_{i=1}^n L_i(x) f(x_i) \quad (14.22)$$

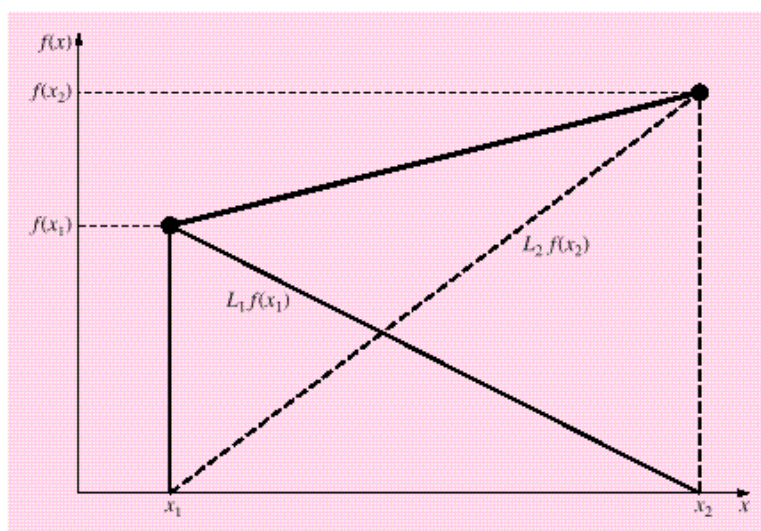


FIGURE 14.8

A visual depiction of the rationale behind Lagrange interpolating polynomials. The figure shows the first-order case. Each of the two terms of Eq. (14.20) passes through one of the points and is zero at the other. The summation of the two terms must, therefore, be the unique straight line that connects the two points.

where

$$L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j} \quad (14.23)$$

where n = the number of data points and \prod designates the “product of.”

EXAMPLE 14.5 Lagrange Interpolating Polynomial

Problem Statement. Use a Lagrange interpolating polynomial of the first and second order to evaluate the density of unused motor oil at $T = 15^\circ\text{C}$ based on the following data:

$$x_1 = 0 \quad f(x_1) = 3.85$$

$$x_2 = 20 \quad f(x_2) = 0.800$$

$$x_3 = 40 \quad f(x_3) = 0.212$$

Solution. The first-order polynomial [Eq. (14.20)] can be used to obtain the estimate at $x = 15$:

$$f_1(x) = \frac{15 - 20}{0 - 20} 3.85 + \frac{15 - 0}{20 - 0} 0.800 = 1.5625$$

14.3 LAGRANGE INTERPOLATING POLYNOMIAL

249

In a similar fashion, the second-order polynomial is developed as [Eq. (14.21)]

$$f_2(x) = \frac{(15-20)(15-40)}{(0-20)(0-40)}3.85 + \frac{(15-0)(15-40)}{(20-0)(20-40)}0.800 \\ + \frac{(15-0)(15-20)}{(40-0)(40-20)}0.212 = 1.3316875$$

14.3.1 MATLAB M-file:

It is straightforward to develop an M-file based on Eqs. (14.22) and (14.23). As in Fig. 14.9, the function is passed two vectors containing the independent (x) and the dependent (y) variables. It is also passed the value of the independent variable where you want to interpolate (xx). The order of the polynomial is based on the length of the x vector that is passed. If n values are passed, an $(n - 1)$ th order polynomial is fit.

FIGURE 14.9

An M-file to implement Lagrange interpolation.

```
function yint = Lagrange(x,y,xx)
% Lagrange(x,y,xx):
%   Lagrange interpolation. Uses an (n - 1)-order Lagrange
%   interpolating polynomial based on n data points (x, y)
%   to determine a value of the dependent variable (yint)
%   at a given value of the independent variable, xx.
% input:
%   x = independent variable
%   y = dependent variable
%   xx = value of independent variable at which the
%        interpolation is calculated
% output:
%   yint = interpolated value of dependent variable

n = length(x);
if length(y)~=n, error('x and y must be same length'); end
s = 0;
for i = 1:n
    product = y(i);
    for j = 1:n
        if i ~= j
            product = product*(xx-x(j))/(x(i)-x(j));
        end
    end
    s = s+product;
end
yint = s;
```

An example of a session using the function would be to predict the density of air at 1 atm pressure at a temperature of 15 °C based on the first four values from Table 14.1. Because four values are passed to the function, a third-order polynomial would be implemented by the `Lagrange` function to give:

```
>> format long
>> T = [-40 0 20 50];
>> d = [1.52 1.29 1.2 1.09];
>> density = Lagrange(T,d,15)

density =
    1.22112847222222
```

14.4 INVERSE INTERPOLATION

As the nomenclature implies, the $f(x)$ and x values in most interpolation contexts are the dependent and independent variables, respectively. As a consequence, the values of the x 's are typically uniformly spaced. A simple example is a table of values derived for the function $f(x) = 1/x$:

x	1	2	3	4	5	6	7
$f(x)$	1	0.5	0.3333	0.25	0.2	0.1667	0.1429

Now suppose that you must use the same data, but you are given a value for $f(x)$ and must determine the corresponding value of x . For instance, for the data above, suppose that you were asked to determine the value of x that corresponded to $f(x) = 0.3$. For this case, because the function is available and easy to manipulate, the correct answer can be determined directly as $x = 1/0.3 = 3.3333$.

Such a problem is called *inverse interpolation*. For a more complicated case, you might be tempted to switch the $f(x)$ and x values [i.e., merely plot x versus $f(x)$] and use an approach like Newton or Lagrange interpolation to determine the result. Unfortunately, when you reverse the variables, there is no guarantee that the values along the new abscissa [the $f(x)$'s] will be evenly spaced. In fact, in many cases, the values will be “telescoped.” That is, they will have the appearance of a logarithmic scale with some adjacent points bunched together and others spread out widely. For example, for $f(x) = 1/x$ the result is

$f(x)$	0.1429	0.1667	0.2	0.25	0.3333	0.5	1
x	7	6	5	4	3	2	1

Such nonuniform spacing on the abscissa often leads to oscillations in the resulting interpolating polynomial. This can occur even for lower-order polynomials. An alternative strategy is to fit an n th-order interpolating polynomial, $f_n(x)$, to the original data [i.e., with $f(x)$ versus x]. In most cases, because the x 's are evenly spaced, this polynomial will not be ill-conditioned. The answer to your problem then amounts to finding the value of x that makes this polynomial equal to the given $f(x)$. Thus, the interpolation problem reduces to a roots problem!

For example, for the problem just outlined, a simple approach would be to fit a quadratic polynomial to the three points: (2, 0.5), (3, 0.3333), and (4, 0.25). The result

14.5 EXTRAPOLATION AND OSCILLATIONS

251

would be

$$f_2(x) = 0.041667x^2 - 0.375x + 1.08333$$

The answer to the inverse interpolation problem of finding the x corresponding to $f(x) = 0.3$ would therefore involve determining the root of

$$0.3 = 0.041667x^2 - 0.375x + 1.08333$$

For this simple case, the quadratic formula can be used to calculate

$$x = \frac{0.375 \pm \sqrt{(-0.375)^2 - 4(0.041667)(0.78333)}}{2(0.041667)} = \frac{5.704158}{3.295842}$$

Thus, the second root, 3.296, is a good approximation of the true value of 3.333. If additional accuracy were desired, a third- or fourth-order polynomial along with one of the root-location methods from Chaps. 5 or 6 could be employed.

14.5 EXTRAPOLATION AND OSCILLATIONS

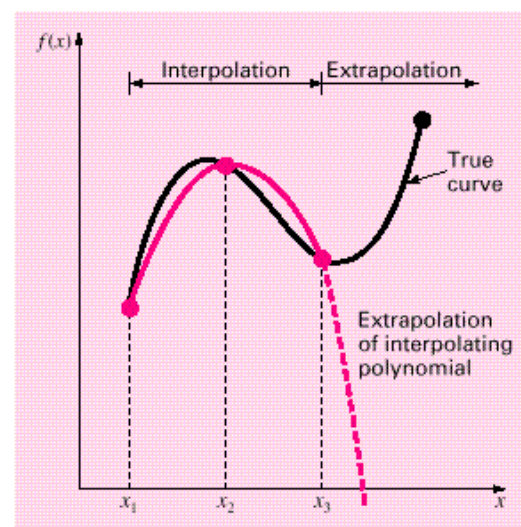
Before leaving this chapter, there are two issues related to polynomial interpolation that must be addressed. These are extrapolation and oscillations.

14.5.1 Extrapolation

Extrapolation is the process of estimating a value of $f(x)$ that lies outside the range of the known base points, x_1, x_2, \dots, x_n . As depicted in Fig. 14.10, the open-ended nature of

FIGURE 14.10

Illustration of the possible divergence of an extrapolated prediction. The extrapolation is based on fitting a parabola through the first three known points.



extrapolation represents a step into the unknown because the process extends the curve beyond the known region. As such, the true curve could easily diverge from the prediction. Extreme care should, therefore, be exercised whenever a case arises where one must extrapolate.

EXAMPLE 14.6 Dangers of Extrapolation

Problem Statement. This example is patterned after one originally developed by Forsythe, Malcolm, and Moler.¹ The population in millions of the United States from 1920 to 2000 can be tabulated as

....	1920	1930	1940	1950	1960	1970	1980	1990	2000
.....	106.46	123.08	132.12	152.27	180.67	205.05	227.23	249.46	281.42

Fit a seventh-order polynomial to the first 8 points (1920 to 1990). Use it to compute the population in 2000 by extrapolation and compare your prediction with the actual result.

Solution. First, the data can be entered as

```
>> t = [1920:10:1990];  
>> pop = [106.46 123.08 132.12 152.27 180.67 205.05 227.23 249.46];
```

The `polyfit` function can be used to compute the coefficients

```
>> p = polyfit(t,pop,7)
```

However, when this is implemented, the following message is displayed:

```
Warning: Polynomial is badly conditioned. Remove repeated data  
points or try centering and scaling as described in HELP  
POLYFIT.
```

We can follow MATLAB's suggestion by scaling and centering the data values as in

```
>> ts = (t - 1955)/35;
```

Now `polyfit` works without an error message:

```
>> p = polyfit(ts,pop,7);
```

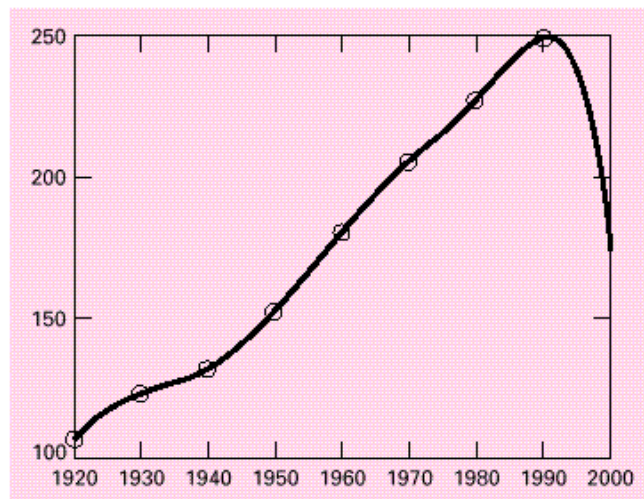
We can then use the polynomial coefficients along with the `polyval` function to predict the population in 2000 as

```
>> polyval(p,(2000-1955)/35)  
  
ans =  
    175.0800
```

which is much lower than the true value of 281.42. Insight into the problem can be gained by generating a plot of the data and the polynomial,

```
>> tt = linspace(1920,2000);  
>> pp = polyval(p,(tt-1955)/35);  
>> plot(t,pop,'o',tt,pp)
```

¹ Cleve Moler is one of the founders of The MathWorks, Inc., the makers of MATLAB.

**FIGURE 14.11**

Use of a seventh-order polynomial to make a prediction of U.S. population in 2000 based on data from 1920 through 1990.

As in Fig. 14.11, the result indicates that the polynomial seems to fit the data nicely from 1920 to 1990. However, once we move beyond the range of the data into the realm of extrapolation, the seventh-order polynomial plunges to the erroneous prediction in 2000.

14.5.2 Oscillations

Although “more is better” in many contexts, it is absolutely not true for polynomial interpolation. Higher-order polynomials tend to be very ill-conditioned—that is, they tend to be highly sensitive to round-off error. The following example illustrates this point nicely.

EXAMPLE 14.7 Dangers of Higher-Order Polynomial Interpolation

Problem Statement. In 1901, Carl Runge published a study on the dangers of higher-order polynomial interpolation. He looked at the following simple-looking function:

$$f(x) = \frac{1}{1 + 25x^2} \quad (14.24)$$

which is now called *Runge’s function*. He took equidistantly spaced data points from this function over the interval $[-1, 1]$. He then used interpolating polynomials of increasing order and found that as he took more points, the polynomials and the original curve differed considerably. Further, the situation deteriorated greatly as the order was increased. Duplicate Runge’s result by using the `polyfit` and `polyval` functions to fit fourth- and tenth-order polynomials to 5 and 11 equally spaced points generated with Eq. (14.24). Create plots of your results along with the sampled values and the complete Runge’s function.

Solution. The five equally spaced data points can be generated as in

```
>> x = linspace(-1,1,5);  
>> y = 1./(1+25*x.^2);
```

Next, a more finely spaced vector of xx values can be computed so that we can create a smooth plot of the results:

```
>> xx = linspace(-1,1);
```

Recall that `linspace` automatically creates 100 points if the desired number of points is not specified. The `polyfit` function can be used to generate the coefficients of the fourth-order polynomial, and the `polyval` function can be used to generate the polynomial interpolation at the finely spaced values of xx :

```
>> p = polyfit(x,y,4);  
>> y4 = polyval(p,xx);
```

Finally, we can generate values for Runge's function itself and plot them along with the polynomial fit and the sampled data:

```
>> yr = 1./(1+25*xx.^2);  
>> plot(x,y,'o',xx,y4,xx,yr,'--')
```

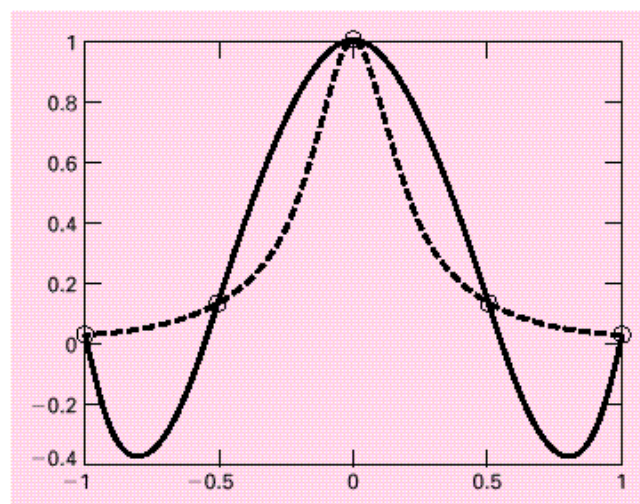
As in Fig. 14.12, the polynomial does a poor job of following Runge's function.

Continuing with the analysis, the tenth-order polynomial can be generated and plotted with

```
>> x = linspace(-1,1,11);  
>> y = 1./(1+25*x.^2);
```

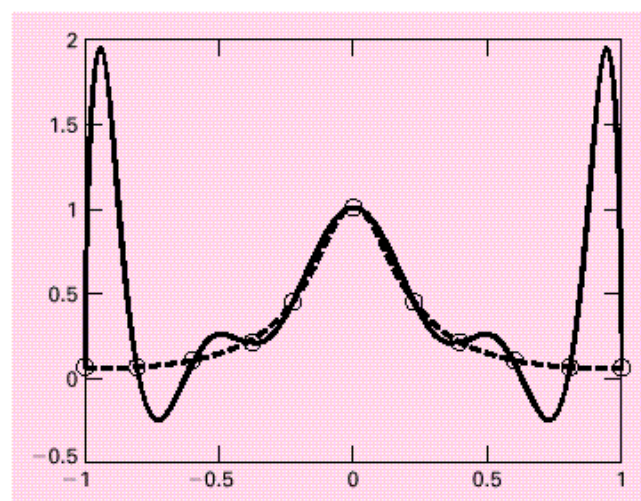
FIGURE 14.12

Comparison of Runge's function (dashed line) with a fourth-order polynomial fit to 5 points sampled from the function.



PROBLEMS

255


FIGURE 14.13

Comparison of Runge's function (dashed line) with a tenth-order polynomial fit to 11 points sampled from the function.

```
>> p = polyfit(x,y,10);
>> y10 = polyval(p,xx);
>> plot(x,y,'o',xx,y10,xx,yr,'--')
```

As in Fig. 14.13, the fit has gotten even worse, particularly at the ends of the interval!

Although there may be certain contexts where higher-order polynomials are necessary, they are usually to be avoided. In most engineering and scientific contexts, lower-order polynomials of the type described in this chapter can be used effectively to capture the curving trends of data without suffering from oscillations.

PROBLEMS

14.1 Given the data

x	1	2	2.5	3	4	5
$f(x)$	0	5	7	6.5	2	0

- Calculate $f(3.4)$ using Newton's interpolating polynomials of order 1 through 3. Choose the sequence of the points for your estimates to attain the best possible accuracy.
- Repeat (a) but use the Lagrange polynomial.

14.2 Given the data

x	1	2	3	5	6
$f(x)$	4.75	4	5.25	19.75	36

Calculate $f(4)$ using Newton's interpolating polynomials of order 1 through 4. Choose your base points to attain good accuracy. What do your results indicate regarding the order of the polynomial used to generate the data in the table?

14.3 Repeat Prob. 14.2 using the Lagrange polynomial of order 1 through 3.

14.4 Table P13.5 lists values for dissolved oxygen concentration in water as a function of temperature and chloride concentration.

- (a) Use quadratic and cubic interpolation to determine the oxygen concentration for $T = 12^\circ\text{C}$ and $c = 10$ g/L.
(b) Use linear interpolation to determine the oxygen concentration for $T = 12^\circ\text{C}$ and $c = 15$ g/L.
(c) Repeat (b) but use quadratic interpolation.

14.5 Employ inverse interpolation using a cubic interpolating polynomial and bisection to determine the value of x that corresponds to $f(x) = 1.6$ for the following tabulated data:

x	1	2	3	4	5	6	7
$f(x)$	3.6	1.8	1.2	0.9	0.72	1.5	0.51429

14.6 Employ inverse interpolation to determine the value of x that corresponds to $f(x) = 0.93$ for the following tabulated data:

x	0	1	2	3	4	5
$f(x)$	0	0.5	0.8	0.9	0.941176	0.961538

Note that the values in the table were generated with the function $f(x) = x^2/(1+x^2)$.

- (a) Determine the correct value analytically.
(b) Use quadratic interpolation and the quadratic formula to determine the value numerically.
(c) Use cubic interpolation and bisection to determine the value numerically.

14.7 Use the portion of the given steam table for superheated water at 200 MPa to find (a) the corresponding entropy s for a specific volume v of 0.118 with linear interpolation, (b) the same corresponding entropy using quadratic interpolation, and (c) the volume corresponding to an entropy of 6.45 using inverse interpolation.

$v, \text{m}^3/\text{kg}$	0.10377	0.11144	0.12547
$s, \text{kJ}/(\text{kg K})$	6.4147	6.5453	6.7664

14.8 The following data for the density of nitrogen gas versus temperature comes from a table that was measured with high precision. Use first- through fifth-order polynomials to estimate the density at a temperature of 330 K. What is your

best estimate? Employ this best estimate and inverse interpolation to determine the corresponding temperature.

T, K	200	250	300	350	400	450
Density, kg/m^3	1.708	1.367	1.139	0.967	0.854	0.759

14.9 Ohm's law states that the voltage drop V across an ideal resistor is linearly proportional to the current i flowing through the resistor as in $V = iR$, where R is the resistance. However, real resistors may not always obey Ohm's law. Suppose that you performed some very precise experiments to measure the voltage drop and corresponding current for a resistor. The following results suggest a curvilinear relationship rather than the straight line represented by Ohm's law:

i	-1	-0.5	-0.25	0.25	0.5	1
V	-193	-41	-13.5625	13.5625	41	193

To quantify this relationship, a curve must be fit to the data. Because of measurement error, regression would typically be the preferred method of curve fitting for analyzing such experimental data. However, the smoothness of the relationship, as well as the precision of the experimental methods, suggests that interpolation might be appropriate. Use a fifth-order interpolating polynomial to fit the data and compute V for $i = 0.10$.

14.10 Bessel functions often arise in advanced engineering analyses such as the study of electric fields. Here are some selected values for the zero-order Bessel function of the first kind

x	0.5	1	1.5
$J_0(x)$	0.938470	0.765198	0.511828
x	2	2.5	3
$J_0(x)$	0.223891	-0.048384	-0.260052

Estimate $J_0(1.82)$ using third-, fourth-, and fifth-order interpolating polynomials. Determine the percent relative error for each case based on the true value, which can be determined with MATLAB's built-in function `besselj`.

14.11 Repeat Example 14.6 but using first-, second-, third-, and fourth-order interpolating polynomials to predict the population in 2000 based on the most recent data. That is, for the linear prediction use the data from 1980 and 1990, for the quadratic prediction use the data from 1970, 1980, and 1990, and so on. Which approach yields the best result?

Curve Fitting: Splines

CHAPTER OBJECTIVES

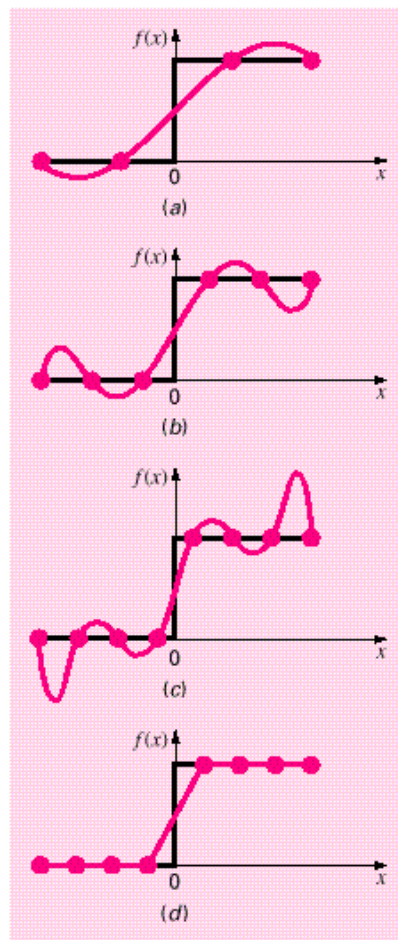
The primary objective of this chapter is to introduce you to splines. Specific objectives and topics covered are

- Understanding that splines minimize oscillations by fitting lower-order polynomials to data in a piecewise fashion.
- Knowing how to develop code to perform a table lookup.
- Recognizing why cubic polynomials are preferable to quadratic and higher-order splines.
- Understanding the conditions that underlie a cubic spline fit.
- Understanding the differences between natural, clamped, and not-a-knot end conditions.
- Knowing how to fit a spline to data with MATLAB's built-in functions.

15.1 INTRODUCTION TO SPLINES

In Chap. 14 $(n - 1)$ th-order polynomials were used to interpolate between n data points. For example, for eight points, we can derive a perfect seventh-order polynomial. This curve would capture all the meanderings (at least up to and including seventh derivatives) suggested by the points. However, there are cases where these functions can lead to erroneous results because of round-off error and oscillations. An alternative approach is to apply lower-order polynomials in a piecewise fashion to subsets of data points. Such connecting polynomials are called *spline functions*.

For example, third-order curves employed to connect each pair of data points are called *cubic splines*. These functions can be constructed so that the connections between adjacent cubic equations are visually smooth. On the surface, it would seem that the third-order approximation of the splines would be inferior to the seventh-order expression. You might wonder why a spline would ever be preferable.

**FIGURE 15.1**

A visual representation of a situation where splines are superior to higher-order interpolating polynomials. The function to be fit undergoes an abrupt increase at $x = 0$. Parts (b) through (c) indicate that the abrupt change induces oscillations in interpolating polynomials. In contrast, because it is limited to straightline connections, a linear spline (d) provides a much more acceptable approximation.

Figure 15.1 illustrates a situation where a spline performs better than a higher-order polynomial. This is the case where a function is generally smooth but undergoes an abrupt change somewhere along the region of interest. The step increase depicted in Fig. 15.1 is an extreme example of such a change and serves to illustrate the point.

Figure 15.1a through c illustrates how higher-order polynomials tend to swing through wild oscillations in the vicinity of an abrupt change. In contrast, the spline also connects the points, but because it is limited to lower-order changes, the oscillations are kept to a minimum. As such, the spline usually provides a superior approximation of the behavior of functions that have local, abrupt changes.

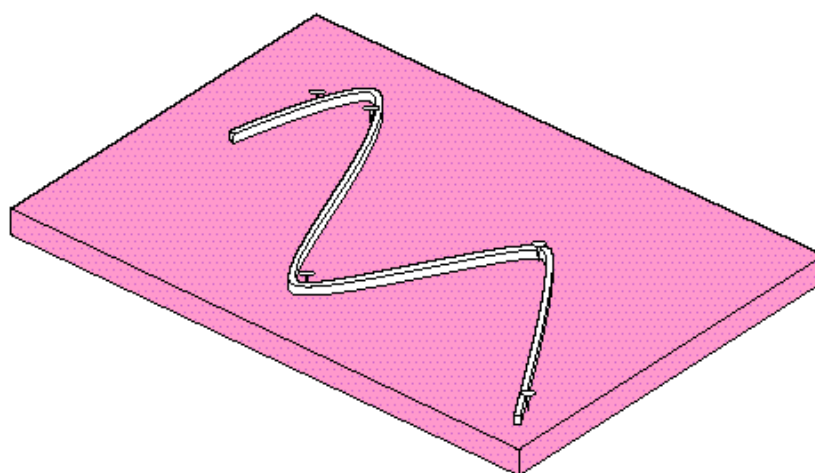


FIGURE 15.2

The drafting technique of using a spline to draw smooth curves through a series of points. Notice how, at the end points, the spline straightens out. This is called a "natural" spline.

The concept of the spline originated from the drafting technique of using a thin, flexible strip (called a *spline*) to draw smooth curves through a set of points. The process is depicted in Fig. 15.2 for a series of five pins (data points). In this technique, the drafter places paper over a wooden board and hammers nails or pins into the paper (and board) at the location of the data points. A smooth cubic curve results from interweaving the strip between the pins. Hence, the name "cubic spline" has been adopted for polynomials of this type.

In this chapter, simple linear functions will first be used to introduce some basic concepts and issues associated with spline interpolation. Then we derive an algorithm for fitting quadratic splines to data. This is followed by material on the cubic spline, which is the most common and useful version in engineering and science. Finally, we describe MATLAB's capabilities for piecewise interpolation including its ability to generate splines.

15.2 LINEAR SPLINES

The notation used for splines is displayed in Fig. 15.3. For n data points ($i = 1, 2, \dots, n$), there are $n - 1$ intervals. Each interval i has its own spline function, $s_i(x)$. For linear splines, each function is merely the straight line connecting the two points at each end of the interval, which is formulated as

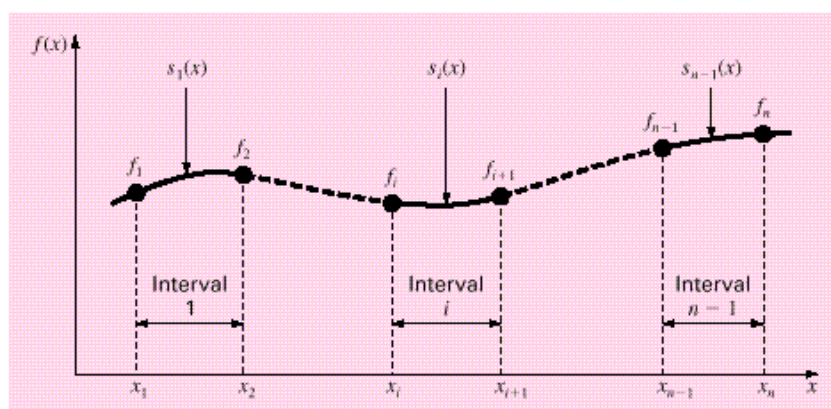
$$s_i(x) = a_i + b_i(x - x_i) \quad (15.1)$$

where a_i is the intercept, which is defined as

$$a_i = f_i \quad (15.2)$$

and b_i is the slope of the straight line connecting the points:

$$b_i = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} \quad (15.3)$$

**FIGURE 15.3**

Notation used to derive splines. Notice that there are $n - 1$ intervals and n data points.

where f_i is shorthand for $f(x_i)$. Substituting Eqs. (15.1) and (15.2) into Eq. (15.3) gives

$$s_i(x) = f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i}(x - x_i) \quad (15.4)$$

These equations can be used to evaluate the function at any point between x_1 and x_n by first locating the interval within which the point lies. Then the appropriate equation is used to determine the function value within the interval. Inspection of Eq. (15.4) indicates that the linear spline amounts to using Newton's first-order polynomial [Eq. (14.5)] to interpolate within each interval.

EXAMPLE 15.1 First-Order Splines

Problem Statement. Fit the data in Table 15.1 with first-order splines. Evaluate the function at $x = 5$.

TABLE 15.1 Data to be fit with spline functions.

x_i	x_{i+1}	f_i
1	3.0	2.5
2	4.5	1.0
3	7.0	2.5
4	9.0	0.5

Solution. The data can be substituted into Eq. (15.4) to generate the linear spline functions. For example, for the second interval from $x = 4.5$ to $x = 7$, the function is

$$s_2(x) = 1.0 + \frac{2.5 - 1.0}{7.0 - 4.5}(x - 4.5)$$

15.2 LINEAR SPLINES

261

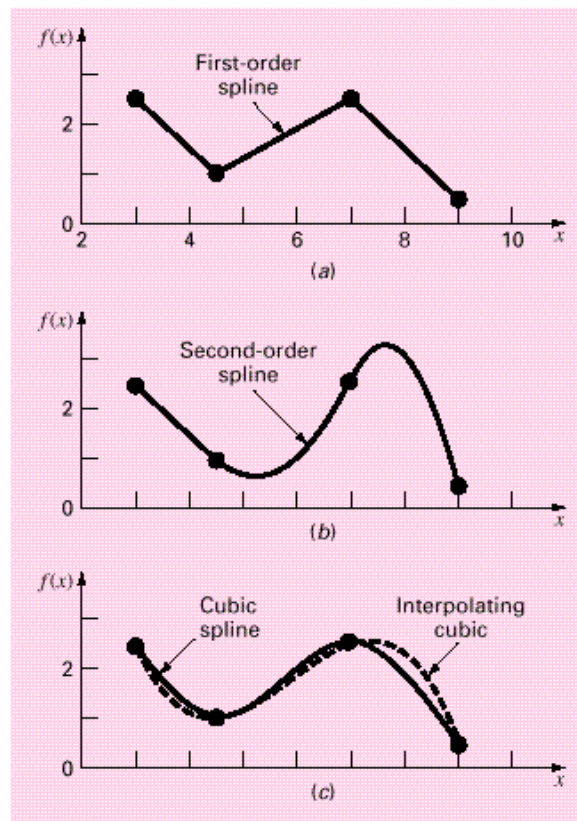


FIGURE 15.4

Spline fits of a set of four points. (—) Linear spline, (—) quadratic spline, and (—) cubic spline, with a cubic interpolating polynomial also plotted.

The equations for the other intervals can be computed, and the resulting first-order splines are plotted in Fig. 15.4a. The value at $x = 5$ is 1.3.

$$s_2(x) = 1.0 + \frac{2.5 - 1.0}{7.0 - 4.5}(5 - 4.5) = 1.3$$

Visual inspection of Fig. 15.4a indicates that the primary disadvantage of first-order splines is that they are not smooth. In essence, at the data points where two splines meet (called a *knot*), the slope changes abruptly. In formal terms, the first derivative of the function is discontinuous at these points. This deficiency is overcome by using higher-order polynomial splines that ensure smoothness at the knots by equating derivatives at these points, as will be discussed subsequently. Before doing that, the following section provides an application where linear splines are useful.

15.2.1 Table Lookup

A table lookup is a common task that is frequently encountered in engineering and science computer applications. It is useful for performing repeated interpolations from a table of independent and dependent variables. For example, suppose that you would like to set up an M-file that would use linear interpolation to determine air density at a particular temperature based on the data from Table 14.1. One way to do this would be to pass the M-file the temperature at which you want the interpolation to be performed along with the two adjoining values. A more general approach would be to pass in vectors containing all the data and have the M-file determine the bracket. This is called a *table lookup*.

Thus, the M-file would perform two tasks. First, it would search the independent variable vector to find the interval containing the unknown. Then it would perform the linear interpolation using one of the techniques described in this chapter or in Chap. 14.

For ordered data, there are two simple ways to find the interval. The first is called a *sequential search*. As the name implies, this method involves comparing the desired value with each element of the vector in sequence until the interval is located. For data in ascending order, this can be done by testing whether the unknown is less than the value being assessed. If so, we know that the unknown falls between this value and the previous one that we examined. If not, we move to the next value and repeat the comparison. Here is a simple M-file that accomplishes this objective:

```
function yi = TableLook(x, y, xx)

n = length(x);
if xx < x(1) | xx > x(n)
    error('Interpolation outside range')
end
% sequential search
i = 1;
while(1)
    if xx <= x(i + 1), break, end
    i = i + 1;
end
% linear interpolation
yi = y(i) + (y(i+1)-y(i))/(x(i+1)-x(i))*(xx-x(i));
```

The table's independent variables are stored in ascending order in the array x and the dependent variables stored in the array y . Before searching, an error trap is included to ensure that the desired value xx falls within the range of the x 's. A `while . . . break` loop compares the value at which the interpolation is desired, xx , to determine whether it is less than the value at the top of the interval, $x(i+1)$. For cases where xx is in the second interval or higher, this will not test true at first. In this case the counter i is incremented by one so that on the next iteration, xx is compared with the value at the top of the second interval. The loop is repeated until the xx is less than or equal to the interval's upper bound, in which case the loop is exited. At this point, the interpolation can be performed simply as shown.

For situations for which there are lots of data, the sequential sort is inefficient because it must search through all the preceding points to find values. In these cases, a simple alternative is the *binary search*. Here is an M-file that performs a binary search followed

15.3 QUADRATIC SPLINES

263

by linear interpolation:

```
function yi = TableLookBin(x, y, xx)

n = length(x);
if xx < x(1) | xx > x(n)
    error('Interpolation outside range')
end
% binary search
iL = 1; iU = n;
while (1)
    if iU - iL <= 1, break, end
    iM = int((iL + iU) / 2);
    if x(iM) < xx
        iL = iM;
    else
        iU = iM;
    end
end
% linear interpolation
yi = y(iL) + (y(iL+1)-y(iL))/(x(iL+1)-x(iL))*(xx - x(iL));
```

The approach is akin to the bisection method for root location. Just as in bisection, the index at the midpoint iM is computed as the average of the first or “lower” index $iL = 1$ and the last or “upper” index $iU = n$. The unknown xx is then compared with the value of x at the midpoint $x(iM)$ to assess whether it is in the lower half of the array or in the upper half. Depending on where it lies, either the lower or upper index is redefined as being the middle index. The process is repeated until the difference between the upper and the lower index is less than or equal to zero. At this point, the lower index lies at the lower bound of the interval containing xx , the loop terminates, and the linear interpolation is performed.

Here is a MATLAB session illustrating how the binary search function can be applied to calculate the air density at 350 °C based on the data from Table 14.1. The sequential search would be similar.

```
>> T = [-40 0 20 50 100 150 200 250 300 400 500];
>> density = [1.52 1.29 1.2 1.09 .946 .935 .746 .675 .616 .525 .457];
>> TableLookBin(T,density,350)

ans =
    0.5705
```

This result can be verified by the hand calculation:

$$f(350) = 0.616 + \frac{0.525 - 0.616}{400 - 300}(350 - 300) = 0.5705$$

15.3 QUADRATIC SPLINES

To ensure that the n th derivatives are continuous at the knots, a spline of at least $n + 1$ order must be used. Third-order polynomials or cubic splines that ensure continuous first and second derivatives are most frequently used in practice. Although third and higher

derivatives can be discontinuous when using cubic splines, they usually cannot be detected visually and consequently are ignored.

Because the derivation of cubic splines is somewhat involved, we have decided to first illustrate the concept of spline interpolation using second-order polynomials. These “quadratic splines” have continuous first derivatives at the knots. Although quadratic splines are not of practical importance, they serve nicely to demonstrate the general approach for developing higher-order splines.

The objective in quadratic splines is to derive a second-order polynomial for each interval between data points. The polynomial for each interval can be represented generally as

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 \quad (15.5)$$

where the notation is as in Fig. 15.3. For n data points ($i = 1, 2, \dots, n$), there are $n - 1$ intervals and, consequently, $3(n - 1)$ unknown constants (the a 's, b 's, and c 's) to evaluate. Therefore, $3n$ equations or conditions are required to evaluate the unknowns. These can be developed as follows:

1. The function must pass through all the points. This is called a *continuity condition*. It can be expressed mathematically as

$$f_i = a_i + b_i(x_i - x_i) + c_i(x_i - x_i)^2$$

which simplifies to

$$a_i = f_i \quad (15.6)$$

Therefore, the constant in each quadratic must be equal to the value of the dependent variable at the beginning of the interval. This result can be incorporated into Eq. (15.5):

$$s_i(x) = f_i + b_i(x - x_i) + c_i(x - x_i)^2$$

Note that because we have determined one of the coefficients, the number of conditions to be evaluated has now been reduced to $2(n - 1)$.

2. The function values of adjacent polynomials must be equal at the knots. This condition can be written for knot $i + 1$ as

$$f_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 = f_{i+1} + b_{i+1}(x_{i+1} - x_{i+1}) + c_{i+1}(x_{i+1} - x_{i+1})^2 \quad (15.7)$$

This equation can be simplified mathematically by defining the width of the i th interval as

$$h_i = x_{i+1} - x_i$$

Thus, Eq. (15.7) simplifies to

$$f_i + b_i h_i + c_i h_i^2 = f_{i+1} \quad (15.8)$$

This equation can be written for the nodes, $i = 1, \dots, n - 1$. Since this amounts to $n - 1$ conditions, it means that there are $2(n - 1) - (n - 1) = n - 1$ remaining conditions.

15.3 QUADRATIC SPLINES

265

3. The first derivatives at the interior nodes must be equal. This is an important condition, because it means that adjacent splines will be joined smoothly, rather than in the jagged fashion that we saw for the linear splines. Equation (15.5) can be differentiated to yield

$$s'_i(x) = b_i + 2c_i(x - x_i)$$

The equivalence of the derivatives at an interior node, $i + 1$ can therefore be written as

$$b_i + 2c_i h_i = b_{i+1} \quad (15.9)$$

Writing this equation for all the interior nodes amounts to $n - 2$ conditions. This means that there is $n - 1 - (n - 2) = 1$ remaining condition. Unless we have some additional information regarding the functions or their derivatives, we must make an arbitrary choice to successfully compute the constants. Although there are a number of different choices that can be made, we select the following condition.

4. Assume that the second derivative is zero at the first point. Because the second derivative of Eq. (15.5) is $2c_i$, this condition can be expressed mathematically as

$$c_1 = 0$$

The visual interpretation of this condition is that the first two points will be connected by a straight line.

EXAMPLE 15.2 Quadratic Splines

Problem Statement. Fit quadratic splines to the same data employed in Example 15.1 (Table 15.1). Use the results to estimate the value at $x = 5$.

Solution. For the present problem, we have four data points and $n = 3$ intervals. Therefore, after applying the continuity condition and the zero second-derivative condition, this means that $2(4 - 1) - 1 = 5$ conditions are required. Equation (15.8) is written for $i = 1$ through 3 (with $c_1 = 0$) to give

$$f_1 + b_1 h_1 = f_2$$

$$f_2 + b_2 h_2 + c_2 h_2^2 = f_3$$

$$f_3 + b_3 h_3 + c_3 h_3^2 = f_4$$

Continuity of derivatives, Eq. (15.9), creates an additional $3 - 1 = 2$ conditions (again, recall that $c_1 = 0$):

$$b_1 = b_2$$

$$b_2 + 2c_2 h_2 = b_3$$

The necessary function and interval width values are

$$f_1 = 2.5 \quad h_1 = 4.5 - 3.0 = 1.5$$

$$f_2 = 1.0 \quad h_2 = 7.0 - 4.5 = 2.5$$

$$f_3 = 2.5 \quad h_3 = 9.0 - 7.0 = 2.0$$

$$f_4 = 0.5$$

These values can be substituted into the conditions which can be expressed in matrix form as

$$\begin{bmatrix} 1.5 & 0 & 0 & 0 & 0 \\ 0 & 2.5 & 6.25 & 0 & 0 \\ 0 & 0 & 0 & 2 & 4 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 5 & -1 & 0 \end{bmatrix} \begin{Bmatrix} b_1 \\ b_2 \\ c_2 \\ b_3 \\ c_3 \end{Bmatrix} = \begin{Bmatrix} -1.5 \\ 1.5 \\ -2 \\ 0 \\ 0 \end{Bmatrix}$$

These equations can be solved using MATLAB with the results:

$$b_1 = -1$$

$$b_2 = -1 \quad c_2 = 0.64$$

$$b_3 = 2.2 \quad c_3 = -1.6$$

These results, along with the values for the a 's (Eq. 15.6), can be substituted into the original quadratic equations to develop the following quadratic splines for each interval:

$$s_1(x) = 2.5 - (x - 3)$$

$$s_2(x) = 1.0 - (x - 4.5) + 0.64(x - 4.5)^2$$

$$s_3(x) = 2.5 + 2.2(x - 7.0) - 1.6(x - 7.0)^2$$

Because $x = 5$ lies in the second interval, we use s_2 to make the prediction,

$$s_2(5) = 1.0 - (5 - 4.5) + 0.64(5 - 4.5)^2 = 0.66$$

The total quadratic spline fit is depicted in Fig. 15.4b. Notice that there are two shortcomings that detract from the fit: (1) the straight line connecting the first two points and (2) the spline for the last interval seems to swing too high. The cubic splines in the next section do not exhibit these shortcomings and, as a consequence, are better methods for spline interpolation.

15.4 CUBIC SPLINES

As stated previously, cubic splines are most frequently used in practice. The shortcomings of linear and quadratic splines have already been discussed. Quartic or higher-order splines are not used because they tend to exhibit the instabilities inherent in higher-order polynomials. Cubic splines are preferred because they provide the simplest representation that exhibits the desired appearance of smoothness.

The objective in cubic splines is to derive a third-order polynomial for each interval between knots as represented generally by

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad (15.10)$$

Thus, for n data points ($i = 1, 2, \dots, n$), there are $n - 1$ intervals and $4(n - 1)$ unknown coefficients to evaluate. Consequently, $4(n - 1)$ conditions are required for their evaluation.

15.4 CUBIC SPLINES

267

The first conditions are identical to those used for the quadratic case. That is, they are set up so that the functions pass through the points and that the first derivatives at the knots are equal. In addition to these, conditions are developed to ensure that the second derivatives at the knots are also equal. This greatly enhances the fit's smoothness.

After these conditions are developed, two additional conditions are required to obtain the solution. This is a much nicer outcome than occurred for quadratic splines where we needed to specify a single condition. In that case, we had to arbitrarily specify a zero second derivative for the first interval, hence making the result asymmetric. For cubic splines, we are in the advantageous position of needing two additional conditions and can, therefore, apply them evenhandedly at both ends.

For cubic splines, these last two conditions can be formulated in several different ways. A very common approach is to assume that the second derivatives at the first and last knots are equal to zero. The visual interpretation of these conditions is that the function becomes a straight line at the end nodes. Specification of such an end condition leads to what is termed a "natural" spline. It is given this name because the drafting spline naturally behaves in this fashion (Fig. 15.2).

There are a variety of other end conditions that can be specified. Two of the more popular are the clamped condition and the not-a-knot conditions. We will describe these options in a subsequent section. For the following derivation, we will limit ourselves to natural splines.

Once the additional end conditions are specified, we would have the $4(n - 1)$ conditions needed to evaluate the $4(n - 1)$ unknown coefficients. Whereas it is certainly possible to develop cubic splines in this fashion, we will present an alternative approach that requires the solution of only $n - 1$ equations. Further, the simultaneous equations will be tridiagonal and hence can be solved very efficiently. Although the derivation of this approach is less straightforward than for quadratic splines, the gain in efficiency is well worth the effort.

15.4.1 Derivation of Cubic Splines

As was the case with quadratic splines, the first condition is that the spline must pass through all the data points.

$$f_i = a_i + b_i(x_i - x_i) + c_i(x_i - x_i)^2 + d_i(x_i - x_i)^3$$

which simplifies to

$$a_i = f_i \quad (15.11)$$

Therefore, the constant in each cubic must be equal to the value of the dependent variable at the beginning of the interval. This result can be incorporated into Eq. (15.10):

$$s_i(x) = f_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad (15.12)$$

Next, we will apply the condition that each of the cubics must join at the knots. For knot $i + 1$, this can be represented as

$$f_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = f_{i+1} \quad (15.13)$$

where

$$h_i = x_{i+1} - x_i$$

The first derivatives at the interior nodes must be equal. Equation (15.12) is differentiated to yield

$$s'_i(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \quad (15.14)$$

The equivalence of the derivatives at an interior node, $i + 1$ can therefore be written as

$$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1} \quad (15.15)$$

The second derivatives at the interior nodes must also be equal. Equation (15.14) can be differentiated to yield

$$s''_i(x) = 2c_i + 6d_i(x - x_i) \quad (15.16)$$

The equivalence of the second derivatives at an interior node, $i + 1$ can therefore be written as

$$c_i + 3d_i h_i = c_{i+1} \quad (15.17)$$

Next, we can solve Eq. (15.17) for d_i :

$$d_i = \frac{c_{i+1} - c_i}{3h_i} \quad (15.18)$$

This can be substituted into Eq. (15.13) to give

$$f_i + b_i h_i + \frac{h_i^2}{3}(2c_i + c_{i+1}) = f_{i+1} \quad (15.19)$$

Equation (15.18) can also be substituted into Eq. (15.15) to give

$$b_{i+1} = b_i + h_i(c_i + c_{i+1}) \quad (15.20)$$

Equation (15.19) can be solved for

$$b_i = \frac{f_{i+1} - f_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1}) \quad (15.21)$$

The index of this equation can be reduced by 1:

$$b_{i-1} = \frac{f_i - f_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{3}(2c_{i-1} + c_i) \quad (15.22)$$

The index of Eq. (15.20) can also be reduced by 1:

$$b_i = b_{i-1} + h_{i-1}(c_{i-1} + c_i) \quad (15.23)$$

Equations (15.21) and (15.22) can be substituted into Eq. (15.23) and the result simplified to yield

$$h_{i-1}c_{i-1} + 2(h_{i-1} - h_i)c_i + h_i c_{i+1} = 3\frac{f_{i+1} - f_i}{h_i} - 3\frac{f_i - f_{i-1}}{h_{i-1}} \quad (15.24)$$

15.4 CUBIC SPLINES

269

This equation can be made a little more concise by recognizing that the terms on the right-hand side are finite divided differences (recall Eq. 14.15):

$$f[x_i, x_j] = \frac{f_i - f_j}{x_i - x_j}$$

Therefore, Eq. (15.24) can be written as

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3(f[x_{i+1}, x_i] - f[x_i, x_{i-1}]) \quad (15.25)$$

Equation (15.25) can be written for the interior knots, $i = 2, 3, \dots, n-2$, which results in $n-3$ simultaneous tridiagonal equations with $n-1$ unknown coefficients, c_1, c_2, \dots, c_{n-1} . Therefore, if we have two additional conditions, we can solve for the c 's. Once this is done, Eqs. (15.21) and (15.18) can be used to determine the remaining coefficients, b and d .

As stated previously, the two additional end conditions can be formulated in a number of ways. One common approach, the natural spline, assumes that the second derivatives at the end knots are equal to zero. To see how these can be integrated into the solution scheme, the second derivative at the first node (Eq. 15.16) can be set to zero as in

$$s_1''(x_1) = 0 = 2c_1 + 6d_1(x_1 - x_1)$$

Thus, this condition amounts to setting c_1 equal to zero.

The same evaluation can be made at the last node:

$$s_{n-1}''(x_n) = 0 = 2c_{n-1} + 6d_{n-1}h_{n-1} \quad (15.26)$$

Recalling Eq. (15.17), we can conveniently define an extraneous parameter c_n , in which case Eq. (15.26) becomes

$$c_{n-1} + 3d_{n-1}h_{n-1} = c_n = 0$$

Thus, to impose a zero second derivative at the last node, we set $c_n = 0$.

The final equations can now be written in matrix form as

$$\begin{bmatrix} 1 & & & & \\ h_1 & 2(h_1 + h_2) & h_2 & & \\ & \ddots & \ddots & \ddots & \\ & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ & & & & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 3(f[x_3, x_2] - f[x_2, x_1]) \\ \vdots \\ 3(f[x_n, x_{n-1}] - f[x_{n-1}, x_{n-2}]) \\ 0 \end{bmatrix} \quad (15.27)$$

As shown, the system is tridiagonal and hence efficient to solve.

EXAMPLE 15.3 Natural Cubic Splines

Problem Statement. Fit cubic splines to the same data used in Examples 15.1 and 15.2 (Table 15.1). Utilize the results to estimate the value at $x = 5$.

Solution. The first step is to employ Eq. (15.27) to generate the set of simultaneous equations that will be utilized to determine the c coefficients:

$$\begin{bmatrix} 1 & & & \\ h_1 & 2(h_1 + h_2) & h_2 & \\ & h_2 & 2(h_2 + h_3) & h_3 \\ & & & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 3(f[x_3, x_2] - f[x_2, x_1]) \\ 3(f[x_4, x_3] - f[x_3, x_2]) \\ 0 \end{bmatrix}$$

The necessary function and interval width values are

$$\begin{aligned} f_1 &= 2.5 & h_1 &= 4.5 - 3.0 = 1.5 \\ f_2 &= 1.0 & h_2 &= 7.0 - 4.5 = 2.5 \\ f_3 &= 2.5 & h_3 &= 9.0 - 7.0 = 2.0 \\ f_4 &= 0.5 \end{aligned}$$

These can be substituted to yield

$$\begin{bmatrix} 1 & & & \\ 1.5 & 8 & 2.5 & \\ & 2.5 & 9 & 2 \\ & & & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 4.8 \\ -4.8 \\ 0 \end{bmatrix}$$

These equations can be solved using MATLAB with the results:

$$\begin{aligned} c_1 &= 0 & c_2 &= 0.839543726 \\ c_3 &= -0.766539924 & c_4 &= 0 \end{aligned}$$

Equations (15.21) and (15.18) can be used to compute the b 's and d 's

$$\begin{aligned} b_1 &= -1.419771863 & d_1 &= 0.186565272 \\ b_2 &= -0.160456274 & d_2 &= -0.214144487 \\ b_3 &= 0.022053232 & d_3 &= 0.127756654 \end{aligned}$$

These results, along with the values for the a 's [Eq. (15.11)], can be substituted into Eq. (15.10) to develop the following cubic splines for each interval:

$$\begin{aligned} s_1(x) &= 2.5 - 1.419771863(x - 3) + 0.186565272(x - 3)^3 \\ s_2(x) &= 1.0 - 0.160456274(x - 4.5) + 0.839543726(x - 4.5)^2 \\ &\quad - 0.214144487(x - 4.5)^3 \\ s_3(x) &= 2.5 + 0.022053232(x - 7.0) - 0.766539924(x - 7.0)^2 \\ &\quad + 0.127756654(x - 7.0)^3 \end{aligned}$$

The three equations can then be employed to compute values within each interval. For example, the value at $x = 5$, which falls within the second interval, is calculated as

$$\begin{aligned} s_2(5) &= 1.0 - 0.160456274(5 - 4.5) + 0.839543726(5 - 4.5)^2 - 0.214144487(5 - 4.5)^3 \\ &= 1.102889734 \end{aligned}$$

The total cubic spline fit is depicted in Fig. 15.4c.

15.4 CUBIC SPLINES

271

The results of Examples 15.1 through 15.3 are summarized in Fig. 15.4. Notice the progressive improvement of the fit as we move from linear to quadratic to cubic splines. We have also superimposed a cubic interpolating polynomial on Fig. 15.4c. Although the cubic spline consists of a series of third-order curves, the resulting fit differs from that obtained using the third-order polynomial. This is due to the fact that the natural spline requires zero second derivatives at the end knots, whereas the cubic polynomial has no such constraint.

15.4.2 End Conditions

Although its graphical basis is appealing, the natural spline is only one of several end conditions that can be specified for splines. Two of the most popular are

- **Clamped End Condition.** This option involves specifying the first derivatives at the first and last nodes. This is sometimes called a “clamped” spline because it is what occurs when you clamp the end of a drafting spline so that it has a desired slope. For example, if zero first derivatives are specified, the spline will level off or become horizontal at the ends.
- **“Not-a-Knot” End Condition.** A third alternative is to force continuity of the third derivative at the second and the next-to-last knots. Since the spline already specifies that the function value and its first and second derivatives are equal at these knots, specifying continuous third derivatives means that the same cubic functions will apply to each of the first and last two adjacent segments. Since the first internal knots no longer represent the junction of two different cubic functions, they are no longer true knots. Hence, this case is referred to as the “not-a-knot” condition. It has the additional property that for four points, it yields the same result as is obtained using an ordinary cubic interpolating polynomial of the sort described in Chap. 14.

These conditions can be readily applied by using Eq. (15.25) for the interior knots, $i = 2, 3, \dots, n - 2$, and using first (1) and last equations ($n - 1$) as written in Table 15.2.

Figure 15.5 shows a comparison of the three end conditions as applied to fit the data from Table 15.1. The clamped case is set up so that the derivatives at the ends are equal to zero.

As expected, the spline fit for the clamped case levels off at the ends. In contrast, the natural and not-a-knot cases follow the trend of the data points more closely. Notice how the natural spline tends to straighten out as would be expected because the second derivatives go to zero at the ends. Because it has nonzero second derivatives at the ends, the not-a-knot exhibits more curvature.

TABLE 15.2 The first and last equations needed to specify some commonly used end conditions for cubic splines.

Condition	First and Last Equations
Natural	$c_1 = 0, c_n = 0$
Clamped (where f'_1 and f'_n are the specified first derivatives at the first and last nodes, respectively).	$2h_1c_1 + h_1c_2 = 3f[x_2, x_1] - 3f'_1$ $h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3f'_n - 3f[x_n, x_{n-1}]$
Not-a-knot	$h_2c_1 - (h_1 + h_2)c_2 + h_1c_3 = 0$ $h_{n-1}c_{n-2} - (h_{n-2} + h_{n-1})c_{n-1} + h_{n-2}c_n = 0$

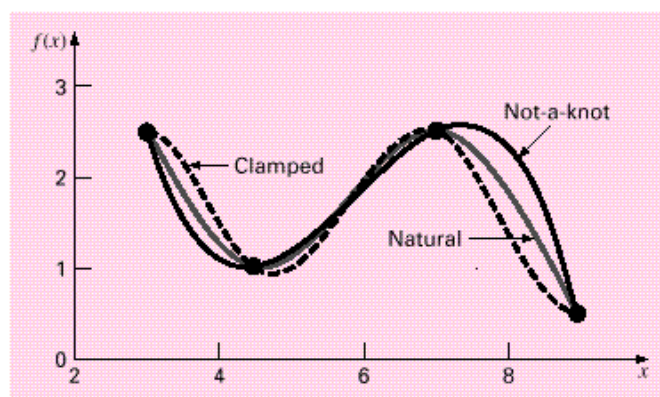


FIGURE 15.5

Comparison of the clamped (with zero first derivatives), not-a-knot, and natural splines for the data from Table 15.1.

15.5 PIECEWISE INTERPOLATION IN MATLAB

MATLAB has several built-in functions to implement piecewise interpolation. The `spline` function performs cubic spline interpolation as described in this chapter. The `pchip` function implements piecewise cubic Hermite interpolation. The `interp1` function can also implement spline and Hermite interpolation, but can also perform a number of other types of piecewise interpolation.

15.5.1 MATLAB Function: `spline`

Cubic splines can be easily computed with the built-in MATLAB function, `spline`. It has the general syntax,

$$yy = \text{spline}(x, y, xx) \quad (15.28)$$

where x and y = vectors containing the values that are to be interpolated, and yy = a vector containing the results of the spline interpolation as evaluated at the points in the vector xx .

By default, `spline` uses the not-a-knot condition. However, if y contains two more values than x has entries, then the first and last value in y are used as the derivatives at the end points. Consequently, this option provides the means to implement the clamped-end condition.

EXAMPLE 15.4 Splines in MATLAB

Problem Statement. Runge's function is a notorious example of a function that cannot be fit well with polynomials (recall Example 14.7):

$$f(x) = \frac{1}{1 + 25x^2}$$

Use MATLAB to fit nine equally spaced data points sampled from this function in the interval $[-1, 1]$. Employ (a) a not-a-knot spline and (b) a clamped spline with end slopes of $f'_1 = 1$ and $f'_{n-1} = -4$.

Solution. (a) The nine equally spaced data points can be generated as in

```
>> x = linspace(-1,1,9);  
>> y = 1./(1+25*x.^2);
```

Next, a more finely spaced vector of values can be generated so that we can create a smooth plot of the results as generated with the `spline` function:

```
>> xx = linspace(-1,1);  
>> yy = spline(x,y,xx);
```

Recall that `linspace` automatically creates 100 points if the desired number of points are not specified. Finally, we can generate values for Runge's function itself and display them along with the spline fit and the original data:

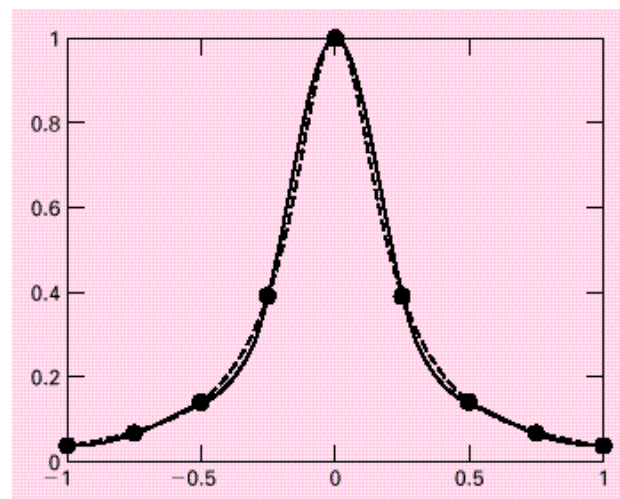
```
>> yr = 1./(1+25*xx.^2);  
>> plot(x,y,'o',xx,yy,xx,yr,'--')
```

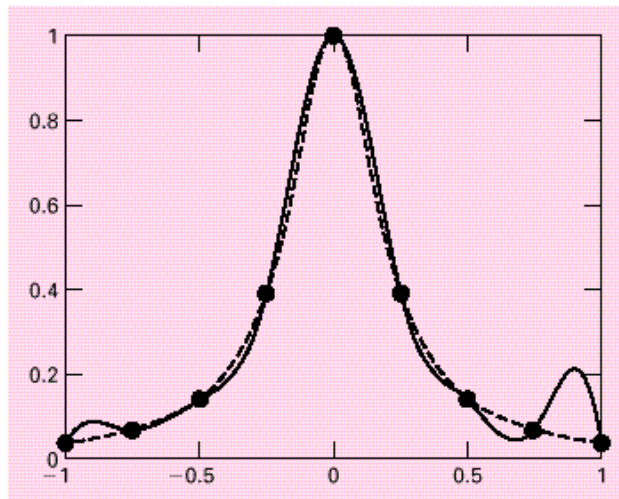
As in Fig. 15.6, the not-a-knot spline does a nice job of following Runge's function without exhibiting wild oscillations between the points.

(b) The clamped condition can be implemented by creating a new vector `yc` that has the desired first derivatives as its first and last elements. The new vector can then be used to

FIGURE 15.6

Comparison of Runge's function (dashed line) with a 9-point not-a-knot spline fit generated with MATLAB (solid line).



**FIGURE 15.7**

Comparison of Runge's function (dashed line) with a 9-point clamped end spline fit generated with MATLAB (solid line). Note that first derivatives of 1 and -4 are specified at the left and right boundaries, respectively.

generate and plot the spline fit:

```
>> yc = [1 y -4];  
>> yyc = spline(x,yc,xx);  
>> plot(x,y,'o',xx,yyc,xx,yr,'--')
```

As in Fig. 15.7, the clamped spline now exhibits some oscillations because of the artificial slopes that we have imposed at the boundaries. In other examples, where we have knowledge of the true first derivatives, the clamped spline tends to improve the fit.

15.5.2 MATLAB Function: `spline`

The built-in function `spline` provides a handy means to implement a number of different types of piecewise one-dimensional interpolation. It has the general syntax

```
yi = spline(x, y, xi, 'method')
```

where x and y = vectors containing values that are to be interpolated, y_i = a vector containing the results of the interpolation as evaluated at the points in the vector x_i , and 'method' = the desired method. The various methods are

- 'nearest'—nearest neighbor interpolation. This method sets the value of an interpolated point to the value of the nearest existing data point. Thus, the interpolation looks like a series of plateaus, which can be thought of as zero-order polynomials.
- 'linear'—linear interpolation. This method uses straight lines to connect the points.

15.5 PIECEWISE INTERPOLATION IN MATLAB

275

- 'spline'—piecewise cubic spline interpolation. This is identical to the `spline` function.
- 'pchip' and 'cubic'—piecewise cubic Hermite interpolation.

If the `'method'` argument is omitted, the default is linear interpolation.

The `pchip` option (short for “piecewise cubic Hermite interpolation”) merits more discussion. As with cubic splines, `pchip` uses cubic polynomials to connect data points with continuous first derivatives. However, it differs from cubic splines in that the second derivatives are not necessarily continuous. Further, the first derivatives at the knots will not be the same as for cubic splines. Rather, they are expressly chosen so that the interpolation is “shape preserving.” That is, the interpolated values do not tend to overshoot the data points as can sometimes happen with cubic splines.

Therefore, there are trade-offs between the `spline` and the `pchip` options. The results of using `spline` will generally appear smoother because the human eye can detect discontinuities in the second derivative. In addition, it will be more accurate if the data are values of a smooth function. On the other hand, `pchip` has no overshoots and less oscillation if the data are not smooth. These trade-offs, as well as those involving the other options, are explored in the following example.

EXAMPLE 15.5 Trade-Offs Using `interp1`

Problem Statement. You perform a test drive on an automobile where you alternately accelerate the automobile and then hold it at a steady velocity. Note that you never decelerate during the experiment. The time series of spot measurements of velocity can be tabulated as

·	0	20	40	56	68	80	84	96	104	110
·	0	20	20	38	80	80	100	100	125	125

Use MATLAB's `interp1` function to fit this data with (a) linear interpolation, (b) nearest neighbor, (c) cubic spline with not-a-knot end conditions, and (d) piecewise cubic Hermite interpolation.

Solution. (a) The data can be entered, fit with linear interpolation, and plotted with the following commands:

```
>> t = [0 20 40 56 68 80 84 96 104 110];  
>> v = [0 20 20 38 80 80 100 100 125 125];  
>> tt = linspace(0,110);  
>> vl = interp1(t,v,tt);  
>> plot(t,v,'o',tt,vl)
```

The results (Fig. 15.8a) are not smooth, but do not exhibit any overshoot.

(b) The commands to implement and plot the nearest neighbor interpolation are

```
>> vn = interp1(t,v,tt,'nearest');  
>> plot(t,v,'o',tt,vn)
```

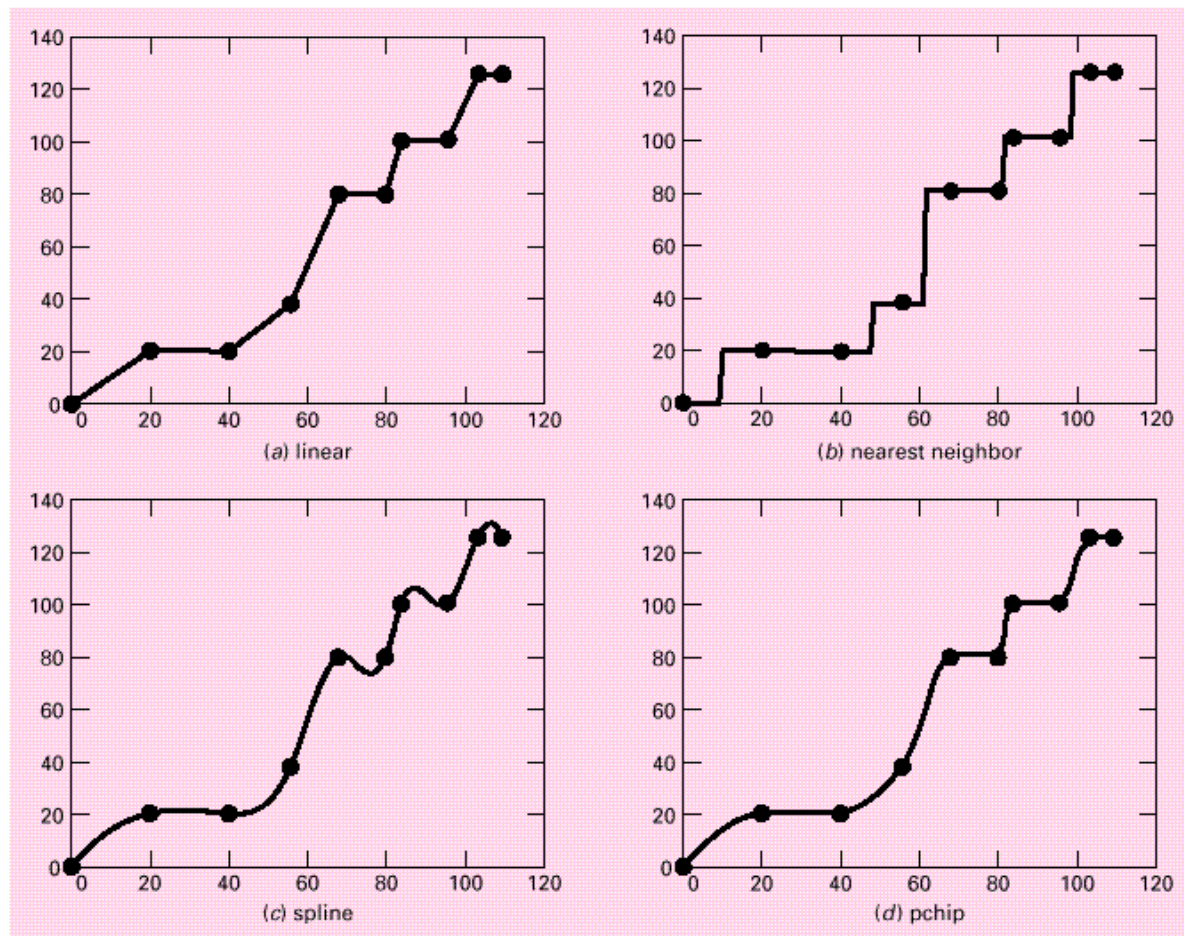


FIGURE 15.8

Use of several options of the `interp1` function to perform piecewise polynomial interpolation on a velocity time series for an automobile.

As in Fig. 15.8b, the results look like a series of plateaus. This option is neither a smooth nor an accurate depiction of the underlying process.

(c) The commands to implement the cubic spline are

```
>> vs = interp1(t,v,tt,'spline');
>> plot(t,v,'o',tt,vs)
```

These results (Fig. 15.8c) are quite smooth. However, severe overshoot occurs at several locations. This makes it appear that the automobile decelerated several times during the experiment.

PROBLEMS

277

(d) The commands to implement the piecewise cubic Hermite interpolation are

```
>> vh = interp1(t,v,tt,'pchip');
>> plot(t,v,'o',tt,vh)
```

For this case, the results (Fig. 15.8d) are physically realistic. Because of its shape-preserving nature, the velocities increase monotonically and never exhibit deceleration. Although the result is not as smooth as for the cubic splines, continuity of the first derivatives at the knots makes the transitions between points more gradual and hence more realistic.

PROBLEMS

15.1 Given the data

	1	2	2.5	3	4	5
$f(x)$	1	5	7	8	2	1

Fit this data with (a) a cubic spline with natural end conditions, (b) a cubic spline with not-a-knot end conditions, and (c) piecewise cubic Hermite interpolation.

15.2 A reactor is thermally stratified as in the following table:

Depth, m	0	0.5	1	1.5	2	2.5	3
Temperature, °C	70	70	55	22	13	10	10

Based on these temperatures, the tank can be idealized as two zones separated by a strong temperature gradient or *thermocline*. The depth of the thermocline can be defined as the inflection point of the temperature-depth curve—that is, the point at which $d^2T/dz^2 = 0$. At this depth, the heat flux from the surface to the bottom layer can be computed with Fourier's law:

$$J = -k \frac{dT}{dz}$$

Use a clamped cubic spline fit with zero end derivatives to determine the thermocline depth. If $k = 0.01$ cal/(s · cm · °C) compute the flux across this interface.

15.3 The following is the built-in `humps` function that MATLAB uses to demonstrate some of its numerical capabilities:

$$f(x) = \frac{1}{(x - 0.3)^2 + 0.01} + \frac{1}{(x - 0.9)^2 + 0.04} - 6$$

The `humps` function exhibits both flat and steep regions over a relatively short x range. Here are some values that

have been generated at intervals of 0.1 over the range from $x = 0$ to 1:

	0	0.1	0.2	0.3	0.4	0.5
$f(x)$	5.176	15.471	45.887	96.500	47.448	19.000
	0.6	0.7	0.8	0.9	1	
$f(x)$	11.692	12.382	17.846	21.703	16.000	

Fit this data with a (a) cubic spline with not-a-knot end conditions and (b) piecewise cubic Hermite interpolation. In both cases, create a plot comparing the fit with the exact `humps` function.

15.4 Develop a plot of a cubic spline fit of the following data with (a) natural end conditions and (b) not-a-knot end conditions. In addition, develop a plot using (c) piecewise cubic Hermite interpolation.

	0	100	200	400
$f(x)$	0	0.82436	1.00000	0.73576
	600	800	1000	
$f(x)$	0.40601	0.19915	0.09158	

In each case, compare your plot with the following equation, which was used to generate the data:

$$f(x) = \frac{x}{200} e^{-\frac{x}{200} + 1}$$

15.5 The following data is sampled from the step function depicted in Fig. 15.1:

	-1	-0.6	-0.2	0.2	0.6	1
$f(x)$	0	0	0	1	1	1

Fit this data with a (a) cubic spline with not-a-knot end conditions, (b) cubic spline with zero-slope clamped end conditions, and (c) piecewise cubic Hermite interpolation. In each case, create a plot comparing the fit with the step function.

15.6 Develop an M-file to compute a cubic spline fit with natural end conditions. Test your code by using it to duplicate Example 15.3.

15.7 The following data was generated with the fifth-order polynomial: $f(x) = 0.0185x^5 - 0.444x^4 + 3.9125x^3 - 15.456x^2 + 27.069x - 14.1$:

	1	3	5	6	7	9
$f(x)$	1.000	2.172	4.220	5.430	4.912	9.120

(a) Fit this data with a cubic spline with not-a-knot end conditions. Create a plot comparing the fit with the function.

(b) Repeat (a) but use clamped end conditions where the end slopes are set at the exact values as determined by differentiating the function.

Numerical Integration Formulas

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to numerical integration. Specific objectives and topics covered are

- Recognizing that Newton-Cotes integration formulas are based on the strategy of replacing a complicated function or tabulated data with a polynomial that is easy to integrate.
- Knowing how to implement the following single application Newton-Cotes formulas:
 - Trapezoidal rule
 - Simpson's 1/3 rule
 - Simpson's 3/8 rule
- Knowing how to implement the following composite Newton-Cotes formulas:
 - Trapezoidal rule
 - Simpson's 1/3 rule
- Recognizing that even-segment-odd-point formulas like Simpson's 1/3 rule achieve higher than expected accuracy.
- Knowing how to use the trapezoidal rule to integrate unequally spaced data.
- Understanding the difference between open and closed integration formulas.

YOU'VE GOT A PROBLEM

Recall that the velocity of a free-falling bungee jumper as a function of time can be computed as

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) \quad (16.1)$$

Suppose that we would like to know the vertical distance z the jumper has fallen after a certain time t . This distance can be evaluated by integration:

$$z(t) = \int_0^t v(t) dt \quad (16.2)$$

Substituting Eq. (16.1) into Eq. (16.2) gives

$$z(t) = \int_0^t \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) dt \quad (16.3)$$

Thus, integration provides the means to determine the distance from the velocity. Calculus can be used to solve Eq. (16.3) for

$$z(t) = \frac{m}{c_d} \ln \left[\cosh\left(\sqrt{\frac{gc_d}{m}} t\right) \right] \quad (16.4)$$

Although a closed form solution can be developed for this case, there are other functions that cannot be integrated analytically. Further, suppose that there was some way to measure the jumper's velocity at various times during the fall. These velocities along with their associated times could be assembled as a table of discrete values. In this situation, it would also be possible to integrate the discrete data to determine the distance. In both these instances, numerical integration methods are available to obtain solutions. Chapters 16 and 17 will introduce you to some of these methods.

16.1 INTRODUCTION AND BACKGROUND

16.1.1 What Is Integration?

According to the dictionary definition, to integrate means “to bring together, as parts, into a whole; to unite; to indicate the total amount. . . .” Mathematically, definite integration is represented by

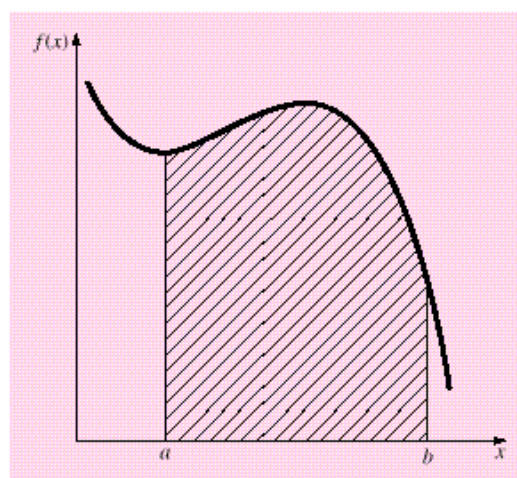
$$I = \int_a^b f(x) dx \quad (16.5)$$

which stands for the integral of the function $f(x)$ with respect to the independent variable x , evaluated between the limits $x = a$ to $x = b$.

As suggested by the dictionary definition, the “meaning” of Eq. (16.5) is the total value, or summation, of $f(x) dx$ over the range $x = a$ to b . In fact, the symbol \int is actually a stylized capital S that is intended to signify the close connection between integration and summation.

Figure 16.1 represents a graphical manifestation of the concept. For functions lying above the x axis, the integral expressed by Eq. (16.5) corresponds to the area under the curve of $f(x)$ between $x = a$ and b .

Numerical integration is sometimes referred to as quadrature. This is an archaic term that originally meant the construction of a square having the same area as some curvilinear figure. Today, the term *quadrature* is generally taken to be synonymous with numerical definite integration.

**FIGURE 16.1**

Graphical representation of the integral of $f(x)$ between the limits $x = a$ to b . The integral is equivalent to the area under the curve.

16.1.2 Integration in Engineering and Science

Integration has so many engineering and scientific applications that you were required to take integral calculus in your first year at college. Many specific examples of such applications could be given in all fields of engineering and science. A number of examples relate directly to the idea of the integral as the area under a curve. Figure 16.2 depicts a few cases where integration is used for this purpose.

Other common applications relate to the analogy between integration and summation. For example, a common application is to determine the mean of a continuous function. Recall that the mean of discrete of n discrete data points can be calculated by [Eq. (12.2)].

$$\text{Mean} = \frac{\sum_{i=1}^n y_i}{n} \quad (16.6)$$

where y_i are individual measurements. The determination of the mean of discrete points is depicted in Fig. 16.3a.

In contrast, suppose that y is a continuous function of an independent variable x , as depicted in Fig. 16.3b. For this case, there are an infinite number of values between a and b . Just as Eq. (16.6) can be applied to determine the mean of the discrete readings, you might also be interested in computing the mean or average of the continuous function $y = f(x)$ for the interval from a to b . Integration is used for this purpose, as specified by

$$\text{Mean} = \frac{\int_a^b f(x) dx}{b - a} \quad (16.7)$$

This formula has hundreds of engineering and scientific applications. For example, it is used to calculate the center of gravity of irregular objects in mechanical and civil engineering and to determine the root-mean-square current in electrical engineering.

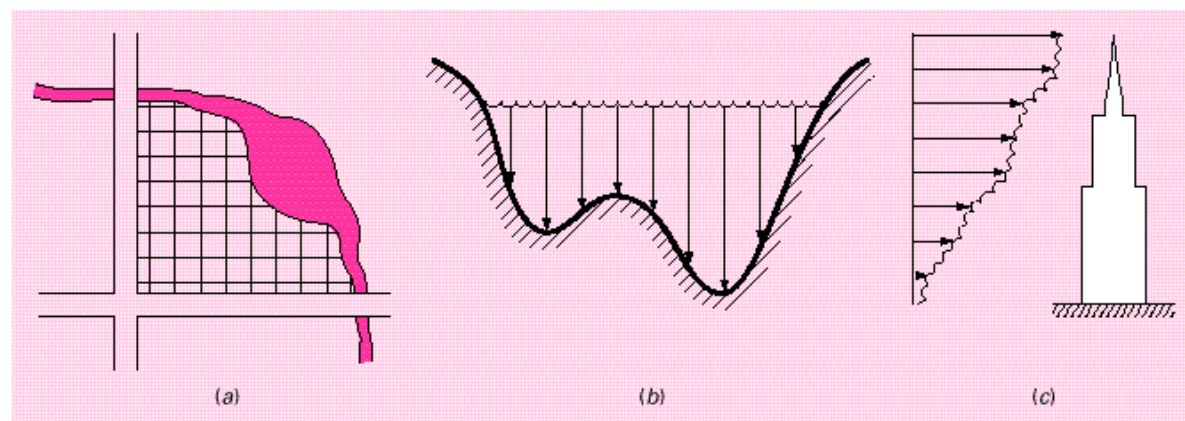


FIGURE 16.2

Examples of how integration is used to evaluate areas in engineering and scientific applications. (·) A surveyor might need to know the area of a field bounded by a meandering stream and two roads. (·) A hydrologist might need to know the cross-sectional area of a river. (·) A structural engineer might need to determine the net force due to a nonuniform wind blowing against the side of a skyscraper.

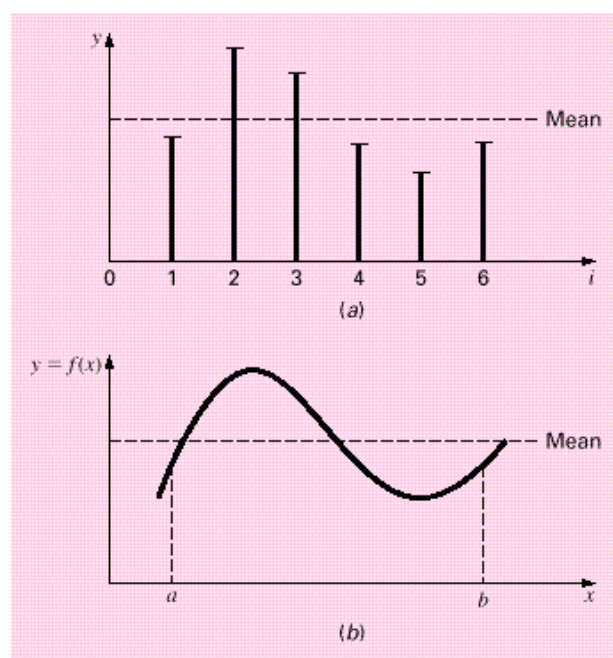


FIGURE 16.3

An illustration of the mean for (·) discrete and (·) continuous data.

16.2 NEWTON-COTES FORMULAS

283

Integrals are also employed by engineers and scientists to evaluate the total amount or quantity of a given physical variable. The integral may be evaluated over a line, an area, or a volume. For example, the total mass of chemical contained in a reactor is given as the product of the concentration of chemical and the reactor volume, or

$$\text{Mass} = \text{concentration} \times \text{volume}$$

where concentration has units of mass per volume. However, suppose that concentration varies from location to location within the reactor. In this case, it is necessary to sum the products of local concentrations c_i and corresponding elemental volumes ΔV_i :

$$\text{Mass} = \sum_{i=1}^n c_i \Delta V_i$$

where n is the number of discrete volumes. For the continuous case, where $c(x, y, z)$ is a known function and x , y , and z are independent variables designating position in Cartesian coordinates, integration can be used for the same purpose:

$$\text{Mass} = \iiint c(x, y, z) dx dy dz$$

or

$$\text{Mass} = \iiint_V c(V) dV$$

which is referred to as a *volume integral*. Notice the strong analogy between summation and integration.

Similar examples could be given in other fields of engineering and science. For example, the total rate of energy transfer across a plane where the flux (in calories per square centimeter per second) is a function of position is given by

$$\text{Flux} = \iint_A \text{flux} dA$$

which is referred to as an *areal integral*, where A = area.

These are just a few of the applications of integration that you might face regularly in the pursuit of your profession. When the functions to be analyzed are simple, you will normally choose to evaluate them analytically. However, it is often difficult or impossible when the function is complicated, as is typically the case in more realistic examples. In addition, the underlying function is often unknown and defined only by measurement at discrete points. For both these cases, you must have the ability to obtain approximate values for integrals using numerical techniques as described next.

16.2 NEWTON-COTES FORMULAS

The *Newton-Cotes formulas* are the most common numerical integration schemes. They are based on the strategy of replacing a complicated function or tabulated data with a polynomial that is easy to integrate:

$$I = \int_a^b f(x) dx \cong \int_a^b f_n(x) dx \quad (16.8)$$

where $f_n(x)$ = a polynomial of the form

$$f_n(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n \quad (16.9)$$

where n is the order of the polynomial. For example, in Fig. 16.4a, a first-order polynomial (a straight line) is used as an approximation. In Fig. 16.4b, a parabola is employed for the same purpose.

FIGURE 16.4

The approximation of an integral by the area under (·) a straight line and (·) a parabola.

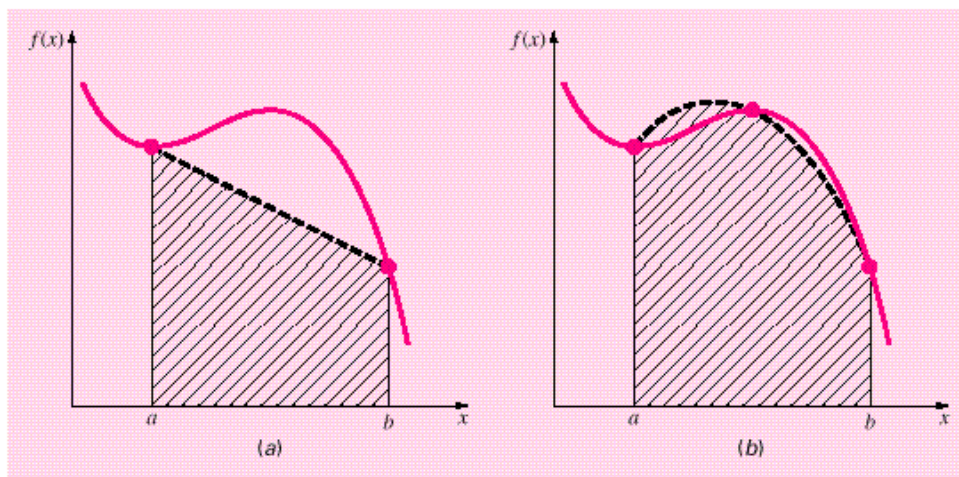
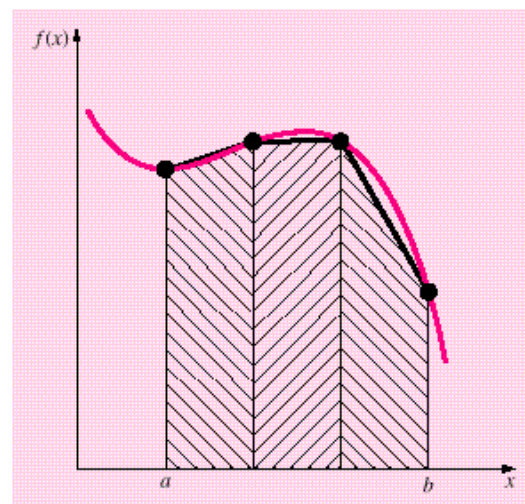
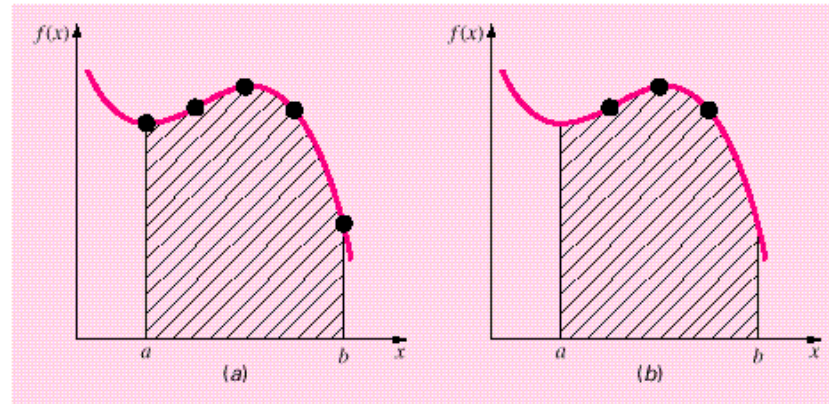


FIGURE 16.5

The approximation of an integral by the area under three straight-line segments.



**FIGURE 16.6**

The difference between $[\cdot]$ closed and (\cdot) open integration formulas.

The integral can also be approximated using a series of polynomials applied piecewise to the function or data over segments of constant length. For example, in Fig. 16.5, three straight-line segments are used to approximate the integral. Higher-order polynomials can be utilized for the same purpose.

Closed and open forms of the Newton-Cotes formulas are available. The *closed forms* are those where the data points at the beginning and end of the limits of integration are known (Fig. 16.6a). The *open forms* have integration limits that extend beyond the range of the data (Fig. 16.6b). This chapter emphasizes the closed forms. However, material on open Newton-Cotes formulas is briefly introduced in Section 16.7.

16.3 THE TRAPEZOIDAL RULE

The *trapezoidal rule* is the first of the Newton-Cotes closed integration formulas. It corresponds to the case where the polynomial in Eq. (16.8) is first-order:

$$I = \int_a^b \left[f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \right] dx \quad (16.10)$$

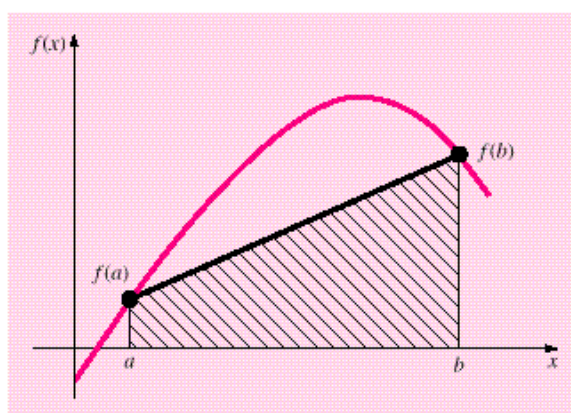
The result of the integration is

$$I = (b - a) \frac{f(a) + f(b)}{2} \quad (16.11)$$

which is called the *trapezoidal rule*.

Geometrically, the trapezoidal rule is equivalent to approximating the area of the trapezoid under the straight line connecting $f(a)$ and $f(b)$ in Fig. 16.7. Recall from geometry that the formula for computing the area of a trapezoid is the height times the average of the bases. In our case, the concept is the same but the trapezoid is on its side. Therefore, the integral estimate can be represented as

$$I = \text{width} \times \text{average height} \quad (16.12)$$

**FIGURE 16.7**

Graphical depiction of the trapezoidal rule.

or

$$I = (b - a) \times \text{average height} \quad (16.13)$$

where, for the trapezoidal rule, the average height is the average of the function values at the end points, or $[f(a) + f(b)]/2$.

All the Newton-Cotes closed formulas can be expressed in the general format of Eq. (16.13). That is, they differ only with respect to the formulation of the average height.

16.3.1 Error of the Trapezoidal Rule

When we employ the integral under a straight-line segment to approximate the integral under a curve, we obviously can incur an error that may be substantial (Fig. 16.8). An estimate for the local truncation error of a single application of the trapezoidal rule is

$$E_t = -\frac{1}{12} f''(\xi)(b - a)^3 \quad (16.14)$$

where ξ lies somewhere in the interval from a to b . Equation (16.14) indicates that if the function being integrated is linear, the trapezoidal rule will be exact because the second derivative of a straight line is zero. Otherwise, for functions with second- and higher-order derivatives (i.e., with curvature), some error can occur.

EXAMPLE 16.1 Single Application of the Trapezoidal Rule

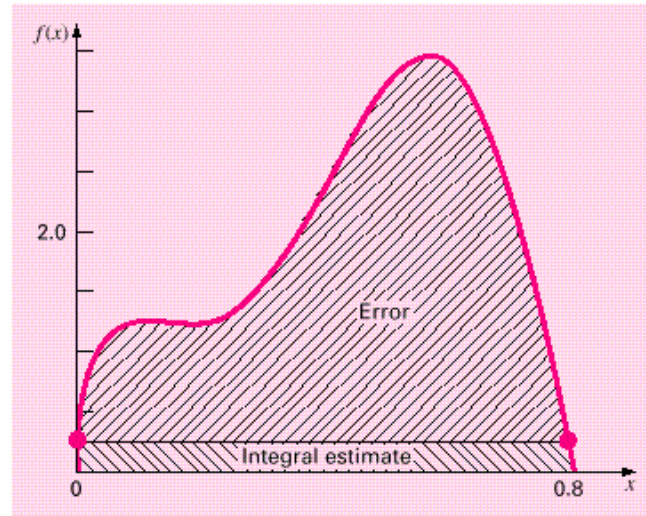
Problem Statement. Use Eq. (16.11) to numerically integrate

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from $a = 0$ to $b = 0.8$. Note that the exact value of the integral can be determined analytically to be 1.640533.

16.3 THE TRAPEZOIDAL RULE

287

**FIGURE 16.8**

Graphical depiction of the use of a single application of the trapezoidal rule to approximate the integral of $f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$ from $x = 0$ to 0.8 .

Solution. The function values $f(0) = 0.2$ and $f(0.8) = 0.232$ can be substituted into Eq. (16.11) to yield

$$I = (0.8 - 0) \frac{0.2 + 0.232}{2} = 0.1728$$

which represents an error of $E_t = 1.640533 - 0.1728 = 1.467733$, which corresponds to a percent relative error of $\varepsilon_t = 89.5\%$. The reason for this large error is evident from the graphical depiction in Fig. 16.8. Notice that the area under the straight line neglects a significant portion of the integral lying above the line.

In actual situations, we would have no foreknowledge of the true value. Therefore, an approximate error estimate is required. To obtain this estimate, the function's second derivative over the interval can be computed by differentiating the original function twice to give

$$f''(x) = -400 + 4,050x - 10,800x^2 + 8,000x^3$$

The average value of the second derivative can be computed as [Eq. (16.7)]

$$\bar{f}''(x) = \frac{\int_0^{0.8} (-400 + 4,050x - 10,800x^2 + 8,000x^3) dx}{0.8 - 0} = -60$$

which can be substituted into Eq. (16.14) to yield

$$E_a = -\frac{1}{12}(-60)(0.8)^3 = 2.56$$

which is of the same order of magnitude and sign as the true error. A discrepancy does exist, however, because of the fact that for an interval of this size, the average second derivative is not necessarily an accurate approximation of $f''(\xi)$. Thus, we denote that the error is approximate by using the notation E_a , rather than exact by using E_T .

16.3.2 The Composite Trapezoidal Rule

One way to improve the accuracy of the trapezoidal rule is to divide the integration interval from a to b into a number of segments and apply the method to each segment (Fig. 16.9). The areas of individual segments can then be added to yield the integral for the entire interval. The resulting equations are called *composite*, or *multiple-application*, *integration formulas*.

Figure 16.9 shows the general format and nomenclature we will use to characterize composite integrals. There are $n + 1$ equally spaced base points ($x_0, x_1, x_2, \dots, x_n$). Consequently, there are n segments of equal width:

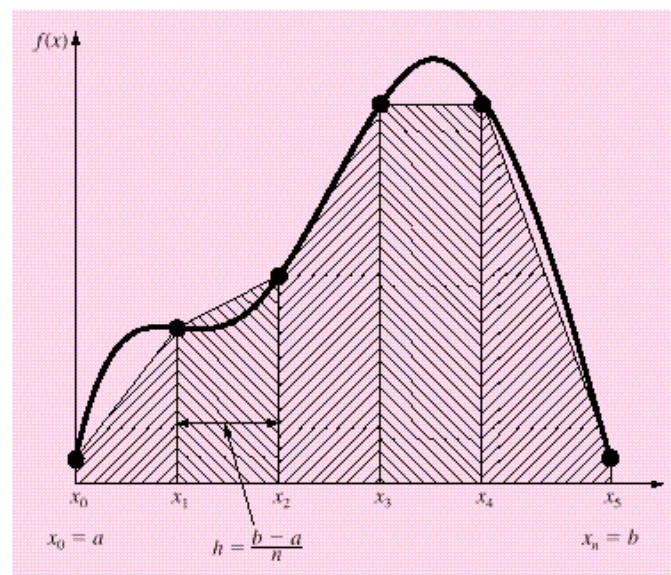
$$h = \frac{b - a}{n} \quad (16.15)$$

If a and b are designated as x_0 and x_n , respectively, the total integral can be represented as

$$I = \int_{x_0}^{x_1} f(x) dx + \int_{x_1}^{x_2} f(x) dx + \cdots + \int_{x_{n-1}}^{x_n} f(x) dx$$

FIGURE 16.9

Composite trapezoidal rule.



16.3 THE TRAPEZOIDAL RULE

289

Substituting the trapezoidal rule for each integral yields

$$I = h \frac{f(x_0) + f(x_1)}{2} + h \frac{f(x_1) + f(x_2)}{2} + \cdots + h \frac{f(x_{n-1}) + f(x_n)}{2} \quad (16.16)$$

or, grouping terms:

$$I = \frac{h}{2} \left[f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right] \quad (16.17)$$

or, using Eq. (16.15) to express Eq. (16.17) in the general form of Eq. (16.13):

$$I = \underbrace{(b-a)}_{\text{Width}} \underbrace{\frac{f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n)}{2n}}_{\text{Average height}} \quad (16.18)$$

Because the summation of the coefficients of $f(x)$ in the numerator divided by $2n$ is equal to 1, the average height represents a weighted average of the function values. According to Eq. (16.18), the interior points are given twice the weight of the two end points $f(x_0)$ and $f(x_n)$.

An error for the composite trapezoidal rule can be obtained by summing the individual errors for each segment to give

$$E_t = -\frac{(b-a)^3}{12n^3} \sum_{i=1}^n f''(\xi_i) \quad (16.19)$$

where $f''(\xi_i)$ is the second derivative at a point ξ_i located in segment i . This result can be simplified by estimating the mean or average value of the second derivative for the entire interval as

$$\bar{f}'' \cong \frac{\sum_{i=1}^n f''(\xi_i)}{n} \quad (16.20)$$

Therefore $\sum f''(\xi_i) \cong n \bar{f}''$ and Eq. (16.19) can be rewritten as

$$E_a = -\frac{(b-a)^3}{12n^2} \bar{f}'' \quad (16.21)$$

Thus, if the number of segments is doubled, the truncation error will be quartered. Note that Eq. (16.21) is an approximate error because of the approximate nature of Eq. (16.20).

EXAMPLE 16.2 Composite Application of the Trapezoidal Rule

Problem Statement. Use the two-segment trapezoidal rule to estimate the integral of

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from $a = 0$ to $b = 0.8$. Employ Eq. (16.21) to estimate the error. Recall that the exact value of the integral is 1.640533.

Solution. For $n = 2$ ($h = 0.4$):

$$f(0) = 0.2 \quad f(0.4) = 2.456 \quad f(0.8) = 0.232$$

$$I = 0.8 \frac{0.2 + 2(2.456) + 0.232}{4} = 1.0688$$

$$E_t = 1.640533 - 1.0688 = 0.57173 \quad \varepsilon_t = 34.9\%$$

$$E_a = -\frac{0.8^3}{12(2)^2}(-60) = 0.64$$

where -60 is the average second derivative determined previously in Example 16.1.

The results of the previous example, along with three- through ten-segment applications of the trapezoidal rule, are summarized in Table 16.1. Notice how the error decreases as the number of segments increases. However, also notice that the rate of decrease is gradual. This is because the error is inversely related to the square of n [Eq. (16.21)]. Therefore, doubling the number of segments quarters the error. In subsequent sections we develop higher-order formulas that are more accurate and that converge more quickly on the true integral as the segments are increased. However, before investigating these formulas, we will first discuss how MATLAB can be used to implement the trapezoidal rule.

16.3.3 MATLAB M-file: trap

A simple algorithm to implement the composite trapezoidal rule can be written as in Fig. 16.10. The function to be integrated is passed into the M-file along with the limits of integration and the number of segments. A loop is then employed to generate the integral following Eq. (16.18).

An application of the M-file can be developed to determine the distance fallen by the free-falling bungee jumper in the first 3 s by evaluating the integral of Eq. (16.3). For this example, assume the following parameter values: $g = 9.81 \text{ m/s}^2$, $m = 68.1 \text{ kg}$, and

TABLE 16.1 Results for the composite trapezoidal rule to estimate the integral of $f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$ from $x = 0$ to 8. The exact value is 1.640533.

			(%)
2	0.4	1.0688	34.9
3	0.2667	1.3695	16.5
4	0.2	1.4848	9.5
5	0.16	1.5399	6.1
6	0.1333	1.5703	4.3
7	0.1143	1.5887	3.2
8	0.1	1.6008	2.4
9	0.0889	1.6091	1.9
10	0.08	1.6150	1.6

16.3 THE TRAPEZOIDAL RULE

291

```
function I = trap(func,a,b,n)
% trap(func,a,b,n):
%   composite trapezoidal rule.
% input:
%   func = name of function to be integrated
%   a, b = integration limits
%   n = number of segments
% output:
%   I = integral estimate

x = a;
h = (b - a)/n;
s = feval(func,a);
for j = 1 : n-1
    x = x + h;
    s = s + 2*feval(func,x);
end
s = s + feval(func,b);
I = (b - a) * s/(2*n);
```

FIGURE 16.10

M-file to implement the composite trapezoidal rule.

$c_d = 0.25$ kg/m. Note that the exact value of the integral can be computed with Eq. (16.4) as 41.94805.

The function to be integrated can be developed as an M-file or with an inline function,

```
>> v = inline('sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t)')
v =
    Inline function:
    v(t) = sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t)
```

First, let's evaluate the integral with a crude five-segment approximation:

```
format long
>> trap(v,0,3,5)

ans =
    41.86992959072735
```

As would be expected, this result has a relatively high true error of 18.6%. To obtain a more accurate result, we can use a very fine approximation based on 10,000 segments:

```
>> trap(v,0,3,10000)

x =
    41.94804999917528
```

which is very close to the true value.

16.4 SIMPSON'S RULES

Aside from applying the trapezoidal rule with finer segmentation, another way to obtain a more accurate estimate of an integral is to use higher-order polynomials to connect the points. For example, if there is an extra point midway between $f(a)$ and $f(b)$, the three points can be connected with a parabola (Fig. 16.11a). If there are two points equally spaced between $f(a)$ and $f(b)$, the four points can be connected with a third-order polynomial (Fig. 16.11b). The formulas that result from taking the integrals under these polynomials are called *Simpson's rules*.

16.4.1 Simpson's 1/3 Rule

Simpson's 1/3 rule corresponds to the case where the polynomial in Eq. (16.8) is second-order:

$$I = \int_{x_0}^{x_2} \left[\frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f(x_0) + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f(x_1) + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f(x_2) \right] dx$$

where a and b are designated as x_0 and x_2 , respectively. The result of the integration is

$$I = \frac{h}{3} [f(x_0) + 4f(x_1) + f(x_2)] \quad (16.22)$$

where, for this case, $h = (b - a)/2$. This equation is known as *Simpson's 1/3 rule*. The label "1/3" stems from the fact that h is divided by 3 in Eq. (16.22). Simpson's 1/3 rule can

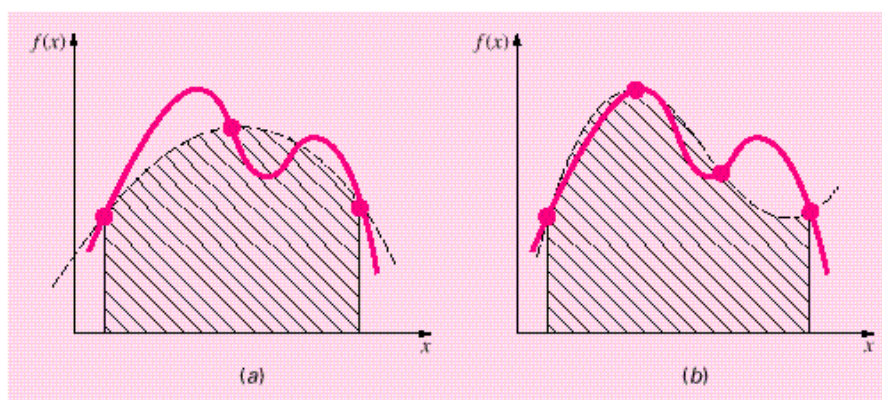


FIGURE 16.11

(· ·) Graphical depiction of Simpson's 1/3 rule: It consists of taking the area under a parabola connecting three points. (· · ·) Graphical depiction of Simpson's 3/8 rule: It consists of taking the area under a cubic equation connecting four points.

16.4 SIMPSON'S RULES

293

also be expressed using the format of Eq. (16.13):

$$I = (b - a) \frac{f(x_0) + 4f(x_1) + f(x_2)}{6} \quad (16.23)$$

where $a = x_0$, $b = x_2$, and x_1 = the point midway between a and b , which is given by $(a + b)/2$. Notice that, according to Eq. (16.23), the middle point is weighted by two-thirds and the two end points by one-sixth.

It can be shown that a single-segment application of Simpson's 1/3 rule has a truncation error of

$$E_t = -\frac{1}{90} h^5 f^{(4)}(\xi)$$

or, because $h = (b - a)/2$:

$$E_t = -\frac{(b - a)^5}{2880} f^{(4)}(\xi) \quad (16.24)$$

where ξ lies somewhere in the interval from a to b . Thus, Simpson's 1/3 rule is more accurate than the trapezoidal rule. However, comparison with Eq. (16.14) indicates that it is more accurate than expected. Rather than being proportional to the third derivative, the error is proportional to the fourth derivative. Consequently, Simpson's 1/3 rule is third-order accurate even though it is based on only three points. In other words, it yields exact results for cubic polynomials even though it is derived from a parabola!

EXAMPLE 16.3 Single Application of Simpson's 1/3 Rule

Problem Statement. Use Eq. (16.23) to integrate

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from $a = 0$ to $b = 0.8$. Employ Eq. (16.24) to estimate the error. Recall that the exact integral is 1.640533.

Solution. $n = 2$ ($h = 0.4$):

$$f(0) = 0.2 \quad f(0.4) = 2.456 \quad f(0.8) = 0.232$$

$$I = 0.8 \frac{0.2 + 4(2.456) + 0.232}{6} = 1.367467$$

$$E_t = 1.640533 - 1.367467 = 0.2730667 \quad \varepsilon_t = 16.6\%$$

which is approximately five times more accurate than for a single application of the trapezoidal rule (Example 16.1). The approximate error can be estimated as

$$E_a = -\frac{0.8^5}{2880} (-2400) = 0.2730667$$

where -2400 is the average fourth derivative for the interval. As was the case in Example 16.1, the error is approximate (E_a) because the average fourth derivative is generally not an exact estimate of $f^{(4)}(\xi)$. However, because this case deals with a fifth-order polynomial, the result matches exactly.

16.4.2 The Composite Simpson's 1/3 Rule

Just as with the trapezoidal rule, Simpson's rule can be improved by dividing the integration interval into a number of segments of equal width (Fig. 16.12). The total integral can be represented as

$$I = \int_{x_0}^{x_2} f(x) dx + \int_{x_2}^{x_4} f(x) dx + \cdots + \int_{x_{n-2}}^{x_n} f(x) dx \quad (16.25)$$

Substituting Simpson's 1/3 rule for each integral yields

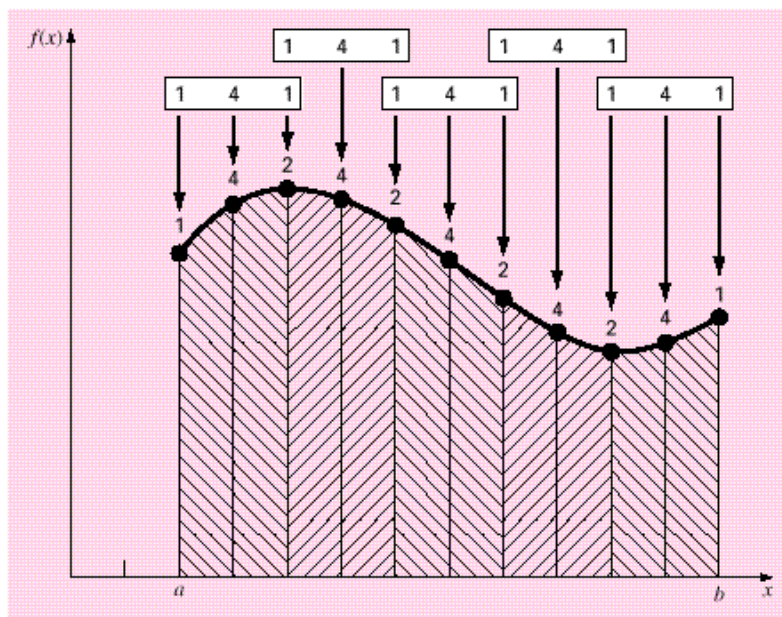
$$I = 2h \frac{f(x_0) + 4f(x_1) + f(x_2)}{6} + 2h \frac{f(x_2) + 4f(x_3) + f(x_4)}{6} \\ + \cdots + 2h \frac{f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)}{6}$$

or, grouping terms and using Eq. (16.15):

$$I = (b - a) \frac{f(x_0) + 4 \sum_{i=1,3,5}^{n-1} f(x_i) + 2 \sum_{j=2,4,6}^{n-2} f(x_j) + f(x_n)}{3n} \quad (16.26)$$

FIGURE 16.12

Composite Simpson's 1/3 rule. The relative weights are depicted above the function values. Note that the method can be employed only if the number of segments is even.



16.4 SIMPSON'S RULES

295

Notice that, as illustrated in Fig. 16.12, an even number of segments must be utilized to implement the method. In addition, the coefficients “4” and “2” in Eq. (16.26) might seem peculiar at first glance. However, they follow naturally from Simpson’s 1/3 rule. As illustrated in Fig. 16.12, the odd points represent the middle term for each application and hence carry the weight of four from Eq. (16.23). The even points are common to adjacent applications and hence are counted twice.

An error estimate for the composite Simpson’s rule is obtained in the same fashion as for the trapezoidal rule by summing the individual errors for the segments and averaging the derivative to yield

$$E_a = -\frac{(b-a)^5}{180n^4} \bar{f}^{(4)} \quad (16.27)$$

where $f^{(4)}$ is the average fourth derivative for the interval.

EXAMPLE 16.4 Composite Simpson’s 1/3 Rule

Problem Statement. Use Eq. (16.26) with $n = 4$ to estimate the integral of

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from $a = 0$ to $b = 0.8$. Employ Eq. (16.27) to estimate the error. Recall that the exact integral is 1.640533.

Solution. $n = 4 (h = 0.2)$:

$$\begin{aligned} f(0) &= 0.2 & f(0.2) &= 1.288 \\ f(0.4) &= 2.456 & f(0.6) &= 3.464 \\ f(0.8) &= 0.232 \end{aligned}$$

From Eq. (16.26):

$$I = 0.8 \frac{0.2 + 4(1.288 + 3.464) + 2(2.456) + 0.232}{12} = 1.623467$$

$$E_t = 1.640533 - 1.623467 = 0.017067 \quad \varepsilon_t = 1.04\%$$

The estimated error (Eq. 16.27) is

$$E_a = -\frac{(0.8)^5}{180(4)^4}(-2400) = 0.017067$$

which is exact (as was also the case for Example 16.3).

As in Example 16.4, the composite version of Simpson’s 1/3 rule is considered superior to the trapezoidal rule for most applications. However, as mentioned previously, it is limited to cases where the values are equispaced. Further, it is limited to situations where there are an even number of segments and an odd number of points. Consequently, as discussed in Section 16.4.3, an odd-segment–even-point formula known as Simpson’s 3/8 rule can be used in conjunction with the 1/3 rule to permit evaluation of both even and odd numbers of equispaced segments.

16.4.3 Simpson's 3/8 Rule

In a similar manner to the derivation of the trapezoidal and Simpson's 1/3 rule, a third-order Lagrange polynomial can be fit to four points and integrated to yield

$$I = \frac{3h}{8} [f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)]$$

where $h = (b - a)/3$. This equation is known as *Simpson's 3/8 rule* because h is multiplied by $3/8$. It is the third Newton-Cotes closed integration formula. The 3/8 rule can also be expressed in the form of Eq. (16.13):

$$I = (b - a) \frac{f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)}{8} \quad (16.28)$$

Thus, the two interior points are given weights of three-eighths, whereas the end points are weighted with one-eighth. Simpson's 3/8 rule has an error of

$$E_T = -\frac{3}{80} h^5 f^{(4)}(\xi)$$

or, because $h = (b - a)/3$:

$$E_T = -\frac{(b - a)^5}{6480} f^{(4)}(\xi) \quad (16.29)$$

Because the denominator of Eq. (16.29) is larger than for Eq. (16.24), the 3/8 rule is somewhat more accurate than the 1/3 rule.

Simpson's 1/3 rule is usually the method of preference because it attains third-order accuracy with three points rather than the four points required for the 3/8 version. However, the 3/8 rule has utility when the number of segments is odd. For instance, in Example 16.4 we used Simpson's rule to integrate the function for four segments. Suppose that you desired an estimate for five segments. One option would be to use a composite version of the trapezoidal rule as was done in Example 16.2. This may not be advisable, however, because of the large truncation error associated with this method. An alternative would be to apply Simpson's 1/3 rule to the first two segments and Simpson's 3/8 rule to the last three (Fig. 16.13). In this way, we could obtain an estimate with third-order accuracy across the entire interval.

EXAMPLE 16.5 Simpson's 3/8 Rule

Problem Statement. (a) Use Simpson's 3/8 rule to integrate

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from $a = 0$ to $b = 0.8$. (b) Use it in conjunction with Simpson's 1/3 rule to integrate the same function for five segments.

Solution. (a) A single application of Simpson's 3/8 rule requires four equally spaced points:

$$\begin{aligned} f(0) &= 0.2 & f(0.2667) &= 1.432724 \\ f(0.5333) &= 3.487177 & f(0.8) &= 0.232 \end{aligned}$$

16.4 SIMPSON'S RULES

297

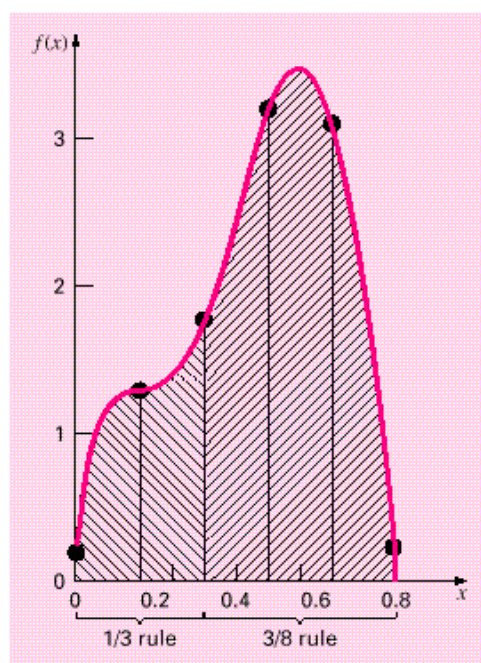


FIGURE 16.13

Illustration of how Simpson's 1/3 and 3/8 rules can be applied in tandem to handle multiple applications with odd numbers of intervals.

Using Eq. (16.28):

$$I = 0.8 \frac{0.2 + 3(1.432724 + 3.487177) + 0.232}{8} = 1.51970$$

(b) The data needed for a five-segment application ($h = 0.16$) is

$$\begin{aligned} f(0) &= 0.2 & f(0.16) &= 1.296919 \\ f(0.32) &= 1.743393 & f(0.48) &= 3.186015 \\ f(0.64) &= 3.181929 & f(0.80) &= 0.232 \end{aligned}$$

The integral for the first two segments is obtained using Simpson's 1/3 rule:

$$I = 0.32 \frac{0.2 + 4(1.296919) + 1.743393}{6} = 0.3803237$$

For the last three segments, the 3/8 rule can be used to obtain

$$I = 0.48 \frac{1.743393 + 3(3.186015 + 3.181929) + 0.232}{8} = 1.264754$$

The total integral is computed by summing the two results:

$$I = 0.3803237 + 1.264754 = 1.645077$$

16.5 HIGHER-ORDER NEWTON-COTES FORMULAS

As noted previously, the trapezoidal rule and both of Simpson's rules are members of a family of integrating equations known as the Newton-Cotes closed integration formulas. Some of the formulas are summarized in Table 16.2 along with their truncation-error estimates.

Notice that, as was the case with Simpson's 1/3 and 3/8 rules, the five- and six-point formulas have the same order error. This general characteristic holds for the higher-point formulas and leads to the result that the even-segment-odd-point formulas (e.g., 1/3 rule and Boole's rule) are usually the methods of preference.

However, it must also be stressed that, in engineering and science practice, the higher-order (i.e., greater than four-point) formulas are not commonly used. Simpson's rules are sufficient for most applications. Accuracy can be improved by using the composite version. Furthermore, when the function is known and high accuracy is required, methods such as Romberg integration or Gauss quadrature, described in Chap. 17, offer viable and attractive alternatives.

16.6 INTEGRATION WITH UNEQUAL SEGMENTS

To this point, all formulas for numerical integration have been based on equispaced data points. In practice, there are many situations where this assumption does not hold and we must deal with unequal-sized segments. For example, experimentally derived data is often of this type. For these cases, one method is to apply the trapezoidal rule to each segment and sum the results:

$$I = h_1 \frac{f(x_0) + f(x_1)}{2} + h_2 \frac{f(x_1) + f(x_2)}{2} + \cdots + h_n \frac{f(x_{n-1}) + f(x_n)}{2} \quad (16.30)$$

where h_i = the width of segment i . Note that this was the same approach used for the composite trapezoidal rule. The only difference between Eqs. (16.16) and (16.30) is that the h 's in the former are constant.

TABLE 16.2 Newton-Cotes closed integration formulas. The formulas are presented in the format of Eq. (16.13) so that the weighting of the data points to estimate the average height is apparent. The step size is given by $h = (b - a)/n$.

Segments ()	Points	Name	Formula	Truncation Error
1	2	Trapezoidal rule	$(b - a) \frac{f(x_0) + f(x_1)}{2}$	$-(1/12)h^3 f''(\xi)$
2	3	Simpson's 1/3 rule	$(b - a) \frac{f(x_0) + 4f(x_1) + f(x_2)}{6}$	$-(1/90)h^5 f^{(4)}(\xi)$
3	4	Simpson's 3/8 rule	$(b - a) \frac{f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)}{8}$	$-(3/80)h^5 f^{(4)}(\xi)$
4	5	Boole's rule	$(b - a) \frac{7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4)}{90}$	$-(8/945)h^7 f^{(6)}(\xi)$
5	6		$(b - a) \frac{19f(x_0) + 75f(x_1) + 50f(x_2) + 50f(x_3) + 75f(x_4) + 19f(x_5)}{288}$	$-(275/12,096)h^7 f^{(6)}(\xi)$

EXAMPLE 16.6 Trapezoidal Rule with Unequal Segments

Problem Statement. The information in Table 16.3 was generated using the same polynomial employed in Example 16.1. Use Eq. (16.30) to determine the integral for this data. Recall that the correct answer is 1.640533.

TABLE 16.3 Data for $f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$, with unequally spaced values of x .

x	$f(x)$	x	$f(x)$
0.00	0.200000	0.44	2.842985
0.12	1.309729	0.54	3.507297
0.22	1.305241	0.64	3.181929
0.32	1.743393	0.70	2.363000
0.36	2.074903	0.80	0.232000
0.40	2.456000		

Solution. Applying Eq. (16.30) yields

$$I = 0.12 \frac{0.2 + 1.309729}{2} + 0.10 \frac{1.309729 + 1.305241}{2} \\ + \cdots + 0.10 \frac{2.363 + 0.232}{2} = 1.594801$$

which represents an absolute percent relative error of $\varepsilon_t = 2.8\%$.

16.6.1 MATLAB M-file: trapuneq

A simple algorithm to implement the trapezoidal rule for unequally spaced data can be written as in Fig. 16.14. Two vectors, x and y , holding the independent and dependent variables are passed into the M-file. Two error traps are included to ensure that (a) the two vectors are of the same length and (b) the x 's are in ascending order. A loop is employed to generate the integral. Notice that we have modified the subscripts from those of Eq. (16.30) to account for the fact that MATLAB does not allow zero subscripts in arrays.

An application of the M-file can be developed for the same problem that was solved in Example 16.6:

```
>> x = [0 .12 .22 .32 .36 .4 .44 .54 .64 .7 .8];
>> y = 0.2+25*x-200*x.^2+675*x.^3-900*x.^4+400*x.^5;
>> trapuneq(x,y)
```

```
ans =
    1.5948
```

which is identical to the result obtained in Example 16.6.

```
function integr = trapuneq(x,y)
% trapuneq(x,y):
%   Applies the trapezoidal rule to determine the integral
%   for n data points (x, y) where x must be in ascending
%   order
% input:
%   x = independent variable
%   y = dependent variable
% output:
%   integr = integral

n = length(x);
if length(y)~=n, error('x and y must be same length'); end
s = 0;
for i = 1:n-1
    if x(i+1)<x(i)
        error('x values must be in ascending order');
    end
    s = s + (x(i+1)-x(i))*(y(i)+y(i+1))/2;
end
integr = s;
```

FIGURE 16.14

M-file to implement the trapezoidal rule for unequally spaced data.

16.6.2 MATLAB Function: `trapz`

MATLAB has a built-in function that evaluates integrals for data in the same fashion as the M-file we just presented in Fig. 16.14. It has the general syntax

```
s = trapz(x, y)
```

where the two vectors, `x` and `y`, hold the independent and dependent variables, respectively. Here is a simple MATLAB session that uses this function to integrate the data from Table 16.3:

```
>> x = [0 .12 .22 .32 .36 .4 .44 .54 .64 .7 .8];
>> y = 0.2+25*x-200*x.^2+675*x.^3-900*x.^4+400*x.^5;
>> trapz(x,y)

ans =

    1.5948
```

16.7 OPEN METHODS

Recall from Fig. 16.6*b* that open integration formulas have limits that extend beyond the range of the data. Table 16.4 summarizes the *Newton-Cotes open integration formulas*. The formulas are expressed in the form of Eq. (16.13) so that the weighting factors are evident.

TABLE 16.4 Newton-Cotes open integration formulas. The formulas are presented in the format of Eq. (16.13) so that the weighting of the data points to estimate the average height is apparent. The step size is given by $h = (b - a)/n$.

Segments (\cdot)	Points	Name	Formula	Truncation Error
2	1	Midpoint method	$(b - a)f(x_1)$	$(1/3)h^3 f''(\xi)$
3	2		$(b - a) \frac{f(x_1) + f(x_2)}{2}$	$(3/4)h^3 f''(\xi)$
4	3		$(b - a) \frac{2f(x_1) - f(x_2) + 2f(x_3)}{3}$	$(14/45)h^5 f^{(4)}(\xi)$
5	4		$(b - a) \frac{11f(x_1) + f(x_2) + f(x_3) + 11f(x_4)}{24}$	$(95/144)h^5 f^{(4)}(\xi)$
6	5		$(b - a) \frac{11f(x_1) - 14f(x_2) + 26f(x_3) - 14f(x_4) + 11f(x_5)}{20}$	$(41/140)h^7 f^{(6)}(\xi)$

As with the closed versions, successive pairs of the formulas have the same-order error. The even-segment-odd-point formulas are usually the methods of preference because they require fewer points to attain the same accuracy as the odd-segment-even-point formulas.

The open formulas are not often used for definite integration. However, they have utility for analyzing improper integrals. In addition, they will have relevance to our discussion of methods for solving ordinary differential equations in Chaps. 18 and 19.

16.8 MULTIPLE INTEGRALS

Multiple integrals was widely used in engineering. For example, a general equation to compute the average of a two-dimensional function can be written as [recall Eq. (16.7)]

$$\bar{f} = \frac{\int_c^d \left(\int_a^b f(x, y) dx \right) dy}{(d - c)(b - a)} \quad (16.31)$$

The numerator is called a *double integral*.

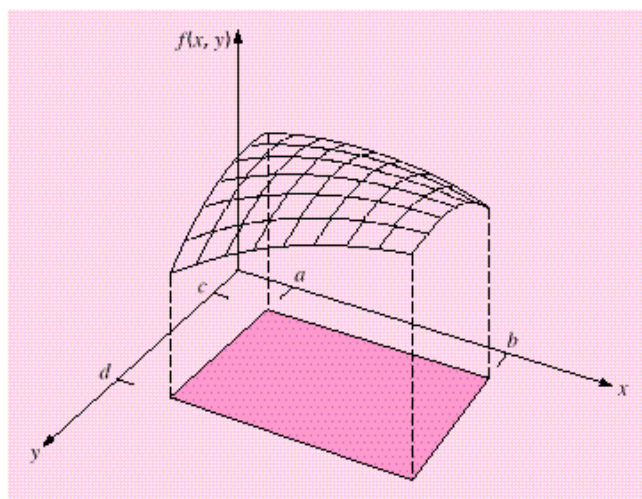
The techniques discussed in this chapter (and Chap. 17) can be readily employed to evaluate multiple integrals. A simple example would be to take the double integral of a function over a rectangular area (Fig. 16.15).

Recall from calculus that such integrals can be computed as iterated integrals:

$$\int_c^d \left(\int_a^b f(x, y) dx \right) dy = \int_a^b \left(\int_c^d f(x, y) dy \right) dx \quad (16.32)$$

Thus, the integral in one of the dimensions is evaluated first. The result of this first integration is integrated in the second dimension. Equation (16.32) states that the order of integration is not important.

A numerical double integral would be based on the same idea. First, methods such as the composite trapezoidal or Simpson's rule would be applied in the first dimension with each value of the second dimension held constant. Then the method would be applied to integrate the second dimension. The approach is illustrated in the following example.

**FIGURE 16.15**

Double integral as the area under the function surface.

EXAMPLE 16.7 Using Double Integral to Determine Average Temperature

Problem Statement. Suppose that the temperature of a rectangular heated plate is described by the following function:

$$T(x, y) = 2xy + 2x - x^2 - 2y^2 + 40$$

If the plate is 8 m long (x dimension) and 6 m wide (y dimension), compute the average temperature.

Solution. First, let us merely use two-segment applications of the trapezoidal rule in each dimension. The temperatures at the necessary x and y values are depicted in Fig. 16.16. Note that a simple average of these values is 47.33. The function can also be evaluated analytically to yield a result of 58.66667.

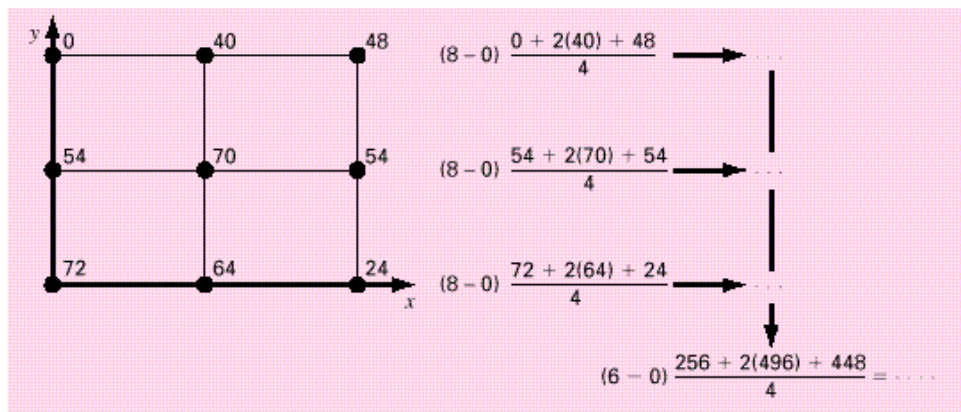
To make the same evaluation numerically, the trapezoidal rule is first implemented along the x dimension for each y value. These values are then integrated along the y dimension to give the final result of 2688. Dividing this by the area yields the average temperature as $2688/(6 \times 8) = 56$.

Now we can apply a single-segment Simpson's 1/3 rule in the same fashion. This results in an integral of 2816 and an average of 58.66667, which is exact. Why does this occur? Recall that Simpson's 1/3 rule yielded perfect results for cubic polynomials. Since the highest-order term in the function is second order, the same exact result occurs for the present case.

For higher-order algebraic functions as well as transcendental functions, it would be necessary to use composite applications to attain accurate integral estimates. In addition, Chap. 17 introduces techniques that are more efficient than the Newton-Cotes formulas for evaluating integrals of given functions. These often provide a superior means to implement the numerical integrations for multiple integrals.

FIGURE 16.16

Numerical evaluation of a double integral using the two-segment trapezoidal rule.



PROBLEMS

16.1 Derive Eq. (16.4) by integrating Eq. (16.3).

16.2 Evaluate the following integral:

$$\int_0^4 (1 - e^{-2x}) dx$$

(a) analytically, (b) single application of the trapezoidal rule, (c) composite trapezoidal rule with $n = 2$ and 4, (d) single application of Simpson's 1/3 rule, (e) composite Simpson's 1/3 rule with $n = 4$, and (f) Simpson's 3/8 rule. For each of the numerical estimates (b) through (f), determine the percent relative error based on (a).

16.3 Evaluate the following integral:

$$\int_0^{\pi/2} (6 + 3 \cos x) dx$$

(a) analytically, (b) single application of the trapezoidal rule, (c) composite trapezoidal rule with $n = 2$ and 4, (d) single application of Simpson's 1/3 rule, (e) composite Simpson's 1/3 rule with $n = 4$, and (f) Simpson's 3/8 rule. For each of the numerical estimates (b) through (f), determine the percent relative error based on (a).

16.4 Evaluate the following integral:

$$\int_{-2}^4 (1 - x - 4x^3 + 2x^5) dx$$

(a) analytically, (b) single application of the trapezoidal rule, (c) composite trapezoidal rule with $n = 2$ and 4, (d) single

application of Simpson's 1/3 rule, (e) Simpson's 3/8 rule, and (f) Boole's rule. For each of the numerical estimates (b) through (f), determine the percent relative error based on (a).

16.5 The function

$$f(x) = e^{-x}$$

can be used to generate the following table of unequally spaced data:

x	0	0.1	0.3	0.5	0.7	0.95	1.2
$f(x)$	1	0.9048	0.7408	0.6065	0.4966	0.3867	0.3012

Evaluate the integral from $a = 0$ to $b = 1.2$ using (a) analytical means, (b) the trapezoidal rule, and (c) a combination of the trapezoidal and Simpson's rules wherever possible to attain the highest accuracy. For (b) and (c), compute the percent relative error.

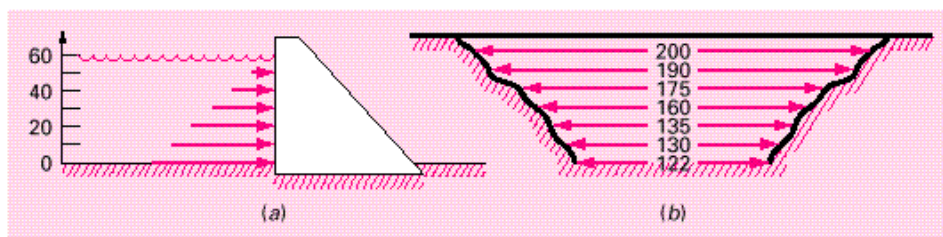
16.6 Evaluate the double integral

$$\int_{-2}^2 \int_0^4 (x^2 - 3y^2 + xy^3) dx dy$$

(a) analytically, (b) using the composite trapezoidal rule with $n = 2$, and (c) using single applications of Simpson's 1/3 rule. For (b) and (c), compute the percent relative error.

16.7 Evaluate the triple integral

$$\int_{-4}^4 \int_0^6 \int_{-1}^3 (x^3 - 2yz) dx dy dz$$

**FIGURE P16.9**

Water exerting pressure on the upstream face of a dam: [-] side view showing force increasing linearly with depth; [-] front view showing width of dam in meters.

(a) analytically, and (b) using single applications of Simpson's 1/3 rule. For (b), compute the percent relative error.

16.8 Determine the distance traveled from the following velocity data:

1	2	3.25	4.5	6	7	8	8.5	9.3	10
5	6	5.5	7	8.5	6	6	7	7	5

(a) Use the trapezoidal rule.

(b) Fit the data with a cubic equation using polynomial regression. Integrate the cubic equation to determine the distance.

16.9 Water exerts pressure on the upstream face of a dam as shown in Fig. P16.9. The pressure can be characterized by

$$p(z) = \rho g(D - z)$$

where $p(z)$ = pressure in pascals (or N/m^2) exerted at an elevation z meters above the reservoir bottom; ρ = density of water, which for this problem is assumed to be a constant 10^3 kg/m^3 ; g = acceleration due to gravity (9.81 m/s^2); and D = elevation (in m) of the water surface above the reservoir bottom. According to Eq. (P16.9), pressure increases linearly with depth, as depicted in Fig. P16.9a. Omitting atmospheric pressure (because it works against both sides of the dam face and essentially cancels out), the total force f_t can be determined by multiplying pressure times the area of the dam face (as shown in Fig. P16.9b). Because both pressure and area vary with elevation, the total force is obtained

by evaluating

$$f_t = \int_0^D \rho g w(z)(D - z) dz$$

where $w(z)$ = width of the dam face (m) at elevation z (Fig. P16.9b). The line of action can also be obtained by evaluating

$$d = \frac{\int_0^D \rho g z w(z)(D - z) dz}{\int_0^D \rho g w(z)(D - z) dz}$$

Use Simpson's rule to compute f_t and d .

16.10 The force on a sailboat mast can be represented by the following function:

$$f(z) = 200 \left(\frac{z}{7 + z} \right) e^{-2.5z/H}$$

where z = the elevation above the deck and H = the height of the mast. The total force F exerted on the mast can be determined by integrating this function over the height of the mast:

$$F = \int_0^H f(z) dz$$

The line of action can also be determined by integration:

$$d = \frac{\int_0^H z f(z) dz}{\int_0^H f(z) dz}$$

(a) Use the composite trapezoidal rule to compute F and d for the case where $H = 30$ ($n = 6$).

(b) Repeat (a), but use the composite Simpson's 1/3 rule.

Numerical Integration of Functions

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to numerical methods for integrating given functions. Specific objectives and topics covered are

- Understanding how Richardson extrapolation provides a means to create a more accurate integral estimate by combining two less accurate estimates.
- Understanding how Gauss quadrature provides superior integral estimates by picking optimal abscissas at which to evaluate the function.
- Knowing how to use MATLAB's built-in functions `quad` and `quadl` to integrate functions.

17.1 INTRODUCTION

In Chap. 16, we noted that functions to be integrated numerically will typically be of two forms: a table of values or a function. The form of the data has an important influence on the approaches that can be used to evaluate the integral. For tabulated information, you are limited by the number of points that are given. In contrast, if the function is available, you can generate as many values of $f(x)$ as are required to attain acceptable accuracy.

At face value, the composite Simpson's $1/3$ rule might seem to be a reasonable tool for such problems. Although it is certainly adequate for many problems, there are more efficient methods that are available. This chapter is devoted to three such techniques. Both capitalize on the ability to generate function values to develop efficient schemes for numerical integration.

The first technique is based on *Richardson's extrapolation*, which is a method for combining two numerical integral estimates to obtain a third, more accurate value. The computational algorithm for implementing Richardson's extrapolation in a highly efficient manner is called *Romberg integration*. This technique can be used to generate an integral estimate within a prespecified error tolerance.

The second method is called *Gauss quadrature*. Recall that, in Chap. 16, values of $f(x)$ for the Newton-Cotes formulas were determined at specified values of x . For example, if we used the trapezoidal rule to determine an integral, we were constrained to take the weighted average of $f(x)$ at the ends of the interval. Gauss-quadrature formulas employ x values that are positioned between the integration limits in such a manner that a much more accurate integral estimate results.

The third approach is called *adaptive quadrature*. This technique applies composite Simpson's 1/3 rule to subintervals of the integration range in a way that allows error estimates to be computed. These error estimates are then used to determine whether more refined estimates are required for a subinterval. In this way, more refined segmentation is only used where it is necessary. Two built-in MATLAB functions that use adaptive quadrature are illustrated.

17.2 ROMBERG INTEGRATION

Romberg integration is one technique that is designed to attain efficient numerical integrals of functions. It is quite similar to the techniques discussed in Chap. 16 in the sense that it is based on successive application of the trapezoidal rule. However, through mathematical manipulations, superior results are attained for less effort.

17.2.1 Richardson's Extrapolation

Techniques are available to improve the results of numerical integration on the basis of the integral estimates themselves. Generally called *Richardson's extrapolation*, these methods use two estimates of an integral to compute a third, more accurate approximation.

The estimate and the error associated with the composite trapezoidal rule can be represented generally as

$$I \approx I(h) + E(h)$$

where I = the exact value of the integral, $I(h)$ = the approximation from an n -segment application of the trapezoidal rule with step size $h = (b - a)/n$, and $E(h)$ = the truncation error. If we make two separate estimates using step sizes of h_1 and h_2 and have exact values for the error:

$$I(h_1) \approx I + E(h_1) \quad I(h_2) \approx I + E(h_2) \quad (17.1)$$

Now recall that the error of the composite trapezoidal rule can be represented approximately by Eq. (16.21) [with $n = (b - a)/h$]:

$$E \approx -\frac{b-a}{12} h^2 f'' \quad (17.2)$$

If it is assumed that f'' is constant regardless of step size, Eq. (17.2) can be used to determine that the ratio of the two errors will be

$$\frac{E(h_1)}{E(h_2)} \approx \frac{h_1^2}{h_2^2} \quad (17.3)$$

This calculation has the important effect of removing the term f'' from the computation. In so doing, we have made it possible to utilize the information embodied by Eq. (17.2)

17.2 ROMBERG INTEGRATION

307

without prior knowledge of the function's second derivative. To do this, we rearrange Eq. (17.3) to give

$$E(h_1) = E(h_2) \left(\frac{h_1}{h_2} \right)^2$$

which can be substituted into Eq. (17.1):

$$I(h_1) = E(h_2) \left(\frac{h_1}{h_2} \right)^2 = I(h_2) + E(h_2)$$

which can be solved for

$$E(h_2) = \frac{I(h_1) - I(h_2)}{1 - (h_1/h_2)^2}$$

Thus, we have developed an estimate of the truncation error in terms of the integral estimates and their step sizes. This estimate can then be substituted into

$$I = I(h_2) + E(h_2)$$

to yield an improved estimate of the integral:

$$I = I(h_2) + \frac{1}{(h_1/h_2)^2 - 1} [I(h_2) - I(h_1)] \quad (17.4)$$

It can be shown (Ralston and Rabinowitz, 1978) that the error of this estimate is $O(h^4)$. Thus, we have combined two trapezoidal rule estimates of $O(h^2)$ to yield a new estimate of $O(h^4)$. For the special case where the interval is halved ($h_2 = h_1/2$), this equation becomes

$$I = \frac{4}{3}I(h_2) - \frac{1}{3}I(h_1) \quad (17.5)$$

EXAMPLE 17.1 Richardson Extrapolation

Problem Statement. Use Richardson extrapolation to evaluate the integral of $f(x) = 0.2 + 25x + 200x^2 + 675x^3 + 900x^4 + 400x^5$ from $a = 0$ to $b = 0.8$.

Solution. Single and composite applications of the trapezoidal rule can be used to evaluate the integral:

Segments		Integral	e
1	0.8	0.1728	89.5%
2	0.4	1.0688	34.9%
4	0.2	1.4848	9.5%

Richardson extrapolation can be used to combine these results to obtain improved estimates of the integral. For example, the estimates for one and two segments can be combined

to yield

$$I \approx \frac{4}{3}(1.0688) - \frac{1}{3}(0.1728) = 1.367467$$

The error of the improved integral is $E_I \approx 1.640533 - 1.367467 = 0.273067$ ($\epsilon_I \approx 16.6\%$), which is superior to the estimates upon which it was based.

In the same manner, the estimates for two and four segments can be combined to give

$$I \approx \frac{4}{3}(1.4848) - \frac{1}{3}(1.0688) = 1.623467$$

which represents an error of $E_I \approx 1.640533 - 1.623467 = 0.017067$ ($\epsilon_I \approx 1.0\%$).

Equation (17.4) provides a way to combine two applications of the trapezoidal rule with error $O(h^2)$ to compute a third estimate with error $O(h^4)$. This approach is a subset of a more general method for combining integrals to obtain improved estimates. For instance, in Example 17.1, we computed two improved integrals of $O(h^4)$ on the basis of three trapezoidal rule estimates. These two improved integrals can, in turn, be combined to yield an even better value with $O(h^6)$. For the special case where the original trapezoidal estimates are based on successive halving of the step size, the equation used for $O(h^6)$ accuracy is

$$I \approx \frac{16}{15}I_m - \frac{1}{15}I_l \quad (17.6)$$

where I_m and I_l are the more and less accurate estimates, respectively. Similarly, two $O(h^6)$ results can be combined to compute an integral that is $O(h^8)$ using

$$I \approx \frac{64}{63}I_m - \frac{1}{63}I_l \quad (17.7)$$

EXAMPLE 17.2 Higher-Order Corrections

Problem Statement. In Example 17.1, we used Richardson's extrapolation to compute two integral estimates of $O(h^4)$. Utilize Eq. (17.6) to combine these estimates to compute an integral with $O(h^6)$.

Solution. The two integral estimates of $O(h^4)$ obtained in Example 17.1 were 1.367467 and 1.623467. These values can be substituted into Eq. (17.6) to yield

$$I \approx \frac{16}{15}(1.623467) - \frac{1}{15}(1.367467) = 1.640533$$

which is the exact value of the integral.

17.2.2 The Romberg Integration Algorithm

Notice that the coefficients in each of the extrapolation equations [Eqs. (17.5), (17.6), and (17.7)] add up to 1. Thus, they represent weighting factors that, as accuracy increases,

17.2 ROMBERG INTEGRATION

309

place relatively greater weight on the superior integral estimate. These formulations can be expressed in a general form that is well suited for computer implementation:

$$I_{j,k} = \frac{4^{k-1}I_{j-1,k-1} - I_{j,k-1}}{4^{k-1} - 1} \quad (17.8)$$

where $I_{j-1,k-1}$ and $I_{j,k-1}$ are the more and less accurate integrals, respectively, and $I_{j,k}$ is the improved integral. The index k signifies the level of the integration, where $k = 1$ corresponds to the original trapezoidal rule estimates, $k = 2$ corresponds to the $O(h^4)$ estimates, $k = 3$ to the $O(h^6)$, and so forth. The index j is used to distinguish between the more ($j = 1$) and the less (j) accurate estimates. For example, for $k = 2$ and $j = 1$, Eq. (17.8) becomes

$$I_{1,2} = \frac{4I_{2,1} - I_{1,1}}{3}$$

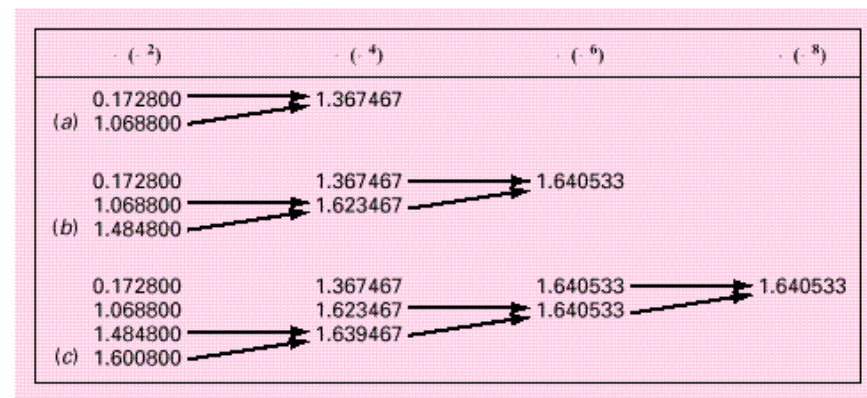
which is equivalent to Eq. (17.5).

The general form represented by Eq. (17.8) is attributed to Romberg, and its systematic application to evaluate integrals is known as *Romberg integration*. Figure 17.1 is a graphical depiction of the sequence of integral estimates generated using this approach. Each matrix corresponds to a single iteration. The first column contains the trapezoidal rule evaluations that are designated $I_{j,1}$, where $j = 1$ is for a single-segment application (step size is $b - a$), $j = 2$ is for a two-segment application [step size is $(b - a)/2$], $j = 3$ is for a four-segment application [step size is $(b - a)/4$], and so forth. The other columns of the matrix are generated by systematically applying Eq. (17.8) to obtain successively better estimates of the integral.

For example, the first iteration (Fig. 17.1a) involves computing the one- and two-segment trapezoidal rule estimates ($I_{1,1}$ and $I_{2,1}$). Equation (17.8) is then used to compute the element $I_{1,2} = 1.367467$, which has an error of $O(h^4)$.

FIGURE 17.1

Graphical depiction of the sequence of integral estimates generated using Romberg integration. [·] First iteration, {·} Second iteration, [·] Third iteration.



Now, we must check to determine whether this result is adequate for our needs. As in other approximate methods in this book, a termination, or stopping, criterion is required to assess the accuracy of the results. One method that can be employed for the present purposes is

$$\varepsilon_a = \left| \frac{I_{1,k} - I_{2,k-1}}{I_{1,k}} \right| \cdot 100\% \quad (17.9)$$

where ε_a = an estimate of the percent relative error. Thus, as was done previously in other iterative processes, we compare the new estimate with a previous value. For Eq. (17.9), the previous value is the most accurate estimate from the previous level of integration (i.e., the $k - 1$ level of integration with $j = 2$). When the change between the old and new values as represented by ε_a is below a prespecified error criterion ε_s , the computation is terminated. For Fig. 17.1a, this evaluation indicates the following percent change over the course of the first iteration:

$$\varepsilon_a = \left| \frac{1.367467 - 1.068800}{1.367467} \right| \cdot 100\% = 21.8\%$$

The object of the second iteration (Fig. 17.1b) is to obtain the $O(h^6)$ estimate— $I_{1,3}$. To do this, a four-segment trapezoidal rule estimate, $I_{3,1} = 1.4848$, is determined. Then it is combined with $I_{2,1}$ using Eq. (17.8) to generate $I_{2,2} = 1.623467$. The result is, in turn, combined with $I_{1,2}$ to yield $I_{1,3} = 1.640533$. Equation (17.9) can be applied to determine that this result represents a change of 1.0% when compared with the previous result $I_{2,2}$.

The third iteration (Fig. 17.1c) continues the process in the same fashion. In this case, an eight-segment trapezoidal estimate is added to the first column, and then Eq. (17.8) is applied to compute successively more accurate integrals along the lower diagonal. After only three iterations, because we are evaluating a fifth-order polynomial, the result ($I_{1,4} = 1.640533$) is exact.

Romberg integration is more efficient than the trapezoidal rule and Simpson's rules. For example, for determination of the integral as shown in Fig. 17.1, Simpson's 1/3 rule would require about a 48-segment application in double precision to yield an estimate of the integral to seven significant digits: 1.640533. In contrast, Romberg integration produces the same result based on combining one-, two-, four-, and eight-segment trapezoidal rules—that is, with only 15 function evaluations!

Figure 17.2 presents an M-file for Romberg integration. By using loops, this algorithm implements the method in an efficient manner. Note that the function uses another function `trap` to implement the composite trapezoidal rule evaluations (recall Fig. 16.10).

17.3 GAUSS QUADRATURE

In Chap. 16, we employed the Newton-Cotes equations. A characteristic of these formulas (with the exception of the special case of unequally spaced data) was that the integral estimate was based on evenly spaced function values. Consequently, the location of the base points used in these equations was predetermined or fixed.

17.3 GAUSS QUADRATURE

311

```
function intg = romberg(func,a,b,es,maxit)
% romberg(func,a,b,es):
%   Romberg integration.
% input:
%   func = name of function to be integrated
%   a, b = integration limits
%   es = (optional) stop criterion (%); default = 0.00001
%   maxit = (optional) max allow iterations; default = 30
% output:
%   intg = integral estimate

% if necessary, assign default values
if nargin<5, maxit=30; end    %if maxit blank set to 30
if nargin<4, es=0.00001; end %if es blank set to 0.00001

n = 1;
I(1,1) = trap(func,a,b,n);
iter = 0;
while iter<maxit
    iter = iter+1;
    n = 2^iter;
    I(iter+1,1) = trap(func,a,b,n);
    for k = 2:iter+1
        j = 2+iter-k;
        I(j,k) = (4^(k-1)*I(j+1,k-1)-I(j,k-1))/(4^(k-1)-1);
    end
    ea = abs((I(1,iter+1)-I(2,iter))/I(1,iter + 1))*100;
    if ea<=es, break; end
end
intg = I(1, iter + 1);
```

FIGURE 17.2

M-file to implement Romberg integration.

For example, as depicted in Fig. 17.3a, the trapezoidal rule is based on taking the area under the straight line connecting the function values at the ends of the integration interval. The formula that is used to compute this area is

$$I \approx (b-a) \frac{f(a) + f(b)}{2} \quad (17.10)$$

where a and b are the limits of integration and $b-a$ is the width of the integration interval. Because the trapezoidal rule must pass through the end points, there are cases such as Fig. 17.3a where the formula results in a large error.

Now, suppose that the constraint of fixed base points was removed and we were free to evaluate the area under a straight line joining any two points on the curve. By positioning these points wisely, we could define a straight line that would balance the positive and

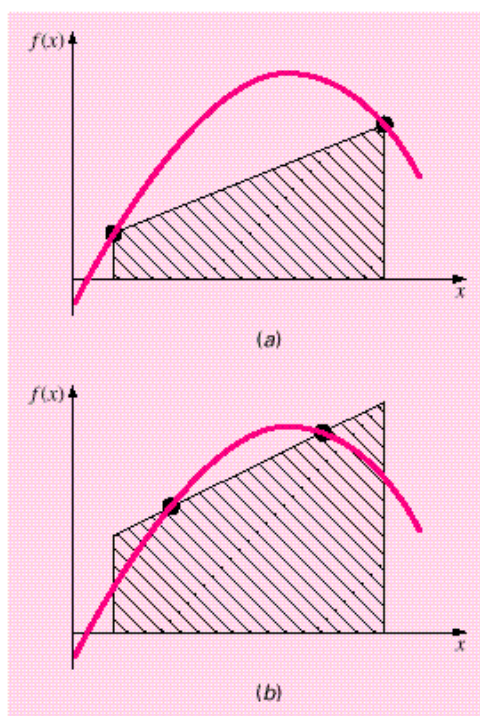


FIGURE 17.3

(a) Graphical depiction of the trapezoidal rule as the area under the straight line joining fixed end points. (b) An improved integral estimate obtained by taking the area under the straight line passing through two intermediate points. By positioning these points wisely, the positive and negative errors are better balanced, and an improved integral estimate results.

negative errors. Hence, as in Fig. 17.3b, we would arrive at an improved estimate of the integral.

Gauss quadrature is the name for a class of techniques to implement such a strategy. The particular Gauss quadrature formulas described in this section are called *Gauss-Legendre* formulas. Before describing the approach, we will show how numerical integration formulas such as the trapezoidal rule can be derived using the method of undetermined coefficients. This method will then be employed to develop the Gauss-Legendre formulas.

17.3.1 Method of Undetermined Coefficients

In Chap. 16, we derived the trapezoidal rule by integrating a linear interpolating polynomial and by geometrical reasoning. The method of undetermined coefficients offers a third approach that also has utility in deriving other integration techniques such as Gauss quadrature.

To illustrate the approach, Eq. (17.10) is expressed as

$$I \approx c_0 f(a) + c_1 f(b) \quad (17.11)$$

17.3 GAUSS QUADRATURE

313

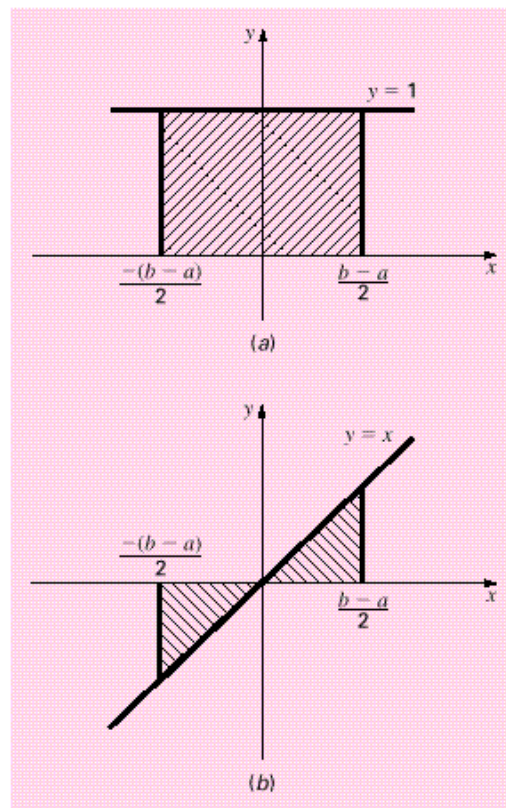


FIGURE 17.4

Two integrals that should be evaluated exactly by the trapezoidal rule: [·] a constant and [·] a straight line.

where the c 's = constants. Now realize that the trapezoidal rule should yield exact results when the function being integrated is a constant or a straight line. Two simple equations that represent these cases are $y = 1$ and $y = x$ (Fig. 17.4). Thus, the following equalities should hold:

$$c_0 = c_1 = \int_{-(b-a)/2}^{(b-a)/2} 1 dx$$

and

$$= c_0 \frac{b-a}{2} = c_1 \frac{b-a}{2} = \int_{-(b-a)/2}^{(b-a)/2} x dx$$

or, evaluating the integrals,

$$c_0 = c_1 = b - a$$

and

$$-c_0 \frac{b-a}{2} + c_1 \frac{b-a}{2} = 0$$

These are two equations with two unknowns that can be solved for

$$c_0 = c_1 = \frac{b-a}{2}$$

which, when substituted back into Eq. (17.11), gives

$$I = \frac{b-a}{2} f(a) + \frac{b-a}{2} f(b)$$

which is equivalent to the trapezoidal rule.

17.3.2 Derivation of the Two-Point Gauss-Legendre Formula

Just as was the case for the previous derivation of the trapezoidal rule, the object of Gauss quadrature is to determine the coefficients of an equation of the form

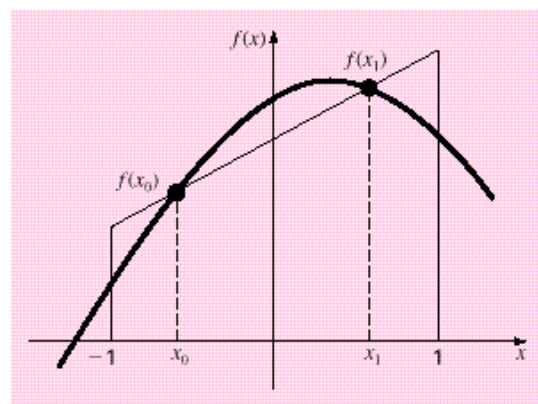
$$I = c_0 f(x_0) + c_1 f(x_1) \quad (17.12)$$

where the c 's are the unknown coefficients. However, in contrast to the trapezoidal rule that used fixed end points a and b , the function arguments x_0 and x_1 are not fixed at the end points, but are unknowns (Fig. 17.5). Thus, we now have a total of four unknowns that must be evaluated, and consequently, we require four conditions to determine them exactly.

Just as for the trapezoidal rule, we can obtain two of these conditions by assuming that Eq. (17.12) fits the integral of a constant and a linear function exactly. Then, to arrive at the other two conditions, we merely extend this reasoning by assuming that it also fits the integral of a parabolic ($y = x^2$) and a cubic ($y = x^3$) function. By doing this, we determine all four unknowns and in the bargain derive a linear two-point integration formula that is

FIGURE 17.5

Graphical depiction of the unknown variables x_0 and x_1 for integration by Gauss quadrature.



17.3 GAUSS QUADRATURE

315

exact for cubics. The four equations to be solved are

$$c_0 + c_1 = \int_{-1}^1 1 dx = 2 \quad (17.13)$$

$$c_0 x_0 + c_1 x_1 = \int_{-1}^1 x dx = 0 \quad (17.14)$$

$$c_0 x_0^2 + c_1 x_1^2 = \int_{-1}^1 x^2 dx = \frac{2}{3} \quad (17.15)$$

$$c_0 x_0^3 + c_1 x_1^3 = \int_{-1}^1 x^3 dx = 0 \quad (17.16)$$

Equations (17.13) through (17.16) can be solved simultaneously for the four unknowns. First, solve Eq. (17.14) for c_1 and substitute the result into Eq. (17.16), which can be solved for

$$x_0^2 = x_1^2$$

Since x_0 and x_1 cannot be equal, this means that $x_0 = -x_1$. Substituting this result into Eq. (17.14) yields $c_0 = c_1$. Consequently from Eq. (17.13) it follows that

$$c_0 = c_1 = 1$$

Substituting these results into Eq. (17.15) gives

$$x_0 = -\frac{1}{\sqrt{3}} = -0.5773503 \dots$$

$$x_1 = \frac{1}{\sqrt{3}} = 0.5773503 \dots$$

Therefore, the two-point Gauss-Legendre formula is

$$I = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \quad (17.17)$$

Thus, we arrive at the interesting result that the simple addition of the function values at $x = -1/\sqrt{3}$ and $1/\sqrt{3}$ yields an integral estimate that is third-order accurate.

Notice that the integration limits in Eqs. (17.13) through (17.16) are from -1 to 1 . This was done to simplify the mathematics and to make the formulation as general as possible. A simple change of variable can be used to translate other limits of integration into this form. This is accomplished by assuming that a new variable x_d is related to the original variable x in a linear fashion, as in

$$x = a_0 x_d + a_1 \quad (17.18)$$

If the lower limit, $x = a$, corresponds to $x_d = -1$, these values can be substituted into Eq. (17.18) to yield

$$a = a_0(-1) + a_1 \quad (17.19)$$

Similarly, the upper limit, $x = b$, corresponds to $x_d = 1$, to give

$$b = a_0(1) + a_1 \quad (17.20)$$

Equations (17.19) and (17.20) can be solved simultaneously for

$$a_0 = \frac{b - a}{2} \quad \text{and} \quad a_1 = \frac{b + a}{2} \quad (17.21)$$

which can be substituted into Eq. (17.18) to yield

$$x = \frac{(b - a) \cdot (b + a)x_d}{2} \quad (17.22)$$

This equation can be differentiated to give

$$dx = \frac{b - a}{2} dx_d \quad (17.23)$$

Equations (17.22) and (17.23) can be substituted for x and dx , respectively, in the equation to be integrated. These substitutions effectively transform the integration interval without changing the value of the integral. The following example illustrates how this is done in practice.

EXAMPLE 17.3 Two-Point Gauss-Legendre Formula

Problem Statement. Use Eq. (17.17) to evaluate the integral of

$$f(x) = 0.2 + 25x + 200x^2 + 675x^3 + 900x^4 + 400x^5$$

between the limits $x = 0$ to 0.8 . The exact value of the integral is 1.640533.

Solution. Before integrating the function, we must perform a change of variable so that the limits are from -1 to 1 . To do this, we substitute $a = 0$ and $b = 0.8$ into Eqs. (17.22) and (17.23) to yield

$$x = 0.4 + 0.4x_d \quad \text{and} \quad dx = 0.4dx_d$$

Both of these can be substituted into the original equation to yield

$$\begin{aligned} & \int_0^{0.8} (0.2 + 25x + 200x^2 + 675x^3 + 900x^4 + 400x^5) dx \\ &= \int_{-1}^1 [0.2 + 25(0.4 + 0.4x_d) + 200(0.4 + 0.4x_d)^2 + 675(0.4 + 0.4x_d)^3 \\ & \quad + 900(0.4 + 0.4x_d)^4 + 400(0.4 + 0.4x_d)^5] 0.4dx_d \end{aligned}$$

Therefore, the right-hand side is in the form that is suitable for evaluation using Gauss quadrature. The transformed function can be evaluated at $x_d = -1/\sqrt{3}$ as 0.516741 and at $x_d = 1/\sqrt{3}$ as 1.305837. Therefore, the integral according to Eq. (17.17) is $0.516741 + 1.305837 = 1.822578$, which represents a percent relative error of $\sim 11.1\%$. This result is comparable in magnitude to a four-segment application of the trapezoidal rule or a single application of Simpson's $1/3$ and $3/8$ rules. This latter result is to be expected because Simpson's rules are also third-order accurate. However, because of the clever choice of base points, Gauss quadrature attains this accuracy on the basis of only two function evaluations.

TABLE 17.1 Weighting factors and function arguments used in Gauss-Legendre formulas.

Points	Weighting Factors	Function Arguments	Truncation Error
2	$c_0 = 1.0000000$ $c_1 = 1.0000000$	$x_0 = -0.577350269$ $x_1 = 0.577350269$	$f^{(4)}(\xi)$
3	$c_0 = 0.5555556$ $c_1 = 0.8888889$ $c_2 = 0.5555556$	$x_0 = -0.774596669$ $x_1 = 0.0$ $x_2 = 0.774596669$	$f^{(6)}(\xi)$
4	$c_0 = 0.3478548$ $c_1 = 0.6521452$ $c_2 = 0.6521452$ $c_3 = 0.3478548$	$x_0 = -0.861136312$ $x_1 = -0.339981044$ $x_2 = 0.339981044$ $x_3 = 0.861136312$	$f^{(8)}(\xi)$
5	$c_0 = 0.2369269$ $c_1 = 0.4786287$ $c_2 = 0.5688889$ $c_3 = 0.4786287$ $c_4 = 0.2369269$	$x_0 = -0.906179846$ $x_1 = -0.538469310$ $x_2 = 0.0$ $x_3 = 0.538469310$ $x_4 = 0.906179846$	$f^{(10)}(\xi)$
6	$c_0 = 0.1713245$ $c_1 = 0.3607616$ $c_2 = 0.4679139$ $c_3 = 0.4679139$ $c_4 = 0.3607616$ $c_5 = 0.1713245$	$x_0 = -0.932469514$ $x_1 = -0.661209386$ $x_2 = -0.238619186$ $x_3 = 0.238619186$ $x_4 = 0.661209386$ $x_5 = 0.932469514$	$f^{(12)}(\xi)$

17.3.3 Higher-Point Formulas

Beyond the two-point formula described in the previous section, higher-point versions can be developed in the general form

$$I \approx c_0 f(x_0) + c_1 f(x_1) + \cdots + c_{n-1} f(x_{n-1}) \quad (17.24)$$

where n = the number of points. Values for c 's and x 's for up to and including the six-point formula are summarized in Table 17.1.

EXAMPLE 17.4 Three-Point Gauss-Legendre Formula

Problem Statement. Use the three-point formula from Table 17.1 to estimate the integral for the same function as in Example 17.3.

Solution. According to Table 17.1, the three-point formula is

$$I \approx 0.5555556 f(-0.7745967) + 0.8888889 f(0) + 0.5555556 f(0.7745967)$$

which is equal to

$$I \approx 0.2813013 + 0.8732444 + 0.4859876 = 1.640533$$

which is exact.

Because Gauss quadrature requires function evaluations at nonuniformly spaced points within the integration interval, it is not appropriate for cases where the function is unknown. Thus, it is not suited for engineering problems that deal with tabulated data. However, where the function is known, its efficiency can be a decided advantage. This is particularly true when numerous integral evaluations must be performed.

17.4 ADAPTIVE QUADRATURE

Although the composite Simpson's 1/3 rule can certainly be used to estimate the integral of given functions, it has the disadvantage that it uses equally spaced points. This constraint does not take into account that some functions have regions of relatively abrupt changes where more refined spacing might be required. Hence, to achieve a desired accuracy, the fine spacing must be applied everywhere even though it is only needed for the regions of sharp change. Adaptive quadrature methods remedy this situation by automatically adjusting the step size so that small steps are taken in regions of sharp variations and larger steps are taken where the function changes gradually.

Most of these techniques are based on applying the composite Simpson's 1/3 rule to subintervals in a manner similar to how the trapezoidal rule was used in Romberg integration. That is, the 1/3 rule is applied at two levels of refinement and the difference between these two levels is used to estimate the truncation error. If the truncation error is acceptable, no further refinement is required and the integral estimate for the subinterval is deemed acceptable. If the error estimate is too large, the step size is refined and the process repeated until the error falls to acceptable levels.

MATLAB includes two built-in functions to implement adaptive quadrature: `quad` and `quadl`. The following section describes how they can be applied.

17.4.1 MATLAB Functions: `quad` and `quadl`

MATLAB has two functions, both based on algorithms developed by Gander and Gautschi (2000), for implementing adaptive quadrature:

- `quad`. This function uses adaptive Simpson quadrature. It may be more efficient for low accuracies or nonsmooth functions.
- `quadl`. This function uses what is called *Lobatto quadrature*. It may be more efficient for high accuracies and smooth functions.

The following function syntax for the `quad` function is the same for the `quadl` function:

```
q = quad(fun, a, b, tol, trace, p1, p2, ...)
```

where `fun` is the function to be integrated, `a` and `b` the integration bounds, `tol` the desired absolute error tolerance (default $\cdot 10^{-6}$), `trace` is a variable that when set to a nonzero value causes additional computational detail to be displayed, and `p1`, `p2`, ... are parameters that you want to pass to `fun`. It should be noted that array operators `.*`, `./` and `.^` should be used in the definition of `fun`. In addition, pass empty matrices for `tol` or `trace` to use the default values.

PROBLEMS

319

EXAMPLE 17.5 Adaptive Quadrature

Problem Statement. Use `quad` to integrate the following function:

$$f(x) = \frac{1}{(x - q)^2 + 0.01} + \frac{1}{(x - r)^2 + 0.04} \cdot s$$

between the limits $x = 0$ to 1. Note that for $q = 0.3$, $r = 0.9$, and $s = 6$, this is the built-in `humps` function that MATLAB uses to demonstrate some of its numerical capabilities. The `humps` function exhibits both flat and steep regions over a relatively short x range. Hence, it is useful for demonstrating and testing routines like `quad` and `quadl`. Note that the `humps` function can be integrated analytically between the given limits to yield an exact integral of 29.85832539549867.

Solution. First, let's evaluate the integral in the simplest way possible, using the built-in version of `humps` along with the default tolerance:

```
>> format long
>> quad(@humps,0,1)

ans =
    29.85832612842764
```

Thus, the solution is correct to seven significant digits.

Next, we can solve the same problem, but using a looser tolerance and passing q , r , and s as parameters. First, we can develop an M-file for the function:

```
function y = myhumps(x,q,r,s)
y = 1./((x-q).^2 + 0.01) + 1./((x-r).^2+0.04) * s;
```

Then, we can integrate it with an error tolerance of 10^{-4} as in

```
>> quad(@myhumps,0,1,1e-4,[],0.3,0.9,6)

ans =
    29.85812133214492
```

Notice that because we used a larger tolerance, the result is now only accurate to five significant digits. However, although it would not be apparent from a single application, fewer function evaluations were made and, hence, the computation executes faster.

PROBLEMS

17.1 Use Romberg integration to evaluate

$$I = \int_1^2 \left(2x + \frac{3}{x} \right)^2 dx$$

to an accuracy of $\varepsilon_s = 0.5\%$. Your results should be presented in the format of Fig. 17.1. Use the analytical solution of the integral to determine the percent relative error of the result obtained with Romberg integration. Check that ε_t is less than ε_s .

17.2 Evaluate the following integral (a) analytically, (b) Romberg integration ($\varepsilon_s = 0.5\%$), (c) the three-point Gauss quadrature formula, and (d) MATLAB `quad` function:

$$I = \int_0^8 \left(0.0547x^4 + 0.8646x^3 + 4.1562x^2 + 6.2917x + 2 \right) dx$$

17.3 Evaluate the following integral with (a) Romberg integration ($\epsilon_s = 0.5\%$), (b) the two-point Gauss quadrature formula, and (c) MATLAB `quad` and `quadl` functions:

$$I = \int_0^3 x e^x dx$$

17.4 There is no closed form solution for the error function

$$\operatorname{erf}(a) = \frac{2}{\sqrt{\pi}} \int_0^a e^{-x^2} dx$$

Use the (a) two-point and (b) three-point Gauss-Legendre formulas to estimate $\operatorname{erf}(1.5)$. Determine the percent relative error for each case based on the true value, which can be determined with MATLAB's built-in function `erf`.

17.5 The force on a sailboat mast can be represented by the following function:

$$F = \int_0^H 200 \left(\frac{z}{7} \right) e^{-2.5z/H} dz$$

where z = the elevation above the deck and H = the height of the mast. Compute F for the case where $H = 30$ using (a) Romberg integration to a tolerance of $\epsilon_s = 0.5\%$, (b) the two-point Gauss-Legendre formula, and (c) the MATLAB `quad` function.

17.6 The root-mean-square current can be computed as

$$I_{RMS} = \sqrt{\frac{1}{T} \int_0^T i^2(t) dt}$$

For $T = 1$, suppose that $i(t)$ is defined as

$$i(t) = 10e^{-t/T} \sin\left(2\pi \frac{t}{T}\right) \quad \text{for } 0 \leq t \leq T/2$$

$$i(t) = 0 \quad \text{for } T/2 \leq t \leq T$$

Evaluate the I_{RMS} using (a) Romberg integration to a tolerance of 0.1% , (b) the two- and three-point Gauss-Legendre formulas, and (c) the MATLAB `quad` function.

17.7 The velocity profile of a fluid in a circular pipe can be represented as

$$v = 10 \left(1 - \frac{r}{r_0} \right)^{1/n}$$

where v = velocity, r = radial distance measured out from the pipes centerline, r_0 = the pipe's radius, and n = a parameter. Determine the flow in the pipe if $r_0 = 0.75$ and $n = 7$ using (a) Romberg integration to a tolerance of 0.1% , (b) the two-point Gauss-Legendre formula, and (c) the MATLAB `quad` function. Note that flow is equal to velocity times area.

17.8 The amount of mass transported via a pipe over a period of time can be computed as

$$M = \int_{t_1}^{t_2} Q(t)c(t) dt$$

where M = mass (mg), t_1 = the initial time (min), t_2 = the final time (min), $Q(t)$ = flow rate (m^3/min), and $c(t)$ = concentration (mg/m^3). The following functional representations define the temporal variations in flow and concentration:

$$Q(t) = 9 - 4 \cos^2(0.4t)$$

$$c(t) = 5e^{-0.5t} - 2e^{0.15t}$$

Determine the mass transported between $t_1 = 2$ and $t_2 = 8$ min with (a) Romberg integration to a tolerance of 0.1% and (b) the MATLAB `quad` function.

17.9 Evaluate the double integral

$$\int_{-2}^2 \int_0^4 (x^2 + 3y^2 - xy^3) dx dy$$

(a) analytically and (b) using the MATLAB `dblquad` function. Use `help` to understand how to implement the function.

ODEs: Initial-Value Problems

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to solving initial-value problems for ODEs (ordinary differential equations). Specific objectives and topics covered are

- Understanding the meaning of local and global truncation errors and their relationship to step size for one-step methods for solving ODEs.
- Knowing how to implement the following Runge-Kutta (RK) methods for a single ODE:
 - Euler
 - Heun
 - Midpoint
 - Fourth-order RK
- Knowing how to iterate the corrector of Heun's method.
- Knowing how to implement the following Runge-Kutta methods for systems of ODEs:
 - Euler
 - Fourth-order RK

YOU'VE GOT A PROBLEM

We started this book with the problem of simulating the velocity of a free-falling bungee jumper. This problem amounted to formulating and solving an ordinary differential equation, the topic of this chapter. Now let's return to this problem and make it more interesting by computing what happens when the jumper reaches the end of the bungee cord.

To do this, we should recognize that the jumper will experience different forces depending on whether the cord is slack or stretched. If it is slack, the situation is that of free fall where the only forces are gravity and drag. However, because the jumper can now move up as well as down, the sign of the drag force must be modified so that it always tends to retard velocity,

$$\frac{dv}{dt} = g - \text{sign}(v) \frac{c_d}{m} v^2 \quad (18.1a)$$

where v is velocity (m/s), t is time (s), g is the acceleration due to gravity (9.81 m/s^2), c_d is the drag coefficient (kg/m), and m is mass (kg). The *signum function*,¹ sign , returns a -1 or a 1 depending on whether its argument is negative or positive, respectively. Thus, when the jumper is falling downward (positive velocity, $\text{sign} = 1$), the drag force will be negative and hence will act to reduce velocity. In contrast, when the jumper is moving upward (negative velocity, $\text{sign} = -1$), the drag force will be positive so that it again reduces the velocity.

Once the cord begins to stretch, it obviously exerts an upward force on the jumper. As done previously in Chap. 7, Hooke's law can be used as a first approximation of this force. In addition, a dampening force should also be included to account for frictional effects as the cord stretches and contracts. These factors can be incorporated along with gravity and drag into a second force balance that applies when the cord is stretched. The result is the following differential equation:

$$\frac{dv}{dt} = g - \text{sign}(v) \frac{c_d}{m} v^2 - \frac{k}{m} (x - L) - \frac{\gamma}{m} v \quad (18.1b)$$

where k is the cord's spring constant (N/m), x is vertical distance measured downward from the bungee jump platform (m), L is the length of the unstretched cord (m), and γ is a dampening coefficient (N · s/m).

Because Eq. (18.1b) only holds when the cord is stretched ($x > L$), the spring force will always be negative. That is, it will always act to pull the jumper back up. The dampening force increases in magnitude as the jumper's velocity increases and always acts to slow the jumper down.

If we want to simulate the jumper's velocity, we would initially solve Eq. (18.1a) until the cord was fully extended. Then, we could switch to Eq. (18.1b) for periods that the cord is stretched. Although this is fairly straightforward, it means that knowledge of the jumper's position is required. This can be done by formulating another differential equation for distance:

$$\frac{dx}{dt} = v \quad (18.2)$$

Thus, solving for the bungee jumper's velocity amounts to solving two ordinary differential equations where one of the equations takes different forms depending on the value

¹Some computer languages represent the signum function as $\text{sgn}(x)$. As represented here, MATLAB uses the nomenclature $\text{sign}(x)$.

of one of the dependent variables. Chapters 18 and 19 explore methods for solving this and similar problems involving ODEs.

18.1 OVERVIEW

This chapter is devoted to solving ordinary differential equations of the form

$$\frac{dy}{dt} = f(t, y) \quad (18.3)$$

In Chap. 1, we developed a numerical method to solve such an equation for the velocity of the free-falling bungee jumper. Recall that the method was of the general form

$$\text{New value} = \text{old value} + \text{slope} \cdot \text{step size}$$

or, in mathematical terms,

$$y_{i+1} = y_i + \phi h \quad (18.4)$$

where the slope ϕ is called an *increment function*. According to this equation, the slope estimate of ϕ is used to extrapolate from an old value y_i to a new value y_{i+1} over a distance h . This formula can be applied step by step to trace out the trajectory of the solution into the future. Such approaches are called *one-step methods* because the value of the increment function is based on information at a single point i . They are also referred to as *Runge-Kutta methods* after the two applied mathematicians who first discussed them in the early 1900s. Another class of methods called *multistep methods* use information from several previous points as the basis for extrapolating to a new value. We will describe multistep methods briefly in Chap. 19.

All one-step methods can be expressed in the general form of Eq. (18.4), with the only difference being the manner in which the slope is estimated. The simplest approach is to use the differential equation to estimate the slope in the form of the first derivative at t_i . In other words, the slope at the beginning of the interval is taken as an approximation of the average slope over the whole interval. This approach, called Euler's method, is discussed next. This is followed by other one-step methods that employ alternative slope estimates that result in more accurate predictions.

18.2 EULER'S METHOD

The first derivative provides a direct estimate of the slope at t_i (Fig. 18.1):

$$\phi = f(t_i, y_i)$$

where $f(t_i, y_i)$ is the differential equation evaluated at t_i and y_i . This estimate can be substituted into Eq. (18.1):

$$y_{i+1} = y_i + f(t_i, y_i)h \quad (18.5)$$

This formula is referred to as *Euler's method* (or the Euler-Cauchy or point-slope method). A new value of y is predicted using the slope (equal to the first derivative at the original value of t) to extrapolate linearly over the step size h (Fig. 18.1).

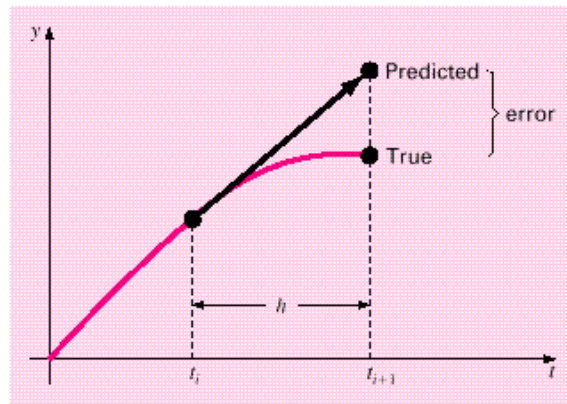


FIGURE 18.1
Euler's method.

EXAMPLE 18.1 Euler's Method

Problem Statement. Use Euler's method to integrate $y' = 4e^{0.8t} - 0.5y$ from $t = 0$ to 4 with a step size of 1. The initial condition at $t = 0$ is $y = 2$. Note that the exact solution can be determined analytically as

$$y = \frac{4}{1.3}(e^{0.8t} - e^{-0.5t}) + 2e^{-0.5t}$$

Solution. Equation (18.5) can be used to implement Euler's method:

$$y(1) = y(0) + f(0, 2)(1)$$

where $y(0) = 2$ and the slope estimate at $t = 0$ is

$$f(0, 2) = 4e^0 - 0.5(2) = 3$$

Therefore,

$$y(1) = 2 + 3(1) = 5$$

The true solution at $t = 1$ is

$$y = \frac{4}{1.3}(e^{0.8(1)} - e^{-0.5(1)}) + 2e^{-0.5(1)} = 6.19463$$

Thus, the percent relative error is

$$\varepsilon_t = \left| \frac{6.19463 - 5}{6.19463} \right| \cdot 100\% = 19.28\%$$

For the second step:

$$\begin{aligned} y(2) &= y(1) + f(1, 5)(1) \\ &= 5 + [4e^{0.8(1)} - 0.5(5)](1) = 11.40216 \end{aligned}$$

18.2 EULER'S METHOD

325

TABLE 18.1 Comparison of true and numerical values of the integral of $y' = 4e^{0.8t} - 0.5y$, with the initial condition that $y = 2$ at $t = 0$. The numerical values were computed using Euler's method with a step size of 1.

t	True	Euler	Error (%)
0	2.00000	2.00000	
1	6.19463	5.00000	19.28
2	14.84392	11.40216	23.19
3	33.67717	25.51321	24.24
4	75.33896	56.84931	24.54

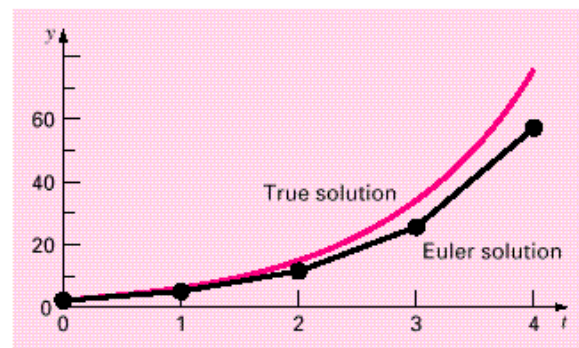


FIGURE 18.2

Comparison of the true solution with a numerical solution using Euler's method for the integral of $y' = 4e^{0.8t} - 0.5y$ from $t = 0$ to 4 with a step size of 1.0. The initial condition at $t = 0$ is $y = 2$.

The true solution at $t = 2.0$ is 14.84392 and, therefore, the true percent relative error is 23.19%. The computation is repeated, and the results compiled in Table 18.1 and Fig. 18.2. Note that although the computation captures the general trend of the true solution, the error is considerable. As discussed in the next section, this error can be reduced by using a smaller step size.

18.2.1 Error Analysis for Euler's Method

The numerical solution of ODEs involves two types of error (recall Chap. 4):

1. *Truncation*, or discretization, errors caused by the nature of the techniques employed to approximate values of y .
2. *Round-off* errors caused by the limited numbers of significant digits that can be retained by a computer.

The truncation errors are composed of two parts. The first is a *local truncation error* that results from an application of the method in question over a single step. The second is a *propagated truncation error* that results from the approximations produced during the

previous steps. The sum of the two is the total error. It is referred to as the *global truncation error*.

Insight into the magnitude and properties of the truncation error can be gained by deriving Euler's method directly from the Taylor series expansion. To do this, realize that the differential equation being integrated will be of the general form of Eq. (18.3), where $dy/dt = y'$, and t and y are the independent and the dependent variables, respectively. If the solution—that is, the function describing the behavior of y —has continuous derivatives, it can be represented by a Taylor series expansion about a starting value (t_i, y_i) , as in [recall Eq. (4.13)]:

$$y_{i+1} = y_i + y'_i h + \frac{y''_i}{2!} h^2 + \dots + \frac{y^{(n)}_i}{n!} h^n + R_n \quad (18.6)$$

where $h = t_{i+1} - t_i$ and $R_n =$ the remainder term, defined as

$$R_n = \frac{y^{(n+1)}(\xi)}{(n+1)!} h^{n+1} \quad (18.7)$$

where ξ lies somewhere in the interval from t_i to t_{i+1} . An alternative form can be developed by substituting Eq. (18.3) into Eqs. (18.6) and (18.7) to yield

$$y_{i+1} = y_i + f(t_i, y_i)h + \frac{f'(t_i, y_i)}{2!} h^2 + \dots + \frac{f^{(n-1)}(t_i, y_i)}{n!} h^n + O(h^{n+1}) \quad (18.8)$$

where $O(h^{n+1})$ specifies that the local truncation error is proportional to the step size raised to the $(n+1)$ th power.

By comparing Eqs. (18.5) and (18.8), it can be seen that Euler's method corresponds to the Taylor series up to and including the term $f(t_i, y_i)h$. Additionally, the comparison indicates that a truncation error occurs because we approximate the true solution using a finite number of terms from the Taylor series. We thus truncate, or leave out, a part of the true solution. For example, the truncation error in Euler's method is attributable to the remaining terms in the Taylor series expansion that were not included in Eq. (18.5). Subtracting Eq. (18.5) from Eq. (18.8) yields

$$E_t = \frac{f'(t_i, y_i)}{2!} h^2 + \dots + O(h^{n+1}) \quad (18.9)$$

where $E_t =$ the true local truncation error. For sufficiently small h , the higher-order terms in Eq. (18.9) are usually negligible, and the result is often represented as

$$E_a = \frac{f'(t_i, y_i)}{2!} h^2 \quad (18.10)$$

or

$$E_a = O(h^2) \quad (18.11)$$

where $E_a =$ the approximate local truncation error.

According to Eq. (18.11), we see that the local error is proportional to the square of the step size and the first derivative of the differential equation. It can also be demonstrated that the global truncation error is $O(h)$ —that is, it is proportional to the step size

18.2 EULER'S METHOD

327

(Carnahan et al., 1969). These observations lead to some useful conclusions:

1. The global error can be reduced by decreasing the step size.
2. The method will provide error-free predictions if the underlying function (i.e., the solution of the differential equation) is linear, because for a straight line the second derivative would be zero.

This latter conclusion makes intuitive sense because Euler's method uses straight-line segments to approximate the solution. Hence, Euler's method is referred to as a *first-order method*.

It should also be noted that this general pattern holds for the higher-order one-step methods described in the following pages. That is, an n th-order method will yield perfect results if the underlying solution is an n th-order polynomial. Further, the local truncation error will be $O(h^{n+1})$ and the global error $O(h^n)$.

18.2.2 MATLAB M-file Function: Eulode

We have already developed a simple M-file to implement Euler's method for the falling bungee jumper problem in Chap. 3. Recall from Section 3.6, that this function used Euler's method to compute the velocity after a given time of free fall. Now, let's develop a more general, all-purpose algorithm.

Figure 18.3 shows an M-file that uses Euler's method to compute values of the dependent variable y over a range of values of the independent variable t . The name of the function holding the right-hand side of the differential equation is passed into the function as the variable `dydt`. The initial and final values of the desired range of the independent variable is passed as a vector `tspan`. The initial value and the desired step size are passed as `y0` and `h`, respectively.

The function first generates a vector t over the desired range of the dependent variable using an increment of h . In the event that the step size is not evenly divisible into the range, the last value will fall short of the final value of the range. If this occurs, the final value is added to t so that the series spans the complete range. The length of the t vector is determined as n . In addition, a vector of the dependent variable y is preallocated with n values of the initial condition to improve efficiency.

At this point, Euler's method (Eq. 18.5) is implemented by a simple loop:

```
for i = 1:n-1
    y(i+1) = y(i) + feval(dydt,t(i),y(i))*(t(i+1)-t(i));
end
```

Notice how the `feval` function is used to generate a value for the derivative at the appropriate values of the independent and dependent variables. Also notice how the time step is automatically calculated based on the difference between adjacent values in the vector t .

The ODE being solved can be set up in several ways. First, the differential equation can be defined as an `inline` function object. For example, for the ODE from Example 18.1:

```
>> dydt = inline('4*exp(0.8*t) - 0.5*y','t','y')
dydt =
    Inline function:
    dydt(t,y) = 4*exp(0.8*t) - 0.5*y
```

```
function [t,y] = Eulode(dydt,tspan,y0,h)
% [t,y] = Eulode(dydt,tspan,y0,h):
%   uses Euler's method to integrate an ODE
% input:
%   dydt = name of the M-file that evaluates the ODE
%   tspan = [ti, tf] where ti and tf = initial and
%           final values of independent variable
%   y0 = initial value of dependent variable
%   h = step size
% output:
%   t = vector of independent variable
%   y = vector of solution for dependent variable

ti = tspan(1);
tf = tspan(2);
t = (ti:h:tf)';
n = length(t);
% if necessary, add an additional value of t
% so that range goes from t = ti to tf
if t(n)<tf
    t(n+1) = tf;
    n = n+1;
end
y = y0*ones(n,1); %preallocate y to improve efficiency
for i = 1:n-1 %implement Euler's method
    y(i+1) = y(i) + feval(dydt,t(i),y(i))*(t(i+1)-t(i));
end
```

FIGURE 18.3

An M-file to implement Euler's method.

The solution can then be generated as

```
>> [t,y] = Eulode(dydt,[0 4],2,1);
>> disp([t,y])
```

with the result (compare with Table 18.1):

0	2.0000
1.0000	5.0000
2.0000	11.4022
3.0000	25.5132
4.0000	56.8493

Alternatively, a separate M-file can be developed to hold the ODE:

```
function dydt = diffeq(t,y)
dydt = 4*exp(0.8*t) - 0.5*y;
```

18.3 IMPROVEMENTS OF EULER'S METHOD

329

The solution can then be generated by passing the name of the M-file to `Euler` as a string:

```
>> [t,y] = Eulode('diffeq',[0 4],2,1);  
>> disp([t,y])
```

or as a function handle:

```
>> [t,y] = Eulode(@diffeq,[0 4],2,1);  
>> disp([t,y])
```

Although these two options are equally valid for the present case, there will be more complex problems where the definition of the ODE requires several lines of code. In such instances, creating a separate M-file is the only option.

18.3 IMPROVEMENTS OF EULER'S METHOD

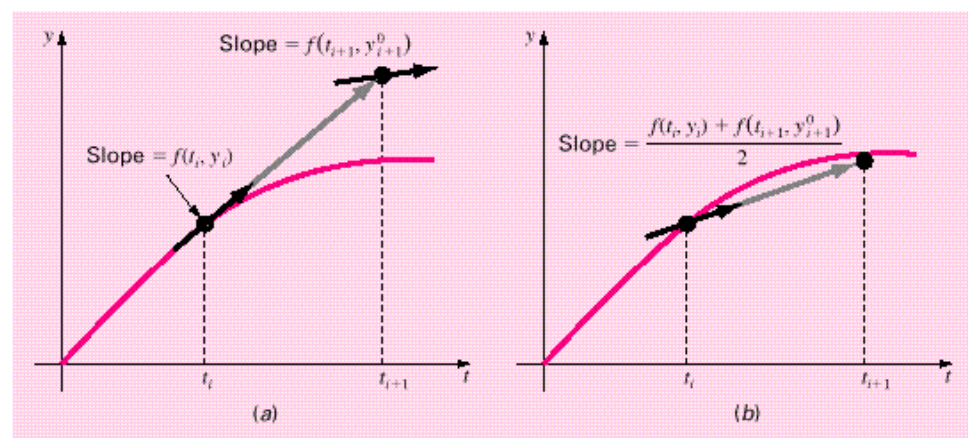
A fundamental source of error in Euler's method is that the derivative at the beginning of the interval is assumed to apply across the entire interval. Two simple modifications are available to help circumvent this shortcoming. As will be demonstrated in Section 18.4, both modifications (as well as Euler's method itself) actually belong to a larger class of solution techniques called Runge-Kutta methods. However, because they have very straightforward graphical interpretations, we will present them prior to their formal derivation as Runge-Kutta methods.

18.3.1 Heun's Method

One method to improve the estimate of the slope involves the determination of two derivatives for the interval—one at the beginning and another at the end. The two derivatives are then averaged to obtain an improved estimate of the slope for the entire interval. This approach, called *Heun's method*, is depicted graphically in Fig. 18.4.

FIGURE 18.4

Graphical depiction of Heun's method. (·) Predictor and (·) corrector.



Recall that in Euler's method, the slope at the beginning of an interval

$$y'_i = f(t_i, y_i) \quad (18.12)$$

is used to extrapolate linearly to y_{i+1} :

$$y_{i+1}^0 = y_i + f(t_i, y_i)h \quad (18.13)$$

For the standard Euler method we would stop at this point. However, in Heun's method the y_{i+1}^0 calculated in Eq. (18.13) is not the final answer, but an intermediate prediction. This is why we have distinguished it with a superscript 0. Equation (18.13) is called a *predictor equation*. It provides an estimate that allows the calculation of a slope at the end of the interval:

$$y'_{i+1} = f(t_{i+1}, y_{i+1}^0) \quad (18.14)$$

Thus, the two slopes [Eqs. (18.12) and (18.14)] can be combined to obtain an average slope for the interval:

$$\bar{y}' = \frac{f(t_i, y_i) + f(t_{i+1}, y_{i+1}^0)}{2} \quad (18.15)$$

This average slope is then used to extrapolate linearly from y_i to y_{i+1} using Euler's method:

$$y_{i+1} = y_i + \frac{f(t_i, y_i) + f(t_{i+1}, y_{i+1}^0)}{2}h \quad (18.16)$$

which is called a *corrector equation*.

The Heun method is a *predictor-corrector approach*. As just derived, it can be expressed concisely as

$$\text{Predictor (Fig. 18.4a): } y_{i+1}^0 = y_i^m + f(t_i, y_i)h \quad (18.17)$$

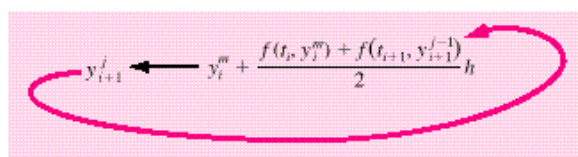
$$\text{Corrector (Fig. 18.4b): } y_{i+1}^j = y_i^m + \frac{f(t_i, y_i^m) + f(t_{i+1}, y_{i+1}^{j-1})}{2}h \quad (18.18)$$

(for $j = 1, 2, \dots, m$)

Note that because Eq. (18.18) has y_{i+1} on both sides of the equal sign, it can be applied in an iterative fashion as indicated. That is, an old estimate can be used repeatedly to provide an improved estimate of y_{i+1} . The process is depicted in Fig. 18.5.

FIGURE 18.5

Graphical representation of iterating the corrector of Heun's method to obtain an improved estimate.



18.3 IMPROVEMENTS OF EULER'S METHOD

331

As with similar iterative methods discussed in previous sections of the book, a termination criterion for convergence of the corrector is provided by

$$|\epsilon_a| = \left| \frac{y_{i-1}^j - y_{i-1}^{j-1}}{y_{i-1}^j} \right| \cdot 100\% \quad (18.19)$$

where y_{i-1}^{j-1} and y_{i-1}^j are the result from the prior and the present iteration of the corrector, respectively. It should be understood that the iterative process does not necessarily converge on the true answer but will converge on an estimate with a finite truncation error, as demonstrated in the following example.

EXAMPLE 18.2 Heun's Method

Problem Statement. Use Heun's method with iteration to integrate $y' = 4e^{0.8t} - 0.5y$ from $t = 0$ to 4 with a step size of 1. The initial condition at $t = 0$ is $y = 2$. Employ a stopping criterion of 0.00001% to terminate the corrector iterations.

Solution. First, the slope at (t_0, y_0) is calculated as

$$y_0' = 4e^0 - 0.5(2) = 3$$

Then, the predictor is used to compute a value at 1.0:

$$y_1^0 = 2 + 3(1) = 5$$

Note that this is the result that would be obtained by the standard Euler method. The true value in Table 18.2 shows that it corresponds to a percent relative error of 19.28%.

Now, to improve the estimate for y_1 , we use the value y_1^0 to predict the slope at the end of the interval

$$y_1' = f(x_1, y_1^0) = 4e^{0.8(1)} - 0.5(5) = 6.402164$$

which can be combined with the initial slope to yield an average slope over the interval from $t = 0$ to 1:

$$\bar{y}' = \frac{3 + 6.402164}{2} = 4.701082$$

This result can then be substituted into the corrector [Eq. (18.18)] to give the prediction at $t = 1$:

$$y_1^1 = 2 + 4.701082(1) = 6.701082$$

which represents a true percent relative error of 8.18%. Thus, the Heun method without iteration of the corrector reduces the absolute value of the error by a factor of about 2.4 as compared with Euler's method. At this point, we can also compute an approximate error as

$$|\epsilon_a| = \left| \frac{6.701082 - 5}{6.701082} \right| \cdot 100\% = 25.39\%$$

Now the estimate of y_1 can be refined by substituting the new result back into the right-hand side of Eq. (18.18) to give

$$y_1^2 = 2 + \frac{3 + 4e^{0.8(1)} - 0.5(6.701082)}{2} \cdot 1 = 6.275811$$

which represents a true percent relative error of 1.31 percent and an approximate error of

$$\epsilon_a = \left| \frac{6.275811 - 6.701082}{6.275811} \right| \cdot 100\% = 6.776\%$$

The next iteration gives

$$y_1^2 = 2 + \frac{3 - 4e^{0.8(1)} - 0.5(6.275811)}{2} = 6.382129$$

which represents a true error of 3.03% and an approximate error of 1.666%.

The approximate error will keep dropping as the iterative process converges on a stable final result. In this example, after 12 iterations the approximate error falls below the stopping criterion. At this point, the result at $t = 1$ is 6.36087, which represents a true relative error of 2.68%. Table 18.2 shows results for the remainder of the computation along with results for Euler's method and for the Heun method without iteration of the corrector.

TABLE 18.2 Comparison of true and numerical values of the integral of $y' = 4e^{0.8t} - 0.5y$, with the initial condition that $y = 2$ at $t = 0$. The numerical values were computed using the Euler and Heun methods with a step size of 1. The Heun method was implemented both without and with iteration of the corrector.

t	true	Euler	Error (%)	Without Iteration		With Iteration	
				Heun	Error (%)	Heun	Error (%)
0	2.00000	2.00000		2.00000		2.00000	
1	6.19463	5.00000	19.28	6.70108	8.18	6.36087	2.68
2	14.84392	11.40216	23.19	16.31978	9.94	15.30224	3.09
3	33.67717	25.51321	24.24	37.19925	10.46	34.74328	3.17
4	75.33896	56.84931	24.54	83.33777	10.62	77.73510	3.18

Insight into the local error of the Heun method can be gained by recognizing that it is related to the trapezoidal rule. In the previous example, the derivative is a function of both the dependent variable y and the independent variable t . For cases such as polynomials, where the ODE is solely a function of the independent variable, the predictor step [Eq. (18.17)] is not required and the corrector is applied only once for each iteration. For such cases, the technique is expressed concisely as

$$y_{i+1} = y_i + \frac{f(t_i) + f(t_{i+1})}{2}h \quad (18.20)$$

Notice the similarity between the second term on the right-hand side of Eq. (18.20) and the trapezoidal rule [Eq. (16.11)]. The connection between the two methods can be formally demonstrated by starting with the ordinary differential equation

$$\frac{dy}{dt} = f(t) \quad (18.21)$$

18.3 IMPROVEMENTS OF EULER'S METHOD

333

This equation can be solved for y by integration:

$$\int_{y_i}^{y_{i+1}} dy = \int_{t_i}^{t_{i+1}} f(t) dt \quad (18.22)$$

which yields

$$y_{i+1} - y_i = \int_{t_i}^{t_{i+1}} f(t) dt \quad (18.23)$$

or

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t) dt \quad (18.24)$$

Now, recall that the trapezoidal rule [Eq. (16.11)] is defined as

$$\int_{t_i}^{t_{i+1}} f(t) dt = \frac{f(t_i) + f(t_{i+1})}{2} h \quad (18.25)$$

where $h = t_{i+1} - t_i$. Substituting Eq. (18.25) into Eq. (18.24) yields

$$y_{i+1} = y_i + \frac{f(t_i) + f(t_{i+1})}{2} h \quad (18.26)$$

which is equivalent to Eq. (18.20). For this reason, Heun's method is sometimes referred to as the trapezoidal rule.

Because Eq. (18.26) is a direct expression of the trapezoidal rule, the local truncation error is given by [recall Eq. (16.14)]

$$E_t = -\frac{f''(\xi)}{12} h^3 \quad (18.27)$$

where ξ is between t_i and t_{i+1} . Thus, the method is second order because the second derivative of the ODE is zero when the true solution is a quadratic. In addition, the local and global errors are $O(h^3)$ and $O(h^2)$, respectively. Therefore, decreasing the step size decreases the error at a faster rate than for Euler's method.

18.3.2 The Midpoint Method

Figure 18.6 illustrates another simple modification of Euler's method. Called the *midpoint method*, this technique uses Euler's method to predict a value of y at the midpoint of the interval (Fig. 18.6a):

$$y_{i+1/2} = y_i + f(t_i, y_i) \frac{h}{2} \quad (18.28)$$

Then, this predicted value is used to calculate a slope at the midpoint:

$$y'_{i+1/2} = f(t_{i+1/2}, y_{i+1/2}) \quad (18.29)$$

which is assumed to represent a valid approximation of the average slope for the entire interval. This slope is then used to extrapolate linearly from t_i to t_{i+1} (Fig. 18.6b):

$$y_{i+1} = y_i + f(t_{i+1/2}, y_{i+1/2}) h \quad (18.30)$$

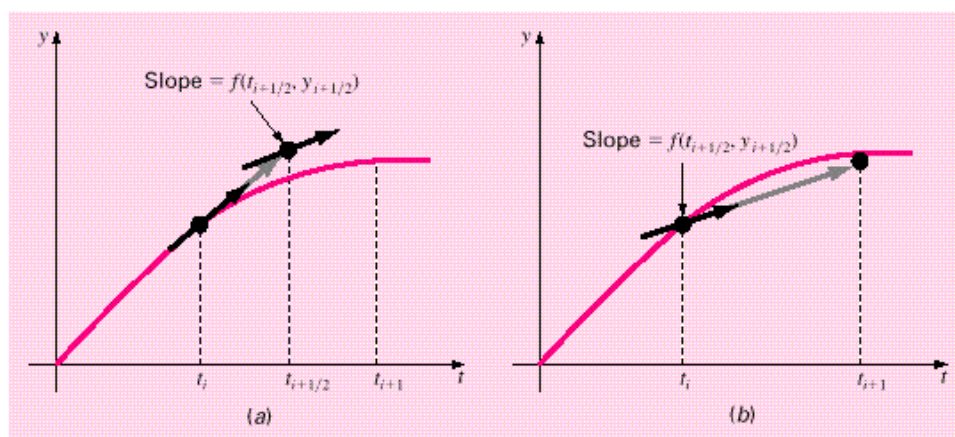


FIGURE 18.6

Graphical depiction of Heun's method. [· ·] Predictor and [· ·] corrector.

Observe that because y_{i+1} is not on both sides, the corrector [Eq. (18.30)] cannot be applied iteratively to improve the solution as was done with Heun's method.

As in our discussion of Heun's method, the midpoint method can also be linked to Newton-Cotes integration formulas. Recall from Table 16.4 that the simplest Newton-Cotes open integration formula, which is called the midpoint method, can be represented as

$$\int_a^b f(x) dx \approx (b - a) f(x_1) \quad (18.31)$$

where x_1 is the midpoint of the interval (a, b) . Using the nomenclature for the present case, it can be expressed as

$$\int_{t_i}^{t_{i+1}} f(t) dt \approx h f(t_{i+1/2}) \quad (18.32)$$

Substitution of this formula into Eq. (18.24) yields Eq. (18.30). Thus, just as the Heun method can be called the trapezoidal rule, the midpoint method gets its name from the underlying integration formula on which it is based.

The midpoint method is superior to Euler's method because it utilizes a slope estimate at the midpoint of the prediction interval. Recall from our discussion of numerical differentiation in Section 4.3.3 that centered finite divided differences are better approximations of derivatives than either forward or backward versions. In the same sense, a centered approximation such as Eq. (18.29) has a local truncation error of $O(h^2)$ in comparison with the forward approximation of Euler's method, which has an error of $O(h)$. Consequently, the local and global errors of the midpoint method are $O(h^3)$ and $O(h^2)$, respectively.

18.4 RUNGE-KUTTA METHODS

Runge-Kutta (RK) methods achieve the accuracy of a Taylor series approach without requiring the calculation of higher derivatives. Many variations exist but all can be cast in the generalized form of Eq. (18.4):

$$y_{i+1} = y_i + \phi h \quad (18.33)$$

where ϕ is called an *increment function*, which can be interpreted as a representative slope over the interval. The increment function can be written in general form as

$$\phi = a_1 k_1 + a_2 k_2 + \cdots + a_n k_n \quad (18.34)$$

where the a 's are constants and the k 's are

$$k_1 = f(t_i, y_i) \quad (18.34a)$$

$$k_2 = f(t_i + p_1 h, y_i + q_{11} k_1 h) \quad (18.34b)$$

$$k_3 = f(t_i + p_2 h, y_i + q_{21} k_1 h + q_{22} k_2 h) \quad (18.34c)$$

$$\vdots$$

$$k_n = f(t_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \cdots + q_{n-1,n-1} k_{n-1} h) \quad (18.34d)$$

where the p 's and q 's are constants. Notice that the k 's are recurrence relationships. That is, k_1 appears in the equation for k_2 , which appears in the equation for k_3 , and so forth. Because each k is a functional evaluation, this recurrence makes RK methods efficient for computer calculations.

Various types of Runge-Kutta methods can be devised by employing different numbers of terms in the increment function as specified by n . Note that the first-order RK method with $n = 1$ is, in fact, Euler's method. Once n is chosen, values for the a 's, p 's, and q 's are evaluated by setting Eq. (18.33) equal to terms in a Taylor series expansion. Thus, at least for the lower-order versions, the number of terms n usually represents the order of the approach. For example, in Section 18.4.1, second-order RK methods use an increment function with two terms ($n = 2$). These second-order methods will be exact if the solution to the differential equation is quadratic. In addition, because terms with h^3 and higher are dropped during the derivation, the local truncation error is $O(h^3)$ and the global error is $O(h^2)$. In Section 18.4.2, the fourth-order RK method ($n = 4$) is presented for which the global truncation error is $O(h^4)$.

18.4.1 Second-Order Runge-Kutta Methods

The second-order version of Eq. (18.33) is

$$y_{i+1} = y_i + (a_1 k_1 + a_2 k_2)h \quad (18.35)$$

where

$$k_1 = f(t_i, y_i) \quad (18.35a)$$

$$k_2 = f(t_i + p_1 h, y_i + q_{11} k_1 h) \quad (18.35b)$$

The values for a_1 , a_2 , p_1 , and q_{11} are evaluated by setting Eq. (18.35) equal to a second-order Taylor series. By doing this, three equations can be derived to evaluate the

four unknown constants (see Chapra and Canale, 2002, for details). The three equations are

$$a_1 + a_2 = 1 \quad (18.36)$$

$$a_2 p_1 = 1/2 \quad (18.37)$$

$$a_2 q_{11} = 1/2 \quad (18.38)$$

Because we have three equations with four unknowns, these equations are said to be underdetermined. We, therefore, must assume a value of one of the unknowns to determine the other three. Suppose that we specify a value for a_2 . Then Eqs. (18.36) through (18.38) can be solved simultaneously for

$$a_1 = 1 - a_2 \quad (18.39)$$

$$p_1 = q_{11} = \frac{1}{2a_2} \quad (18.40)$$

Because we can choose an infinite number of values for a_2 , there are an infinite number of second-order RK methods. Every version would yield exactly the same results if the solution to the ODE were quadratic, linear, or a constant. However, they yield different results when (as is typically the case) the solution is more complicated. Three of the most commonly used and preferred versions are presented next.

Heun Method without Iteration ($a_2 = 1/2$). If a_2 is assumed to be $1/2$, Eqs. (18.39) and (18.40) can be solved for $a_1 = 1/2$ and $p_1 = q_{11} = 1$. These parameters, when substituted into Eq. (18.35), yield

$$y_{i+1} = y_i + \left(\frac{1}{2}k_1 + \frac{1}{2}k_2 \right) h \quad (18.41)$$

where

$$k_1 = f(t_i, y_i) \quad (18.41a)$$

$$k_2 = f(t_i + h, y_i + k_1 h) \quad (18.41b)$$

Note that k_1 is the slope at the beginning of the interval and k_2 is the slope at the end of the interval. Consequently, this second-order Runge-Kutta method is actually Heun's technique without iteration of the corrector.

The Midpoint Method ($a_2 = 1$). If a_2 is assumed to be 1, then $a_1 = 0$, $p_1 = q_{11} = 1/2$, and Eq. (18.35) becomes

$$y_{i+1} = y_i + k_2 h \quad (18.42)$$

where

$$k_1 = f(t_i, y_i) \quad (18.42a)$$

$$k_2 = f(t_i + h/2, y_i + k_1 h/2) \quad (18.42b)$$

This is the midpoint method.

Ralston's Method ($a_2 = 2/3$). Ralston (1962) and Ralston and Rabinowitz (1978) determined that choosing $a_2 = 2/3$ provides a minimum bound on the truncation error for the second-order RK algorithms. For this version, $a_1 = 1/3$ and $p_1 = q_{11} = 3/4$, and

18.4 RUNGE-KUTTA METHODS

337

Eq. (18.35) becomes

$$y_{i+1} = y_i + \left(\frac{1}{3}k_1 + \frac{2}{3}k_2 \right) h \quad (18.43)$$

where

$$k_1 = f(x_i, y_i) \quad (18.43a)$$

$$k_2 = f\left(t_i + \frac{3}{4}h, y_i + \frac{3}{4}k_1h\right) \quad (18.43b)$$

18.4.2 Classical Fourth-Order Runge-Kutta Method

The most popular RK methods are fourth order. As with the second-order approaches, there are an infinite number of versions. The following is the most commonly used form, and we therefore call it the *classical fourth-order RK method*:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \quad (18.44)$$

where

$$k_1 = f(t_i, y_i) \quad (18.44a)$$

$$k_2 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right) \quad (18.44b)$$

$$k_3 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2h\right) \quad (18.44c)$$

$$k_4 = f(t_i + h, y_i + k_3h) \quad (18.44d)$$

Notice that for ODEs that are a function of t alone, the classical fourth-order RK method is similar to Simpson's 1/3 rule. In addition, the fourth-order RK method is similar to the Heun approach in that multiple estimates of the slope are developed to come up with an improved average slope for the interval. As depicted in Fig. 18.7, each of the k 's represents a slope. Equation (18.44) then represents a weighted average of these to arrive at the improved slope.

EXAMPLE 18.3 Classical Fourth-Order RK Method

Problem Statement. Employ the classical fourth-order RK method to integrate $y' = 4e^{0.8t} - 0.5y$ from $t = 0$ to 1 using a step size of 1 with $y(0) = 2$.

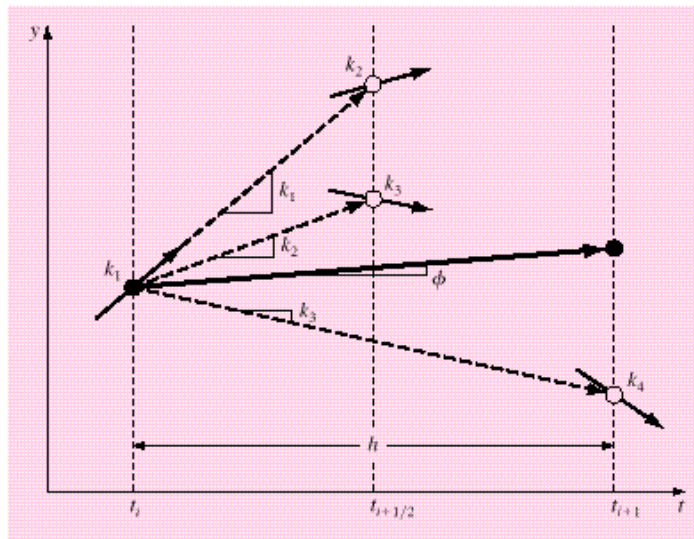
Solution. For this case, the slope at the beginning of the interval is computed as

$$k_1 = f(0, 2) = 4e^{0.8(0)} - 0.5(2) = 3$$

This value is used to compute a value of y and a slope at the midpoint:

$$y(0.5) = 2 + 3(0.5) = 3.5$$

$$k_2 = f(0.5, 3.5) = 4e^{0.8(0.5)} - 0.5(3.5) = 4.217299$$


FIGURE 18.7

Graphical depiction of the slope estimates comprising the fourth-order RK method.

This slope in turn is used to compute another value of y and another slope at the midpoint:

$$y(0.5) = 2 + 4.217299(0.5) = 4.108649$$

$$k_3 = f(0.5, 4.108649) = 4e^{0.8(0.5)} = 0.5(4.108649) = 3.912974$$

Next, this slope is used to compute a value of y and a slope at the end of the interval:

$$y(1.0) = 2 + 3.912974(1.0) = 5.912974$$

$$k_4 = f(1.0, 5.912974) = 4e^{0.8(1.0)} = 0.5(5.912974) = 5.945677$$

Finally, the four slope estimates are combined to yield an average slope. This average slope is then used to make the final prediction at the end of the interval.

$$\phi = \frac{1}{6} [3 + 2(4.217299) + 2(3.912974) + 5.945677] = 4.201037$$

$$y(1.0) = 2 + 4.201037(1.0) = 6.201037$$

which compares favorably with the true solution of 6.194631 ($\epsilon_t = 0.103\%$).

It is certainly possible to develop fifth- and higher-order RK methods. For example, Butcher's (1964) fifth-order RK method is written as

$$y_{i+1} = y_i + \frac{1}{90}(7k_1 + 32k_3 + 12k_4 + 32k_5 + 7k_6)h \quad (18.45)$$

18.5 SYSTEMS OF EQUATIONS

339

where

$$k_1 = f(t_i, y_i) \quad (18.45a)$$

$$k_2 = f\left(t_i + \frac{1}{4}h, y_i + \frac{1}{4}k_1h\right) \quad (18.45b)$$

$$k_3 = f\left(t_i + \frac{1}{4}h, y_i + \frac{1}{8}k_1h + \frac{1}{8}k_2h\right) \quad (18.45c)$$

$$k_4 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2h + k_3h\right) \quad (18.45d)$$

$$k_5 = f\left(t_i + \frac{3}{4}h, y_i + \frac{3}{16}k_1h + \frac{9}{16}k_4h\right) \quad (18.45e)$$

$$k_6 = f\left(t_i + h, y_i + \frac{3}{7}k_1h + \frac{2}{7}k_2h + \frac{12}{7}k_3h + \frac{12}{7}k_4h + \frac{8}{7}k_5h\right) \quad (18.45f)$$

Note the similarity between Butcher's method and Boole's rule in Table 16.2. As expected, this method has a global truncation error of $O(h^5)$.

Although the fifth-order version provides more accuracy, notice that six function evaluations are required. Recall that up through the fourth-order versions, n function evaluations are required for an n th-order RK method. Interestingly, for orders higher than four, one or two additional function evaluations are necessary. Because the function evaluations account for the most computation time, methods of order five and higher are usually considered relatively less efficient than the fourth-order versions. This is one of the main reasons for the popularity of the fourth-order RK method.

18.5 SYSTEMS OF EQUATIONS

Many practical problems in engineering and science require the solution of a system of simultaneous ordinary differential equations rather than a single equation. Such systems may be represented generally as

$$\begin{aligned} \frac{dy_1}{dt} &= f_1(t, y_1, y_2, \dots, y_n) \\ \frac{dy_2}{dt} &= f_2(t, y_1, y_2, \dots, y_n) \\ &\vdots \\ \frac{dy_n}{dt} &= f_n(t, y_1, y_2, \dots, y_n) \end{aligned} \quad (18.46)$$

The solution of such a system requires that n initial conditions be known at the starting value of t .

An example is the calculation of the bungee jumper's velocity and position that we set up at the beginning of this chapter. For the free-fall portion of the jump, this problem

amounts to solving the following system of ODEs:

$$\frac{dx}{dt} = v \quad (18.47)$$

$$\frac{dv}{dt} = g - \frac{c_d}{m}v^2 \quad (18.48)$$

If the stationary platform from which the jumper launches is defined as $x = 0$, the initial conditions would be $x(0) = v(0) = 0$.

18.5.1 Euler's Method

All the methods discussed in this chapter for single equations can be extended to systems of ODEs. Engineering applications can involve thousands of simultaneous equations. In each case, the procedure for solving a system of equations simply involves applying the one-step technique for every equation at each step before proceeding to the next step. This is best illustrated by the following example for Euler's method.

EXAMPLE 18.4 Solving Systems of ODEs with Euler's Method

Problem Statement. Solve for the velocity and position of the free-falling bungee jumper using Euler's method. Assuming that at $t = 0$, $x = v = 0$, and integrate to $t = 10$ s with a step size of 2 s. As was done previously in Examples 1.1 and 1.2, the gravitational acceleration is 9.81 m/s^2 , and the jumper has a mass of 68.1 kg with a drag coefficient of 0.25 kg/m .

Recall that the analytical solution for velocity is [Eq. (1.9)]:

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}}t\right)$$

This result can be substituted into Eq. (18.47) which can be integrated to determine an analytical solution for distance as

$$x(t) = \frac{m}{c_d} \ln \left[\cosh\left(\sqrt{\frac{gc_d}{m}}t\right) \right]$$

Use these analytical solutions to compute the true relative errors of the results.

Solution. The ODEs can be used to compute the slopes at $t = 0$ as

$$\begin{aligned} \frac{dx}{dt} &= 0 \\ \frac{dv}{dt} &= 9.81 - \frac{0.25}{68.1}(0)^2 = 9.81 \end{aligned}$$

Euler's method is then used to compute the values at $t = 2$ s,

$$\begin{aligned} x &= 0 + 0(2) = 0 \\ v &= 0 + 9.81(2) = 19.62 \end{aligned}$$

The analytical solutions can be computed as $x(2) = 19.16629$ and $v(2) = 18.72919$. Thus, the percent relative errors are 100% and 4.756%, respectively.

18.5 SYSTEMS OF EQUATIONS

341

The process can be repeated to compute the results at $t = 4$ as

$$x = 0 + 19.62(2) = 39.24$$

$$v = 19.62 + \left(9.81 + \frac{0.25}{68.1}(19.62)^2\right)2 = 36.41368$$

Proceeding in a like manner gives the results displayed in Table 18.3.

TABLE 18.3 Distance and velocity of a free-falling bungee jumper as computed numerically with Euler's method.

t	x_{true}	v_{true}	x_{Euler}	v_{Euler}	$\frac{ x_{\text{Euler}} - x_{\text{true}} }{x_{\text{true}}}$	$\frac{ v_{\text{Euler}} - v_{\text{true}} }{v_{\text{true}}}$
0	0	0	0	0		
2	19.1663	18.7292	0	19.6200	100.00%	4.76%
4	71.9304	33.1118	39.2400	36.4137	45.45%	9.97%
6	147.9462	42.0762	112.0674	46.2983	24.25%	10.03%
8	237.5104	46.9575	204.6640	50.1802	13.83%	6.86%
10	334.1782	49.4214	305.0244	51.3123	8.72%	3.83%

Although the foregoing example illustrates how Euler's method can be implemented for systems of ODEs, the results are not very accurate because of the large step size. In addition, the results for distance are a bit unsatisfying because x does not change until the second iteration. Using a much smaller step greatly mitigates these deficiencies. As described next, using a higher-order solver provides decent results even with a relatively large step size.

18.5.2 Runge-Kutta Methods

Note that any of the higher-order RK methods in this chapter can be applied to systems of equations. However, care must be taken in determining the slopes. Figure 18.7 is helpful in visualizing the proper way to do this for the fourth-order method. That is, we first develop slopes for all variables at the initial value. These slopes (a set of k_1 's) are then used to make predictions of the dependent variable at the midpoint of the interval. These midpoint values are in turn used to compute a set of slopes at the midpoint (the k_2 's). These new slopes are then taken back to the starting point to make another set of midpoint predictions that lead to new slope predictions at the midpoint (the k_3 's). These are then employed to make predictions at the end of the interval that are used to develop slopes at the end of the interval (the k_4 's). Finally, the k 's are combined into a set of increment functions [as in Eq. (18.44)] that are brought back to the beginning to make the final predictions. The following example illustrates the approach.

EXAMPLE 18.5 Solving Systems of ODEs with the Fourth-Order RK Method

Problem Statement. Use the fourth-order RK method to solve for the same problem we addressed in Example 18.4.

Solution. First, it is convenient to express the ODEs in the functional format of Eq. (18.46) as

$$\begin{aligned}\frac{dx}{dt} &= f_1(t, x, v) = v \\ \frac{dv}{dt} &= f_2(t, x, v) = g = \frac{c_d}{m}v^2\end{aligned}$$

The first step in obtaining the solution is to solve for all the slopes at the beginning of the interval:

$$\begin{aligned}k_{1,1} &= f_1(0, 0, 0) = 0 \\ k_{1,2} &= f_2(0, 0, 0) = 9.81 = \frac{0.25}{68.1}(0)^2 = 9.81\end{aligned}$$

where $k_{i,j}$ is the i th value of k for the j th dependent variable. Next, we must calculate the first values of x and v at the midpoint of the first step:

$$\begin{aligned}x(1) &= x(0) + k_{1,1}\frac{h}{2} = 0 + 0\frac{2}{2} = 0 \\ v(1) &= v(0) + k_{1,2}\frac{h}{2} = 0 + 9.81\frac{2}{2} = 9.81\end{aligned}$$

which can be used to compute the first set of midpoint slopes:

$$\begin{aligned}k_{2,1} &= f_1(1, 0, 9.81) = 9.8100 \\ k_{2,2} &= f_2(1, 0, 9.81) = 9.4567\end{aligned}$$

These are used to determine the second set of midpoint predictions:

$$\begin{aligned}x(1) &= x(0) + k_{2,1}\frac{h}{2} = 0 + 9.8100\frac{2}{2} = 9.8100 \\ v(1) &= v(0) + k_{2,2}\frac{h}{2} = 0 + 9.4567\frac{2}{2} = 9.4567\end{aligned}$$

which can be used to compute the second set of midpoint slopes:

$$\begin{aligned}k_{3,1} &= f_1(1, 9.8100, 9.4567) = 9.4567 \\ k_{3,2} &= f_2(1, 9.8100, 9.4567) = 9.4817\end{aligned}$$

These are used to determine the predictions at the end of the interval:

$$\begin{aligned}x(2) &= x(0) + k_{3,1}h = 0 + 9.4567(2) = 18.9134 \\ v(2) &= v(0) + k_{3,2}h = 0 + 9.4817(2) = 18.9634\end{aligned}$$

which can be used to compute the endpoint slopes:

$$\begin{aligned}k_{4,1} &= f_1(2, 18.9134, 18.9634) = 18.9634 \\ k_{4,2} &= f_2(2, 18.9134, 18.9634) = 8.4898\end{aligned}$$

PROBLEMS

343

The values of k can then be used to compute [Eq. (18.44)]:

$$x(2) = 0 + \frac{1}{6} [0 + 2(9.8100 + 9.4567) + 18.9634] 2 = 19.1656$$

$$v(2) = 0 + \frac{1}{6} [9.8100 + 2(9.4567 + 9.4817) + 8.4898] 2 = 18.7256$$

Proceeding in a like manner for the remaining steps yields the values displayed in Table 18.4. In contrast to the results obtained with Euler's method, the fourth-order RK predictions are much closer to the true values. Further, a highly accurate, nonzero value is computed for distance on the first step.

TABLE 18.4 Distance and velocity of a free-falling bungee jumper as computed numerically with the fourth-order RK method.

t	x (true)	v (true)	x (RK4)	v (RK4)	ϵ_x (%)	ϵ_v (%)
0	0	0	0	0		
2	19.1663	18.7292	19.1656	18.7256	0.004%	0.019%
4	71.9304	33.1118	71.9311	33.0995	0.001%	0.037%
6	147.9462	42.0762	147.9521	42.0547	0.004%	0.051%
8	237.5104	46.9575	237.5104	46.9345	0.000%	0.049%
10	334.1782	49.4214	334.1626	49.4027	0.005%	0.038%

PROBLEMS

18.1 Solve the following initial value problem over the interval from $t = 0$ to 2 where $y(0) = 1$. Display all your results on the same graph.

$$\frac{dy}{dt} = yt^3 + 1.5y$$

- (a) Analytically.
- (b) Using Euler's method with $h = 0.5$ and 0.25 .
- (c) Using the midpoint method with $h = 0.5$.
- (d) Using the fourth-order RK method with $h = 0.5$.

18.2 Solve the following problem over the interval from $x = 0$ to 1 using a step size of 0.25 where $y(0) = 1$. Display all your results on the same graph.

$$\frac{dy}{dx} = (1 - 2x) \sqrt{y}$$

- (a) Analytically.
- (b) Using Euler's method.
- (c) Using Heun's method without the corrector.
- (d) Using Ralston's method.
- (e) Using the fourth-order RK method.

18.3 Solve the following problem over the interval from $x = 0$ to 2 using a step size of 0.5 where $y(0) = 1$. Display

all your results on the same graph.

$$\frac{dy}{dt} = -2y + t^2$$

Obtain your solutions with (a) Heun's method without iterating the corrector, (b) Heun's method with iterating the corrector until $\epsilon_s < 0.1\%$, (c) the midpoint method, and (d) Ralston's method.

18.4 The growth of populations of organisms has many engineering and scientific applications. One of the simplest models assumes that the rate of change of the population p is proportional to the existing population at any time t :

$$\frac{dp}{dt} = k_g p \quad (\text{P18.4.1})$$

where k_g = the growth rate. The world population in millions from 1950 through 2000 was

t	1950	1955	1960	1965	1970	1975
p	2555	2780	3040	3346	3708	4087
t	1980	1985	1990	1995	2000	
p	4454	4850	5276	5686	6079	

- (a) Assuming that Eq. (P18.4.1) holds, use the data from 1950 through 1970 to estimate k_g .
- (b) Use the fourth-order RK method along with the results of (a) to stimulate the world population from 1950 to 2050 with a step size of 5 years. Display your simulation results along with the data on a plot.

18.5 Although the model in Prob. 18.4 works adequately when population growth is unlimited, it breaks down when factors such as food shortages, pollution, and lack of space inhibit growth. In such cases, the growth rate is not a constant, but can be formulated as

$$k_g = k_{gm}(1 - p/p_{\max})$$

where k_{gm} = the maximum growth rate under unlimited conditions, p = population, and p_{\max} = the maximum population. Note that p_{\max} is sometimes called the *carrying capacity*. Thus, at low population density $p \ll p_{\max}$, $k_g \approx k_{gm}$. As p approaches p_{\max} , the growth rate approaches zero. Using this growth rate formulation, the rate of change of population can be modeled as

$$\frac{dp}{dt} = k_{gm}(1 - p/p_{\max})p$$

This is referred to as the *logistic model*. The analytical solution to this model is

$$p = p_0 \frac{p_{\max}}{p_0 + (p_{\max} - p_0)e^{-k_{gm}t}}$$

Simulate the world's population from 1950 to 2050 using (a) the analytical solution, and (b) the fourth-order RK method with a step size of 5 years. Employ the following initial conditions and parameter values for your simulation: p_0 (in 1950) = 2,555 million people, $k_{gm} = 0.026/\text{yr}$, and $p_{\max} = 12,000$ million people. Display your results as a plot along with the data from Prob. 18.4.

18.6 The free-fall velocity of a parachutist can be simulated as

$$\frac{dv}{dt} = g - \frac{c_d}{m}v^2$$

where v = velocity (m/s), t = time (s), g = acceleration due to gravity (9.81 m/s^2), c_d = drag coefficient (kg/m), and m = mass (kg). For a 80-kg parachutist, solve this equation numerically from $t = 0$ to 30 s given that $v(0) = 0$. During free fall, $c_d = 0.25 \text{ kg/m}$. However, at $t = 10$ s, the chute opens, whereupon $c_d = 5 \text{ kg/m}$.

18.7 Solve the following pair of ODEs over the interval from $t = 0$ to 0.4 using a step size of 0.1. The initial conditions are $y(0) = 2$ and $z(0) = 4$. Obtain your solution with (a) Euler's method and (b) the fourth-order RK method. Display your results as a plot.

$$\begin{aligned}\frac{dy}{dt} &= 2y - 5ze^{-t} \\ \frac{dz}{dt} &= \frac{yz^2}{2}\end{aligned}$$

18.8 The *van der Pol equation* is a model of an electronic circuit that arose back in the days of vacuum tubes:

$$\frac{d^2y}{dt^2} + (1 - y^2)\frac{dy}{dt} - y = 0$$

Given the initial conditions, $y(0) = y'(0) = 1$, solve this equation from $t = 0$ to 10 using Euler's method with a step size of (a) 0.2 and (b) 0.1. Plot both solutions on the same graph.

18.9 Given the initial conditions, $y(0) = 1$ and $y'(0) = 0$, solve the following initial-value problem from $t = 0$ to 4:

$$\frac{d^2y}{dt^2} - 9y = 0$$

Obtain your solutions with (a) Euler's method and (b) the fourth-order RK method. In both cases, use a step size of 0.1. Plot both solutions on the same graph along with the exact solution $y = \cos 3t$.

18.10 Develop an M-file to solve a single ODE with Heun's method with iteration. Design the M-file so that it creates a plot of the results. Test your program by using it to solve for population as described in Prob. 18.5. Employ a step size of 5 years and iterate the corrector until $\epsilon_s < 0.1\%$.

18.11 Develop an M-file to solve a single ODE with the midpoint method. Design the M-file so that it creates a plot of the results. Test your program by using it to solve for population as described in Prob. 18.5. Employ a step size of 5 years.

18.12 Develop an M-file to solve a single ODE with the fourth-order RK method. Design the M-file so that it creates a plot of the results. Test your program by using it to solve Prob. 18.2. Employ a step size of 0.1.

18.13 Develop an M-file to solve a pair of ODEs with the fourth-order RK method. Design the M-file so that it creates a plot of the results. Test your program by using it to solve Prob. 18.7 with a step size of 0.25.

ODEs: Adaptive Methods and Stiff Systems

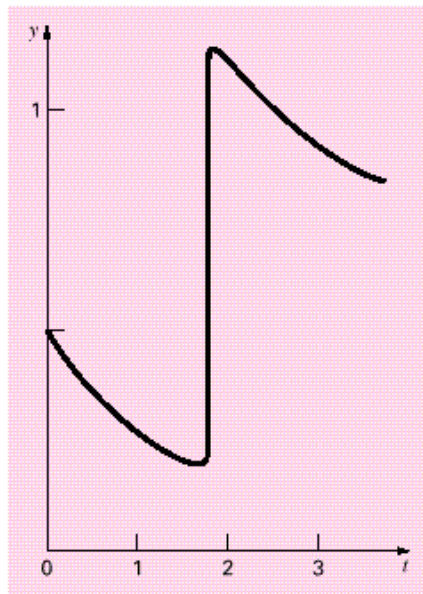
CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to more advanced methods for solving initial-value problems for ordinary differential equations. Specific objectives and topics covered are

- Understanding how the Runge-Kutta Fehlberg methods use RK methods of different orders to provide error estimates that are used to adjust the step size.
- Familiarizing yourself with the built-in MATLAB functions for solving ODEs.
- Learning how to adjust the options for MATLAB's ODE solvers.
- Learning how to pass parameters via MATLAB's ODE solvers.
- Understanding the difference between one-step and multistep methods for solving ODEs.
- Understanding what is meant by stiffness and its implications for solving ODEs.

19.1 ADAPTIVE RUNGE-KUTTA METHODS

To this point, we have presented methods for solving ODEs that employ a constant step size. For a significant number of problems, this can represent a serious limitation. For example, suppose that we are integrating an ODE with a solution of the type depicted in Fig. 19.1. For most of the range, the solution changes gradually. Such behavior suggests that a fairly large step size could be employed to obtain adequate results. However, for a localized region from $t = 1.75$ to 2.25 , the solution undergoes an abrupt change. The practical consequence of dealing with such functions is that a very small step size would be required to accurately capture the impulsive behavior. If a constant step-size algorithm were employed, the smaller step size required for the region of abrupt change would have

**FIGURE 19.1**

An example of a solution of an ODE that exhibits an abrupt change. Automatic step-size adjustment has great advantages for such cases.

to be applied to the entire computation. As a consequence, a much smaller step size than necessary—and, therefore, many more calculations—would be wasted on the regions of gradual change.

Algorithms that automatically adjust the step size can avoid such overkill and hence be of great advantage. Because they “adapt” to the solution’s trajectory, they are said to have *adaptive step-size control*. Implementation of such approaches requires that an estimate of the local truncation error be obtained at each step. This error estimate can then serve as a basis for either shortening or lengthening the step size.

Before proceeding, we should mention that aside from solving ODEs, the methods described in this chapter can also be used to evaluate definite integrals. The evaluation of the definite integral

$$I = \int_a^b f(x) dx$$

is equivalent to solving the differential equation

$$\frac{dy}{dx} = f(x)$$

for $y(b)$ given the initial condition $y(a) = 0$. Thus, the following techniques can be employed to efficiently evaluate definite integrals involving functions that are generally smooth but exhibit regions of abrupt change.

There are two primary approaches to incorporate adaptive step-size control into one-step methods. *Step halving* involves taking each step twice, once as a full step and then as two half steps. The difference in the two results represents an estimate of the local truncation error. The step size can then be adjusted based on this error estimate.

In the second approach, called *embedded RK methods*, the local truncation error is estimated as the difference between two predictions using different-order RK methods. These are currently the methods of choice because they are more efficient than step halving.

The embedded methods were first developed by Fehlberg. Hence, they are sometimes referred to as *RK-Fehlberg methods*. At face value, the idea of using two predictions of different order might seem too computationally expensive. For example, a fourth- and fifth-order prediction amounts to a total of 10 function evaluations per step [recall Eqs. (18.44) and (18.45)]. Fehlberg cleverly circumvented this problem by deriving a fifth-order RK method that employs most of the same function evaluations required for an accompanying fourth-order RK method. Thus, the approach yielded the error estimate on the basis of only six function evaluations!

19.1.1 MATLAB Functions for Nonstiff Systems

Since Fehlberg originally developed his approach, other even better approaches have been developed. Several of these are available as built-in functions in MATLAB.

..... The `ode23` function uses the BS23 algorithm (Bogacki and Shampine, 1989; Shampine, 1994), which simultaneously uses second- and third-order RK formulas to solve the ODE and make error estimates for step-size adjustment. The formulas to advance the solution are

$$y_{i+1} = y_i + \frac{1}{9}(2k_1 + 3k_2 + 4k_3)h \quad (19.1)$$

where

$$k_1 = f(t_i, y_i) \quad (19.1a)$$

$$k_2 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right) \quad (19.1b)$$

$$k_3 = f\left(t_i + \frac{3}{4}h, y_i + \frac{3}{4}k_2h\right) \quad (19.1c)$$

The error is estimated as

$$E_{i+1} = \frac{1}{72}(-5k_1 + 6k_2 + 8k_3 + 9k_4)h \quad (19.2)$$

where

$$k_4 = f(t_{i+1}, y_{i+1}) \quad (19.2a)$$

Note that although there appear to be four function evaluations, there are really only three because after the first step, the k_1 for the present step will be the k_4 from the previous step. Thus, the approach yields a prediction and error estimate based on three evaluations rather

than the five that would ordinarily result from using second- (two evaluations) and third-order (three evaluations) RK formulas in tandem.

After each step, the error is checked to determine whether it is within a desired tolerance. If it is, the value of y_{i+1} is accepted, and k_4 becomes k_1 for the next step. If the error is too large, the step is repeated with reduced step sizes until the estimated error satisfies

$$E \leq \max(\text{RelTol} \cdot \|y\|, \text{AbsTol}) \quad (19.3)$$

where RelTol is the relative tolerance (default $\cdot 10^{-3}$) and AbsTol is the absolute tolerance (default $\cdot 10^{-6}$). Observe that the criteria for the relative error uses a fraction rather than a percent relative error as we have done on many occasions prior to this point.

..... The `ode45` function uses an algorithm developed by Dormand and Prince (1990), which simultaneously uses fourth- and fifth-order RK formulas to solve the ODE and make error estimates for step-size adjustment. MATLAB recommends that `ode45` is the best function to apply as a “first try” for most problems.

..... The `ode113` function uses a variable-order Adams-Bashforth-Moulton solver. It is useful for stringent error tolerances or computationally intensive ODE functions. Note that this is a multistep method as we will describe subsequently in Section 19.2.

These functions can be called in a number of different ways. The simplest approach is

$$[t, y] = \text{ode45}(\text{odefun}, tspan, y0)$$

where y is the solution array where each column is one of the dependent variables and each row corresponds to a time in the column vector t , `odefun` is the name of the function returning a column vector of the right-hand-sides of the differential equations, `tspan` specifies the integration interval, and $y0$ is a vector containing the initial values.

Note that `tspan` can be formulated in two ways. First, if it is entered as a vector of two numbers,

$$tspan = [ti \ tf];$$

the integration is performed from ti to tf . Second, to obtain solutions at specific times $t0, t1, \dots, tn$ (all increasing or all decreasing), use

$$tspan = [t0 \ t1 \ \dots \ tn];$$

EXAMPLE 19.1 Using MATLAB to Solve a System of

Problem Statement. Employ `ode45` to solve the following set of nonlinear ODEs from $t = 0$ to 20:

$$\frac{dy_1}{dt} = 1.2y_1 - 0.6y_1y_2 \quad \frac{dy_2}{dt} = -0.8y_2 + 0.3y_1y_2$$

where $y_1 = 2$ and $y_2 = 1$ at $t = 0$. Such equations are referred to as *predator-prey equations*.

Solution. Before obtaining a solution with MATLAB, you must create a function to compute the right-hand side of the ODEs. One way to do this is to create an M-file as in

```
function yp = predprey(t,y)
yp = [1.2*y(1)-0.6*y(1)*y(2); -0.8*y(2)+0.3*y(1)*y(2)];
```

We stored this M-file under the name: `predprey.m`.

Next, enter the following commands to specify the integration range and the initial conditions:

```
>> tspan = [0 20];  
>> y0 = [2, 1];
```

The solver can then be invoked by

```
>> [t,y] = ode45(@predprey, tspan, y0);
```

This command will then solve the differential equations in `predprey.m` over the range defined by `tspan` using the initial conditions found in `y0`. The results can be displayed by simply typing

```
>> plot(t,y)
```

which yields Fig. 19.2.

In addition to a time series plot, it is also instructive to generate a *state-space plot*—that is, a plot of the dependent variables versus each other by

```
>> plot(y(:,1),y(:,2))
```

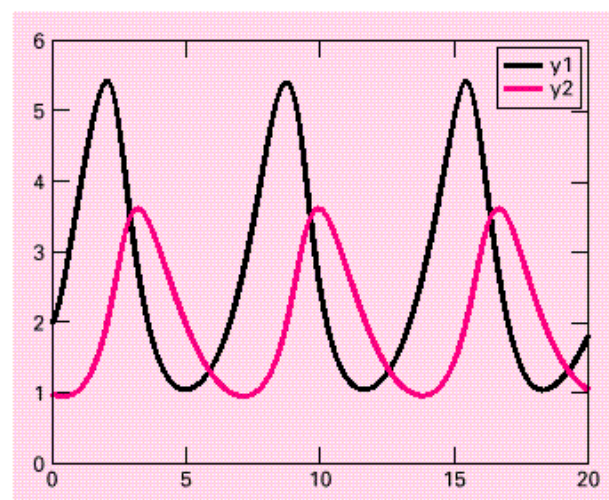
which yields Fig. 19.3.

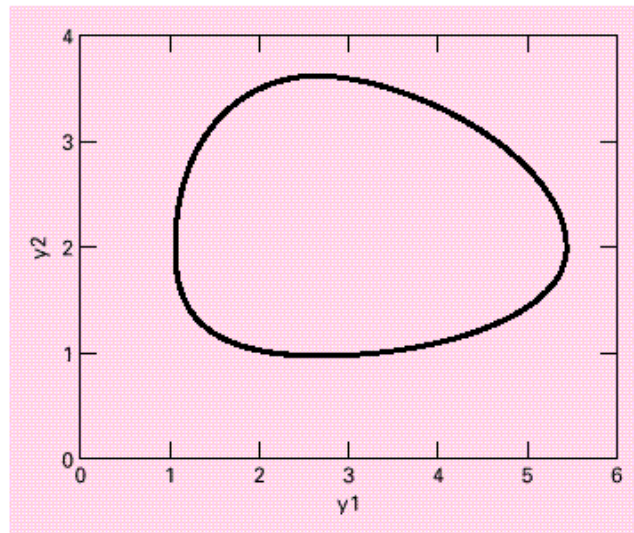
As in the previous example, the MATLAB solver uses default parameters to control various aspects of the integration. In addition, there is also no control over the differential equations' parameters. To have control over these features, additional arguments are included as in

```
[t, y] = ode45(odefun, tspan, y0, options, p1, p2,...)
```

FIGURE 19.2

Solution of predator-prey model with MATLAB.



**FIGURE 19.3**

State-space plot of predator-prey model with MATLAB.

where *options* is a data structure that is created with the *odeset* function to control features of the solution, and *p1*, *p2*, ... are parameters that you want to pass into *odefun*.

The *odeset* function has the general syntax

```
options = odeset('par1',val1,'par2',val2,...)
```

where the parameter *par_i* has the value *val_i*. A complete listing of all the possible parameters can be obtained by merely entering *odeset* at the command prompt. Some commonly used parameters are

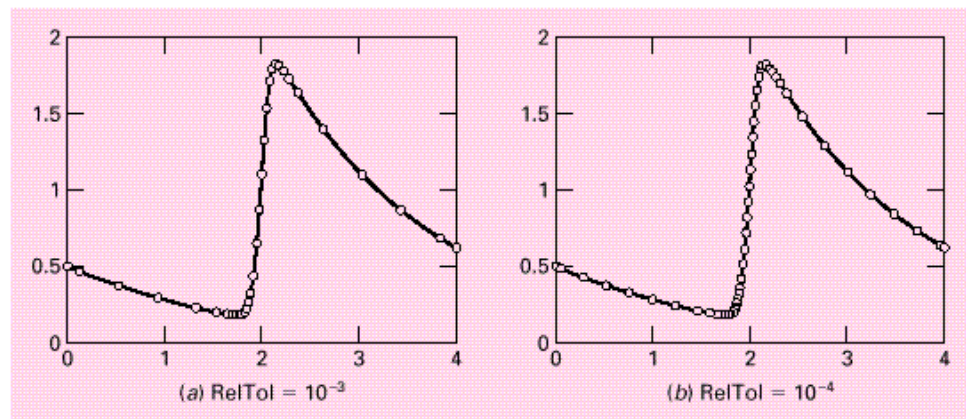
'RelTol'	Allows you to adjust the relative tolerance.
'AbsTol'	Allows you to adjust the absolute tolerance.
'InitialStep'	The solver automatically determines the initial step. This option allows you to set your own.
'MaxStep'	The maximum step defaults to one-tenth of the <i>tspan</i> interval. This option allows you to override this default.

EXAMPLE 19.2 Using to Control Integration Options

Problem Statement. Use *ode23* to solve the following ODE from $t = 0$ to 4:

$$\frac{dy}{dt} = 10e^{-(t-2)^2/[2(0.075)^2]} - 0.6y$$

where $y(0) = 0.5$. Obtain solutions for the default (10^{-3}) and for a more stringent (10^{-4}) relative error tolerance.

**FIGURE 19.4**

Solution of ODE with MATLAB. For [-], a smaller relative error tolerance is used and hence many more steps are taken.

Solution. First, we will create an M-file to compute the right-hand side of the ODE:

```
function yp = dydt(t, y)
yp = 10*exp(-(t-2)*(t-2)/(2*.075^2))-0.6*y;
```

Then, we can implement the solver without setting the options. Hence the default value for the relative error (10^{-3}) is automatically used:

```
>> ode23(@dydt, [0 4], 0.5);
```

Note that we have not set the function equal to output variables $[t, y]$. When we implement one of the ODE solvers in this way, MATLAB automatically creates a plot of the results displaying circles at the values it has computed. As in Fig. 19.4a, notice how `ode23` takes relatively large steps in the smooth regions of the solution whereas it takes smaller steps in the region of rapid change around $t = 2$.

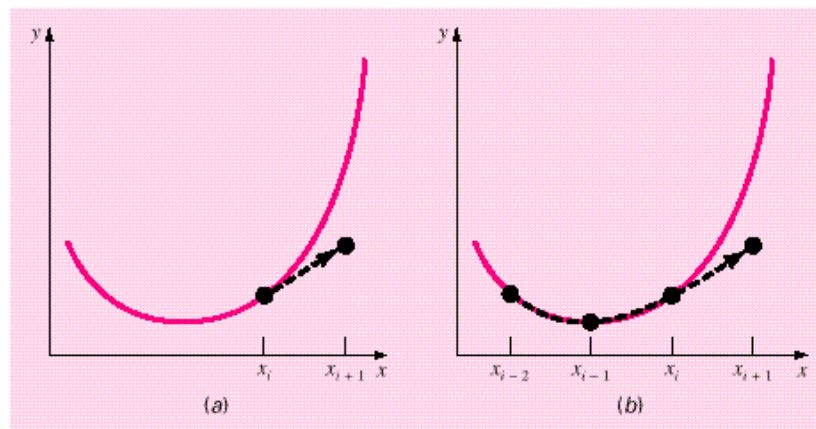
We can obtain a more accurate solution by using the `odeset` function to set the relative error tolerance to 10^{-4} :

```
>> options=odeset('RelTol',1e-4);
>> ode23(@dydt, [0, 4], 0.5, options);
```

As in Fig. 19.4b, the solver takes more small steps to attain the increased accuracy.

19.2 MULTISTEP METHODS

The one-step methods described in the previous sections utilize information at a single point t_i to predict a value of the dependent variable y_{i+1} at a future point t_{i+1} (Fig. 19.5a). Alternative approaches, called *multistep methods* (Fig. 19.5b), are based on the insight that, once the computation has begun, valuable information from previous points is at our


FIGURE 19.5

Graphical depiction of the fundamental difference between [·] one-step and [·] multistep methods for solving ODEs.

command. The curvature of the lines connecting these previous values provides information regarding the trajectory of the solution. Multistep methods exploit this information to solve ODEs. In this section, we will present a simple second-order method that serves to demonstrate the general characteristics of multistep approaches.

19.2.1 The Non-Self-Starting Heun Method

Recall that the Heun approach uses Euler's method as a predictor [Eq. (18.13)]:

$$y_{i+1}^0 = y_i + f(t_i, y_i)h \quad (19.4)$$

and the trapezoidal rule as a corrector [Eq. (18.16)]:

$$y_{i+1} = y_i + \frac{f(t_i, y_i) + f(t_{i+1}, y_{i+1}^0)}{2}h \quad (19.5)$$

Thus, the predictor and the corrector have local truncation errors of $O(h^2)$ and $O(h^3)$, respectively. This suggests that the predictor is the weak link in the method because it has the greatest error. This weakness is significant because the efficiency of the iterative corrector step depends on the accuracy of the initial prediction. Consequently, one way to improve Heun's method is to develop a predictor that has a local error of $O(h^3)$. This can be accomplished by using Euler's method and the slope at y_i , and extra information from a previous point y_{i-1} , as in

$$y_{i+1}^0 = y_{i-1} + f(t_i, y_i)2h \quad (19.6)$$

This formula attains $O(h^3)$ at the expense of employing a larger step size $2h$. In addition, note that the equation is not self-starting because it involves a previous value of the dependent variable y_{i-1} . Such a value would not be available in a typical initial-value problem. Because of this fact, Eqs. (19.5) and (19.6) are called the *non-self-starting Heun method*.

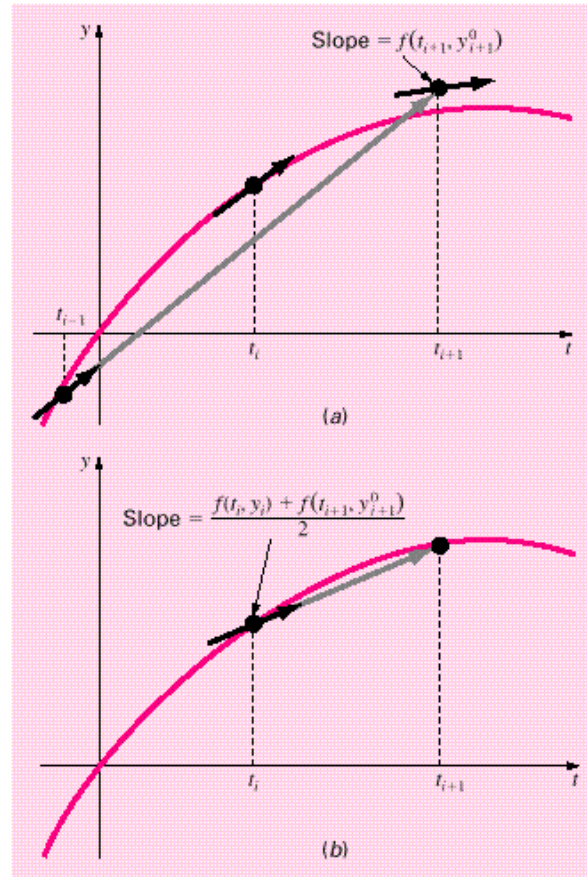


FIGURE 19.6

A graphical depiction of the non-self-starting Heun method. (a) The midpoint method that is used as a predictor. (b) The trapezoidal rule that is employed as a corrector.

As depicted in Fig. 19.6, the derivative estimate in Eq. (19.6) is now located at the midpoint rather than at the beginning of the interval over which the prediction is made. This centering improves the local error of the predictor to $O(h^3)$.

The non-self-starting Heun method can be summarized as

$$\text{Predictor (Fig. 19.6a): } y_{i+1}^0 = y_i^m + f(t_i, y_i^m)2h \quad (19.7)$$

$$\text{Corrector (Fig. 19.6b): } y_{i+1}^j = y_i^m + \frac{f(t_i, y_i^m) + f(t_{i+1}, y_{i+1}^{j-1})}{2}h \quad (19.8)$$

(for $j = 1, 2, \dots, m$)

where the superscripts denote that the corrector is applied iteratively from $j = 1$ to m to obtain refined solutions. Note that y_i^m and y_{i+1}^m are the final results of the corrector

iterations at the previous time steps. The iterations are terminated based on an estimate of the approximate error,

$$\varepsilon_a = \left| \frac{y_{i-1}^j - y_{i-1}^{j-1}}{y_{i-1}^j} \right| \cdot 100\% \quad (19.9)$$

When ε_a is less than a prespecified error tolerance ε_s , the iterations are terminated. At this point, $j = m$. The use of Eqs. (19.7) through (19.9) to solve an ODE is demonstrated in the following example.

EXAMPLE 19.3 Non-Self-Starting Heun's Method

Problem Statement. Use the non-self-starting Heun method to perform the same computations as were performed previously in Example 18.2 using Heun's method. That is, integrate $y' = 4e^{0.8t} + 0.5y$ from $t = 0$ to 4 with a step size of 1. As with Example 18.2, the initial condition at $t = 0$ is $y = 2$. However, because we are now dealing with a multistep method, we require the additional information that y is equal to -0.3929953 at $t = -1$.

Solution. The predictor [Eq. (19.7)] is used to extrapolate linearly from $t = -1$ to 1:

$$y_1^0 = -0.3929953 + [4e^{0.8(0)} + 0.5(2)]2 = 5.607005$$

The corrector [Eq. (19.8)] is then used to compute the value:

$$y_1^1 = 2 + \frac{4e^{0.8(0)} + 0.5(2) - 4e^{0.8(1)} - 0.5(5.607005)}{2}1 = 6.549331$$

which represents a true percent relative error of -5.73% (true value = 6.194631). This error is somewhat smaller than the value of -8.18% incurred in the self-starting Heun.

Now, Eq. (19.8) can be applied iteratively to improve the solution:

$$y_1^2 = 2 + \frac{3 - 4e^{0.8(1)} - 0.5(6.549331)}{2}1 = 6.313749$$

which represents an error of -1.92% . An approximate estimate of the error can be determined using Eq. (19.9):

$$\varepsilon_a = \left| \frac{6.313749 - 6.549331}{6.313749} \right| \cdot 100\% = 3.7\%$$

Equation (19.8) can be applied iteratively until ε_a falls below a prespecified value of ε_s . As was the case with the Heun method (recall Example 18.2), the iterations converge on a value of 6.36087 ($\varepsilon_t = -2.68\%$). However, because the initial predictor value is more accurate, the multistep method converges at a somewhat faster rate.

For the second step, the predictor is

$$y_2^0 = 2 + [4e^{0.8(1)} + 0.5(6.36087)]2 = 13.44346 \quad \varepsilon_t = 9.43\%$$

which is superior to the prediction of 12.0826 ($\varepsilon_t = 18\%$) that was computed with the original Heun method. The first corrector yields 15.76693 ($\varepsilon_t = 6.8\%$), and subsequent iterations converge on the same result as was obtained with the self-starting Heun method: 15.30224 ($\varepsilon_t = -3.09\%$). As with the previous step, the rate of convergence of the corrector is somewhat improved because of the better initial prediction.

19.2.2 Error Estimates

Aside from providing increased efficiency, the non-self-starting Heun can also be used to estimate the local truncation error. As with the adaptive RK methods in Section 19.1, the error estimate then provides a criterion for changing the step size.

The error estimate can be derived by recognizing that the predictor is equivalent to the midpoint rule. Hence, its local truncation error is (Table 16.4)

$$E_p \approx \frac{1}{3}h^3 y^{(3)}(\xi_p) \approx \frac{1}{3}h^3 f''(\xi_p) \quad (19.10)$$

where the subscript p designates that this is the error of the predictor. This error estimate can be combined with the estimate of y_{i-1} from the predictor step to yield

$$\text{True value} \approx y_{i-1}^0 + \frac{1}{3}h^3 y^{(3)}(\xi_p) \quad (19.11)$$

By recognizing that the corrector is equivalent to the trapezoidal rule, a similar estimate of the local truncation error for the corrector is (Table 16.2)

$$E_c \approx \frac{1}{12}h^3 y^{(3)}(\xi_c) \approx \frac{1}{12}h^3 f''(\xi_c) \quad (19.12)$$

This error estimate can be combined with the corrector result y_{i-1} to give

$$\text{True value} \approx y_{i-1}^m + \frac{1}{12}h^3 y^{(3)}(\xi_c) \quad (19.13)$$

Equation (19.11) can be subtracted from Eq. (19.13) to yield

$$0 \approx y_{i-1}^m - y_{i-1}^0 + \frac{5}{12}h^3 y^{(3)}(\xi) \quad (19.14)$$

where ξ is now between t_{i-1} and t_i . Now, dividing Eq. (19.14) by 5 and rearranging the result gives

$$\frac{y_{i-1}^0 - y_{i-1}^m}{5} \approx \frac{1}{12}h^3 y^{(3)}(\xi) \quad (19.15)$$

Notice that the right-hand sides of Eqs. (19.12) and (19.15) are identical, with the exception of the argument of the third derivative. If the third derivative does not vary appreciably over the interval in question, we can assume that the right-hand sides are equal, and therefore, the left-hand sides should also be equivalent, as in

$$E_c \approx \frac{y_{i-1}^0 - y_{i-1}^m}{5} \quad (19.16)$$

Thus, we have arrived at a relationship that can be used to estimate the per-step truncation error on the basis of two quantities that are routine by-products of the computation: the predictor (y_{i-1}^0) and the corrector (y_{i-1}^m).

EXAMPLE 19.4 Estimate of Per-Step Truncation Error

Problem Statement. Use Eq. (19.16) to estimate the per-step truncation error of Example 19.3. Note that the true values at $t = 1$ and 2 are 6.194631 and 14.84392, respectively.

Solution. At $t_{i-1} = 1$, the predictor gives 5.607005 and the corrector yields 6.360865. These values can be substituted into Eq. (19.16) to give

$$E_c = \frac{6.360865 - 5.607005}{5} = 0.150722$$

which compares well with the exact error,

$$E_t = 6.194631 - 6.360865 = -0.1662341$$

At $t_{i-1} = 2$, the predictor gives 13.44346 and the corrector yields 15.30224, which can be used to compute

$$E_c = \frac{15.30224 - 13.44346}{5} = 0.37176$$

which also compares favorably with the exact error, $E_t = 14.84392 - 15.30224 = -0.45831$.

The foregoing has been a brief introduction to multistep methods. Additional information can be found elsewhere (e.g., Chapra and Canale, 2002). Although they still have their place for solving certain types of problems, multistep methods are usually not the method of choice for most problems routinely confronted in engineering and science. That said, they are still used. For example, the MATLAB function `ode113` is a multistep method. We have therefore included this section to introduce you to their basic principles.

19.3 STIFFNESS

Stiffness is a special problem that can arise in the solution of ordinary differential equations. A *stiff system* is one involving rapidly changing components together with slowly changing ones. In some cases, the rapidly varying components are ephemeral transients that die away quickly, after which the solution becomes dominated by the slowly varying components. Although the transient phenomena exist for only a short part of the integration interval, they can dictate the time step for the entire solution.

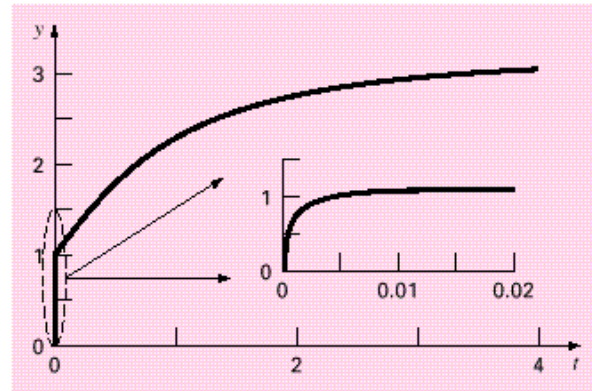
Both individual and systems of ODEs can be stiff. An example of a single stiff ODE is

$$\frac{dy}{dt} = -1000y + 3000 + 2000e^{-t} \quad (19.17)$$

If $y(0) = 0$, the analytical solution can be developed as

$$y = 3 + 0.998e^{-1000t} - 2.002e^{-t} \quad (19.18)$$

As in Fig. 19.7, the solution is initially dominated by the fast exponential term (e^{-1000t}). After a short period ($t < 0.005$), this transient dies out and the solution becomes governed by the slow exponential (e^{-t}).


FIGURE 19.7

Plot of a stiff solution of a single ODE. Although the solution appears to start at 1, there is actually a fast transient from $y = 0$ to 1 that occurs in less than 0.005 time unit. This transient is perceptible only when the response is viewed on the finer timescale in the inset.

Insight into the step size required for stability of such a solution can be gained by examining the homogeneous part of Eq. (19.17):

$$\frac{dy}{dt} = -ay \quad (19.19)$$

If $y(0) = y_0$, calculus can be used to determine the solution as

$$y = y_0 e^{-at}$$

Thus, the solution starts at y_0 and asymptotically approaches zero.

Euler's method can be used to solve the same problem numerically:

$$y_{i+1} = y_i + \frac{dy_i}{dt} h$$

Substituting Eq. (19.19) gives

$$y_{i+1} = y_i + ay_i h$$

or

$$y_{i+1} = y_i (1 + ah) \quad (19.20)$$

The stability of this formula clearly depends on the step size h . That is, $|1 + ah|$ must be less than 1. Thus, if $h > 2/a$, y_i will grow as $i \rightarrow \infty$.

For the fast transient part of Eq. (19.18), this criterion can be used to show that the step size to maintain stability must be $< 2/1000 = 0.002$. In addition, we should note that, whereas this criterion maintains stability (i.e., a bounded solution), an even smaller step size would be required to obtain an accurate solution. Thus, although the transient occurs for only a small fraction of the integration interval, it controls the maximum allowable step size.

Rather than using explicit approaches, implicit methods offer an alternative remedy. Such representations are called *implicit* because the unknown appears on both sides of the

equation. An implicit form of Euler's method can be developed by evaluating the derivative at the future time:

$$y_{i+1} = y_i + \frac{dy_{i+1}}{dt}h$$

This is called the *backward*, or *implicit*, Euler's method. Substituting Eq. (19.19) yields

$$y_{i+1} = y_i + ay_{i+1}h$$

which can be solved for

$$y_{i+1} = \frac{y_i}{1 - ah} \quad (19.21)$$

For this case, regardless of the size of the step, $|y_{i+1}| \rightarrow 0$ as $i \rightarrow \infty$. Hence, the approach is called *unconditionally stable*.

EXAMPLE 19.5 Explicit and Implicit Euler

Problem Statement. Use both the explicit and implicit Euler methods to solve Eq. (19.17), where $y(0) = 0$. **(a)** Use the explicit Euler with step sizes of 0.0005 and 0.0015 to solve for y between $t = 0$ and 0.006. **(b)** Use the implicit Euler with a step size of 0.05 to solve for y between 0 and 0.4.

Solution. **(a)** For this problem, the explicit Euler's method is

$$y_{i+1} = y_i + (-1000y_i + 3000 + 2000e^{-t_i})h$$

The result for $h = 0.0005$ is displayed in Fig. 19.8a along with the analytical solution. Although it exhibits some truncation error, the result captures the general shape of the analytical solution. In contrast, when the step size is increased to a value just below the stability limit ($h = 0.0015$), the solution manifests oscillations. Using $h > 0.002$ would result in a totally unstable solution—that is, it would go infinite as the solution progressed.

(b) The implicit Euler's method is

$$y_{i+1} = y_i + (-1000y_{i+1} + 3000 + 2000e^{-t_{i+1}})h$$

Now because the ODE is linear, we can rearrange this equation so that y_{i+1} is isolated on the left-hand side:

$$y_{i+1} = \frac{y_i + 3000h + 2000he^{-t_{i+1}}}{1 + 1000h}$$

The result for $h = 0.05$ is displayed in Fig. 19.8b along with the analytical solution. Notice that even though we have used a much bigger step size than the one that induced instability for the explicit Euler, the numerical result tracks nicely on the analytical solution.

Systems of ODEs can also be stiff. An example is

$$\frac{dy_1}{dt} = -5y_1 + 3y_2 \quad (19.22a)$$

$$\frac{dy_2}{dt} = 100y_1 + 301y_2 \quad (19.22b)$$

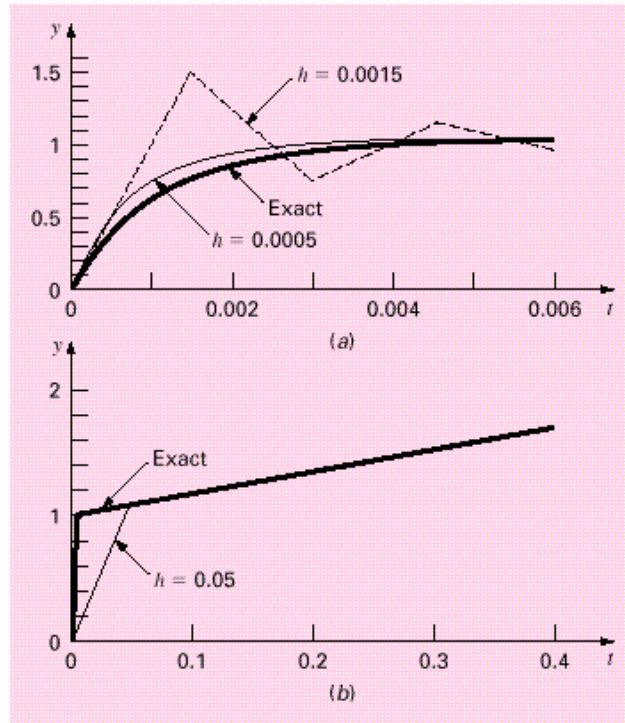


FIGURE 19.8
Solution of a stiff ODE with (—) the explicit and (---) implicit Euler methods.

For the initial conditions $y_1(0) = 52.29$ and $y_2(0) = 83.82$, the exact solution is

$$y_1 = 52.96e^{-3.9899t} + 0.67e^{-302.0101t} \quad (19.23a)$$

$$y_2 = 17.83e^{-3.9899t} + 65.99e^{-302.0101t} \quad (19.23b)$$

Note that the exponents are negative and differ by about two orders of magnitude. As with the single equation, it is the large exponents that respond rapidly and are at the heart of the system's stiffness.

An implicit Euler's method for systems can be formulated for the present example as

$$y_{1,i+1} = y_{1,i} + (-5y_{1,i+1} - 3y_{2,i+1})h \quad (19.24a)$$

$$y_{2,i+1} = y_{2,i} + (100y_{1,i+1} - 301y_{2,i+1})h \quad (19.24b)$$

Collecting terms gives

$$(1 - 5h)y_{1,i+1} - 3y_{2,i+1} = y_{1,i} \quad (19.25a)$$

$$100y_{1,i+1} - (1 - 301h)y_{2,i+1} = y_{2,i} \quad (19.25b)$$

Thus, we can see that the problem consists of solving a set of simultaneous equations for each time step.

For nonlinear ODEs, the solution becomes even more difficult since it involves solving a system of nonlinear simultaneous equations (recall Section 11.2). Thus, although stability is gained through implicit approaches, a price is paid in the form of added solution complexity.

19.3.1 MATLAB Functions for Stiff Systems

MATLAB has a number of built-in functions for solving stiff systems of ODEs. These are

..... This function is a variable-order solver based on numerical differentiation formulas. It is a multistep solver that optionally uses the Gear backward differentiation formulas. This is used for stiff problems of low to medium accuracy.

..... This function is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems better than `ode15s`.

..... This function is an implementation of the trapezoidal rule with a “free” interpolant. This is used for moderately stiff problems with low accuracy where you need a solution without numerical damping.

..... This is an implementation of an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule and a second stage that is a backward differentiation formula of order 2. This solver may also be more efficient than `ode15s` at crude tolerances.

EXAMPLE 19.6 MATLAB for Stiff ODEs

Problem Statement. The van der Pol equation is a model of an electronic circuit that arose back in the days of vacuum tubes,

$$\frac{d^2 y_1}{dt^2} + \mu(1 - y_1^2) \frac{dy_1}{dt} - y_1 = 0 \quad (\text{E19.6.1})$$

The solution to this equation becomes progressively stiffer as μ gets large. Given the initial conditions, $y_1(0) = dy_1/dt = 1$, use MATLAB to solve the following two cases: (a) for $\mu = 1$, use `ode45` to solve from $t = 0$ to 20; and (b) for $\mu = 1000$, use `ode23s` to solve from $t = 0$ to 6000.

Solution. (a) The first step is to convert the second-order ODE into a pair of first-order ODEs by defining

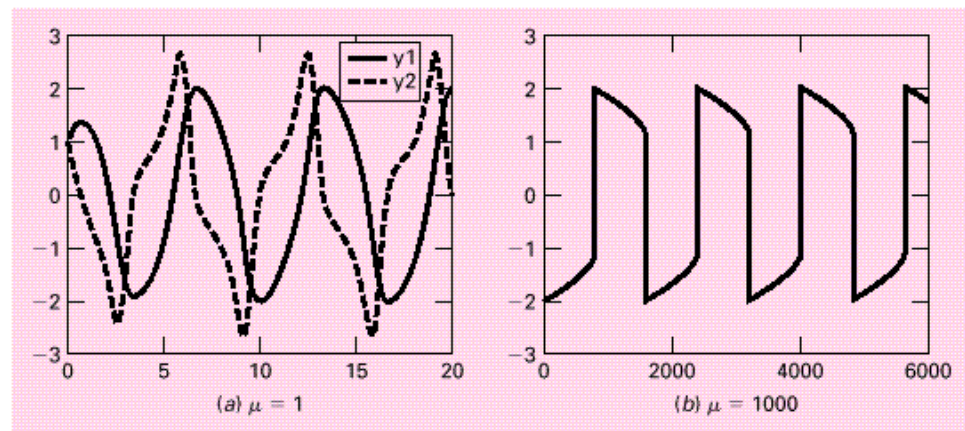
$$\frac{dy_1}{dt} = y_2$$

Using this equation, Eq. (E19.6.1) can be written as

$$\frac{dy_2}{dt} = \mu(1 - y_1^2)y_2 - y_1 = 0$$

An M-file can now be created to hold this pair of differential equations:

```
function yp = vanderpol(t,y,mu)
yp = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
```

**FIGURE 19.9**

Solutions for van der Pol's equation. (·) Nonstiff form solved with `ode45` and (---) stiff form solved with `ode23s`.

Notice how the value of μ is passed as a parameter. As in Example 19.1, `ode45` can be invoked and the results plotted:

```
>> [t,y] = ode45(@vanderpol,[0 20],[1 1],[],1);
>> plot(t,y(:,1),'-',t,y(:,2),'--')
>> legend('y1','y2');
```

Observe that because we are not specifying any options, we must use open brackets `[]` as a place holder. The smooth nature of the plot (Fig. 19.9a) suggests that the van der Pol equation with $\mu = 1$ is not a stiff system.

(b) If a standard solver like `ode45` is used for the stiff case ($\mu = 1000$), it will fail miserably (try it, if you like). However, `ode23s` does an efficient job:

```
>> [t,y] = ode23s(@vanderpol,[0 6000],[1 1],[],1000);
>> plot(t,y(:,1))
```

We have only displayed the y_1 component because the result for y_2 has a much larger scale. Notice how this solution (Fig. 19.9b) has much sharper edges than is the case in Fig. 19.9a. This is a visual manifestation of the “stiffness” of the solution.

19.4 MATLAB APPLICATION: BUNGEE JUMPER WITH CORD

In this section, we will use MATLAB to solve for the vertical dynamics of a jumper connected to a stationary platform with a bungee cord. As developed at the beginning of Chap. 18, the problem consisted of solving two coupled ODEs for vertical position and velocity. The differential equation for position is

$$\frac{dx}{dt} = v \quad (19.26)$$

The differential equation for velocity is different depending on whether the jumper has fallen to a distance where the cord is fully extended and begins to stretch. Thus, if the distance fallen is less than the cord length, the jumper is only subject to gravitational and drag forces,

$$\frac{dv}{dt} = g - \text{sign}(v) \frac{c_d}{m} v^2 \quad (19.27a)$$

Once the cord begins to stretch, the spring and dampening forces of the cord must also be included:

$$\frac{dv}{dt} = g - \text{sign}(v) \frac{c_d}{m} v^2 - \frac{k}{m} (x - L) - \frac{\gamma}{m} v \quad (19.27b)$$

The following example shows how MATLAB can be used to solve this problem.

EXAMPLE 19.7 MATLAB for Stiff ODEs

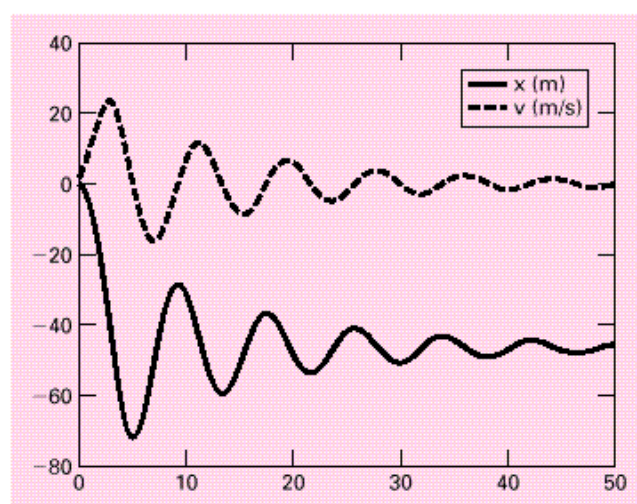
Problem Statement. Determine the position and velocity of a bungee jumper with the following parameters: $L = 30$ m, $g = 9.81$ m/s², $m = 68.1$ kg, $c_d = 0.25$ kg/m, $k = 40$ N/m, and $\gamma = 8$ N·s/m. Perform the computation from $t = 0$ to 50 s and assume that the initial conditions are $x(0) = v(0) = 0$.

Solution. The following M-file can be set up to compute the right-hand sides of the ODEs:

```
function dydt = bungee(t,y,L,cd,m,k,gamma)
g = 9.81;
cord = 0;
if y(1) > L %determine if the cord exerts a force
    cord = k/m*(y(1)-L)+gamma/m*y(2);
end
dydt = [y(2); g - sign(y(2))*cd/m*y(2)^2 - cord];
```

FIGURE 19.10

Plot of distance and velocity of a bungee jumper.



PROBLEMS

363

Notice that the derivatives are returned as a column vector because this is the format required by the MATLAB solvers.

Because these equations are not stiff, we can use `ode45` to obtain the solutions and display them on a plot:

```
>> [t,y] = ode45(@bungee,[0 50],[0 0],[1,30,0.25,68.1,40,8]);
>> plot(t,-y(:,1),'- ',t,y(:,2),' : ');
>> legend('x (m)', 'v (m/s)')
```

As in Fig. 19.10, we have reversed the sign of distance for the plot so that negative distance is in the downward direction. Notice how the simulation captures the jumper's bouncing motion.

PROBLEMS

19.1 Predator-prey models were developed independently in the early part of the twentieth century by the Italian mathematician Vito Volterra and the American biologist Alfred Lotka. These equations are commonly called *Lotka-Volterra equations*. The simplest example is the following pairs of ODEs:

$$\begin{aligned}\frac{dx}{dt} &= ax - bxy \\ \frac{dy}{dt} &= cy - dxy\end{aligned}$$

where x and y = the number of prey and predators, respectively, a = the prey growth rate, c = the predator death rate, and b and d = the rates characterizing the effect of the predator-prey interactions on the prey death and the predator growth, respectively. Given the parameter values: $a = 1.5$, $b = 0.7$, $c = 0.9$, and $d = 0.4$, integrate these equations from $t = 0$ to 30 given the initial conditions $x = 2$ and $y = 1$. Develop plots of x and y versus t and y versus x . Obtain your solutions with (a) Euler's method with a step-size of 0.1, (b) the fourth-order RK method with a step-size of 0.1, and (c) the `ode45` function.

19.2 An example of an interesting nonlinear model based on atmospheric fluid dynamics is the *Lorenz equations* developed by the American meteorologist Edward Lorenz:

$$\begin{aligned}\frac{dx}{dt} &= \sigma x - \sigma y \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dz}{dt} &= bz - xy\end{aligned}$$

Lorenz developed these equations to relate the intensity of atmospheric fluid motion x to temperature variations y and z in the horizontal and vertical directions, respectively. Given the parameter values: $\sigma = 10$, $b = 2.666667$, and $r = 28$, integrate these equations from $t = 0$ to 20 given the initial conditions $x = y = z = 5$. Develop plots of x , y , and z versus t , y versus x , z versus x and z versus y . Obtain your solutions with (a) the fourth-order RK method with a step-size of 0.1, (b) the `ode23` function, (c) the `ode45` function, and (d) the `ode23tb` function.

19.3 Solve the following initial-value problem over the interval from $t = 2$ to 3:

$$\frac{dy}{dt} = 0.5y \cdot e^{-t}$$

Use the non-self-starting Heun method with a step size of 0.5 and initial conditions of $y(1.5) = 5.222138$ and $y(2.0) = 4.143883$. Iterate the corrector to $\epsilon_s = 0.1\%$. Compute the percent relative errors for your results based on the exact solutions obtained analytically: $y(2.5) = 3.273888$ and $y(3.0) = 2.577988$.

19.4 Solve the following initial-value problem over the interval from $t = 0$ to 0.5:

$$\frac{dy}{dt} = yr^2 \cdot y$$

Use the fourth-order RK method to predict the first value at $t = 0.25$. Then use the non-self-starting Heun method to make the prediction at $t = 0.5$. Note: $y(0) = 1$.

19.5 Given

$$\frac{dy}{dt} = 100,000y \cdot 99,999e^{-t}$$

- (a) Estimate the step size required to maintain stability using the explicit Euler method.
- (b) If $y(0) = 0$, use the implicit Euler to obtain a solution from $t = 0$ to 2 using a step size of 0.1.

19.6 Given

$$\frac{dy}{dt} = 30(\sin t - y) - 3 \cos t$$

If $y(0) = 0$, use the implicit Euler to obtain a solution from $t = 0$ to 4 using a step size of 0.4.

19.7 Given

$$\frac{dx_1}{dt} = 999x_1 - 1999x_2$$

$$\frac{dx_2}{dt} = 1000x_1 - 2000x_2$$

If $x_1(0) = x_2(0) = 1$, obtain a solution from $t = 0$ to 0.2 using a step size of 0.05 with the (a) explicit and (b) implicit Euler methods.

19.8 The following nonlinear, parasitic ODE was suggested by Hornbeck (1975):

$$\frac{dy}{dt} = 5(y - t^2)$$

If the initial condition is $y(0) = 0.08$, obtain a solution from $t = 0$ to 5:

- (a) Analytically.
- (b) Using the fourth-order RK method with a constant step size of 0.03125.
- (c) Using the MATLAB function `ode45`.
- (d) Using the MATLAB function `ode23s`.
- (e) Using the MATLAB function `ode23tb`.

Present your results in graphical form.

19.9 Recall from Example 17.5 that the following humps function exhibits both flat and steep regions over a relatively short x range,

$$f(x) = \frac{1}{(x - 0.3)^2 + 0.01} + \frac{1}{(x - 0.9)^2 + 0.04} - 6$$

Determine the value of the definite integral of this function between $x = 0$ and 1 using (a) the `quad` function, and

(b) `ode45`.

19.10 The oscillations of a swinging pendulum can be simulated with the following nonlinear model:

$$\frac{d^2\theta}{dt^2} + \frac{g}{l} \sin \theta = 0$$

where θ = the angle of displacement, g = the gravitational constant, and l = the pendulum length. For small angular displacements, the $\sin \theta$ is approximately equal to θ and the model can be linearized as

$$\frac{d^2\theta}{dt^2} + \frac{g}{l} \theta = 0$$

Use `ode45` to solve for θ as a function of time for both the linear and nonlinear models where $l = 0.6$ m and $g = 9.81$ m/s². First, solve for the case where the initial condition is for a small displacement ($\theta = \pi/8$ and $d\theta/dt = 0$). Then repeat the calculation for a large displacement ($\theta = \pi/2$ and $d\theta/dt = 0$). For each case, plot the linear and nonlinear simulations on the same plot.

19.11 The logistic model is used to simulate population as in

$$\frac{dp}{dt} = k_{gm}(1 - p/p_{\max})p$$

where p = population, k_{gm} = the maximum growth rate under unlimited conditions, and p_{\max} = the carrying capacity. Simulate the world's population from 1950 to 2000 using the `ode45` function. Employ the following initial conditions and parameter values for your simulation: p_0 (in 1950) = 2,555 million people, $k_{gm} = 0.026$ /yr, and $p_{\max} = 12,000$ million people. Have the function generate output corresponding to the dates for the following measured population data. Use the results to compute the sum of the squares of the residuals between the data and the simulation output.

· 1950	1955	1960	1965	1970	1975
· 2555	2780	3040	3346	3708	4087
· 1980	1985	1990	1995	2000	
· 4454	4850	5276	5686	6079	

Eigenvalues

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to methods for solving eigenvalue problems. Specific objectives and topics covered are

- Understanding how eigenvalues provide the solution for a special form of homogeneous linear algebraic equations.
- Understanding how eigenvalue problems arise in engineering and scientific problems dealing with vibrating and oscillating systems.
- Knowing how to determine and interpret eigenvectors.
- Knowing how to implement the polynomial method with MATLAB.
- Knowing how to implement the power method to determine either the highest or lowest eigenvalue along with its accompanying eigenvector.
- Knowing how to determine eigenvalues and eigenvectors with the MATLAB `eig` function.

Eigenvalue, or characteristic-value, problems are a special class of problems that are common in engineering and scientific problem contexts involving vibrations and elasticity. In addition, they are used in a wide variety of other areas including the solution of linear differential equations and statistics.

Before describing numerical methods for solving such problems, we will present some general background information. This includes discussion of both the mathematics and the engineering and scientific significance of eigenvalues.

20.1 MATHEMATICAL BACKGROUND

Chapters 7 through 11 dealt with methods for solving sets of linear algebraic equations of the general form

$$[A] \cdot x = b$$

Such systems are called *nonhomogeneous* because of the presence of the vector $\{b\}$ on the right-hand side of the equality. If the equations comprising such a system are linearly independent (i.e., have a nonzero determinant), they will have a unique solution. In other words, there is one set of x values that will make the equations balance.

In contrast, a *homogeneous* linear algebraic system has the general form

$$[A] \cdot x = 0$$

Although nontrivial solutions (i.e., solutions other than all $x's = 0$) of such systems are possible, they are generally not unique. Rather, the simultaneous equations establish relationships among the x 's that can be satisfied by various combinations of values.

Eigenvalue problems associated with engineering are typically of the general form

$$\begin{aligned} (a_{11} - \lambda)x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= 0 \\ a_{21}x_1 + (a_{22} - \lambda)x_2 + \cdots + a_{2n}x_n &= 0 \\ \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + (a_{nn} - \lambda)x_n &= 0 \end{aligned}$$

where λ is an unknown parameter called the *eigenvalue*, or *characteristic value*. A solution $\{x\}$ for such a system is referred to as an *eigenvector*. The above set of equations may also be expressed concisely as

$$([A] - \lambda[I]) \cdot x = 0 \quad (20.1)$$

The solution of Eq. (20.1) hinges on determining λ . One way to accomplish this is based on the fact that the determinant of the matrix $[A] - \lambda[I]$ must equal zero for nontrivial solutions to be possible. Expanding the determinant yields a polynomial in λ , which is called the *characteristic polynomial*. The roots of this polynomial are the solutions for the eigenvalues. An example of this approach, called the *polynomial method*, will be provided in Section 20.3. Before describing the method, we will first describe how eigenvalues arise in engineering and science.

20.2 PHYSICAL BACKGROUND

The mass-spring system in Fig. 20.1a is a simple context to illustrate how eigenvalues occur in physical problem settings. It also will help to illustrate some of the mathematical concepts introduced in Section 20.1.

To simplify the analysis, assume that each mass has no external or damping forces acting on it. In addition, assume that each spring has the same natural length l and the same spring constant k . Finally, assume that the displacement of each spring is measured relative to its own local coordinate system with an origin at the spring's equilibrium position (Fig. 20.1a). Under these assumptions, Newton's second law can be employed to develop a force balance for each mass:

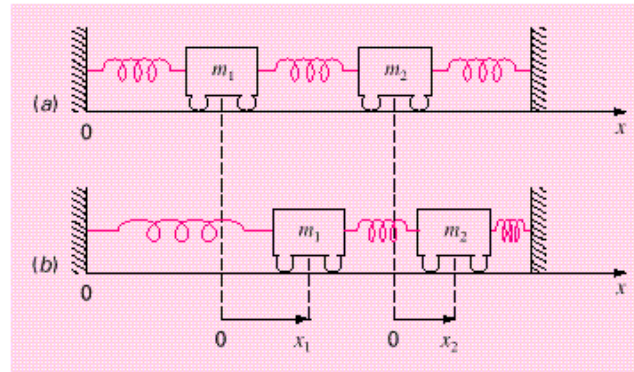
$$m_1 \frac{d^2 x_1}{dt^2} = -kx_1 - k(x_2 - x_1)$$

and

$$m_2 \frac{d^2 x_2}{dt^2} = k(x_2 - x_1) - kx_2$$

20.2 PHYSICAL BACKGROUND

367

**FIGURE 20.1**

Positioning the masses away from equilibrium creates forces in the springs that on release lead to oscillations of the masses. The positions of the masses can be referenced to local coordinates with origins at their respective equilibrium positions.

where x_i is the displacement of mass i away from its equilibrium position (Fig. 20.1b). By collecting terms, these equations can be expressed as

$$m_1 \frac{d^2 x_1}{dt^2} + k(-2x_1 + x_2) = 0 \quad (20.2a)$$

$$m_2 \frac{d^2 x_2}{dt^2} + k(x_1 - 2x_2) = 0 \quad (20.2b)$$

From vibration theory, it is known that solutions to Eq. (20.2) can take the form

$$x_i = X_i \sin(\omega t) \quad (20.3)$$

where X_i = the amplitude of the vibration of mass i and ω = the frequency of the vibration, which is equal to

$$\omega = \frac{2\pi}{T_p} \quad (20.4)$$

where T_p is the period. From Eq. (20.3) it follows that

$$\ddot{x}_i = -X_i \omega^2 \sin(\omega t) \quad (20.5)$$

Equations (20.3) and (20.5) can be substituted into Eq. (20.2), which, after collection of terms, can be expressed as

$$\left(\frac{2k}{m_1} - \omega^2 \right) X_1 - \frac{k}{m_1} X_2 = 0 \quad (20.6a)$$

$$- \frac{k}{m_1} X_1 + \left(\frac{2k}{m_2} - \omega^2 \right) X_2 = 0 \quad (20.6b)$$

Comparison of Eq. (20.6) with Eq. (20.1) indicates that at this point, the solution has been reduced to an eigenvalue problem. That is, we can determine values of the eigenvalue

ω^2 that satisfy the equations. For a two-degree-of-freedom system such as Fig. 20.1, there will be two such values. Each of these eigenvalues establishes a unique relationship between the unknowns X called an *eigenvector*. Section 20.3 describes a simple approach to determine both the eigenvalues and eigenvectors. It also illustrates the physical significance of these quantities for the mass-spring system.

20.3 THE POLYNOMIAL METHOD

As stated at the end of Section 20.1, the *polynomial method* consists of expanding the determinant to generate the characteristic polynomial. The roots of this polynomial are the solutions for the eigenvalues. The following example illustrates how it can be used to determine both the eigenvalues and eigenvectors for the mass-spring system (Fig. 20.1).

EXAMPLE 20.1 The Polynomial Method

Problem Statement. Evaluate the eigenvalues and the eigenvectors of Eq. (20.6) for the case where $m_1 = m_2 = 40$ kg and $k = 200$ N/m.

Solution. Substituting the parameter values into Eq. (20.6) yields

$$\begin{aligned}(10 - \omega^2)X_1 - 5X_2 &= 0 \\ -5X_1 + (10 - \omega^2)X_2 &= 0\end{aligned}$$

The determinant of this system is

$$(\omega^2)^2 - 20\omega^2 + 75 = 0$$

which can be solved by the quadratic formula for $\omega^2 = 15$ and 5 s^{-2} . Therefore, the frequencies for the vibrations of the masses are $\omega = 3.873 \text{ s}^{-1}$ and 2.236 s^{-1} , respectively. These values can be used to determine the periods for the vibrations with Eq. (20.4). For the first mode, $T_p = 1.62$ s, and for the second, $T_p = 2.81$ s.

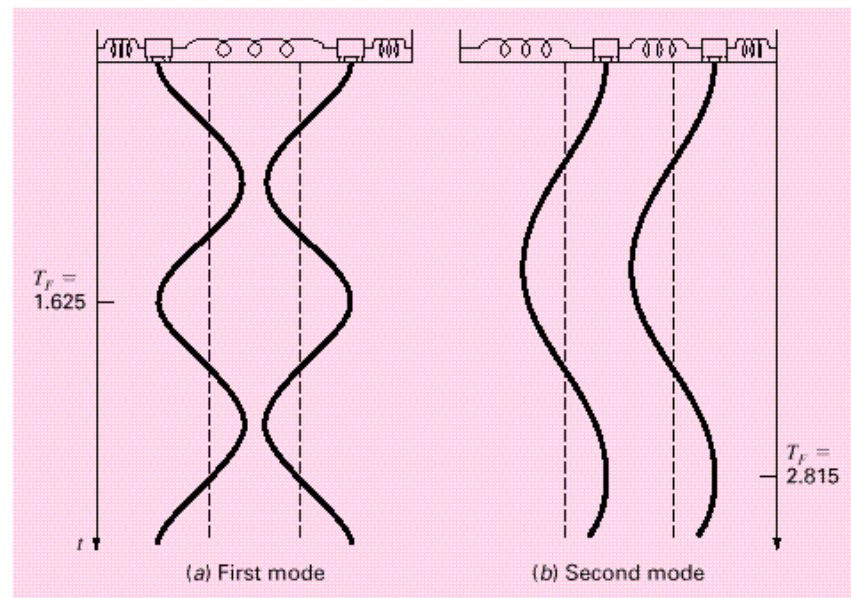
As stated in Section 20.1, a unique set of values cannot be obtained for the unknown amplitudes X . However, their ratios can be specified by substituting the eigenvalues back into the equations. For example, for the first mode ($\omega^2 = 15 \text{ s}^{-2}$):

$$\begin{aligned}(10 - 15)X_1 - 5X_2 &= 0 \\ -5X_1 + (10 - 15)X_2 &= 0\end{aligned}$$

Thus, we conclude that $X_1 = -X_2$. In a similar fashion for the second mode ($\omega^2 = 5 \text{ s}^{-2}$), $X_1 = X_2$. These relationships are the eigenvectors.

This example provides valuable information regarding the behavior of the system in Fig. 20.1. Aside from its period, we know that if the system is vibrating in the first mode, the eigenvector tells us that the amplitude of the second mass will be equal but of opposite sign to the amplitude of the first. As in Fig. 20.2a, the masses vibrate apart and then together indefinitely.

In the second mode, the eigenvector specifies that the two masses have equal amplitudes at all times. Thus, as in Fig. 20.2b, they vibrate back and forth in unison. We should note that the configuration of the amplitudes provides guidance on how to set their initial values to attain pure motion in either of the two modes. Any other configuration will lead to superposition of the modes.

**FIGURE 20.2**

The principal modes of vibration of two equal masses connected by three identical springs between fixed walls.

We should recognize that MATLAB has built-in functions to facilitate the polynomial method. For Example 20.1, the `poly` function can be used to generate the characteristic polynomial as in

```
>> A = [10 -5; -5 10];  
>> p = poly(A)  
  
p =  
    1   -20    75
```

Then, the `roots` function can be employed to compute the eigenvalues:

```
>> roots(p)  
  
ans =  
    15  
     5
```

20.4 THE POWER METHOD

The power method is an iterative approach that can be employed to determine the largest or *dominant eigenvalue*. With slight modification, it can also be employed to determine the smallest value. It has the additional benefit that the corresponding eigenvector is obtained as a by-product of the method.

To implement the power method, the system being analyzed is expressed in the form

$$[A] \cdot x = \lambda \cdot x \quad (20.7)$$

As illustrated by the following example, Eq. (20.7) forms the basis for an iterative solution technique that eventually yields the highest eigenvalue and its associated eigenvector.

EXAMPLE 20.2 Power Method for Highest Eigenvalue

Problem Statement. Using the same approach as in Section 20.2, we can derive the following homogeneous set of equations for a three mass–four spring system between two fixed walls:

$$\begin{aligned} \left(\frac{2k}{m_1} - \omega^2 \right) X_1 - \frac{k}{m_1} X_2 &= 0 \\ -\frac{k}{m_2} X_1 + \left(\frac{2k}{m_2} - \omega^2 \right) X_2 - \frac{k}{m_2} X_3 &= 0 \\ -\frac{k}{m_3} X_2 + \left(\frac{2k}{m_3} - \omega^2 \right) X_3 &= 0 \end{aligned}$$

If all the masses $m = 1$ kg and all the spring constants $k = 20$ N/m, the system can be expressed in the matrix format of Eq. (20.1) as

$$\begin{bmatrix} 40 & -20 & 0 \\ -20 & 40 & -20 \\ 0 & -20 & 40 \end{bmatrix} \cdot \lambda [I] = 0$$

where the eigenvalue λ is the square of the angular frequency ω^2 . Employ the power method to determine the highest eigenvalue and its associated eigenvector.

Solution. The system is first written in the form of Eq. (20.7):

$$\begin{aligned} 40X_1 - 20X_2 &= \lambda X_1 \\ -20X_1 + 40X_2 - 20X_3 &= \lambda X_2 \\ -20X_2 + 40X_3 &= \lambda X_3 \end{aligned}$$

At this point, we can make initial values of the X 's and use the left-hand side to compute an eigenvalue and eigenvector. A good first choice is to assume that all the X 's on the left-hand side of the equation are equal to one:

$$\begin{aligned} 40(1) - 20(1) &= 20 \\ -20(1) + 40(1) - 20(1) &= 0 \\ -20(1) + 40(1) &= 20 \end{aligned}$$

Next, the right-hand side is normalized by 20 to make the largest element equal to one:

$$\begin{Bmatrix} 20 \\ 0 \\ 20 \end{Bmatrix} = 20 \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix}$$

Thus, the normalization factor is our first estimate of the eigenvalue (20) and the corresponding eigenvector is $\begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T$. This iteration can be expressed concisely in matrix

20.4 THE POWER METHOD

371

form as

$$\begin{bmatrix} 40 & -20 & 0 \\ -20 & 40 & -20 \\ 0 & -20 & 40 \end{bmatrix} \begin{Bmatrix} 1 \\ 1 \\ 1 \end{Bmatrix} = \begin{Bmatrix} 20 \\ 0 \\ 20 \end{Bmatrix} = 20 \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix}$$

The next iteration consists of multiplying the matrix by $\begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T$ to give

$$\begin{bmatrix} 40 & -20 & 0 \\ -20 & 40 & -20 \\ 0 & -20 & 40 \end{bmatrix} \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix} = \begin{Bmatrix} 40 \\ -40 \\ 40 \end{Bmatrix} = 40 \begin{Bmatrix} 1 \\ -1 \\ 1 \end{Bmatrix}$$

Therefore, the eigenvalue estimate for the second iteration is 40, which can be employed to determine the error estimate:

$$\epsilon_a = \left| \frac{40 - 20}{40} \right| = 100\% = 50\%$$

The process can then be repeated.

Third iteration:

$$\begin{bmatrix} 40 & -20 & 0 \\ -20 & 40 & -20 \\ 0 & -20 & 40 \end{bmatrix} \begin{Bmatrix} 1 \\ -1 \\ 1 \end{Bmatrix} = \begin{Bmatrix} 60 \\ -80 \\ 60 \end{Bmatrix} = 80 \begin{Bmatrix} 0.75 \\ -1 \\ 0.75 \end{Bmatrix}$$

where $\epsilon_a = 150\%$ (which is high because of the sign change).

Fourth iteration:

$$\begin{bmatrix} 40 & -20 & 0 \\ -20 & 40 & -20 \\ 0 & -20 & 40 \end{bmatrix} \begin{Bmatrix} 0.75 \\ 1 \\ 0.75 \end{Bmatrix} = \begin{Bmatrix} 50 \\ 70 \\ 50 \end{Bmatrix} = 70 \begin{Bmatrix} 0.71429 \\ 1 \\ 0.71429 \end{Bmatrix}$$

where $\epsilon_a = 214\%$ (which is high because of the sign change).

Fifth iteration:

$$\begin{bmatrix} 40 & -20 & 0 \\ -20 & 40 & -20 \\ 0 & -20 & 40 \end{bmatrix} \begin{Bmatrix} 0.71429 \\ 1 \\ 0.71429 \end{Bmatrix} = \begin{Bmatrix} 48.5714 \\ 68.5714 \\ 48.5714 \end{Bmatrix} = 68.5714 \begin{Bmatrix} 0.70833 \\ 1 \\ 0.70833 \end{Bmatrix}$$

where $\epsilon_a = 2.08\%$.

Thus, the eigenvalue is converging. After several more iterations, it stabilizes on a value of 68.28427 with a corresponding eigenvector of $\begin{bmatrix} 0.707107 & 1 & 0.707107 \end{bmatrix}^T$.

Note that there are some instances where the power method will converge to the second-largest eigenvalue instead of to the largest. James, Smith, and Wolford (1985) provide an illustration of such a case. Other special cases are discussed in Fadeev and Fadeeva (1963).

In addition, there are sometimes cases where we are interested in determining the smallest eigenvalue. This can be done by applying the power method to the matrix inverse

of $[A]$. For this case, the power method will converge on the largest value of $1/\lambda$ —in other words, the smallest value of λ . An application to find the smallest eigenvalue will be left as a problem exercise.

Finally, after finding the largest eigenvalue, it is possible to determine the next highest by replacing the original matrix by one that includes only the remaining eigenvalues. The process of removing the largest known eigenvalue is called *deflation*.

We should mention that although the power method can be used to locate intermediate values, better methods are available for cases where we need to determine all the eigenvalues as described in Section 20.5. Thus, the power method is primarily used when we want to locate the largest or the smallest eigenvalue.

20.5 MATLAB FUNCTION: . . .

As might be expected, MATLAB has powerful and robust capabilities for evaluating eigenvalues and eigenvectors. The function `eig`, which is used for this purpose, can be used to generate a vector of the eigenvalues as in

```
>> v = eig(A)
```

where v is a vector containing the eigenvalues of a square matrix A . Alternatively, it can be invoked as

```
>> [D, V] = eig(A)
```

where D is a diagonal matrix of the eigenvalues and V is a full matrix whose columns are the corresponding eigenvectors.

EXAMPLE 20.3 Use of MATLAB to Determine Eigenvalues and Eigenvectors

Problem Statement. Use MATLAB to determine all the eigenvalues and eigenvectors for the system described in Example 20.2.

Solution. Recall that the matrix to be analyzed is

$$\begin{bmatrix} 40 & -20 & 0 \\ -20 & 40 & -20 \\ 0 & -20 & 40 \end{bmatrix}$$

The matrix can be entered as

```
>> A = [40 -20 0; -20 40 -20; 0 -20 40];
```

If we just desire the eigenvalues we can enter

```
>> e = eig(A)
```

```
e =  
11.7157  
40.0000  
68.2843
```

PROBLEMS

373

Notice that the highest eigenvalue (68.2843) is consistent with the value previously determined with the power method in Example 20.2.

If we want both the eigenvalues and eigenvectors, we can enter

```
>> [v,d] = eig(A)

v =
    0.5000    -0.7071   -0.5000
    0.7071    -0.0000    0.7071
    0.5000     0.7071   -0.5000

d =
   11.7157         0         0
         0   40.0000         0
         0         0   68.2843
```

Again, although the results are scaled differently, the eigenvector corresponding to the highest eigenvalue $\approx 0.5 \ 0.7071 \ -0.5^T$ is consistent with the value previously determined with the power method in Example 20.2 $\approx 0.707107 \ 1 \ -0.707107^T$.

PROBLEMS

20.1 Determine the smallest eigenvalue and the corresponding eigenvector for the system in Example 20.2.

20.2 Given the following system

$$\begin{bmatrix} 2 & \lambda & 2 & 10 \\ 8 & 3 & \lambda & 4 \\ 10 & 4 & 5 & \lambda \end{bmatrix}$$

- Use minors to expand the determinant.
- Evaluate the eigenvalues using the polynomial method.
- Use the power method to determine the highest eigenvalue.
- Use the power method to determine the lowest eigenvalue.

20.3 Use the polynomial method to determine the eigenvalues and the eigenvectors for the following system:

$$\begin{bmatrix} 4 & 7 & 3 \\ 7 & 8 & 2 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \lambda \begin{bmatrix} 9 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

20.4 Given the following system

$$\begin{bmatrix} 2 & 0.36p^2 & 1 & 0 & 0 \\ 1 & 2 & 0.36p^2 & 1 & 0 \\ 0 & 1 & 2 & 0.36p^2 & 1 \\ 0 & 0 & 1 & 2 & 0.36p^2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = 0$$

- Evaluate the eigenvalues using the polynomial method.
- Use the power method to determine the highest eigenvalue.
- Use the power method to determine the lowest eigenvalue.

20.5 Engineers and scientists use mass-spring models to gain insight into the dynamics of structures under the influence of disturbances such as earthquakes. Figure P20.5 shows such a representation for a three-story building.

For this case, the analysis is limited to horizontal motion of the structure. Using the same approach as developed in Section 20.1, force balances can be developed for this system as

$$\begin{aligned} \left(\frac{k_1 + k_2}{m_1} - \omega^2 \right) X_1 - \frac{k_2}{m_1} X_2 &= 0 \\ -\frac{k_2}{m_2} X_1 + \left(\frac{k_2 + k_3}{m_2} - \omega^2 \right) X_2 - \frac{k_3}{m_2} X_3 &= 0 \\ -\frac{k_3}{m_3} X_2 + \left(\frac{k_3}{m_3} - \omega^2 \right) X_3 &= 0 \end{aligned}$$

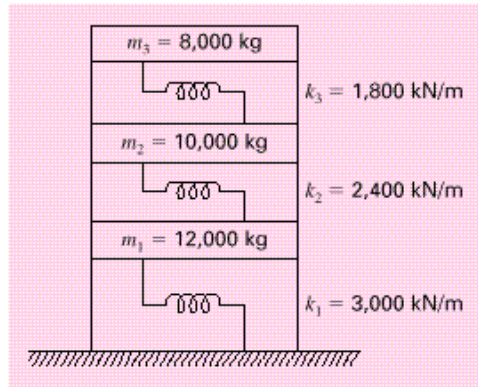


FIGURE P20.5

Determine the eigenvalues and eigenvectors for this system. Graphically represent the modes of vibration for the structure by displaying the amplitudes versus height for each of the eigenvectors. Normalize the amplitudes so that the displacement of the third floor is one.

20.6 Figure P20.6 shows an LC network for which Kirchhoff's voltage law can be used to develop the following system of ODEs:

$$\begin{aligned} L_1 \frac{d^2 i_1}{dt^2} + \frac{1}{C_1} (i_1 - i_2) &= 0 \\ L_2 \frac{d^2 i_2}{dt^2} + \frac{1}{C_2} (i_2 - i_3) + \frac{1}{C_1} (i_1 - i_2) &= 0 \\ L_3 \frac{d^2 i_3}{dt^2} + \frac{1}{C_3} i_3 + \frac{1}{C_2} (i_2 - i_3) &= 0 \end{aligned}$$

where L_j = inductance of inductor j ($\text{H} = 1 \text{ s}^2/\text{F}$), i = current (amp), and C_j = capacitance of capacitor j (F). Using the same approach as described in Section 20.2, determine the eigenvalues and eigenvectors for this system if all L 's = 0.005 H and the C 's = 0.001 F.

20.7 Consider the mass-spring system in Fig. P20.7. Determine the eigenvalues and eigenvectors for this system for the case where $m_1 = 1 \text{ kg}$, $m_2 = 2 \text{ kg}$, and $k = 1 \text{ N/m}$.

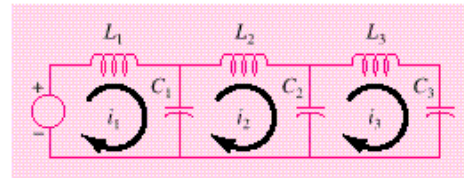


FIGURE P20.6

An LC circuit.

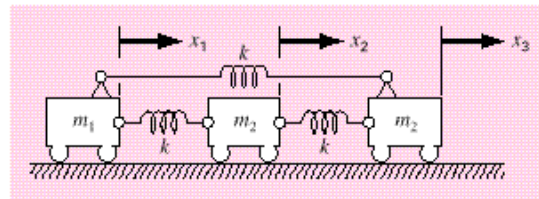


FIGURE P20.7

20.8 The two spring system studied in Example 20.1 can be simulated by solving the following differential equations:

$$\begin{aligned} \frac{d^2 x_1}{dt^2} + 5x_1 - 5(x_2 - x_1) &= 0 \\ \frac{d^2 x_2}{dt^2} + 5(x_2 - x_1) - 5x_2 &= 0 \end{aligned}$$

Integrate these equations and display the results graphically for the following initial conditions: (a) $x_1 = x_2 = 1$, and (b) $x_1 = 1$, $x_2 = 0.6$. In both cases, use zero initial conditions for the velocities. Discuss your results.

20.9 Develop an M-file to determine the highest eigenvalue and the associated eigenvector with the power method. Test it by for the system from Prob. 20.2.

20.10 Develop an M-file to determine the lowest eigenvalue and the associated eigenvector with the power method. Test it by for the system from Prob. 20.2.

APPENDIX A

MATLAB BUILT-IN FUNCTIONS

abs, 25	inline, 50, 327	pchip, 275
acos, 25	input, 34	pi, 18
besselj, 256	interp1, 274–277	plot, 26–28
chol, 169–170	inv, 133, 135–136, 176	poly, 118, 369
cond, 180–181, 238	length, 26	polyfit, 217, 230, 239, 252–255
conv, 119	linspace, 21	polyval, 118, 218, 239, 252–255
dblquad, 320	log, 25	quad, 318–319
deconv, 118	log10, 25	quad1, 318
disp, 35	log2, 25, 94	realmax, 64
eig, 372–373	loglog, 30	realmin, 65
elfun, 25	logm, 25	roots, 117–119, 369
eps, 65	logspace, 21	semilogy, 30
erf, 320	lookfor, 28	sign, 41, 322
error, 38	lu, 166–167	sin, 25
exp, 25	max, 29, 155	size, 134
eye, 133	mean, 51	spline, 272–274
factorial, 30, 44	norm, 180–181	sqrt, 25–26
feval, 49–50, 327	ode113, 348, 356	sqrtm, 25
fminsearch, 231	ode15s, 360	sum, 169
format long, 17	ode23, 347–348, 350–351	tanh, 5, 25–26
format short, 18	ode23s, 360–361	title, 27
fprintf, 36–37	ode23t, 360	trapz, 300
fzero, 113–116	ode23tb, 360	who, 19
grid, 27	ode45, 348–350, 361, 363	whos, 19
help, 33	odeset, 350–351	xlabel, 27
help elfun, 25	ones, 20	ylabel, 27
humps, 277, 319, 364	optimset, 115–116, 231	zeros, 20

APPENDIX B

MATLAB M-FILE FUNCTIONS

M-file Name	Description	Page
bisection	Root location with bisection	95
Eulode	Integration of a single ordinary differential equation with Euler's method	328
GaussNaive	Solving linear systems with Gauss elimination without pivoting	150
GaussPivot	Solving linear systems with Gauss elimination with partial pivoting	155
GaussSeidel	Solving linear systems with the Gauss-Seidel method	188
incsearch	Root location with an incremental search	88
Lagrange	Interpolation with the Lagrange polynomial	249
linregr	Fitting a straight line with linear regression	217
Newtint	Interpolation with the Newton polynomial	246
newtraph	Root location with the Newton-Raphson method	111
Romberg	Integration of a function with Romberg integration	311
trap	Integration of a function with the composite trapezoidal rule	291
trapuneq	Integration of unequidistant data with the trapezoidal rule	300
Tridiag	Solving tridiagonal linear systems	157

BIBLIOGRAPHY

- Bogacki, P. and L. F. Shampine, "A 3(2) Pair of Runge-Kutta Formulas," *Appl. Math. Letters*, 2(1989):1–9, 1989.
- Butcher, J. C., "On Runge-Kutta Processes of Higher Order," *J. Austral. Math. Soc.*, 4:179, 1964.
- Carnahan, B., H. A. Luther, and J. O. Wilkes, *Applied Numerical Methods*, Wiley, New York, 1969.
- Chapra, S. C. and R. P. Canale, *Numerical Methods for Engineers*, 4th ed., McGraw-Hill, New York, 2002.
- Dormand, J. and P. A. Prince, "A Family of Embedded Runge-Kutta Formulae," *J. Comput. and Appl. Math.*, 6:19–26, 1980.
- Draper, N. R. and H. Smith, *Applied Regression Analysis*, 2d ed., Wiley, New York, 1981.
- Fadeev, D. K. and V. N. Fadeeva, *Computational Methods of Linear Algebra*, Freeman, San Francisco, 1963.
- Gander, W. and W. Gautschi, "Adaptive Quadrature—Revisited," *BIT Numerical Mathematics*, 40:84–101, 2000.
- Gerald, C. F. and P. O. Wheatley, *Applied Numerical Analysis*, 3d ed., Addison-Wesley, Reading, MA, 1989.
- Hornbeck, R. W., *Numerical Methods*, Quantum, New York, 1975.
- James, M. L., G. M. Smith, and J. C. Wolford, *Applied Numerical Methods for Digital Computations with FORTRAN and CSMP*, 3d ed., Harper & Row, New York, 1985.
- Ortega, J. M., *Numerical Analysis—A Second Course*, Academic Press, New York, 1972.
- Palm, W. J. III, *Introduction to MATLAB 7 for Engineers*, McGraw-Hill, New York, 2005.
- Ralston, A., "Runge-Kutta Methods with Minimum Error Bounds," *Math. Comp.*, 16:431, 1962.
- Ralston, A. and P. Rabinowitz, *A First Course in Numerical Analysis*, 2d ed., McGraw-Hill, New York, 1978.
- Recktenwald, G., *Numerical Methods with MATLAB*, Prentice Hall, Englewood Cliffs, NJ, 2000.
- Scarborough, J. B., *Numerical Mathematical Analysis*, 6th ed., Johns Hopkins Press, Baltimore, MD, 1966.
- Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.
- White, F. M., *Fluid Mechanics*, McGraw-Hill, New York, 1999.

Accuracy, 59–60
Adams-Bashforth-Moulton solver, 348
Adaptive methods and stiff systems,
345–364
error estimates, 355–356
multistep methods, 351–356
non-self-starting Heun method,
352–354
Runge-Kutta. *See* Adaptive Runge-
Kutta methods
stiffness/stiff systems. *See*
Stiffness/stiff systems
Adaptive quadrature, 306, 318–319
Adaptive Runge-Kutta methods, 345–351
ode23 function, 347–348
ode45 function, 348
ode113 function, 348
odeset function, 350–351
solving a system of ODEs, 348–350
Adaptive Simpson quadrature, 318
Adaptive step-size control, 346
Addition, 21
flops, naive Gauss elimination, 151
large numbers with small numbers, 67
of two matrices, 129, 132
Analytical (closed-form) solutions, 6, 280
& (And) logical condition, 39
Approximate error, 61
Areal integral, 283
Arithmetic, round-off error and, 65–67
Arithmetic mean, 200
Array operations. *See* Element-by-element
operations
Arrays, 18–20, 24. *See also*
Matrix/Matrices
Arrow keys, up (↑), 24
Assignment, 16–21
arrays, vectors, and matrices, 18–20
colon operator, 20–21
linspace command, 21, 273
logspace command, 21
scalars, 16–18

Associative properties, 129–130
Augmentation, of a matrix, 131, 134
Backslash operator, 230. *See also*
Left division
Back substitution, 146, 148, 152, 162, 165
Backward difference approximations,
73, 74
Backward Euler's method, 358
Banded matrix, 129. *See also* Tridiagonal
matrix
Base-2 (binary) number system, 64
Base-8 (octal) number system, 64
Base-10 (decimal) number system, 63
“Best” fit criteria, 204–205
Bias, 59
Binary (base-2) number system, 64
Binary digits (bits), 63, 64
Binary search, 262–263
Bisection, 90–96
binary search and, 263
error estimates for, 92–93
false position vs., 97–99
M-files and, 95–96
Bits, 63, 64
Blunders, 78
Boole's rule, 298, 339
Bracketing methods, 81–100
bisection. *See* Bisection
defined, 85
false position, 96–99
graphical methods, 83–85
incremental search, 87–90
initial guesses and, 85–90
Built-in functions, 4n, 19, 25–26.
See also individual functions
(in ***Bold/Italics***)
Butcher's fifth-order RK method,
338–339
Calculus, 2, 4, 280
Calls, of other functions, 35–36

Carriage return. *See* ENTER key
(carriage return)
Cartesian coordinates, 141
Case-sensitivity
of function names, 33
of variable names, 17
Centered finite difference approximations,
73, 74
Characteristic polynomial, 366
Characteristic value. *See* Eigenvalues
Chemical engineering, conservation laws
for, 11–13
Cholesky decomposition, 167–170
Cholesky factorization, 168
chol function, 169–170
Circuits, conservation laws for, 11–13
Civil engineering, conservation laws for,
11–13
Clamped end condition, 271–272
Classical fourth-order RK methods,
337–339, 341–343
Closed-form solutions, 6, 280
Closed Newton-Cotes formulas, 285
Coefficient matrix, 134
Coefficient of determination, 210
Coefficient of variation, 201
Colon operator, 20–21
Colors, specifiers for, 28
Columns of a matrix, 127
Column-sum norm, 178
Column vector(s), 18, 127, 134
Command window, 16, 32, 34
Commas, separating commands, 17
Commutative properties, 129, 130
Companion matrix, 117
Complete pivoting, 153
Complex quantities, 17, 22
Composite integration formulas,
288–290
Composite Simpson's 1/3 rule,
294–295, 305
Composite trapezoidal rule, 288–290,
301–302

- Computation effort, accuracy and, 9
- Computer numbers, round-off errors and, 63–65
- Condition number, matrices, 179–181
- Conservation laws, 1, 9–13, 125–126
- Continuity condition, 264
- Control codes, 36
- Convergence, 101, 105
 - diagonal dominance and, 186
- Corrector, non-self-starting Heun method, 353
- Corrector equation, 330
- Correlation coefficient, 210
- Cramer's rule, 142–145, 192
- Cubic splines, 257, 266–272
 - end conditions, 271–272, 275–277
 - piecewise cubic Hermite interpolation (*pehip*), 275–277
- Current balance, 11–13
- Curve fitting, 12
 - defined, 198–199
 - engineering practice and, 198–199
 - fitting a straight line. *See* Fitting a straight line
 - polynomial interpolation. *See* Polynomial interpolation
 - splines. *See* Splines
 - statistics. *See* Statistics review
- Data distribution, 202
- Data uncertainty, errors and, 79
- Debug, Run, 32**
- Decimal (base-10) number system, 63
- Decimal places, 17–18
- Decisions, structured programming. *See* Structured programming
- Deflation, 372
- Degrees of freedom, 200–201, 208, 223, 227
- Dependent variables, 3
- Determinants, Cramer's rule and, 142–145
- Devices, conservation laws for, 11–13
- Diagonal dominance, 186
- Diagonal matrix, 128
- Differential equations, 2, 4, 12. *See also* Ordinary differential equations (ODEs)
- Digital computers, size/precision limits, 63
- display** parameter, 115
- Divergence, 101, 105, 186
- Divided differences. *See* Finite divided differences
- Divided difference table, 244
- Division, 21
 - left, 21, 21n, 136, 170–171, 230
 - matrix, 130
 - multiplication/division flops, 151
- Dominant eigenvalue, 369–372
- Dot product of two vectors, 23
- Double integral method, 301–303
- Drag coefficient, 4
- Echo printing, 17
- Edit window, 16
- Eigenvalues, 179, 365–374
 - characteristic-value problems, 365
 - eig* function, 372–373
 - highest, power method, 370–371
 - mathematical background, 365–366
 - physical background, 366–368
 - polynomial method, 368–369
 - power method, 369–372
 - roots* function, 117–118, 369
- Eigenvector, 368–369, 373
- eig* function, 372–373
- Electrical engineering, conservation laws for, 11–13
- Element-by-element operations, 24
- Element of a matrix, 127
- Elimination of unknowns, 145–148
- Embedded RK methods, 347
- End conditions, cubic splines, 271–272, 275–277
- Energy balance, 83
- Engineering practice
 - curve fitting, 198–199
 - linear algebraic equations, 125–126
 - roots of equations, 82–83
- ENTER key (carriage return), 19
 - to repeat calculations, 24
- (Equal), 39
- Error(s), 58–80
 - accuracy and precision, 59–60
 - blunders, 78
 - data uncertainty and, 79
 - defined, 59–63
 - true error, 60
 - true fractional relative error, 60–61
 - error analysis
 - for Euler's method, 325–327
 - matrix inversion, 176–181
 - and system condition, 176–181
 - error* function, 38
 - error messages, 23
 - estimates of
 - for iterative methods, 62–63, 103
 - multistep methods, 355–356
 - truncation errors. *See* Truncation errors
 - linear least-squares regression and, 207–211
 - model errors, 79
 - quantification of, 59–63
 - relative, 61
 - round-off. *See* Round-off errors
 - standard error of the estimate, 208
 - system condition and, 176–181
 - total numerical, 76–78
 - true error, 60
 - true fractional relative error, 60–61
 - truncation. *See* Truncation errors
- Estimates, error. *See* Error(s)
- Euclidean norm, 178–179
- Euler-Cauchy method, 323
- Euler's method, 8
 - error analysis for, 325–327
- Eulode* function, 327–329
 - explicit, 358–359
- feval* function, 49–50, 327
 - implicit, 358–359
 - modifications of, 329–334
 - ODEs and, 323–329, 340–341
- Explicit Euler's method, 358–359
- Explicit expression, in formulas, 83
- Exponential model, 211
- Exponentiation, 21
- External stimuli, 174
- Extrapolation, polynomial interpolation and, 251–253
- eye* function, 133–134
- factorial* function, 44n
- False position
 - bracketing methods and, 96–99
 - formula, 96
- feval* function, 49–50, 327
- File, New, M-file, 32**
- Finish value, loops, 43
- Finite difference, 72, 76
 - approximations, 76
- Finite divided difference
 - approximations, 7–8
 - of derivatives, 75–76
 - table, 244
- First-order approximation, 69
- First-order method, 327
- First-order splines, 260–261
- Fitting a straight line, 196–220
 - curve fitting, defined, 198–199
 - linear least-squares. *See* Linear least-squares regression
 - normal distribution, 202–203
 - statistics review, 199–203
- Floating-point arithmetic/operations (Flops), 150–153, 156, 166
- Floating point numbers, 64–65
- fminsearch* function, 231–232
- Force balance, 3–4, 11–13, 124
- Forcing functions, 3, 174
- for loop**, 43–44
- Format codes, 36
- format* function, 36
- format long** command, 17–18
- format short** command, 17–18
- for** structure, 43
- Forward difference approximations, 72–74
- Forward elimination of unknowns, 146, 147–148

- Fourth-order RK methods, 337–339, 341–343
- fprintf** function, 36
- Free-body diagram, 2–3, 124
- Functions
- listed. *See* MATLAB functions
 - M-files. *See* M-files
 - naming, 33
 - that call other functions, 35–36
- fzero** function, 113–116
- Gauss elimination, 140–161, 194
- GaussNaive** function, 149–150
 - GaussPivot** function, 154–155
 - as LU decomposition, 162–167
 - naive. *See* Naive Gauss elimination
 - pivoting, 153–155
 - small numbers of equations and, 141–146
 - Cramer’s rule, 142–145
 - determinants, 142–145
 - graphical method, 141–142
 - Tridiag** function, 157–158
 - Tridiagonal systems, 156–158
- Gauss-Legendre formula, 312
- three-point, 317
 - two-point, 314–316
- Gauss quadrature, 306, 310–318
- Gauss-Legendre formula, 312, 314–317
 - higher-point formulas, 317–318
 - undetermined coefficients method, 312–314
- Gauss-Seidel method, 183–189
- convergence and diagonal dominance, 186
 - GaussSeidel** function, 187, 188
 - relaxation, 187
- General linear least squares, 227–229
- Global truncation error, 326
- Graphics/Graphical methods
- graphics window, 16
 - with MATLAB, 26–28
 - root location, 83–85, 104–105
 - solving small numbers of equations, 141–142
- Gravity, force of, 4
- > (Greater Than), 39
 - >= (Greater than or equal to), 39
- Heat balance, 83
- help** command, 28–29
- Heun’s method, 329–333
- without iteration, second-order RK, 336
- Higher-order polynomials
- ill-conditioned systems, 253–255
 - polynomial interpolation and, 253–255
 - splines vs., 258
- Higher-point formulas, Gauss quadrature, 317–318
- Highest eigenvalue, 370–371
- Hilbert matrix, 179–180
- Histogram, 202
- Homogeneous linear algebraic systems, 366
- Hyperbolic tangent, 5
- Hypothesis testing, 198–199
- Identity matrix, 128, 130–131, 133, 135
- if . . . else** structure, 41–43
 - if . . . elseif** structure, 41–43
 - if** structure, 37–38
- Ill-conditioned systems, 141–142, 176–177
- higher-order polynomials, 253–255
 - matrices, 142
- Imaginary numbers, 16
- Implicit Euler’s method, 358–359
- Implicit expression, in formulas, 83
- Implicit solution of ODEs, 357–358
- Imprecision, 59
- Inaccuracy, 59
- Increment function, 323, 335
- Indentation, 47–49
- Independent variables, 3
- Index variable, loops, 43
- Initial guesses
- bracketing methods and, 85–90
 - defined, 85
 - incremental search and, 87–90
- inline** function, 49–50
- Inner product (dot product)
- round-off error and, 67
 - of two vectors, 23
- input** function, 34–35
- Input/output in programming, 34–37
- Integration, 12. *See also* Numerical integration formulas
- defined, 280–281
 - in engineering/science, 281–283
- Interactive M-file function, 35
- Interpolation, 198, 236–239
- interp1** function, 274–277
 - linear. *See* Linear interpolation
 - polynomial. *See* Polynomial interpolation
 - table lookup, 262–263
- Inverse
- inverse interpolation, 114, 250–251
 - inverse quadratic interpolation, 114
 - inv** function, 133
 - of matrices, 130–131, 133, 136
- Iteration/Iterative methods, 61
- error estimates for, 62–63, 103
 - for systems of equations, 183–195
 - Gauss-Seidel method. *See* Gauss-Seidel method
 - Newton-Raphson method, 191–194
 - nonlinear systems, 189–194
 - successive substitution, 190–191
- Jacobian, of the system, 192
- Jacobian matrix, 193–194
- Jacobi iteration, 185–186
- Kirchhoff’s laws, 83
- Knots, 261, 271
- Lagrange** function, 249
- Lagrange interpolating polynomial, 247–250
- Large computations, round-off errors and, 66
- Least-squares fit, straight lines, 205–207
- Least-squares regression, 198, 199
- Left division, 21, 21n, 136, 170–171, 230
- < (Less Than), 39
- <= (Less than or equal to), 39
- Linear algebraic equations, 12
- defined, 125–126
 - engineering practice and, 125–126
 - matrices and, 123–139
 - matrix algebra overview, 126–135
 - matrix form representation, 134–135
 - matrix manipulation, 131–134
 - matrix notation, 127–129
 - matrix operating rules, 129–134
 - solving with MATLAB, 135–137
- Linear convergence, 103
- Linear interpolation, 236, 274–277
- false position and, 96–99
 - Newton’s interpolating polynomials, 239–241
 - splines, 262–263
- Linearization, of nonlinear relationships, 211–215
- Linear least-squares regression, 203–215
- “best” fit criteria, 204–205
 - computer applications, 215–218
 - error, quantification of, 207–211
 - general comments on, 215
 - general linear least squares, 227–229
 - least-squares fit of a straight line, 205–207
 - linearization of nonlinear relationships, 211–215
 - linreg** function, 215–217
 - multiple linear regression, 225–227
 - polyfit** function, 217–218
 - polyval** function, 217–218
- Linear regression, defined, 227
- Linear splines, 259–263
- table lookup, 262–263
- Line types, specifiers, 28
- linreg** function, 215–217
- linspace** command, 21, 273
- Lobatto quadrature, 318

- Local error
 - Heun method, 332
 - truncation error, 355
 - with Euler's method, 325–327
 - log** function, 25
 - Logical conditions, 38–40
 - logspace** command, 21
 - lookfor** command, 28
 - Loops. *See* Structured programming
 - Lower triangular matrix, 129
 - LU** decomposition, 160–171
 - Cholesky decomposition, 167–170
 - chol** function, 169–170
 - Gauss elimination as, 162–167
 - left division, 170–171
 - lu** function, 166–167
 - Machine epsilon, 65
 - Machines, conservation laws for, 11–13
 - Maclaurin series expansion, 30, 55, 62
 - Main diagonal, of a matrix, 128
 - Mass balance, 11–13
 - Mathematical modeling, 1–9
 - defined, 3
 - Mathematical operations, 21–24
 - MathWorks, Inc., 29, 252
 - mathworks.com, 29
 - MATLAB, 5n. *See also* MATLAB
 - functions; M-files; Programming with MATLAB
 - assignment. *See* Assignment
 - built-in functions, 25–26
 - command window, 16
 - defined, 15
 - edit window, 16
 - fundamentals, 15–30
 - graphics window, 16
 - graphics with, 26–28
 - for linear algebraic equations, 135–137
 - mathematical operations, 21–24
 - M-files. *See* M-files
 - other resources, 28–29
 - for polynomial manipulation, 118–119
 - primary windows, 16
 - programming with. *See* Programming with MATLAB
 - MATLAB functions. *See also* MATLAB; M-files
 - bisection** function, 95
 - chol** function, 169–170
 - cond** function, 180–181, 238
 - eig** function, 372–373
 - error** function, 38
 - Eulode** function, 327–329
 - eye** function, 133–134
 - factorial** function, 44n
 - feval** function, 49–50, 327
 - fminsearch** function, 231–232
 - format long** function, 17
 - format short** function, 18
 - fprintf** function, 36
 - fzero** function, 113–116
 - GaussNaive** function, 149–150
 - GaussPivot** function, 154–155
 - GaussSeidel** function, 187, 188
 - incsearch** function, 18
 - inline** function, 49–50
 - input** function, 34–35
 - interp1** function, 274–277
 - inv** function, 133
 - Lagrange** function, 249
 - linreg** function, 215–217
 - log** function, 25
 - lu** function, 166–167
 - mean** function, 51
 - Newton** function, 246–247
 - newtraph** function, 110–111
 - for nonstiff systems, 347–351
 - nom** function, 180–181
 - ode15s** function, 360
 - ode23** function, 347–348
 - ode23s** function, 360
 - ode23t** function, 360
 - ode23tb** function, 360
 - ode45** function, 348, 361
 - ode113** function, 348
 - odeset** function, 350–351
 - optimset** function, 115–116, 231
 - plot** function, 26–28, 349
 - polyfit** function, 217–218, 230, 239, 253–254
 - polyval** function, 217–218, 239, 253–254
 - quad** function, 318–319
 - quadl** function, 318–319
 - realmax** function, 64–65
 - realmin** function, 65
 - Romberg** function, 311
 - roots** function, 117–119, 369
 - semilogy** function, 30
 - sign** function, 41, 322, 322n
 - size** function, 134
 - spline** function, 257, 272–274
 - for stiff ODEs, 362–363
 - for stiff systems, 360–361
 - trap** function, 290–291
 - Trapuneq** function, 299–300
 - trapz** function, 300
 - Tridiag** function, 157–158
 - tspan** function, 348
 - who** function, 19
 - whos** function, 19
 - zeros** function, 20
 - Matrix/Matrices, 18–20
 - of coefficients, 134
 - condition number, 179–181
 - form representation, 134–135
 - inverse/inversion, 172–181
 - calculating, 172–174
 - ill-conditioned systems, 176–177
 - matrix condition number, 179–181
 - norms in MATLAB, 180–181
 - stimulus-response computations, 174–176
 - vector and matrix norms, 177–179
 - linear algebraic equations and, 123–139
 - manipulation of, 131–134
 - matrix algebra overview, 126–135
 - matrix division, 130
 - matrix-matrix multiplication, 23
 - matrix norms, 177–181
 - matrix notation, 127–129
 - operating rules, 129–134
- Maximum error, minimizing, 204, 205
- Maximum likelihood principle, 207–208
- Mean, 51, 202, 208, 281
- Mechanical engineering, conservation laws for, 11–13
- Memory locations, 16
- .m file extension, 32
- M-files, 32–34. *See also* MATLAB functions
 - bisection and, 95–96
 - bungee jumper velocity, 52–55
 - function files, 32–34
 - .m extension, 32
 - passing functions to, 49–52
 - script files, 32
- Midpoint method, 333–334
- Newton-Cotes open integration formulas, 301
- second-order RK methods, 336
- Midtest loop, 46
- Minimax criterion, 205
- Minimization, 231–232
- Mixed operations, with scalars, 24
- Model errors, 79
- Modified secant method, 112–113
- Moler, Cleve, 252
- Multiple-application integration formulas, 288–290
- Multiple integrals, 301–303
- Multiple linear regression, 225–227
- Multiple roots, 85
- Multiplication, 21
 - of matrices, 129–130, 132–133
 - matrix-matrix, 23
- Multiplication/division flops, 151
- Multistep methods, 323
 - adaptive methods and stiff systems, 351–356
 - error estimates, 355–356
 - non-self-starting Heun method, 352–354
- Naive Gauss elimination, 146–153
 - back substitution, 146, 148, 152
 - forward elimination of unknowns, 146, 147–148
- GaussNaive** function, 149–150
- operation counting, 150–153

- Natural cubic splines, 270, 271
Natural end condition, 271
Nearest neighbor interpolation, 274, 275–277
Negation, 21
Nested structures, 47–49
Nesting and indentation, 47–49
newtint function, 246–247
Newton-Cotes formulas, 283–285, 296, 298
 open integration, 300–301
Newton linear-interpolation formula, 239–240
Newton-Raphson method, 106–111
 newtraph function, 110–111
 for nonlinear systems, 191–194
 slowly converging function with, 107–110
Newton's interpolating polynomials, 239–247
 general form, 243–245
 linear interpolation, 239–241
 newtint function, 246–247
 quadratic interpolation, 241–243
Newton's Second Law, 1, 3, 9
newtraph function, 110–111
Nonhomogeneous systems, 366
Nonlinear regression, 231–232
Nonlinear systems, iterative methods for.
 See Iteration/iterative methods
Non-self-starting Heun method, 352–354
Nonstiff systems, MATLAB functions
 for, 347–351
Normal distribution, 202–203
Normalization, 147
Norms
 column-sum, 178
 defined, 177
 Euclidean, 178–179
 Frobenius, 178
 in MATLAB, 180–181
 matrix, 177–181
 row-sum, 178
 spectral, 179
 vector and matrix, 177–179
~ (Not), 39
"Not-a-Knot" end condition, 271–272, 275–277
~= (Not equal), 39
*n*th-order Taylor series expansion, 70
Number systems, 63–64
Numerical differentiation. *See* Truncation errors
Numerical integration formulas, 279–304.
 See also Numerical integration of functions
 Boole's rule, 298, 339
 composite Simpson's 1/3 rule, 294–295, 305
 integration
 defined, 280–281
 in engineering/science, 281–283
 with unequal segments, 298–300
 midpoint method, 301
 multiple integrals, 301–303
 Newton-Cotes formulas, 283–285, 296, 298
 open methods, 300–301
 Simpson's 1/3 rule, 292–295, 298, 301–302
 Simpson's 3/8 rule, 296–297, 298
 trapezoidal rule, 285–291, 298
 single application of, 286–288
 trap function, 290–291
 Trapuneq function, 299–300
 trapz function, 300
 unequal segments, integration with, 298–300
Numerical integration of functions, 305–320. *See also* Numerical integration formulas
 adaptive quadrature, 306, 318–319
 Gauss quadrature. *See* Gauss quadrature
 higher-order corrections, 308
 quad function, 318–319
 quadr function, 318–319
 Richardson's extrapolation, 305, 306–308
 Romberg integration, 305, 306–310
Numerical methods, 1
 covered in this book, 12, 13
 defined, 6–8
 MATLAB implementation of, 15.
 See also MATLAB summary, 12
Numerical optimization, 135, 231

Octal (base-8) number system, 64
ode15s function, 360
ode23 function, 347–348
ode23s function, 360
ode23t function, 360
ode23tb function, 360
ode45 function, 348, 361
ode113 function, 348
ODEs. *See* Ordinary differential equations (ODEs)
odeset function, 350–351
ones function, 19–20
One-step methods, 323
Open integration, Newton-Cotes, 285, 301
Open methods, 101–122
 defined, 85
 fzero function, 113–116
 inverse quadratic interpolation, 114
 Newton-Raphson, 106–111
 newtraph function, 110–111
 numerical integration, 300–301
 optimset function, 115–116, 231
 polynomial manipulation, root determination, 118–119
 polynomials, 117–119
 roots function for polynomials, 117–119
 secant methods, 111–113, 114
 simple fixed-point iteration, 102–106
 slowly converging function with
 Newton-Raphson, 107–110
 two-curve method, 103–106
Operating systems, not case sensitive, 33
Operation counting, naive Gauss elimination, 150–153
optimset function, 115–116, 231
Ordinary differential equations (ODEs)
 adaptive methods/stiff systems. *See* Adaptive methods and stiff systems
 adaptive Runge-Kutta methods, 345–351
 initial value problems, 321–344
 Euler's method, 323–329
 error analysis for, 325–327
 Eulode function, 327–329
 feval function, 327
 modifications of, 329–334
 Heun's method, 329–333
 midpoint method, 333–334
 Ralston's method, 336–337
 Runge-Kutta methods. *See* Runge-Kutta methods
 sign function, 322, 322n
 signum function, 322, 322n
 systems of equations, 339–343
 Euler's method, 340–341
 Runge-Kutta methods, 341–343
! (Or) logical condition, 39
Oscillations, polynomial interpolation and, 253–255
Outer product of two vectors, 23
Overdetermined systems, 135, 230
Overflow, 65
Overrelaxation, 187

Parameters, 3, 175
Parentheses, overriding priority order, 22, 39
Partial pivoting, 153–154
Passing functions to M-files, 49–52
pchip option, 275–277
Per-step truncation error, 355–356
Physical background, eigenvalues, 366–368
Pi (π), 17
Piecewise interpolation, 272–277
 cubic Hermite (**pchip**), 275–277
Pivot element, 147
Pivot equation, 147
Pivoting, 153–155
Place value, 64
plot function, 26–28, 276–277, 349

- Point-slope method, 323
polyfit function, 217–218, 230, 239, 253–254
 Polynomial, 67, 117–119
 Polynomial coefficients, 237–239
 Polynomial interpolation, 235–256
 basics, 236–239
 extrapolation, 251–253
 higher-order, 253–255
 inverse, 250–251
 Lagrange function, 249
 Lagrange interpolating polynomial, 247–250
 Newton's. *See* Newton's interpolating polynomials
 oscillations, 253–255
 polyfit function, 217–218, 230, 239, 253–254
 polynomial coefficients, determining, 237–239
 polyval function, 217–218, 239, 253–254
 Runge's function, 253–255
 Polynomial method, for eigenvalues, 368–369
 Polynomial regression, 211, 221–225, 228–229
polyval function, 217–218, 239, 253–254
 Positional notation, 64
 Posttest loop, 46
 Power equation, 211, 212–215, 227
 Power method, for eigenvalues, 369–372
 Preallocation of memory, loops, 45
 Precision, 17–18, 59–60
 limits of digital computers, 63
 round-off errors and, 64
 Predefined variables, 17
 Predictor, non-self-starting Heun method, 353
 Predictor equation, 330
 Pretest loop, 46
 Primary windows, MATLAB, 16
 Principal diagonal, of a matrix, 128
 Priority order, 21–22, 39
 Programming with MATLAB, 31–57
 input-output, 34–37
 M-files. *See* M-files
 nesting and indentation, 47–49
 passing functions to M-files, 49–52
 feval function, 49–50
 inline function, 49–50
 structured programming. *See* Structured programming
 Propagated truncation error, 325–327
 Proportionality, 175, 176
 QR factorization, 171n, 230
quad function, 318–319
quadr function, 318–319
 Quadratic convergence, 107
 Quadratic interpolation, Newton's polynomials, 241–243
 Quadratic splines, 263–266
 Quadrature, 280. *See also* Numerical integration formulas
 Quantification, of error, 59–63
 Ralston's method, 336–337
 Range of representation, round-off errors and, 64–65
 Reactors, conservation laws for, 11–13
realmax function, 64–65
realmin function, 65
 Regression line, spread and, 208
 Relational operators, 38–39
 Relative error, 61
 Relaxation, Gauss-Seidel method, 187
 Residual error, 208–210
 Residuals, 204–205
 Resolution, of floating-point arithmetic, 65
 Response, of systems, 174
 Richardson's extrapolation, 305, 306–308
 Right-hand-side vector, 136
 RK methods. *See* Runge-Kutta methods
 Romberg integration, 305, 306–310
 Roots of equations, 12
 bracketing methods. *See* Bracketing methods
 engineering practice and, 82–83
 open methods. *See* Open methods
 roots defined, 82
 roots function, 117–119, 369
 Rosenbrock formula, 360
 Round-off errors, 63–67
 arithmetic manipulations and, 65–67
 computer number representations, 63–65
 with Euler's method, 325–327
 inner products, 67
 large computations, 66
 large numbers added to small numbers, 67
 precision and, 64
 range of representation and, 64–65
 smearing, 67
 subtractive cancellation, 66, 76
 truncation errors vs., 76–77
 Row of a matrix, 127
 Row-sum norm, 178
 Row vectors, 18, 127
 Runge-Kutta methods, 323, 329, 335–339, 360
 Butcher's fifth-order method, 338–339
 classical fourth-order method, 337–339, 341–343
 Heun method, without iteration, 336
 midpoint method, 336
 Ralston's method, 336–337
 RK-Fehlberg, 347
 second-order, 335–337
 for systems of equations, 341–343
 Runge's function, 253–255
 Saturation-growth-rate equation, 211–212
 Scalars, 16–18, 22, 24
 Script files, 32
 Secant methods, 111–113, 114
 Second-order polynomial, 221–223
 Second-order Runge-Kutta methods, 335–337
 Sequential performance of instructions, 37
 Sequential search, 262
sign function, 322, 322n
 Signum function, 322, 322n
 Simple fixed-point iteration, 102–106
 Simple statistics, 200–202
 Simpson's 1/3 rule, 292–295, 298, 301–302, 305
 Simpson's 3/8 rule, 296–297, 298
 Simultaneous equations, polynomial coefficients with, 237–238
 Single application, of trapezoidal rule, 286–288
 Singular systems, 141
 Singular value decomposition, 230
size function, 134
 Size limits, of digital computers, 63
 Slowly converging function, with Newton-Raphson, 107–110
 Small numbers of equations, graphical methods for, 141–142
 Smearing, round-off errors and, 67
 Spectral norm, 179
spline function, 257, 272–274
 Splines, 257–278
 cubic, 257, 266–272
 end conditions, 271–272, 275–277
 natural, 270, 271
 defined, 259
 higher-order polynomials vs., 258
 interp1 function, 274–277
 knots and, 261
 linear, 259–263
 first-order, 260–261
 table lookup, 262–263
 piecewise interpolation, 272–277
 quadratic, 263–266
 spline function, 257, 272–274
 Spread
 around the mean, 208
 around the regression line, 208
 degree of, 200
 Square matrix, 128
 Standard deviation, 200, 202
 Standard error of the estimate, 208
 Start value, loops, 43
 Statistics review, 199–203
 normal distribution, 202–203
 simple statistics, 200–202

- Steady-state calculation, 10
- Step halving, 347
- Step value, loops, 43
- Stiffness/stiff systems, 356–361
 - explicit Euler's method, 358–359
 - implicit Euler's method, 358–359
 - ode15s** function, 360
 - ode23s** function, 360
 - ode23t** function, 360
 - ode23tb** function, 360
 - ode45** function, 361
- Stimulus-response computations, 174–176
- Stopping criterion, 61–62
- Structured programming, 37–46
 - decisions, 37–43
 - error** function, 38
 - if . . . else** structure, 41–43
 - if . . . elseif** structure, 41–43
 - if** structure, 37–38
 - logical conditions, 38–39
 - priority order, 39
 - relational operators, 38–39
 - truth tables, 39
 - loops, 37, 43–46
 - to compute factorials, 44, 44n
 - index variable, 43
 - for loop**, 43
 - midtest loop, 46
 - posttest loop, 46
 - preallocation of memory, 45
 - pretest loop, 46
 - vectorization, 44–45
 - while . . . break**, 46, 54
 - while loop**, 43
 - while** structure, 45–46
- Structures, conservation laws for, 11–13
- Subtraction, 21, 129, 132
- Subtractive cancellation, 66, 76
- Successive overrelaxation (SOR), 187
- Successive substitution, 190–191
- Sum of residuals, 204
- Sum of the squares, 202
- Superposition, 175, 176
- Symbols, specifiers for, 28
- Symmetric matrix, 128
- System(s)
 - of equations, initial value problems, 339–343
 - interactions, 175
 - response of, 174
 - state of, 174
- TableLookBin**, 263
- TableLook** function, 262
- Table lookup, linear splines, 262–263
- Taylor series, 67–72, 326
- Taylor theorem, 67
- Terminal velocity, 6, 10
- Three-point Gauss-Legendre formula, 317
- Time-variable (transient) computations, 9–12
- tolx** parameter, 115
- Top-down design, 47
- Total numerical error, 76–78
- Trade-offs, round-off error vs. truncation, 77
- Transient computations, 9–12
- Transpose, of a matrix, 131, 132
- Trapezoidal rule, 285–291, 298
 - single application, 286–288
 - with unequal segments, 299
- trap** function, 290–291
- TrapUneq** function, 299–300
- trapz** function, 300
- Trend analysis, 198
- Trial and error, 82
- Tridiag** function, 157–158
- Tridiagonal matrix, 129
- Tridiagonal systems, 156–158
- True error, 60–61, 332
- True fractional relative error, 60–61
- True value, 60, 355
- Truncation errors, 67–76
 - with Euler's method, 325–327
 - numerical differentiation and, 72–76
 - backward difference
 - approximations, 73, 74
 - centered difference
 - approximations, 73, 74
 - centered finite difference
 - approximations, 73, 74
 - finite difference approximations, 73, 74, 76
 - finite-divided-difference of derivatives, 75–76
 - forward difference
 - approximations, 72–74
 - round-off errors vs., 76–77
 - Taylor series, 67–72
- Truth tables, 39
- tspan** parameter, 348
- Two-curve method, 103–106
- Two-point Gauss-Legendre formula, 314–316
- Uncertainty, 59
- Unconditionally stable method, 358
- Underdetermined systems, 135, 336
- Underflow, 65
- Underrelaxation, 187
- Undetermined coefficients method, 312–314
- Unequal segments
 - integration with, 298–300
 - trapezoidal rule with, 299
- Unknowns
 - column vector of, 134
 - elimination of, 145–148
- Up (\cdot) arrow key, 24
- Upper triangular matrix, 128
- User-defined functions, 32
- Vandermonde matrices, 238
- van der Pol equation 360–361
- Variable names
 - assignment of. *See* Assignment
 - case-sensitivity of, 17
- Variance, 200–201
- Vector, 18–20
- Vector and matrix norms, 177–179
- Vectorization, 44–45
- Vector-matrix calculations, 22–24
- Voltage, 11–13
- Volume integral, 283
- while . . . break** loops, 46, 262
- while . . . break** structure, 46, 54
- while loop**, 43, 45–46
- while** structure, 43, 45–46
- who** command, 19
- whos** command, 19
- www.mathworks.com, 29
- Zero-order approximation, 68
- Zeros, of equations, 82
- zeros** command, 19–20