Contents

PREFACE

This book is designed to support a one-semester course in numerical methods. It has been written for students who want to learn and apply numerical methods in order 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 basic content, organization, and pedagogy of the second edition are essentially preserved in the third edition. In particular, the conversational writing style is intentionally maintained in order to make the book easier to read. This book tries to speak directly to the reader and is designed in part to be a tool for self-teaching.

That said, this edition differs from the past edition in three major ways: (1) two new chapters, (2) several new sections, and (3) revised homework problems.

1. **New Chapters.** As shown in 1, I have developed two new chapters for this edition. Their inclusion was primarily motivated by my classroom experience. That is, they are included because they work well in the undergraduate numerical methods course I teach at Tufts. The students in that class typically represent all areas of engineering and range from sophomores to seniors with the majority at the junior level. In addition, we typically draw a few math and science majors. The two new chapters are:

   - **Eigenvalues.** When I first developed this book, I considered that eigenvalues might be deemed an "advanced" topic. I therefore presented the material on this topic at the end of the semester and covered it in the book as an appendix. This sequencing had the ancillary advantage that the subject could be partly motivated by the role of eigenvalues in the solution of linear systems of ODEs. In recent years, I have begun

     to move this material up to what I consider to be its more natural mathematical position at the end of the section on linear algebraic equations. By stressing applications (in particular, the use of eigenvalues to study vibrations), I have found that students respond very positively to the subject in this position. In addition, it allows me to return to the topic in subsequent chapters which serves to enhance the studentsâĂŹ appreciation of the topic.

   - **Fourier Analysis.** In past years, if time permitted, I also usually presented a lecture at the end of the semester on Fourier analysis. Over the past two years, I have begun presenting this material at its more natural position just after the topic of linear least squares. I motivate the subject matter by using the linear least-squares approach to fit sinusoids to data. Then, by stressing applications (again vibrations), I have found that the students readily absorb the topic and appreciate its value in engineering and science.

     It should be noted that both chapters are written in a modular fashion and could be skipped without detriment to the course's pedagogical arc. Therefore, if you choose, you can either omit them from your course or perhaps move them to the end of the semester. In any event, I would not have included them in the current edition if they did not represent an enhancement within my current experience in the classroom. In particular, based on my teaching evaluations, I find that the stronger, more motivated students actually see these topics as highlights. This is particularly true because MATLAB greatly facilitates their application and interpretation.

2. **New Content.** Beyond the new chapters, I have included new and enhanced sections on a number of topics. The primary additions include sections on animation (Chap. 3), Brent's method for root location (Chap. 6), *LU* factorization with pivoting (Chap. 8), *random numbers and Monte Carlo simulation* (Chap. 14), *adaptive quadrature* (Chap. 20),and *event termination of ODEs* (Chap. 23).

3. **New Homework Problems.** Most of the end-of-chapter problems have been modified, and a variety of new problems have been added. In particular, an effort has been made to include several new problems for each chapter that are more challenging and difficult than the problems in the previous edition.

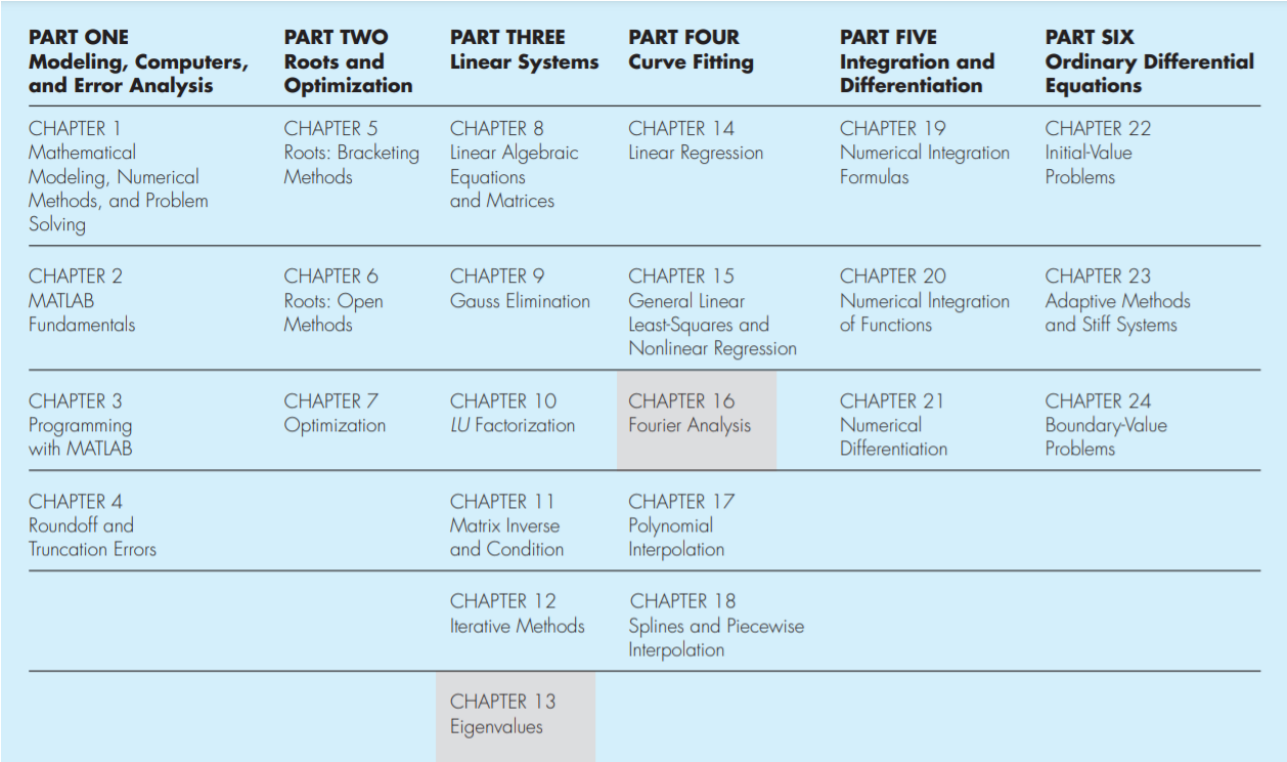


Figure 1: An outline of this edition. The shaded areas represent new material. In addition, several of the original chapters have been supplemented with new topics.

Aside from the new material and problems, the third edition is very similar to the second. In particular, I have endeavored to maintain most of the features contributing to its pedagogical effectiveness including extensive use of worked examples and engineering and scientific applications. As with the previous edition, I have made a concerted effort to make this book as "student-friendly" as possible. Thus, IâĂŹve tried to keep my explanations straightforward and practical. 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, mathemati cs and yes programming, will ultimately make better professionals. If my book fosters enthusiasm and appreciation for these subjects, 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 Lorraine Buczek, and Bill Stenquist, and Melissa Leick for their encouragement, support, and direction. Ruma Khurana of MPS Limited, a Macmillan Company also did an outstanding job in the bookâĂŹs final production phase. Last, but not least, Beatrice Sussman once again demonstrated why she is the best copyeditor in the business. During the course of this project, the folks at The MathWorks, Inc., have truly demonstrated their overall excellence as well as their strong commitment to engineering and science education. In particular, Courtney Esposito and Naomi Fernandes of The MathWorks, Inc., Book Program have been especially helpful. The generosity of the Berger family, and in particular Fred Berger, has provided me with the opportunity to work on creative projects such as this book dealing with computing and engineering. In addition, my colleagues in the School of Engineering at Tufts, notably Masoud Sanayei, Lew Edgers, Vince Manno, Luis Dorfmann, Rob White, Linda Abriola, and Laurie Baise, have been very supportive and helpful. Significant suggestions were also given by a number of colleagues. In particular, Dave Clough (University of ColoradoâĂŞBoulder), and Mike Gustafson (Duke University) provided valuable ideas and suggestions. In addition, a number of reviewers provided useful feedback and advice including Karen Dow Ambtman (University of Alberta), Jalal Behzadi (Shahid Chamran University), Eric Cochran (Iowa State University), Frederic Gibou (University of California at Santa Barbara), Jane Grande-Allen (Rice University), Raphael Haftka (University of Florida), Scott Hendricks (Virginia Tech University), Ming Huang (University of San Diego), Oleg Igoshin (Rice University), David Jack (Baylor University), Clare McCabe (Vanderbilt University), Eckart Meiburg (University of California at Santa Barbara), Luis Ricardez (University of Waterloo), James Rottman (University of California, San Diego), Bingjing Su (University of Cincinnati), Chin-An Tan (Wayne State University), Joseph Tipton (The University of Evansville), Marion W. Vance (Arizona State University), Jonathan Vande Geest (University of Arizona), and Leah J. Walker (Arkansas State University). 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 the love, patience, and support they have provided through the time IâĂŹve spent on this project.

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

PEDAGOGICAL TOOLS

Theory Presented as It Informs Key Concepts. The text is intended for Numerical Methods users, not developers. Therefore, theory is not included for âĂIJtheoryâĂŹs sake,âĂİ 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.

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 MATLABâĂŹs 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 MATLABâĂŹs 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.

Worked Examples and Case Studies. Extensive worked examples are laid out in detail so that students can clearly follow the steps in each numerical computation. The case studies consist of engineering and science applications which are more complex and richer than the worked examples. They are placed at the ends of selected chapters with the intention of (1) illustrating the nuances of the methods, and (2) showing more realistically how the methods along with MATLAB are applied for problem solving.

Problem Sets. The text includes a wide variety of problems. Many are drawn from engineering and scientific disciplines. Others are used to illustrate numerical techniques and theoretical concepts. Problems include those that can be solved with a pocket calculator as well as others that require computer solution with MATLAB.

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

Textbook Website. A text-specific website is available at www.mhhe.com/chapra. Resources include the text images in PowerPoint, M-files, and additional MATLAB resources.

Part I

Modeling, Computers and Error Analysis

0.1. MOTIVATION

What are numerical methods and why should you study them?

Numerical methods are techniques by which mathematical problems are formulated so that they can be solved with arithmetic and logical operations. Because digital computers excel at performing such operations, numerical methods are sometimes referred to as computer mathematics.

In the preâĂŞcomputer era, the time and drudgery of implementing such calculations seriously limited their practical use. However, with the advent of fast, inexpensive digital computers, the role of numerical methods in engineering and scientific problem solving has exploded. Because they figure so prominently in much of our work, I believe that numerical methods should be a part of every engineerâĂŹs and scientistâĂŹs basic education. Just as we all must have solid foundations in the other areas of mathematics and science, we should also have a fundamental understanding of numerical methods. In particular, we should have a solid appreciation of both their capabilities and their limitations. Beyond contributing to your overall education, there are several additional reasons why you should study numerical methods:

1. Numerical methods greatly expand the types of problems you can address. They are capable of handling large systems of equations, nonlinearities, and complicated geometries that are not uncommon in engineering and science and that are often impossible to solve analytically with standard calculus. As such, they greatly enhance your problem-solving skills.

2. Numerical methods allow you to use âĂIJcannedâĂİ software with insight. During your career, you will invariably have occasion to use commercially available prepackaged computer programs that involve numerical methods. The intelligent use of these programs is greatly enhanced by an understanding of the basic theory underlying the methods. In the absence of such understanding, you will be left to treat such packages as âĂIJblack boxesâĂİ with little critical insight into their inner workings or the validity of the results they produce.

3. Many problems cannot be approached using canned programs. If you are conversant with numerical methods, and are adept at computer programming, you can design your own programs to solve problems without having to buy or commission expensive software.

4. Numerical methods are an efficient vehicle for learning to use computers. Because numerical methods are expressly designed for computer implementation, they are ideal for illustrating the computerâĂŹs powers and limitations. When you successfully implement numerical methods on a computer, and then apply them to solve otherwise intractable problems, you will be provided with a dramatic demonstration of how computers can serve your professional development. At the same time, you will also learn to acknowledge and control the errors of approximation that are part and parcel of large-scale numerical calculations.

5. Numerical methods provide a vehicle for you to reinforce your understanding of mathematics. Because one function of numerical methods is to reduce higher mathematics to basic arithmetic operations, they get at the âĂIJnuts and boltsâĂİ of some otherwise obscure topics. Enhanced understanding and insight can result from this alternative perspective.

With these reasons as motivation, we can now set out to understand how numerical methods and digital computers work in tandem to generate reliable solutions to mathematical problems. The remainder of this book is devoted to this task.

0.2. PART ORGANIZATION

This book is divided into six parts. The latter five parts focus on the major areas of numerical methods. Although it might be tempting to jump right into this material, *Part One* consists of four chapters dealing with essential background material.

Chapter 1 provides a concrete example of how a numerical method can be employed to solve a real problem. To do this, we develop a *mathematical model* of a free-falling bungee jumper. The model, which is based on NewtonâĂŹs second law, results in an ordinary differential equation. After first using calculus to develop a closed-form solution, we then show how a comparable solution can be generated with a simple numerical method. We end the chapter with an overview of the major areas of numerical methods that we cover in Parts Two through Six.

Chapters 2 and 3 provide an introduction to the MATLAB© software environment. *Chapter 2* deals with the standard way of operating MATLAB by entering commands one at a time in the so-called calculator, or command, mode. This interactive mode provides a straightforward means to orient you to the environment and illustrates how it is used for common operations such as performing calculations and creating plots.

Chapter 3 shows how MATLABâĂŹs programming mode provides a vehicle for assembling individual commands into algorithms. Thus, our intent is to illustrate how MATLAB serves as a convenient programming environment to develop your own software.

Chapter 4 deals with the important topic of error analysis, which must be understood for the effective use of numerical methods. The first part of the chapter focuses on the *roundoff errors* that result because digital computers cannot

represent some quantities exactly. The latter part addresses *truncation errors* that arise from using an approximation in place of an exact mathematical procedure.

Chapter 1

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 cover in this book.

YOU'VE GOT A PROBLEM

Suppose that a bungee-jumping company hires you. YouâĂŹre 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 (NewtonâĂŹs second law). Based on this insight and your knowledge of physics and fluid mechanics, you develop the following mathematical model for the rate of change of velocity with respect to time,

Figure 1.1: Forces acting on a free-falling bungee jumper

$\frac{dv}{dt} = g - \frac{c_d}{m}v^2$ where v = downward vertical velocity (m/s), t = time (s), g = the acceleration due to gravity ($\cong$ 9.81 m/s2), $c_d = a$ lumped drag coefficient (kg/m), and m = the jumperâĂŹs mass (kg). The drag coefficient is called âĂIJlumpedâĂİ because its magnitude depends on factors such as the jumperâĂŹs area and the fluid density (see Sec 1.4).

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 computeroriented 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

$$\overset{Dependent}{variable} = f\left(\overset{independent}{variables}, parameters, \overset{forcing}{functions}\right) \tag{1.1}$$

where the *dependent variable* is a characteristic that typically reflects the behavior or state of the system; the independent variables are usually dimensions, such as time and space, along which the systemâĂŹs 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

$$F = ma \tag{1.2}$$

where F is the net force acting on the body $(N, orkgm/s^2)$, 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

$$a = \frac{F}{m} \tag{1.3}$$

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.

- 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 earthâĂŹs 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, NewtonâĂŹs second law can be used to determine the terminal velocity of a free-falling body near the earthâĂŹs 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

$$\frac{dv}{dt} = \frac{F}{m} \tag{1.4}$$

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):

$$F = F_D + F_U \tag{1.5}$$

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

$$F_D = mg \tag{1.6}$$

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,

$$F_U = -c_d v^2 \tag{1.7}$$

where c_d is a proportionality constant called the *lumped 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

$$\frac{dv}{dt} = g - \frac{C_d}{m} v^2 \tag{1.8}$$

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

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

where tanh is the hyperbolic tangent that can be either computed directly[1] or via the more elementary exponential function as in

$$tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.10}$$

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.

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

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

[1] MATLAB allows direct calculation of the hyperbolic tangent via the built-in function $tanh(x)$.

which can be used to compute

t, s	v, 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/hr) is attained after 10 s. Note also that after a sufficiently long

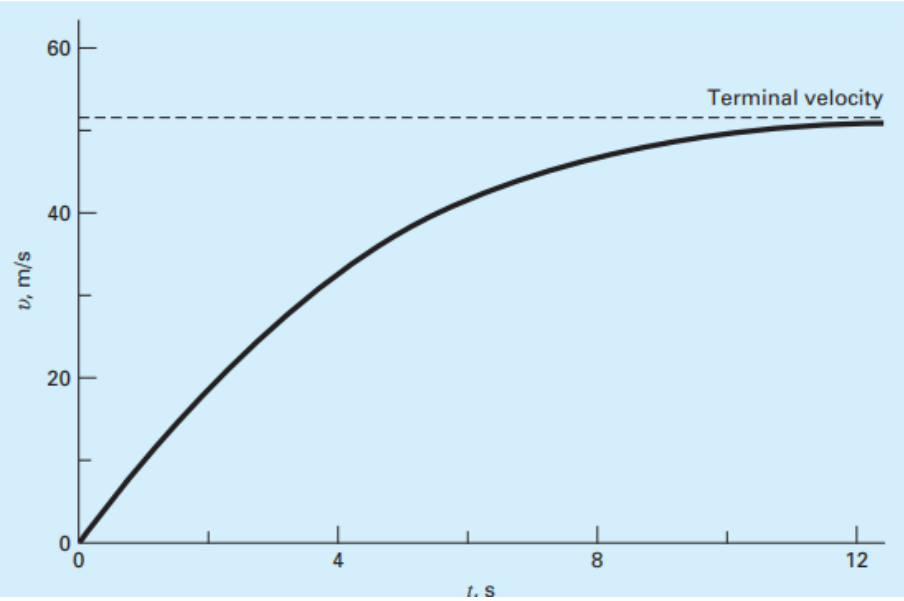


Figure 1.2: The analytical solution for the bungee jumper problem as computed in Example 1.1. Velocity increases with time and asymptotically approaches a terminal velocity.

time, a constant velocity, called the terminal velocity, of 51.6983 m/s (115.6 mi/hr) 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):

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

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 $dv/dt \cong \Delta v/\Delta t$ is approximate because Δt is finite. Remember from calculus that

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

Equation (1.11) represents the reverse process.

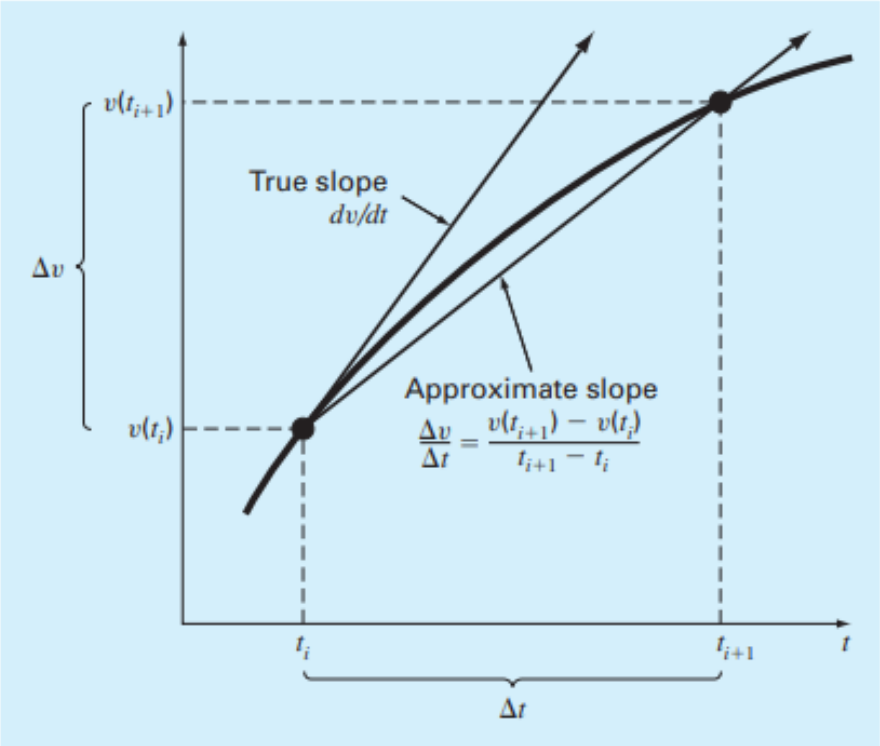


Figure 1.3: The use of a finite difference to approximate the first derivative of v with respect to t.

```
>> 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
```

Chapter 2

Roundoff 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 roundoff errors occur because digital computers have a limited ability to represent numbers.
- Understanding why floating-point numbers have limits on their range and precision.
- 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.
- Recognizing that efforts to minimize truncation errors can sometimes increase roundoff errors.

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 finite difference.

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

Thus, the resulting solution is not exact — 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. Consequently, 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? In particular, is it possible to understand, quantify and control such errors in order to obtain acceptable results? This chapter introduces you to some approaches and concepts that engineers and scientists use to deal with this dilemma.

2.1. 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: roundoff error (due to computer approximations) and truncation error (due to mathematical approximations). We also describe how strategies to reduce truncation error sometimes increase roundoff. Finally, we briefly discuss errors not directly connected with the numerical methods themselves. These include blunders, model errors, and data uncertainty.

2.1.1. 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.

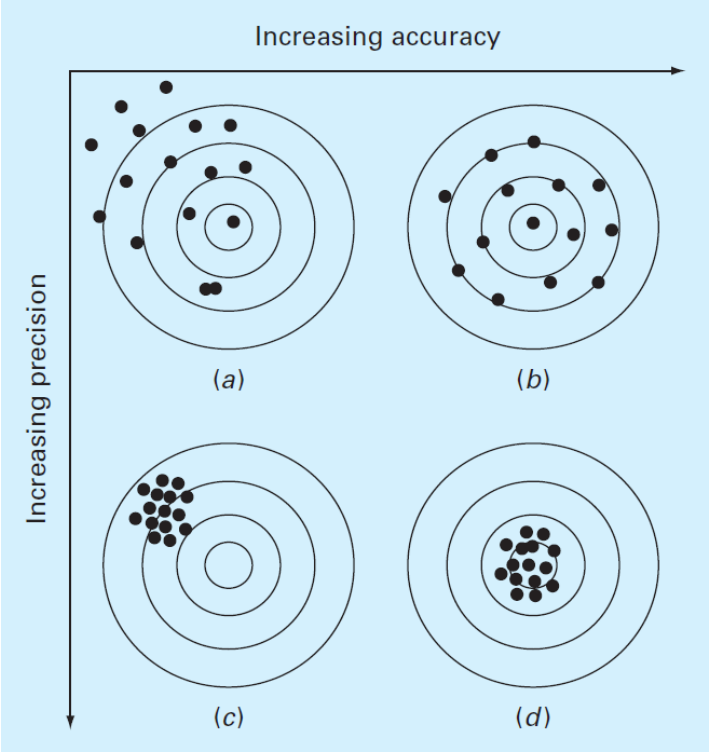


Figure 2.1: An example from marksmanship illustrating the concepts of accuracy and precision: (*a*) inaccurate and imprecise, (*b*) accurate and imprecise, (*c*) inaccurate and precise, and (*d*) accurate and precise.

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.

2.1.2. 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

$$\text{True value} = \text{approximation} + \text{error} \tag{4.1}$$

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

$$E_t = \text{true value - approximation} \tag{4.2}$$

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. Note that the true error is commonly expressed as an absolute value and referred to as the *absolute error*.

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

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

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

$$\varepsilon_t = \frac{\text{true value} - \text{approximation}}{\text{true value}} 100\% \tag{4.3}$$

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

$$\varepsilon_a = \frac{\text{approximate error}}{\text{approximation}} 100\% \tag{4.4}$$

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 cases, 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 the previous and present approximations. Thus, percent relative error is determined according to

$$\varepsilon_a = \frac{\text{present approximation} - \text{previous approximation}}{\text{present approximation}} 100\% \tag{4.5}$$

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 absolute value of the percent relative error is lower than a prespecified 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

$$|\varepsilon_a| < \varepsilon_s \tag{4.6}$$

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.

$$\varepsilon_s = (0.5 \times 10^{2-n})\% \tag{4.7}$$

Example 2.1. 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

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} \tag{E4.1.1}$$

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 $e^{0.5} = 1.648721\ldots$ 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:

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

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)]

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

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

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

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_t, %	ε_a, %
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.

■

2.1.3. Computer Algorithm for Iterative Calculations

Many of the numerical methods described in the remainder of this text involve iterative calculations of the sort illustrated in Example 4.1. These all entail solving a mathematical problem by computing successive approximations to the solution starting from an initial guess.

The computer implementation of such iterative solutions involves loops. As we saw in Sec. 3.3.2, these come in two basic flavors: count-controlled and decision loops. Most iterative solutions use decision loops. Thus, rather than employing a pre-specified number of iterations, the process typically is repeated until an approximate error estimate falls below a stopping criterion as in Example 4.1.

To do this for the same problem as Example 4.1, the series expansion can be expressed as

$$e^x \cong \sum_{i=0}^{n} \frac{x^n}{n!}$$

An M-file to implement this formula is shown in Fig. 4.2. The function is passed the value to be evaluated (`x`) along with a stopping error criterion (`es`) and a maximum allowable number of iterations (`maxit`). If the user omits either of the latter two parameters, the function assigns default values.

```
function [fx, ea, iter] = IterMeth(x,es,maxit)
% Maclaurin series of exponential function
%   [fx,ea,iter] = IterMeth(x,es,maxit)
% input:
%   x = value at which series evaluated
%   es = stopping criterion (default = 0.0001)
%   maxit = maximum iterations (default = 50)
% output:
%   fx = estimated value
%   ea = approximate relative error (%)
%   iter = number of iterations
% defaults:
if nargin<2|isempty(es), es=0.0001; end
if nargin<3|isempty(maxit), maxit=50; end
% initialization
iter = 1; sol = 1; ea = 100;
% iterative calculation
while (1)
    solold = sol;
    sol = sol + x ^ iter / factorial(iter);
    iter = iter + 1;
    if sol~=0
        ea=abs((sol - solold)/sol)*100;
    end
    if ea<=es | iter>=maxit,break,end
end
fx = sol
end
```

Figure 2.2: An M-file to solve an iterative calculation. This example is set up to evaluate the Maclaurin series expansion for ex as described in Example 4.1.

The function then initializes three variables: *(a)* `iter`, which keeps track of the number of iterations, *(b)* `sol`, which holds the current estimate of the solution, and *(c)* a variable, `ea`, which holds the approximate percent relative error. Note that *ea* is initially set to a value of 100 to ensure that the loop executes at least once.

These initializations are followed by a decision loop that actually implements the iterative calculation. Prior to generating a new solution, the previous value, `sol`, is first assigned to `solold`. Then a new value of `sol` is computed and the iteration counter is incremented. If the new value of `sol` is nonzero, the percent relative error, `ea`, is determined. The stopping criteria are then tested. If both are false, the loop repeats. If either is true, the loop terminates and the final solution is sent back to the function call.

When the M-file is implemented, it generates an estimate for the exponential function which is returned along with the approximate error and the number of iterations. For example, e^1 can be evaluated as

```
» format long
» [approxval, ea, iter] = IterMeth(1,1e-6,100)
approxval =
      2.718281826198493
ea =
      9.216155641522974e-007
iter =
      12
```

We can see that after 12 iterations, we obtain a result of 2.7182818 with an approximate error estimate of $= 9.2162 \times 10^{-7}\%$. The result can be verified by using the built-in `exp` function to directly calculate the exact value

and the true percent relative error,

```
» trueval=exp(1)
trueval =
      2.718281828459046
» et=abs((trueval- approxval)/trueval)*100
et =
      8.316108397236229e-008
```

As was the case with Example 4.1, we obtain the desirable outcome that the true error is less than the approximate error.

2.2. ROUNDOFF ERRORS

Roundoff 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. Such calculations are said to be *ill-conditioned*. Worse still, they can lead to subtler discrepancies that are difficult to detect.

There are two major facets of roundoff errors involved in numerical calculations:

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

2.2.1. Computer Number Representation

Numerical roundoff 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

$$(8\times10^3)+(6\times10^2)+(4\times10^1)+(2\times10^0)+(9\times10^{-1})=8642.9$$

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 $(1\times2^2)+(0\times2^1)+(1\times2^0)+(1\times2^{-1})=4+0+1+0.5=5.5$ in the decimal system.

Integer Representation. Now that we have reviewed how base-10 numbers can be represented in binary form, it is simple to conceive of how integers are represented on a computer. The most straightforward approach, called the *signed magnitude method*, employs the first bit of a word to indicate the sign, with a 0 for positive and a 1 for negative. The remaining bits are used to store the number. For example, the integer value of 173 is represented in binary as 10101101:

$$(10101101)_2=2^7+2^5+2^3+2^2+2^0=128+32+8+4+1=(173)_{10}$$

Therefore, the binary equivalent of –173 would be stored on a 16-bit computer, as depicted in Fig. 4.3.

If such a scheme is employed, there clearly is a limited range of integers that can be represented. Again assuming a 16-bit word size, if one bit is used for the sign, the 15 remaining bits can represent binary integers from 0 to 111111111111111. The upper limit can be converted to a decimal integer, as in

$$(1\times 2^{14})+(1\times 2^{13})+\ldots+(1\times 2^{1})+(1\times 2^{0})=32,767.$$

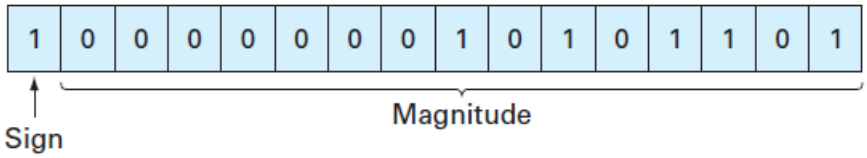


Figure 2.3: The binary representation of the decimal integer -173 on a 16-bit computer using the signed magnitude method.

Note that this value can be simply evaluated as $2^{15}-1$. Thus, a 16-bit computer word can store decimal integers ranging from –32,767 to 32,767.

In addition, because zero is already defined as 0000000000000000, it is redundant to use the number 1000000000000000 to define a "minus zero". Therefore, it is conventionally employed to represent an additional negative number: –32,768, and the range is from –32,768 to 32,767. For an n-bit word, the range would be from -2^{n-1} to $2^{n-1}-1$. Thus, 32-bit integers would range from –2,147,483,648 to +2,147,483,647.

Note that, although it provides a nice way to illustrate our point, the signed magnitude method is not actually used to represent integers for conventional computers. A preferred approach called the 2s *complement* technique directly incorporates the sign into the number's magnitude rather than providing a separate bit to represent plus or minus. Regardless, the range of numbers is still the same as for the signed magnitude method described above.

The foregoing serves to illustrate how all digital computers are limited in their capability to represent integers. That is, numbers above or below the range cannot be represented. A more serious limitation is encountered in the storage and manipulation of fractional quantities as described next.

Floating-Point Representation. Fractional quantities are typically represented in computers using *floating-point format*. In this approach, which is very much like scientific notation, the number is expressed as

$$\pm s\times b^{e}$$

where s = the *significand* (or *mantissa*), b = the base of the number system being used, and e = the exponent.

Prior to being expressed in this form, the number is *normalize*d by moving the decimal place over so that only one significant digit is to the left of the decimal point. This is done so computer memory is not wasted on storing useless nonsignificant zeros. For example, a value like 0.005678 could be represented in a wasteful manner as 0.005678×10^{0}. However, normalization would yield 5.678×10^{-3} which eliminates the useless zeroes.

Before describing the base-2 implementation used on computers, we will first explore the fundamental implications of such floating-point representation. In particular, what are the ramifications of the fact that in order to be stored in the computer, both the mantissa and the exponent must be limited to a finite number of bits? As in the next example, a nice way to do this is within the context of our more familiar base-10 decimal world.

Example 2.2. Implications of Floating-Point Representation

Problem Statement. Suppose that we had a hypothetical base-10 computer with a 5-digit word size. Assume that one digit is used for the sign, two for the exponent, and two for the mantissa. For simplicity, assume that one of the exponent digits is used for its sign, leaving a single digit for its magnitude.

Solution. A general representation of the number following normalization would be

$$s_1 d_1 d_2\times 10^{s_0 d_0}$$

where s_0 and s_1 = the signs, d_0 = the magnitude of the exponent, and d_1 and d_2 = the magnitude of the significand digits.

Now, let's play with this system. First, what is the largest possible positive quantity that can be represented? Clearly, it would correspond to both signs being positive and all magnitude digits set to the largest possible value in base-10, that is, 9:

$$\text{Largest value} = +9.9\times 10^{+9}$$

So the largest possible number would be a little less than 10 billion. Although this might seem like a big number, it's really not that big. For example, this computer would be incapable of representing a commonly used constant like Avogadro's number (6.022×1023). In the same sense, the smallest possible positive number would be

$$\text{Smallest value} = +1.0 \times 10^{-9}$$

Again, although this value might seem pretty small, you could not use it to represent a quantity like Planck's constant ($6.626 \times 10^{-34} J \cdot s$).

Similar negative values could also be developed. The resulting ranges are displayed in Fig. 4.4. Large positive and negative numbers that fall outside the range would cause an overflow error. In a similar sense, for very small quantities there is a "hole" at zero, and very small quantities would usually be converted to zero.

Recognize that the exponent overwhelmingly determines these range limitations. For example, if we increase the mantissa by one digit, the maximum value increases slightly to 9.99×10^9. In contrast, a one-digit increase in the exponent raises the maximum by 90 orders of magnitude to 9.9×10^{99}!

When it comes to precision, however, the situation is reversed. Whereas the significand plays a minor role in defining the range, it has a profound effect on specifying the precision. This is dramatically illustrated for this example where we have limited the significand to only 2 digits. As in Fig. 4.5, just as there is a "hole" at zero, there are also "holes" between values.

For example, a simple rational number with a finite number of digits like $2^{-5} = 0.03125$ would have to be stored as 3.1×102 or 0.031. Thus, a *roundoff error* is introduced. For this case, it represents a relative error of

$$\frac{0.03125 - 0.031}{0.03125} = 0.008$$

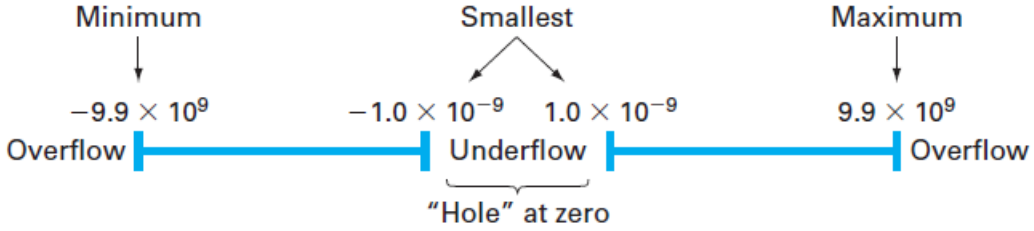


Figure 2.4: The number line showing the possible ranges corresponding to the hypothetical base-10 floating-point scheme described in Example 4.2.

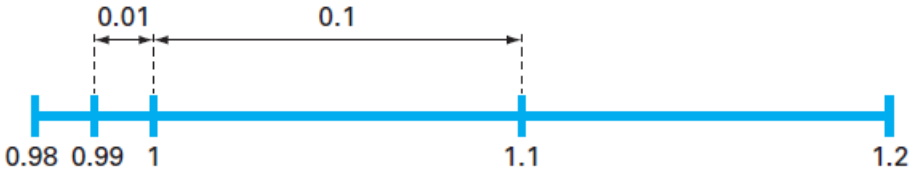


Figure 2.5: A small portion of the number line corresponding to the hypothetical base-10 floating-point scheme described in Example 4.2. The numbers indicate values that can be represented exactly. All other quantities falling in the "holes" between these values would exhibit some roundoff error.

While we could store a number like 0.03125 exactly by expanding the digits of the significand, quantities with infinite digits must always be approximated. For example, a commonly used constant such as $\pi (= 3.14159\ldots)$ would have to be represented as 3.1×10^0 or 3.1. For this case, the relative error is

$$\frac{3.14159 - 3.1}{3.14159} = 0.0132$$

Although adding significand digits can improve the approximation, such quantities will always have some roundoff error when stored in a computer.

Another more subtle effect of floating-point representation is illustrated by Fig. 4.5. Notice how the interval between numbers increases as we move between orders of magnitude. For numbers with an exponent of –1 (i.e., between 0.1 and 1), the spacing is 0.01. Once we cross over into the range from 1 to 10, the spacing increases to 0.1. This means that the roundoff error of a number will be proportional to its magnitude. In addition, it means that the relative error will have an upper bound. For this example, the maximum relative error would be 0.05. This value is called the *machine epsilon* (or machine precision).

■

As illustrated in Example 4.2, the fact that both the exponent and significand are finite means that there are both range and precision limits on floating-point representation. Now, let us examine how floating-point quantities are actually represented in a real computer using base-2 or binary numbers.

First, let's look at normalization. Since binary numbers consist exclusively of 0s and 1s, a bonus occurs when they are normalized. That is, the bit to the left of the binary point will always be one! This means that this leading bit does not have to be stored. Hence, nonzero binary floating-point numbers can be expressed as

$$\pm(1+f)\times 2^e$$

where f = the *mantissa* (i.e., the fractional part of the significand). For example, if we normalized the binary number 1101.1, the result would be $1.1011\times(2)^{-3}$ or $(1+0.1011)\times 2^{-3}$.

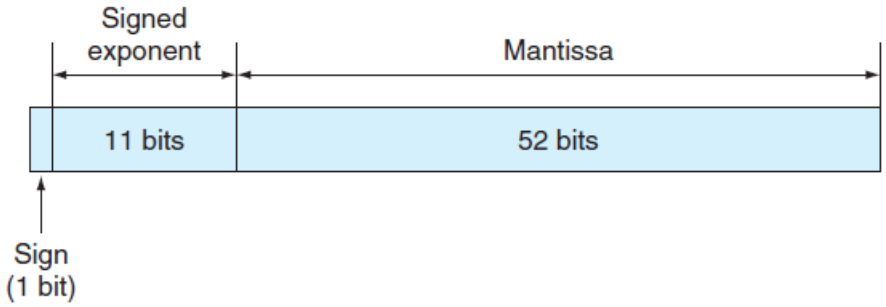


Figure 2.6: The manner in which a floating-point number is stored in an 8-byte word in IEEE doubleprecision format.

Thus, although the original number has five significant bits, we only have to store the four fractional bits: 0.1011.

By default, MATLAB has adopted the *IEEE double-precision format* in which eight bytes (64 bits) are used to represent floating-point numbers. As in Fig. 4.6, one bit is reserved for the number's sign. In a similar spirit to the way in which integers are stored, the exponent and its sign are stored in 11 bits. Finally, 52 bits are set aside for the mantissa. However, because of normalization, 53 bits can be stored.

Now, just as in Example 4.2, this means that the numbers will have a limited range and precision. However, because the IEEE format uses many more bits, the resulting number system can be used for practical purposes.

Range In a fashion similar to the way in which integers are stored, the 11 bits used for the exponent translates into a range from –1022 to 1023. The largest positive number can be represented in binary as

$$\text{Largest value} = +1.1111\ldots1111\times 2^{+1023}$$

where the 52 bits in the mantissa are all 1. Since the significant is approximately 2 (it is actually $2-2^{-52}$), the largest value is therefore $2^{1024}=1.7977\times 10^{308}$. In a similar fashion, the smallest positive number can be represented as

$$\text{Smallest value} = +1.0000\ldots0000\times 2^{-1022}$$

This value can be translated into a base-10 value of $2^{-1022}=2.2251\times 10^{-308}$

Precision. The 52 bits used for the mantissa correspond to about 15 to 16 base-10 digits. Thus, π would be expressed as

```
» format long
» pi
ans =
      3.14159265358979
```

Note that the machine epsilon is $2^{-52}=2.2204\times 10^{-16}$

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
```

Numbers occurring in computations that exceed this value create an overflow. In MATLAB they are set to infinity, `inf`. 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, in MATLAB, are set to zero. Finally, the `eps` function displays the machine epsilon:

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

2.2.2. 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 roundoff 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 roundoff 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 $1.557 + 0.04341$, the computer would express the numbers as $0.1557 \times 10^1 + 0.004341 \times 10^1$. Then the mantissas are added to give 0.160041×10^1. Now, because this hypothetical computer only carries a 4-digit mantissa, the excess number of digits get chopped off and the result is 0.1600×10^1. 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,

$$\begin{array}{rl} & 0.3641 \times 10^2 \\ - & 0.2686 \times 10^2 \\ \hline & 0.0955 \times 10^2 \end{array}$$

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 $0.9550 \times 10^1 = 9.550$. 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

$$\begin{array}{rl} & 0.7642 \times 10^3 \\ - & 0.7641 \times 10^3 \\ \hline & 0.0001 \times 10^3 \end{array}$$

which would be converted to $0.1000 \times 10^0 = 0.1000$. 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 roundoff error could be small, the cumulative effect over the course of a large computation can be significant. Avery 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 =
      0.99999999999991
```

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 0.99999999999991, the last entry will be exactly one as verified by

```
» s(10001)
ans =
      1
```

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,

$$\begin{array}{ll} 0.4000 & \times 10^4 \\ 0.0000001 & \times 10^4 \\ \hline 0.4000001 & \times 10^4 \end{array}$$

which is chopped to 0.4000×10^4 . 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 a series of mixed signs.

Inner Products. As should be clear from the last sections, some infinite series are particularly prone to roundoff 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

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

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

2.3. 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-difference equation of the form [Eq. (1.11)]

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

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.

2.3.1. 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 generate practical results.

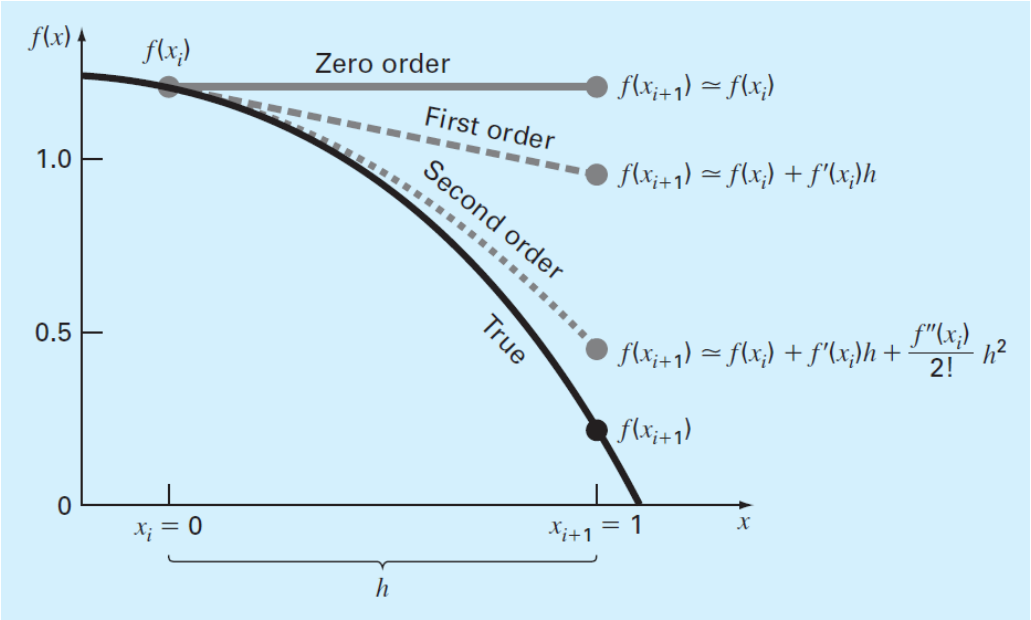


Figure 2.7: The approximation of $f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x = 1.2$ at $x = 1$ by zero-order, first-order, and second-order Taylor series expansions.

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. 4.7). 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

$$f(x_{i+1}) \cong f(x_i) \tag{4.9}$$

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 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

$$f(x_{i+1}) \cong f(x_i) + f'(x_i)h \tag{4.10}$$

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" downward 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

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

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

$$f''(x_{i+1}) \cong \frac{f'_f(x_i) - f'_b(x_i)}{\Delta x} \tag{4.12}$$

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}. Thus, it is a quadratic equation of the form

$$f(h) \cong a_2h^2 + a_1h + a_0$$

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

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

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f^{(3)}(x_i)}{3!}h^3 + \ldots + \frac{f^{(n)}(x_i)}{n!}h^n + R_n \tag{4.13}$$

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:

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

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

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 nth-order Taylor series expansion will be exact for an nth-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 will be demonstrated in Example 4.3. 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 often 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 2.3. Approximation of a Function with a Taylor Seres Expansion

Problem Statement. Use Taylor series expansions with $n = 0$ to 6 to approximate $f(x) = cosx$ 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 allows us to determine the correct value $f(\pi/3) = 0.5$. The zero-order approximation is [Eq. (4.9)]

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

which represents a percent relative error of

$$\varepsilon_t = \left|\frac{0.5 - 0.707106781}{0.5}\right| 100\% = 41.1\%$$

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

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

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

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

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 n	$f^{(n)}(x)$	$f(\pi/3)$	$\lvert\varepsilon_t\rvert$
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.

■

2.3.2. The Remainder for the Taylor Series Expansion

Before demonstrating how the Taylor series is actually used to estimate numerical errors, we must explain why we included the argument ξ in Eq. (4.14). To do this, we will use a simple, visually based explanation.
Suppose that we truncated the Taylor series expansion [Eq. (4.13)] after the zero-order term to yield

$$f(x_{i+1}) \cong f(x_i)$$

A visual depiction of this zero-order prediction is shown in Fig. 4.8. The remainder, or error, of this prediction, which is also shown in the illustration, consists of the infinite series of terms that were truncated

$$R_0 = f'(x_i)h + \frac{f''(x_1)}{2!}h^2 + \frac{f^{(3)}(x_i)}{3!}h^3 \ldots$$

It is obviously inconvenient to deal with the remainder in this infinite series format. One simplification might be to truncate the remainder itself, as in

$$R_0 \cong f'(x_i)h \tag{4.15}$$

Although, as stated in the previous section, lower-order derivatives usually account for a greater share of the remainder than the higher-order terms, this result is still inexact because of the neglected second- and higher-order terms. This

"inexactness" is implied by the approximate equality symbol (≅) employed in Eq. (4.15). An alternative simplification that transforms the approximation into an equivalence is based on a graphical insight. As in Fig. 4.9, the *derivative mean-value theorem* states that if a function $f(x)$ and its first derivative are continuous over an interval from x_i to x_{i+1}, then

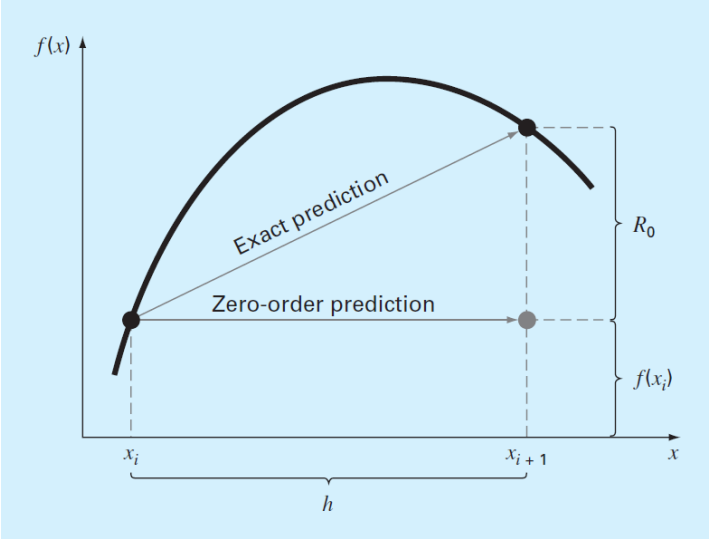


Figure 2.8: Graphical depiction of a zero-order Taylor series prediction and remainder.

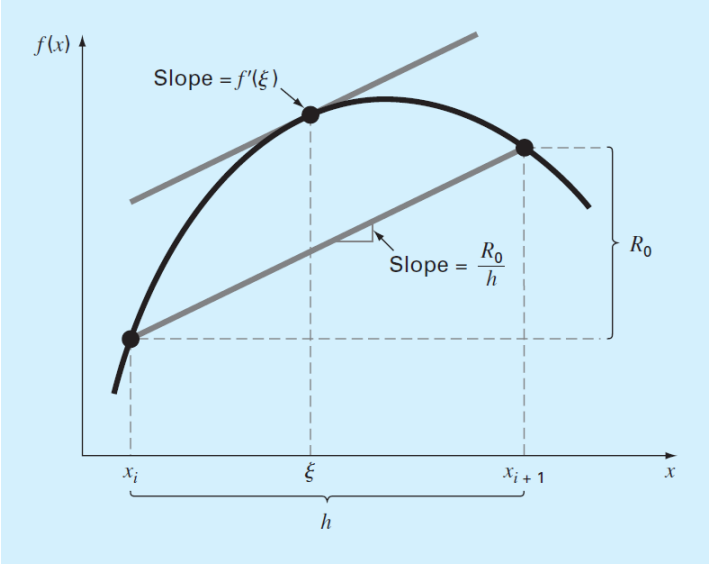


Figure 2.9: Graphical depiction of the derivative mean-value theorem.

there exists at least one point on the function that has a slope, designated by $f'(\xi)$, that is parallel to the line joining $f(x_i)$ and $f(x_{i+1})$. The parameter ξ marks the x value where this slope occurs (Fig. 4.9). A physical illustration of this theorem is that, if you travel between two points with an average velocity, there will be at least one moment during the course of the trip when you will be moving at that average velocity. By invoking this theorem, it is simple to realize that, as illustrated in Fig. 4.9, the slope $f'(\xi)$ is equal to the rise R_0 divided by the run h, or

$$f'(\xi) = \frac{R_0}{h}$$

which can be rearranged to give

$$R_0 = f'(\xi)h \tag{4.16}$$

Thus, we have derived the zero-order version of Eq. (4.14). The higher-order versions are merely a logical extension of the reasoning used to derive Eq. (4.16). The first-order version is

$$R_1 = \frac{f''(\xi)}{2!}h^2 \tag{4.17}$$

For this case, the value of ξ conforms to the x value corresponding to the second derivative that makes Eq. (4.17) exact. Similar higher-order versions can be developed from Eq. (4.14).

2.3.3. 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:

$$v(t_{i+1}) = v(t_i) + v'(t_i)(t_{i+1} - t_i) + \frac{v''(t_i)}{2!}(t_{i+1} - t_i)^2 + \ldots + R_n$$

Now let us truncate the series after the first derivative term:

$$v(t_{i+1}) = v(t_i) + v'(t_i)(t_{i+1} - t_i) + R_1 \tag{4.18}$$

Equation (4.18) can be solved for

$$v'(t_i) = \underbrace{\frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i}}_{\text{First-order approximation}} - \underbrace{\frac{R_1}{t_{i+1} - t_i}}_{\text{Truncation error}} \tag{4.19}$$

The first part of Eq. (4.19) 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 (4.19) yields

$$\frac{R_1}{t_{i+1} - t_i} = \frac{v''(\xi)}{2!}(t_{i+1} - t_i)$$

or

$$\frac{R_1}{t_{i+1} - t_i} = O(t_{i+1} - t_i)$$

Thus, the estimate of the derivative [Eq. (1.11) or the first part of Eq. (4.19)] 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.

2.3.4. Numerical Differentiation

Equation (4.19) is given a formal label in numerical methods—it is called a *finite difference*. It can be represented generally as

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} + O(x_{i+1} - x_i) \tag{4.20}$$

or

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h) \tag{4.21}$$

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. 4.10a).

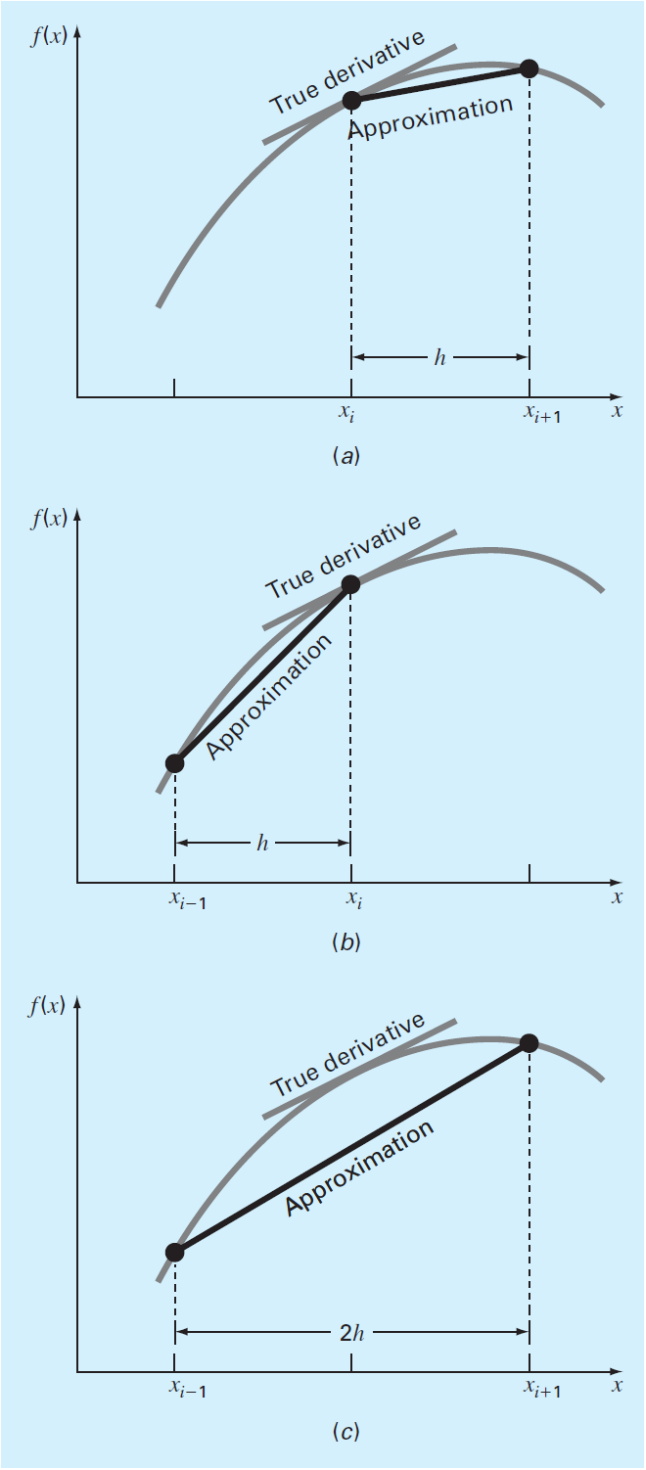


Figure 2.10: Graphical depiction of (a) forward, (b) backward, and (c) centered finite-difference approximations of the first derivative.

This forward 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. (4.19). The former utilizes values at x_{i-1} and x_i (Fig. 4.10b), whereas the latter uses values that are equally spaced around the point at which the derivative is estimated (Fig. 4.10c). 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

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 - \ldots \tag{4.22}$$

Truncating this equation after the first derivative and rearranging yields

$$f'(x_i) \cong \frac{f(x_i) - f(x_{i-1})}{h} \tag{4.23}$$

where the error is $O(h)$.

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:

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \dots \tag{4.24}$$

to yield

$$f(x_{i+1}) = f(x_{i-1}) + 2f'(x_i)h + 2\frac{f^{(3)}(x_i)}{3!}h^3 + \dots$$

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

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} - O(h^2) \tag{4.25}$$

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. 4.10c). For example, if we halve the step size using a forward or backward difference, we would approximately halve the truncation error, whereas for the central difference, the error would be quartered.

Example 2.4. Finite-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

$$x_{i-1} = 0 \qquad f(X_{i-1}) = 1.2$$

$$x_i = 0.5 \qquad f(x_i) = 0.925$$

$$x_{i+1} = 1.0 \qquad f(x_{i+1}) = 0.2$$

These values can be used to compute the forward difference [Eq. (4.21)],

$$f'(0.5) \cong \frac{0.2 - 0.925}{0.5} = -1.45 \qquad |\varepsilon_t| = 58.9\%$$

the backward difference [Eq. (4.23)],

$$f'(0.5) \cong \frac{0.925 - 1.2}{0.5} = -0.55 \qquad |\varepsilon_t| = 39.7\%$$

and the centered difference [Eq. (4.25)],

$$f'(0.5) \cong \frac{0.2-1.2}{1.0} = -1.0 \qquad |\varepsilon_t| = 9.6\%$$

For $h = 0.25$,

$$x_{i-1} = 0.25 \qquad f(x_{i-1}) = 1.10351563$$

$$x_i = 0.5 \qquad f(x_i) = 0.925$$

$$x_{i+1} = 0.75 \qquad f(x_{i+1}) = 0.63632813$$

which can be used to compute the forward difference,

$$f'(0.5) \cong \frac{0.63632813-0.925}{0.25} = -1.155 \qquad |\varepsilon_t| = 26.5\%$$

the backward difference,

$$f'(0.5) \cong \frac{0.925-1.10351563}{0.25} = -0.714 \qquad |\varepsilon_t| = 21.7\%$$

and the centered difference,

$$f'(0.5) \cong \frac{0.63632813-1.10351563}{0.5} = -0.934 \qquad |\varepsilon_t| = 2.4\%$$

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 $f(x_i)$:

$$f(x_{i+2}) = f(x_i) + f'(x_i)(2h) + \frac{f''(x_i)}{2!}(2h)^2 + \ldots \tag{4.26}$$

Equation (4.24) can be multiplied by 2 and subtracted from Eq. (4.26) to give

$$f(x_{i+2}) - 2f(x_{i+1}) = -f(x_i) + f''(x_i)h^2 + \ldots$$

which can be solved for

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + O(h) \tag{4.27}$$

This relationship is called the *second forward finite difference*. Similar manipulations can be employed to derive a backward version

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2} + O(h)$$

A centered difference approximation for the second derivative can be derived by adding Eqs. (4.22) and (4.24) and rearranging the result to give

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} + O(h^2)$$

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

$$f''(x_i) \cong \frac{\dfrac{f(x_{i+1}) - f(x_i)}{h} - \dfrac{f(x_i) - f(x_{i-1})}{h}}{h}$$

Thus, just as the second derivative is a derivative of a derivative, the second finite difference approximation is a difference of two first finite differences [recall Eq. (4.12)].

2.4. TOTAL NUMERICAL ERROR

The *total numerical error* is the summation of the truncation and roundoff errors. In general, the only way to minimize roundoff errors is to increase the number of significant figures of the computer. Further, we have noted that roundoff error may *increase* due to subtractive cancellation or due to an increase in the number of computations in an analysis. In contrast, Example 4.4 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 roundoff errors are *increased*.

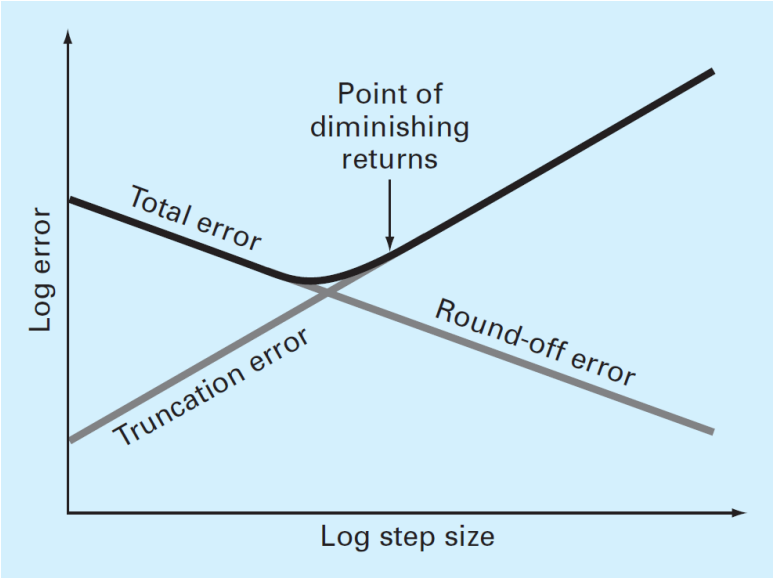


Figure 2.11: A graphical depiction of the trade-off between roundoff and truncation error that sometimes comes into play in the course of a numerical method. The point of diminishing returns is shown, where roundoff error begins to negate the benefits of step-size reduction.

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 roundoff error begins to dominate the solution and the total error grows! Thus, our remedy becomes our problem (Fig. 4.11). 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 roundoff errors without incurring the penalty of a large truncation error. If the total error is as shown in Fig. 4.11, the challenge is to identify the point of diminishing returns where roundoff error begins to negate the benefits of step-size reduction. When using MATLAB, such situations are relatively uncommon because of its 15- to 16- 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. We explore such a case in the following section.

2.4.1. Error Analysis of Numerical Differentiation

As described in Sec. 4.3.4, a centered difference approximation of the first derivative can be written as (Eq. 4.25)

$$\underbrace{f'(x_i)}_{\text{True value}} = \underbrace{\frac{f(x_{i+1})-f(x_{i-1})}{2h}}_{\text{Finite-difference approximation}} - \underbrace{\frac{f^{(3)}(\xi)}{6}h^2}_{\text{Truncation error}} \tag{4.28}$$

Thus, if the two function values in the numerator of the finite-difference approximation have no roundoff error, the only error is due to truncation.
However, because we are using digital computers, the function values do include roundoff error as in

$$f(x_{i-1}) = \tilde{f}(x_{i-1}) + e_{i-1}$$

$$f(x_{i+1}) = \tilde{f}(x_{i+1}) + e_{i+1}$$

where the $\tilde{f}$'s are the rounded function values and the e's are the associated roundoff errors. Substituting these values into Eq. (4.28) gives

$$\underset{\text{True value}}{f'(x_i)} = \underbrace{\frac{\tilde{f}(x_{i+1}) - \tilde{f}(x_{i-1})}{2h}}_{\text{Finite-difference approximation}} + \underbrace{\frac{e_{i+1} - e_{i-1}}{2h}}_{\text{Roundoff error}} - \underbrace{\frac{f^{(3)}(\xi)}{6}h^2}_{\text{Truncation error}}$$

We can see that the total error of the finite-difference approximation consists of a roundoff error that decreases with step size and a truncation error that increases with step size. Assuming that the absolute value of each component of the roundoff error has an upper bound of ε, the maximum possible value of the difference $e_{i+1} - e_{i-1}$ will be 2ε. Further, assume that the third derivative has a maximum absolute value of M. An upper bound on the absolute value of the total error can therefore be represented as

$$\text{Total error} = \left| f'(x_i) - \frac{\tilde{f}(x_{i+1}) - \tilde{f}(x_{i-1})}{2h} \right| \leq \frac{\varepsilon}{h} + \frac{h^2 M}{6} \tag{4.29}$$

An optimal step size can be determined by differentiating Eq. (4.29), setting the result equal to zero and solving for

$$h_{opt} = \sqrt[3]{\frac{3\varepsilon}{M}} \tag{4.30}$$

Example 2.5. Roundoff and Truncation Errors in Numerical Differentiation

Problem Statement. In Example 4.4, we used a centered difference approximation of $O(h^2)$ to estimate the first derivative of the following function at $x = 0.5$,

$$f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2$$

Perform the same computation starting with $h = 1$. Then progressively divide the step size by a factor of 10 to demonstrate how roundoff becomes dominant as the step size is reduced. Relate your results to Eq. (4.30). Recall that the true value of the derivative is -0.9125.

Solution. We can develop the following M-file to perform the computations and plot the results. Notice that we pass both the function and its analytical derivative as arguments:

```
function diffex(func,dfunc,x,n)
format long
dftrue=dfunc(x);
h=1;
H(1)=h;
D(1)=(func(x+h)-func(x-h))/(2*h);
E(1)=abs(dftrue-D(1));
for i=2:n
      h=h/10;
      H(i)=h;
      D(i)=(func(x+h)-func(x-h))/(2*h);
      E(i)=abs(dftrue-D(i));
end
L=[H' D' E']';
fprintf(' step size finite difference true error\n');
fprintf('%14.10f %16.14f %16.13f \n', L);
loglog(H, E), xlabel('Step Size'), ylabel('Error')
title('Plot of Error Versus Step Size')
format short
```

The M-file can then be run using the following commands:

```
» ff=@(x) -0.1*x^4-0.15*x^3-0.5*x^2-0.25*x+1.2;
» df=@(x) -0.4*x^3-0.45*x^2-x-0.25;
» diffex(ff,df,0.5,11)
```

```
  step size    finite difference     true error
1.0000000000  -1.26250000000000  0.3500000000000
0.1000000000  -0.91600000000000  0.0035000000000
0.0100000000  -0.91253500000000  0.0000350000000
0.0010000000  -0.91250035000001  0.0000003500000
0.0001000000  -0.91250000349985  0.0000000034998
0.0000100000  -0.91250000003318  0.0000000000332
0.0000010000  -0.91250000000542  0.0000000000054
0.0000001000  -0.91249999945031  0.0000000005497
0.0000000100  -0.91250000333609  0.0000000033361
0.0000000010  -0.91250001998944  0.0000000199894
0.0000000001  -0.91250007550059  0.0000000755006
```

As depicted in Fig. 4.12, the results are as expected. At first, roundoff is minimal and the estimate is dominated by truncation error. Hence, as in Eq. (4.29), the total error drops by a factor of 100 each time we divide the step by 10. However, starting at about $h = 0.0001$, we see roundoff error begin to creep in and erode the rate at which the error diminishes. A minimum error is reached at $h = 10^{-6}$. Beyond this point, the error increases as roundoff dominates.

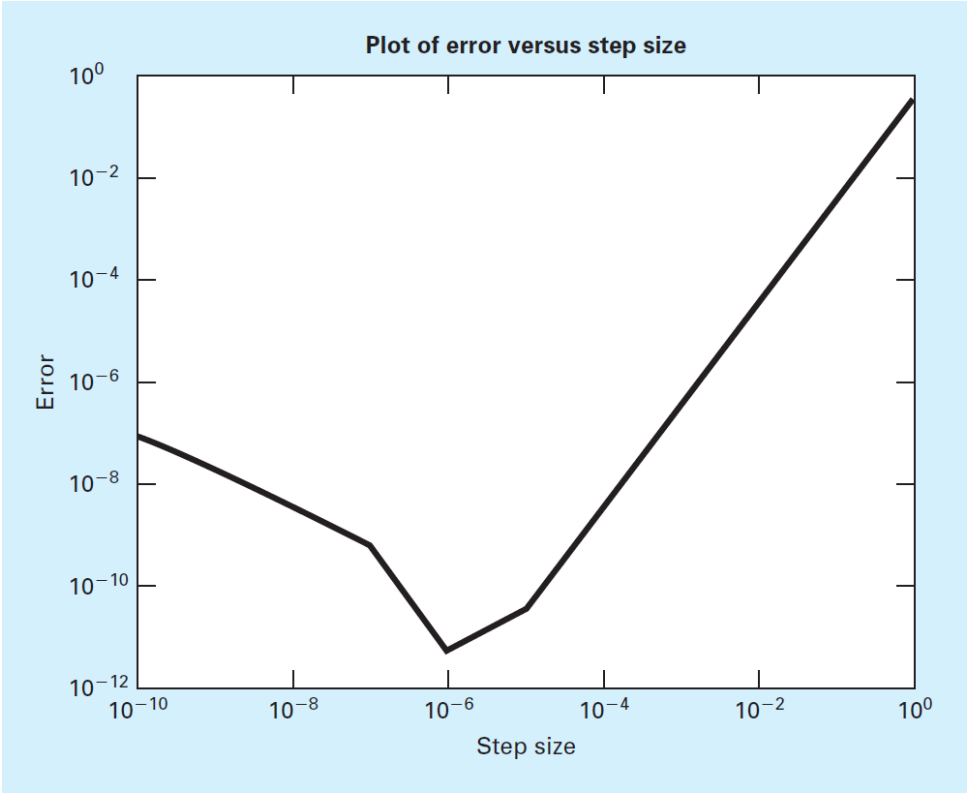
Because we are dealing with an easily differentiable function, we can also investigate whether these results are consistent with Eq. (4.30). First, we can estimate M by evaluating the function's third derivative as

$$M = \left| f^{(3)}(0.5) \right| = |-2.4(0.5) - 0.9| = 2.1$$

Because MATLAB has a precision of about 15 to 16 base-10 digits, a rough estimate of the upper bound on roundoff would be about $\varepsilon = 0.5 \times 10^{-16}$. Substituting these values into Eq. (4.30) gives

$$h_{opt} = \sqrt[3]{\frac{3(0.5 \times 10^{-16})}{2.1}} = 4.3 \times 10^{-6}$$

which is on the same order as the result of 1×10^{-6} obtained with MATLAB.



■

2.4.2. 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.

2.5. 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.

2.5.1. 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.

2.5.2. 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.

2.5.3. 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 wellchosen 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 Part Four.

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 "divide and average" method, an old-time method for approximating the square root of any positive number a, can be formulated as

$$x = \frac{x + a/x}{2}$$

Write a well-structured function to implement this algorithm based on the algorithm outlined in Fig. 4.2.

4.2 Convert the following base-2 numbers to base 10: **(a)** 1011001, **(b)** 0.01011, and **(c)** 110.01001.

4.3 Convert the following base-8 numbers to base 10: 61,565 and 2.71.

4.4 For computers, the machine epsilon ε can also be thought of as the smallest number that when added to one gives a number greater than 1. An algorithm based on this idea can be developed as
Step 1: Set $\varepsilon = 1$
Step 2: If $1 + \varepsilon$ is less than or equal to 1, then go to Step 5. Otherwise go to Step 3.
Step 3: $\varepsilon = \varepsilon/2$
Step 4: Return to Step 2
Step 5: $\varepsilon = 2 \times \varepsilon$

Write your own M-file based on this algorithm to determine the machine epsilon. Validate the result by comparing it with the value computed with the built-in function `eps`.

4.5 In a fashion similar to Prob. 4.4, develop your own M-file to determine the smallest positive real number used in MATLAB. Base your algorithm on the notion that your computer will be unable to reliably distinguish between zero and a quantity that is smaller than this number. Note that the result you obtain will differ from the value computed with `realmin`.Challenge question: Investigate the results by taking the base-2 logarithm of the number generated by your code and those obtained with `realmin`.

4.6 Although it is not commonly used, MATLAB allows numbers to be expressed in single precision. Each value is stored in 4 bytes with 1 bit for the sign, 23 bits for the mantissa, and 8 bits for the signed exponent. Determine the smallest and largest positive floating-point numbers as well as the machine epsilon for single precision representation. Note that the exponents range from -126 to 127.

4.7 For the hypothetical base-10 computer in Example 4.2, prove that the machine epsilon is 0.05.

4.8 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.

4.9 (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)**.

4.10 The following infinite series can be used to approximate e^x:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

(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.

4.11 The Maclaurin series expansion for $cosx$ is

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

Starting with the simplest version, $cosx = 1$, 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.

4.12 Perform the same computation as in Prob. 4.11, but use the Maclaurin series expansion for the $sinx$ to estimate $sin(\pi/4)$.

$$sinx = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

4.13 Use zero- through third-order Taylor series expansions to predict $f(3)$ 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.

4.14 Prove that Eq. (4.11) is exact for all values of x if $f(x) = ax^2 + bx + c$

4.15 Use zero- through fourth-order Taylor series expansions to predict $f(2)$ 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.

4.16 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. 4.13. 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.

4.17 Use a centered difference approximation of $O(h^2)$ to estimate the second derivative of the function examined in Prob. 4.13. 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.

4.18 If $|x| < 1$ it is known that

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \ldots$$

Repeat Prob. 4.11 for this series for $x = 0.1$.

4.19 To calculate a planet's space coordinates, we have to solve the function

$$f(x) = x - 1 - 0.5sinx$$

Let the base point be $a = x_i = \pi/2$ 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.)

4.20 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.

4.21 Derive Eq. (4.30).

4.22 Repeat Example 4.5, but for $f(x) = cos(x)$ at $x = \pi/6$.

4.23 Repeat Example 4.5, but for the forward divided difference (Eq. 4.21).

4.24 One common instance where subtractive cancellation occurs involves finding the roots of a parabola, $ax^2 + bx + c$, with the quadratic formula:

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

For cases where $b^2 >> 4ac$, the difference in the numerator can be very small and roundoff errors can occur. In such cases, an alternative formulation can be used to minimize subtractive cancellation:

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

Use 5-digit arithmetic with chopping to determine the roots of the following equation with both versions of the quadratic formula.

$$x^2 - 5000.002x + 10$$

Part II

Roots and Optimization

2.6. OVERVIEW

Years ago, you learned to use the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{PT2.1}$$

to solve

$$f(x) = ax^2 + bx + c = 0 \tag{PT2.2}$$

The values calculated with Eq. (PT2.1) are called the "roots" of Eq. (PT2.2). They represent the values of x that make Eq. (PT2.2) equal to zero. For this reason, roots are sometimes called the *zeros* of the equation.

Although the quadratic formula is handy for solving Eq. (PT2.2), 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. (PT2.1). 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.

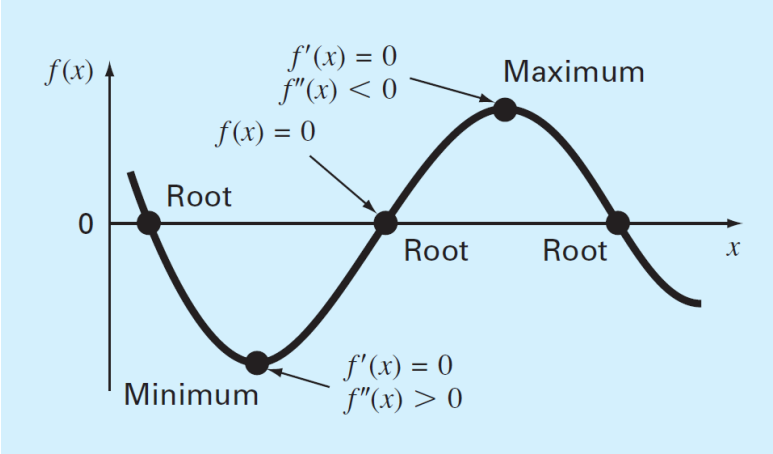


Figure 2.12: A function of a single variable illustrating the difference between roots and optima.

Such haphazard methods are obviously inefficient and inadequate for the requirements of engineering and science 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.

Besides roots, another feature of interest to engineers and scientists are a function's minimum and maximum values. The determination of such optimal values is referred to as *optimization*. As you learned in calculus, such solutions can be obtained analytically by determining the value at which the function is flat; that is, where its derivative is zero. Although such analytical solutions are sometimes feasible, most practical optimization problems require numerical, computer solutions. From a numerical standpoint, such optimization methods are similar in spirit to the root-location methods we just discussed. That is, both involve guessing and searching for a location on a function. The fundamental difference between the two types of problems is illustrated in Figure PT2.1. Root location involves searching for the location where the function equals zero. In contrast, optimization involves searching for the function's extreme points.

2.7. PART ORGANIZATION

The first two chapters in this part are devoted to root location. *Chapter 5* focuses on *bracketing methods* for finding roots. These methods start with guesses that bracket, or contain, the root and then systematically reduce the width of the bracket. Two specific methods are covered: *bisection* and *false position*. Graphical methods are used to provide visual insight into

the techniques. Error formulations are developed to help you determine how much computational effort is required to estimate the root to a prespecified level of precision.

Chapter 6 covers open methods. These methods also involve systematic trial-and-error iterations but do not require that the initial guesses bracket the root. We will discover that these methods are usually more computationally efficient than bracketing methods but that they do not always work. We illustrate several open methods including the *fixed-point iteration, Newton-Raphson*, and *secant* methods.

Following the description of these individual open methods, we then discuss a hybrid approach called *Brent's root-finding* method that exhibits the reliability of the bracketing methods while exploiting the speed of the open methods. As such, it forms the basis for MATLAB's root-finding function, `fzero`. After illustrating how `fzero` can be used for engineering and scientific problems solving, Chap. 6 ends with a brief discussion of special methods devoted to finding the roots of *polynomials*. In particular, we describe MATLAB's excellent built-in capabilities for this task.

Chapter 7 deals with *optimization*. First, we describe two bracketing methods, *goldensection search* and *parabolic interpolation*, for finding the optima of a function of a single variable. Then, we discuss a robust, hybrid approach that combines golden-section search and quadratic interpolation. This approach, which again is attributed to Brent, forms the basis for MATLAB's one-dimensional root-finding function: `fminbnd`. After describing and illustrating `fminbnd`, the last part of the chapter provides a brief description of optimization of multidimensional functions. The emphasis is on describing and illustrating the use of MATLAB's capability in this area: the `fminsearch` function. Finally, the chapter ends with an example of how MATLAB can be employed to solve optimization problems in engineering and science.

Chapter 3

Roots: 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:

$$v(t) = \sqrt{\frac{gm}{c_d}}\tanh\left(\sqrt{\frac{gc_d}{m}}t\right) \tag{5.1}$$

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:

$$f(m) = \sqrt{\frac{gm}{c_d}}\tanh\left(\sqrt{\frac{gc_d}{m}}t\right) - v(t) \tag{5.2}$$

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.

3.1. ROOTS IN ENGINEERING AND SCIENCE

Although they arise in other problem contexts, roots of equations frequently occur in the area of 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 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*.

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)

This represents a real dilemma, because many 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.

3.2. 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 3.1. 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 $9.81 m/s^2$.

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

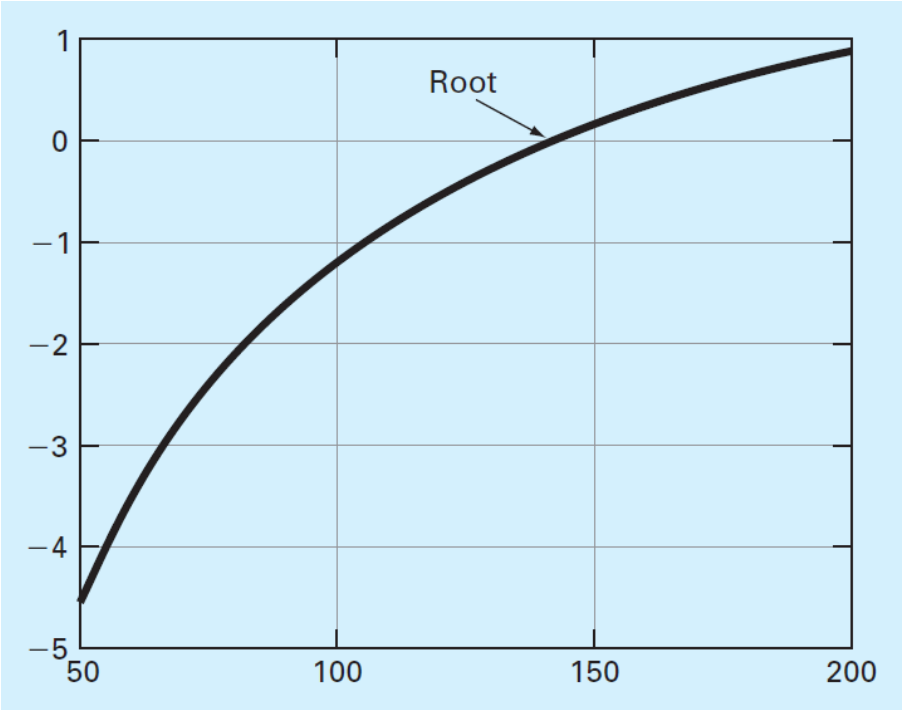
```
» sqrt(g*145/cd)*tanh(sqrt(g*cd/145)*t)-v
ans =
      0.0456
```

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 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 5.1b depicts the case where a single root is bracketed by negative and positive values of f (x). However, Fig. 5.1d, 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. 5.1a and c, $iff(x_l)$ 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. 5.2a) and discontinuous functions (Fig. 5.2b) 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, 2010).

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.

3.3. 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.

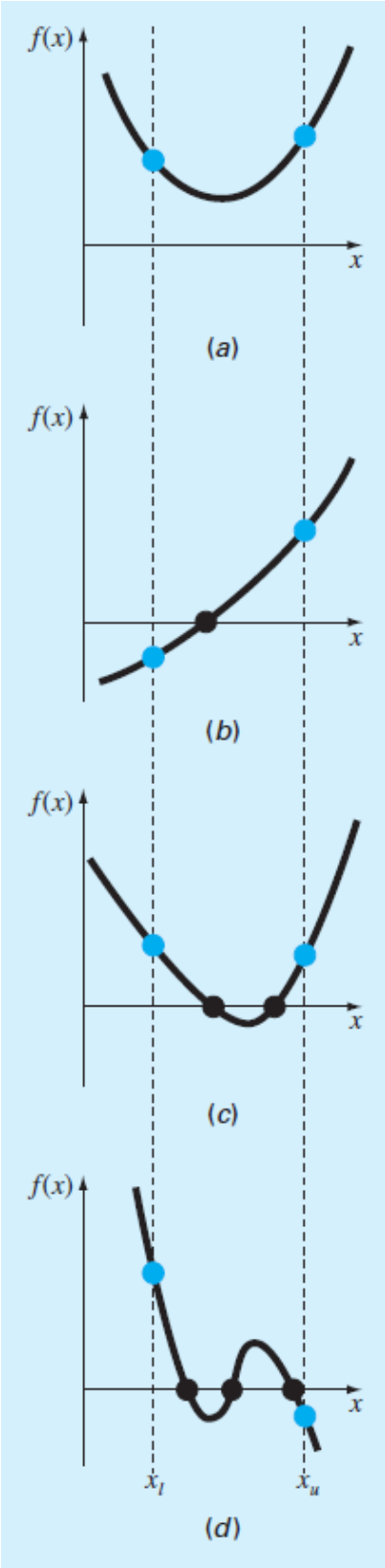


Figure 3.1: Illustration of a number of general ways that a root may occur in an interval prescribed by a lower bound x_l and an upper bound x_u. Parts (a) and (c) indicate that if both $f(x_l)$ and $f(x_u)$ have the same sign, either there will be no roots or there will be an even number of roots within the interval. Parts (b) 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.

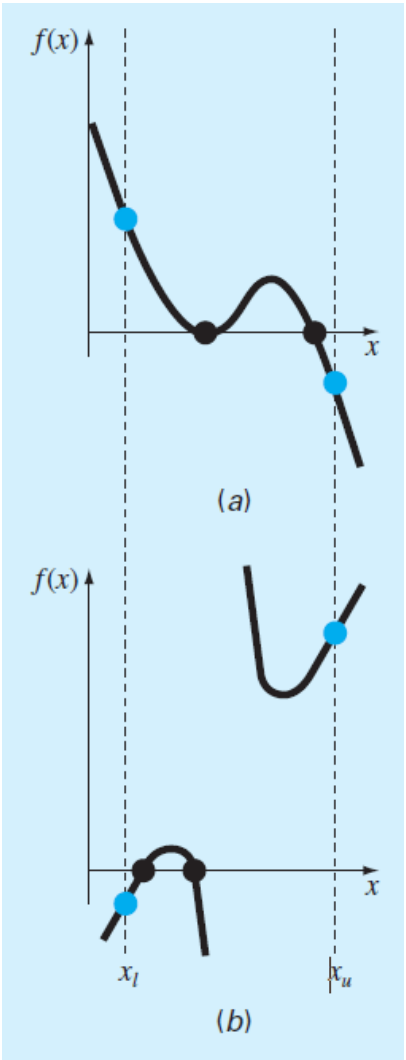


Figure 3.2: Illustration of some exceptions to the general cases depicted in Fig. 5.1. (a) Multiple roots that occur when the function is tangential to the x axis. For this case, although the end points are of opposite signs, there are an even number of axis interceptions for the interval. (b) Discontinuous functions where end points of opposite sign bracket an even number of roots. Special strategies are required for determining the roots for these cases.

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.

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.

3.3.1. 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,

$$f(x_l)f(x_u) < 0 \tag{5.3}$$

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 developed[1] 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 lower bounds are stored in an array `xb`.

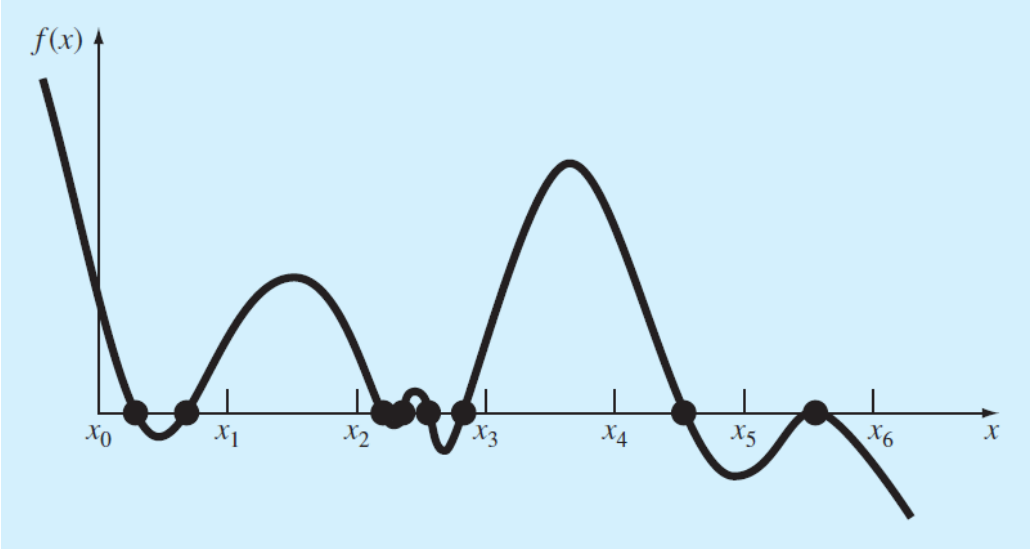


Figure 3.3: Cases where roots could be missed because the incremental length of the search procedure is too large. Note that the last root on the right is multiple and would be missed regardless of the increment length.

[1]This function is a modified version of an M-file originally presented by Recktenwald (2000).

```
function xb = incsearch(func,xmin,xmax,ns)
% incsearch: incremental search root locator
%    xb = 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 = number of subintervals (default = 50)
% 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 < 3, error('at least 3 arguments required'), end
if nargin < 4, ns = 50; end %if ns blank set to 50
% Incremental search
x = linspace(xmin,xmax,ns);
f = 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 3.4: An M-file to implement an incremental search.

Example 3.2. Incremental Search

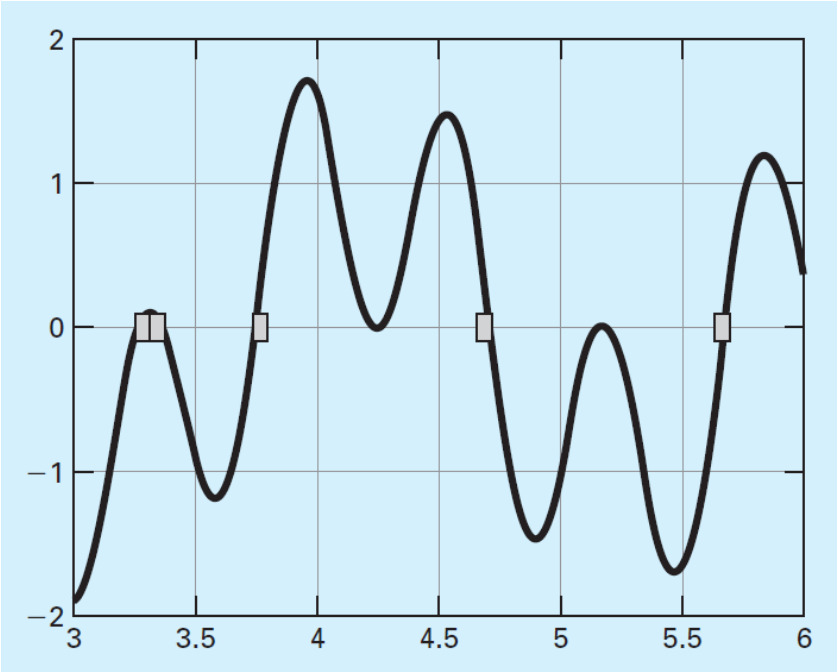
Problem Statement. Use the M-file incsearch (Fig. 5.4) to identify brackets within the interval [3, 6] for the function:

$$f(x) = sin(10x) + cos(3x) \tag{5.4}$$

Solution. The MATLAB session using the default number of intervals (50) is

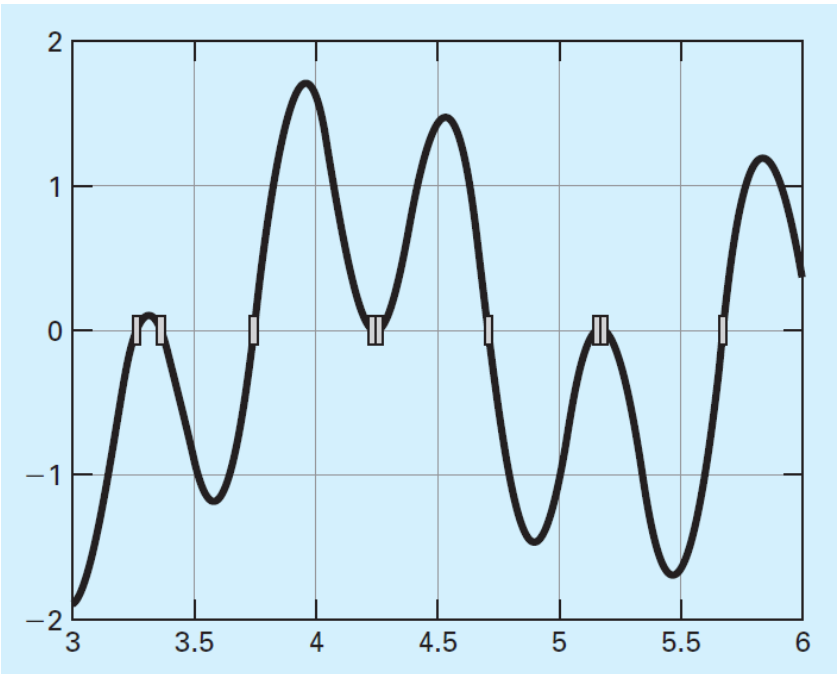
```
» incsearch(@(x) sin(10*x)+cos(3*x),3,6)
number of brackets:
      5
ans =
      3.2449       3.3061
      3.3061       3.3061
      3.7347       3.7959
      4.6531       4.7143
      5.6327       5.6939
```

A plot of Eq. (5.4) 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 $x \cong 4.25$ 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(@(x) sin(10*x)+cos(3*x),3,6,100)
number of brackets:
      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. ■

3.4. 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 3.3. The Bisection Method

Problem Statement. Use bisection to solve the same problem approached graphically in Example 5.1.

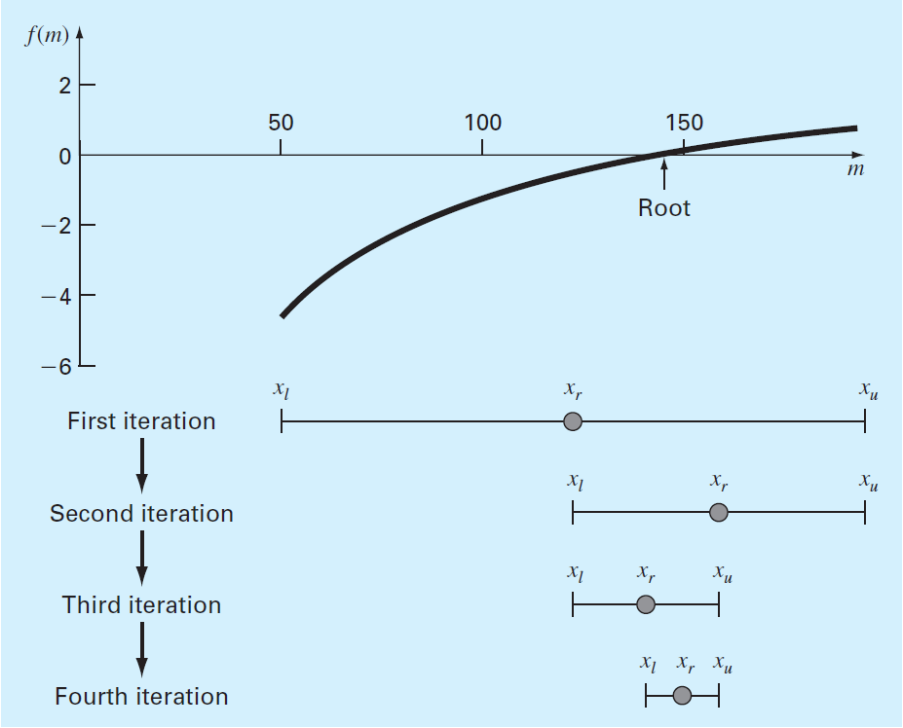


Figure 3.5: A graphical depiction of the bisection method. This plot corresponds to the first four iterations from Example 5.3.

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. 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

$$|\varepsilon_t| = \left|\frac{142.7376-125}{142.7376}\right| \times 100\% = 12.43\%$$

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 $x_u = 200$. Arevised root estimate can then be calculated as

$$x_r = \frac{125+200}{2} = 162.5$$

which represents a true percent error of $|\varepsilon_t| = 13.85\%$. 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 $\varepsilon_t = 0.709\%$. 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)]

$$|\varepsilon_a| = \left|\frac{x_r^{new} - x_r^{old}}{x_r^{new}}\right| 100\% \tag{5.5}$$

where x_r^{new} is the new 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 3.4. Error Estimates for Bisection

Problem Statement. Continue Example 5.3 until the approximate error falls below a stopping criterion of $\varepsilon_s = 0.5\%$. 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

$$|\varepsilon_a| = \left|\frac{162.5-125}{162.5}\right| 100\% = 23.08\%$$

Recall that the true percent relative error for the root estimate of 162.5 was 13.85%. Therefore, $|\varepsilon_a|$ is greater than ε_t. This behavior is manifested for the other iterations:

Iteration	x_l	x_u	x_r	$\|\varepsilon_a\|$ (%)	$\|\varepsilon_t\|$ (%)
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 $|\varepsilon_a|$ finally falls below $\varepsilon_s = 0.5\%$, 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.

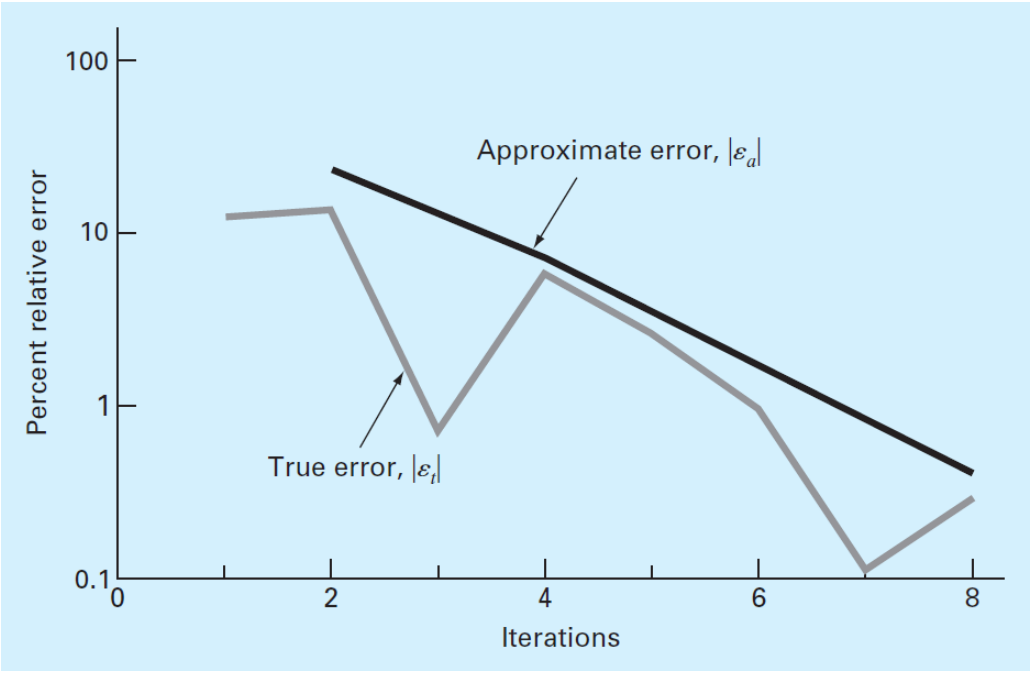


Figure 3.6: Errors for the bisection method. True and approximate errors are plotted versus the number of iterations.

■

Although the approximate error does not provide an exact estimate of the true error, Fig. 5.6 suggests that $|\varepsilon_a|$ 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 $\pm\Delta x/2$ 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

$$E_a^0 = x_u^0 - x_l^0 = \Delta x^0$$

where the superscript designates the iteration. Hence, before starting the method we are at the "zero iteration". After the first iteration, the error becomes

$$E_a^1 = \frac{\Delta x^0}{2}$$

Because each succeeding iteration halves the error, a general formula relating the error and the number of iterations n is

$$E_a^n = \frac{\Delta x^0}{2^n}$$

If $E_{a,d}$ is the desired error, this equation can be solved for[2]

$$n = \frac{log(\Delta x^0/E_{a,d})}{log2} = log_2\left(\frac{\Delta x^0}{E_{a,d}}\right) \tag{5.6}$$

Let's test the formula. For Example 5.4, the initial interval was $\Delta x = 200 - 50 = 150$. 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

$$n = log_2(150/0.5859) = 8$$

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

3.4.1. MATLAB M-file: bisect

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 (`es`) and maximum iterations (`maxit`) can be entered. The function first checks whether there are sufficient arguments and if the initial guesses bracket a sign change. If not, an error message is displayed and the function is terminated. It also assigns 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`.

We can employ this function to solve the problem posed at the beginning of the chapter. Recall that you need to determine the mass at which a bungee jumper's free-fall velocity exceeds 36 m/s after 4 s of free fall given a drag coefficient of 0.25 kg/m. Thus, you have to find the root of

$$f(m) = \sqrt{\frac{9.81m}{0.25}}\tanh\left(\sqrt{\frac{9.81(0.25)}{m}}4\right) - 36$$

In Example 5.1 we generated a plot of this function versus mass and estimated that the root fell between 140 and 150 kg. The `bisect` function from Fig. 5.7 can be used to determine the root as

```
» fm=@(m) sqrt(9.81*m/0.25)*tanh(sqrt(9.81*0.25/m)*4)-36;
» [mass fx ea iter]=bisect(fm,40,200)
mass =
      142.74
fx =
      4.6089e-007
ea =
      5.345e-005
iter =
      21
```

Thus, a result of m = 142.74 kg is obtained after 21 iterations with an approximate relative error of $\varepsilon_a = 0.00005345\%$, and a function value close to zero.

[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, $log_b(x) = log(x)/log(b)$.

```
function [root,fx,ea,iter]=bisect(func,xl,xu,es,maxit,varargin)
% bisect: root location zeroes
%    [root,fx,ea,iter]=bisect(func,xl,xu,es,maxit,p1,p2,...):
%       uses bisection method to find the root of func
% input:
%    func = name of function
%    xl, xu = lower and upper guesses
%    es = desired relative error (default = 0.0001%)
%    maxit = maximum allowable iterations (default = 50)
%    p1,p2,... = additional parameters used by func
% output:
%    root = real root
%    fx = function value at root
%    ea = approximate relative error (%)
%    iter = number of iterations
if nargin<3,error('at least 3 input arguments required'),end
test = func(xl,varargin{:})*func(xu,varargin{:});
if test>0,error('no sign change'),end
if nargin<4|isempty(es), es=0.0001;end
if nargin<5|isempty(maxit), maxit=50;end
iter = 0; xr = xl; ea = 100;
while (1)
   xrold = xr;
   xr = (xl + xu)/2;
   iter = iter + 1;
   if xr ~= 0,ea = abs((xr - xrold)/xr) * 100;end
   test = func(xl,varargin{:})*func(xr,varargin{:});
   if test < 0
      xu = xr;
   elseif test > 0
      xl = xr;
   else
      ea = 0;
   end
   if ea <= es | iter >= maxit,break,end
end
root = xr; fx = func(xr, varargin{:});
```

Figure 3.7: An M-file to implement the bisection method.

3.5. FALSE POSITIVE

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, 2010, for details),

$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)} \tag{5.7}$$

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.

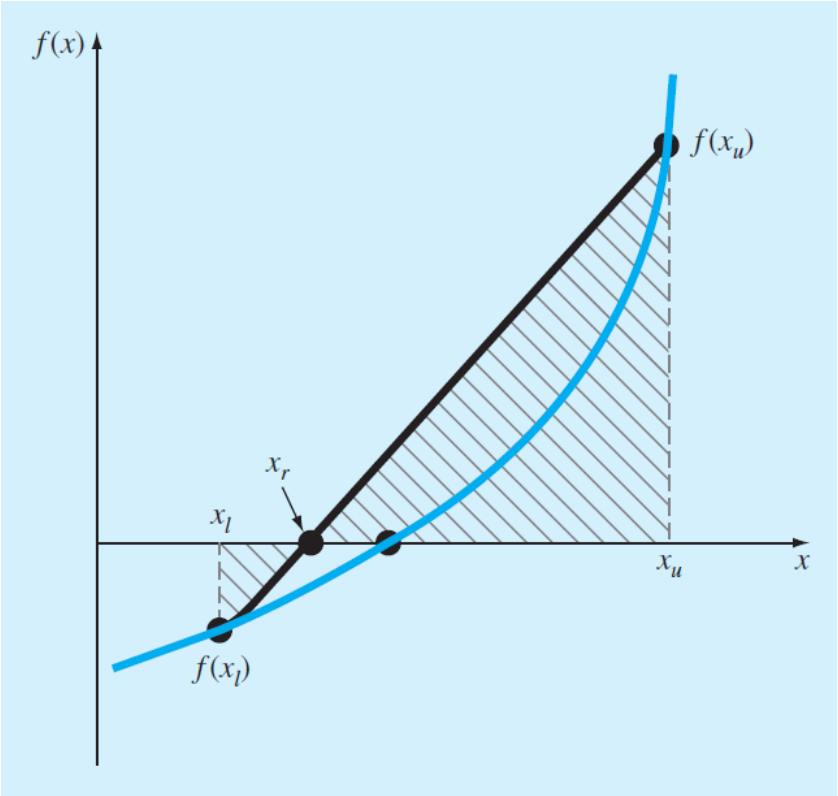


Figure 3.8: False position.

Example 3.5. 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:

$$x_l = 50 \qquad f(x_l)$$

$$x_u = 200 \qquad f(x_u) = 0.860291$$

$$x_r = 200 - \frac{0.860291(50 - 200)}{-4.579387 - 0.860291} = 176.2773$$

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, $x_u = 176.2773$.

$$x_l = 50 \qquad f(x_l) = -4.579387$$

$$x_u = 176.2773 \qquad f(x_u) = 0.566174$$

$$x_r = 176.2773 - \frac{0.566174(50 - 176.2773)}{-4.579387 - 0.566174} = 162.3828$$

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 3.6. 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	x_l	x_u	x_r	ε_a (%)	ε_t (%)
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_a (%)	ε_t (%)
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 should be much closer to x_l than to x_u (recall Fig. 5.8). Because of the shape of the present function, the opposite is true.

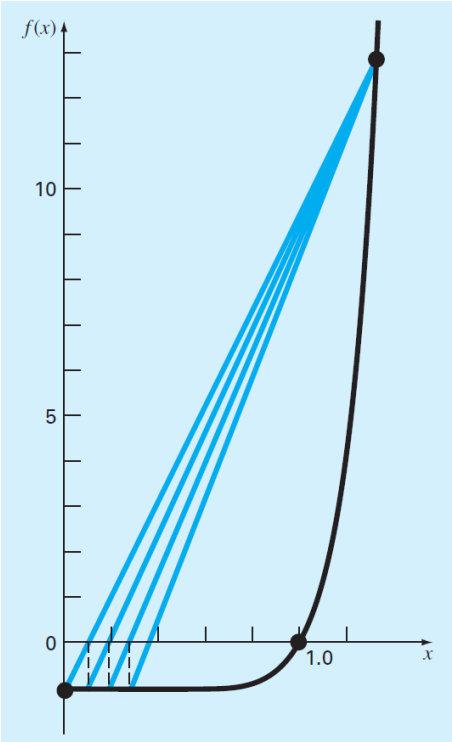


Figure 3.9: Plot of $f(x) = x^{10} - 1$, illustrating slow convergence of the false-position method.

■

The foregoing example illustrates that blanket generalizations regarding rootlocation 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 onesidedness. 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, 2010).

3.6. CASE STUDY: GREENHOUSE GASES AND RAINWATER

Background. It is well documented that the atmospheric levels of several so-called "greenhouse" gases have been increasing over the past 50 years. For example, Fig. 5.10 shows data for the partial pressure of carbon dioxide (CO_2) collected at Mauna Loa, Hawaii from 1958 through 2008. The trend in these data can be nicely fit with a quadratic polynomial, [3]

$$p_{CO_2} = 0.012226(t-1983)^2 + 1.418542(t-1983) + 342.38309$$

where $p_{CO_2} = CO_2$ partial pressure (ppm). These data indicate that levels have increased a little over 22% over the period from 315 to 386 ppm. One question that we can address is how this trend is affecting the pH of rainwater. Outside of urban and industrial areas, it is well documented that carbon dioxide is the primary determinant of the pH of the rain. pH is the measure of the activity of hydrogen ions and, therefore, its acidity or alkalinity. For dilute aqueous solutions, it can be computed as

$$pH = -log_{10}[H^+] \tag{5.8}$$

where $[H^+]$ is the molar concentration of hydrogen ions.
The following five equations govern the chemistry of rainwater:

$$K_1 = 10^6 \frac{[H^+][HCO_3^-]}{K_H p_{CO_2}} \tag{5.9}$$

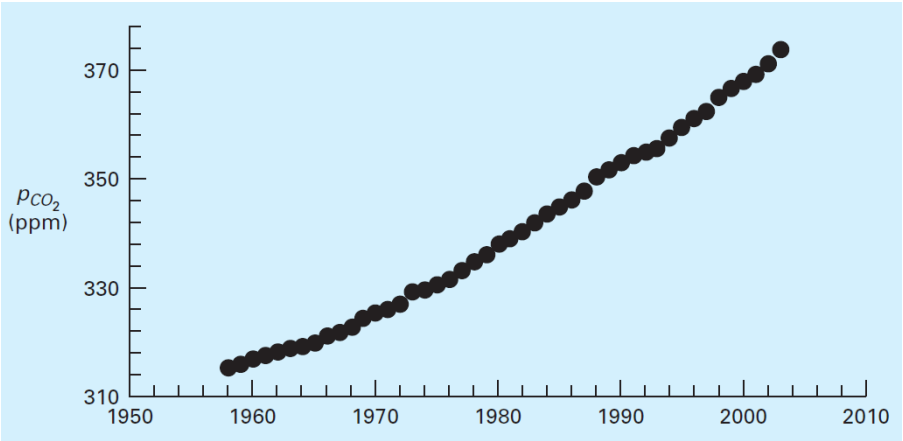


Figure 3.10: Average annual partial pressures of atmospheric carbon dioxide (ppm) measured at Mauna Loa, Hawaii.

$$K_2 = \frac{[H^+][CO_3^{-2}]}{HCO_3^-} \tag{5.10}$$

$$K_w = [H^+][OH^-] \tag{5.11}$$

$$c_T = \frac{K_H p_{CO_2}}{10^6} + [HCO_3^-] + [CO_3^{-2}] \tag{5.12}$$

$$0 = [HCO_3^-] + 2[CO_3^{-2}] + [OH^-] - [H^+] \tag{5.13}$$

where K_H = Henry's constant, and K_1, K_2, and K_w are equilibrium coefficients. The five unknowns are c_T = total inorganic carbon, $[HCO_3^-]$ = bicarbonate, $[CO_3^{-2}]$ = carbonate, $[H^+]$ = hydrogen ion, and $[OH^-]$ = hydroxyl ion. Notice how the partial pressure of CO_2 shows up in Eqs. (5.9) and (5.12).

[3] In Part Four, we will learn how to determine such polynomials.

Use these equations to compute the pH of rainwater given that $K_H = 10^{-1.46}$, $K_1 = 10^{-6.3}$, $K_2 = 10^{-10.3}$, and $K_w = 10^{-14}$. Compare the results in 1958 when the p_{CO_2} was 315 and in 2008 when it was 386 ppm. When selecting a numerical method for your computation, consider the following:

- You know with certainty that the pH of rain in pristine areas always falls between 2 and 12.
- You also know that pH can only be measured to two places of decimal precision.

Solution. There are a variety of ways to solve this system of five equations. One way is to eliminate unknowns by combining them to produce a single function that only depends on $[H^+]$. To do this, first solve Eqs. (5.9) and (5.10) for

$$[HCO_3^-] = \frac{K_1}{10^6[H^+]} K_H p_{CO_2} \tag{5.14}$$

$$[CO_3^{-2}] = \frac{K_2[HCO_3^-]}{[H^+]} \tag{5.15}$$

Substitute Eq. (5.14) into (5.15)

$$[CO_3^{-2}] = \frac{K_2 K_1}{10^6[H^+]^2} K_H p_{CO_2} \tag{5.16}$$

Equations (5.14) and (5.16) can be substituted along with Eq. (5.11) into Eq. (5.13) to give

$$0 = \frac{K_1}{10^6[H^+]} K_H p_{CO_2} + 2\frac{K_2 K_1}{10^6[H^+]^2} K_H p_{CO_2} + \frac{K_w}{[H^+]} - [H^+] \tag{5.17}$$

Although it might not be immediately apparent, this result is a third-order polynomial in $[H^+]$. Thus, its root can be used to compute the pH of the rainwater.

Now we must decide which numerical method to employ to obtain the solution. There are two reasons why bisection would be a good choice. First, the fact that the pH always falls within the range from 2 to 12, provides us with two good initial guesses. Second, because the pH can only be measured to two decimal places of precision, we will be satisfied with an absolute error of $E_{a,d} = \pm 0.005$. Remember that given an initial bracket and the desired error, we can compute the number of iteration *a priori*. Substituting the present values into Eq. (5.6) gives

```
» dx=12-2;
» Ead=0.005;
» n=log2(dx/Ead)
n =
      10.9658
```

Eleven iterations of bisection will produce the desired precision.
Before implementing bisection, we must first express Eq. (5.17) as a function. Because it is relatively complicated, we will store it as an M-file:

```
function f = fpH(pH,pCO2)
K1=10^-6.3;K2=10^-10.3;Kw=10^-14;
KH=10^-1.46;
H=10^-pH;
f=K1/(1e6*H)*KH*pCO2+2*K2*K1/(1e6*H)*KH*pCO2+Kw/H-H;
```

We can then use the M-file from Fig. 5.7 to obtain the solution. Notice how we have set the value of the desired relative error ($\varepsilon_a = 1 \times 10^-8$) at a very low level so that the iteration limit (`maxit`) is reached first so that exactly 11 iterations are implemented

```
» [pH1958 fx ea iter]=bisect(@fpH,2,12,1e-8,11,315)
pH1958 =
      5.6279
fx =
      -2.7163e-008
ea =
      0.08676
iter =
      11
```

Thus, the pH is computed as 5.6279 with a relative error of 0.0868%. We can be confident that the rounded result of 5.63 is correct to two decimal places. This can be verified by performing another run with more iterations. For example, setting

maxit to 50 yields

```
» [pH1958 fx ea iter] = bisect(@fpH,2,12,1e-8,50,315)
pH1958 =
      5.6304
fx =
      1.615e-015
ea =
      5.169e-009
iter =
35
```

For 2008, the result is

```
» [pH2008 ea iter]=bisect(@fpH,2,12,1e-8,50,386)
pH2008 =
      5.5864
fx =
      3.2926e-015
ea =
      5.2098e-009
iter =
      35
```

Interestingly, the results indicate that the 22.5% rise in atmospheric CO_2 levels has produced only a 0.78% drop in pH. Although this is certainly true, remember that the pH represents a logarithmic scale as defined by Eq. (5.8). Consequently, a unit drop in pH represents an order-of-magnitude (i.e., a 10-fold) increase in the hydrogen ion. The concentration can be computed as $[H^+] = 10^{-pH}$ and its percent change can be calculated as.

```
» ((10^-pH2008-10^-pH1958)/10^-pH1958)*100
ans =
      10.6791
```

Therefore, the hydrogen ion concentration has increased about 10.7%.

There is quite a lot of controversy related to the meaning of the greenhouse gas trends. Most of this debate focuses on whether the increases are contributing to global warming. However, regardless of the ultimate implications, it is sobering to realize that something as large as our atmosphere has changed so much over a relatively short time period. This case study illustrates how numerical methods and MATLAB can be employed to analyze and interpret such trends. Over the coming years, engineers and scientists can hopefully use such tools to gain increased understanding of such phenomena and help rationalize the debate over their ramifications.

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) 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%.

5.6 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 Îţa is less than $\varepsilon_s = 2\%$.

5.7 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)**.

5.8 The saturation concentration of dissolved oxygen in

freshwater can be calculated with the equation

$$\ln o_{sf} = -139.34411 + \frac{1.575701 \times 10^5}{T_a} - \frac{6.642308 \times 10^7}{T_a^2} + \frac{1.243800 \times 10^{10}}{T_a^3} - \frac{8.621949 \times 10^{11}}{T_a^4}$$

where o_{sf} = the saturation concentration of dissolved oxygen in freshwater at 1 atm (mgL^{-1}); and T_a = absolute temperature (K). Remember that $T_a = T + 273.15$, where T = temperature (°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 °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?
(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 o_{sf} = 8, 10 and 14 mg/L. Check your results.

5.9 A beam is loaded as shown in Fig. P5.9. Use the bisection method to solve for the position inside the beam where there is no moment.

5.10 Water is flowing in a trapezoidal channel at a rate of $Q = 20m^3/s$. The critical depth y for such a channel must satisfy the equation

$$0 = 1 - \frac{Q^2}{gA_c^3}B$$

where $g = 9.81m/s^2$, 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

$$B = 3 + y$$

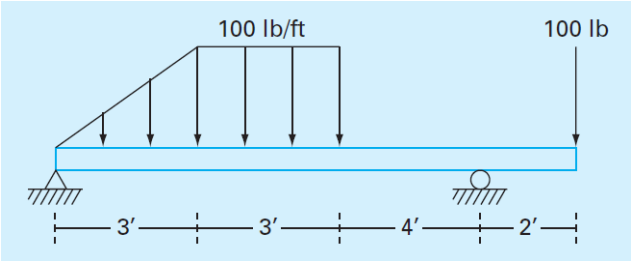


Figure P5.9

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.

5.11 The Michaelis-Menten model describes the kinetics of enzyme mediated reactions:

$$\frac{dS}{dt} = -v_m \frac{S}{k_s + S}$$

where S = substrate concentration (moles/L), v_m = maximum uptake rate (moles/L/d), and k_s = the half-saturation constant, which is the substrate level at which uptake is half of the maximum [moles/L]. If the initial substrate level at t = 0 is S_0, this differential equation can be solved for

$$S = S_0 - v_m t + k_s \ln(S_0/S)$$

Develop an M-file to generate a plot of S versus t for the case where S_0 = 8 moles/L, v_m = 0.7 moles/L/d, and k_s = 2.5 moles/L.

5.12 A reversible chemical reaction

$$2A + B \leftrightarrows C$$

can be characterized by the equilibrium relationship

$$K = \frac{c_c}{c_a^2 c_b}$$

where the nomenclature c_i represents the concentration of constituent i. Suppose that we define a variable x as representing the number of moles of C that are produced. Conservation of mass can be used to reformulate the equilibrium relationship as

$$K = \frac{(c_{c,0} + x)}{(c_{a,0-2x})^2 (c_{b,0} - x)}$$

where the subscript 0 designates the initial concentration of each constituent. If $K = 0.016$, $c_{a,0} = 42$, $c_{b,0} = 28$, and $c_{c,0} = 4$, determine the value of x.
(a) Obtain the solution graphically.
(b) On the basis of **(a)**, solve for the root with initial guesses of $x_l = 0$ and $x_u = 20$ to $\varepsilon_s = 0.5\%$. Choose either bisection or false position to obtain your solution. Justify your choice.

5.13 Figure P5.13a shows a uniform beam subject to a linearly increasing distributed load. The equation for the resulting elastic curve is (see Fig. P5.13b)

$$y = \frac{w_0}{120EIL}(-x^5 + 2L^2x^3 - L^4x) \qquad \text{(P5.13)}$$

Use bisection to determine the point of maximum deflection (i.e., the value of x where $dy/dx = 0$). Then substitute this value into Eq. (P5.13) to determine the value of the maximum deflection. Use the following parameter values in your computation: $L = 600cm$, $E = 50{,}000kN/cm^2$, $I = 30{,}000cm^4$, and $w_0 = 2.5kN/cm$.

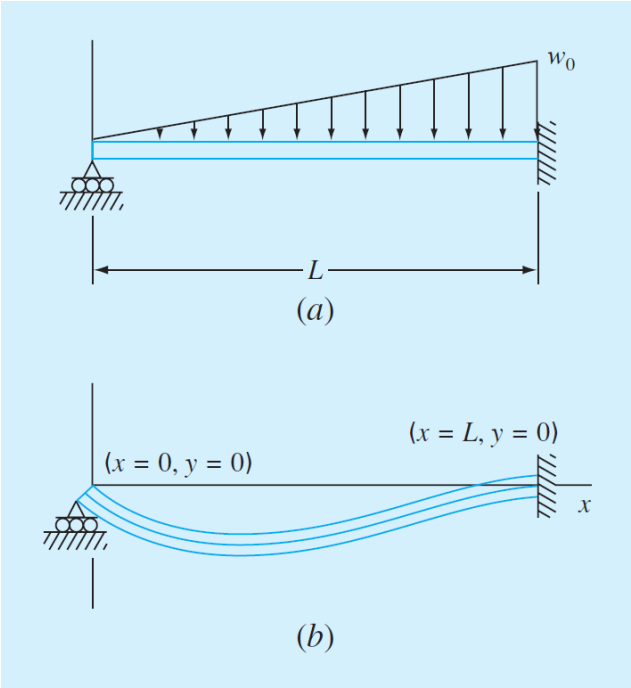


Figure P5.13

5.14 You buy a $35,000 vehicle for nothing down at $8,500 per year for 7 years. Use the `bisect` function from Fig. 5.7 to determine the interest rate that you are paying. Employ initial guesses for the interest rate of 0.01 and 0.3 and a stopping criterion of 0.00005. The formula relating present worth P, annual payments A, number of years n, and interest rate i is

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

5.15 Many fields of engineering require accurate population estimates. For example, transportation engineers might find it necessary to determine separately the population growth trends of a city and adjacent suburb. The population of the urban area is declining with time according to

$$P_u(t) = P_{u,max}e^{k_u t} + P_{u,min}$$

while the suburban population is growing, as in

$$P_s(t) = \frac{P_{s,max}}{1 + [P_{s,max}/P_0 - 1]e^{-k_s t}}$$

where $P_{u,max}$, k_u, $P_{s,max}$, P_0, and k_s = empirically derived parameters. Determine the time and corresponding values of $P_u(t)$ and $P_s(t)$ when the suburbs are 20% larger than the city. The parameter values are $O_{u,max} = 80,000$, $k_u = 0.05/yr$, $P_{u,min} = 110,000$ people, $P_{s,max} = 320,000$ people, $P_0 = 10,000$ people, and $k_s = 0.09/yr$. To obtain your solutions, use **(a)** graphical, and **(b)** false-position methods.

5.16 The resistivity ρ of doped silicon is based on the charge q on an electron, the electron density n, and the electron mobility μ. The electron density is given in terms of the doping density N and the intrinsic carrier density n_i. The electron mobility is described by the temperature T, the reference temperature T_0, and the reference mobility $_0$. The equations required to compute the resistivity are

$$\rho = \frac{1}{qn\mu}$$

where

$$n = \frac{1}{2}\left(N + \sqrt{N^2 + 4n_i^2}\right) \quad \text{and} \quad \mu = \mu_0\left(\frac{T}{T_0}\right)^{-2.42}$$

Determine N, given $T_0 = 300$ K, $T = 1000$ K, $\mu_0 = 1360 cm^2$ (V s)$^-1$, $q = 1.7 \times 10^{-19}$ C, $n_i = 6.21 \times 10^9 cm^{-3}$, and a desired $\rho = 6.5 \times 10^6$ V s cm/C. Employ initial guesses of $N = 0$ and 2.5×10^{10}. Use **(a)** bisection and **(b)** the false position method.

5.17 Atotal charge Q is uniformly distributed around a ring-shaped conductor with radius a. A charge q is located at a distance x from the center of the ring (Fig. P5.17). The force exerted on the charge by the ring is given by

$$F = \frac{1}{4\pi e_0}\frac{qQx}{(x^2 + a^2)^{3/2}}$$

where $e_0 = 8.9 \times 10^{-12} C^2/(\text{N m}^2)$. Find the distance x where the force is 1.25 N if q and Q are 2×10^{-5} C for a ring with a radius of 0.85m.

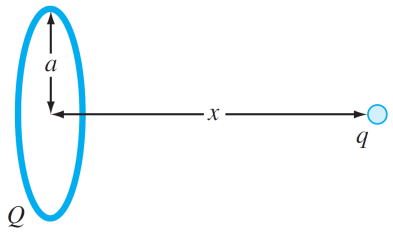


Figure P5.17

5.18 For fluid flow in pipes, friction is described by a dimensionless number, the *Fanning friction factor f*. The Fanning friction factor is dependent on a number of parameters related to the size of the pipe and the fluid, which can all be represented by another dimensionless quantity, the *Reynolds number* Re. A formula that predicts f given Re is the *von Karman equation*:

$$\frac{1}{\sqrt{f}} = 4log_{10}(Re\sqrt{f}) - 0.4$$

Typical values for the Reynolds number for turbulent flow are 10,000 to 500,000 and for the Fanning friction factor are 0.001 to 0.01. Develop a function that uses bisection to solve for f given a user-supplied value of Re between 2,500 and 1,000,000. Design the function so that it ensures that the absolute error in the result is $E_{a,d} < 0.000005$.

5.19 Mechanical engineers, as well as most other engineers, use thermodynamics extensively in their work. The following polynomial can be used to relate the zero-pressure specific heat of dry air c_p kJ/(kg K) to temperature (K):

$$c_p = 0.99403 + 1.671 \times 10^{-4}T + 9.7215 \times 10^{-8}T^2 - 9.5838 \times 10^{-11}T^3 + 1.9520 \times 10^{-14}T^4$$

Develop a plot of c_p versus a range of $T = 0$ to 1200 K, and then use bisection to determine the temperature that

corresponds to a specific heat of 1.1 kJ/(kg K).

5.20 The upward velocity of a rocket can be computed by the following formula:

$$v = u\ln\frac{m_0}{m_0 - qt} - gt$$

where v = upward velocity, u = the velocity at which fuel is expelled relative to the rocket, m_0 = the initial mass of the rocket at time $t = 0$, q=the fuel consumption rate, and g=the downward acceleration of gravity (assumed constant = $9.81 m/s^2$). If $u = 1800 m/s$, $m_0 = 160{,}000 kg$, and $q = 2600 kg/s$, compute the time at which $v = 750 m/s$. (Hint: t is somewhere between 10 and 50 s.) Determine your result so that it is within 1% of the true value. Check your answer.

5.21 Although we did not mention it in Sec. 5.6, Eq. (5.13) is an expression of *electroneutrality*—that is, that positive and negative charges must balance. This can be seen more clearly by expressing it as

$$[H^+] = [HCP_3^-] + 2[CO_3^{2-}] + [OH^-]$$

In other words, the positive charges must equal the negative charges. Thus, when you compute the pH of a natural water body such as a lake, you must also account for other ions that may be present. For the case where these ions originate from nonreactive salts, the net negative minus positive charges due to these ions are lumped together in a quantity called *alkalinity*, and the equation is reformulated as

$$Alk + [H^+] = [HCO_3^-] + 2[CO_3^{2-}] + [OH^-] \qquad \text{(P5.21)}$$

where Alk = alkalinity (eq/L). For example, the alkalinity of Lake Superior is approximately 0.4×10^{-3} eq/L. Perform the same calculations as in Sec. 5.6 to compute the pH of Lake Superior in 2008. Assume that just like the raindrops, the lake is in equilibrium with atmospheric CO_2 but account for the alkalinity as in Eq. (P5.21).

5.22 According to *Archimedes' principle*, the *buoyancy* force is equal to the weight of fluid displaced by the submerged portion of the object. For the sphere depicted in Fig. P5.22, use bisection to determine the height, h, of the portion that is above water. Employ the following values for your computation: $r = 1$ m, ρ_s = density of sphere = $200 kg/m^3$, and ρ_w= density of water = $1{,}000 kg/m^3$. Note that the volume of the above-water portion of the sphere can be computed with

$$V = \frac{\pi h^2}{3}(3r - h)$$

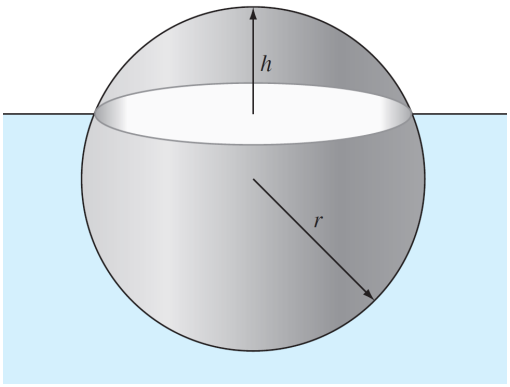


Figure P5.22

5.23 Perform the same computation as in Prob. 5.22, but for the frustrum of a cone as depicted in Fig. P5.23. Employ the following values for your computation: $r_1 = 0.5$ m, $r_2 = 1$ m, $h = 1$ m, ρ_f = frustrum density = $200 kg/m^3$, and ρ_w = water density = $1{,}000 kg/m^3$. Note that the volume of a frustrum is given by

$$V = \frac{\pi h}{3}(r_1^2 + r_2^2 + r_1 r_2)$$

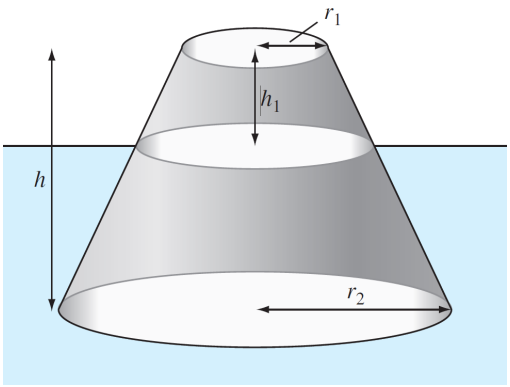


Figure P5.23

Chapter 4

Roots: 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.
- Understanding how Brent's method combines reliable bracketing methods with fast open methods to locate roots in a robust and efficient manner.
- 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.

4.1. 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:

$$x = g(x) \tag{6.1}$$

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_i , Eq. (6.1) can be used to compute a new estimate x_{i+1} as expressed by the iterative formula

$$x_{i+1} = g(x_i) \tag{6.2}$$

As with many other iterative formulas in this book, the approximate error for this equation can be determined using the error estimator:

$$\varepsilon_a = \left| \frac{x_{i+1} - x_i}{x_{i+1}} \right| 100\% \tag{6.3}$$

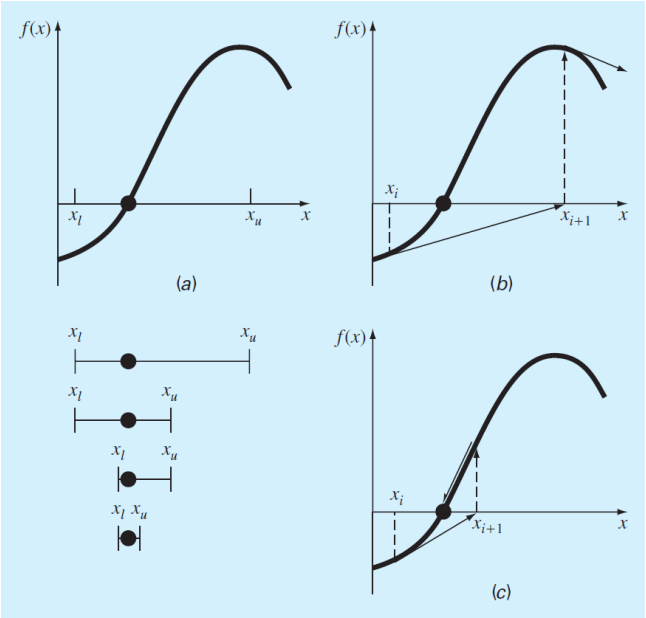


Figure 4.1: Graphical depiction of the fundamental difference between the (a) bracketing and (b) and (c) open methods for root location. In (a), which is bisection, the root is constrained within the interval prescribed by x_l and x_u . In contrast, for the open method depicted in (b) and (c), which is Newton-Raphson, a formula is used to project from x_i to $x_i + 1$ in an iterative fashion. Thus the method can either (b) diverge or (c) converge rapidly, depending on the shape of the function and the value of the initial guess.

Example 4.1. 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	$\lvert\varepsilon_a\rvert$, %	$\lvert\varepsilon_t\rvert$, %	$\lvert\varepsilon_t\rvert_i / \lvert\varepsilon_t\rvert_{i-1}$
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 (for this case, 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

$$y_1 = f_1(x) \tag{6.4}$$

and

$$y_2 = f_2(x) \tag{6.5}$$

can be plotted separately (Fig. 6.2b). The x values corresponding to the intersections of these functions represent the roots of $f(x) = 0$.

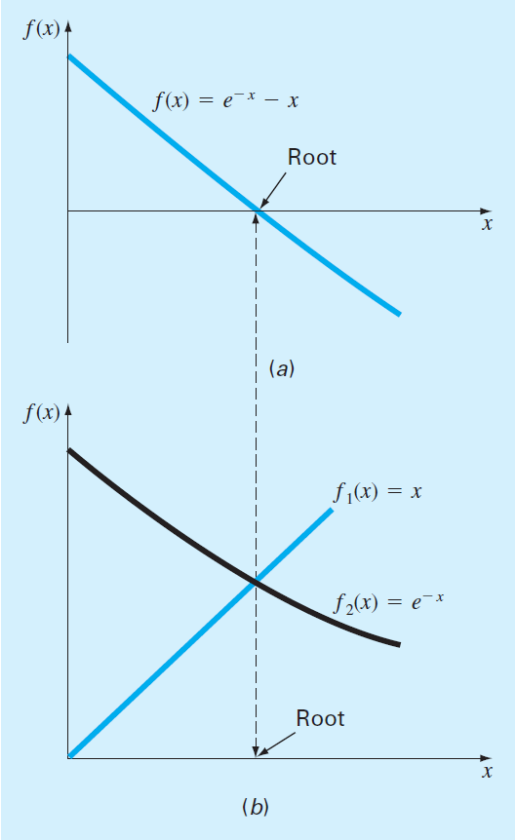


Figure 4.2: Two alternative graphical methods for determining the root of $f(x) = e^{-x} - x$. (a) Root at the point where it crosses the x axis; (b) root at the intersection of the component functions.

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 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 (2010), 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:

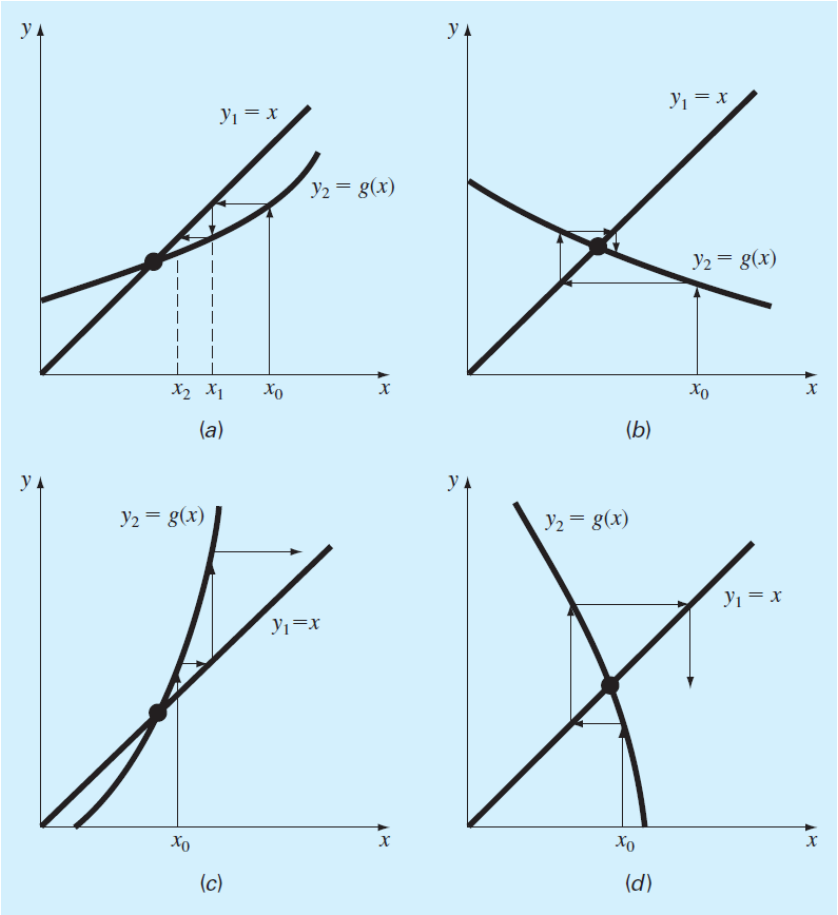


Figure 4.3: Graphical depiction of (a) and (b) convergence and (c) and (d) divergence of simple fixed-point iteration. Graphs (a) and (c) are called monotone patterns whereas (b) and (c) are called oscillating or spiral patterns. Note that convergence occurs when $|g'(x)| < 1$

$$E_{i+1} = g'(\xi)E_i$$

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).

4.2. 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

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{6.6}$$

which is called the *Newton-Raphson formula.*

Example 4.2. 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

i	x_i	$\lvert\varepsilon_t\rvert$, %
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). ■

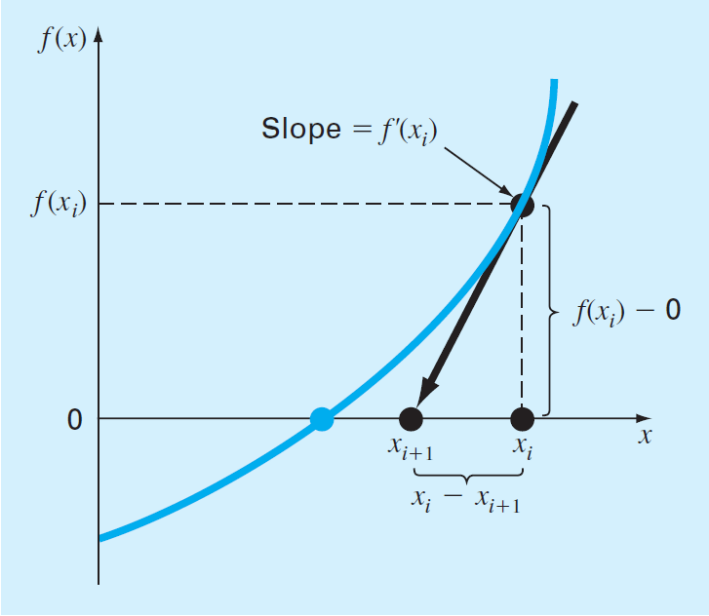


Figure 4.4: Graphical depiction of the Newton-Raphson method. A tangent to the function of x_i [that is,$f'(x)$] is extrapolated down to the x axis to provide an estimate of the root at x_{i+1}.

As with other root-location methods, Eq. (6.3) can be used as a termination criterion. In addition, a theoretical analysis (Chapra and Canale, 2010) provides insight regarding the rate of convergence as expressed by

$$E_{t,i+1} = \frac{-f''(x_r)}{2f'(x_r)} E_{t,i}^2 \tag{6.7}$$

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, 2010). However, even when dealing with simple roots, difficulties can also arise, as in the following example.

Example 4.3. 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

i	x_i	$\lvert\varepsilon_a\rvert$, %
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.

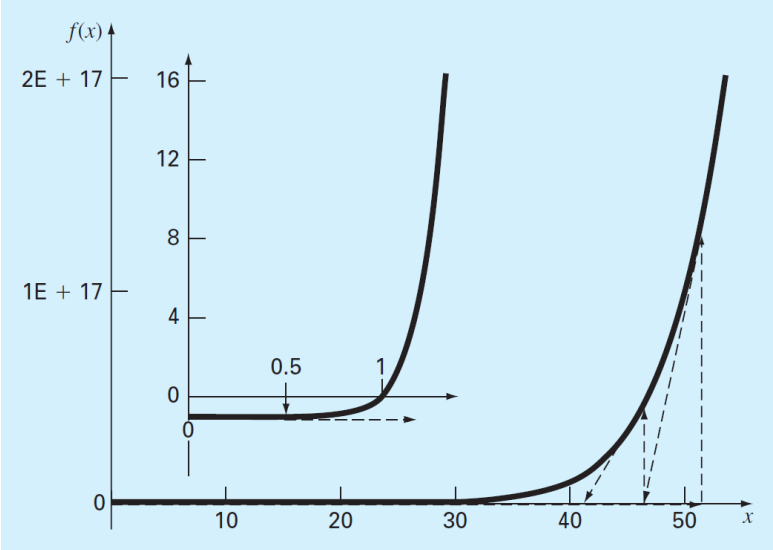


Figure 4.5: Graphical depiction of the Newton-Raphson method for a case with slow convergence. The inset shows how a near-zero slope initially shoots the solution far from the root. Thereafter, the solution very slowly converges on the root.

■

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 point (i.e., $f'(x) = 0$) occurs in the vicinity of a root. Notice that iterations beginning at x0 progressively diverge from the root. Fig. 6.6b illustrates the tendency of the Newton-Raphson technique to oscillate around a local maximum or minimum. Such oscillations may persist, or, as in Fig. 6.6b, a near-zero slope is reached whereupon the solution is sent far from the area of interest. Figure 6.6c 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. 6.6d, 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.

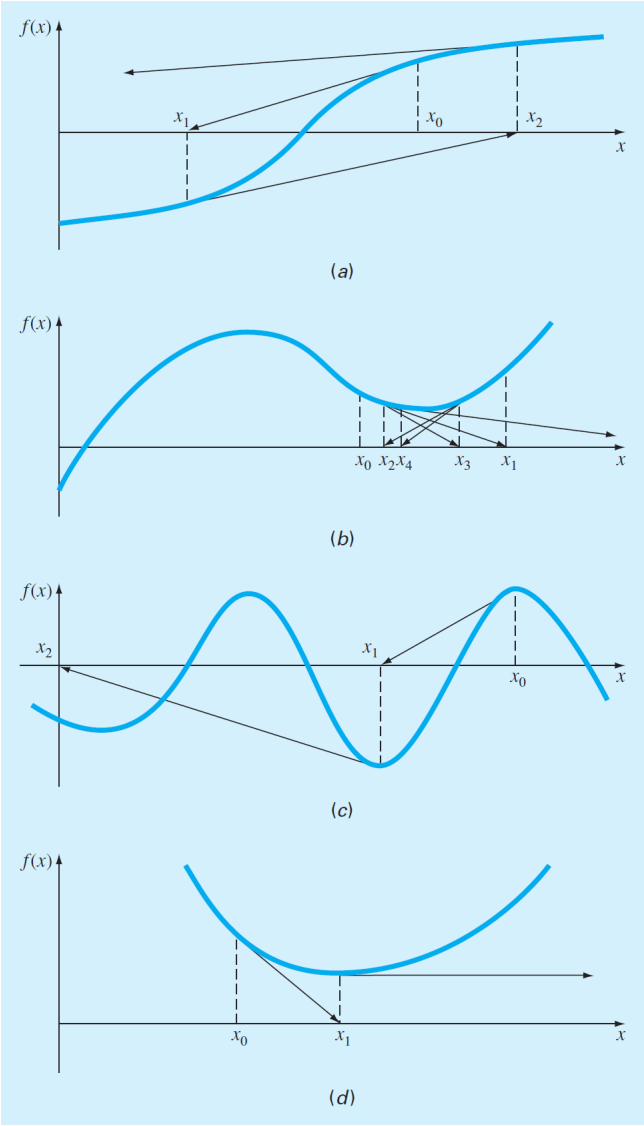


Figure 4.6: Four cases where the Newton-Raphson method exhibits poor convergence.

4.2.1. MATLAB M-file: newtraph

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(@(x) x^2-9,@(x) 2*x,5)
ans =
      3
```

Example 4.4. 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 $9.81 m/s^2$.

Solution. The function to be evaluated is

$$f(m) = \sqrt{\frac{gm}{c_d}} tanh\left(\sqrt{\frac{gc_d}{m}} t\right) - v(t) \tag{E6.4.1}$$

To apply the Newton-Raphson method, the derivative of this function must be evaluated with respect to the unknown, m:

$$\frac{df(m)}{dm} = \frac{1}{2}\sqrt{\frac{g}{mc_d}}tanh\left(\sqrt{\frac{gc_d}{m}}t\right) - \frac{g}{2m}t\ \text{sech}^2\left(\sqrt{\frac{gc_d}{m}}t\right) \tag{E6.4.2}$$

```
function [root,ea,iter]=newtraph(func,dfunc,xr,es,maxit,varargin)
% newtraph: Newton-Raphson root location zeroes
%    [root,ea,iter]=newtraph(func,dfunc,xr,es,maxit,p1,p2,...):
%          uses Newton-Raphson method to find the root of func
% input:
%    func = name of function
%    dfunc = name of derivative of function
%    xr = initial guess
%    es = desired relative error (default = 0.0001%)
%    maxit = maximum allowable iterations (default = 50)
%    p1,p2,... = additional parameters used by function
% output:
%    root = real root
%    ea = approximate relative error (%)
%    iter = number of iterations

if nargin<3,error('at least 3 input arguments required'),end
if nargin<4|isempty(es),es=0.0001;end
if nargin<5|isempty(maxit),maxit=50;end
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 4.7: An M-file to implement the Newton-Raphson method.

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.

The two formulas can now be used in conjunction with the function `newtraph` to evaluate the root:

```
» y = @m sqrt(9.81*m/0.25)*tanh(sqrt(9.81*0.25/m)*4)-36;
» dy = @m 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;
» newtraph(y,dy,140,0.00001)
ans =
      142.7376
```

■

4.3. 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:

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)} \tag{6.8}$$

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:

$$x_{i+1} = x_i - \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i)} \tag{6.9}$$

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 4.5. 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 $9.81 m/s^2$. 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:

$$x_0 = 50 \qquad f(x_0) = -4.57938708$$

$$x_0 + \delta x_0 = 50.00005 \qquad f(x_0 + \delta x_0) = -4.579381118$$

$$x_1 = 50 - \frac{10^{-6}(50)(-4.57938708))}{-4.579381118 - (-4.57938708)} = 88.39931 (|\varepsilon_t| = 38.1\%; |\varepsilon_a| = 43.4\%)$$

Second iteration:

$$x_1 = 88.39931 \qquad f(x_1) = -1.69220771$$

$$x_1 + \delta x_1 = 88.39940 \qquad f(x_1 + \delta x_1) = -1.692203516$$

$$x_2 = 88.39931 - \frac{10^{-6}(88.39931)(-1.69220771)}{-1.692203516 - (-1.69220771)} = 124.08970 (|\varepsilon_t| = 13.1\%; |\varepsilon_a| = 28.76\%)$$

The calculation can be continued to yield

i	x_i	$\lvert\varepsilon_t\rvert$, %	$\lvert\varepsilon_a\rvert$, %
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.

■

4.4. BRENT'S METHOD

Wouldn't it be nice to have a hybrid approach that combined the reliability of bracketing with the speed of the open methods? *Brent's root-location method* is a clever algorithm that does just that by applying a speedy open method wherever possible, but reverting to a reliable bracketing method if necessary. The approach was developed by Richard Brent (1973) based on an earlier algorithm of Theodorus Dekker (1969).

The bracketing technique is the trusty bisection method (Sec. 5.4), whereas two different open methods are employed. The first is the secant method described in Sec. 6.3. As explained next, the second is inverse quadratic interpolation.

4.4.1. 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. For this reason, it is sometimes referred to as a *linear interpolation method.*

Now suppose that we had three points. In that case, we could determine a quadratic function of x that goes through the three points (Fig. 6.8b). Just as with the linear secant method, the intersection of this parabola with the x axis would represent the new root estimate. And as illustrated in Fig. 6.8b, using a curve rather than a straight line often yields a better estimate.

Although this would seem to represent a great improvement, the approach has a fundamental flaw: it is possible that the parabola might not intersect the x axis! Such would be the case when the resulting parabola had complex roots. This is illustrated by the parabola, $y = f(x)$, in Fig. 6.9.

The difficulty can be rectified by employing inverse quadratic interpolation. That is, rather than using a parabola in x, we can fit the points with a parabola in y. This amounts to reversing the axes and creating a "sideways" parabola [the curve, $x = f(y)$, in Fig. 6.9].

If the three points are designated as (x_{i-2}, y_{i-2}), (x_{i-1}, y_{i-1}), and (x_i, y_i), a quadratic function of y that passes through the points can be generated as

$$g(y) = \frac{(y-y_{i-1})(y-y_i)}{(y_{i-2}-y_{i-1})(y_{i-2}-y_i)}x_{i-2} + \frac{(y-y_{i-2})(y-y_i)}{(y_{i-1}-y_{i-2})(y_{i-1}-y_i)}x_{i-1} + \frac{(y-y_{i-2})(y-y_{i-1})}{(y_i-y_{i-2})(y_i-y_{i-1})}x_i \tag{6.10}$$

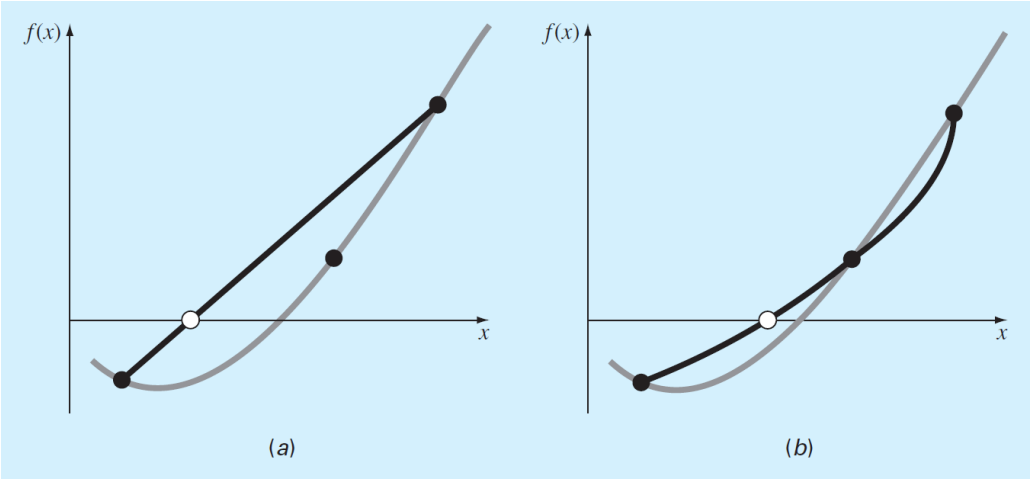


Figure 4.8: Comparison of (a) the secant method and (b) inverse quadratic interpolation. Note that the approach in (b) is called "inverse" because the quadratic function is written in y rather than in x.

As we will learn in Sec. 18.2, this form is called a *Lagrange polynomial*. The root, x_{i+1}, corresponds to $y = 0$, which when substituted into Eq. (6.10) yields

$$x_{i+1} = \frac{y_{i-1}y_i}{(y_{i-2}-y_{i-1})(y_{i-2}-y_i)}x_{i-2} + \frac{y_{i-2}y_i}{(y_{i-1}-y_{i-2})(y_{i-1}-y_i)}x_{i-1} + \frac{y_{i-2}y_{i-1}}{(y_i-y_{i-2})(y_i-y_{i-1})}x_i \tag{6.11}$$

As shown in Fig. 6.9, such a "sideways" parabola always intersects the x axis.

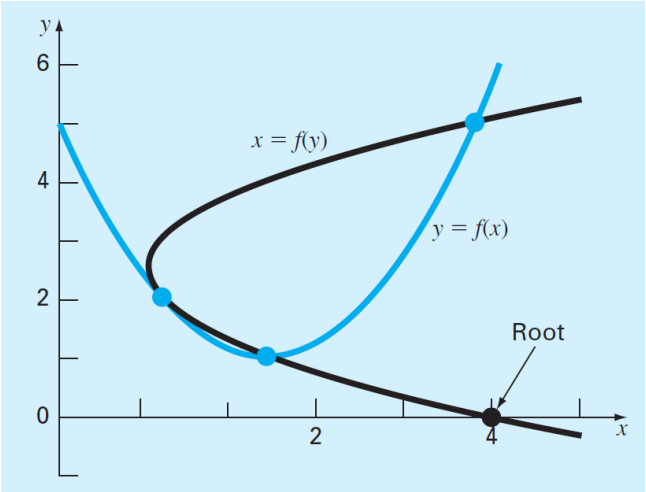


Figure 4.9: Two parabolas fit to three points. The parabola written as a function of $x, y = f(x)$, has complex roots and hence does not intersect the x axis. In contrast, if the variables are reversed, and the parabola developed as $x = f(y)$, the function does intersect the x axis.

Example 4.6. Inverse Quadratic Interpolation

Problem Statement. Develop quadratic equations in both x and y for the data points depicted in Fig. 6.9: $(1,2)$, $(2,1)$, and $(4,5)$. For the first, $y = f(x)$, employ the quadratic formula to illustrate that the roots are complex. For the latter, $x = g(y)$, use inverse quadratic interpolation (Eq. 6.11) to determine the root estimate.

Solution. By reversing the x's and y's, Eq. (6.10) can be used to generate a quadratic in x as

$$f(x) = \frac{(x-2)(x-4)}{(1-2)(1-4)}2 + \frac{(x-1)(x-4)}{(2-1)(2-4)}1 + \frac{(x-1)(x-2)}{(4-1)(4-2)}5$$

or collecting terms

$$f(x) = x^2 - 4x + 5$$

This equation was used to generate the parabola, $y = f(x)$, in Fig. 6.9. The quadratic formula can be used to determine that the roots for this case are complex,

$$x = \frac{4 \pm \sqrt{(-4)^2 - 4(1)(5)}}{2} = 2 \pm i$$

Equation (6.10) can be used to generate the quadratic in y as

$$g(y) = \frac{(y-1)(y-5)}{(2-1)(2-5)}1 + \frac{(y-2)(y-5)}{(1-2)(1-5)}2 + \frac{(y-2)(y-1)}{(5-2)(5-1)}4$$

or collecting terms:

$$g(y) = 0.5x^2 - 2.5x + 4$$

Finally, Eq. (6.11) can be used to determine the root as

$$x_{i+1} = \frac{-1(-5)}{(2-1)(2-5)}1 + \frac{-2(-5)}{(1-2)(1-5)}2 + \frac{-2(-1)}{(5-2)(5-1)}4 = 4$$

■

Before proceeding to Brent's algorithm, we need to mention one more case where inverse quadratic interpolation does not work. If the three y values are not distinct (i.e., $y_{i-2} = y_{i-1}$ or $y_{i-1} = y_i$), an inverse quadratic function does not exist. So this is where the secant method comes into play. If we arrive at a situation where the y values are not distinct, we can always revert to the less efficient secant method to generate a root using two of the points. If $y_{i-2} = y_{i-1}$, we use the secant method with x_{i-1} and x_i. If $y_{i-1} = y_i$, we use x_{i-2} and x_{i-1}.

4.4.2. Brent's Method Algorithm

The general idea behind the *Brent's root-finding method* is whenever possible to use one of the quick open methods. In the event that these generate an unacceptable result (i.e., a root estimate that falls outside the bracket), the algorithm reverts to the more conservative bisection method. Although bisection may be slower, it generates an estimate guaranteed to fall within the bracket. This process is then repeated until the root is located to within an acceptable tolerance. As might be expected, bisection typically dominates at first but as the root is approached, the technique shifts to the faster open methods.

Figure 6.10 presents a function based on a MATLAB M-file developed by Cleve Moler (2004). It represents a stripped down version of the `fzero` function which is the professional root-location function employed in MATLAB. For that reason, we call the simplified version: `fzerosimp`. Note that it requires another function `f` that holds the equation for which the root is being evaluated.

The `fzerosimp` function is passed two initial guesses that must bracket the root. Then, the three variables defining the search interval `(a,b,c)` are initialized, and `f` is evaluated at the endpoints.

```
function b = fzerosimp(xl,xu)
a = xl; b = xu; fa = f(a); fb = f(b);
c = a; fc = fa; d = b - c; e = d;
while (1)
   if fb == 0, break, end
   if sign(fa) == sign(fb) %If needed, rearrange points
      a = c; fa = fc; d = b - c; e = d;
   end
   if abs(fa) < abs(fb)
      c = b; b = a; a = c;
      fc = fb; fb = fa; fa = fc;
   end
   m = 0.5*(a - b); %Termination test and possible exit
   tol = 2 * eps * max(abs(b), 1);
   if abs(m) <= tol | fb == 0.
      break
   end
   %Choose open methods or bisection
   if abs(e) >= tol & abs(fc) > abs(fb)
      s = fb/fc;
      if a == c %Secant method
         p = 2*m*s;
         q = 1 - s;
      else %Inverse quadratic interpolation
         q = fc/fa; r = fb/fa;
         p = s * (2*m*q * (q - r) - (b - c)*(r - 1));
         q = (q - 1)*(r - 1)*(s - 1);
      end
      if p > 0, q = -q; else p = -p; end;
      if 2*p < 3*m*q - abs(tol*q) & p < abs(0.5*e*q)
         e = d; d = p/q;
      else
         d = m; e = m;
      end
   else %Bisection
      d = m; e = m;
   end
   c = b; fc = fb;
   if abs(d) > tol, b=b+d; else b=b-sign(b-a)*tol; end
   fb = f(b);
end
```

Figure 4.10: Function for Brent's root-finding algorithm based on a MATLAB M-file developed by Cleve Moler (2005).

A main loop is then implemented. If necessary, the three points are rearranged to satisfy the conditions required for the algorithm to work effectively. At this point, if the stopping criteria are met, the loop is terminated. Otherwise, a decision structure chooses among the three methods and checks whether the outcome is acceptable. Afinal section then evaluates `f` at the new point and the loop is repeated. Once the stopping criteria are met, the loop terminates and the final root estimate is returned.

4.5. MATLAB FUNCTION: FZERO

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(@(x) x^2-9,-4)
x =
      3
```

If we want to find the positive root, use a guess that is near it:

```
» x = fzero(@(x) x^2-9,4
)   x =
      3
```

If we put in an initial guess of zero, it finds the negative root:

```
» x = fzero(@(x) 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(@(x) 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(@(x) x^2-9,[-4 4])
???  Error using ==> fzero
The function values at the interval endpoints must differ in sign.
```

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 an unacceptable 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.

A more complete representation of the fzero syntax can be written as

```
[x,fx] = fzero(function,x0,options,p1,p2,...)
```

where `[x,fx]` = 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...` 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_i has the value val_i. 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 4.7. 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(@(x) 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(@(x) x^10-1,0.5,options)
x =
      1.0009
fx =
      0.0090
```

■

4.6. POLYNOMIALS

Polynomials are a special type of nonlinear algebraic equation of the general form

$$f_n(x) = a_1x^n + a_2x^{n-1} + \cdots + a_{n-1}x^2 + a_nx + a_{n+1} \tag{6.12}$$

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 nth 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.

4.6.1. MATLAB Function: `roots`

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 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

$$a_1x^5 + a_2x^4 + a_3x^3 + a_4x^2 + a_5x + a_6 = 0 \tag{6.13}$$

Dividing by a_1 and rearranging yields

$$x^5 = -\frac{a_2}{a_1}x^4 - \frac{a_3}{a_1}x^3 - \frac{a_4}{a_1}x^2 - \frac{a_5}{a_1}x - \frac{a_6}{a_1}$$

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:

$$\begin{bmatrix} -a_2/a_1 & -a_3/a_1 & -a_4/a_1 & -a_5/a_1 & -a_6/a_1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{hfill (6.14)}$$

Equation (6.14) 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 4.8. 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:

$$f_5(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25 \tag{E6.8.1}$$

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. (E6.8.1): 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)
```

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. (E6.8.1) 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
```

■

4.7. CASE STUDY - PIPE FRICTION

Background. Determining fluid flow through pipes and tubes has great relevance in many areas of engineering and science. In engineering, typical applications include the flow of liquids and gases through pipelines and cooling systems. Scientists are interested in topics ranging from flow in blood vessels to nutrient transmission through a plant's vascular system.

The resistance to flow in such conduits is parameterized by a dimensionless number called the *friction factor*. For turbulent flow, the *Colebrook equation* provides a means to calculate the friction factor:

$$0 = \frac{1}{\sqrt{f}} + 2.0\log\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{\mathrm{Re}\sqrt{f}}\right) \tag{6.15}$$

where ε = the roughness (m), D = diameter (m), and Re = the Reynolds number:

$$\mathrm{Re} = \frac{\rho V D}{\mu}$$

where ρ = the fluid's density $\left(\mathrm{kg/m^3}\right)$, V = its velocity (m/s), and μ = dynamic viscosity $\left(\mathrm{N\cdot s/m^2}\right)$. In addition to appearing in Eq. (6.15), the Reynolds number also serves as the criterion for whether flow is turbulent ($\mathrm{Re} > 4000$).

In this case study, we will illustrate how the numerical methods covered in this part of the book can be employed to determine f for air flow through a smooth, thin tube. For this case, the parameters are $\rho = 1.23\ \mathrm{kg/m^3}$, $\mu = 1.79 \times 10^{-5}\ \mathrm{N\cdot s/m^2}$, $D = 0.005$ m, $V = 40$ m/s and $\varepsilon = 0.0015$ mm. Note that friction factors range from about 0.008 to 0.08. In addition, an explicit formulation called the Swamee-Jain equation provides an approximate estimate:

$$f = \frac{1.325}{\left[\ln\left(\frac{\varepsilon}{3.7D} + \frac{5.74}{\mathrm{Re}^{0.9}}\right)\right]^2} \tag{6.16}$$

Solution. The Reynolds number can be computed as

$$\mathrm{Re} = \frac{\rho V D}{\mu} = \frac{1.23(40)0.005}{1.79 \times 10^{-5}} = 13,743$$

This value along with the other parameters can be substituted into Eq. (6.15) to give

$$g(f) = \frac{1}{\sqrt{f}} + 2.0\log\left(\frac{0.0000015}{3.7(0.005)} + \frac{2.51}{13,743\sqrt{f}}\right)$$

Before determining the root, it is advisable to plot the function to estimate initial guesses and to anticipate possible difficulties. This can be done easily with MATLAB:

```
>> rho=1.23;mu=1.79e-5;D=0.005;V=40;e=0.0015/1000;
>> Re=rho*V*D/mu;
>> g=@(f) 1/sqrt(f)+2*log10(e/(3.7*D)+2.51/(Re*sqrt(f)));
>> fplot(g,[0.008 0.08]),grid,xlabel('f'),ylabel('g(f)')
```

As in Fig. 6.11, the root is located at about 0.03.

Because we are supplied initial guesses ($x_l = 0.008$ and $x_u = 0.08$), either of the bracketing methods from Chap. 5 could be used. For example, the bisect function developed in Fig. 5.7 gives a value of $f = 0.0289678$ with a percent relative error of error of 5.926×10^{-5} in 22 iterations. False position yields a result of similar precision in 26 iterations. Thus, although they produce the correct result, they are somewhat inefficient. This would not be important for a single application, but could become prohibitive if many evaluations were made.

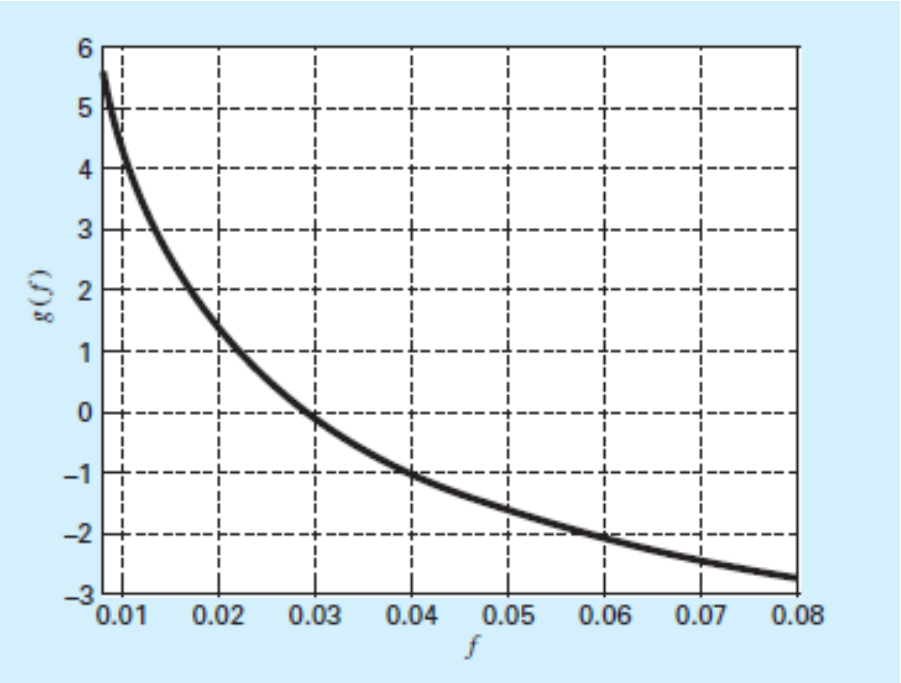


Figure 6.11

We could try to attain improved performance by turning to an open method. Because Eq. (6.15) is relatively straightforward to differentiate, the Newton-Raphson method is a good candidate. For example, using an initial guess at the lower end of the range ($x_0 = 0.008$), the newt raph function developed in Fig. 6.7 converges quickly:

```
>> dg=@(f) -2/log(10)*1.255/Re*f^(-3/2)/(e/D/3.7 ...
                          +2.51/Re/sqrt(f))-0.5/f^(3/2);
>> [f ea iter]=newtraph(g,dg,0.008)
f =
    0.02896781017144
ea =
    6.870124190058040e-006
iter =
    6
```

However, when the initial guess is set at the upper end of the range ($x_0 = 0.08$), the routine diverges,

```
>> [f ea iter]=newtraph(g,dg,0.08)
f =
    NaN + NaNi
```

As can be seen by inspecting Fig. 6.11, this occurs because the function's slope at the initial guess causes the first iteration to jump to a negative value. Further runs demonstrate that for this case, convergence only occurs when the initial guess is below about 0.066.

So we can see that although the Newton-Raphson is very efficient, it requires good initial guesses. For the Colebrook equation, a good strategy might be to employ the Swamee-Jain equation (Eq. 6.16) to provide the initial guess as in

```
>> fSJ=1.325/log(e/(3.7*D)+5.74/Re^0.9)^2
fSJ =
    0.02903099711265
>> [f ea iter]=newtraph(g,dg,fSJ)
f =
    0.02896781017144
ea =
    8.510189472800060e-010
iter =
    3
```

Aside from our homemade functions, we can also use MATLAB's built-in `fzero` function. However, just as with the Newton-Raphson method, divergence also occurs when `fzero` function is used with a single guess. However, in this case, guesses at the lower end of the range cause problems. For example,

```
>> fzero(g,0.008)

Exiting fzero: aborting search for an interval containing a sign
change because complex function value encountered ...
                                  during search.
(Function value at -0.0028 is -4.92028-20.2423i.)
Check function or try again with a different starting value.
ans =
    NaN
```

If the iterations are displayed using optimset (recall Example 6.7), it is revealed that a negative value occurs during the search phase before a sign change is detected and the routine aborts. However, for single initial guesses above about 0.016, the routine works nicely. For example, for the guess of 0.08 that caused problems for Newton-Raphson, f zero does just fine:

```
>> fzero(q,0.08)
ans =
    0.02896781017144
```

As a final note, let's see whether convergence is possible for simple fixed-point iteration. The easiest and most straightforward version involves solving for the first f in Eq. (6.15):

$$f_{i+1} = \frac{0.25}{\left(\log\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{\mathrm{Re}\sqrt{f_i}}\right)\right)^2} \tag{6.17}$$

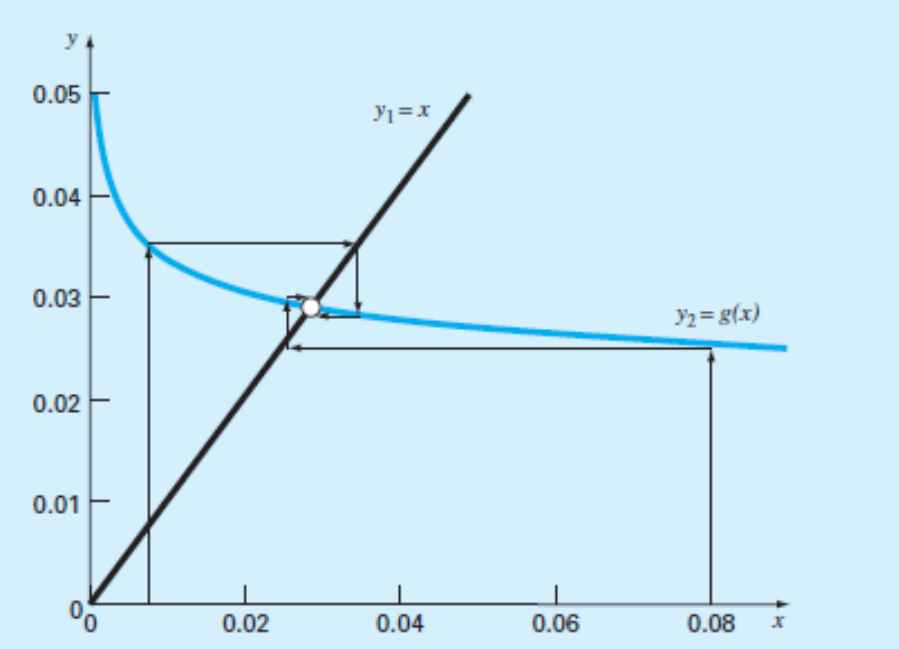


Figure 6.12

The two-curve display of this function depicted indicates a surprising result (Fig. 6.12). Recall that fixed-point iteration converges when the y_2 curve has a relatively flat slope (i.e., $|g'(\xi)| < 1$). As indicated by Fig. 6.12, the fact that the y_2 curve is quite flat in the range from $f = 0.008$ to 0.08 means that not only does fixed-point iteration converge, but it converges fairly rapidly! In fact, for initial guesses anywhere between 0.008 and 0.08, fixedpoint iteration yields predictions with percent relative errors less than 0.008% in six or fewer iterations! Thus, this simple approach that requires only one guess and no derivative estimates performs really well for this particular case.

The take-home message from this case study is that even great, professionally developed software like MATLAB is not always foolproof. Further, there is usually no single method that works best for all problems. Sophisticated users understand the strengths and weaknesses of the available numerical techniques. In addition, they understand enough of the underlying theory so that they can effectively deal with situations where a method breaks down.

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 $\varepsilon_a \leq 0.01\%$. Verify that the process is linearly convergent as described at the end of Sec. 6.1.

6.2 Use (a) fixed-point iteration and (b) the NewtonRaphson 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, $x_i = 3.5, \delta = 0.01$).

(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, $x_{i-1} = 0.5$ and $x_i = 0.4$.

(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 $f(x) = \tanh\left(x^2 - 9\right)$ 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 Mechanical engineers, as well as most other engineers, use thermodynamics extensively in their work. The following polynomial can be used to relate the zero-pressure specific heat of dry air c_p in kJ/(kgK) to temperature in K :

$$c_p = 0.99403 + 1.671 \times 10^{-4} T + 9.7215 \times 10^{-8} T^2 - 9.5838 \times 10^{-11} T^3 + 1.9520 \times 10^{-14} T^4$$

Write a MATLAB script (a) to plot c_p versus a range of $T = 0$ to 1200 K, and (b) to determine the temperature that corresponds to a specific heat of 1.1 kJ/(kgK) with MATLAB polynomial functions.

6.14 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) :

$$H_2O \rightleftarrows H_2 + \frac{1}{2}O_2$$

If it is assumed that this is the only reaction involved, the mole fraction x of H_2O that dissociates can be represented by

$$K = \frac{x}{1-x}\sqrt{\frac{2p_t}{2+x}} \qquad \text{(P6.14.1)}$$

where K is the reaction's equilibrium constant and p_t is the total pressure of the mixture. If $p_t = 3$ atm and $K = 0.05$, determine the value of x that satisfies Eq. (P6.14.1).

6.15 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/(kgK)], T = absolute temperature (K), p = absolute pressure (kPa), and v = the volume of a kg of gas $\left(\mathrm{m}^3/\mathrm{kg}\right)$. The parameters a and b are calculated by

$$a = 0.427\frac{R^2 T_c^{2.5}}{p_c} \qquad b = 0.0866R\frac{T_c}{p_c}$$

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 65,000kPa. Use a root-locating method of your choice to calculate v and then determine the mass of methane contained in the tank.

6.16 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, $L = 5$ m^3, and $V = 8$ m^3.

6.17 A catenary cable is one which is hung between two points not in the same vertical line. As depicted in Fig. P6.17a, 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. P6.17b, where T_A and T_B are 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 .

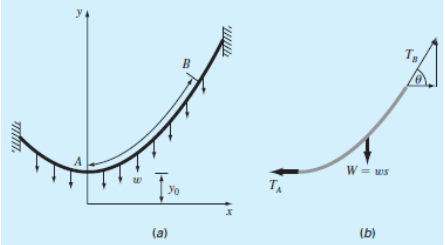


Figure P6.17

6.18 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 $I = 3.5$

6.19 Figure P6.19 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: $R = 225\Omega, C = 0.6 \times 10^{-6}$ F, and $L = 0.5$H.

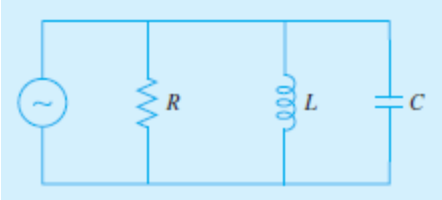


Figure P6.19

6.20 Real mechanical systems may involve the deflection of nonlinear springs. In Fig. P6.20, a block of mass m is released a distance h above a nonlinear spring. The resistance force F of the spring is given by

$$F = -\left(k_1 d + k_2 d^{3/2}\right)$$

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: $k_1 = 40{,}000$ g/s^2, $k_2 = 40$ g/ (s^2 m^5), $m = 95$ g, g = 9.81 m/s^2 and $h = 0.43$ m.

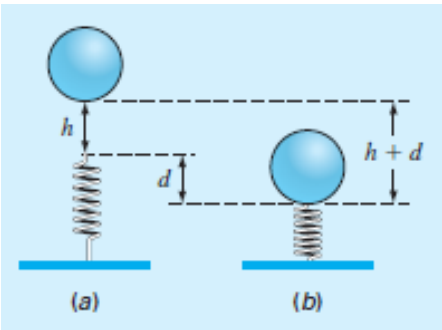


Figure P6.20

6.21 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. P6.21. The trajectory can be modeled as

$$y = (\tan\theta_0)x - \frac{g}{2v_0^2\cos^2\theta_0}x^2 + y_0$$

Find the appropriate initial angle θ_0, if $v_0 = 30$ m/s, and the distance to the catcher 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.

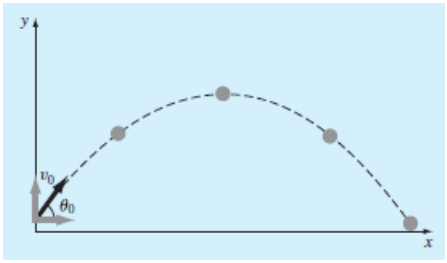


Figure P6.21

6.22 You are designing a spherical tank (Fig. P6.22) 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 $\left[\text{m}^3\right]$, h = depth of water in tank [m], and R = the tank radius [m].

If $R = 3$ m, what depth must the tank be filled to so that it holds 30 m^3 ? Use three iterations of the most efficient numerical method possible to determine your answer. Determine 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.

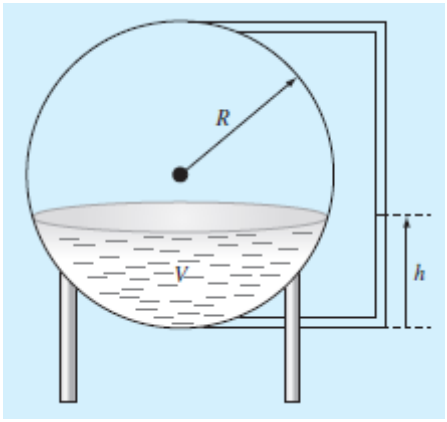


Figure P6.22

6.23 Perform the identical MATLAB operations as those in Example 6.8 to manipulate and find all the roots of the polynomial

$$f_5(x) = (x+2)(x+5)(x-6)(x-4)(x-8)$$

6.24 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.

6.25 The Manning equation can be written for a rectangular open channel as

$$Q = \frac{\sqrt{S}(BH)^{5/3}}{n(B+2H)^{2/3}}$$

where Q = flow $\left(\mathrm{m}^3/\mathrm{s}\right), S$ = slope $(\mathrm{m}/\mathrm{m}), H$ = depth(m), and n = the Manning roughness coefficient. Develop a fixed-point iteration scheme to solve this equation for H given $Q = 5, S = 0.0002, B = 20$, and $n = 0.03$. Perform the computation until ε_α is less than $\varepsilon_s = 0.05\%$. Prove that your scheme converges for all initial guesses greater than or equal to zero.

6.26 See if you can develop a foolproof function to c pute the friction factor based on the Colebrook equatio described in Sec. 6.7. Your function should retum a pre result for Reynolds number ranging from 4000 to 10^7 for ε/D ranging from 0.00001 to 0.05.

6.27 Use the Newton-Raphson method to find the roo $f(x) = e^{-0.5x}(4-x) - 2$

Employ initial guesses of (a) 2 , (b) 6 , and (c) Explain your results.

6.28 Given

$$f(x) = -2x^6 - 1.5x^4 + 10x + 2$$

Use a root-location technique to determine the maximur this function. Perform iterations until the approximate relative error falls below 5%. If you use a bracketing method, use initial guesses of $x_l = 0$ and $x_u = 1$. If you use the Newton-Raphson or the modified secant method, use an initial guess of $x_i = 1$. If you use the secant method, use initial guesses of $x_{i-1} = 0$ and $x_i = 1$. Assuming that convergence is not an issue, choose the technique that is best suited to this problem. Justify your choice.

6.29 You must determine the root of the following easily differentiable function:

$$e^{0.5x} = 5 - 5x$$

Pick the best numerical technique, justify your choice, and then use that technique to determine the root. Note that it is known that for positive initial guesses, all techniques except fixed-point iteration will eventually converge. Perform iterations until the approximate relative error falls below 2%. If you use a bracketing method, use initial guesses of $x_I = 0$ and $x_u = 2$. If you use the Newton-Raphson or the modified secant method, use an initial guess of $x_i = 0.7$. If you use the secant method, use initial guesses of $x_{i-1} = 0$ and $x_i = 2$.

6.30 (a) Develop an M-file function to implement Brent's root-location method. Base your function on Fig. 6.10, but with the beginning of the function changed to

```
 function [b,fb] = fzeronew(f,xl,xu,
varargin)
 % fzeronew: Brent root location
zeroes
 % [b,fb] = fzeronew(f,xl,xu,p1,p2
,...):
 % uses Brent's method to find the
root of f
 % input:
 % f = name of function
 % xl, xu = lower and upper guesses
 % p1,p2,... = additional parameters
 used by f
 % output:
 % b = real root
 % fb = function value at root
```

Make the appropriate modifications so that the function performs as outlined in the documentation statements. In addition, include error traps to ensure that the function's three required arguments `(f, x 1, x u)` are prescribed, and that the initial guesses bracket a root. (b) Test your function by using it to solve for the root of the function from Example 5.6 using

```
 >> [x,fx] = fzeronew(@(x,n) x^n
-1,0,1.3,10)
```

Chapter 5

Roots: 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.
- Understanding how Brent's method combines reliable bracketing methods with fast open methods to locate roots in a robust and efficient manner.
- 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.

5.1. 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:

$$x = g(x) \tag{6.1}$$

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_i , Eq. (6.1) can be used to compute a new estimate x_{i+1} as expressed by the iterative formula

$$x_{i+1} = g(x_i) \tag{6.2}$$

As with many other iterative formulas in this book, the approximate error for this equation can be determined using the error estimator:

$$\varepsilon_a = \left| \frac{x_{i+1} - x_i}{x_{i+1}} \right| 100\% \tag{6.3}$$

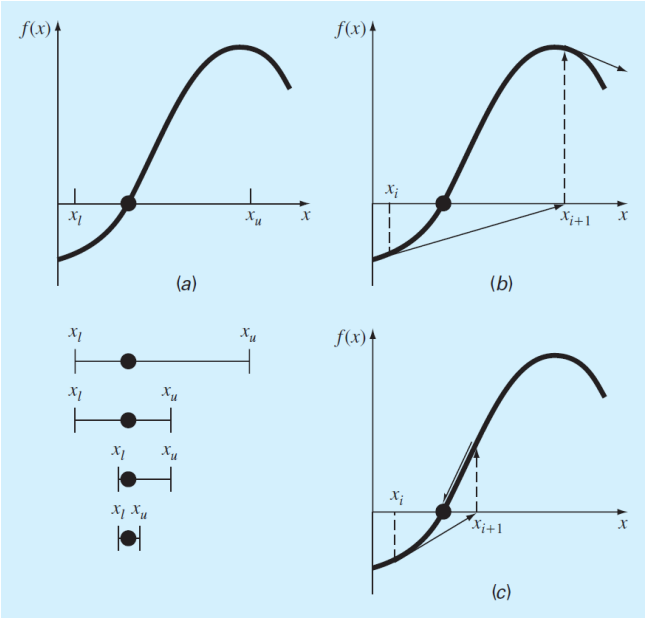


Figure 5.1: Graphical depiction of the fundamental difference between the (a) bracketing and (b) and (c) open methods for root location. In (a), which is bisection, the root is constrained within the interval prescribed by x_l and x_u . In contrast, for the open method depicted in (b) and (c), which is Newton-Raphson, a formula is used to project from x_i to $x_i + 1$ in an iterative fashion. Thus the method can either (b) diverge or (c) converge rapidly, depending on the shape of the function and the value of the initial guess.

Example 5.1. 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	$\lvert\varepsilon_a\rvert$, %	$\lvert\varepsilon_t\rvert$, %	$\lvert\varepsilon_t\rvert_i / \lvert\varepsilon_t\rvert_{i-1}$
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 (for this case, 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

$$y_1 = f_1(x) \tag{6.4}$$

and

$$y_2 = f_2(x) \tag{6.5}$$

can be plotted separately (Fig. 6.2b). The x values corresponding to the intersections of these functions represent the roots of $f(x) = 0$.

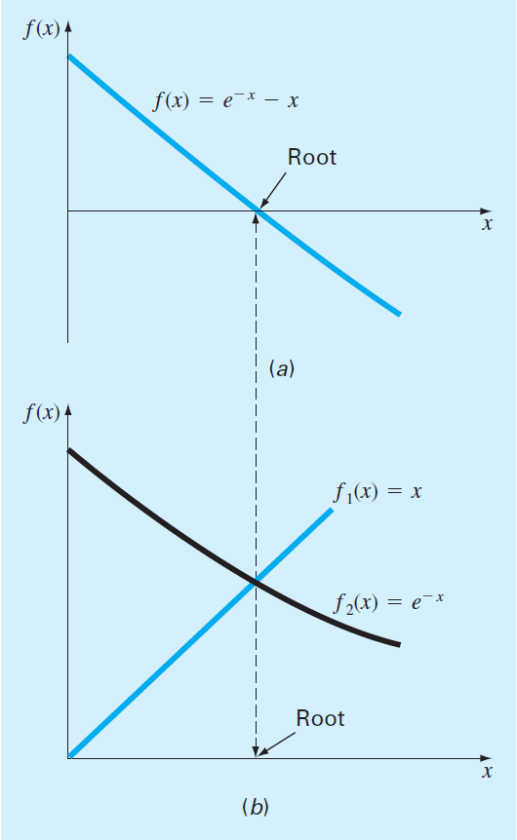


Figure 5.2: Two alternative graphical methods for determining the root of $f(x) = e^{-x} - x$. (a) Root at the point where it crosses the x axis; (b) root at the intersection of the component functions.

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 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 (2010), 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:

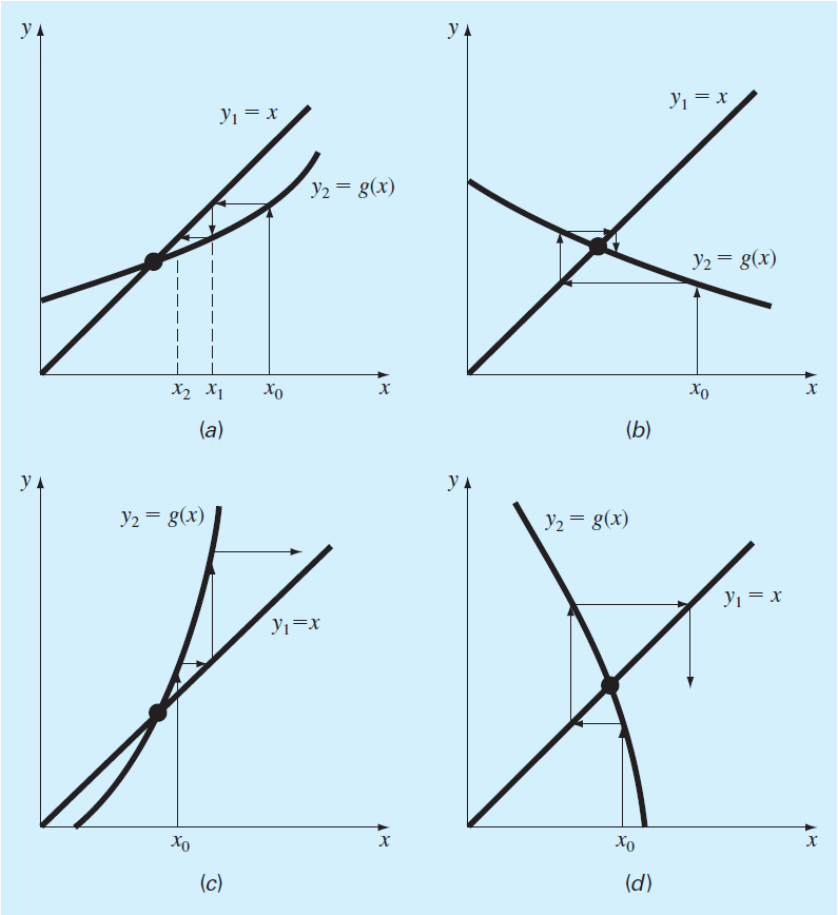


Figure 5.3: Graphical depiction of (a) and (b) convergence and (c) and (d) divergence of simple fixed-point iteration. Graphs (a) and (c) are called monotone patterns whereas (b) and (c) are called oscillating or spiral patterns. Note that convergence occurs when $|g'(x)| < 1$

$$E_{i+1} = g'(\xi)E_i$$

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).

5.2. 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

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{6.6}$$

which is called the *Newton-Raphson formula.*

Example 5.2. 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

i	x_i	$\lvert\varepsilon_t\rvert$, %
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). ■

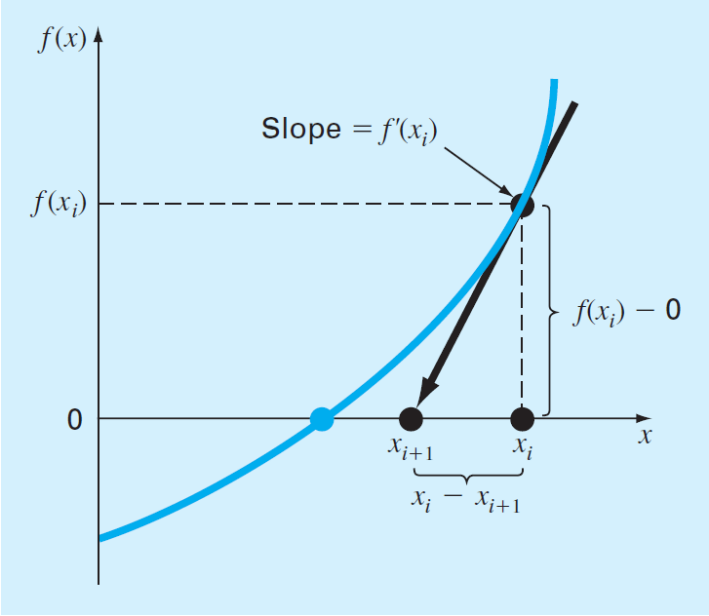


Figure 5.4: Graphical depiction of the Newton-Raphson method. A tangent to the function of x_i [that is,$f'(x)$] is extrapolated down to the x axis to provide an estimate of the root at x_{i+1}.

As with other root-location methods, Eq. (6.3) can be used as a termination criterion. In addition, a theoretical analysis (Chapra and Canale, 2010) provides insight regarding the rate of convergence as expressed by

$$E_{t,i+1} = \frac{-f''(x_r)}{2f'(x_r)} E_{t,i}^2 \tag{6.7}$$

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, 2010). However, even when dealing with simple roots, difficulties can also arise, as in the following example.

Example 5.3. 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

i	x_i	$\lvert\varepsilon_a\rvert$, %
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.

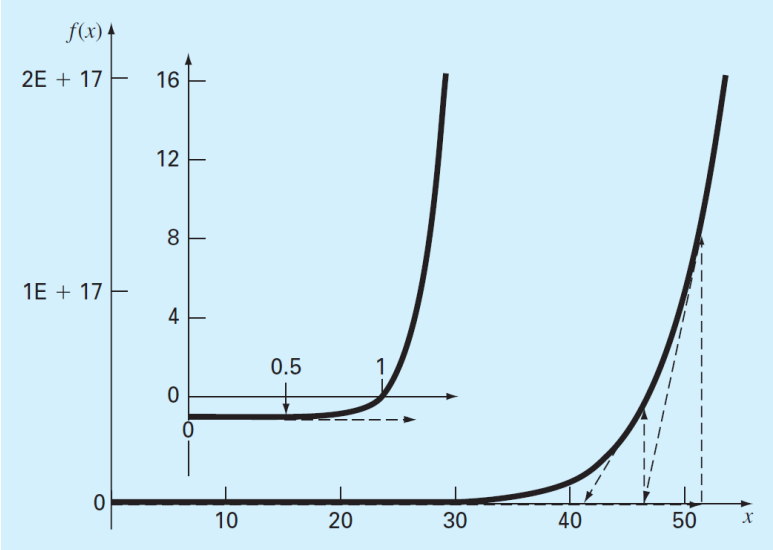


Figure 5.5: Graphical depiction of the Newton-Raphson method for a case with slow convergence. The inset shows how a near-zero slope initially shoots the solution far from the root. Thereafter, the solution very slowly converges on the root.

■

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 point (i.e., $f'(x) = 0$) occurs in the vicinity of a root. Notice that iterations beginning at x0 progressively diverge from the root. Fig. 6.6b illustrates the tendency of the Newton-Raphson technique to oscillate around a local maximum or minimum. Such oscillations may persist, or, as in Fig. 6.6b, a near-zero slope is reached whereupon the solution is sent far from the area of interest. Figure 6.6c 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. 6.6d, 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.

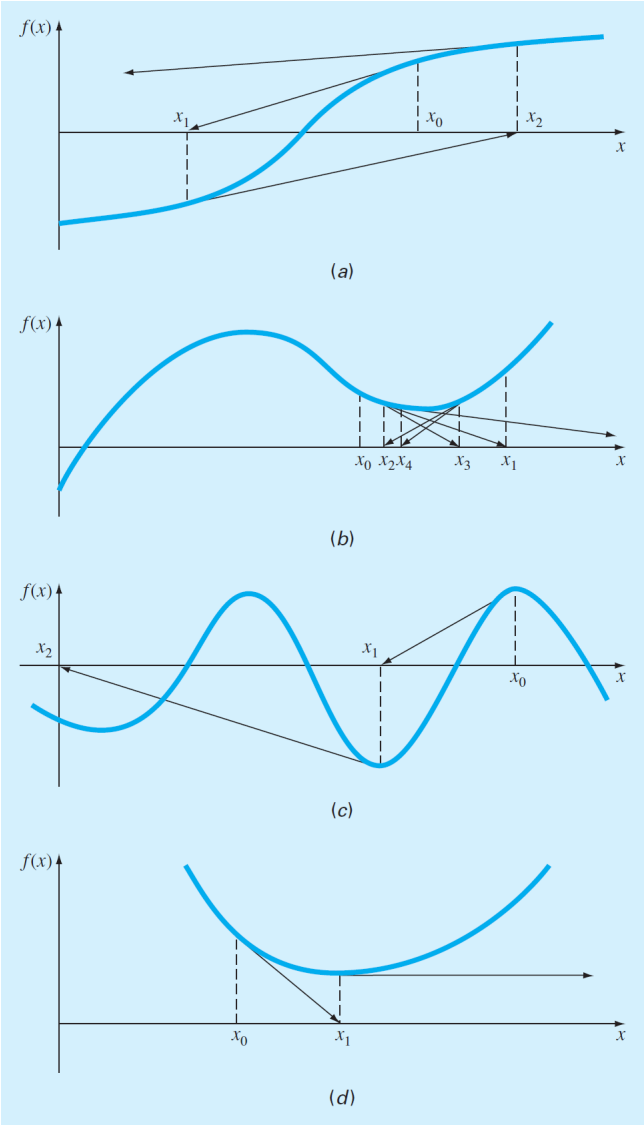


Figure 5.6: Four cases where the Newton-Raphson method exhibits poor convergence.

5.2.1. MATLAB M-file: newtraph

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(@(x) x^2-9,@(x) 2*x,5)
ans =
      3
```

Example 5.4. 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 $9.81 m/s^2$.

Solution. The function to be evaluated is

$$f(m) = \sqrt{\frac{gm}{c_d}} tanh\left(\sqrt{\frac{gc_d}{m}} t\right) - v(t) \tag{E6.4.1}$$

To apply the Newton-Raphson method, the derivative of this function must be evaluated with respect to the unknown, m:

$$\frac{df(m)}{dm} = \frac{1}{2}\sqrt{\frac{g}{mc_d}}tanh\left(\sqrt{\frac{gc_d}{m}}t\right) - \frac{g}{2m}t\ \text{sech}^2\left(\sqrt{\frac{gc_d}{m}}t\right) \tag{E6.4.2}$$

```
function [root,ea,iter]=newtraph(func,dfunc,xr,es,maxit,varargin)
% newtraph: Newton-Raphson root location zeroes
%    [root,ea,iter]=newtraph(func,dfunc,xr,es,maxit,p1,p2,...):
%          uses Newton-Raphson method to find the root of func
% input:
%    func = name of function
%    dfunc = name of derivative of function
%    xr = initial guess
%    es = desired relative error (default = 0.0001%)
%    maxit = maximum allowable iterations (default = 50)
%    p1,p2,... = additional parameters used by function
% output:
%    root = real root
%    ea = approximate relative error (%)
%    iter = number of iterations

if nargin<3,error('at least 3 input arguments required'),end
if nargin<4|isempty(es),es=0.0001;end
if nargin<5|isempty(maxit),maxit=50;end
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 5.7: An M-file to implement the Newton-Raphson method.

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.

The two formulas can now be used in conjunction with the function `newtraph` to evaluate the root:

```
» y = @m sqrt(9.81*m/0.25)*tanh(sqrt(9.81*0.25/m)*4)-36;
» dy = @m 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;
» newtraph(y,dy,140,0.00001)
ans =
      142.7376
```

■

5.3. 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:

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)} \tag{6.8}$$

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:

$$x_{i+1} = x_i - \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i)} \tag{6.9}$$

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 5.5. 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 $9.81 m/s^2$. 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:

$$x_0 = 50 \qquad f(x_0) = -4.57938708$$

$$x_0 + \delta x_0 = 50.00005 \qquad f(x_0 + \delta x_0) = -4.579381118$$

$$x_1 = 50 - \frac{10^{-6}(50)(-4.57938708))}{-4.579381118 - (-4.57938708)} = 88.39931 (|\varepsilon_t| = 38.1\%; |\varepsilon_a| = 43.4\%)$$

Second iteration:

$$x_1 = 88.39931 \qquad f(x_1) = -1.69220771$$

$$x_1 + \delta x_1 = 88.39940 \qquad f(x_1 + \delta x_1) = -1.692203516$$

$$x_2 = 88.39931 - \frac{10^{-6}(88.39931)(-1.69220771)}{-1.692203516 - (-1.69220771)} = 124.08970 (|\varepsilon_t| = 13.1\%; |\varepsilon_a| = 28.76\%)$$

The calculation can be continued to yield

i	x_i	$\lvert\varepsilon_t\rvert$, %	$\lvert\varepsilon_a\rvert$, %
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.

■

5.4. BRENT'S METHOD

Wouldn't it be nice to have a hybrid approach that combined the reliability of bracketing with the speed of the open methods? *Brent's root-location method* is a clever algorithm that does just that by applying a speedy open method wherever possible, but reverting to a reliable bracketing method if necessary. The approach was developed by Richard Brent (1973) based on an earlier algorithm of Theodorus Dekker (1969).

The bracketing technique is the trusty bisection method (Sec. 5.4), whereas two different open methods are employed. The first is the secant method described in Sec. 6.3. As explained next, the second is inverse quadratic interpolation.

5.4.1. 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. For this reason, it is sometimes referred to as a *linear interpolation method.*

Now suppose that we had three points. In that case, we could determine a quadratic function of x that goes through the three points (Fig. 6.8b). Just as with the linear secant method, the intersection of this parabola with the x axis would represent the new root estimate. And as illustrated in Fig. 6.8b, using a curve rather than a straight line often yields a better estimate.

Although this would seem to represent a great improvement, the approach has a fundamental flaw: it is possible that the parabola might not intersect the x axis! Such would be the case when the resulting parabola had complex roots. This is illustrated by the parabola, $y = f(x)$, in Fig. 6.9.

The difficulty can be rectified by employing inverse quadratic interpolation. That is, rather than using a parabola in x, we can fit the points with a parabola in y. This amounts to reversing the axes and creating a "sideways" parabola [the curve, $x = f(y)$, in Fig. 6.9].

If the three points are designated as (x_{i-2}, y_{i-2}), (x_{i-1}, y_{i-1}), and (x_i, y_i), a quadratic function of y that passes through the points can be generated as

$$g(y) = \frac{(y-y_{i-1})(y-y_i)}{(y_{i-2}-y_{i-1})(y_{i-2}-y_i)}x_{i-2} + \frac{(y-y_{i-2})(y-y_i)}{(y_{i-1}-y_{i-2})(y_{i-1}-y_i)}x_{i-1} + \frac{(y-y_{i-2})(y-y_{i-1})}{(y_i-y_{i-2})(y_i-y_{i-1})}x_i \tag{6.10}$$

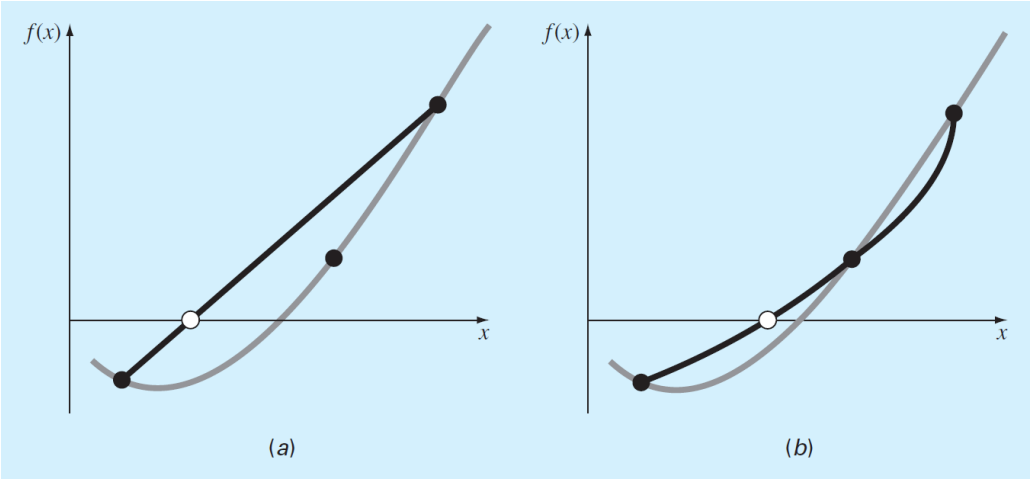


Figure 5.8: Comparison of (a) the secant method and (b) inverse quadratic interpolation. Note that the approach in (b) is called "inverse" because the quadratic function is written in y rather than in x.

As we will learn in Sec. 18.2, this form is called a *Lagrange polynomial*. The root, x_{i+1}, corresponds to $y = 0$, which when substituted into Eq. (6.10) yields

$$x_{i+1} = \frac{y_{i-1}y_i}{(y_{i-2}-y_{i-1})(y_{i-2}-y_i)}x_{i-2} + \frac{y_{i-2}y_i}{(y_{i-1}-y_{i-2})(y_{i-1}-y_i)}x_{i-1} + \frac{y_{i-2}y_{i-1}}{(y_i-y_{i-2})(y_i-y_{i-1})}x_i \tag{6.11}$$

As shown in Fig. 6.9, such a "sideways" parabola always intersects the x axis.

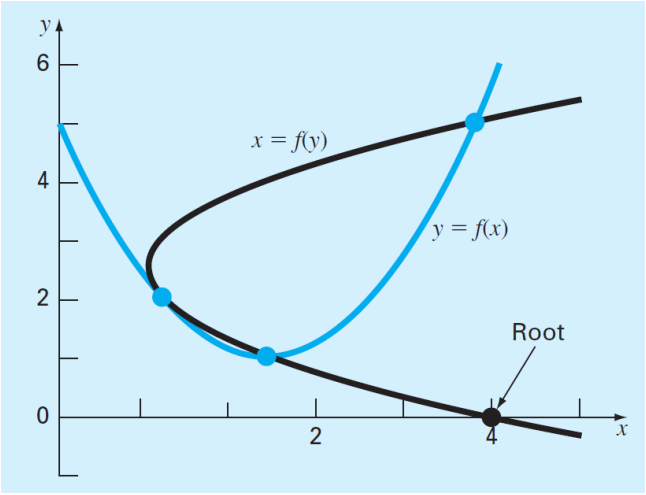


Figure 5.9: Two parabolas fit to three points. The parabola written as a function of $x, y = f(x)$, has complex roots and hence does not intersect the x axis. In contrast, if the variables are reversed, and the parabola developed as $x = f(y)$, the function does intersect the x axis.

Example 5.6. Inverse Quadratic Interpolation

Problem Statement. Develop quadratic equations in both x and y for the data points depicted in Fig. 6.9: $(1,2)$, $(2,1)$, and $(4,5)$. For the first, $y = f(x)$, employ the quadratic formula to illustrate that the roots are complex. For the latter, $x = g(y)$, use inverse quadratic interpolation (Eq. 6.11) to determine the root estimate.

Solution. By reversing the x's and y's, Eq. (6.10) can be used to generate a quadratic in x as

$$f(x) = \frac{(x-2)(x-4)}{(1-2)(1-4)}2 + \frac{(x-1)(x-4)}{(2-1)(2-4)}1 + \frac{(x-1)(x-2)}{(4-1)(4-2)}5$$

or collecting terms

$$f(x) = x^2 - 4x + 5$$

This equation was used to generate the parabola, $y = f(x)$, in Fig. 6.9. The quadratic formula can be used to determine that the roots for this case are complex,

$$x = \frac{4 \pm \sqrt{(-4)^2 - 4(1)(5)}}{2} = 2 \pm i$$

Equation (6.10) can be used to generate the quadratic in y as

$$g(y) = \frac{(y-1)(y-5)}{(2-1)(2-5)}1 + \frac{(y-2)(y-5)}{(1-2)(1-5)}2 + \frac{(y-2)(y-1)}{(5-2)(5-1)}4$$

or collecting terms:

$$g(y) = 0.5x^2 - 2.5x + 4$$

Finally, Eq. (6.11) can be used to determine the root as

$$x_{i+1} = \frac{-1(-5)}{(2-1)(2-5)}1 + \frac{-2(-5)}{(1-2)(1-5)}2 + \frac{-2(-1)}{(5-2)(5-1)}4 = 4$$

■

Before proceeding to Brent's algorithm, we need to mention one more case where inverse quadratic interpolation does not work. If the three y values are not distinct (i.e., $y_{i-2} = y_{i-1}$ or $y_{i-1} = y_i$), an inverse quadratic function does not exist. So this is where the secant method comes into play. If we arrive at a situation where the y values are not distinct, we can always revert to the less efficient secant method to generate a root using two of the points. If $y_{i-2} = y_{i-1}$, we use the secant method with x_{i-1} and x_i. If $y_{i-1} = y_i$, we use x_{i-2} and x_{i-1}.

5.4.2. Brent's Method Algorithm

The general idea behind the *Brent's root-finding method* is whenever possible to use one of the quick open methods. In the event that these generate an unacceptable result (i.e., a root estimate that falls outside the bracket), the algorithm reverts to the more conservative bisection method. Although bisection may be slower, it generates an estimate guaranteed to fall within the bracket. This process is then repeated until the root is located to within an acceptable tolerance. As might be expected, bisection typically dominates at first but as the root is approached, the technique shifts to the faster open methods.

Figure 6.10 presents a function based on a MATLAB M-file developed by Cleve Moler (2004). It represents a stripped down version of the `fzero` function which is the professional root-location function employed in MATLAB. For that reason, we call the simplified version: `fzerosimp`. Note that it requires another function `f` that holds the equation for which the root is being evaluated.

The `fzerosimp` function is passed two initial guesses that must bracket the root. Then, the three variables defining the search interval `(a,b,c)` are initialized, and `f` is evaluated at the endpoints.

```
function b = fzerosimp(xl,xu)
a = xl; b = xu; fa = f(a); fb = f(b);
c = a; fc = fa; d = b - c; e = d;
while (1)
   if fb == 0, break, end
   if sign(fa) == sign(fb) %If needed, rearrange points
      a = c; fa = fc; d = b - c; e = d;
   end
   if abs(fa) < abs(fb)
      c = b; b = a; a = c;
      fc = fb; fb = fa; fa = fc;
   end
   m = 0.5*(a - b); %Termination test and possible exit
   tol = 2 * eps * max(abs(b), 1);
   if abs(m) <= tol | fb == 0.
      break
   end
   %Choose open methods or bisection
   if abs(e) >= tol & abs(fc) > abs(fb)
      s = fb/fc;
      if a == c %Secant method
         p = 2*m*s;
         q = 1 - s;
      else %Inverse quadratic interpolation
         q = fc/fa; r = fb/fa;
         p = s * (2*m*q * (q - r) - (b - c)*(r - 1));
         q = (q - 1)*(r - 1)*(s - 1);
      end
      if p > 0, q = -q; else p = -p; end;
      if 2*p < 3*m*q - abs(tol*q) & p < abs(0.5*e*q)
         e = d; d = p/q;
      else
         d = m; e = m;
      end
   else %Bisection
      d = m; e = m;
   end
   c = b; fc = fb;
   if abs(d) > tol, b=b+d; else b=b-sign(b-a)*tol; end
   fb = f(b);
end
```

Figure 5.10: Function for Brent's root-finding algorithm based on a MATLAB M-file developed by Cleve Moler (2005).

A main loop is then implemented. If necessary, the three points are rearranged to satisfy the conditions required for the algorithm to work effectively. At this point, if the stopping criteria are met, the loop is terminated. Otherwise, a decision structure chooses among the three methods and checks whether the outcome is acceptable. Afinal section then evaluates `f` at the new point and the loop is repeated. Once the stopping criteria are met, the loop terminates and the final root estimate is returned.

5.5. MATLAB FUNCTION: FZERO

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(@(x) x^2-9,-4)
x =
      3
```

If we want to find the positive root, use a guess that is near it:

```
» x = fzero(@(x) x^2-9,4
)   x =
      3
```

If we put in an initial guess of zero, it finds the negative root:

```
» x = fzero(@(x) 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(@(x) 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(@(x) x^2-9,[-4 4])
???  Error using ==> fzero
The function values at the interval endpoints must differ in sign.
```

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 an unacceptable 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.

A more complete representation of the fzero syntax can be written as

```
[x,fx] = fzero(function,x0,options,p1,p2,...)
```

where `[x,fx]` = 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...` 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_i has the value val_i. 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 5.7. 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(@(x) 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(@(x) x^10-1,0.5,options)
x =
      1.0009
fx =
      0.0090
```

■

5.6. POLYNOMIALS

Polynomials are a special type of nonlinear algebraic equation of the general form

$$f_n(x) = a_1x^n + a_2x^{n-1} + \cdots + a_{n-1}x^2 + a_nx + a_{n+1} \tag{6.12}$$

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 nth 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.

5.6.1. MATLAB Function: `roots`

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 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

$$a_1x^5 + a_2x^4 + a_3x^3 + a_4x^2 + a_5x + a_6 = 0 \tag{6.13}$$

Dividing by a_1 and rearranging yields

$$x^5 = -\frac{a_2}{a_1}x^4 - \frac{a_3}{a_1}x^3 - \frac{a_4}{a_1}x^2 - \frac{a_5}{a_1}x - \frac{a_6}{a_1}$$

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:

$$\begin{bmatrix} -a_2/a_1 & -a_3/a_1 & -a_4/a_1 & -a_5/a_1 & -a_6/a_1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{hfill (6.14)}$$

Equation (6.14) 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 5.8. 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:

$$f_5(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25 \tag{E6.8.1}$$

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. (E6.8.1): 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)
```

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. (E6.8.1) 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
```

■

5.7. CASE STUDY - PIPE FRICTION

Background. Determining fluid flow through pipes and tubes has great relevance in many areas of engineering and science. In engineering, typical applications include the flow of liquids and gases through pipelines and cooling systems. Scientists are interested in topics ranging from flow in blood vessels to nutrient transmission through a plant's vascular system.

The resistance to flow in such conduits is parameterized by a dimensionless number called the *friction factor*. For turbulent flow, the *Colebrook equation* provides a means to calculate the friction factor:

$$0 = \frac{1}{\sqrt{f}} + 2.0\log\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{\mathrm{Re}\sqrt{f}}\right) \tag{6.15}$$

where ε = the roughness (m), D = diameter (m), and Re = the Reynolds number:

$$\mathrm{Re} = \frac{\rho V D}{\mu}$$

where ρ = the fluid's density $\left(\mathrm{kg/m^3}\right)$, V = its velocity (m/s), and μ = dynamic viscosity $\left(\mathrm{N\cdot s/m^2}\right)$. In addition to appearing in Eq. (6.15), the Reynolds number also serves as the criterion for whether flow is turbulent ($\mathrm{Re} > 4000$).

In this case study, we will illustrate how the numerical methods covered in this part of the book can be employed to determine f for air flow through a smooth, thin tube. For this case, the parameters are $\rho = 1.23\ \mathrm{kg/m^3}$, $\mu = 1.79 \times 10^{-5}\ \mathrm{N\cdot s/m^2}$, $D = 0.005$ m, $V = 40$ m/s and $\varepsilon = 0.0015$ mm. Note that friction factors range from about 0.008 to 0.08. In addition, an explicit formulation called the Swamee-Jain equation provides an approximate estimate:

$$f = \frac{1.325}{\left[\ln\left(\frac{\varepsilon}{3.7D} + \frac{5.74}{\mathrm{Re}^{0.9}}\right)\right]^2} \tag{6.16}$$

Solution. The Reynolds number can be computed as

$$\mathrm{Re} = \frac{\rho V D}{\mu} = \frac{1.23(40)0.005}{1.79 \times 10^{-5}} = 13,743$$

This value along with the other parameters can be substituted into Eq. (6.15) to give

$$g(f) = \frac{1}{\sqrt{f}} + 2.0\log\left(\frac{0.0000015}{3.7(0.005)} + \frac{2.51}{13,743\sqrt{f}}\right)$$

Before determining the root, it is advisable to plot the function to estimate initial guesses and to anticipate possible difficulties. This can be done easily with MATLAB:

```
>> rho=1.23;mu=1.79e-5;D=0.005;V=40;e=0.0015/1000;
>> Re=rho*V*D/mu;
>> g=@(f) 1/sqrt(f)+2*log10(e/(3.7*D)+2.51/(Re*sqrt(f)));
>> fplot(g,[0.008 0.08]),grid,xlabel('f'),ylabel('g(f)')
```

As in Fig. 6.11, the root is located at about 0.03.

Because we are supplied initial guesses ($x_l = 0.008$ and $x_u = 0.08$), either of the bracketing methods from Chap. 5 could be used. For example, the bisect function developed in Fig. 5.7 gives a value of $f = 0.0289678$ with a percent relative error of error of 5.926×10^{-5} in 22 iterations. False position yields a result of similar precision in 26 iterations. Thus, although they produce the correct result, they are somewhat inefficient. This would not be important for a single application, but could become prohibitive if many evaluations were made.

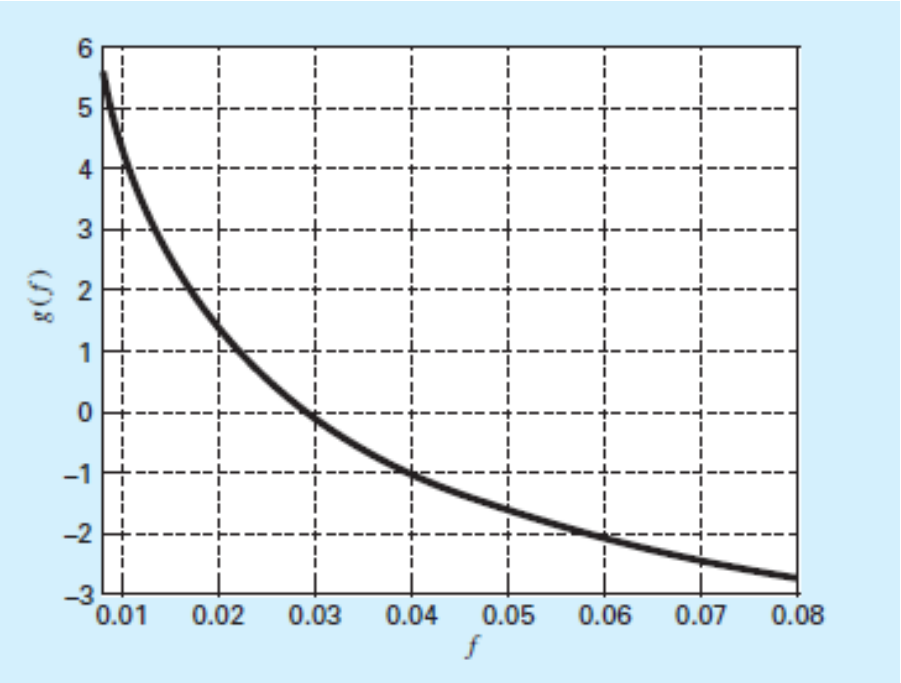


Figure 6.11

We could try to attain improved performance by turning to an open method. Because Eq. (6.15) is relatively straightforward to differentiate, the Newton-Raphson method is a good candidate. For example, using an initial guess at the lower end of the range ($x_0 = 0.008$), the newt raph function developed in Fig. 6.7 converges quickly:

```
>> dg=@(f) -2/log(10)*1.255/Re*f^(-3/2)/(e/D/3.7 ...
                          +2.51/Re/sqrt(f))-0.5/f^(3/2);
>> [f ea iter]=newtraph(g,dg,0.008)
f =
    0.02896781017144
ea =
    6.870124190058040e-006
iter =
    6
```

However, when the initial guess is set at the upper end of the range ($x_0 = 0.08$), the routine diverges,

```
>> [f ea iter]=newtraph(g,dg,0.08)
f =
    NaN + NaNi
```

As can be seen by inspecting Fig. 6.11, this occurs because the function's slope at the initial guess causes the first iteration to jump to a negative value. Further runs demonstrate that for this case, convergence only occurs when the initial guess is below about 0.066.

So we can see that although the Newton-Raphson is very efficient, it requires good initial guesses. For the Colebrook equation, a good strategy might be to employ the Swamee-Jain equation (Eq. 6.16) to provide the initial guess as in

```
>> fSJ=1.325/log(e/(3.7*D)+5.74/Re^0.9)^2
fSJ =
    0.02903099711265
>> [f ea iter]=newtraph(g,dg,fSJ)
f =
    0.02896781017144
ea =
    8.510189472800060e-010
iter =
    3
```

Aside from our homemade functions, we can also use MATLAB's built-in `fzero` function. However, just as with the Newton-Raphson method, divergence also occurs when `fzero` function is used with a single guess. However, in this case, guesses at the lower end of the range cause problems. For example,

```
>> fzero(g,0.008)

Exiting fzero: aborting search for an interval containing a sign
change because complex function value encountered ...
                                  during search.
(Function value at -0.0028 is -4.92028-20.2423i.)
Check function or try again with a different starting value.
ans =
    NaN
```

If the iterations are displayed using optimset (recall Example 6.7), it is revealed that a negative value occurs during the search phase before a sign change is detected and the routine aborts. However, for single initial guesses above about 0.016, the routine works nicely. For example, for the guess of 0.08 that caused problems for Newton-Raphson, f zero does just fine:

```
>> fzero(q,0.08)
ans =
    0.02896781017144
```

As a final note, let's see whether convergence is possible for simple fixed-point iteration. The easiest and most straightforward version involves solving for the first f in Eq. (6.15):

$$f_{i+1} = \frac{0.25}{\left(\log\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{\mathrm{Re}\sqrt{f_i}}\right)\right)^2} \tag{6.17}$$

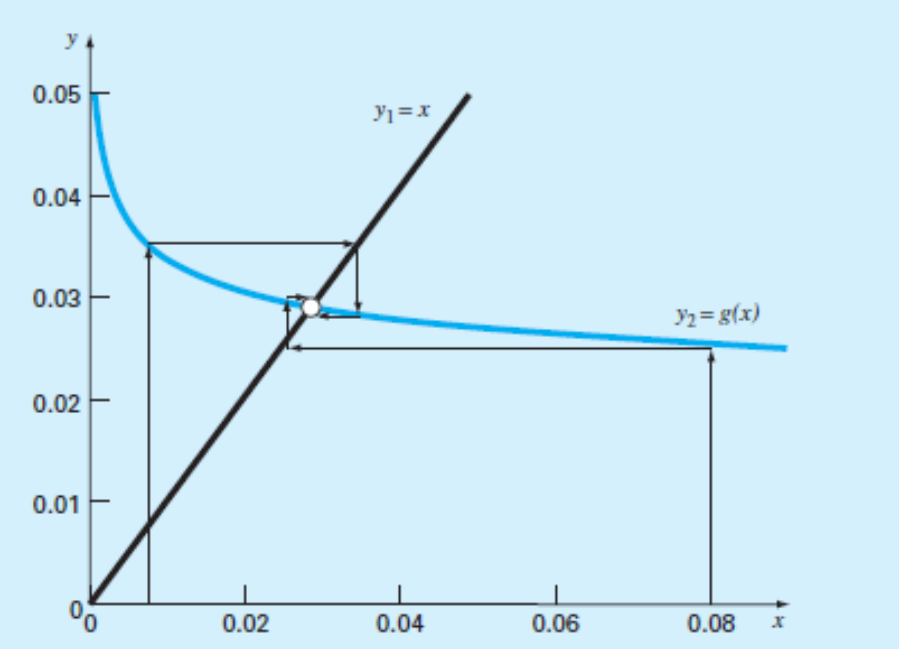


Figure 6.12

The two-curve display of this function depicted indicates a surprising result (Fig. 6.12). Recall that fixed-point iteration converges when the y_2 curve has a relatively flat slope (i.e., $|g'(\xi)| < 1$). As indicated by Fig. 6.12, the fact that the y_2 curve is quite flat in the range from $f = 0.008$ to 0.08 means that not only does fixed-point iteration converge, but it converges fairly rapidly! In fact, for initial guesses anywhere between 0.008 and 0.08, fixedpoint iteration yields predictions with percent relative errors less than 0.008% in six or fewer iterations! Thus, this simple approach that requires only one guess and no derivative estimates performs really well for this particular case.

The take-home message from this case study is that even great, professionally developed software like MATLAB is not always foolproof. Further, there is usually no single method that works best for all problems. Sophisticated users understand the strengths and weaknesses of the available numerical techniques. In addition, they understand enough of the underlying theory so that they can effectively deal with situations where a method breaks down.

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 $\varepsilon_a \leq 0.01\%$. Verify that the process is linearly convergent as described at the end of Sec. 6.1.

6.2 Use (a) fixed-point iteration and (b) the NewtonRaphson 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, $x_i = 3.5, \delta = 0.01$).

(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, $x_{i-1} = 0.5$ and $x_i = 0.4$.

(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 $f(x) = \tanh\left(x^2 - 9\right)$ 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 Mechanical engineers, as well as most other engineers, use thermodynamics extensively in their work. The following polynomial can be used to relate the zero-pressure specific heat of dry air c_p in kJ/(kgK) to temperature in K :

$$c_p = 0.99403 + 1.671 \times 10^{-4} T + 9.7215 \times 10^{-8} T^2 - 9.5838 \times 10^{-11} T^3 + 1.9520 \times 10^{-14} T^4$$

Write a MATLAB script (a) to plot c_p versus a range of $T = 0$ to 1200 K, and (b) to determine the temperature that corresponds to a specific heat of 1.1 kJ/(kgK) with MATLAB polynomial functions.

6.14 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) :

$$H_2O \rightleftarrows H_2 + \frac{1}{2}O_2$$

If it is assumed that this is the only reaction involved, the mole fraction x of H_2O that dissociates can be represented by

$$K = \frac{x}{1-x}\sqrt{\frac{2p_t}{2+x}} \qquad \text{(P6.14.1)}$$

where K is the reaction's equilibrium constant and p_t is the total pressure of the mixture. If $p_t = 3$ atm and $K = 0.05$, determine the value of x that satisfies Eq. (P6.14.1).

6.15 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/(kgK)], T = absolute temperature (K), p = absolute pressure (kPa), and v = the volume of a kg of gas $\left(\mathrm{m}^3/\mathrm{kg}\right)$. The parameters a and b are calculated by

$$a = 0.427\frac{R^2 T_c^{2.5}}{p_c} \qquad b = 0.0866R\frac{T_c}{p_c}$$

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 65,000kPa. Use a root-locating method of your choice to calculate v and then determine the mass of methane contained in the tank.

6.16 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, $L = 5$ m^3, and $V = 8$ m^3.

6.17 A catenary cable is one which is hung between two points not in the same vertical line. As depicted in Fig. P6.17a, 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. P6.17b, where T_A and T_B are 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 .

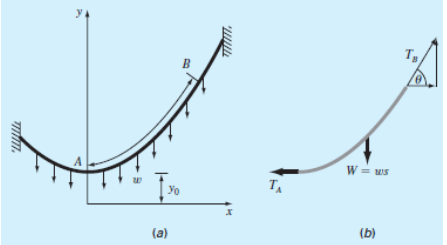


Figure P6.17

6.18 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 $I = 3.5$

6.19 Figure P6.19 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: $R = 225\Omega, C = 0.6 \times 10^{-6}$ F, and $L = 0.5$H.

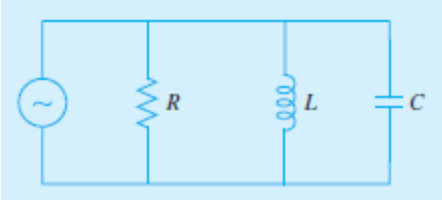


Figure P6.19

6.20 Real mechanical systems may involve the deflection of nonlinear springs. In Fig. P6.20, a block of mass m is released a distance h above a nonlinear spring. The resistance force F of the spring is given by

$$F = -\left(k_1 d + k_2 d^{3/2}\right)$$

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: $k_1 = 40{,}000$ g/s^2, $k_2 = 40$ g/ (s^2 m^5), $m = 95$ g, g = 9.81 m/s^2 and $h = 0.43$ m.

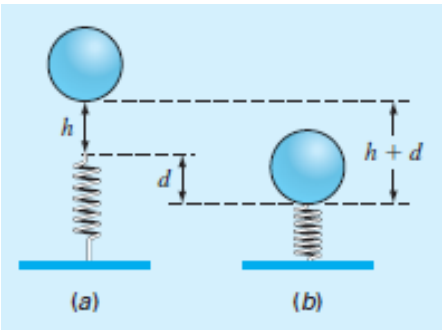


Figure P6.20

6.21 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. P6.21. The trajectory can be modeled as

$$y = (\tan\theta_0)x - \frac{g}{2v_0^2\cos^2\theta_0}x^2 + y_0$$

Find the appropriate initial angle θ_0, if $v_0 = 30$ m/s, and the distance to the catcher 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.

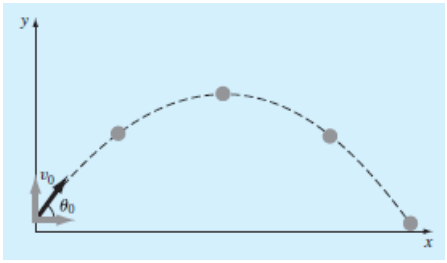


Figure P6.21

6.22 You are designing a spherical tank (Fig. P6.22) 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 $\left[\text{m}^3\right]$, h = depth of water in tank [m], and R = the tank radius [m].

If $R = 3$ m, what depth must the tank be filled to so that it holds 30 m^3 ? Use three iterations of the most efficient numerical method possible to determine your answer. Determine 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.

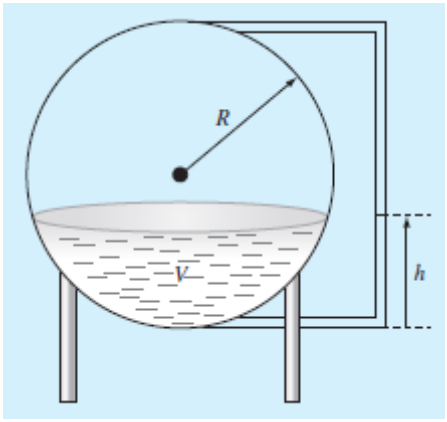


Figure P6.22

6.23 Perform the identical MATLAB operations as those in Example 6.8 to manipulate and find all the roots of the polynomial

$$f_5(x) = (x+2)(x+5)(x-6)(x-4)(x-8)$$

6.24 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.

6.25 The Manning equation can be written for a rectangular open channel as

$$Q = \frac{\sqrt{S}(BH)^{5/3}}{n(B+2H)^{2/3}}$$

where $Q =$ flow $\left(\mathrm{m}^3/\mathrm{s}\right), S =$ slope $(\mathrm{m}/\mathrm{m}), H =$ depth(m), and $n =$ the Manning roughness coefficient. Develop a fixed-point iteration scheme to solve this equation for H given $Q = 5, S = 0.0002, B = 20$, and $n = 0.03$. Perform the computation until ε_α is less than $\varepsilon_s = 0.05\%$. Prove that your scheme converges for all initial guesses greater than or equal to zero.

6.26 See if you can develop a foolproof function to compute the friction factor based on the Colebrook equation as described in Sec. 6.7. Your function should return a precise result for Reynolds number ranging from 4000 to 10^7 and for ε/D ranging from 0.00001 to 0.05.

6.27 Use the Newton-Raphson method to find the root of $f(x) = e^{-0.5x}(4-x) - 2$

Employ initial guesses of (a) 2 , (b) 6 , and (c) 8 Explain your results.

6.28 Given

$$f(x) = -2x^6 - 1.5x^4 + 10x + 2$$

Use a root-location technique to determine the maximum of this function. Perform iterations until the approximate relative error falls below 5%. If you use a bracketing method, use initial guesses of $x_l = 0$ and $x_u = 1$. If you use the Newton-Raphson or the modified secant method, use an initial guess of $x_i = 1$. If you use the secant method, use initial guesses of $x_{i-1} = 0$ and $x_i = 1$. Assuming that convergence is not an issue, choose the technique that is best suited to this problem. Justify your choice.

6.29 You must determine the root of the following easily differentiable function:

$$e^{0.5x} = 5 - 5x$$

Pick the best numerical technique, justify your choice, and then use that technique to determine the root. Note that it is known that for positive initial guesses, all techniques except fixed-point iteration will eventually converge. Perform iterations until the approximate relative error falls below 2%. If you use a bracketing method, use initial guesses of $x_I = 0$ and $x_u = 2$. If you use the Newton-Raphson or the modified secant method, use an initial guess of $x_i = 0.7$. If you use the secant method, use initial guesses of $x_{i-1} = 0$ and $x_i = 2$.

6.30 (a) Develop an M-file function to implement Brent's root-location method. Base your function on Fig. 6.10, but with the beginning of the function changed to

```
function [b,fb] = fzeronew(f,xl,xu,
varargin)
% fzeronew: Brent root location
zeroes
% [b,fb] = fzeronew(f,xl,xu,p1,p2
,...):
% uses Brent's method to find the
root of f
% input:
% f = name of function
% xl, xu = lower and upper guesses
% p1,p2,... = additional parameters
used by f
% output:
% b = real root
% fb = function value at root
```

Make the appropriate modifications so that the function performs as outlined in the documentation statements. In addition, include error traps to ensure that the function's three required arguments `(f, x 1, x u)` are prescribed, and that the initial guesses bracket a root. (b) Test your function by using it to solve for the root of the function from Example 5.6 using

```
>> [x,fx] = fzeronew(@(x,n) x^n
-1,0,1.3,10)
```

Chapter 6

Optimization

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to how optimization can be used to determine minima and maxima of both one-dimensional and multidimensional functions. Specific objectives and topics covered are

- Understanding why and where optimization occurs in engineering and scientific problem solving.
- Recognizing the difference between one-dimensional and multidimensional optimization.
- Distinguishing between global and local optima.
- Knowing how to recast a maximization problem so that it can be solved with a minimizing algorithm.
- Being able to define the golden ratio and understand why it makes onedimensional optimization efficient.
- Locating the optimum of a single-variable function with the golden-section search.
- Locating the optimum of a single-variable function with parabolic interpolation.
- Knowing how to apply the `fminbnd` function to determine the minimum of a one-dimensional function.
- Being able to develop MATLAB contour and surface plots to visualize twodimensional functions.
- Knowing how to apply the `fminsearch` function to determine the minimum of a multidimensional function.

YOU'VE GOT A PROBLEM

An object like a bungee jumper can be projected upward at a specified velocity. If it is subject to linear drag, its altitude as a function of time can be computed as

$$z = z_0 + \frac{m}{c}(v_0 + \frac{mg}{c})(1 - e^{-(c/m)t}) - \frac{mg}{c}t \tag{7.1}$$

Order n	$f^{(n)}(x)$	$f(\pi/3)$	$\lvert\varepsilon_t\rvert$
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}

Figure 6.1: Elevation as a function of time for an object initially projected upward with an initial velocity.

where z = altitude (m) above the earth's surface (defined as $z = 0$), z_0 = the initial altitude (m), m = mass (kg), c = a linear drag coefficient (kg/s), v_0 = initial velocity (m/s), and t = time (s). Note that for this formulation, positive velocity is considered to be in the upward direction. Given the following parameter values: $g = 9.81 m/s^2$, z_0 = 100 m, v_0 = 55 m/s, m = 80 kg, and c = 15 kg/s, Eq. (7.1) can be used to calculate the jumper's altitude. As displayed in Fig. 7.1, the jumper rises to a peak elevation of about 190 m at about t = 4 s. Suppose that you are given the job of determining the exact time of the peak elevation. The determination of such extreme values is referred to as optimization. This chapter will introduce you to how the computer is used to make such determinations.

6.1. INTRODUCTION AND BACKGROUND

In the most general sense, optimization is the process of creating something that is as effective as possible. As engineers, we must continuously design devices and products that perform tasks in an efficient fashion for the least cost. Thus, engineers are always confronting optimization problems that attempt to balance performance and limitations. In addition, scientists have interest in optimal phenomena ranging from the peak elevation of projectiles to the minimum free energy.

From a mathematical perspective, optimization deals with finding the maxima and minima of a function that depends on one or more variables. The goal is to determine the values of the variables that yield maxima or minima for the function. These can then be substituted back into the function to compute its optimal values.

Although these solutions can sometimes be obtained analytically, most practical optimization problems require numerical, computer solutions. From a numerical standpoint, optimization is similar in spirit to the root-location methods we just covered in Chaps. 5 and 6. That is, both involve guessing and searching for a point on a function. The fundamental difference between the two types of problems is illustrated in Fig. 7.2. Root location involves searching for the location where the function equals zero. In contrast, optimization involves searching for the function's extreme points.

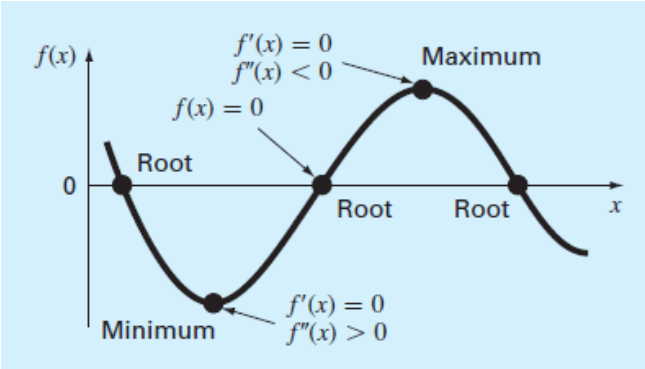


Figure 6.2: A function of a single variable illustrating the difference between roots and optima.

As can be seen in Fig. 7.2, the optimums are the points where the curve is flat. In mathematical terms, this corresponds to the x value where the derivative $f'(x)$ is equal to zero. Additionally, the second derivative, $f''(x)$, indicates whether the optimum is a minimum or a maximum: if $f''(x) > 0$, the point is a maximum; if $f''(x) > 0$, the point is a minimum.

Now, understanding the relationship between roots and optima would suggest a possible strategy for finding the latter. That is, you can differentiate the function and locate the root (i.e., the zero) of the new function. In fact, some optimization methods do just this by solving the root problem: $f''(x) = 0$.

Example 6.1. Determining the Optimum Analytically by Root Location

Problem Statement: Determine the time and magnitude of the peak elevation based on Eq. (7.1). Use the following parameter values for your calculation: $g = 9.81 m/s^2$, z0 = 100 m, v0 = 55 m/s, m = 80 kg, and c = 15 kg/s.
Solution: Equation (7.1) can be differentiated to give.

$$\frac{dz}{dt} = v_0 e^{-(c/m)t} - \frac{mg}{c}(1 - e^{-(c/m)t}) \qquad \text{E7.1.1}$$

Note that because $v = dz/dt$, this is actually the equation for the velocity. The maximum elevation occurs at the value of t that drives this equation to zero. Thus, the problem amounts to determining the root. For this case, this can be accomplished by setting the derivative to zero and solving Eq. (E7.1.1) analytically for

$$t = \frac{m}{c} ln(1 + \frac{cv_0}{mg})$$

Substituting the parameters gives

$$t = \frac{80}{15} ln(1 + \frac{15(55)}{80(9.81)}) = 3.83166s$$

This value along with the parameters can then be substituted into Eq. (7.1) to compute the maximum elevation as

$$z = 100 + \frac{80}{15}(50 + \frac{80(9.81)}{15})(1 - e^{-(15/80)3.83166}) - \frac{80(9.81)}{15}(3.83166) = 192.8609m$$

We can verify that the result is a maximum by differentiating Eq. (E7.1.1) to obtain the second derivative

$$\frac{d^2z}{dt^2} = -\frac{c}{m} v_0 e^{-(c/m)t} - g e^{-(c/m)t} = -9.81 \frac{m}{s^2}$$

The fact that the second derivative is negative tells us that we have a maximum. Further, the result makes physical sense since the acceleration should be solely equal to the force of gravity at the maximum when the vertical velocity (and hence drag) is zero.

Although an analytical solution was possible for this case, we could have obtained the same result using the root-location methods described in Chaps. 5 and 6. This will be left as a homework exercise.

■

Although it is certainly possible to approach optimization as a roots problem, a variety of direct numerical optimization methods are available. These methods are available for both one-dimensional and multidimensional problems. As the name implies, one-dimensional problems involve functions that depend on a single dependent variable. As in Fig. 7.3a, the search then consists of climbing or descending one-dimensional peaks and valleys. Multidimensional problems involve functions that depend on two or more dependent variables.

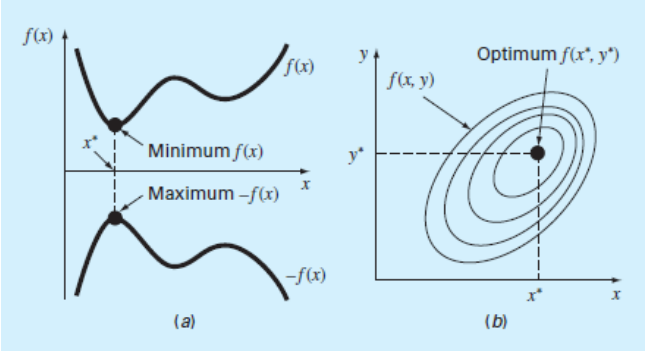


Figure 6.3: (a) One-dimensional optimization. This figure also illustrates how minimization of $f(x)$ is equivalent to the maximization of $-f(x)$. (b) Two-dimensional optimization. Note that this figure can be taken to represent either a maximization (contours increase in elevation up to the maximum like a mountain) or a minimization (contours decrease in elevation down to the minimum like a valley).

In the same spirit, a two-dimensional optimization can again be visualized as searching out peaks and valleys (Fig. 7.3b). However, just as in real hiking, we are not constrained to walk a single direction; instead the topography is examined to efficiently reach the goal.

Finally, the process of finding a maximum versus finding a minimum is essentially identical because the same value $x*$ both minimizes $f(x)$ and maximizes $-f(x)$. This equivalence is illustrated graphically for a one-dimensional function in Fig. 7.3a.

In the next section, we will describe some of the more common approaches for onedimensional optimization. Then we will provide a brief description of how MATLAB can be employed to determine optima for multidimensional functions.

6.2. ONE-DIMENSIONAL OPTIMIZATION

This section will describe techniques to find the minimum or maximum of a function of a single variable $f(x)$. A useful image in this regard is the one-dimensional "roller coaster" -like function depicted in Fig. 7.4. Recall from Chaps. 5 and 6 that root location was complicated by the fact that several roots can occur for a single function. Similarly, both local and global optima can occur in optimization.

A *global optimum* represents the very best solution. A *local optimum*, though not the very best, is better than its immediate neighbors. Cases that include local optima are called *multimodal*. In such cases, we will almost always be interested in finding the global optimum. In addition, we must be concerned about mistaking a local result for the global optimum.

Just as in root location, optimization in one dimension can be divided into bracketing and open methods. As described in the next section, the golden-section search is an example of a bracketing method that is very similar in spirit to the bisection method for root location. This is followed by a somewhat more sophisticated bracketing approach-parabolic interpolation. We will then show how these two methods are combined and implemented with MATLAB's `fminbnd` function.

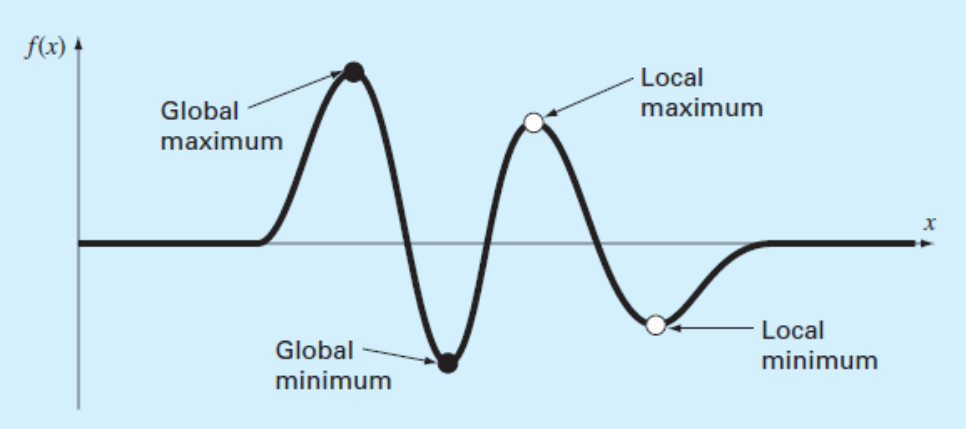


Figure 6.4: (a) A function that asymptotically approaches zero at plus and minus ∞ and has two maximum and two minimum points in the vicinity of the origin. The two points to the right are local optima, whereas the two to the left are global.

6.2.1. Golden-Section Search

In many cultures, certain numbers are ascribed magical qualities. For example, we in theWest are all familiar with "lucky 7" and "Friday the 13th." Beyond such superstitious quantities, there are several well-known numbers that have such interesting and powerful mathematical properties that they could truly be called "magical". The most common of these are the ratio of a circle's circumference to its diameter π and the base of the natural logarithm e.

Although not as widely known, the golden ratio should surely be included in the pantheon of remarkable numbers. This quantity, which is typically represented by the Greek letter ϕ (pronounced: fee), was originally defined by Euclid (ca. 300 BCE) because of its role in the construction of the pentagram or five-pointed star. As depicted in Fig. 7.5, Euclid's definition reads: "A straight line is said to have been cut in extreme and mean ratio when, as the whole line is to the greater segment, so is the greater to the lesser."

The actual value of the golden ratio can be derived by expressing Euclid's definition as

$$\frac{l_1+l_2}{l_1}=\frac{l_1}{l_2} \tag{7.2}$$

Multiplying by l_1/l_2 and collecting terms yields

$$\phi^2-\phi-1=0 \tag{7.3}$$

where $\phi=l_1/l_2$. The positive root of this equation is the golden ratio:

$$\phi=\frac{1+\sqrt{5}}{2}=1.61803398874989... \tag{7.4}$$

The golden ratio has long been considered aesthetically pleasing in Western cultures. In addition, it arises in a variety of other contexts including biology. For our purposes, it provides the basis for the golden-section search, a simple, general-purpose method for determining the optimum of a single-variable function.

The golden-section search is similar in spirit to the bisection approach for locating roots in Chap. 5. Recall that bisection hinged on defining an interval, specified by a lower guess (x_l) and an upper guess (x_u) that bracketed a single root. The presence of a root between these bounds was verified by determining that $f(x_l)$ and $f(x_u)$ had different signs. The root was then estimated as the midpoint of this interval:

$$x_r = \frac{x_l + x_u}{2} \tag{7.5}$$

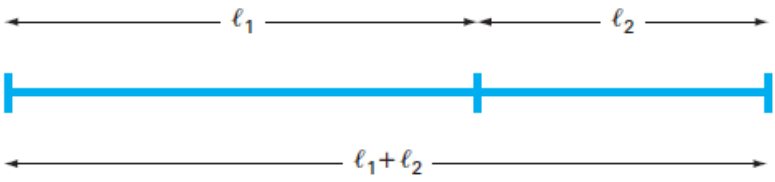


Figure 6.5: Euclid's definition of the golden ratio is based on dividing a line into two segments so that the ratio of the whole line to the larger segment is equal to the ratio of the larger segment to the smaller segment. This ratio is called the golden ratio.

The final step in a bisection iteration involved determining a new smaller bracket. This was done by replacing whichever of the bounds x"$_l$ or x_u had a function value with the same sign as $f(x_r)$. Akey advantage of this approach was that the new value x_r replaced one of the old bounds.

Now suppose that instead of a root, we were interested in determining the minimum of a one-dimensional function. As with bisection, we can start by defining an interval that contains a single answer. That is, the interval should contain a single minimum, and hence is called *unimodal*. We can adopt the same nomenclature as for bisection, where x_l and x_u defined the lower and upper bounds, respectively, of such an interval. However, in contrast to bisection, we need a new strategy for finding a minimum within the interval. Rather than using a single intermediate value (which is sufficient to detect a sign change, and hence a zero), we would need two intermediate function values to detect whether a minimum occurred.

The key to making this approach efficient is the wise choice of the intermediate points. As in bisection, the goal is to minimize function evaluations by replacing old values with new values. For bisection, this was accomplished by choosing the midpoint. For the golden-section search, the two intermediate points are chosen according to the golden ratio:

$$x_1 = x_l + d \tag{7.6}$$

$$x_2 = x_u - d \tag{7.7}$$

where

$$d = (\phi - 1)(x_u - x_l) \tag{7.8}$$

The function is evaluated at these two interior points. Two results can occur:

1. If, as in Fig. 7.6a, $f(x_1) < f(x_2)$, then $f(x_1)$ is the minimum, and the domain of x to the left of x_2, from x_l to x_2, can be eliminated because it does not contain the minimum. For this case, x_2 becomes the new x_l for the next round.

2. If $f(x_2) < f(x_1)$, then $f(x_2)$ is the minimum and the domain of x to the right of x_1, from x_1 to x_u would be eliminated. For this case, x_1 becomes the new x_u for the next round.

Now, here is the real benefit from the use of the golden ratio. Because the original x_1 and x_2 were chosen using the golden ratio, we do not have to recalculate all the function values for the next iteration. For example, for the case illustrated in Fig. 7.6, the old x_1 becomes the new x_2. This means that we already have the value for the new $f(x_2)$, since it is the same as the function value at the old x_1.

To complete the algorithm, we need only determine the new x_1. This is done with Eq. (7.6) with d computed with Eq. (7.8) based on the new values of x_l and x_u. A similar approach would be used for the alternate case where the optimum fell in the left subinterval. For this case, the new x_2 would be computed with Eq. (7.7).

As the iterations are repeated, the interval containing the extremum is reduced rapidly. In fact, each round the interval is reduced by a factor of ϕ - 1(about 61.8%). That means that after 10 rounds, the interval is shrunk to about 0.618^{10} or 0.008 or 0.8% of its initial length. After 20 rounds, it is about 0.0066%. This is not quite as good as the reduction achieved with bisection (50%), but this is a harder problem.

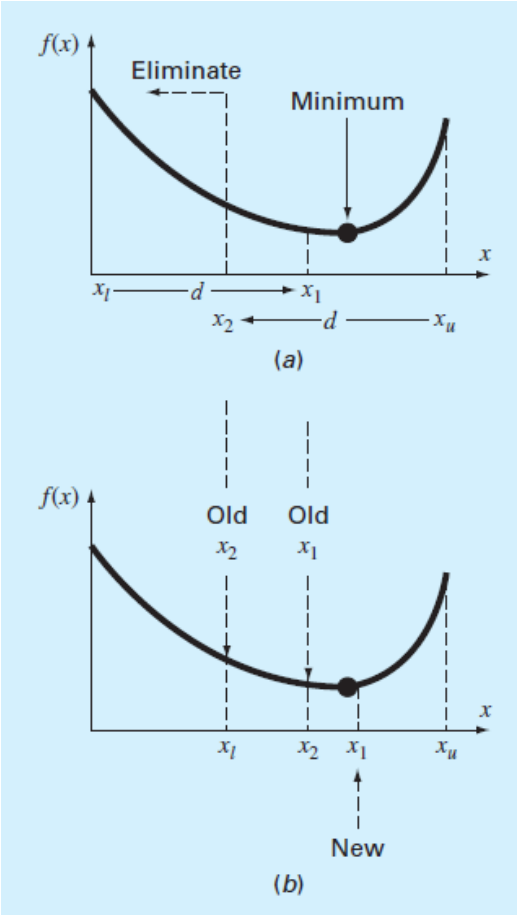


Figure 6.6: (a) The initial step of the golden-section search algorithm involves choosing two interior points according to the golden ratio. (b) The second step involves defining a new interval that encompasses the optimum.

Example 6.2. Golden-Section Search

Problem Statement: Use the golden-section search to find the minimum of

$$f(x) = \frac{x^2}{10} - 2sinx$$

within the interval from $x_l = 0$ to $x_u = 4$

Solution: First, the golden ratio is used to create the two interior points:

$$d = 0.61803(4-0) = 2.4721$$

$$x_1 = 0 + 2.4721 = 2.4721$$

$$x_2 = 4 - 2.4721 = 1.5279$$

The function can be evaluated at the interior points:

$$f(x_2) = \frac{1.5279^2}{10} - 2sin(1.5279) = -1.7647)$$

$$f(x_1) = \frac{2.4721^2}{10} - 2sin(2.4721) = -0.6300)$$

Because $f(x_2) < f(x_1)$, our best estimate of the minimum at this point is that it is located at x = 1.5279 with a value of $f(x) = -1.7647$. In addition, we also know that the minimum is in the interval defined by x_l, x_2, and x_1. Thus, for the next iteration, the lower bound remains $x_l = 0$, and x_1 becomes the upper bound, that is, $x_u = 2.4721$. In addition, the former x_2 value becomes the new x_1, that is, $x_1 = 1.5279$. In addition, we do not have to recalculate $f(x_1)$, it was determined on the previous iteration as $f(1.5279) = -1.7647$.

All that remains is to use Eqs. (7.8) and (7.7) to compute the new value of d and x_2:

$$d = 0.61803(2.4721-0) = 1.5279$$

$$x_2 = 2.4721 - 1.5279 = 0.9443$$

The function evaluation at x_2 is $f(0.9943) = -1.5310$. Since this value is less than the function value at x1, the minimum is $f(1.5279) = -1.7647$, and it is in the interval prescribed by x_2, x_1, and x_u. The process can be repeated, with the results tabulated here:

i	x_l	$f(x_l)$	x_2	$f(x_2)$	x_l	$f(x_l)$	x_u	$f(x_u)$	d
1	0	0	1.5279	-1.7647	2.4721	-0.6300	4.0000	3.1136	2.4721
2	0	0	0.9443	-1.5310	1.5279	-1.7647	2.4721	-0.6300	1.5279
3	0.9443	-1.5310	1.5279	-1.7647	1.8885	-1.5432	2.4721	-0.6300	0.9443
4	0.9443	-1.5310	1.3050	-1.7595	1.5279	-1.7647	1.8885	-1.5432	0.5836
5	1.3050	-1.7595	1.5279	-1.7647	1.6656	-1.7136	1.8885	-1.5432	0.3607
6	1.3050	-1.7595	1.4427	-1.7755	1.5279	-1.7647	1.6656	-1.7136	0.2229
7	1.3050	-1.7595	1.3901	-1.7742	1.4427	-1.7755	1.5279	-1.7647	0.1378
8	1.3901	-1.7742	1.4427	-1.7755	1.4752	-1.7732	1.5279	-1.7647	0.0851

Note that the current minimum is highlighted for every iteration. After the eighth iteration, the minimum occurs at x = 1.4427 with a function value of -1.7755. Thus, the result is converging on the true value of -1.7757 at x = 1.4276. ■

Recall that for bisection (Sec. 5.4), an exact upper bound for the error can be calculated at each iteration. Using similar reasoning, an upper bound for golden-section search can be derived as follows: Once an iteration is complete, the optimum will either fall in one of two intervals. If the optimum function value is at x_2, it will be in the lower interval (x_l, x_2, x_1). If the optimum function value is at x_1, it will be in the upper interval (x_2, x_1, x_u). Because the interior points are symmetrical, either case can be used to define the error.

Looking at the upper interval (x_2, x_1, x_u), if the true value were at the far left, the maximum distance from the estimate would be

$$\begin{aligned}\Delta x_a &= x_1 - x_2\\ &= x_l + (\phi-1)(x_u - x_l) - x_u + (\phi-1)(x_u - x_l)\\ &= (x_l - x_u) + 2(\phi-1)(x_u - x_l)\\ &= (2\phi - 3)(x_u - x_l)\end{aligned}$$

or 0.2361 $(x_u - x_l)$. If the true value were at the far right, the maximum distance from the estimate would be

$$\begin{aligned}\Delta x_b &= x_u - x_1\\ &= x_u - x_l - (\phi-1)(x_u - x_l)\\ &= (x_u - x_l) - (\phi-1)(x_u - x_l)\\ &= (2-\phi)(x_u - x_l)\end{aligned}$$

or 0.3820 $(x_u - x_l)$. Therefore, this case would represent the maximum error. This result can then be normalized to the optimal value for that iteration x_opt to yield

$$\varepsilon_a = (2-\phi)\left|\frac{x_u - x_l}{x_{opt}}\right| \times 100\% \tag{7.9}$$

This estimate provides a basis for terminating the iterations.

An M-file function for the golden-section search for minimization is presented in Fig. 7.7. The function returns the location of the minimum, the value of the function, the approximate error, and the number of iterations.

The M-file can be used to solve the problem from Example 7.1.

```
>> g=9.81;v0=55;m=80;c=15;z0=100;
>> z=@(t) -(z0+m/c*(v0+m*g/c)*(1-exp(-c/m*t))-m*g/c*t);
>> [xmin,fmin,ea,iter]=goldmin(z,0,8)

xmin =
   3.8317
fmin =
  -192.8609
ea =
   6.9356e-005
```

Notice how because this is a maximization, we have entered the negative of Eq. (7.1). Consequently, `fmin` corresponds to a maximum height of 192.8609.

You may be wondering why we have stressed the reduced function evaluations of the golden-section search. Of course, for solving a single optimization, the speed savings would be negligible. However, there are two important contexts where minimizing the number of function evaluations can be important. These are

1. Many evaluations. There are cases where the golden-section search algorithm may be a part of a much larger calculation. In such cases, it may be called many times. Therefore, keeping function evaluations to a minimum could pay great dividends for such cases.

```
function [x,fx,ea,iter]=goldmin(f,xl,xu,es,maxit,varargin)
% goldmin: minimization golden section search
% [x,fx,ea,iter]=goldmin(f,xl,xu,es,maxit,p1,p2,...):
% uses golden section search to find the minimum of f
% input:
% f = name of function
% xl, xu = lower and upper guesses
% es = desired relative error (default = 0.0001%)
% maxit = maximum allowable iterations (default = 50)
% p1,p2,... = additional parameters used by f
% output:
% x = location of minimum
% fx = minimum function value
% ea = approximate relative error (%)
% iter = number of iterations
if nargin<3,error('at least 3 input arguments required'),end
if nargin<4|isempty(es), es=0.0001;end
if nargin<5|isempty(maxit), maxit=50;end
phi=(1+sqrt(5))/2;
iter=0;
while(1)
  d = (phi-1)*(xu - xl);
  x1 = xl + d;
  x2 = xu - d;
  if f(x1,varargin{:}) < f(x2,varargin{:})
    xopt = x1;
    xl = x2;
  else
    xopt = x2;
    xu = x1;
  end
  iter=iter+1;
  if xopt~=0, ea = (2 - phi) * abs((xu - xl) / xopt) * 100;end
  if ea <= es | iter >= maxit,break,end
end
x=xopt;fx=f(xopt,varargin{:});
```

Figure 6.7: An M-file to determine the minimum of a function with the golden-section search.

2. Time-consuming evaluation. For pedagogical reasons, we use simple functions in most of our examples. You should understand that a function can be very complex and time-consuming to evaluate. For example, optimization can be used to estimate the parameters of a model consisting of a system of differential equations. For such cases, the "function" involves time-consuming model integration. Any method that minimizes such evaluations would be advantageous.

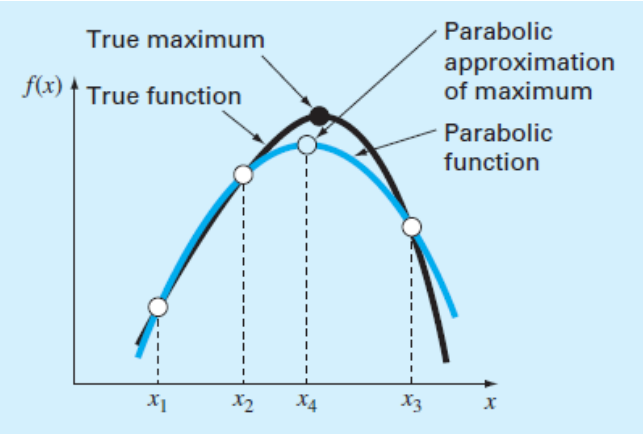


Figure 6.8: Graphical depiction of parabolic interpolation.

6.2.2. Parabolic Interpolation

Parabolic interpolation takes advantage of the fact that a second-order polynomial often provides a good approximation to the shape of f (x) near an optimum (Fig. 7.8).

Just as there is only one straight line connecting two points, there is only one parabola connecting three points. Thus, if we have three points that jointly bracket an optimum, we can fit a parabola to the points. Then we can differentiate it, set the result equal to zero, and solve for an estimate of the optimal x. It can be shown through some algebraic manipulations that the result is

$$x_4 = x_2 - \frac{1}{2}\frac{(x_2-x_1)^2[f(x_2)-f(x_3)]-(x_2-x_3)^2[f(x_2)-f(x_1)]}{(x_2-x_1)[f(x_2)-f(x_3)]-(x_2-x_3)[f(x_2)-f(x_1)]} \tag{7.10}$$

where $x1$, $x2$, and $x3$ are the initial guesses, and $x4$ is the value of x that corresponds to the optimum value of the parabolic fit to the guesses.

Example 6.3. Determining the Optimum Analytically by Root Location

Problem Statement: Use parabolic interpolation to approximate the minimum of

$$f(x) = \frac{x^2}{10} - 2sinx$$

with initial guesses of $x_1 = 0$, $x_2 = 1$, and $x_3 = 4$.

Solution: The function values at the three guesses can be evaluated:

$$\begin{array}{ll} x_1 = 0 & f(x_1) = 0 \\ x_2 = 1 & f(x_2) = -1.5829 \\ x_3 = 4 & f(x_3) = 3.1136 \end{array}$$

and substituted into Eq. (7.10) to give

$$x_4 = 1 - \frac{1}{2}\frac{(1-0)^2[-1.5829-3.1136]-(1-4)^2[-1.5829-0]}{(1-0)[-1.5829-3.1136]-(1-4)[-1.5829-0]} = 1.5055$$

which has a function value of f(1.5055) =-1.7691.

Next, a strategy similar to the golden-section search can be employed to determine which point should be discarded. Because the function value for the new point is lower than for the intermediate point (x_2) and the new x value is to the right of the intermediate point, the lower guess (x_1) is discarded. Therefore, for the next iteration:

$$\begin{array}{ll} x_1 = 1 & f(x_1) = -1.5829 \\ x_2 = 1.5055 & f(x_2) = -1.76919 \\ x_3 = 4 & f(x_3) = 3.1136 \end{array}$$

which can be substituted into Eq. (7.10) to give

$$x_4 = 1.5055 - \frac{1}{2}\frac{(1.5055-1)^2[-1.7691-3.1136]-(1.5055-4)^2[-1.7691-(-1.5829)]}{(1.5055-1)[-1.7691-3.1136]-(1.5055-4)[-1.7691-(-1.5829)]} = 1.4903$$

which has a function value of f(1.4903) =-1.7714. The process can be repeated, with the results tabulated here:

i	x_l	$f(x_l)$	x_2	$f(x_2)$	x_3	$f(x_3)$	x_4	$f(x_4)$
1	0.0000	0.0000	1.0000	-1.5828	4.0000	3.1136	1.5055	-1.7691
2	1.0000	-1.5829	1.5055	-1.7691	4.0000	3.1136	1.4903	-1.7714
3	1.0000	-1.5829	1.4903	-1.7714	1.5055	-1.7691	1.4256	-1.7757
4	1.0000	-1.5829	1.4256	-1.7757	1.4903	-1.7714	1.4266	-1.7757
5	1.4265	-1.7757	1.4266	-1.7757	1.4903	-1.7714	1.4275	-1.7757

Thus, within five iterations, the result is converging rapidly on the true value of -1.7757 at x = 1.4276. ■

6.2.3. MATLAB Function `fminbnd`

Recall that in Sec. 6.4 we described Brent's method for root location, which combined several root-finding methods into a single algorithm that balanced reliability with efficiency. Because of these qualities, it forms the basis for the built-in MATLAB function `fzero`.

Brent also developed a similar approach for one-dimensional minimization which forms the basis for the MATLAB `fminbnd` function. It combines the slow, dependable golden-section search with the faster, but possibly unreliable, parabolic interpolation. It first attempts parabolic interpolation and keeps applying it as long as acceptable results are obtained. If not, it uses the golden-section search to get matters in hand.

A simple expression of its syntax is

```
[xmin, fval] = fminbnd(function,x1,x2)
```

where x and `fval` are the location and value of the minimum, `function` is the name of the function being evaluated, and `x1` and `x2` are the bounds of the interval being searched.

Here is a simple MATLAB session that uses `fminbnd` to solve the problem from Example 7.1.

```
>> g=9.81;v0=55;m=80;c=15;z0=100;
>> z=@(t) -(z0+m/c*(v0+m*g/c)*(1-exp(-c/m*t))-m*g/c*t);
>> [x,f]=fminbnd(z,0,8)
x =
  3.8317
f =
  -192.8609
```

As with `fzero`, optional parameters can be specified using `optimset`. For example, we can display calculation details:

```
>> options = optimset('display','iter');
>> fminbnd(z,0,8,options)
Func-count    x          f(x)  Procedure
  1     3.05573   -189.759  initial
  2     4.94427   -187.19   golden
  3     1.88854   -171.871  golden
  4     3.87544   -192.851  parabolic
  5     3.85836   -192.857  parabolic
  6     3.83332   -192.861  parabolic
  7     3.83162   -192.861  parabolic
  8     3.83166   -192.861  parabolic
  9     3.83169   -192.861  parabolic
Optimization terminated:
the current x satisfies the termination criteria using
OPTIONS.TolX of 1.000000e-004
ans =
  3.8317
```

Thus, after three iterations, the method switches from golden to parabolic, and after eight iterations, the minimum is determined to a tolerance of 0.0001.

6.3. MULTIDIMENSIONAL OPTIMIZATION

Aside from one-dimensional functions, optimization also deals with multidimensional functions. Recall from Fig. 7.3a that our visual image of a one-dimensional search was like a roller coaster. For two-dimensional cases, the image becomes that of mountains and valleys (Fig. 7.3b). As in the following example, MATLAB's graphic capabilities provide a handy means to visualize such functions.

Example 6.4. Visualizing a Two-Dimensional Function

Problem Statement: Use MATLAB's graphical capabilities to display the following function and visually estimate its minimum in the range $-2 \le x_1 \le 0$ and $0 \le x_2 \le 3$:

$$f(x_1,x_2) = 2 + x_1 - x_2 + 2x_1^2 + 2x_1x_2 + x_2^2$$

with initial guesses of $x_1 = 0$, $x_2 = 1$, and $x_3 = 4$.

Solution: The following script generates contour and mesh plots of the function:

```
x=linspace(-2,0,40);y=linspace(0,3,40);
[X,Y] = meshgrid(x,y);
Z=2+X-Y+2*X.^2+2*X.*Y+Y.^2;
subplot(1,2,1);
cs=contour(X,Y,Z);clabel(cs);
xlabel('x_1');ylabel('x_2');
title('(a) Contour plot');grid;
subplot(1,2,2);
cs=surfc(X,Y,Z);
zmin=floor(min(Z));
zmax=ceil(max(Z));
xlabel('x_1');ylabel('x_2');zlabel('f(x_1,x_2)');
title('(b) Mesh plot');
```

As displayed in Fig. 7.9, both plots indicate that function has a minimum value of about $f(x1,x2) = 0$ to 1 located at about x_1 =-1 and x_2 = 1.5.

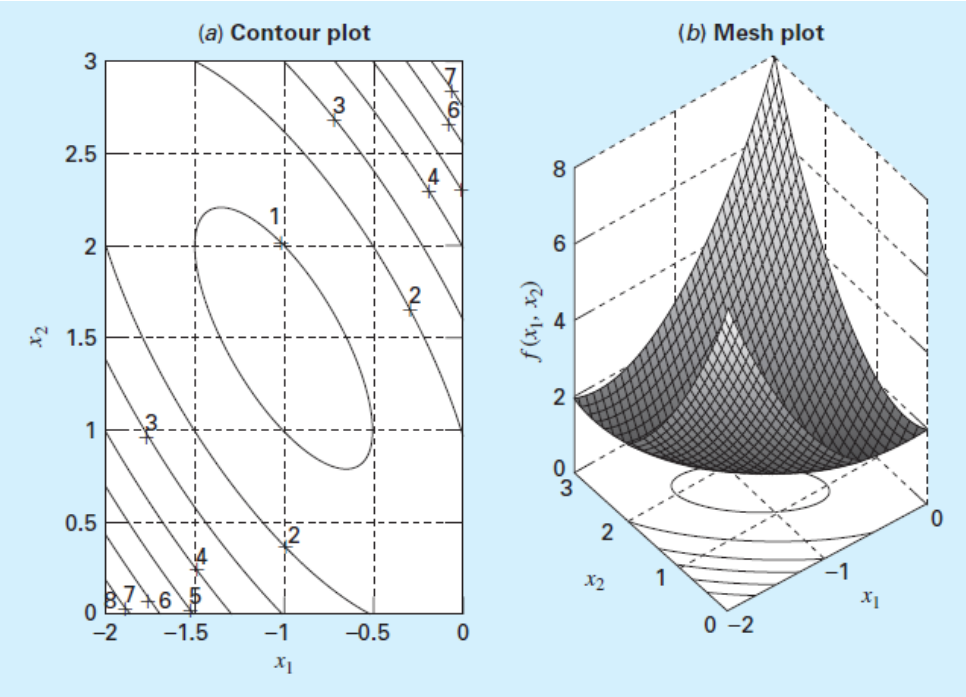


Figure 6.9: (a) Contour and (b) mesh plots of a two-dimensional function.

■

Techniques for multidimensional unconstrained optimization can be classified in a number of ways. For purposes of the present discussion, we will divide them depending on whether they require derivative evaluation. Those that require derivatives are called `gradient`, or `descent` (or ascent), methods. The approaches that do not require derivative evaluation are called nongradient, or direct, methods. As described next, the built-in MATLAB function `fminsearch` is a direct method.

6.3.1. MATLAB Function: `fminsearch`

Standard MATLAB has a function `fminsearch` that can be used to determine the minimum of a multidimensional function. It is based on the Nelder-Mead method, which is a direct-search method that uses only function values (does not require derivatives) and handles non-smooth objective functions. A simple expression of its syntax is

```
fminsearch
```

where `xmin` and `fval` are the location and value of the minimum, `function` is the name of the function being evaluated, and `x0` is the initial guess. Note that `x0` can be a scalar, vector, or a matrix.

Here is a simple MATLAB session that uses `fminsearch` to determine minimum for the function we just graphed in Example 7.4:

```
>> f=@(x) 2+x(1)-x(2)+2*x(1)^2+2*x(1)*x(2)+x(2)^2;
>> [x,fval]=fminsearch(f,[-0.5,0.5])
x =
  -1.0000 1.5000
fval =
  0.7500
```

6.4. CASE STUDY - EQUILIBRIUM AND MINIMUM POTENTIAL ENERGY

Background. As in Fig. 7.10a, an unloaded spring can be attached to a wall mount. When a horizontal force is applied, the spring stretches. The displacement is related to the force by *Hookeslaw*, $F = kx$. The potential *energy* of the deformed state consists of the difference between the strain energy of the spring and the work done by the force:

$$PE(x) = 0.5kx^2 - Fx \tag{7.11}$$

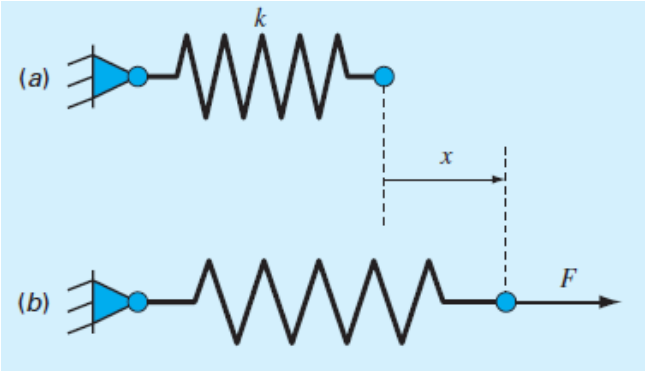


Figure 6.10: (a) An unloaded spring attached to a wall mount. (b) Application of a horizontal force stretches the spring where the relationship between force and displacement is described by Hooke's law.

Equation (7.11) defines a parabola. Since the potential energy will be at a minimum at equilibrium, the solution for displacement can be viewed as a one-dimensional optimization problem. Because this equation is so easy to differentiate, we can solve for the displacement as $x = F/k$. For example, if $k = 2N/cm$ and $F = 5N, x = 5N/(2N/cm) = 2.5cm$.

A more interesting two-dimensional case is shown in Fig. 7.11. In this system, there are two degrees of freedom in that the system can move both horizontally and vertically. In the same way that we approached the one-dimensional system, the equilibrium deformations are the values of x_1 and x_2 that minimize the potential energy:

$$PE(x_1,x_2) = 0.5k_a(\sqrt{x_1^2+(L_a-x_2)^2}-L_a)^2+0.5k_b(\sqrt{x_1^2+(L_b+x_2)^2}-L_b)^2-F_1x_1-F_2x_2 \qquad 7.12$$

If the parameters are k_a = 9 N/cm, k_b = 2 N/cm, L_a = 10 cm, L_b= 10 cm, F_1= 2 N, and F_2 = 4 N, use MATLAB to solve for the displacements and the potential energy.

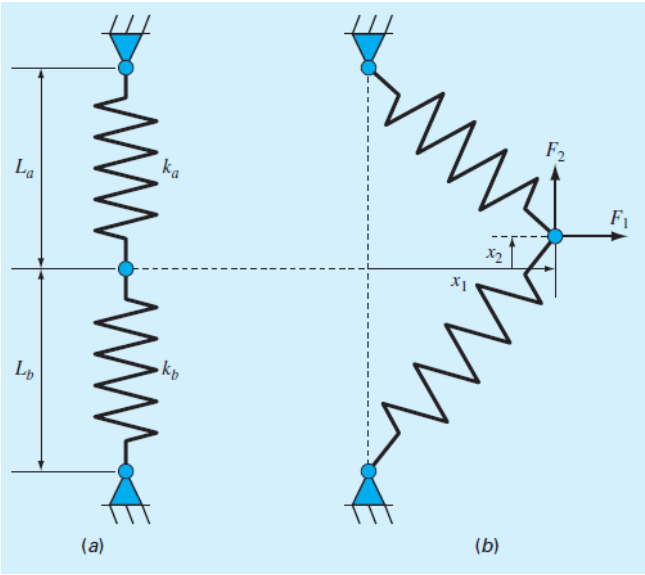


Figure 6.11: A two-spring system: (a) unloaded and (b) loaded.

Solution. An M-file can be developed to hold the potential energy function:

```
function p=PE(x,ka,kb,La,Lb,F1,F2)
PEa=0.5*ka*(sqrt(x(1)^2+(La-x(2))^2)-La)^2;
PEb=0.5*kb*(sqrt(x(1)^2+(Lb+x(2))^2)-Lb)^2;
W=F1*x(1)+F2*x(2);
p=PEa+PEb-W;
```

The solution can be obtained with the `fminsearch` function:

```
>> ka=9;kb=2;La=10;Lb=10;F1=2;F2=4;
>> [x,f]=fminsearch(@PE,[-0.5,0.5],[],ka,kb,La,Lb,F1,F2)
x =
   4.9523   1.2769
f =
   -9.6422
```

Thus, at equilibrium, the potential energy is -9.6422 NÂů cm. The connecting point is located 4.9523 cm to the right and 1.2759 cm above its original position.

PROBLEMS

7.1 Perform three iterations of the Newton-Raphson method to determine the root of Eq. (E7.1.1). Use the parameter values from Example 7.1 along with an initial guess of t = 3 s.

7.2 Given the formula

$f(x) = -x^2 + 8x - 12$

(a) Determine the maximum and the corresponding value of x for this function analytically (i.e., using differentiation).
(b) Verify that Eq. (7.10) yields the same results based on initial guesses of x_1 = 0, x_2 = 2, and x_3 = 6.

7.3 Consider the following function:

$f(x) = 3 + 6x + 5x^2 + 3x^3 + 4x^4$

Locate the minimum by finding the root of the derivative of this function. Use bisection with initial guesses of $x_l = -2$ and $x_u = 1$.

7.4 Given

$f(x) = -1.5x^6 - 2x^4 + 12x$

(a) Plot the function.
(b) Use analytical methods to prove that the function is concave for all values of x.
(c) Differentiate the function and then use a root-location method to solve for the maximum $f(x)$ and the corresponding value of x.

7.5 Solve for the value of x that maximizes $f(x)$ in Prob. 7.4 using the golden-section search. Employ initial guesses of x_l = 0 and x_u = 2, and perform three iterations.

7.6 Repeat Prob. 7.5, except use parabolic interpolation. Employ initial guesses of x_1 = 0, x_2 = 1, and x_3 = 2, and perform three iterations.

7.7 Employ the following methods to find the maximum of

$f(x) = 4x - 1.8x^2 + 1.2x^3 - 0.3x^4$

(a)Golden-section search (x_l = - 2, x_u = 4, ε_s = 1%).
(b)Parabolic interpolation (x_1 = 1.75, x_2 = 2, x_3 = 2.5, iterations = 5).

7.8 Consider the following function:

$f(x) = x^4 + 2x^3 + 8x^2 + 5x$

Use analytical and graphical methods to show the function has a minimum for some value of x in the range

$$-2 \le x \le 1.$$

7.9 Employ the following methods to find the minimum of the function from Prob. 7.8:
(a) Golden-section search (x_l = -2, x_u= 1, ε_s = 1%).
(b) Parabolic interpolation (x_1= -2, x_2= -1, x_3= 1, iterations = 5).

7.10 Consider the following function:

$$f(x) = 2x + \frac{3}{x}$$

Perform 10 iterations of parabolic interpolation to locate the minimum. Comment on the convergence of your results (x_1 = 0.1, x_2 = 0.5, x_3 = 5)

7.11 Develop a single script to (a) generate contour and mesh subplots of the following temperature field in a similar fashion to Example 7.4:

$T(x,y) = 2x^2 + 3y^2 - 4xy - y - 3x$

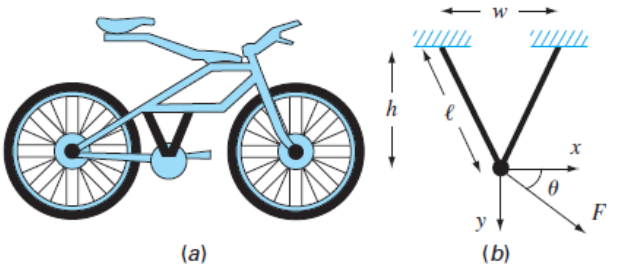
and (b) determine the minimum with `fminsearch`.

7.12 The head of a groundwater aquifer is described in Cartesian coordinates by

$$h(x,y) = \frac{1}{1 + x^2 + y^2 + x + xy}$$

Develop a single script to (a) generate contour and mesh subplots of the function in a similar fashion to Example 7.4, and (b) determine the maximum with fminsearch.

7.13 Recent interest in competitive and recreational cycling has meant that engineers have directed their skills toward the design and testing of mountain bikes (Fig. P7.13a). Suppose that you are given the task of predicting the horizontal and vertical displacement of a bike bracketing system in response to a force. Assume the forces you must analyze can be simplified as depicted in Fig. P7.13b. You are interested



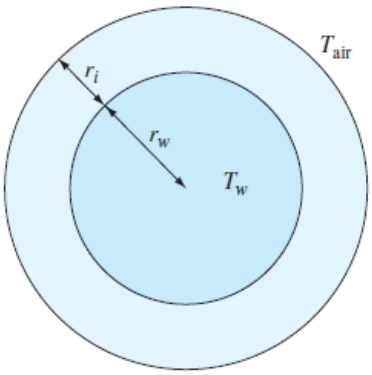
(a) A mountain bike along with (b) a free-body diagram for a part of the frame.

in testing the response of the truss to a force exerted in any number of directions designated by the angle θ. The parameters for the problem are E = Young's modulus = $2 \times 10^1 1$ Pa, A = cross-sectional area = 0.0001 m^2, w = width = 0.44 m, l = length = 0.56 m, and h = height = 0.5 m. The displacements x and y can be solved by determining the values that yield a minimum potential energy. Determine the displacements for a force of 10,000 N and a range of θ's from 0Âř (horizontal) to 90Âř (vertical).

7.14 As electric current moves through a wire (Fig. P7.14), heat generated by resistance is conducted through a layer of insulation and then convected to the surrounding air. The steady-state temperature of the wire can be computed as

$$T = T_{\text{air}} + \frac{q}{2\pi}\left[\frac{1}{k}\ln\left(\frac{r_w + r_i}{r_w}\right) + \frac{1}{h}\frac{1}{r_w + r_i}\right]$$

Determine the thickness of insulation r_i(m) that minimizes the wire's temperature given the following parameters: q = heat generation rate = 75 W/m, r_w = wire radius = 6 mm, k = thermal conductivity of insulation = 0.17 W/(mK), h = convective heat transfer coefficient = 12 W/ $\left(\text{m}^2\ \text{K}\right)$, and T_{air} = air temperature = 293 K.



Cross-section of an insulated wire.

7.15 Develop an M-file that is expressly designed to locate a maximum with the golden-section search. In other words, set if up so that it directly finds the maximum rather than finding the minimum of $-f(x)$. The function should have the following features:

- Iterate until the relative error falls below a stopping criterion or exceeds a maximum number of iterations.
- Return both the optimal x and $f(x)$.

Test your program with the same problem as Example 7.1.

7.16 Develop an M-file to locate a minimum with the golden-section search. Rather than using the maximum iterations and Eq. (7.9) as the stopping criteria, determine the number of iterations needed to attain a desired tolerance. Test your function by solving Example 7.2 using $E_{a,d} = 0.0001$.

7.17 Develop an M-file to implement parabolic interpolation to locate a minimum. The function should have the following features:

- Base it on two initial guesses, and have the program generate the third initial value at the midpoint of the interval.
- Check whether the guesses bracket a maximum. If not, the function should not implement the algorithm, but should return an error message.
- Iterate until the relative error falls below a stopping criterion or exceeds a maximum number of iterations.
- Return both the optimal x and $f(x)$.

Test your program with the same problem as Example 7.3.

7.18 Pressure measurements are taken at certain points behind an airfoil over time. These data best fit the curve $y = 6\cos x - 1.5\sin x$ from $x = 0$ to 6 s. Use four iterations of the golden-search method to find the minimum pressure. Set $x_l = 2$ and $x_u = 4$

7.19 The trajectory of a ball can be computed with

$$y = (\tan\theta_0)x - \frac{g}{2v_0^2\cos^2\theta_0}x^2 + y_0$$

where $y =$ the height (m), $\theta_0 =$ the initial angle (radians), $v_0 =$ the initial velocity (m/s), $g =$ the gravitational constant $= 9.81$ m/s^2, and $y_0 =$ the initial height (m). Use the golden-section search to determine the maximum height given $y_0 = 1$ m, $v_0 = 25$ m/s, and $\theta_0 = 50^\circ$. Iterate until the approximate error falls below $\varepsilon_s = 1\%$ using initial guesses of $x_l = 0$ and $x_u = 60$ m.

7.20 The deflection of a uniform beam subject to a linearly increasing distributed load can be computed as

$$y = \frac{w_0}{120EIL}\left(-x^5 + 2L^2x^3 - L^4x\right)$$

Given that $L = 600$ cm, $E = 50{,}000$kN/cm^2, $I = 30{,}000$ cm^4, and $w_0 = 2.5$kN/cm, determine the point of maximum deflection (a) graphically, (b) using the golden-section search until the approximate error falls below $\varepsilon_s = 1\%$ with initial guesses of $x_1 = 0$ and $x_u = L$.

7.21 A object with a mass of 90 kg is projected upward from the surface of the earth at a velocity of 60 m/s. If the object is subject to linear drag ($c = 15$ kg/s), use the golden-section search to determine the maximum height the object attains.

7.22 The normal distribution is a bell-shaped curve defined by

$$y = e^{-x^2}$$

Use the golden-section search to determine the location of the inflection point of this curve for positive x. **7.23** Use the fminsearch function to determine the minimum of

$$f(x,y) = 2y^2 - 2.25xy - 1.75y + 1.5x^2$$

7.24 Use the fminsearch function to determine the maximum of

$$f(x,y) = 4x + 2y + x^2 - 2x^4 + 2xy - 3y^2$$

7.25 Given the following function:

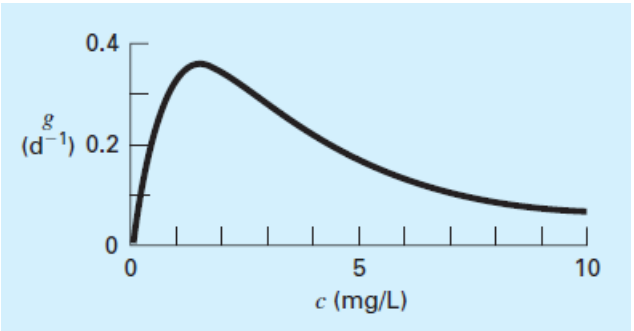
$$f(x,y) = -8x + x^2 + 12y + 4y^2 - 2xy$$

Determine the minimum (a) graphically, (b) numerically with the fminsearch function, and (c) substitute the result of (b) back into the function to determine the minimum $f(x,y)$.

7.26 The specific growth rate of a yeast that produces an antibiotic is a function of the food concentration c :

$$g = \frac{2c}{4 + 0.8c + c^2 + 0.2c^3}$$

As depicted in Fig. P7.26, growth goes to zero at very low concentrations due to food limitation. It also goes to zero at high concentrations due to toxicity effects. Find the value of c at which growth is a maximum.



The specific growth rate of a yeast that produces an antibiotic versus the food concentration.

7.27 A compound A will be converted into B in a stirred tank reactor. The product B and unreacted A are purified in a separation unit. Unreacted A is recycled to the reactor. A process engineer has found that the initial cost of the system is a function of the conversion x_A. Find the conversion that will result in the lowest cost system. C is a proportionality constant.

$$\text{Cost} = C\left[\left(\frac{1}{(1-x_A)^2}\right)^{0.6} + 6\left(\frac{1}{x_A}\right)^{0.6}\right]$$

7.28 A finite-element model of a cantilever beam subject to loading and moments (Fig. P7.28) is given by optimizing

$$f(x,y) = 5x^2 - 5xy + 2.5y^2 - x - 1.5y$$

where x = end displacement and y = end moment. Find the values of x and y that minimize $f(x,y)$.

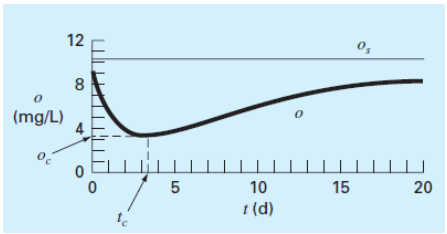
A cantilever beam.

7.29 The Streeter-Phelps model can be used to compute the dissolved oxygen concentration in a river below a point discharge of sewage (Fig. P7.29),

$$o = o_s - \frac{k_d L_o}{k_d + k_s - k_a}\left(e^{-k_a t} - e^{-(k_d+k_s)t}\right) - \frac{S_b}{k_a}\left(1 - e^{-k_a t}\right)$$

where o = dissolved oxygen concentration (mg/L), o_s = oxygen saturation concentration (mg/L), t = travel time (d), L_o = biochemical oxygen demand (BOD) concentration at the mixing point (mg/L), k_d = rate of decomposition of BOD $\left(\text{d}^{-1}\right)$, k_s = rate of settling of BOD $\left(\text{d}^{-1}\right)$, k_a = reaeration rate $\left(\text{d}^{-1}\right)$, and S_b = sediment oxygen demand (mg/L/d). As indicated in Fig. P7.29, Eq. (P7.29) produces an oxygen "sag" that reaches a critical minimum level o_c, some travel time t_c below the point discharge. This point is called "critical" because it represents the location where biota that depend on oxygen (like fish) would be the most stressed. Determine the critical travel time and concentration, given the following values:

$o_s = 10\text{mg/L}$ $k_d = 0.1\ \text{d}^{-1}$ $k_a = 0.6\ \text{d}^{-1}$
$k_s = 0.05\ \text{d}^{-1}$ $L_o = 50\text{mg/L}$ $S_b = 1\text{mg/L/d}$



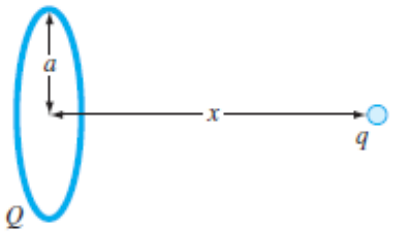
A dissolved oxygen âĂIJsagâĂİ below a point discharge of sewage into a river.

7.30 The two-dimensional distribution of pollutant concentration in a channel can be described by $c(x,y) = 7.9 + 0.13x + 0.21y - 0.05x^2 \ -0.016y^2 - 0.007xy$ the function and the knowledge that the peak lies within the bounds $-10 \le x \le 10$ and $0 \le y \le 20$.

7.31 A total charge Q is uniformly distributed around a ring-shaped conductor with radius a. A charge q is located at a distance x from the center of the ring (Fig. P7.31). The force exerted on the charge by the ring is given by

$$F = \frac{1}{4\pi e_0}\frac{qQx}{(x^2+a^2)^{3/2}}$$

where $e_0 = 8.85 \times 10^{-12}\text{C}^2/\left(\text{Nm}^2\right)$, $q = Q = 2 \times 10^{-5}\text{C}$, and $a = 0.9$ m. Determine the distance x where the force is a maximum.



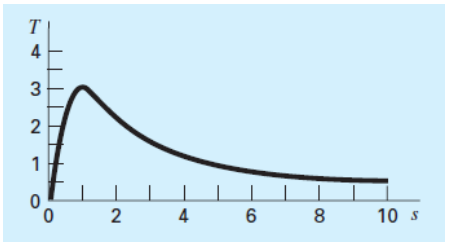
7.32 The torque transmitted to an induction motor is a function of the slip between the rotation of the stator field and the rotor speed s, where slip is defined as

$$s = \frac{n - n_R}{n}$$

where n = revolutions per second of rotating stator speed and n_R = rotor speed. Kirchhoff's laws can be used to show that the torque (expressed in dimensionless form) and slip are related by

$$T = \frac{15s(1-s)}{(1-s)(4s^2 - 3s + 4)}$$

Figure P7.32 shows this function. Use a numerical method to determine the slip at which the maximum torque occurs.



Torque transmitted to an inductor as a function of slip.

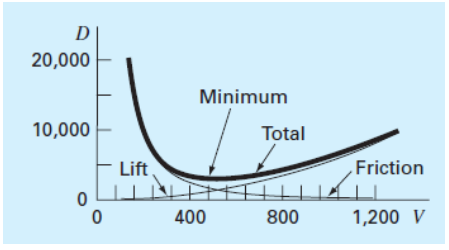
7.33 The total drag on an airfoil can be estimated by

$$D = 0.01\sigma V^2 + \frac{0.95}{\sigma}\left(\frac{W}{V}\right)^2$$

Friction Lift where D = drag, σ = ratio of air density between the flight altitude and sea level, W = weight, and V = velocity. As seen in Fig. P7.33, the two factors contributing to drag are affected differently as velocity increases. Whereas friction drag increases with velocity, the drag due to lift decreases. The combination of the two factors leads to a minimum drag.

(a) If $\sigma = 0.6$ and $W = 16{,}000$, determine the minimum drag and the velocity at which it occurs.

(b) In addition, develop a sensitivity analysis to determine how this optimum varies in response to a range of $W = 12{,}000$ to $20{,}000$ with $\sigma = 0.6$.

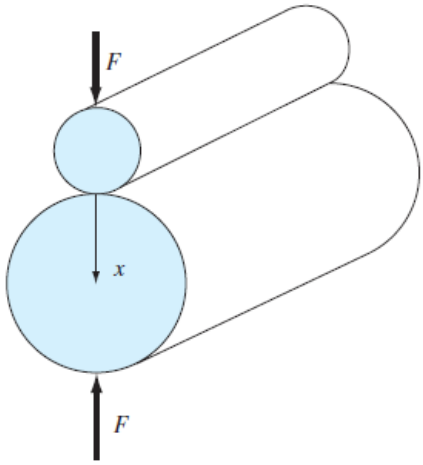


Plot of drag versus velocity for an airfoil.

7.34 Roller bearings are subject to fatigue failure caused by large contact loads F (Fig. P7.34). The problem of finding the location of the maximum stress along the x axis can be shown to be equivalent to maximizing the function:

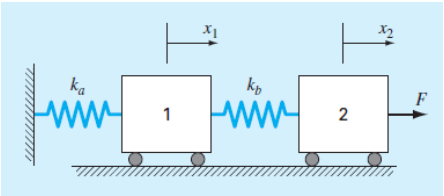
$$f(x) = \frac{0.4}{\sqrt{1+x^2}} - \sqrt{1+x^2}\left(1 - \frac{0.4}{1+x^2}\right) + x$$

Find the x that maximizes $f(x)$.



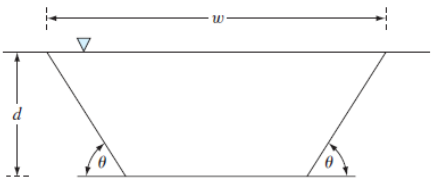
Plot of drag versus velocity for an airfoil.

7.35 In a similar fashion to the case study described in Sec. 7.4, develop the potential energy function for the system depicted in Fig. P7.35. Develop contour and surface plots in MATLAB. Minimize the potential energy function to determine the equilibrium displacements x_1 and x_2 given the forcing function $F = 100$ N and the parameters $k_a = 20$ and $k_b = 15$ N/m.

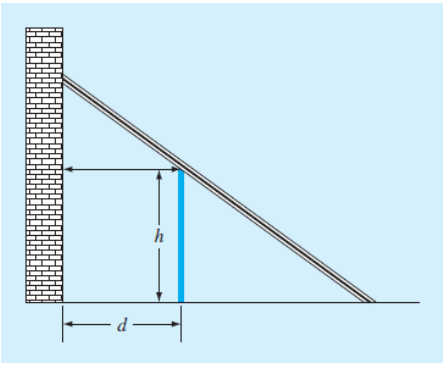


Two frictionless masses connected to a wall by a pair of linear elastic springs.

7.36 As an agricultural engineer, you must design a trapezoidal open channel to carry irrigation water (Fig. P7.36). Determine the optimal dimensions to minimize the wetted perimeter for a cross-sectional area of 50 m^2. Are the relative dimensions universal?



7.37 Use the function `fminsearch` to determine the length of the shortest ladder that reaches from the ground over the fence to the building's wall (Fig. P7.37). Test it for the case where $h = d = 4$ m.

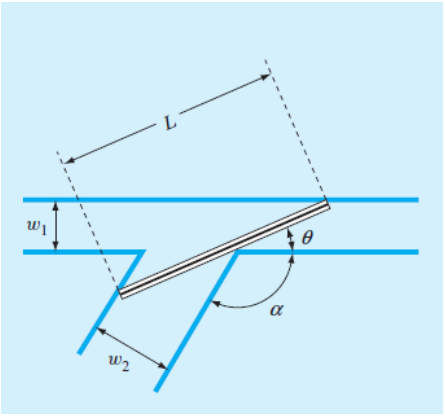


A ladder leaning against a fence and just touching a wall.

7.38 The length of the longest ladder that can negotiate the corner depicted in Fig. P7.38 can be determined by computing the value of θ that minimizes the following function:

$$L(\theta) = \frac{w_1}{\sin\theta} + \frac{w_2}{\sin(\pi - \alpha - \theta)}$$

For the case where $w_1 = w_2 = 2$ m, use a numerical method described in this chapter (including MATLAB's built-in capabilities) to develop a plot of L versus a range of α 's from 45 to 135°.



A ladder negotiating a corner formed by two hallways.

Part III

Linear Systems

6.5. OVERVIEW

What Are Linear Algebraic Equations?

In Part Two, we determined the value x that satisfied a single equation, $f(x) = 0$. Now, we deal with the case of determining the values $x_1, x_2, \ldots, x_n$ that simultaneously satisfy a set of equations:

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

Such systems are either linear or nonlinear. In Part Three, we deal with linear algebraic equations that are of the general form

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

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.

Linear Algebraic Equations in Engineering and Science

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

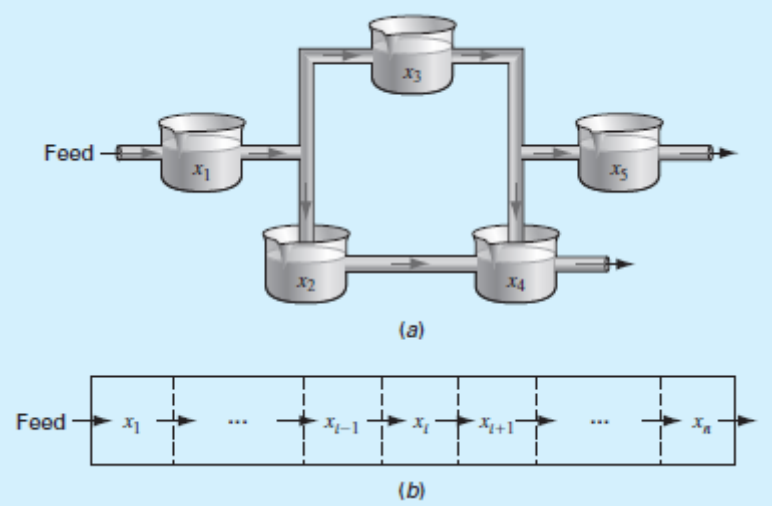


FIGURE PT3.1 - Two types of systems that can be modeled using linear algebraic equations: (a) lumped variable system that involves coupled finite components and (b) distributed variable system that involves a continuum.

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. PT3.1a). For this case, the quantity being modeled is the mass of the chemical in each reactor. The system properties are the reaction characteristics 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. PT3.1a, 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. (PT3.1). The x 's are usually measures

of the magnitudes of the responses of the individual components. Using Fig. PT3.1 a 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. PT3. la 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.

Multicomponent problems of these types arise from both lumped (macro-) or distributed (micro-) variable mathematical models. Lumped variable problems involve coupled

finite components. The three interconnected bungee jumpers described at the beginning of Chap. 8 are a lumped system. Other examples include trusses, reactors, and electric circuits.

Conversely, distributed variable problems attempt to describe the spatial detail on a continuous or semicontinuous basis. The distribution of chemicals along the length of an elongated, rectangular reactor (Fig. PT3.1b) 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 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. PT3.1 b 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.

6.6. PART ORGANIZATION

Due to its importance in formulating and solving linear algebraic equations, Chap. 8 provides a brief overview of matrix algebra. Aside from covering the rudiments of matrix representation and manipulation, the chapter also describes how matrices are handled in MATLAB.

Chapter 9 is devoted to the most fundamental technique for solving linear algebraic systems: Gauss elimination. Before launching into a detailed discussion of this technique, a preliminary section deals with simple methods for solving small systems. These approaches are presented to provide you with visual insight and because one of the methodsthe elimination of unknowns-represents the basis for Gauss elimination.

After this preliminary material, "naive" Gauss elimination is discussed. We start with this "stripped-down" version because it allows the fundamental technique to be elaborated on without complicating details. Then, in subsequent sections, we discuss potential problems of the naive approach and present a number of modifications to minimize and circumvent these problems. The focus of this discussion will be the process of switching rows, or partial pivoting. The chapter ends with a brief description of efficient methods for solving tridiagonal matrices.

Chapter 10 illustrates how Gauss elimination can be formulated as an LU factorization. Such solution techniques are valuable for cases where many right-hand-side vectors need to be evaluated. The chapter ends with a brief outline of how MATLAB solves linear systems.

Chapter 11 starts with a description of how LU factorization can be employed to efficiently calculate the matrix inverse, which has tremendous utility in analyzing stimulusresponse relationships of physical systems. The remainder of the chapter is devoted to the important concept of matrix condition. The condition number is introduced as a measure of the roundoff errors that can result when solving ill-conditioned matrices.

Chapter 12 deals with iterative solution techniques, which are similar in spirit to the approximate methods for roots of equations discussed in Chap. 6. That is, they involve guessing a solution and then iterating to obtain a refined estimate. The emphasis is on the GaussSeidel method, although a description is provided of an alternative approach, the Jacobi method. The chapter ends with a brief description of how nonlinear simultaneous equations can be solved.

Finally, Chap. 13 is devoted to eigenvalue problems. These have general mathematical relevance as well as many applications in engineering and science. We describe two simple methods as well as MATLAB's capabilities for determining eigenvalues and eigenvectors. In terms of applications, we focus on their use to study the vibrations and oscillations of mechanical systems and structures.

Chapter 7

Linear Algebraic Equations and Matrices

CHAPTER OBJECTIVES

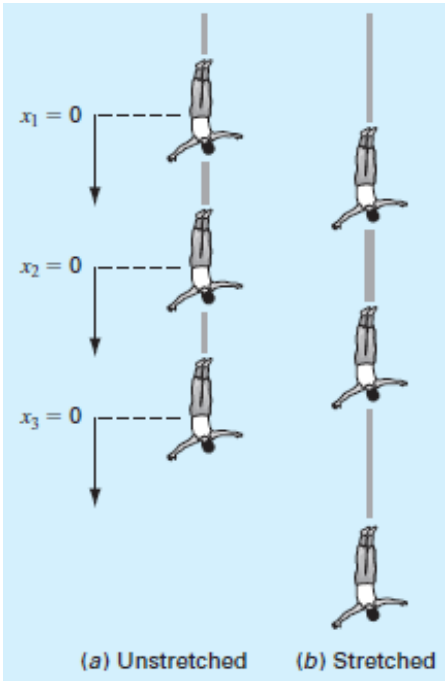
The primary objective of this chapter is to acquaint you with linear algebraic equations and their relationship to matrices and matrix algebra. Specific objectives and topics covered are

- Understanding 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 8.1 *a* 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. 8.1b.

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. 8.2.



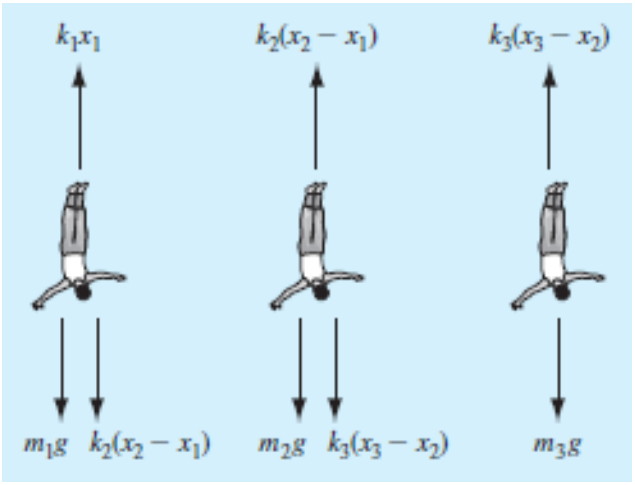


Figure 7.2: Free-body diagrams.

Figure 7.1: Three individuals connected by bungee cords.

Using Newton's second law, force balances can be written for each jumper:

$$\begin{aligned} m_1\frac{d^2x_1}{dt^2} &= m_1g + k_2(x_2 - x_1) - k_1x_1 \\ m_2\frac{d^2x_2}{dt^2} &= m_2g + k_3(x_3 - x_2) + k_2(x_1 - x_2) \\ m_3\frac{d^2x_3}{dt^2} &= m_3g + k_3(x_2 - x_3) \end{aligned} \tag{8.1}$$

where $m_i =$ the mass of jumper i(kg), $t =$ time (s), $k_j =$ 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 $(9.81\ \text{m/s}^2)$. Because we are interested in the steady-state solution, the second derivatives can be set to zero. Collecting terms gives

$$\begin{aligned} (k_1 + k_2)x_1 - k_2x_2 &= m_1g \\ -k_2x_1 + (k_2 + k_3)x_2 - k_3x_3 &= m_2g \\ -k_3x_2 + k_3x_3 &= m_3g \end{aligned} \tag{8.2}$$

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 8 through 12 will introduce you to how MATLAB is used to solve such systems of equations.

7.1. 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.

7.1.1. Matrix Notation

A matrix consists of a rectangular array of elements represented by a single symbol. As depicted in Fig. 8.3, $[A]$ 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. 8.3 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

$$[b] = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix}$$

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 $\lfloor b \rfloor$.

Matrices with column dimension $n = 1$, such as

$$[c] = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

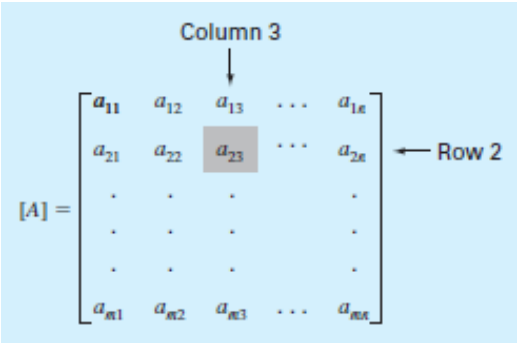


Figure 7.3: A matrix.

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 $\{c\}$.

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

$$[I] = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}$$

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

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ & a_{22} & a_{23} \\ & & a_{33} \end{bmatrix}$$

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.

7.1.2. 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 $i = 1,2,\ldots,m$ and $j = 1,2,\ldots,n$. 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 $i = 1,2,\ldots,m$ and $j = 1,2,\ldots,n$. 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:

$$[D] = g[A] = \begin{bmatrix} ga_{11} & ga_{12} & ga_{13} \\ ga_{21} & ga_{22} & ga_{23} \\ ga_{31} & ga_{32} & ga_{33} \end{bmatrix}$$

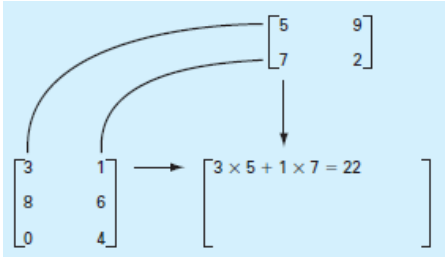


Figure 7.4: Visual depiction of how the rows and columns line up in matrix multiplication.

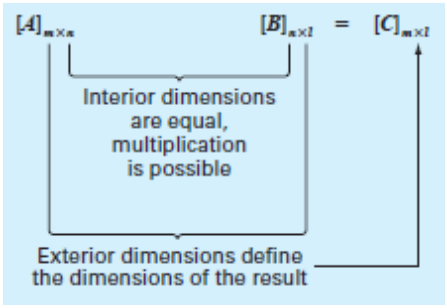


Figure 7.5: Matrix multiplication can be performed only if the inner dimensions are equal.

The product of two matrices is represented as $[C] = [A][B]$, where the elements of $[C]$ are defined as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{8.4}$$

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 8.4 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 8.5 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 matrix 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. 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 11 will deal with techniques for using numerical methods and the computer to calculate the inverse for such systems.

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

$$\{c\} = \begin{Bmatrix} c_1 \\ c_1 \\ c_1 \end{Bmatrix}$$

then

$$\{c\}^T = \begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix}$$

In addition, the transpose has numerous mathematical applications. A permutation matrix (also called a transposition matrix) is an identity matrix with rows and columns interchanged. For example, here is a permutation matrix that is constructed by switching the first and third rows and columns of a 3×3 identity matrix:

$$[P] = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Left multiplying a matrix $[A]$ by this matrix, as in $[P][A]$, will switch the corresponding rows of $[A]$. Right multiplying, as in $[A][P]$, will switch the corresponding columns. Here is an example of left multiplication:

$$[P][A] = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & -7 & 4 \\ 8 & 3 & -6 \\ 5 & 1 & 9 \end{bmatrix} = \begin{bmatrix} 5 & 1 & 9 \\ 8 & 3 & -6 \\ 2 & -7 & 4 \end{bmatrix}$$

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 3×6 dimensional matrix:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{11} & a_{11} & 1 & 0 & 0 \\ a_{21} & a_{21} & a_{21} & 0 & 1 & 0 \\ a_{31} & a_{31} & a_{31} & 0 & 0 & 1 \end{array}\right]$$

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 $[A]$:

```
>> A = C-B
A =
     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
```

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 ==> mtimes
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
```

We can set up a permutation matrix to switch the first and third rows and columns of a 3×3 matrix as

```
>> P=[0 0 1;0 1 0;1 0 0]
P =
    0 0 1
    0 1 0
    1 0 0
```

We can then either switch the rows:

```
>> PA=P*A
PA =
    -3 6 7
    7 4 2
    1 5 6
```

or the columns:

```
>> AP=A*P
AP =
    6 5 1
    2 4 7
    7 6 -3
```

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
```

■

7.1.3. 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,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \tag{8.5}$$

can be expressed as

$$[A]\{x\} = \{b\}$$

where $[A]$ is the matrix of coefficients:

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{8.6}$$

$\{b\}$ is the column vector of constants:

$$\{b\}^T = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

and $\{x\}$ is the column vector of unknowns:

$$\{x\}^T = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

Recall the definition of matrix multiplication [Eq. (8.4)] to convince yourself that Eqs. (8.5) and (8.6) are equivalent. Also, realize that Eq. (8.6) 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. (8.6) 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

$$[A]^{-1}[A]\{x\} = [A]^{-1}\{b\}$$

Because $[A]^{-1}[A]$ equals the identity matrix, the equation becomes

$$\{x\} = [A]^{-1}\{b\} \tag{8.7}$$

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 11.1.2, 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.

7.2. 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 8.1.3, 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 ($\mathbf{N/m}$)	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. (8.2) 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. 8.6, 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 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.

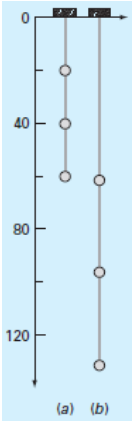


Figure 7.6: Positions of three individuals connected by bungee cords. (a) Unstretched and (b) stretched.

■

7.3. CASE STUDY - CURRENTS AND VOLTAGES IN CIRCUITS

Background. Recall that in Chap. 1 (Table 1.1), we summarized some models and associated conservation laws that figure prominently in engineering. As in Fig. 8.7, each model represents a system of interacting elements. Consequently, steady-state balances derived from the conservation laws yield systems of simultaneous equations. In many cases, such systems are linear and hence can be expressed in matrix form. The present case study focuses on one such application: circuit analysis.

A common problem in electrical engineering involves determining the currents and voltages at various locations in resistor circuits. These problems are solved using Kirchhoff's current and voltage rules. The current (or point) rule states that the algebraic sum of all currents entering a node must be zero (Fig. 8.8*a*), or

$$\sum i = 0 \tag{8.8}$$

where all current entering the node is considered positive in sign. The current rule is an application of the principle of conservation of charge (recall Table 1.1).

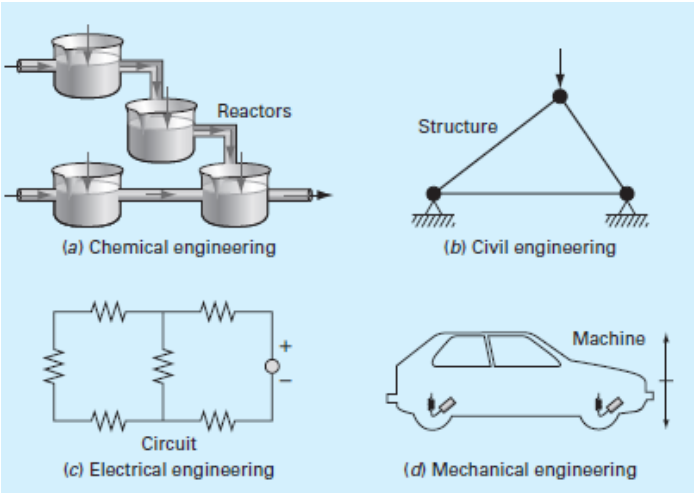


Figure 7.7: Engineering systems which, at steady state, can be modeled with linear algebraic equations.

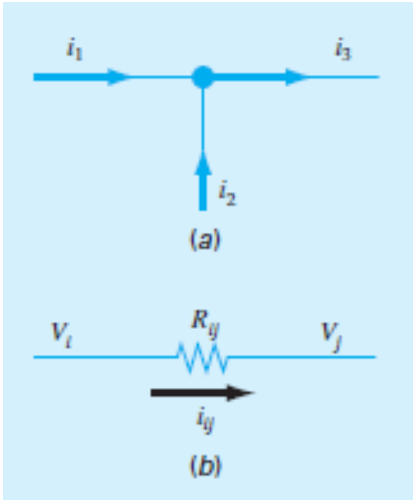


Figure 7.8: Schematic representations of (a) Kirchhoff's current rule and (b) Ohm's law.

The voltage (or loop) rule specifies that the algebraic sum of the potential differences (i.e., voltage changes) in any loop must equal zero. For a resistor circuit, this is expressed as

$$\sum \xi - \sum iR = 0 \tag{8.9}$$

where ξ is the emf (electromotive force) of the voltage sources, and R is the resistance of any resistors on the loop. Note that the second term derives from Ohm's law (Fig. 8.8 b), which states that the voltage drop across an ideal resistor is equal to the product of the current and the resistance. Kirchhoff's voltage rule is an expression of the conservation of energy.

Solution. Application of these rules results in systems of simultaneous linear algebraic equations because the various loops within a circuit are interconnected. For example, consider the circuit shown in Fig. 8.9. The currents associated with this circuit are unknown both in magnitude and direction. This presents no great difficulty because one simply assumes a direction for each current. If the resultant solution from Kirchhoff's laws is negative, then the assumed direction was incorrect. For example, Fig. 8.10 shows some assumed currents.

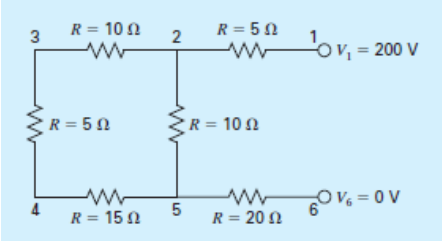


Figure 7.9: A resistor circuit to be solved using simultaneous linear algebraic equations.

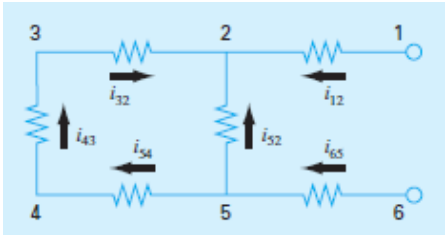


Figure 7.10: Assumed current directions.

Given these assumptions, Kirchhoff's current rule is applied at each node to yield

$$\begin{aligned} i_{12} + i_{52} + i_{32} &= 0 \\ i_{65} - i_{52} - i_{54} &= 0 \\ i_{43} - i_{32} &= 0 \\ i_{54} - i_{43} &= 0 \end{aligned}$$

Application of the voltage rule to each of the two loops gives

$$\begin{aligned} -i_{54}R_{54} - i_{43}R_{43} - i_{32}R_{32} + i_{52}R_{52} &= 0 \\ -i_{65}R_{65} - i_{52}R_{52} + i_{12}R_{12} - 200 &= 0 \end{aligned}$$

or, substituting the resistances from Fig. 8.9 and bringing constants to the right-hand side,

$$\begin{aligned} -15i_{54} - 5i_{43} - 10i_{32} + 10i_{52} &= 0 \\ -20i_{65} - 10i_{52} + 5i_{12} &= 200 \end{aligned}$$

Therefore, the problem amounts to solving six equations with six unknown currents. These equations can be expressed in matrix form as

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 10 & -10 & 0 & -15 & -5 \\ 5 & -10 & 0 & -20 & 0 & 0 \end{bmatrix} \begin{Bmatrix} i_{12} \\ i_{52} \\ i_{32} \\ i_{65} \\ i_{54} \\ i_{43} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 200 \end{Bmatrix}$$

Although impractical to solve by hand, this system is easily handled by MATLAB. The solution is

```
>> A=[1 1 1 0 0 0
0 -1 0 1 -1 0
0 0 -1 0 0 1
0 0 0 0 1 -1
0 10 -10 0 -15 -5
5 -10 0 -20 0 0];
>> b=[0 0 0 0 0 200]';
>> current=A\b
current =
    6.1538
    -4.6154
    -1.5385
    -6.1538
    -1.5385
    -1.5385
```

Thus, with proper interpretation of the signs of the result, the circuit currents and voltages are as shown in Fig. 8.11. The advantages of using MATLAB for problems of this type should be evident.

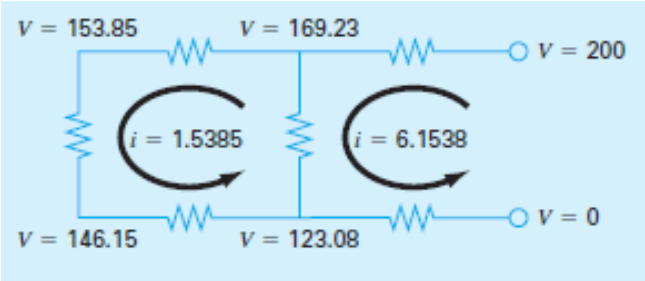


Figure 7.11: The solution for currents and voltages obtained using MATLAB.

PROBLEMS

8.1 Given a square matrix [A], write a single line MATLAB command that will create a new matrix $[Au_g]$ that consists of the original matrix $[A]$ augmented by an identity matrix $[I]$.

8.2 A number of matrices are defined as

$$[A]=\begin{bmatrix}4&5\\1&2\\5&6\end{bmatrix}\quad[B]=\begin{bmatrix}4&3&7\\1&2&6\\2&0&4\end{bmatrix}$$

$$\{C\}=\begin{Bmatrix}2\\6\\1\end{Bmatrix}\quad[D]=\begin{bmatrix}5&4&3&-7\\2&1&7&5\end{bmatrix}$$

$$[E]=\begin{bmatrix}1&5&6\\7&1&3\\4&0&6\end{bmatrix}$$

$$[F]=\begin{bmatrix}2&0&1\\1&7&4\end{bmatrix}\quad\lfloor G\rfloor=\begin{bmatrix}8&6&4\end{bmatrix}$$

Answer the following questions regarding these matrices:
(a) What are the dimensions of the matrices?
(b) Identify the square, column, and row matrices.
(c) What are the values of the elements: $a_{12}, b_{23}, d_{32}, e_{22}, f_{12}, g_{12}$?
(d) Perform the following operations:
(1) $[E]+[B]$
(2) $[A]+[F]$
(3) $[B]-[E]$
(4) $7\times[B]$
(5) $\{C\}^T$
(6) $[E]\times[B]$
(7) $[B]\times[E]$
(8) $[D]^T$
(9) $[G]\times\{C\}$
(10) $[I]\times[B]$
(11) $[E]^T\times[E]$
(12) $\{C\}^T\times\{C\}$

8.3 Write the following set of equations in matrix form:

$$50=5x_3-6x_2$$
$$2x_2+7x_3+30=0$$
$$x_1-7x_3=50-3x_2+5x_1$$

Use MATLAB to solve for the unknowns. In addition, use it to compute the transpose and the inverse of the coefficient matrix.

8.4 Three matrices are defined as

$$[A]=\begin{bmatrix}6&-1\\12&7\\-5&3\end{bmatrix}\quad[B]=\begin{bmatrix}4&0\\0.6&8\end{bmatrix}\quad[C]=\begin{bmatrix}1&-2\\-6&1\end{bmatrix}$$

(a) Perform all possible multiplications that can be computed between pairs of these matrices.
(b) Justify why the remaining pairs cannot be multiplied.
(c) Use the results of (a) to illustrate why the order of multiplication is important.

8.5 Solve the following system with MATLAB:

$$\begin{bmatrix}3+2i&4\\-i&1\end{bmatrix}\begin{Bmatrix}z_1\\z_2\end{Bmatrix}=\begin{Bmatrix}2+i\\3\end{Bmatrix}$$

8.6 Develop, debug, and test your own M-file to multiply two matrices that is, $[X]=[Y][Z]$, where $[Y]$ is m by n and $[Z]$ is n by p. Employ `for...  end` loops to implement the multiplication and include error traps to flag bad cases. Test the program using the matrices from Prob. 8.4.

8.7 Develop, debug, and test your own M-file to generate the transpose of a matrix. Employ `for...  end` loops to implement the transpose. Test it on the matrices from Prob. 8.4.

8.8 Develop, debug, and test your own M-file function to switch the rows of a matrix using a permutation matrix. The first lines of the function should be as follows:

```
function B = permut(A,r1,r2)
% Permut: Switch rows of matrix
A
% with a permutation matrix
% B = permut(A,r1,r2)
% input:
% A = original matrix
% r1, r2 = rows to be switched
% output:
% B = matrix with rows switched
```

Include error traps for erroneous inputs (e.g., user specifies rows that exceed the dimensions of the original matrix).

8.9 Five reactors linked by pipes are shown in Fig. P8.9. 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. P8.9 and express the equations in matrix form. Then use MATLAB to solve for the concentrations in each reactor.

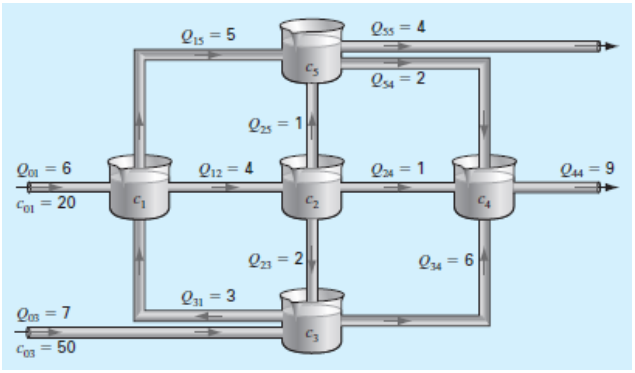


Figure 7.12: FIGURE P8.9

8.10 An important problem in structural engineering is that of finding the forces in a statically detemminate truss (Fig. P8.10). 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:

$$\sum F_H=0=-F_1\cos 30^\circ+F_3\cos 60^\circ+F_{1,h}$$
$$\sum F_V=0=-F_1\sin 30^\circ-F_3\sin 60^\circ+F_{1,v}$$

for node 2:

$$\sum F_H=0=F_2+F_1\cos 30^\circ+F_{2,h}+H_2$$
$$\sum F_V=0=F_1\sin 30^\circ+F_{2,v}+V_2$$

for node 3 :

$$\sum F_H = 0 = -F_2 - F_3 \cos 60^\circ + F_{3,h}$$
$$\sum F_V = 0 = F_3 \sin 60^\circ + F_{3,V} + V_3$$

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 $2000-$N downward force on node 1 corresponds to $F_{i,v} = -2000$. For this case, all other $F_{i,b}$'s and $F_{i\lambda h}$'s are zero. Express this set of linear algebraic equations in matrix form and then use MATLAB to solve for the unknowns.

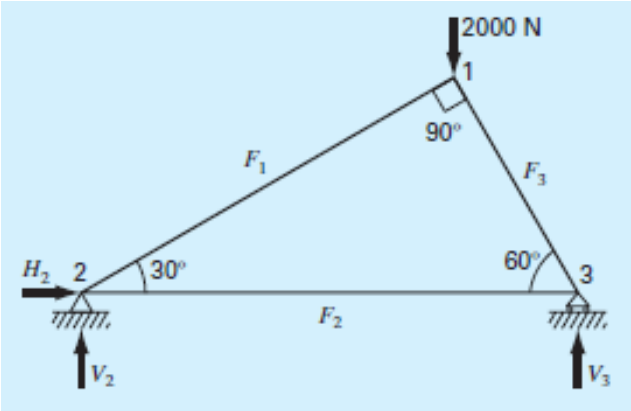


Figure 7.13: FIGURE P8.10

8.11 Consider the three mass-four spring system in Fig. P8.11. Determining the equations of motion from $\Sigma F_x = ma_x$ for each mass using its free-body diagram results in the following differential equations:

$$\ddot{x}_1 + \left(\frac{k_1+k_2}{m_1}\right)x_1 - \left(\frac{k_2}{m_1}\right)x_2 = 0$$
$$\ddot{x}_2 - \left(\frac{k_2}{m_2}\right)x_1 + \left(\frac{k_2+k_3}{m_2}\right)x_2 - \left(\frac{k_3}{m_2}\right)x_3 = 0$$
$$\ddot{x}_3 - \left(\frac{k_3}{m_3}\right)x_2 + \left(\frac{k_3+k_4}{m_3}\right)x_3 = 0$$

where $k_1 = k_4 = 10$ N/m, $k_2 = k_3 = 40$ N/m, and $m_1 = m_2 = m_3 = 1$ kg. The three equations can be written in matrix form: $0 = \{$ Acceleration vector $\} + [k/m$ matrix $]$ displacement vector $x\}$ 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.

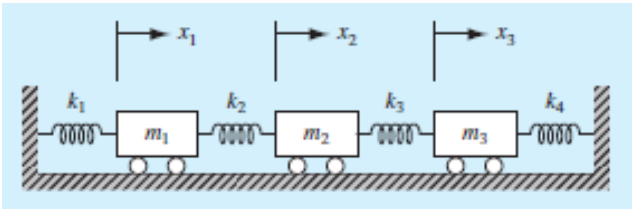


Figure 7.14: FIGURE P8.11

8.12 Perfom the same computation as in Example 8.2, but use five jumpers with the following characteristics:

Jumper	Mass (**kg**)	Spring Constant (**N/m**)	Unstretched Cord Length (**m**)
1	65	80	10
2	75	40	10
3	60	70	10
4	75	100	10
5	90	20	10

8.13 Three masses are suspended vertically by a series of identical springs where mass 1 is at the top and mass 3 is at the bottom. If $g = 9.81$ m/s^2, $m_1 = 2$ kg, $m_2 = 2.5$ kg, $m_3 = 3$ kg, and the k's $= 15$ kg/s^2, use MATLAB to solve for the displacements x.

8.14 Perform the same computation as in Sec. 8.3, but for the circuit in Fig. P8.14.

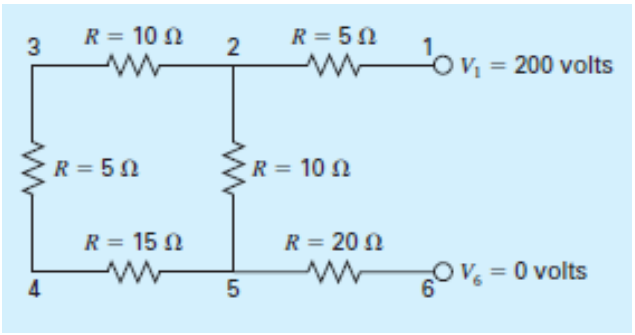


Figure 7.15: FIGURE P8.14

8.15 Perform the same computation as in Sec. 8.3, but for the circuit in Fig. P8.15.

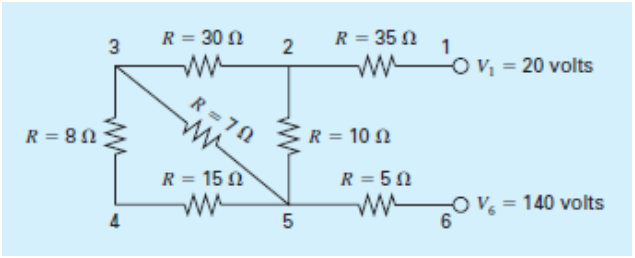


Figure 7.16: FIGURE P8.15

Chapter 8

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.
- Knowing how to compute the determinant as part of the Gauss elimination algorithm with partial pivoting.
- Recognizing how the banded structure of a tridiagonal system can be exploited to obtain extremely efficient solutions.

At the end of Chap. 8, 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 9 and 10 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, $x_2 =$ (slope) $x_1 +$ intercept. When these equations are graphed, the values of x_1 and x_2 at the intersection of the lines represent the solution (Fig. 9.1). 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. 9.2 depicts three cases that can pose problems when solving sets of linear equations. Fig. 9.2*a* shows the case where the two equations represent parallel lines. For such situations, there is no solution because the lines never cross. Figure 9.2*b* 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. 9.2c) 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 roundoff error.

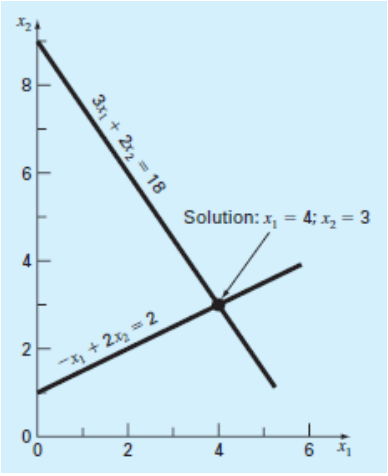


Figure 8.1: Graphical solution of a set of two simultaneous linear algebraic equations. The intersection of the lines represents the solution.

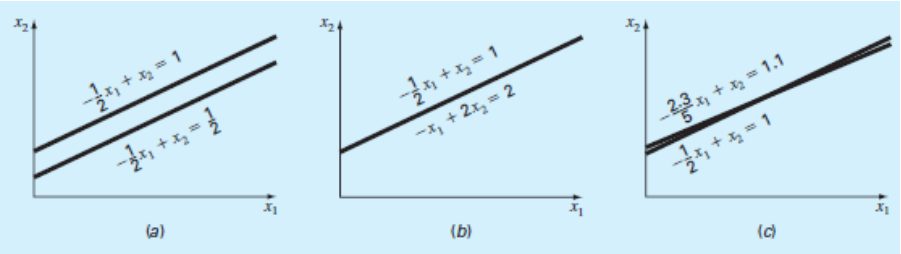


Figure 8.2: Graphical depiction of singular and ill-conditioned systems: (a) no solution, (b) infinite solutions, and (c) ill-conditioned system where the slopes are so close that the point of intersection is difficult to detect visually.

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

$$D = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

(9.1) where the 2×2 determinants are called minors.

Example 8.1. Determinants

Problem Statement. Compute values for the determinants of the systems represented in Figs. 9.1 and 9.2.

Solution. For Fig. 9.1:

$$D = \begin{vmatrix} 3 & 2 \\ -1 & 2 \end{vmatrix} = 3(2) - 2(-1) = 8$$

For Fig. 9.2a:

$$D = \begin{vmatrix} -\frac{1}{2} & 1 \\ -\frac{1}{2} & 1 \end{vmatrix} = -\frac{1}{2}(1) - 1\left(\frac{-1}{2}\right) = 0$$

For Fig. 9.2b:

$$D = \begin{vmatrix} -\frac{1}{2} & 1 \\ -1 & 2 \end{vmatrix} = -\frac{1}{2}(2) - 1(-1) = 0$$

For Fig. 9.2c:

$$D = \begin{vmatrix} -\frac{1}{2} & 1 \\ -\frac{2.3}{5} & 1 \end{vmatrix} = -\frac{1}{2}(1) - 1\left(\frac{-2.3}{5}\right) = -0.04$$

■

In the foregoing example, the singular systems had zero determinants. Additionally, the results suggest that the system that is almost singular (Fig. 9.2c) has a determinant that is close to zero. These ideas will be pursued further in our subsequent discussion of ill-conditioning in Chap. 11 .

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 $b_1, b_2, \ldots, b_n$. 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. (9.1)]:

$$D = 0.3\begin{vmatrix} 1 & 1.9 \\ 0.3 & 0.5 \end{vmatrix} - 0.52\begin{vmatrix} 0.5 & 1.9 \\ 0.1 & 0.5 \end{vmatrix} + 1\begin{vmatrix} 0.5 & 1 \\ 0.1 & 0.3 \end{vmatrix} = -0.0022$$

The solution can be calculated as

$$x_1 = \frac{\begin{vmatrix} -0.01 & 0.52 & 1 \\ 0.67 & 1 & 1.9 \\ -0.44 & 0.3 & 0.5 \end{vmatrix}}{-0.0022} = \frac{0.03278}{-0.0022} = -14.9$$

$$x_2 = \frac{\begin{vmatrix} 0.3 & -0.01 & 1 \\ 0.5 & 0.67 & 1.9 \\ 0.1 & -0.44 & 0.5 \end{vmatrix}}{-0.0022} = \frac{0.0649}{-0.0022} = -29.5$$

$$x_3 = \frac{\begin{vmatrix} 0.3 & 0.52 & -0.01 \\ 0.5 & 1 & 0.67 \\ 0.1 & 0.3 & -0.44 \end{vmatrix}}{-0.0022} = \frac{-0.04356}{-0.0022} = 19.8$$

■

The det Function. The determinant can be computed directly in MATLAB with the det function. For example, using the system from the previous example:

```
>> A=[0.3 0.52 1;0.5 1 1.9;0.1 0.3 0.5];
>> D=det(A)
D =
    -0.0022
```

Cramer's rule can be applied to compute x_1 as in

```
>> A(:,1)=[-0.01;0.67;-0.44]
A =
    -0.0100 0.5200 1.0000
    0.6700 1.0000 1.9000
    -0.4400 0.3000 0.5000
>> x1=det(A)/D
x1 =
    -14.9000
```

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 9.1.3-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:

$$a_{11}x_1 + a_{12}x_2 = b_1 \tag{9.2}$$
$$a_{21}x_1 + a_{22}x_2 = b_2 \tag{9.3}$$

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. (9.2) might be multiplied by a_{21} and Eq. (9.3) by a_{11} to give

$$a_{21}a_{11}x_1 + a_{21}a_{12}x_2 = a_{21}b_1 \tag{9.4}$$
$$a_{11}a_{21}x_1 + a_{11}a_{22}x_2 = a_{11}b_2 \tag{9.5}$$

Subtracting Eq. (9.4) from Eq. (9.5) 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

$$x_2 = \frac{a_{11}b_2 - a_{21}b_1}{a_{11}a_{22} - a_{21}a_{12}} \tag{9.6}$$

Equation (9.6) can then be substituted into Eq. (9.2), which can be solved for

$$x_1 = \frac{a_{22}b_1 - a_{12}b_2}{a_{11}a_{22} - a_{21}a_{12}} \tag{9.7}$$

Notice that Eqs. (9.6) and (9.7) 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 9.2, the technique can be formalized and readily programmed for the computer.

8.2. NAIVE GAUSS ELIMINATION

In Section 9.1.3, the elimination of unknowns was used to solve a pair of simultaneous equations. The procedure consisted of two steps (Fig. 9.3): 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 9.3 will deal with the additional features required for an effective computer program. The approach is designed to solve a general set of n equations:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \qquad (9.8a)$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \qquad (9.8b)$$
$$\vdots$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n \qquad (9.8c)$$

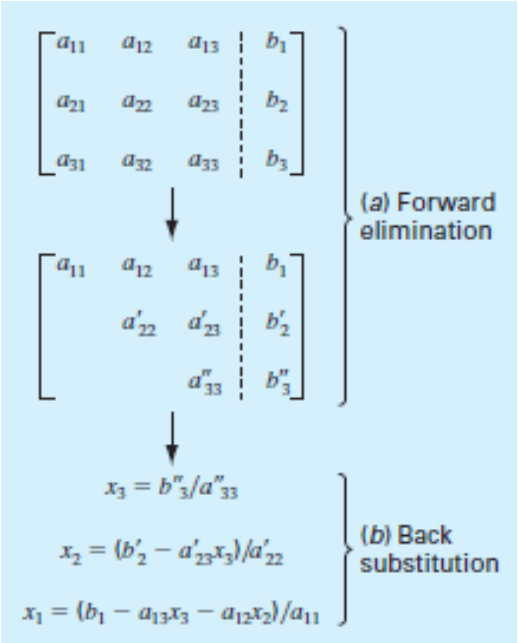


Figure 8.3: The two phases of Gauss elimination: (a) forward elimination and (b) back substitution.

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. 9.3a). The initial step will be to eliminate the first unknown x_1 from the second through the nth equations. To do this, multiply Eq. (9.8a) by a_{21}/a_{11} to give

$$a_{21}x_1 + \frac{a_{21}}{a_{11}}a_{12}x_2 + \frac{a_{21}}{a_{11}}a_{13}x_3 + \cdots + \frac{a_{21}}{a_{11}}a_{1n}x_n = \frac{a_{21}}{a_{11}}b_1 \tag{9.9}$$

This equation can be subtracted from Eq. (9.8b) 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. (9.8a) 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:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \qquad (9.10a)\\
a'_{22}x_2 + a'_{23}x_3 + \cdots + a'_{2n}x_n &= b'_2 \qquad (9.10b)\\
a'_{32}x_2 + a'_{33}x_3 + \cdots + a'_{3n}x_n &= b'_3 \qquad (9.10c)\\
&\vdots\\
a'_{n2}x_2 + a'_{n3}x_3 + \cdots + a'_{nn}x_n &= b'_n \qquad (9.10d)
\end{aligned}$$

For the foregoing steps, Eq. $(9.8a)$ 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. $(9.10c)$ through $(9.10d)$. To do this, multiply Eq. (9.10b) by a'_{32}/a'_{22} and subtract the result from Eq. (9.10c). Perform a similar elimination for the remaining equations to yield

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1\\
a'_{22}x_2 + a'_{23}x_3 + \cdots + a'_{2n}x_n &= b'_2\\
a''_{33}x_3 + \cdots + a''_{3n}x_n &= b''_3\\
&\vdots\\
a''_{n3}x_3 + \cdots + a'_{nn}x_n &= b''_n
\end{aligned}$$

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 nth equation. At this point, the system will have been transformed to an upper triangular system:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \qquad (9.11a)\\ a'_{22}x_2 + a'_{23}x_3 + \cdots + a'_{2n}x_n &= b'_2 \qquad (9.11b)\\ a''_{33}x_3 + \cdots + a''_{3n}x_n &= b''_3 \qquad (9.11c)\\ \ddots &\vdots\\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \qquad (9.11d) \end{aligned}$$

Back Substitution. Equation $(9.11d)$ can now be solved for x_n :

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}} \tag{9.12}$$

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:

$$x_i = \frac{b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)} x_j}{a_{ii}^{(i-1)}} \quad \text{for } i = n-1, n-2, \ldots, 1 \tag{9.13}$$

Example 8.3. Naive Gauss Elimination

Problem Statement. Use Gauss elimination to solve

$$\begin{aligned} 3x_1 - 0.1x_2 - 0.2x_3 &= 7.85 \qquad (E9.3.1)\\ 0.1x_1 + 7x_2 - 0.3x_3 &= -19.3 \qquad (E9.3.2)\\ 0.3x_1 - 0.2x_2 + 10x_3 &= 71.4 \qquad (E9.3.3) \end{aligned}$$

Solution. The first part of the procedure is forward elimination. Multiply Eq. (E9.3.1) by $0.1/3$ and subtract the result from Eq. (E9.3.2) to give

$$7.00333x_2 - 0.293333x_3 = -19.5617$$

Then multiply Eq. (E9.3.1) by $0.3/3$ and subtract it from Eq. (E9.3.3). After these operations, the set of equations is

$$\begin{aligned} 3x_1 - \quad 0.1x_2 - \quad 0.2x_3 &= 7.85 \qquad (E9.3.4)\\ 7.00333x_2 - 0.293333x_3 &= -19.5617 \qquad (E9.3.5)\\ -0.190000x_2 + 10.0200x_3 &= 70.6150 \qquad (E9.3.6) \end{aligned}$$

To complete the forward elimination, x_2 must be removed from Eq. (E9.3.6). To accomplish this, multiply Eq. (E9.3.5) by $-0.190000/7.00333$ and subtract the result from Eq. (E9.3.6). This eliminates x_2 from the third equation and reduces the system to an upper triangular form, as in

$$\begin{aligned} 3x_1 - 0.1x_2 - \quad 0.2x_3 &= 7.85 \qquad (E9.3.7)\\ 7.00333x_2 - 0.293333x_3 &= -19.5617 \qquad (E9.3.8)\\ 10.0120x_3 &= 70.0843 \qquad (E9.3.9) \end{aligned}$$

We can now solve these equations by back substitution. First, Eq. (E9.3.9) can be solved for

$$x_3 = \frac{70.0843}{10.0120} = 7.00003$$

This result can be back-substituted into Eq. (E9.3.8), 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. (E9.3.7), 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:

$$3(3) - 0.1(-2.5) - 0.2(7.00003) = 7.84999 \cong 7.85$$
$$0.1(3) + 7(-2.5) - 0.3(7.00003) = -19.30000 = -19.3$$
$$0.3(3) - 0.2(-2.5) + 10(7.00003) = 71.4003 \cong 71.4$$

■

```
function x = GaussNaive(A,b)
% GaussNaive: naive Gauss elimination
% x = 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.

8.2.1. MATLAB M-file: `GaussNaive`

An M-file that implements naive Gauss elimination is listed in Fig. 9.4. 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 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. (9.12) and (9.13). Again, MATLAB's ability to perform matrix operations allows Eq. (9.13) 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.

Before analyzing naive Gauss elimination, we will first define some quantities that facilitate operation counting:

$$\sum_{i=1}^{m} cf(i) = c\sum_{i=1}^{m} f(i) \sum_{i=1}^{m} f(i) + g(i) = \sum_{i=1}^{m} f(i) + \sum_{i=1}^{m} g(i) \qquad (9.14a,b)$$

$$\sum_{i=1}^{m} 1 = 1+1+1+\cdots+1 = m\sum_{i=k}^{m} 1 = m-k+1 \qquad (9.14c,d)$$

$$\sum_{i=1}^{m} i = 1+2+3+\cdots+m = \frac{m(m+1)}{2} = \frac{m^2}{2} + O(m) \qquad (9.14e)$$

$$\sum_{i=1}^{m} i^2 = 1^2+2^2+3^2+\cdots+m^2 = \frac{m(m+1)(2m+1)}{6} = \frac{m^3}{3} + O\left(m^2\right) \qquad (9.14f)$$

where $O(m^n)$ means "terms of order m^n and lower." Now let us examine the naive Gauss elimination algorithm (Fig. 9.4) 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. $(9.14d)$, this means that the number of iterations of the inner loop will be

$$\sum_{i=2}^{n} 1 = n - 2 + 1 = n - 1 \tag{9.15}$$

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 k	Inner Loop i	Addition/Subtraction Flops	Multiplication/Division Flops
1	$2,n$	$(n-1)(n)$	$(n-1)(n+1)$
2	$3,n$	$(n-2)(n-1)$	$(n-2)(n)$
$\vdots$	$\vdots$		
k	$k+1,n$	$(n-k)(n+1-k)$	$(n-k)(n+2-k)$
$\vdots$	$\vdots$		
$n-1$	n,n	(1)(2)	(1)(3)

Therefore, the total addition/subtraction flops for elimination can be computed as

$$\sum_{k=1}^{n-1}(n-k)(n+1-k) = \sum_{k=1}^{n-1}\left[n(n+1) - k(2n+1) + k^2\right] \tag{9.16}$$

Or

$$n(n+1)\sum_{k=1}^{n-1} 1 - (2n+1)\sum_{k=1}^{n-1} k + \sum_{k=1}^{n-1} k^2 \tag{9.17}$$

Applying some of the relationships from Eq. (9.14) yields

$$\left[n^3 + O(n)\right] - \left[n^3 + O\left(n^2\right)\right] + \left[\frac{1}{3}n^3 + O\left(n^2\right)\right] = \frac{n^3}{3} + O(n) \tag{9.18}$$

A similar analysis for the multiplication/division flops yields

$$\left[n^3 + O\left(n^2\right)\right] - \left[n^3 + O(n)\right] + \left[\frac{1}{3}n^3 + O\left(n^2\right)\right] = \frac{n^3}{3} + O\left(n^2\right) \tag{9.19}$$

Summing these results gives

$$\frac{2n^3}{3} + O\left(n^2\right) \tag{9.20}$$

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 $O\left(n^2\right)$ 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

$$n^2 + O(n) \tag{9.21}$$

Thus, the total effort in naive Gauss elimination can be represented as

$$\frac{2n^3}{3} + O\left(n^2\right) + n^2 + O(n) \xrightarrow{\text{as } n \text{ increases}} \frac{2n^3}{3} + O\left(n^2\right) \tag{9.22}$$

Two useful general conclusions can be drawn from this analysis:

1. As the system gets larger, the computation time increases greatly. As in Table 9.1, the amount of flops increases nearly three orders of magnitude for every order of magnitude increase in the number of equations.

TABLE 9.1 Number of flops for naive Gauss elimination.

n	Elimination	Back Substitution	Total Flops	$\mathbf{2n^3/3}$	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	6.67×10^9	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

$$2x_2 + 3x_3 = 8$$
$$4x_1 + 6x_2 + 7x_3 = -3$$
$$2x_1 - 3x_2 + 6x_3 = 5$$

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 most of the improvement comes from partial pivoting. In addition, 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

$$0.0003x_1 + 3.0000x_2 = 2.0001$$
$$1.0000x_1 + 1.0000x_2 = 1.0000$$

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 :

$$x_1 = \frac{2.0001 - 3(2/3)}{0.0003}$$

Due to subtractive cancellation, the result is very sensitive to the number of significant figures carried in the computation:

Signlficant 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. (E9.4.1), 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 Figgures	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. 9.5. It is identical to the M-file for naive Gauss elimination presented previously in Section 9.2.1 with the exception of the bold portion that implements 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, 1]=\max (x)
```

where y is the largest element in the vector x, and i is the index corresponding to that element.

```
function x = GaussPivot(A,b)
% GaussPivot: Gauss elimination pivoting
% x = GaussPivot(A,b): Gauss elimination with 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
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 Gauss elimination with partial pivoting.

8.3.2. Determinant Evaluation with Gauss Elimination

At the end of Sec. 9.1.2, we suggested that determinant evaluation by expansion of minors was impractical for large sets of equations. However, because the determinant has value in assessing system condition, it would be useful to have a practical method for computing this quantity.

Fortunately, Gauss elimination provides a simple way to do this. The method is based on the fact that the determinant of a triangular matrix can be simply computed as the product of its diagonal elements:

$$D = a_{11}a_{22}a_{33}\cdots a_{nn}$$

The validity of this formulation can be illustrated for a 3×3 system:

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{vmatrix}$$

where the determinant can be evaluated as [recall Eq. (9.1)]:

$$D = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ 0 & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} 0 & a_{23} \\ 0 & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} 0 & a_{22} \\ 0 & 0 \end{vmatrix}$$

or, by evaluating the minors:

$$D = a_{11}a_{22}a_{33} - a_{12}(0) + a_{13}(0) = a_{11}a_{22}a_{33}$$

Recall that the forward-elimination step of Gauss elimination results in an upper triangular system. Because the value of the determinant is not changed by the forwardelimination process, the determinant can be simply evaluated at the end of this step via

$$D = a_{11}a'_{22}a''_{33}\cdots a^{(n-1)}_{nA}$$

where the superscripts signify the number of times that the elements have been modified by the elimination process. Thus, we can capitalize on the effort that has already been expended in reducing the system to triangular form and, in the bargain, come up with a simple estimate of the determinant.

There is a slight modification to the above approach when the program employs partial pivoting. For such cases, the determinant changes sign every time a row is switched. One way to represent this is by modifying the determinant calculation as in

$$D = a_{11}a'_{22}a''_{33}\cdots a^{(n-1)}_{n\pi}(-1)^p$$

where p represents the number of times that rows are pivoted. This modification can be incorporated simply into a program by merely keeping track of the number of pivots that take place during the course of the computation.

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

$$\begin{bmatrix} f_1 & g_1 & & & & & & \\ e_2 & f_2 & g_2 & & & & & \\ & e_3 & f_3 & g_3 & & & & \\ & & \cdot & \cdot & \cdot & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & \cdot & \cdot & \cdot & \\ & & & & & e_{n-1} & f_{n-1} & g_{n-1} \\ & & & & & & e_n & f_n \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_{n-1} \\ x_n \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ \cdot \\ \cdot \\ r_{n-1} \\ r_n \end{Bmatrix} \qquad 9.23$$

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 eliminationthat 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:

$$\begin{bmatrix} 2.04 & -1 & & \\ -1 & 2.04 & -1 & \\ & -1 & 2.04 & -1 \\ & & -1 & 2.04 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 40.8 \\ 0.8 \\ 0.8 \\ 200.8 \end{Bmatrix}$$

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

$$\begin{bmatrix} 2.04 & -1 & & \\ & 1.550 & -1 & \\ & & 1.395 & -1 \\ & & & 1.323 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 40.8 \\ 20.8 \\ 14.221 \\ 210.996 \end{Bmatrix}$$

Now back substitution can be applied to generate the final solution:

$$x_4 = \frac{r_4}{f_4} = \frac{210.996}{1.323} = 159.480$$
$$x_3 = \frac{r_3 - g_3 x_4}{f_3} = \frac{14.221 - (-1)159.480}{1.395} = 124.538$$
$$x_2 = \frac{r_2 - g_2 x_3}{f_2} = \frac{20.800 - (-1)124.538}{1.550} = 93.778$$
$$x_1 = \frac{r_1 - g_1 x_2}{f_1} = \frac{40.800 - (-1)93.778}{2.040} = 65.970$$

■

```
function x = Tridiag(e,f,g,r)
% Tridiag: Tridiagonal equation solver banded system
% x = 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
```

Figure 8.6: An M-file to solve a tridiagonal system.

8.4.1. MATLAB M-file: `Tridiag`

An M-file that solves a tridiagonal system of equations is listed in Fig. 9.6. 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. 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. 9.6 executes much, much faster than Gauss elimination, particularly for large systems.

8.5. CASE STUDY - MODEL OF A HEATED ROD

Background. Linear algebraic equations can arise when modeling distributed systems. For example, Fig. 9.7 shows a long, thin rod positioned between two walls that are held at constant temperatures. Heat flows through the rod as well as between the rod and the surrounding air. For the steady-state case, a differential equation based on heat conservation can be written for such a system as

$$\frac{d^2T}{dx^2} + h'(T_a - T) = 0 \tag{9.24}$$

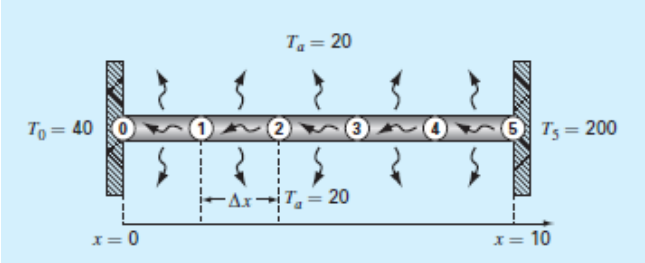


Figure 8.7: A noninsulated uniform rod positioned between two walls of constant but different temperature. The finite-difference representation employs four interior nodes.

where T = temperature $(°\mathrm{C})$, x = distance along the rod(m), h' = a heat transfer coefficient between the rod and the surrounding air $\left(\mathrm{m}^{-2}\right)$, and T_a = the air temperature $(°\mathrm{C})$.

Given values for the parameters, forcing functions, and boundary conditions, calculus can be used to develop an analytical solution. For example, if $h' = 0.01, T_a = 20, T(0) = 40$, and $T(10) = 200$, the solution is

$$T = 73.4523e^{0.1x} - 53.4523e^{-0.1x} + 20$$

9.25

Although it provided a solution here, calculus does not work for all such problems. In such instances, numerical methods provide a valuable alternative. In this case study, we will use finite differences to transform this differential equation into a tridiagonal system of linear algebraic equations which can be readily solved using the numerical methods described in this chapter.

Solution. Equation (9.24) can be transformed into a set of linear algebraic equations by conceptualizing the rod as consisting of a series of nodes. For example, the rod in Fig. 9.7 is divided into six equispaced nodes. Since the rod has a length of 10 , the spacing between nodes is $\Delta x = 2$.

Calculus was necessary to solve Eq. (9.24) because it includes a second derivative. As we learned in Sec. 4.3.4, finite-difference approximations provide a means to transform derivatives into algebraic form. For example, the second derivative at each node can be approximated as

$$\frac{d^2T}{dx^2} = \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$

where T_i designates the temperature at node i. This approximation can be substituted into Eq. (9.24) to give

$$\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} + h'(T_a - T_i) = 0$$

Chapter 9

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. As was previously mentioned in Section 1.4, 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 17.1 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 17.1 Density (ρ), dynamic viscosity (μ), and kinematic viscosity (v) as a function of temperature (T) at 1 atm as reported by White (1999).

T°C	ρ, kg/m^3	μ, N·s/m^2	v, m^2/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 exhibit significant curvature. In this chapter, we will explore a number of different approaches for obtaining adequate estimates for such situations.

9.1. 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

$$f(x) = a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1} \tag{17.1}$$

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. 17.1a). Similarly, only one parabola connects a set of three points (Fig. 17.1b). Polynomial interpolation consists of determining the unique $(n-1)$thorder 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. (17.1). Rather than using increasing powers of x, it uses decreasing powers as in

$$f(x) = p_1x^{n-1} + p_2x^{n-2} + \cdots + p_{n-1}x + p_n \tag{17.2}$$

To be consistent with MATLAB, we will adopt this scheme in the following section

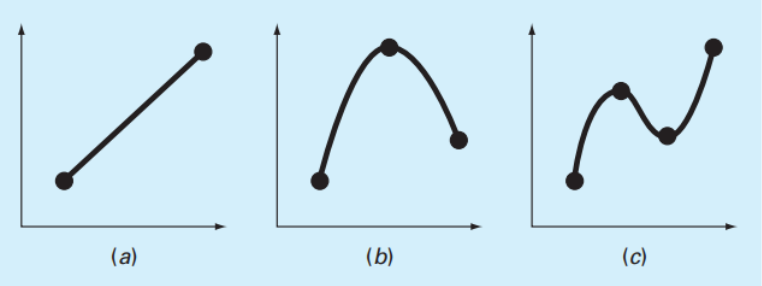


Figure 9.1: Examples of interpolating polynomials: (a) first-order (linear) connecting two points, (b) second-order (quadratic or parabolic) connecting three points, and (c) third-order (cubic) connecting four points.

9.1.1. Determining Polynomial Coefficients

A straightforward way for computing the coefficients of Eq. (17.2) 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 9. 1. First-Order Splines *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 17.1 :

$$\begin{aligned} x_1 &= 300 \quad f(x_1) = 0.616 \\ x_2 &= 400 \quad f(x_2) = 0.525 \\ x_3 &= 500 \quad f(x_3) = 0.457 \end{aligned}$$

Each of these pairs can be substituted into Eq. (17.2) to yield a system of three equations:

$$\begin{aligned} 0.616 &= p_1(300)^2 + p_2(300) + p_3 \\ 0.525 &= p_1(400)^2 + p_2(400) + p_3 \\ 0.457 &= p_1(500)^2 + p_2(500) + p_3 \end{aligned}$$

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 17.1 provides an easy way to perform interpolation, it has a serious deficiency. To understand this flaw, notice that the coefficient matrix in Example 17.1 has a decided structure. This can be seen clearly by expressing it in general terms:

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{Bmatrix} p_1 \\ p_2 \\ p_3 \end{Bmatrix} = \begin{Bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{Bmatrix}$$

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 17.1 as

```
>> cond(A)
ans =
    5.8932e+006
```

This condition number, which is quite large for a 3 x 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.

9.1.2. MATLAB Functions: polyfit and polyval

Recall from Section 14.5.2, 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 17.1:

```
>> 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 agree with those obtained previously in Example 17.1 with simultaneous equations.

9.2. NEWTON INTERPOLATING POLYNOMIAL

There are a variety of alternative forms for expressing an interpolating polynomial beyond the familiar format of Eq. (17.2). Newton's 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.

9.2.1. 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. 17.2. Using similar triangles,

$$\frac{f_1(x) - f(x_1)}{x - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \tag{17.4}$$

which can be rearranged to yield

$$f_1(x) = f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1) \tag{17.5}$$

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 line connecting the points, the term $[f_1(x_2) - f(x_1)]/(x_2 - x_1)$ is a finite-difference

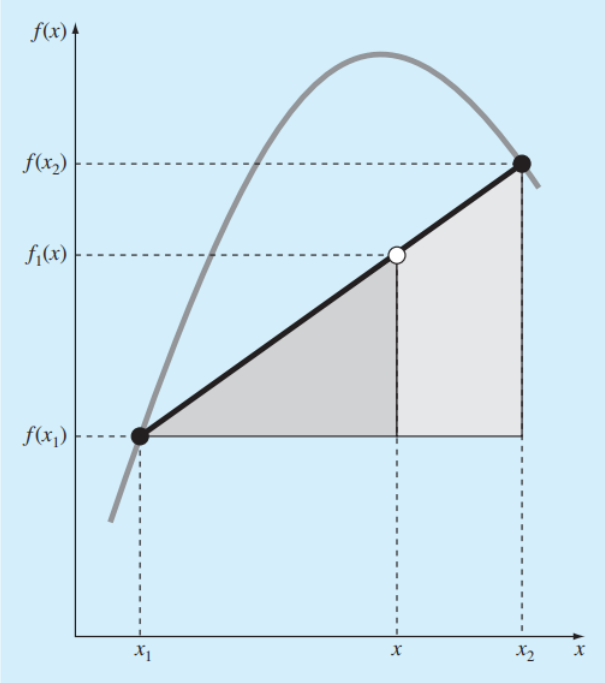


Figure 9.2: Graphical depiction of linear interpolation. The shaded areas indicate the similar triangles used to derive the Newton linear-interpolation formula [Eq. (17.5)].

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 9. 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. (17.5) 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 $\varepsilon_t = 48.3\%$. Using the smaller interval from $x_1 = 1$ to $x_2 = 4$ yields

$$f_1(2) = 0 + \frac{1.386294 - 0}{4 - 1}(2 - 1) = 0.4620981$$

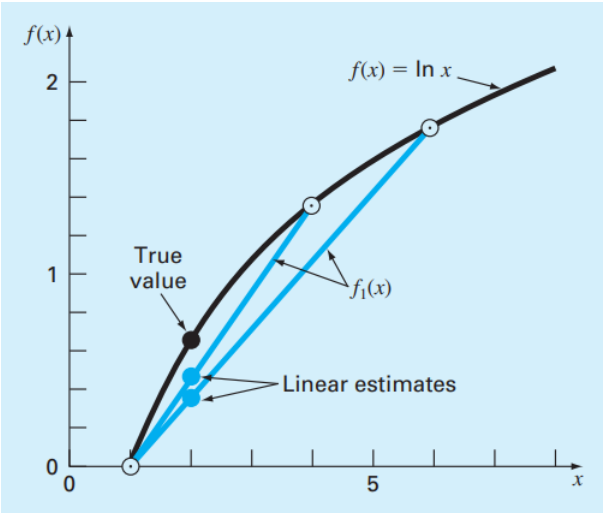


Figure 9.3: Two linear interpolations to estimate ln 2. Note how the smaller interval provides a better estimate.

Thus, using the shorter interval reduces the percent relative error to $\varepsilon_t = 33.3\%$. Both interpolations are shown in Fig. 17.3, along with the true function.

9.2.2. Quadratic Interpolation

The error in Example 17.2 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

$$f_2(x) = b_1 + b_2(x - x_1) + b_3(x - x_1)(x - x_2) \tag{17.6}$$

A simple procedure can be used to determine the values of the coefficients. For b_1, Eq. (17.6) with $x = x_1$ can be used to compute

$$b_1 = f(x_1) \tag{17.7}$$

Equation (17.7) can be substituted into Eq. (17.6), which can be evaluated at $x = x_2$ for

$$b_2 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \tag{17.8}$$

Finally, Eqs. (17.7) and (17.8) can be substituted into Eq. (17.6), which can be evaluated at $x = x_3$ and solved (after some algebraic manipulations) for

$$b_3 = \frac{\frac{f(x_3) - f(x_2)}{x_3 - x_2} - \frac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_1} \tag{17.9}$$

Notice that, as was the case with linear interpolation, b2 still represents the slope of the line connecting points x1 and x2. Thus, the first two terms of Eq. (17.6) are equivalent to linear interpolation between x1 and x2, as specified previously in Eq. (17.5). 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. (17.6), we should examine the form of the coefficient b_3. It is very similar to the finite-difference approximation of the second derivative introduced previously in Eq. (4.27). Thus, Eq. (17.6) 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 9. 3. Quadratic Interpolation *Problem Statement.* Employ a second-order Newton polynomial to estimate ln 2 with the same three points used in Example 17.2:

$$\begin{aligned} x_1 &= 1 \quad f(x_1) = 0 \\ x_2 &= 4 \quad f(x_2) = 1.386294 \\ x_3 &= 6 \quad f(x_3) = 1.791759 \end{aligned}$$

Solution. Applying Eq. (17.7) yields

$$b_1 = 0$$

Equation (17.8) gives

$$b_2 = \frac{1.386294 - 0}{4 - 1} = 0.4620981$$

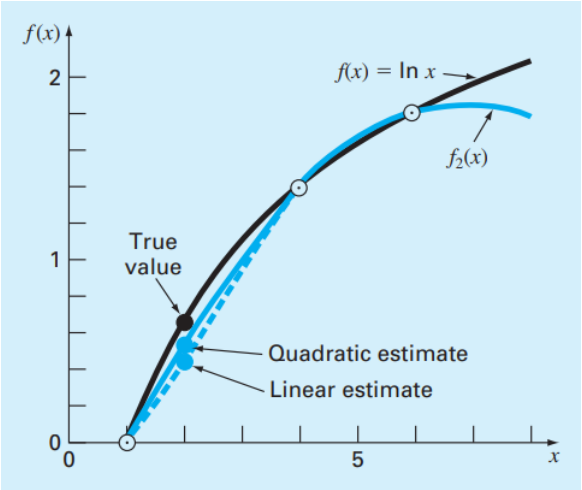


Figure 9.4: The use of quadratic interpolation to estimate ln 2. The linear interpolation from x = 1 to 4 is also included for comparison.

and Eq. (17.9) yields

$$b_3 = \frac{\frac{1.791759-1.386294}{6-4} - 0.4620981}{6-1} = -0.0518731$$

Substituting these values into Eq. (17.6) 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. 17.4) improves the interpolation compared with the result obtained using straight lines in Example 17.2 and Fig. 17.3.

9.2.3. 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

$$f_{n-1}(x) = b_1 + b_2(x-x_1) + \cdots + b_n(x-x_1)(x-x_2)\cdots(x-x_{n-1}) \tag{17.10}$$

As was done previously with linear and quadratic interpolation, data points can be used to evaluate the coefficients $b_1, b_2, \ldots, b_n$. For an $(n-1)$ th-order polynomial, n data points are required: $[x_1, f(x_1)], [x_2, f(x_2)], \ldots, [x_n, f(x_n)]$. We use these data points and the following equations to evaluate the coefficients:

$$b_1 = f(x_1) \tag{17.11}$$

$$b_2 = f[x_2, x_1] \tag{17.12}$$

$$b_3 = f[x_3, x_2, x_1] \tag{17.13}$$

$$\vdots$$

$$b_n = f[x_n, x_{n-1}, \ldots, x_2, x_1] \tag{17.14}$$

where the bracketed function evaluations are finite divided differences. For example, the first finite divided difference is represented generally as

$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j} \tag{17.15}$$

The second finite divided difference, which represents the difference of two first divided differences, is expressed generally as

$$f[x_i, x_j, x_k] = \frac{f[x_i, x_j] - f[x_j, x_k]}{x_i - x_k} \tag{17.16}$$

Similarly, the nth finite divided difference is

$$f[x_n, x_{n-1}, \ldots, x_2, x_1] = \frac{f[x_n, x_{n-1}, \ldots, x_2] - f[x_{n-1}, x_{n-2}, \ldots, x_1]}{x_n - x_1} \tag{17.17}$$

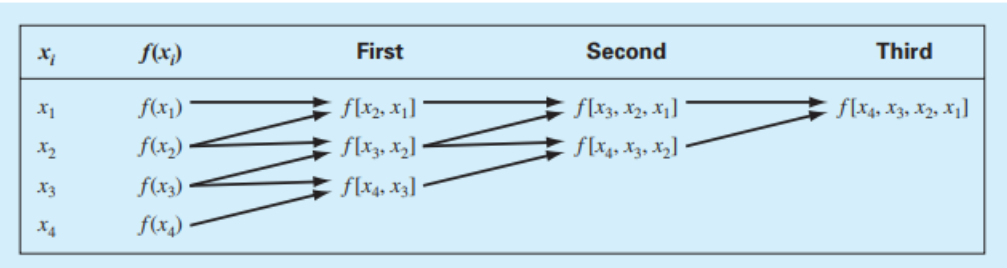


Figure 9.5: Graphical depiction of the recursive nature of finite divided differences. This representation is referred to as a divided difference table.

These differences can be used to evaluate the coefficients in Eqs. (17.11) through (17.14), which can then be substituted into Eq. (17.10) to yield the general form of Newton's interpolating polynomial:

$$\begin{aligned} f_{n-1}(x) = {} & f(x_1) + (x-x_1) f[x_2, x_1] + (x-x_1)(x-x_2) f[x_3, x_2, x_1] \\ & + \cdots + (x-x_1)(x-x_2)\cdots(x-x_{n-1}) f[x_n, x_{n-1}, \ldots, x_2, x_1] \end{aligned} \tag{17.18}$$

We should note that it is not necessary that the data points used in Eq. (17.18) be equally spaced or that the abscissa values necessarily be in ascending order, as illustrated in the following example. However, the points should be ordered so that they are centered around and as close as possible to the unknown. Also, notice how Eqs. (17.15) through (17.17) are recursiveâĂŤthat is, higher-order differences are computed by taking differences of lower-order differences (Fig. 17.5). This property will be exploited when we develop an efficient M-file to implement the method.

Example 9. 4. Newton Interpolating Polynomial *Problem Statement.*In Example 17.3, 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. (17.10) 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. (17.15)]

$$f[x_2, x_1] = \frac{1.386294 - 0}{4 - 1} = 0.4620981$$
$$f[x_3, x_2] = \frac{1.791759 - 1.386294}{6 - 4} = 0.2027326$$
$$f[x_4, x_3] = \frac{1.609438 - 1.791759}{5 - 6} = 0.1823216$$

The second divided differences are [Eq. (17.16)]

$$f[x_3, x_2, x_1] = \frac{0.2027326 - 0.4620981}{6 - 1} = -0.05187311$$
$$f[x_4, x_3, x_2] = \frac{0.1823216 - 0.2027326}{5 - 4} = -0.02041100$$

The third divided difference is [Eq. (17.17) 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. (17.10). 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 $\varepsilon_t = 9.3\%$. The complete cubic polynomial is shown in Fig. 17.6.

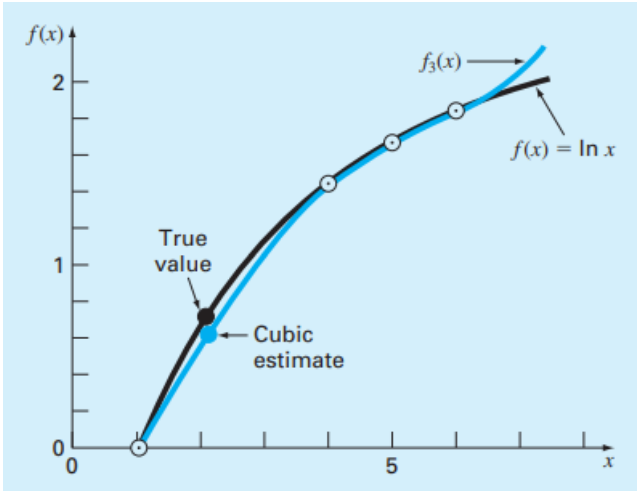


Figure 9.6: The use of cubic interpolation to estimate ln 2.

9.2.4. MATLAB M-file: Newtint

It is straightforward to develop an M-file to implement Newton interpolation.As in Fig. 17.7, 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. (17.18) to perform the interpolation.
An example of a session using the function would be to duplicate the calculation we just performed in Example 17.3:

```
>> format long
>> x = [1 4 6 5]';
```

Figure 9.7: An M-file to implement Newton interpolation.

```
function yint = Newtint(x,y,xx)
% Newtint: Newton interpolating polynomial
% yint = Newtint(x,y,xx): 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(y);
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
```

9.3. 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:

$$f(x) = L_1 f(x_1) + L_2 f(x_2) \tag{17.19}$$

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. 17.19 yields the straight line that connects the points (Fig. 17.8):

$$f_1(x) = \frac{x - x_2}{x_1 - x_2} f(x_1) + \frac{x - x_1}{x_2 - x_1} f(x_2) \tag{17.20}$$

where the nomenclature $f_1(x)$ designates that this is a first-order polynomial. Equation (17.20) 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

$$f_2(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} f(x_1) + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} f(x_2) + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} f(x_3) \tag{17.21}$$

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

$$f_{n-1}(x) = \sum_{i=1}^{n} L_i(x) f(x_i) \tag{17.22}$$

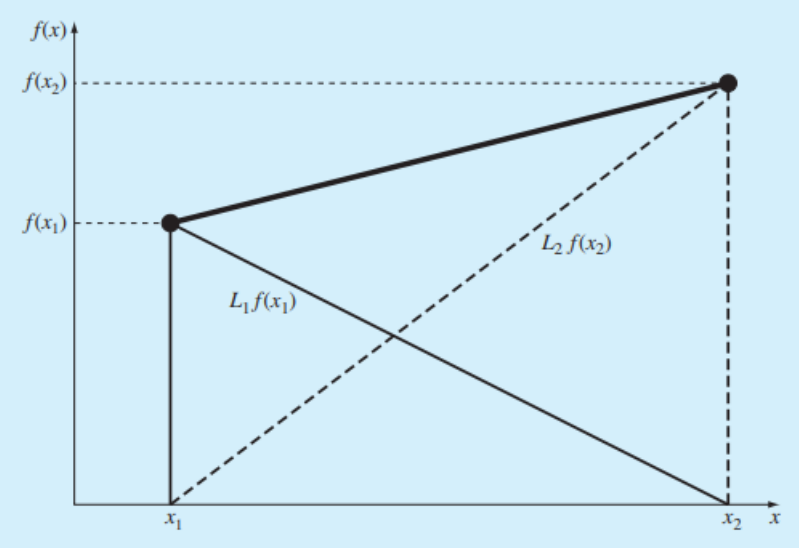


Figure 9.8: A visual depiction of the rationale behind Lagrange interpolating polynomials. The figure shows the first-order case. Each of the two terms of Eq. (17.20) passes through one of the points and is zero at the other. The summation of the two terms must, therefore, be the unique straight line that connects the two points.

where

$$L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{17.23}$$

where $n =$ the number of data points and $\prod$ designates the "product of."

Example 9. 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 ÂřC based on the following data:

$$\begin{array}{ll} x_1 = 0 & f(x_1) = 3.85 \\ x_2 = 20 & f(x_2) = 0.800 \\ x_3 = 40 & f(x_3) = 0.212 \end{array}$$

Solution. The first-order polynomial [Eq. (17.20)] can be used to obtain the estimate at $x = 15$:

$$f_1(x) = \frac{15-20}{0-20} 3.85 + \frac{15-0}{20-0} 0.800 = 1.5625$$

In a similar fashion, the second-order polynomial is developed as [Eq. (17.21)]

$$f_2(x) = \frac{(15-20)(15-40)}{(0-20)(0-40)} 3.85 + \frac{(15-0)(15-40)}{(20-0)(20-40)} 0.800 + \frac{(15-0)(15-20)}{(40-0)(40-20)} 0.212 = 1.3316875$$

9.3.1. MATLAB M-file: Lagrange

It is straightforward to develop an M-file based on Eqs. (17.22) and (17.23). As in Fig. 17.9, 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 9.9: An M-file to implement Lagrange interpolation.

```
function yint = Lagrange(x,y,xx)
% Lagrange: Lagrange interpolating polynomial
%   yint = Lagrange(x,y,xx): Uses an (n - 1)-order
%       Lagrange interpolating polynomial based on n data points
%       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 ÂřC based on the first four values from Table 17.1. 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
```

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 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

$$x = \frac{0.375 \pm \sqrt{(-0.375)^2 - 4(0.041667)0.78333}}{2(0.041667)} = \begin{matrix} 5.704158 \\ 3.295842 \end{matrix}$$

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.

9.4. EXTRAPOLATION AND OSCILLATIONS

Before leaving this chapter, there are two issues related to polynomial interpolation that must be addressed. These are extrapolation and oscillations.

9.4.1. Extrapolation

Extrapolation is the process of estimating a value of $f(x)$ that lies outside the range of the known base points, $x_1, x_2, \ldots, x_n$. As depicted in Fig. 17.10, the open-ended nature of

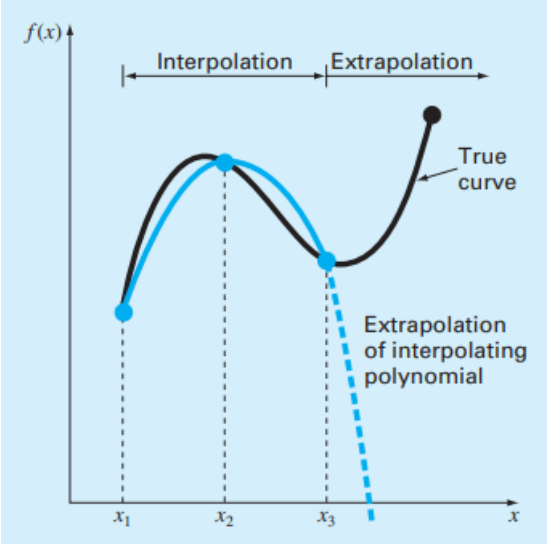


Figure 9.10: Illustration of the possible divergence of an extrapolated prediction. The extrapolation is based on fitting a parabola through the first three known points.

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 9. 6. Dangers of Extrapolation *Problem Statement.* This example is patterned after one originally developed by Forsythe, Malcolm, and Moler. The population in millions of the United States from 1920 to 2000 can be tabulated as

Dafe	1920	1930	1940	1950	1960	1970	1980	1990	2000
Population	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)
```

The polyfit function can be used to compute the coefficients

```
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 that 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)
```

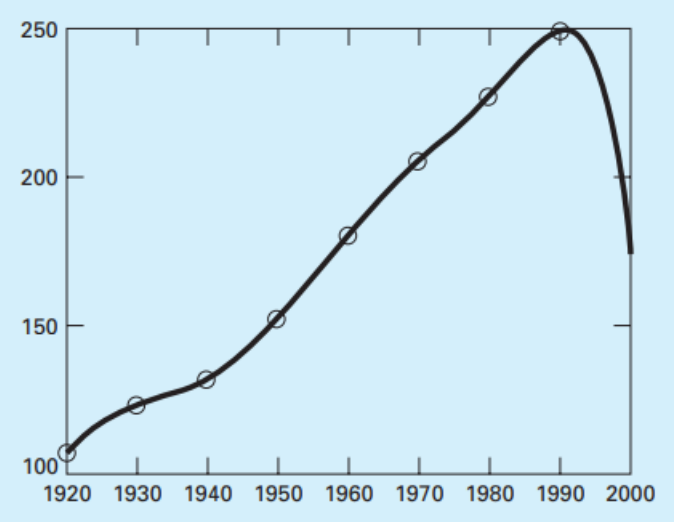


Figure 9.11: Use of a seventh-order polynomial to make a prediction of U.S. population in 2000 based on data from 1920 through 1990..

As in Fig. 17.11, 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.

9.4.2. Oscillations

Although âĂIJmore is betterâĂİ in many contexts, it is absolutely not true for polynomial interpolation. Higher-order polynomials tend to be very ill-conditionedâĂŤthat is, they tend to be highly sensitive to round-off error. The following example illustrates this point nicely.

Example 9. 7. Dangers of Higher-Order Polynomial Interpolation *Problem Statement.* In 1901, Carl Runge published a study on the dangers of higherorder polynomial interpolation. He looked at the following simple-looking function:

$$f(x) = \frac{1}{1+25x^2} \tag{17.24}$$

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 tenthorder polynomials to 5 and 11 equally spaced points generated with Eq. (17.24). 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 finally 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 fourthorder polynomial, and the *polval* 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. 17.12, 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);
```

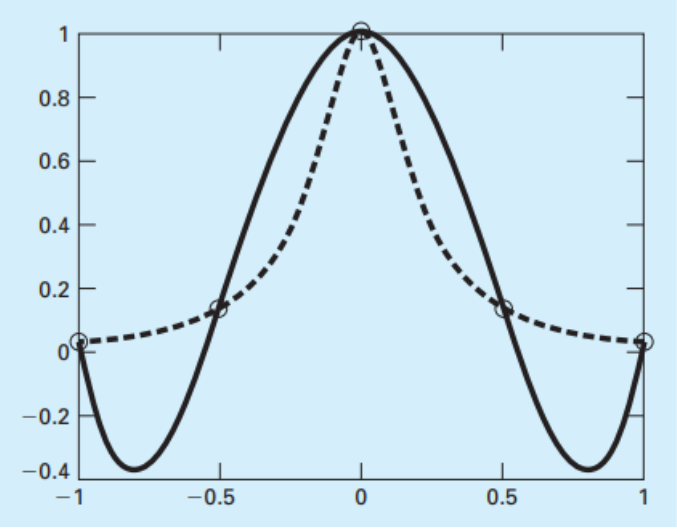


Figure 9.12: Comparison of Runge's function (dashed line) with a fourth-order polynomial fit to 5 points sampled from the function.

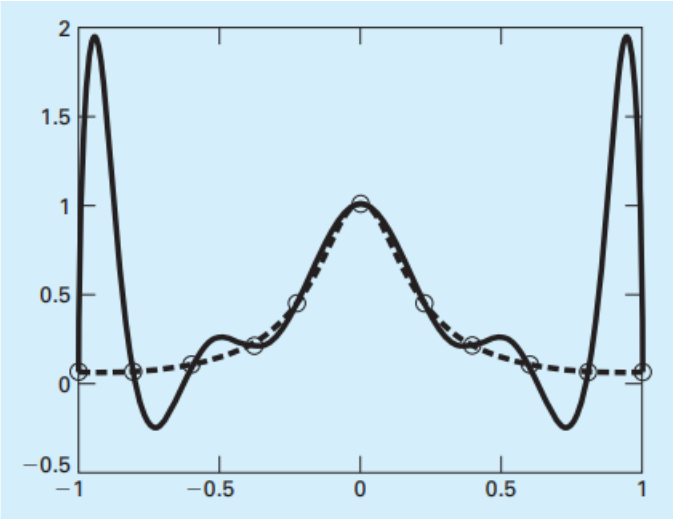


Figure 9.13: Comparison of Runge's function (dashed line) with a tenth-order polynomial fit to 11 points sampled from the function.

```
>> p = polyfit(x,y,10);
>> y10 = polyval(p,xx);
>> plot(x,y,'o',xx,y10,xx,yr,'--')
```

As in Fig. 17.13, 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

17.1 The following data come from a table that was measured with high precision. Use the best numerical method (for this type of problem) to determine y at x = 3.5. Note that a polynomial will yield an exact value. Your solution should prove that your result is exact.

x	0	1.8	5	6	8.2	9.2	12
y	26	16.415	5.375	3.5	2.015	2.54	8

17.2 Use Newton's interpolating polynomial to determine y at x = 3.5 to the best possible accuracy. Compute the finite divided differences as in Fig. 17.5, and order your points to attain optimal accuracy and convergence. That is, the points should be centered around and as close as possible to the unknown.

x	0	1	2.5	3	4.5	5	6
y	2	5.4375	7.3516	7.5625	8.4453	9.1875	12

17.3 Use Newton's interpolating polynomial to determine y at x = 8 to the best possible accuracy. Compute the finite divided differences as in Fig. 17.5, and order your points to attain optimal accuracy and convergence. That is, the points should be centered around and as close as possible to the unknown.

x	0	1	2	5.5	11	13	16	18
y	0.5	3.134	5.3	9.9	10.2	9.35	7.2	6.2

17.4 Given the data

x	1	2	2.5	3	4	5
$f(x)$	0	5	6.5	7	3	1

(a) 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. That is, the points should be centered around and as close as possible to the unknown.
(b) Repeat (a) but use the Lagrange polynomial.
17.5 Given the data

x	1	2	3	5	6
$f(x)$	7	4	5.5	40	82

Calculate f (4) using Newton's interpolating polynomials of order 1 through 4. Choose your base points to attain good accuracy. That is, the points should be centered around and as close as possible to the unknown. What do your results indicate regarding the order of the polynomial used to generate the data in the table?
17.6 Repeat Prob. 17.5 using the Lagrange polynomial of order 1 through 3.
17.7 Table P15.5 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 ÂřC and c = 10 g/L.
(b) Use linear interpolation to determine the oxygen concentration for T = 12 ÂřC and c = 15 g/L.
(c) Repeat (b) but use quadratic interpolation.

17.8 Employ inverse interpolation using a cubic interpolating polynomial and bisection to determine the value of x that corresponds to f (x) = 1.7 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

17.9 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.
17.10 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.

$v, \mathbf{m^3/kg}$	0.10377	0.11144	0.12547
$s, \mathbf{kJ/(kgK)}$	0.4147	0.5453	6.7664

17.11 The following data for the density of nitrogen gas versus temperature come 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.

$T, \mathbf{K}$	200	250	300	350	400	450
Density, $\mathbf{kg/m^3}$	1.708	1.367	1.139	0.967	0.854	0.759

17.12 Ohm's law states that the voltage drop V across an ideal resistor is linearly proportional to the current i flowing through the resister as in V = i R, 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	−637	−96.5	−20.5	20.5	96.5	637

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.
17.13 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	1.8	2.0	2.2	2.4	2.6
$J_1(x)$	0.5815	0.5767	0.5560	0.5202	0.4708

Estimate $J_1(2.1)$ using third- and fourth-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 *bessel j*.
17.14 4 Repeat Example 17.6 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?
17.15 The specific volume of a superheated steam is listed in steam tables for various temperatures. For example, at a pressure of 3000lb/in^2, absolute:

$T, ^\circ\mathbf{C}$	370	382	394	406	418
$v, \mathbf{Lt^3/kg}$	5.9313	7.5838	8.8428	9.796	10.5311

Determine v at T = 750 ÂřF.

17.16 The vertical stress σ_z under the corner of a rectangular area subjected to a uniform load of intensity q is given by the solution of Boussinesq's equation:

$$\sigma = \frac{q}{4\pi}\left[\frac{2mn\sqrt{m^2+n^2+1}}{m^2+n^2+1+m^2n^2}\,\frac{m^2+n^2+2}{m^2+n^2+1} + \sin^{-1}\left(\frac{2mn\sqrt{m^2+n^2+1}}{m^2+n^2+1+m^2n^2}\right)\right]$$

Because this equation is inconvenient to solve manually, it has been reformulated as

$$\sigma_z = q f_z(m,n)$$

where $f_z(m,n)$ is called the influence value, and m and n are dimensionless ratios, with $m = a/z$ and $n = b/z$ and a and b are defined in Fig. P17.16. The influence value is then tabulated, a portion of which is given in Table P17.16. If $a = 4.6$ and $b = 14$, use a third-order interpolating polynomial to compute σ_z at a depth 10 m below the corner of a rectangular footing that is subject to a total load of 100t

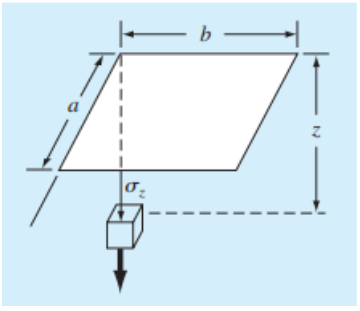


Figure 9.14:

(metric tons). Express your answer in tonnes per square meter. Note that q is equal to the load per area.

m	$n=1.2$	$n=1.4$	$n=1.6$
0.1	0.02926	0.03007	0.03058
0.2	0.05733	0.05894	0.05994
0.3	0.08323	0.08561	0.08709
0.4	0.10631	0.10941	0.11135
0.5	0.12626	0.13003	0.13241
0.6	0.14309	0.14749	0.15027
0.7	0.15703	0.16199	0.16515
0.8	0.16843	0.17389	0.17739

TABLE P17.16

17.17 You measure the voltage drop V across a resistor for a number of different values of current i. The results are

i	0.25	0.75	1.25	1.5	2.0
V	−0.45	−0.6	0.70	1.88	6.0

Use first- through fourth-order polynomial interpolation to estimate the voltage drop for i = 1.15. Interpret your results.

17.18 The current in a wire is measured with great precision as a function of time:

t	0	0.1250	0.2500	0.3750	0.5000
i	0	6.24	7.75	4.85	0.0000

Determine i at t = 0.23.

17.19 The acceleration due to gravity at an altitude y above the surface of the earth is given by

y, m	0	30,000	60,000	90,000	120,000
g, m/s^2	9.8100	9.7487	9.6879	9.6278	9.5682

Compute g at y = 55,000 m.

17.20 Temperatures are measured at various points on a heated plate (Table P17.20). Estimate the temperature at (a) x = 4, y = 3.2, and (b) x = 4.3, y = 2.7.

	$x=0$	$x=2$	$x=4$	$x=6$	$x=8$
$y=0$	100.00	90.00	80.00	70.00	60.00
$y=2$	85.00	64.49	53.50	48.15	50.00
$y=4$	70.00	48.90	38.43	35.03	40.00
$y=6$	55.00	38.78	30.39	27.07	30.00
$y=8$	40.00	35.00	30.00	25.00	20.00

TABLE P17.20 Temperatures (ÂřC) at various points on a square heated plate.

17.21

Use the portion of the given steam table for superheated H_2O at 200 MPa to (a) find the corresponding entropy s for a specific volume v of m^3/kg with linear interpolation, (b) find the same corresponding entropy using quadratic interpolation, and (c) find the volume corresponding to an entropy of 6.6 using inverse interpolation.

v (m^3/kg)	0.10377	0.11144	0.12540
s (kJ/kg · K)	6.4147	6.5453	6.7664

Chapter 10

Splines and Piecewise Interpolation

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
- Understanding how multidimensional interpolation is implemented with MATLAB.

10.1.INTRODUCTION TO SPLINES

In Chap. 17 (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

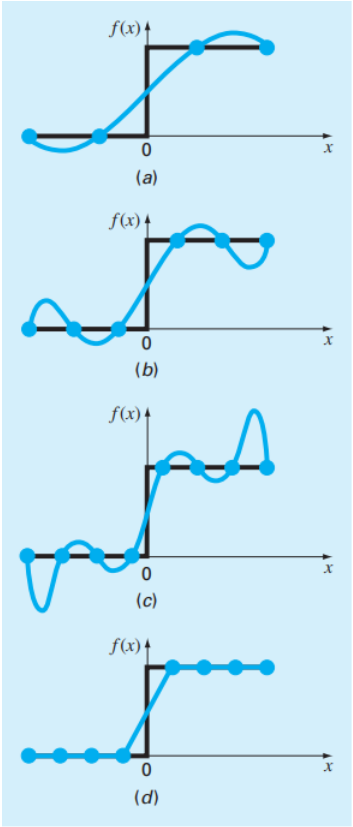


Figure 10.1: A visual representation of a situation where splines are superior to higher-order interpolating polynomials. The function to be fit undergoes an abrupt increase at x = 0. Parts (a) through (c) indicate that the abrupt change induces oscillations in interpolating polynomials. In contrast, because it is limited to straight-line connections, a linear spline (d) provides a much more acceptable approximation.

adjacent cubic equations are visually smooth. On the surface, it would seem that the thirdorder approximation of the splines would be inferior to the seventh-order expression. You might wonder why a spline would ever be preferable. Figure 18.1 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. 18.1 is an extreme example of such a change and serves to illustrate the point. Figure 18.1a 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

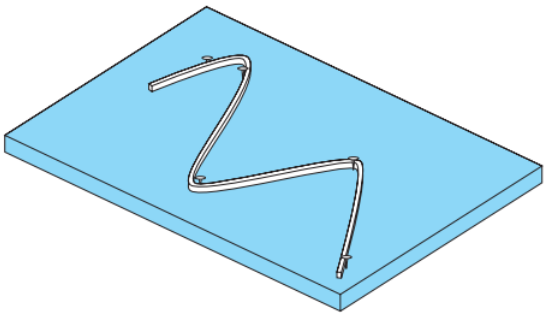


Figure 10.2: The drafting technique of using a spline to draw smooth curves through a series of points. Notice how, at the end points, the spline straightens out. This is called a âĂIJnaturalâĂİ spline.

minimum. As such, the spline usually provides a superior approximation of the behavior of functions that have local, abrupt changes. 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. 18.2 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 âĂIJcubic splineâĂİ 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.

10.2.LINEAR SPLINES

The notation used for splines is displayed in Fig. 18.3. For n data points (i = 1, 2,..., 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

$$s_i(x) = a_i + b_i(x - x_i) \tag{18.1}$$

where a_i is the intercept, which is defined as

$$a_i = f_i \tag{18.2}$$

and b_i is the slope of the straight line connecting the points:

$$b_i = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} \tag{18.3}$$

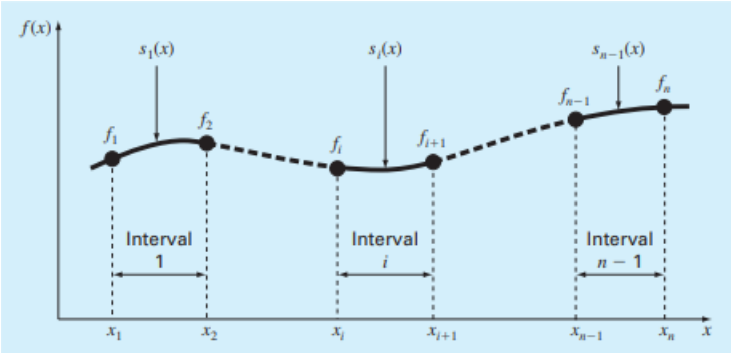


Figure 10.3: Notation used to derive splines. Notice that there are n - 1 intervals and n data points.

where f_i is shorthand for f (x_i). Substituting Eqs. (18.1) and (18.2) into Eq. (18.3) gives

$$s_i(x) = f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i}(x - x_i) \tag{18.4}$$

These equations can be used to evaluate the function at any point between x1 and xn 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. (18.4) indicates that the linear spline amounts to using Newton's first-order polynomial [Eq. (17.5)] to interpolate within each interval.

Example 10. 8. First-Order Splines *Problem Statement.* Fit the data in Table 18.1 with first-order splines. Evaluate the function at x = 5.

TABLE 18.1 Data to be fit with spline functions.

i	x_i	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. (18.4) to generate the linear spline functions. For example, for the second interval from x = 4.5 to x = 7, the function is

$$s_2(x) = 1.0 + \frac{2.5 - 1.0}{7.0 - 4.5}(x - 4.5)$$

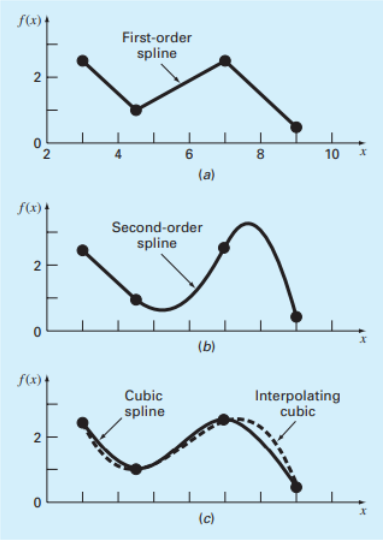


Figure 10.4: Spline fits of a set of four points. (a) Linear spline, (b) quadratic spline, and (c) cubic spline, with a cubic interpolating polynomial also plotted.

The equations for the other intervals can be computed, and the resulting first-order splines are plotted in Fig. 18.4a. 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. 18.4a 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.

10.2.1Table 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 17.1. 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. 17. 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 tableâĂŹs 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 xâĂŹs. 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 intervalâĂŹs 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 434 SPLINES AND PIECEWISE INTERPOLATION 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 = fix((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 âĂIJlowerâĂİ index iL = 1 and the last or âĂIJupperâĂİ 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 ÂřC based on the data from Table 17.1. 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:

$$f(350) = 0.616 + \frac{0.525 - 0.616}{400 - 300}(350 - 300) = 0.5705$$

10.3.QUADRATIC SPLINES

To ensure that the nth 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 âĂIJquadratic splinesâĂİ 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

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 \tag{18.5}$$

where the notation is as in Fig. 18.3. For n data points (i = 1, 2,..., n), there are n - 1 intervals and, consequently, 3(n - 1) unknown constants (the a's, b's, and c's) to evaluate. Therefore, 3(n - 1) 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

$$f_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 = f_{i+1} + b_{i+1}(x_{i+1} - x_{i+1}) + c_{i+1}(x_{i+1} - x_{i+1})^2$$

which simplifies to

$$a_i = f_i \tag{18.6}$$

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. (18.5):

$$s_i(x) = f_i + b_i(x - x_i) + c_i(x - x_i)^2$$

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

$$f_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 = f_{i+1} + b_{i+1}(x_{i+1} - x_{i+1}) + c_{i+1}(x_{i+1} - x_{i+1})^2 \tag{18.7}$$

This equation can be simplified mathematically by defining the width of the ith interval as

$$h_i = x_{i+1} - x_i$$

Thus, Eq. (18.7) simplifies to

$$f_i + b_i h_i + c_i h_i^2 = f_{i+1} \tag{18.8}$$

This equation can be written for the nodes, i = 1,..., n - 1. Since this amounts to n - 1 conditions, it means that there are 2(n - 1) - (n - 1) = n - 1 remaining conditions.

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 (18.5) 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

$$b_i + 2c_i h_i = b_{i+1} \tag{18.9}$$

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. (18.5) 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 10. 9. Quadratic Splines *Problem Statement.* Fit quadratic splines to the same data employed in Example 18.1 (Table 18.1). 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 (18.8) is written for i = 1 through 3 (with $c_1 = 0$) to give

$$\begin{aligned} f_1 + b_1 h_1 &= f_2 \\ f_2 + b_2 h_2 + c_2 h_2^2 &= f_3 \\ f_3 + b_3 h_3 + c_3 h_3^2 &= f_4 \end{aligned}$$

Continuity of derivatives, Eq. (18.9), creates an additional 3 - 1 = 2 conditions (again, recall that $c_1 = 0$):

$$\begin{aligned} b_1 &= b_2 \\ b_2 + 2c_2 h_2 &= b_3 \end{aligned}$$

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 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:

$$\begin{aligned} b_1 &= -1 \\ b_2 &= -1 \quad c_2 = 0.64 \\ b_3 &= 2.2 \quad c_3 = -1.6 \end{aligned}$$

These results, along with the values for the a 's (Eq. 18.6), can be substituted into the original quadratic equations to develop the following quadratic splines for each interval:

$$\begin{aligned} s_1(x) &= 2.5 - (x-3) \\ s_2(x) &= 1.0 - (x-4.5) + 0.64(x-4.5)^2 \\ s_3(x) &= 2.5 + 2.2(x-7.0) - 1.6(x-7.0)^2 \end{aligned}$$

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. 18.4b. 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.

10.4.CUBIC SPLINES

As stated at the beginning of the previous section, 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

$$s_i(x) = a_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3 \tag{18.10}$$

Thus, for n data points (i = 1, 2,..., n), there are n - 1 intervals and 4(n - 1) unknown coefficients to evaluate. Consequently, 4(n - 1) conditions are required for their evaluation. 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 âĂIJnaturalâĂİ spline. It is given this name because the drafting spline naturally behaves in this fashion (Fig. 18.2). 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 Section 18.4.2. 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.

10.4.1Derivation 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

$$a_i = f_i \tag{18.11}$$

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. (18.10):

$$s_i(x) = f_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3 \tag{18.12}$$

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

$$f_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = f_{i+1} \tag{18.13}$$

where

$$h_i = x_{i+1} - x_i$$

The first derivatives at the interior nodes must be equal. Equation (18.12) is differentiated to yield

$$s_i'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \tag{18.14}$$

The equivalence of the derivatives at an interior node, $i+1$ can therefore be written as

$$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1} \tag{18.15}$$

The second derivatives at the interior nodes must also be equal. Equation (18.14) can be differentiated to yield

$$s_i''(x) = 2c_i + 6d_i(x - x_i) \tag{18.16}$$

The equivalence of the second derivatives at an interior node, $i+1$ can therefore be written as

$$c_i + 3d_i h_i = c_i \tag{18.17}$$

Next, we can solve Eq. (18.17) for d_i :

$$d_i = \frac{c_{i+1} - c_i}{3h_i} \tag{18.18}$$

This can be substituted into Eq. (18.13) to give

$$f_i + b_i h_i + \frac{h_i^2}{3}(2c_i + c_{i+1}) = f_{i+1} \tag{18.19}$$

Equation (18.18) can also be substituted into Eq. (18.15) to give

$$b_{i+1} = b_i + h_i(c_i + c_{i+1}) \tag{18.20}$$

Equation (18.19) can be solved for

$$b_i = \frac{f_{i+1} - f_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1}) \tag{18.21}$$

The index of this equation can be reduced by 1 :

$$b_{i-1} = \frac{f_i - f_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{3}(2c_{i-1} + c_i) \tag{18.22}$$

The index of Eq. (18.20) can also be reduced by 1 :

$$b_i = b_{i-1} + h_{i-1}(c_{i-1} + c_i) \tag{18.23}$$

Equations (18.21) and (18.22) can be substituted into Eq. (18.23) and the result simplified to yield

$$h_{i-1}c_{i-1} + 2(h_{i-1} - h_i)c_i + h_i c_{i+1} = 3\frac{f_{i+1} - f_i}{h_i} - 3\frac{f_i - f_{i-1}}{h_{i-1}} \tag{18.24}$$

This equation can be made a little more concise by recognizing that the terms on the right-hand side are finite differences (recall Eq. 17.15):

$$f[x_i, x_j] = \frac{f_i - f_j}{x_i - x_j}$$

Therefore, Eq. (18.24) can be written as

$$h_{i-1}c_{i-1} + 2(h_{i-1} - h_i)c_i + h_i c_{i+1} = 3(f[x_{i+1}, x_i] - f[x_i, x_{i-1}]) \tag{18.25}$$

Equation (18.25) can be written for the interior knots, $i = 2, 3, \ldots, n-2$, which results in $n-3$ simultaneous tridiagonal equations with $n-1$ unknown coefficients, $c_1, c_2, \ldots, c_{n-1}$. Therefore, if we have two additional conditions, we can solve for the c 's. Once this is done, Eqs. (18.21) and (18.18) 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. 18.16) 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:

$$s''_{n-1}(x_n) = 0 = 2c_{n-1} + 6d_{n-1}h_{n-1} \tag{18.26}$$

Recalling Eq. (18.17), we can conveniently define an extraneous parameter c_n, in which case Eq. (18.26) 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

$$\begin{bmatrix} 1 & & & \\ h_1 & 2(h_1+h_2) & h_2 & \\ \ddots & \ddots & \ddots & \\ h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ & & & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 3(f[x_3,x_2] - f[x_2,x_1]) \\ \vdots \\ 3(f[x_n,x_{n-1}] - f[x_{n-1},x_{n-2}]) \\ 0 \end{bmatrix} \tag{18.27}$$

As shown, the system is tridiagonal and hence efficient to solve.

Example 10. 10. Natural Cubic Splines *Problem Statement.* Fit cubic splines to the same data used in Examples 18.1 and 18.2 (Table 18.1). Utilize the results to estimate the value at x = 5.

Solution. The first step is to employ Eq. (18.27) 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

$$\begin{array}{ll} f_1 = 2.5 & h_1 = 4.5 - 3.0 = 1.5 \\ f_2 = 1.0 & h_2 = 7.0 - 4.5 = 2.5 \\ f_3 = 2.5 & h_3 = 9.0 - 7.0 = 2.0 \\ f_4 = 0.5 & \end{array}$$

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:

$$\begin{array}{ll} c_1 = 0 & c_2 = 0.839543726 \\ c_3 = -0.766539924 & c_4 = 0 \end{array}$$

Equations (18.21) and (18.18) can be used to compute the b 's and d 's

$$\begin{array}{ll} b_1 = -1.419771863 & d_1 = 0.186565272 \\ b_2 = -0.160456274 & d_2 = -0.214144487 \\ b_3 = 0.022053232 & d_3 = 0.127756654 \end{array}$$

These results, along with the values for the a's [Eq. (18.11)], can be substituted into Eq. (18.10) to develop the following cubic splines for each interval:

$$\begin{aligned} s_1(x) =& 2.5 - 1.419771863(x-3) + 0.186565272(x-3)^3 \\ s_2(x) =& 1.0 - 0.160456274(x-4.5) + 0.839543726(x-4.5)^2 \\ & - 0.214144487(x-4.5)^3 \\ s_3(x) =& 2.5 + 0.022053232(x-7.0) - 0.766539924(x-7.0)^2 \\ & + 0.127756654(x-7.0)^3 \end{aligned}$$

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. 18.4c.

The results of Examples 18.1 through 18.3 are summarized in Fig. 18.4. 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. 18.4c. 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.

10.4.2End 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 âĂIJclampedâĂİ 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

- âĂIJNot-a-KnotâĂİ 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 âĂIJnot-a-knotâĂİ 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. 17.

These conditions can be readily applied by using Eq. (18.25) for the interior knots, i = 2, 3,..., n - 2, and using first (1) and last equations (n - 1) as written in Table 18.2. Figure 18.5 shows a comparison of the three end conditions as applied to fit the data from Table 18.1. 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 nota-knot exhibits more curvature.

TABLE 18.2 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	$h_2c_1 - (h_1+h_2)c_2 + h_1c_3 = 0$ $h_{n-1}c_{n-2} - (h_{n-2}+h_{n-1})c_{n-1} + h_{n-2}c_n = 0$

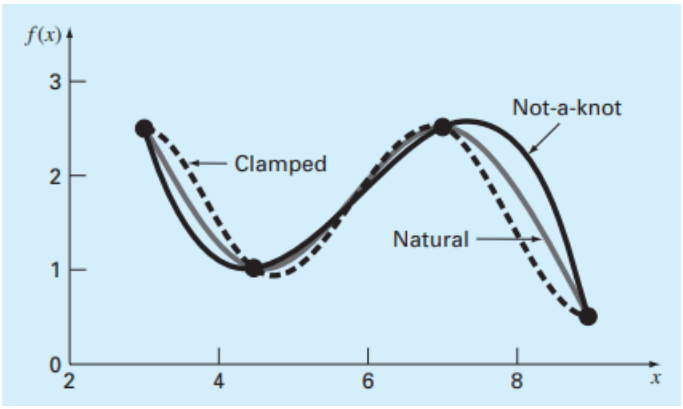


Figure 10.5: Comparison of the clamped (with zero first derivatives), not-a-knot, and natural splines for the data from Table 18.1.

10.5.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.

10.5.1MATLAB Function: spline

Cubic splines can be easily computed with the built-in MATLAB function, spline. It has the general syntax,

```
yy = spline(x, y, xx)
```

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 10. 11. Splines in MATLAB *Problem Statement.* RungeâĂŹs function is a notorious example of a function that cannot be fit well with polynomials (recall Example 17.7):

$$f(x) = \frac{1}{1+25x^2}$$

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. 18.6, 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

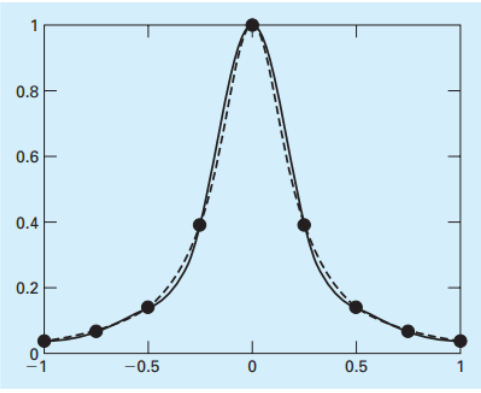


Figure 10.6: Comparison of Runge's function (dashed line) with a 9-point not-a-knot spline fit generated with MATLAB (solid line).

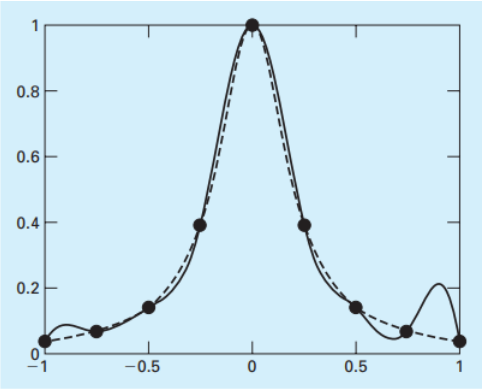


Figure 10.7: Comparison of Runge's function (dashed line) with a 9-point clamped end spline fit generated with MATLAB (solid line). Note that first derivatives of 1 and -4 are specified at the left and right boundaries, respectively.

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. 18.7, 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.

10.5.2MATLAB Function: interp1

The built-in function interp1 provides a handy means to implement a number of different types of piecewise one-dimensional interpolation. It has the general syntax

```
yi = interp1(x, y, xi, 'method')
```

where x and y = vectors containing values that are to be interpolated, yi = a vector containing the results of the interpolation as evaluated at the points in the vector xi, and 'method' = the desired method. The various methods are

- 'nearest'âĂŤnearest 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'âĂŤlinear interpolation. This method uses straight lines to connect the points.
- 'spline'âĂŤpiecewise cubic spline interpolation. This is identical to the spline function.
- 'pchip' and 'cubic'âĂŤpiecewise cubic Hermite interpolation.

If the 'method' argument is omitted, the default is linear interpolation. The pchip option (short for âĂIJpiecewise cubic Hermite interpolationâĂİ) 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 âĂIJshape preserving.âĂİ 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 10. 12. 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

t	0	20	40	56	68	80	84	96	104	110
v	0	20	20	38	80	80	100	100	125	125

Use MATLAB's interp1 function to fit these 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. 18.8a) 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)
```

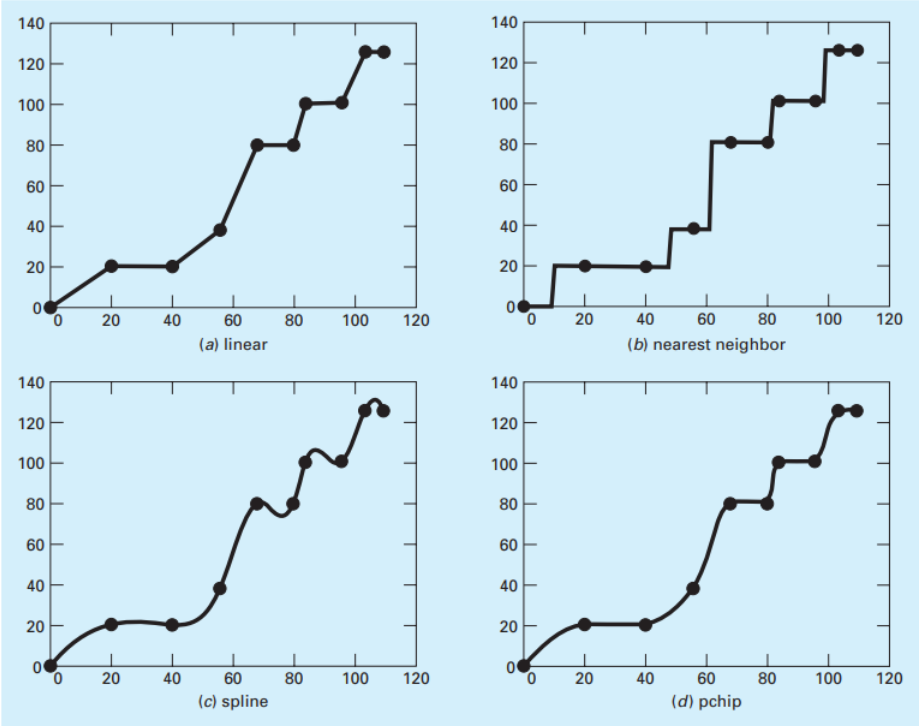


Figure 10.8: Use of several options of the interp1 function to perform piecewise polynomial interpolation on a velocity time series for an automobile.

As in Fig. 18.8b, 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. 18.8c) are quite smooth. However, severe overshoot occurs at several locations. This makes it appear that the automobile decelerated several times during the experiment.

(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. 18.8d) 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.

10.6.MULTIDIMENSIONAL INTERPOLATION

The interpolation methods for one-dimensional problems can be extended to multidimensional interpolation. In this section, we will describe the simplest case of two-dimensional interpolation in Cartesian coordinates. In addition, we will describe MATLAB's capabilities for multidimensional interpolation.

10.6.1Bilinear Interpolation

Two-dimensional interpolation deals with determining intermediate values for functions of two variables z = f (x_i, y_i). As depicted in Fig. 18.9, we have values at four points: f (x_1, y_1), f (x_2, y1), f (x_1, y_2), and f (x_2, y_2). We want to

interpolate between these points

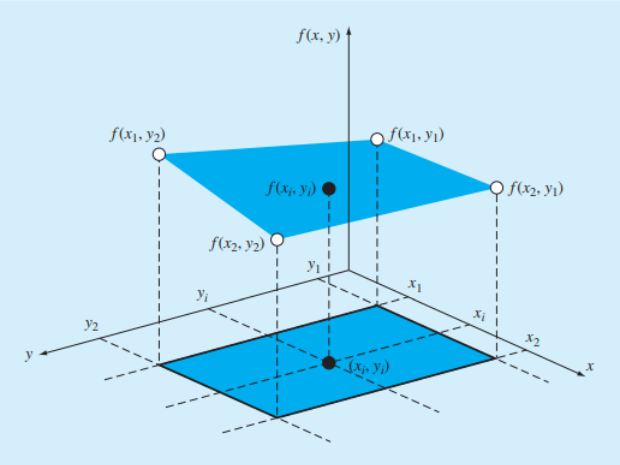


Figure 10.9: Graphical depiction of two-dimensional bilinear interpolation where an intermediate value (filled circle) is estimated based on four given values (open circles).

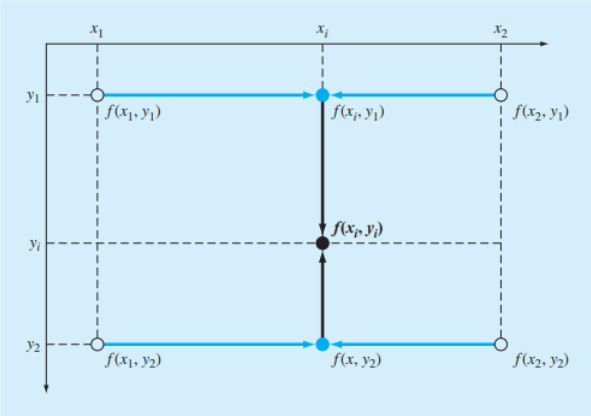


Figure 10.10: Two-dimensional bilinear interpolation can be implemented by first applying one-dimensional linear interpolation along the x dimension to determine values at x_i . These values can then be used to linearly interpolate along the y dimension to yield the final result at x_i , y_i ...

to estimate the value at an intermediate point f (x_i, y_i). If we use a linear function, the result is a plane connecting the points as in Fig. 18.9. Such functions are called bilinear. A simple approach for developing the bilinear function is depicted in Fig. 18.10. First, we can hold the y value fixed and apply one-dimensional linear interpolation in the x direction. Using the Lagrange form, the result at (x_i, y_i) is

$$f(x_i,y_1) = \frac{x_i - x_2}{x_1 - x_2} f(x_1,y_1) + \frac{x_i - x_1}{x_2 - x_1} f(x_2,y_1) \tag{18.29}$$

and at (x_i, y_2) is

$$f(x_i,y_2) = \frac{x_i - x_2}{x_1 - x_2} f(x_1,y_2) + \frac{x_i - x_1}{x_2 - x_1} f(x_2,y_2) \tag{18.30}$$

These points can then be used to linearly interpolate along the y dimension to yield the final result:

$$f(x_i,y_i) = \frac{y_i - y_2}{y_1 - y_2} f(x_i,y_1) + \frac{y_i - y_1}{y_2 - y_1} f(x_i,y_2) \tag{18.31}$$

A single equation can be developed by substituting Eqs. (18.29) and (18.30) into Eq. (18.31) to give

$$\begin{aligned} f(x_i,y_i) = &\frac{x_i - x_2}{x_1 - x_2}\frac{y_i - y_2}{y_1 - y_2} f(x_1,y_1) + \frac{x_i - x_1}{x_2 - x_1}\frac{y_i - y_2}{y_1 - y_2} f(x_2,y_1) \\ &+ \frac{x_i - x_2}{x_1 - x_2}\frac{y_i - y_1}{y_2 - y_1} f(x_1,y_2) + \frac{x_i - x_1}{x_2 - x_1}\frac{y_i - y_1}{y_2 - y_1} f(x_2,y_2) \end{aligned} \tag{18.32}$$

Example 10. 13. Bilinear Interpolation *Problem Statement.*Suppose you have measured temperatures at a number of coordinates on the surface of a rectangular heated plate:

$$T(2,1) = 60 \quad T(9,1) = 57.5$$
$$T(2,6) = 55 \quad T(9,6) = 70$$

Use bilinear interpolation to estimate the temperature at $x_i = 5.25$ and $y_i = 4.8$. **Solution.** Substituting these values into Eq. (18.32) gives

$$f(5.25,4.8) = \frac{5.25-9}{2-9}\frac{4.8-6}{1-6}60 + \frac{5.25-2}{9-2}\frac{4.8-6}{1-6}57.5 + \frac{5.25-9}{2-9}\frac{4.8-1}{6-1}55 + \frac{5.25-2}{9-2}\frac{4.8-1}{6-1}70 = 61.2143$$

10.6.2.Multidimensional Interpolation in MATLAB

MATLAB has two built-in functions for two- and three-dimensional piecewise interpolation: interp2 and interp3. As you might expect from their names, these functions operate in a similar fashion to interp1 (Section 18.5.2). For example, a simple representation of the syntax of interp2 is

```
zi = interp2(x, y, z, xi, yi, 'method')
```

where x and y = matrices containing the coordinates of the points at which the values in the matrix z are given, zi = a matrix containing the results of the interpolation as evaluated at the points in the matrices xi and yi, and method = the desired method. Note that the methods are identical to those used by interp1; that is, linear, nearest, spline, and cubic.
As with interp1, if the method argument is omitted, the default is linear interpolation. For example, interp2 can be used to make the same evaluation as in Example 18.6 as

```
>> x=[2 9];
>> y=[1 6];
>> z=[60 57.5;55 70];
>> interp2(x,y,z,5.25,4.8)
ans =
61.2143
```

10.7.CASE STUDY: HEAT TRANSFER

Background. Lakes in the temperate zone can become thermally stratified during the summer. As depicted in Fig. 18.11, warm, buoyant water near the surface overlies colder, denser bottom water. Such stratification effectively divides the lake vertically into two layers: the epilimnion and the hypolimnion, separated by a plane called the thermocline.

Thermal stratification has great significance for environmental engineers and scientists studying such systems. In particular, the thermocline greatly diminishes mixing between the two layers. As a result, decomposition of organic matter can lead to severe depletion of oxygen in the isolated bottom waters.

The location of the thermocline can be defined as the inflection point of the temperaturedepth curve-that is, the point at which $d^2T/dz^2 = 0$. It is also the point at which the absolute value of the first derivative or gradient is a maximum.

The temperature gradient is important in its own right because it can be used in conjunction with Fourier's law to determine the heat flux across the thermocline:

$$J = -D\rho C\frac{dT}{dz} \tag{18.33}$$

where J = heat flux $\left[\text{cal}/\left(\text{cm}^2 \cdot \text{s}\right)\right]$, α = an eddy diffusion coefficient $\left(\text{cm}^2/\text{s}\right)$, ρ = density $\left(\cong 1\ \text{g/cm}^3\right)$, and C = specific heat $[\cong 1\text{cal}/(\text{g} \cdot \text{C})]$.

In this case study, natural cubic splines are employed to determine the thermocline depth and temperature gradient for Platte Lake, Michigan (Table 18.3). The latter is also used to determine the heat flux for the case where $\alpha = 0.01\ \text{cm}^2/\text{s}$.

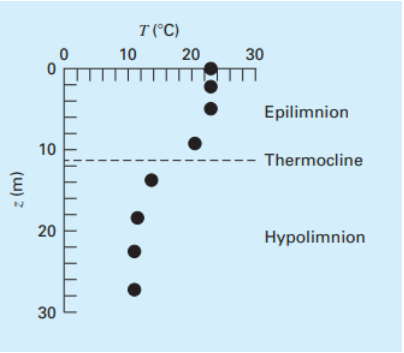


Figure 10.11: Temperature versus depth during summer for Platte Lake, Michigan.

Solution. As just described, we want to use natural spline end conditions to perform this analysis. Unfortunately, because it uses not-a-knot end conditions, the built-in MATLAB spline function does not meet our needs. Further, the spline function does not return the first and second derivatives we require for our analysis. However, it is not difficult to develop our own M-file to implement a natural spline and return the derivatives. Such a code is shown in Fig. 18.12. After some preliminary error trapping, we set up and solve Eq. (18.27) for the second-order coefficients (c). Notice how

Figure 10.12: M-file to determine intermediate values and derivatives with a natural spline. Note that the diff function employed for error trapping is described in Section 21.7.1.

```
function [yy,dy,d2] = natspline(x,y,xx)
% natspline: natural spline with differentiation
% [yy,dy,d2] = natspline(x,y,xx): uses a natural cubic spline
% interpolation to find yy, the values of the underlying function
% y at the points in the vector xx. The vector x specifies the
% points at which the data y is given.
% input:
% x = vector of independent variables
% y = vector of dependent variables
% xx = vector of desired values of dependent variables
% output:
% yy = interpolated values at xx
% dy = first derivatives at xx
% d2 = second derivatives at xx
n = length(x);
if length(y)~=n, error('x and y must be same length'); end
if any(diff(x)<=0),error('x not strictly ascending'),end
m = length(xx);
b = zeros(n,n);
aa(1,1) = 1; aa(n,n) = 1; %set up Eq. 18.27
bb(1)=0; bb(n)=0;
for i = 2:n-1
aa(i,i-1) = h(x, i - 1);
aa(i,i) = 2 * (h(x, i - 1) + h(x, i));
aa(i,i+1) = h(x, i);
bb(i) = 3 * (fd(i + 1, i, x, y) - fd(i, i - 1, x, y));
end
c=aa\bb'; %solve for c coefficients
for i = 1:n - 1 %solve for a, b and d coefficients
a(i) = y(i);
b(i) = fd(i + 1, i, x, y) - h(x, i) / 3 * (2 * c(i) + c(i + 1));
d(i) = (c(i + 1) - c(i)) / 3 / h(x, i);
end
for i = 1:m %perform interpolations at desired values
[yy(i),dy(i),d2(i)] = SplineInterp(x, n, a, b, c, d, xx(i));
end
end
function hh = h(x, i)
hh = x(i + 1) - x(i);
end
function fdd = fd(i, j, x, y)
fdd = (y(i) - y(j)) / (x(i) - x(j));
end
function [yyy,dyy,d2y]=SplineInterp(x, n, a, b, c, d, xi)
for ii = 1:n - 1
if xi >= x(ii) - 0.000001 & xi <= x(ii + 1) + 0.000001
yyy=a(ii)+b(ii)*(xi-x(ii))+c(ii)*(xi-x(ii))^2+d(ii)...
*(xi-x(ii))^3;
dyy=b(ii)+2*c(ii)*(xi-x(ii))+3*d(ii)*(xi-x(ii))^2;
d2y=2*c(ii)+6*d(ii)*(xi-x(ii));
break
end
end
end
```

we use two subfunctions, h and fd, to compute the required finite differences. Once Eq. (18.27) is set up, we solve for the c's with back division. A loop is then employed to generate the other coefficients (a, b, and d). At this point, we have all we need to generate intermediate values with the cubic equation:

$$f(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

We can also determine the first and second derivatives by differentiating this equation twice to give

$$f'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2$$
$$f''(x) = 2c_i + 6d_i(x - x_i)$$

As in Fig. 18.12, these equations can then be implemented in another subfunction, SplineInterp, to determine the values

and the derivatives at the desired intermediate values. Here is a script file that uses the natspline function to generate the spline and create plots of the results:

```
[TT,dT,dT2] = natspline(z,T,zz);
subplot(1,3,1),plot(T,z,'o',TT,zz)
title('(a) T'),legend('data','T')
set(gca,'YDir','reverse'),grid
subplot(1,3,2),plot(dT,zz)
title('(b) dT/dz')
set(gca,'YDir','reverse'),grid
subplot(1,3,3),plot(dT2,zz)
title('(c) d2T/dz2')
set(gca,'YDir','reverse'),grid
```

As in Fig. 18.13, the thermocline appears to be located at a depth of about 11.5 m. We can use root location (zero second derivative) or optimization methods (minimum first derivative) to refine this estimate. The result is that the thermocline is located at 11.35 m where the gradient is -1.61 ÂřC/m.

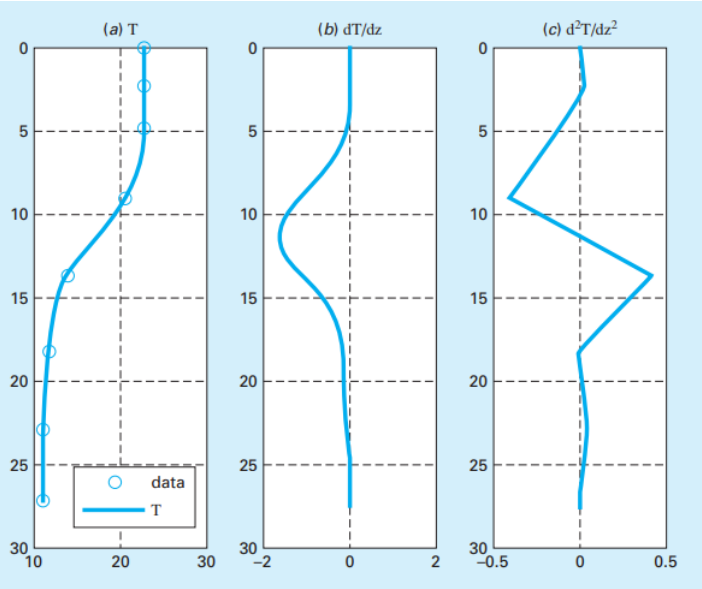


Figure 10.13: Plots of (a) temperature, (b) gradient, and (c) second derivative versus depth (m) generated with the cubic spline program. The thermocline is located at the inflection point of the temperaturedepth curve.

The gradient can be used to compute the heat flux across the thermocline with Eq. (18.33):

$$J = -0.01\frac{\text{cm}^2}{\text{s}} \times 1\frac{\text{g}}{\text{cm}^3} \times 1\frac{\text{cal}}{\text{g}\cdot^\circ\text{C}} \times \left(-1.61\frac{^\circ\text{C}}{\text{m}}\right) \times \frac{1\text{ m}}{100\text{ cm}} \times \frac{86{,}400\text{ s}}{\text{d}} = 13.9\frac{\text{cal}}{\text{cm}^2\cdot\text{d}}$$

The foregoing analysis demonstrates how spline interpolation can be used for engineering and scientific problem solving. However, it also is an example of numerical differentiation. As such, it illustrates how numerical approaches from different areas can be used in tandem for problem solving. We will be describing the topic of numerical differentiation in detail in Chap. 21.

10.8.PROBLEMS

18.1 Given the data

x	1	2	2.5	3	4	5
$f(x)$	1	5	7	8	2	1

Fit these 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.

18.2 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 $k = 0.01\text{cal}/(\text{s}\cdot\text{cm}\cdot^\circ\text{C})$ compute the flux across this interface.

18.3 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 :

x	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
x	0.6	0.7	0.8	0.9	1	
$f(x)$	11.692	12.382	17.846	21.703	16.000	

Fit these 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.

18.4 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.

x	0	100	200	400
$f(x)$	0	0.82436	1.00000	0.73576
x	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:

$$f(x) = \frac{x}{200} e^{-x/200+1}$$

18.5 The following data are sampled from the step function depicted in Fig. 18.1:

x	-1	-0.6	-0.2	0.2	0.6	1
$f(x)$	0	0	0	1	1	1

Fit these 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. *18.6 Develop an M-file to compute a cubic spline fit with natural end conditions. Test your code by using it to duplicate Example 18.3. 18.7 The following data were generated with the fifthorder polynomial:* $f(x) = 0.0185x^5 - 0.444x^4 + 3.9125x^3 - 15.456x^2 + 27.069x - 14.1$ *: (a) Fit these 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. 18.8 Bessel functions often arise in advanced engineering and scientific analyses such as the study of electric fields. These functions are usually not amenable to straightforward evaluation and, therefore, are often compiled in standard mathematical tables. For example,*

x	1.8	2	2.2	2.4	2.6
$J_1(x)$	0.5815	0.5767	0.556	0.5202	0.4708

Estimate J1(2.1), (a) using an interpolating polynomial and (b) using cubic splines. Note that the true value is 0.5683. *18.9 The following data define the sea-level concentration of dissolved oxygen for fresh water as a function of temperature:*

T,°C	0	8	16	24	32	40
o, mg/L	14.021	11.843	9.870	8.418	7.305	6.413

Use MATLAB to fit the data with (a) piecewise linear interpolation, (b) a fifth-order polynomial, and (c) a spline. Display the results graphically and use each approach to estimate o(27). Note that the exact result is 7.986 mg/L *18.10 (a) Use MATLAB to fit a cubic spline to the following data to determine y at x = 1.5:*

xC	0	2	4	7	10	12
y	20	20	12	7	6	6

(b) Repeat (a), but with zero first derivatives at the end knots.

18.11 Runge's function is written as

$$f(x) = \frac{1}{1+25x^2}$$

Generate five equidistantly spaced values of this function over the interval: $[-1, 1]$. Fit these data with (a) a fourthorder polynomial, (b) a linear spline, and (c) a cubic spline. Present your results graphically. *18.12 Use MATLAB to generate eight points from the function*

$$f(t) = \sin^2 t$$

from $t = 0$ to 2π. Fit these data using (a) cubic spline with not-a-knot end conditions, (b) cubic spline with derivative end conditions equal to the exact values calculated with differentiation, and (c) piecewise cubic hermite interpolation. Develop plots of each fit as well as plots of the absolute error (E_t = approximation − true) for each.

from t = 0 to 2π. Fit these data using (a) cubic spline with not-a-knot end conditions, (b) cubic spline with derivative end conditions equal to the exact values calculated with differentiation, and (c) piecewise cubic hermite interpolation. Develop plots of each fit as well as plots of the absolute error (E_t = approximation - true) for each.

18.13 The drag coefficient for spheres such as sporting balls is known to vary as a function of the Reynolds number Re, a dimensionless number that gives a measure of the ratio of inertial forces to viscous forces:

$$\mathrm{Re} = \frac{\rho V D}{\mu}$$

where ρ = the fluid's density $\left(\mathrm{kg/m^3}\right)$, V = its velocity (m/s), D = diameter (m), and μ = dynamic viscosity $\left(\mathrm{N \cdot s/m^2}\right)$. Although the relationship of drag to the Reynolds number is sometimes available in equation form, it is frequently tabulated. For example, the following table provides values for a smooth spherical ball:

$\mathrm{Re}\left(\times 10^{-4}\right)$	2	5.8	16.8	27.2	29.9	33.9	
C_D	0.52	0.52	0.52	0.5	0.49	0.44	
$\mathrm{Re}\left(\times 10^{-4}\right)$	36.3	40	46	60	100	200	400
C_D	0.18	0.074	0.067	0.08	0.12	0.16	0.19

(a) Develop a MATLAB function that employs the spline function to return a value of C_D as a function of the Reynolds number. The first line of the function should be

```
function CDout = Drag(ReCD, ReIn)
```

where ReCD = a 2-row matrix containing the table, ReIn = the Reynolds number at which you want to estimate the drag, and CDout = the corresponding drag coefficient.

(b) Write a script that uses the function developed in part (a) to generate a labeled plot of the drag force versus velocity (recall Sec. 1.4). Use the following parameter values for the script: $D = 22$ cm, $\rho = 1.3$ kg/m^3, and $\mu = 1.78 \times 10^{-5}$ Pa $\cdot$ s. Employ a range of velocities from 4 to 40 m/s for your plot. 18.14 The following function describes the temperature distribution on a rectangular plate for the range $-2 \leq x \leq 0$ and $0 \leq y \leq 3$

$$T = 2 + x - y + 2x^2 + 2xy + y^2$$

Develop a script to: (a) Generate a meshplot of this function using the MATLAB function surfc. Employ the linspace function with default spacing (i.e., 100 interior points) to generate the x and y values. (b) Use the MATLAB function interp2 with the default interpolation option ('linear') to compute the temperature at x = -1.63 and y = 1.627. Determine the percent relative error of your result. (c) Repeat (b), but with 'spline'. Note: for parts (b) and (c), employ the linspace function with 9 interior points.

Chapter 11

Numerical Integration Formulas

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to splines. 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

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) \tag{19.1}$$

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:

$$z(t) = \int_0^t v(t)dt \tag{19.2}$$

Substituting Eq. (19.1) into Eq. (19.2) gives

$$z(t) = \int_0^t \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right) dt \tag{19.3}$$

Thus, integration provides the means to determine the distance from the velocity. Calculus can be used to solve Eq. (19.3) for

$$z(t) = \frac{m}{c_d} \ln\left[\cosh\left(\sqrt{\frac{gc_d}{m}} t\right)\right] \tag{19.4}$$

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 19 and 20 will introduce you to some of these methods.

11.1.INTRODUCTION AND BACKGROUND

11.1.1.What Is Integration?

According to the dictionary definition, to integrate means âĂIJto bring together, as parts, into a whole; to unite; to indicate the total amount. . . .âĂİ Mathematically, definite integration is represented by

$$I = \int_a^b f(x)dx \tag{19.5}$$

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. (19.5) 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 19.1 represents a graphical manifestation of the concept. For functions lying above the x axis, the integral expressed by Eq. (19.5) 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.

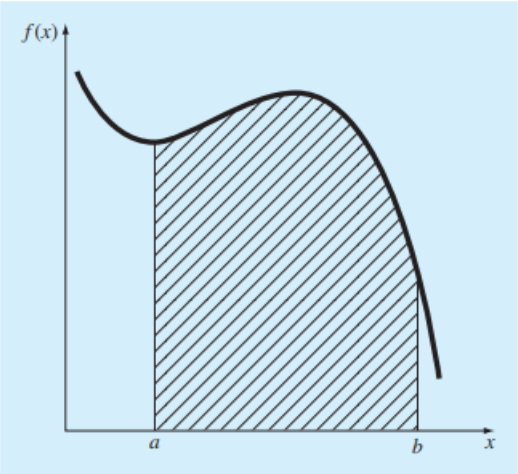


Figure 11.1: Graphical representation of the integral of f (x) between the limits x = a to b. The integral is equivalent to the area under the curve.

11.2.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 19.2 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 n discrete data points can be calculated by [Eq. (14.2)].

$$\text{Mean} = \frac{\sum_{i=1}^{n} y_i}{n} \tag{19.6}$$

where y_i are individual measurements. The determination of the mean of discrete points is depicted in Fig. 19.3a.

In contrast, suppose that y is a continuous function of an independent variable x, as depicted in Fig. 19.3b. For this case, there are an infinite number of values between a and b. Just as Eq. (19.6) 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

$$\text{Mean} = \frac{\int_a^b f(x)dx}{b-a} \tag{19.7}$$

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.

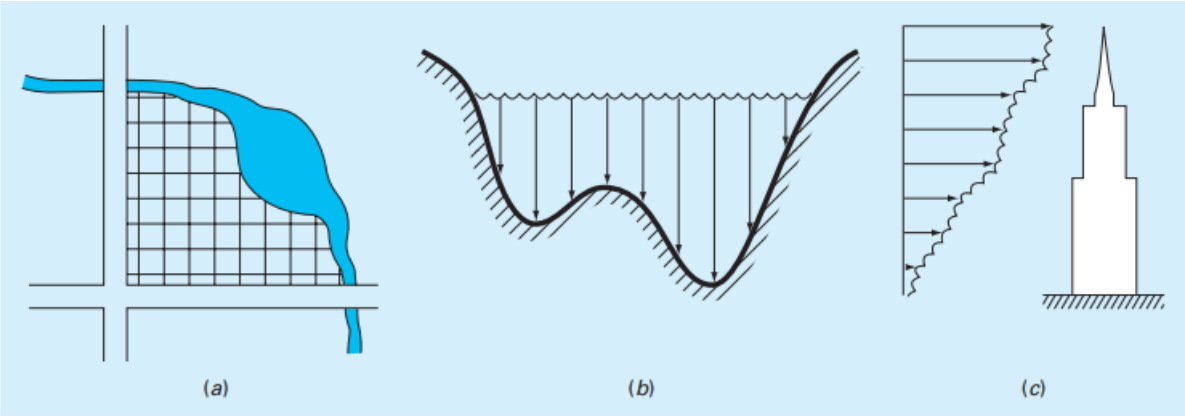


Figure 11.2: Examples of how integration is used to evaluate areas in engineering and scientific applications. (a) A surveyor might need to know the area of a field bounded by a meandering stream and two roads. (b) A hydrologist might need to know the cross-sectional area of a river. (c) A structural engineer might need to determine the net force due to a nonuniform wind blowing against the side of a skyscraper.

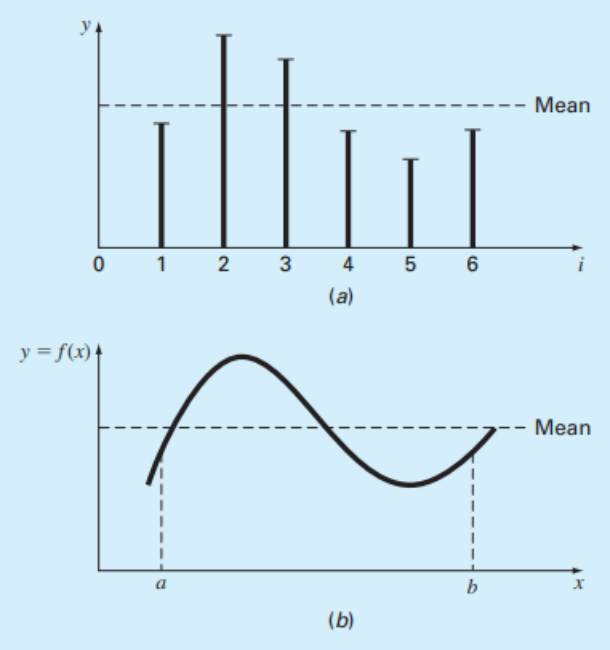


Figure 11.3: An illustration of the mean for (a) discrete and (b) continuous data

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:

$$\text{Mass} = \iiint c(x,y,z)dxdydz$$

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

$$\text{Flux} = \iint_A \text{ux}\, dA$$

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.

11.3.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:

$$I = \int_a^b f(x)dx \cong \int_a^b f_n(x)dx \tag{19.8}$$

where $f_n(x) =$ a polynomial of the form

$$f_n(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n \tag{19.9}$$

where n is the order of the polynomial. For example, in Fig. 19.4a, a first-order polynomial (a straight line) is used as an approximation. In Fig. 19.4b, a parabola is employed for the same purpose.
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. 19.5, three

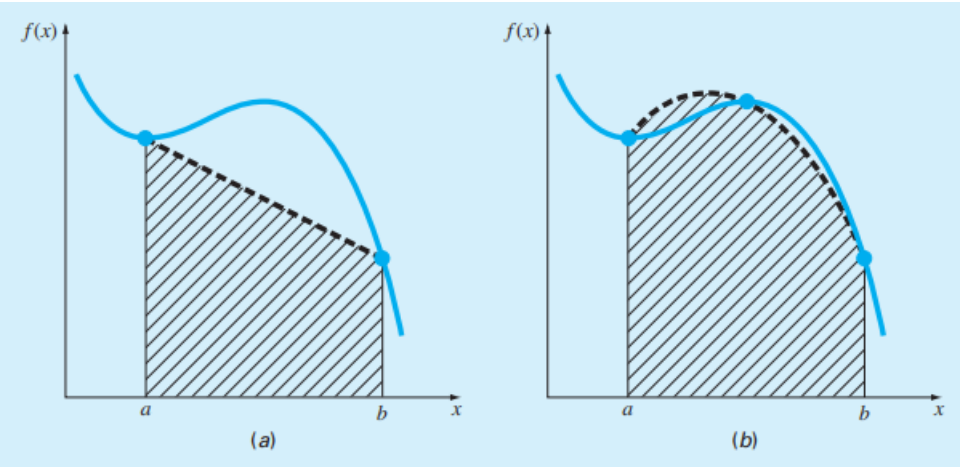


Figure 11.4: The approximation of an integral by the area under (a) a straight line and (b) a parabola.

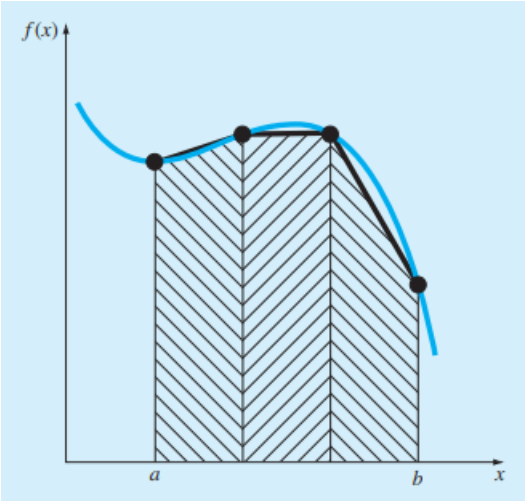


Figure 11.5: The approximation of an integral by the area under three straight-line segments

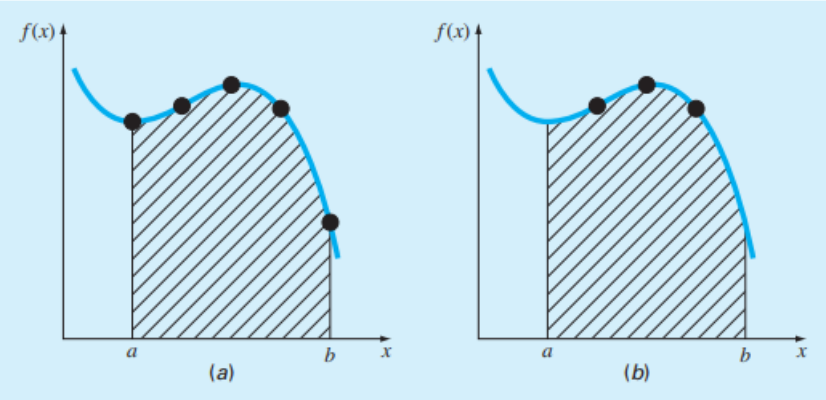


Figure 11.6: The difference between (a) closed and (b) open integration formulas

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. 19.6a). The open forms have integration limits that extend beyond the range of the data (Fig. 19.6b). This chapter emphasizes the closed forms. However, material on open Newton-Cotes formulas is briefly introduced in Section 19.7.

11.4.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. (19.8) is first-order:

$$I = \int_a^b \left[f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \right] dx \tag{19.10}$$

The result of the integration is

$$I = (b - a)\frac{f(a) + f(b)}{2} \tag{19.11}$$

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. 19.7. 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

$$I = \text{ width } \times \text{ average height} \tag{19.12}$$

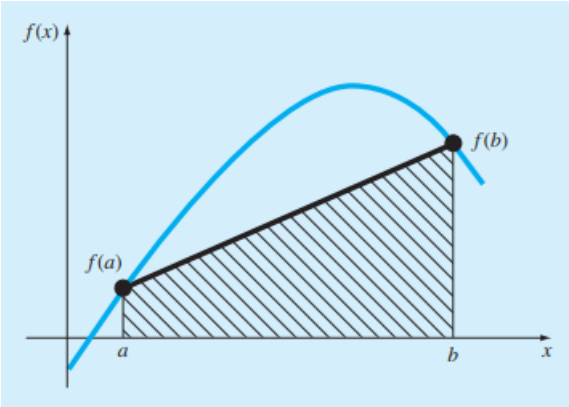


Figure 11.7: Graphical depiction of the trapezoidal rule

or

$$I = (b-a) \times \text{ average height} \tag{19.13}$$

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. (19.13). That is, they differ only with respect to the formulation of the average height.

11.4.1Error 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. 19.8). An estimate for the local truncation error of a single application of the trapezoidal rule is

$$E_t = -\frac{1}{12} f''(\xi)(b-a)^3 \tag{19.14}$$

where ξ lies somewhere in the interval from a to b. Equation (19.14) 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 11. 14. Single Application of the Trapezoidal Rule *Problem Statement.* Use Eq. (19.11) 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.

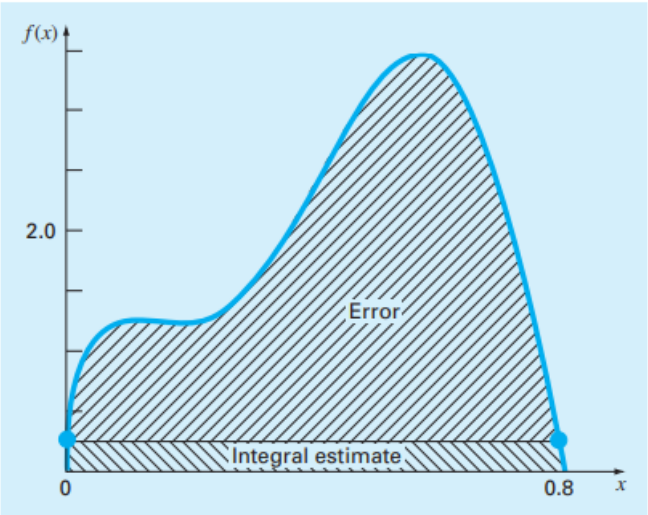


Figure 11.8: Graphical depiction of the use of a single application of the trapezoidal rule to approximate the integral of $f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$ from $x = 0$ to 0.8.

Solution. The function values f (0) = 0.2 and f (0.8) = 0.232 can be substituted into Eq. (19.11) 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. 19.8. 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. (19.7)]

$$\bar{f}''(x) = \frac{\int_0^{0.8} \left(-400 + 4,050x - 10,800x^2 + 8,000x^3\right) dx}{0.8 - 0} = -60$$

which can be substituted into Eq. (19.14) 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.

11.4.2The 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. 19.9). 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-segment, integration formulas

Figure 19.9 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,..., x_n). Consequently, there are n segments of equal width:

$$h = \frac{b-a}{n} \tag{19.15}$$

If a and b are designated as x_0 and x_n, respectively, the total integral can be represented as

$$I = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx$$

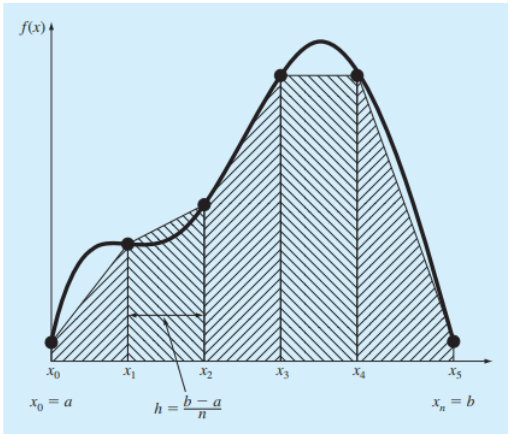


Figure 11.9: Composite trapezoidal rule.

Substituting the trapezoidal rule for each integral yields

$$I = h\frac{f(x_0) + f(x_1)}{2} + h\frac{f(x_1) + f(x_2)}{2} + \cdots + h\frac{f(x_{n-1}) + f(x_n)}{2} \tag{19.16}$$

or, grouping terms:

$$I = \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right] \tag{19.17}$$

or, using Eq. (19.15) to express Eq. (19.17) in the general form of Eq. (19.13):

$$I = \underbrace{(b-a)}_{\text{Width}} \underbrace{\frac{f(x_0)+2\sum_{i=1}^{n-1} f(x_i)+f(x_n)}{2n}}_{\text{Average height}} \tag{19.18}$$

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. (19.18), 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

$$E_t = -\frac{(b-a)^3}{12n^3}\sum_{i=1}^{n} f''(\xi_i) \tag{19.19}$$

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

$$\overline{f''} \cong \frac{\sum_{i=1}^{n} f''(\xi_i)}{n} \tag{19.20}$$

Therefore $\sum f''(\xi_i) \cong n\bar{f}''$ and Eq. (19.19) can be rewritten as

$$E_a = -\frac{(b-a)^3}{12n^2}\bar{f}'' \tag{19.21}$$

Thus, if the number of segments is doubled, the truncation error will be quartered. Note that Eq. (19.21) is an approximate error because of the approximate nature of Eq. (19.20).

Example 11. 15. 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. (19.21) 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 19.1.

The results of the previous example, along with three- through ten-segment applications of the trapezoidal rule, are summarized in Table 19.1. 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. (19.21)]. 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.

11.4.3MATLAB M-file: trap

A simple algorithm to implement the composite trapezoidal rule can be written as in Fig. 19.10. 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. (19.18). *TABLE 19.1* 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 0.8. The exact value is 1.640533.

n	h	I	$\varepsilon_t(\%)$
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

```
function I = trap(func,a,b,n,varargin)
% trap: composite trapezoidal rule quadrature
% I = trap(func,a,b,n,p1,p2,...):
% composite trapezoidal rule
% input:
% func = name of function to be integrated
% a, b = integration limits
% n = number of segments (default = 100)
% p1,p2,... = additional parameters used by func
% output:
% I = integral estimate
if nargin<3,error('at least 3 input arguments required'),end
if ~(b>a),error('upper bound must be greater than lower'),end
if nargin<4|isempty(n),n=100;end
x = a; h = (b - a)/n;
s=func(a,varargin{:});
for i = 1 : n-1
x = x + h;
s = s + 2*func(x,varargin{:});
end
s = s + func(b,varargin{:});
I = (b - a) * s/(2*n);
```

Figure 11.10: M-file to implement the composite trapezoidal rule

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. (19.3). For this example, assume the following parameter values: $g = 9.81$ m/s^2, $m =$ 68.1 kg, and $c_d = 0.25$ kg/m. Note that the exact value of the integral can be computed with Eq. (19.4) as 41.94805. The function to be integrated can be developed as an M-file or with an anonymous function,

```
>> v=@(t) sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t)
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.

11.5.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. 19.11a). 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. 19.11b). The formulas that result from taking the integrals under these polynomials are called Simpson's rules.

11.5.1Simpson's 1/3 Rule

Simpson's 1/3 rule corresponds to the case where the polynomial in Eq. (19.8) is secondorder:

$$I = \int_{x_0}^{x_2} \left[\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2) \right] dx$$

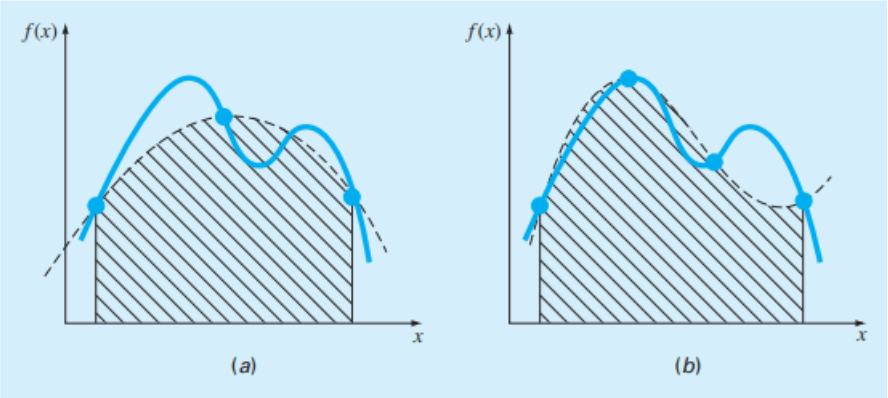


Figure 11.11: (a) Graphical depiction of Simpson's 1/3 rule: It consists of taking the area under a parabola connecting three points. (b) Graphical depiction of Simpson's 3/8 rule: It consists of taking the area under a cubic equation connecting four points.

where a and b are designated as x_0 and x_2, respectively. The result of the integration is

$$I = \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] \tag{19.22}$$

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. (19.22). Simpson's 1/3 rule can also be expressed using the format of Eq. (19.13):

$$I = (b-a)\frac{f(x_0) + 4f(x_1) + f(x_2)}{6} \tag{19.23}$$

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. (19.23), the middle point is weighted by twothirds 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 $h = (b-a)/2$:

$$E_t = -\frac{(b-a)^5}{2880}f^{(4)}(\xi) \tag{19.24}$$

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. (19.14) 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 thirdorder 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 11. 16. Single Application of Simpson's 1/3 Rule *Problem Statement.* Use Eq. (19.23) to integrate

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from $a = 0$ to $b = 0.8$. Employ Eq. (19.24) 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 19.1). 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 19.1, 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.

11.5.2The 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. 19.12). The total integral can be represented as

$$I = \int_{x_0}^{x_2} f(x)dx + \int_{x_2}^{x_4} f(x)dx + \cdots + \int_{x_{n-2}}^{x_n} f(x)dx \tag{19.25}$$

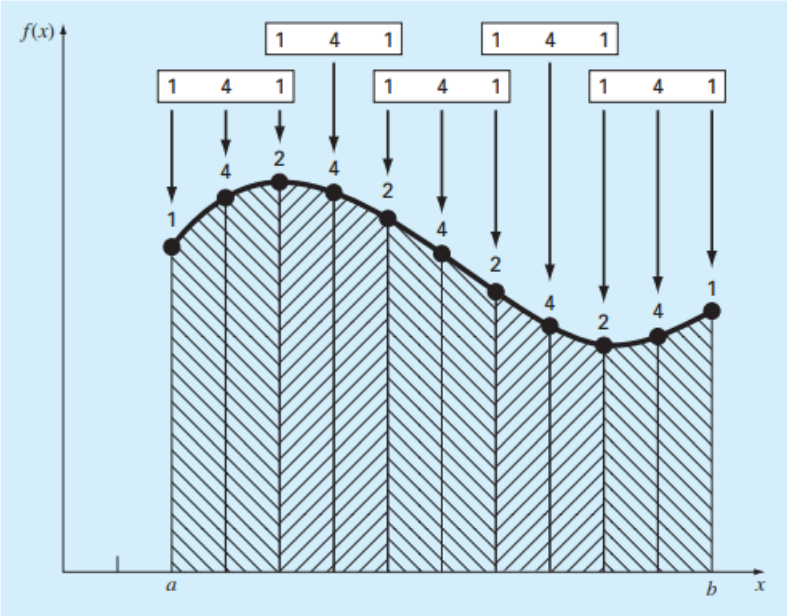


Figure 11.12: Composite Simpson's 1/3 rule. The relative weights are depicted above the function values. Note that the method can be employed only if the number of segments is even.

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. (19.15):

$$I = (b-a)\frac{f(x_0)+4\sum_{i=1,3,5}^{n-1} f(x_i)+2\sum_{j=2,4,6}^{n-2} f(x_j)+f(x_n)}{3n} \tag{19.26}$$

Notice that, as illustrated in Fig. 19.12, an even number of segments must be utilized to implement the method. In addition, the coefficients "4" and " 2 " in Eq. (19.26) might seem peculiar at first glance. However, they follow naturally from Simpson's 1/3 rule. As illustrated in Fig. 19.12, the odd points represent the middle term for each application and hence carry the weight of four from Eq. (19.23). 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

$$E_a = -\frac{(b-a)^5}{180n^4}\bar{f}^{(4)} \tag{19.27}$$

where $f^{(4)}$ is the average fourth derivative for the interval.

Example 11. 17. Composite Simpson's 1/3 Rule *Problem Statement.*Use Eq. (19.26) 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. (19.27) to estimate the error. Recall that the exact integral is 1.640533.
Solution.$n = 4(h = 0.2)$:

$$f(0) = 0.2 \qquad f(0.2) = 1.288$$
$$f(0.4) = 2.456 \quad f(0.6) = 3.464$$
$$f(0.8) = 0.232$$

From Eq. (19.26):

$$I = 0.8\frac{0.2+4(1.288+3.464)+2(2.456)+0.232}{12} = 1.623467$$
$$E_t = 1.640533 - 1.623467 = 0.017067 \quad \varepsilon_t = 1.04\%$$

The estimated error (Eq. 19.27) is

$$E_a = -\frac{(0.8)^5}{180(4)^4}(-2400) = 0.017067$$

which is exact (as was also the case for Example 19.3).

As in Example 19.4, 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 19.4.3, 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.

11.5.3.Simpson's 3/8 Rule

In a similar manner to the derivation of the trapezoidal and SimpsonâĂŹs 1/3 rule, a thirdorder Lagrange polynomial can be fit to four points and integrated to yield

$$I = \frac{3h}{8}\left[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\right]$$

where $h = (b-a)/3$. This equation is known as Simpsons $3/8$ 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. (19.13):

$$I = (b-a)\frac{f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)}{8} \tag{19.28}$$

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 $h = (b-a)/3$:

$$E_t = -\frac{(b-a)^5}{6480}f^{(4)}(\xi) \tag{19.29}$$

Because the denominator of Eq. (19.29) is larger than for Eq. (19.24), 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 19.4 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 19.2. 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

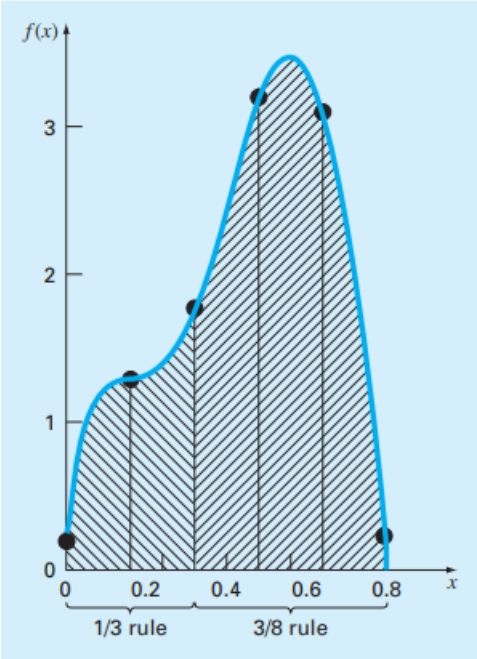


Figure 11.13: Illustration of how Simpson's 1/3 and 3/8 rules can be applied in tandem to handle multiple applications with odd numbers of intervals.

three (Fig. 19.13). In this way, we could obtain an estimate with third-order accuracy across the entire interval.

Example 11. 18. 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:

$$\begin{array}{ll} f(0) = 0.2 & f(0.2667) = 1.432724 \\ f(0.5333) = 3.487177 & f(0.8) = 0.232 \end{array}$$

Using Eq. (19.28):

$$I = 0.8\frac{0.2 + 3(1.432724 + 3.487177) + 0.232}{8} = 1.51970$$

(b) The data needed for a five-segment application ($h = 0.16$) are

$$\begin{array}{ll} f(0) = 0.2 & f(0.16) = 1.296919 \\ f(0.32) = 1.743393 & f(0.48) = 3.186015 \\ f(0.64) = 3.181929 & f(0.80) = 0.232 \end{array}$$

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:

$$I = 0.3803237 + 1.264754 = 1.645077$$

11.6. 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 19.2 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.

TABLE 19.2 Newton-Cotes closed integration formulas. The formulas are presented in the format of Eq. (19.13) 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 (n)	Points	Name	Formula	Trunction 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	$(b-a)\frac{f(x_0)+3f(x_1)+3f(x_2)+f(x_3)}{8}$	$-\left(3/80 \mid h^5 f^{(4)}(\xi)\right.$
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		$(b-a)\frac{19f(x_0)+75f(x_1)+50f(x_2)+50f(x_3)+75f(x_4)+19f(x_5)}{288}$	$-(275/12,096)h^7 f^{(6)}(\xi)$

However, it must also be stressed that, in engineering and science practice, the higherorder (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. 20, offer viable and attractive alternatives.

11.7.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 are often of this type. For these cases, one method is to apply the trapezoidal rule to each segment and sum the results:

$$I = h_1 \frac{f(x_0)+f(x_1)}{2} + h_2 \frac{f(x_1)+f(x_2)}{2} + \cdots + h_n \frac{f(x_{n-1})+f(x_n)}{2} \tag{19.30}$$

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. (19.16) and (19.30) is that the h 's in the former are constant.

Example 11. 19. Trapezoidal Rule with Unequal Segments *Problem Statement.*The information in Table 19.3 was generated using the same polynomial employed in Example 19.1. Use Eq. (19.30) to determine the integral for these data. Recall that the correct answer is 1.640533.

TABLE 19.3 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. (19.30) 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 $\varepsilon_t = 2.8\%$.

11.7.1MATLAB M-file: trapuneq

A simple algorithm to implement the trapezoidal rule for unequally spaced data can be written as in Fig. 19.14. 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.1 A loop is employed to generate the integral. Notice that we have modified the subscripts from those of Eq. (19.30) 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 19.6:

```
>> 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 19.6.

Figure 11.14: M-file to implement the trapezoidal rule for unequally spaced data.

```
function I = trapuneq(x,y)
% trapuneq: unequal spaced trapezoidal rule quadrature
% I = trapuneq(x,y):
% Applies the trapezoidal rule to determine the integral
% for n data points (x, y) where x and y must be of the
% same length and x must be monotonically ascending
% input:
% x = vector of independent variables
% y = vector of dependent variables
% output:
% I = integral estimate
if nargin<2,error('at least 2 input arguments required'),end
if any(diff(x)<0),error('x not monotonically ascending'),end
n = length(x);
if length(y)~=n,error('x and y must be same length'); end
s = 0;
for k = 1:n-1
s = s + (x(k+1)-x(k))*(y(k)+y(k+1))/2;
end
I = s;
```

The diff function is described in Section 21.7.1.

11.7.2.MATLAB Functions: trapz and cumtrapz

MATLAB has a built-in function that evaluates integrals for data in the same fashion as the M-file we just presented in Fig. 19.14. It has the general syntax

```
z = 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 19.3:

```
>> 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
```

In addition, MATLAB has another function, cumtrapz, that computes the cumulative integral. A simple representation of its syntax is

```
z = cumtrapz(x, y)
```

where the two vectors, x and y, hold the independent and dependent variables, respectively, and z = a vector whose elements z(k) hold the integral from x(1) to x(k).

Example 11. 20. Using Numerical Integration to Compute Distance from Velocity *Problem Statement.* As described at the beginning of this chapter, a nice application of integration is to compute the distance z(t) of an object based on its velocity v(t) as in (recall Eq. 19.2):

$$z(t) = \int_0^t v(t)dt$$

Suppose that we had measurements of velocity at a series of discrete unequally spaced times during free fall. Use Eq. (19.2) to synthetically generate such information for a 70-kg jumper with a drag coefficient of 0.275 kg/m. Incorporate some random error by rounding the velocities to the nearest integer. Then use cumtrapz to determine the distance fallen and compare the results to the analytical solution (Eq. 19.4). In addition, develop a plot of the analytical and computed distances along with velocity on the same graph.

Solution. Some unequally spaced times and rounded velocities can be generated as

```
>> format short g
>> t=[0 1 1.4 2 3 4.3 6 6.7 8];
>> g=9.81;m=70;cd=0.275;
>> v=round(sqrt(g*m/cd)*tanh(sqrt(g*cd/m)*t));
```

The distances can then be computed as

```
>> z=cumtrapz(t,v)
z=
   0  5  9.6  19.2  41.7  80.7  144.45  173.85  231.7
```

Thus, after 8 seconds, the jumper has fallen 231.7 m. This result is reasonably close to the analytical solution (Eq. 19.4):

$$z(t) = \frac{70}{0.275} \ln\left[\cosh\left(\sqrt{\frac{9.81(0.275)}{70}}8\right)\right] = 234.1$$

A graph of the numerical and analytical solutions along with both the exact and rounded velocities can be generated with the following commands:

```
>> ta=linspace(t(1),t(length(t)));
>> za=m/cd*log(cosh(sqrt(g*cd/m)*ta));
>> plot(ta,za,t,z,'o')
>> title('Distance versus time')
>> xlabel('t (s)'),ylabel('x (m)')
>> legend('analytical','numerical')
```

As in Fig. 19.15, the numerical and analytical results match fairly well.

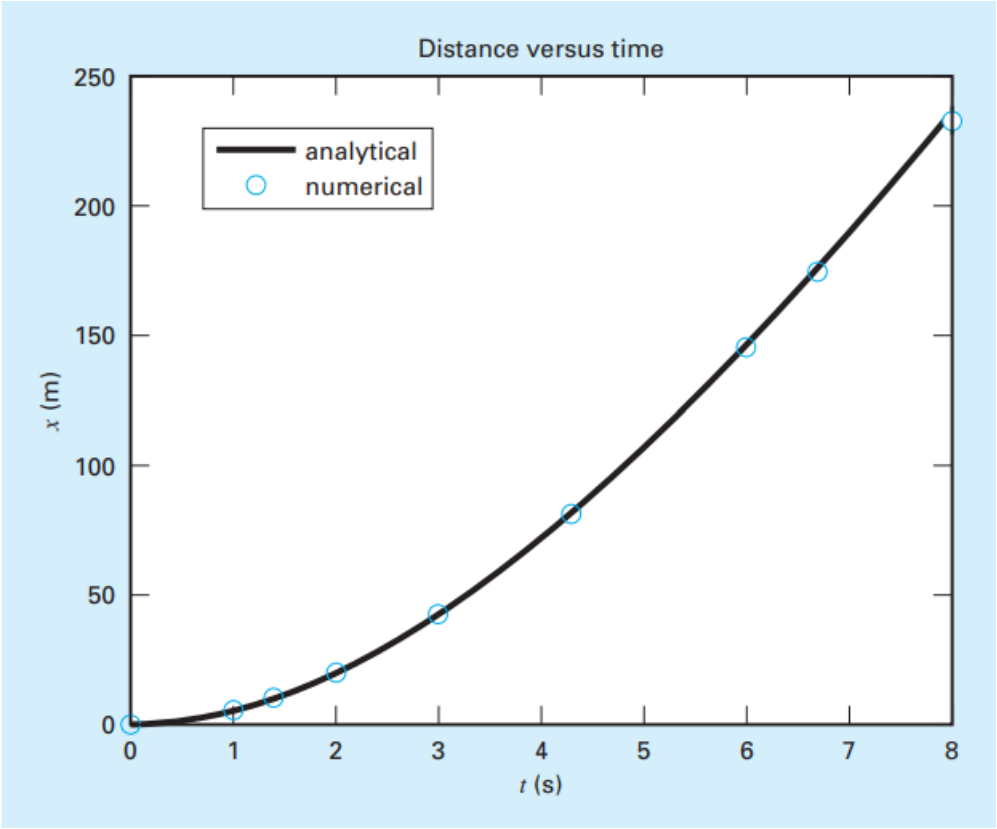


Figure 11.15: Plot of distance versus time. The line was computed with the analytical solution, whereas the points were determined numerically with the cumtrapz function.

Segments

(n)	Points	Name	Formula	Trunction Error
2	1	Midpoint method	$(b-a)f(x_1)$	$(1/3)h^3f''(\xi)$
3	2		$(b-a)\frac{f(x_1)+f(x_2)}{2}$	$(3/4)h^3f''(\xi)$
4	3		$(b-a)\frac{2f(x_1)-f(x_2)+2f(x_3)}{3}$	$(14/45)h^5f^{(4)}(\xi)$
5	4		$(b-a)\frac{11f(x_1)+f(x_2)+f(x_3)+11f(x_4)}{24}$	$(95/144)h^5f^{(4)}(\xi)$
6	5		$(b-a)\frac{11f(x_1)-14f(x_2)+26f(x_3)-14f(x_4)+11f(x_5)}{20}$	$(41/140)h^7f^{(6)}(\xi)$

11.8.OPEN METHODS

Recall from Fig. 19.6b that open integration formulas have limits that extend beyond the range of the data. Table 19.4 summarizes the Newton-Cotes open integration formulas. The formulas are expressed in the form of Eq. (19.13) so that the weighting factors are evident. 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. 22 and 23.

11.9.MULTIPLE INTEGRALS

Multiple integrals are widely used in engineering and science. For example, a general equation to compute the average of a two-dimensional function can be written as [recall Eq. (19.7)]

$$\bar{f} = \frac{\int_c^d \left(\int_a^b f(x,y)dx \right)}{(d-c)(b-a)} \tag{19.31}$$

The numerator is called a double integral. The techniques discussed in this chapter (and Chap. 20) 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. 19.16). Recall from calculus that such integrals can be computed as iterated integrals:

$$\int_c^d \left(\int_a^b f(x,y)dx \right) dy = \int_a^b \left(\int_c^d f(x,y)dy \right) dx \tag{19.32}$$

Thus, the integral in one of the dimensions is evaluated first. The result of this first integration is integrated in the second dimension. Equation (19.32) states that the order of integration is not important.

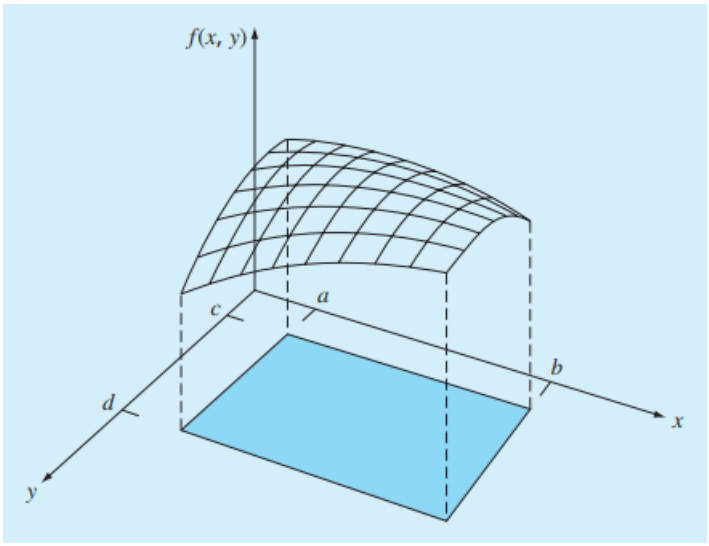


Figure 11.16: Double integral as the area under the function surface.

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.

Example 11. 21. Using Double Integral to Determine Average Temperature *Problem Statement.* Suppose that the temperature of a rectangular heated plate is described by the following function:

$$T(x,y) = 2xy + 2x - x^2 - 2y^2 + 72$$

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. 19.17. 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 2544. Dividing this by the area yields the average temperature as $2544/(6\times 8) = 53$
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.

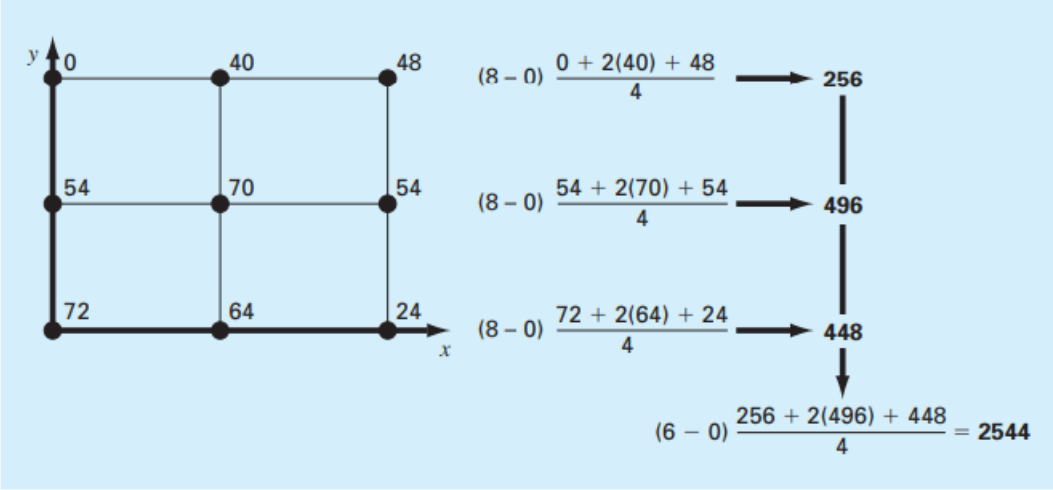


Figure 11.17: Numerical evaluation of a double integral using the two-segment trapezoidal rule

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. 20 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.

11.9.1 MATLAB Functions: dblquad and triplequad

MATLAB has functions to implement both double (*dblquad*) and triple (*triplequad*) integration. A simple representation of the syntax for *dblquad* is

```
q = dblquad(fun, xmin, xmax, ymin, ymax, tol)
```

where q is the double integral of the function fun over the ranges from xmin to xmax and ymin to ymax. If tol is not specified, a default tolerance of 1×10^{-6} is used.

Here is an example of how this function can be used to compute the double integral evaluated in Example 19.7:

```
>> q = dblquad(@(x,y) 2*x*y+2*x-x.^2-2*y.^2+72,0,8,0,6)
q =
   2816
```

11.10 CASE STUDY: COMPUTING WORK WITH NUMERICAL INTEGRATION

Background. The calculation of work is an important component of many areas of engineering and science. The general formula is

$$\text{Work} = \text{force} \times \text{distance}$$

When you were introduced to this concept in high school physics, simple applications were presented using forces that remained constant throughout the displacement. For example, if a force of 10 N was used to pull a block a distance of 5 m, the work would be calculated as 50 J (1 joule = 1 N Âů m).
Although such a simple computation is useful for introducing the concept, realistic problem settings are usually more complex. For example, suppose that the force varies during the course of the calculation. In such cases, the work equation is reexpressed as

$$W = \int_{x_0}^{x_n} F(x)dx \tag{19.33}$$

where W = work (J), x_0 and x_n = the initial and final positions (m), respectively, and $F(x)$ = a force that varies as a function of position (N). If $F(x)$ is easy to integrate, Eq. (19.33) can be evaluated analytically. However, in a realistic

problem setting, the force might not be expressed in such a manner. In fact, when analyzing measured data, the force might be available only in tabular form. For such cases, numerical integration is the only viable option for the evaluation.

Further complexity is introduced if the angle between the force and the direction of movement also varies as a function of position (Fig. 19.18). The work equation can be modified further to account for this effect, as in

$$W = \int_{x_0}^{x_n} F(x)\cos[\theta(x)]dx \tag{19.34}$$

Again, if $F(x)$ and $\theta(x)$ are simple functions, Eq. (19.34) might be solved analytically. However, as in Fig. 19.18, it is more likely that the functional relationship is complicated. For this situation, numerical methods provide the only alternative for determining the integral. Suppose that you have to perform the computation for the situation depicted in Fig. 19.18. Although the figure shows the continuous values for $F(x)$ and $\theta(x)$, assume that, because of experimental constraints, you are provided with only discrete measurements at $x = 5 - \text{m}$ intervals (Table 19.5). Use single- and composite versions of the trapezoidal rule and Simpson's $1/3$ and $3/8$ rules to compute work for these data.

Solution. The results of the analysis are summarized in Table 19.6. A percent relative error ε_t was computed in reference to a true value of the integral of 129.52 that was estimated on the basis of values taken from Fig. 19.18 at 1 -m intervals.

The results are interesting because the most accurate outcome occurs for the simple two-segment trapezoidal rule. More refined estimates using more segments, as well as Simpson's rules, yield less accurate results.

The reason for this apparently counterintuitive result is that the coarse spacing of the points is not adequate to capture the variations of the forces and angles. This is particularly

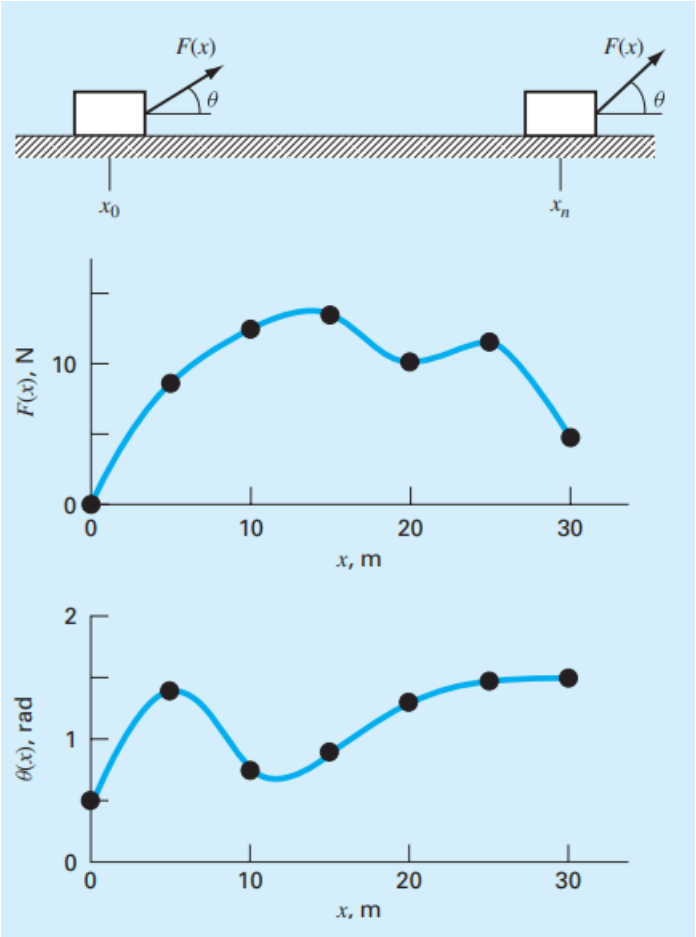


Figure 11.18: The case of a variable force acting on a block. For this case the angle, as well as the magnitude, of the force varies.

TABLE 19.5 Data for force $F(x)$ and angle $\theta(x)$ as a function of position x.

x, m	$F(x)$, N	θ, rad	$F(x)\cos\theta$
0	0.0	0.50	0.0000
5	9.0	1.40	1.5297
10	13.0	0.75	9.5120
15	14.0	0.90	8.7025
20	10.5	1.30	2.8087
25	12.0	1.48	1.0881
30	5.0	1.50	0.3537

TABLE 19.6 Estimates of work calculated using the trapezoidal rule and Simpson's rules. The percent relative error ε_t as computed in reference to a true value of the integral (129.52 Pa) that was estimated on the basis of values at 1 – m intervals.

Technique	Segments	Work	ε_t%
Trapezoidal rule	1	5.31	95.9
	2	133.19	2.84
	3	124.98	3.51
	6	119.09	8.05
Simpson's 1/3 rule	2	175.82	35.75
	6	117.13	9.57
Simpson's 3/8 rule	3	139.93	8.04

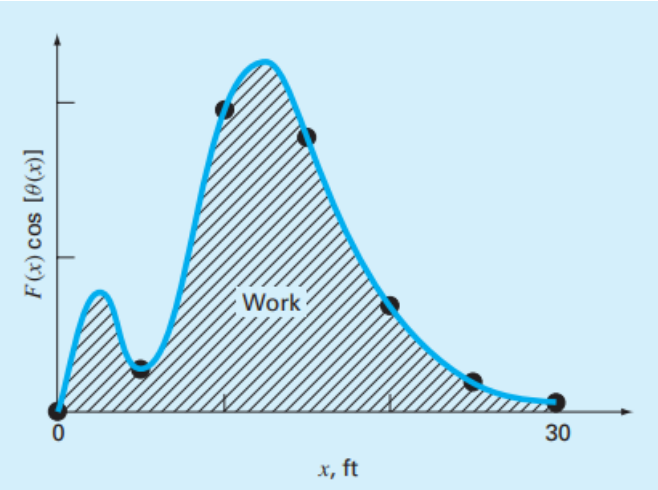


Figure 11.19: A continuous plot of F(x) cos [θ(x)] versus position with the seven discrete points used to develop the numerical integration estimates in Table 19.6. Notice how the use of seven points to characterize this continuously varying function misses two peaks at x = 2.5 and 12.5 m.

evident in Fig. 19.19, where we have plotted the continuous curve for the product of F(x) and cos [θ(x)]. Notice how the use of seven points to characterize the continuously varying function misses the two peaks at x = 2.5 and 12.5 m. The omission of these two points effectively limits the accuracy of the numerical integration estimates in Table 19.6. The fact that the two-segment trapezoidal rule yields the most accurate result is due to the chance positioning of the points for this particular problem (Fig. 19.20).

The conclusion to be drawn from Fig. 19.20 is that an adequate number of measurements must be made to accurately compute integrals. For the present case, if data were available at F(2.5) cos [θ(2.5)] = 3.9007 and F(12.5) cos [θ(12.5)] = 11.3940, we could

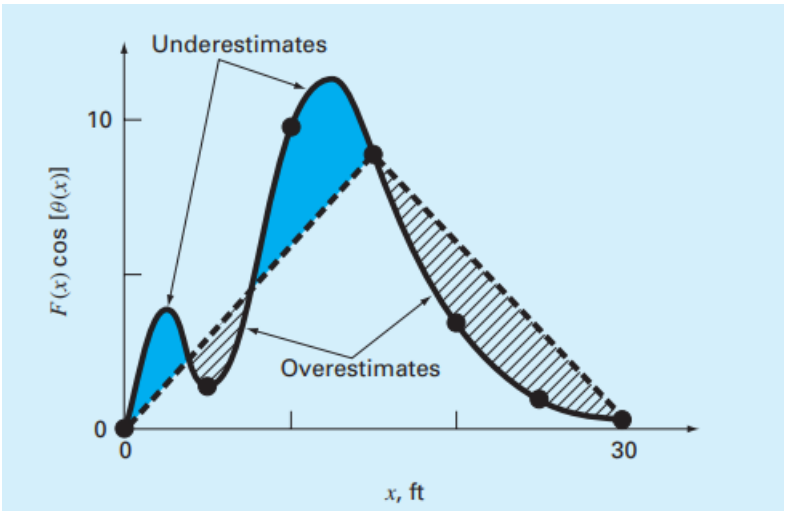


Figure 11.20: Graphical depiction of why the two-segment trapezoidal rule yields a good estimate of the integral for this particular case. By chance, the use of two trapezoids happens to lead to an even balance between positive and negative errors.

determine an improved integral estimate. For example, using the MATLAB trapz function, we could compute

```
>> x=[0 2.5 5 10 12.5 15 20 25 30];
>> y=[0 3.9007 1.5297 9.5120 11.3940 8.7025 2.8087 ...
1.0881 0.3537];
>> trapz(x,y)
ans =
132.6458
```

Including the two additional points yields an improved integral estimate of 132.6458 (ε_t = 2.16%). Thus, the inclusion of the additional data incorporates the peaks that were missed previously and, as a consequence, lead to better results.

11.12 PROBLEMS

19.1 Derive Eq. (19.4) by integrating Eq. (19.3).
19.2 Evaluate the following integral:

$$\int_0^4 \left(1-e^{-x}\right) dx$$

(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, (f) Simpson's 3/8 rule, and (g) composite Simpson's rule, with n = 5. For each of the numerical estimates (b) through (g), determine the true percent relative error based on (a).
19.3 Evaluate the following integral:

$$\int_0^{\pi/2} (8+4\cos x) dx$$

(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, (f) Simpson's 3/8 rule, and (g) composite Simpson's rule, with n = 5. For each of the numerical estimates (b) through (g), determine the true percent relative error based on (a).
19.4 Evaluate the following integral:

$$\int_{-2}^4 \left(1-x-4x^3+2x^5\right) dx$$

(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 true percent relative error based on (a).
19.5 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 = 0.6 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 true percent relative error.
19.6 Evaluate the double integral

$$\int_{-2}^2 \int_0^4 \left(x^2-3y^2+xy^3\right) dxdy$$

(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.

19.7 Evaluate the triple integral

$$\int_{-4}^4 \int_0^6 \int_{-1}^3 \left(x^3-2yz\right) dxdydz$$

(a) analytically, and (b) using single applications of Simpson's $1/3$ rule. For (b), compute the true percent relative error.

19.8 Determine the distance traveled from the following velocity data:

t	1	2	3.25	4.5	6	7	8	8.5	9	10
v	5	6	5.5	7	8.5	8	6	7	7	5

(a) Use the trapezoidal rule. In addition, determine the average velocity. (b) Fit the data with a cubic equation using polynomial regression. Integrate the cubic equation to determine the distance.
19.9

Chapter 12

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 to 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.

12.1. 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. 23.1. 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 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 $adaptive\ step - sizecontrol$. 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 em- ployed to efficiently evaluate definite integrals involving functions that are generally smooth but exhibit regions of abrupt change.

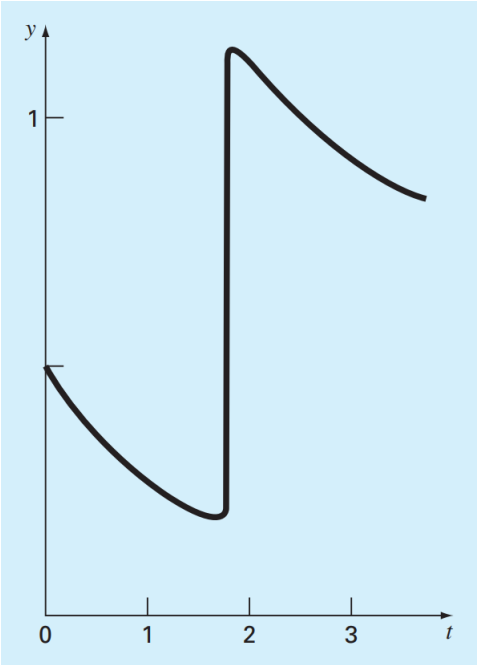


Figure 12.1: An example of a solution of an ODE that exhibits an abrupt change. Automatic step-size adjustment has great advantages for such cases.

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 trunca- tion 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 – Fehlbergmethods*. At face value, the idea of using two predictions of different order might seem too computationally expensive. For example, a fourth-and fift-horder prediction amounts to a total of 10 function evaluations per step [recall Eqs. (22.44) and (22.45)]. 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!

12.1.1MATLAB Function for Nonstiff System

Since Fehlberg originally developed his approach, other even better approaches have been developed. Several of these are available as built-in functions in MATLAB.

ode23. 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

$$y_{i+1} = y_i + \frac{1}{9}(2k_1 + 3k_2 + 4k_3)h \tag{23.1}$$

where

$$k_1 = f(t_i, y_i) \tag{23.1a}$$

$$k_2 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h\right) \tag{23.1b}$$

$$k_3 = f\left(t_i + \frac{3}{4}h, y_i + \frac{3}{4}k_2 h\right) \tag{23.1c}$$

The error is estimated as

$$E_{i+1} = \frac{1}{72}(-5k_1 + 6k_2 + 8k_3 - 9k_4)h \tag{23.2}$$

where

$$k_4 = f(t_{i+1}, y_{i+1}) \tag{23.2a}$$

Note that although there appear to be four function evaluations, there are really only three because after the first step, the $k1$ for the present step will be the $k4$ 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 thirdorder (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 $k4$ becomes $k1$ for the next step. If the error is too large, the step is repeated with reduced step sizes until the estimated error satisfies

$$E \leq \max(\text{RelTol} \times |y|, \text{ AbsTol }) \tag{23.3}$$

where RelTol is the relative tolerance (default = 10^{-3}) and AbsTol is the absolute tolerance (default = 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.

ode45. The `ode45` function uses an algorithm developed by Dormand and Prince (1980), 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.

ode113. 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 23.2.

These functions can be called in a number of different ways. The simplest approach is

```
[t, y] = ode45(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` = 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,..., tn` (all increasing or all decreasing), use

```
tspan = [t0 t1...tn];
```

Here is an example of how `ode45` can be used to solve a single ODE, $y' = 4e^{0.8t}$ - 0.5y from t = 0 to 4 with an initial condition of y (0) = 2. Recall from Example 22.1 that the analytical solution at t = 4 is 75.33896. Representing the ODE as an anonymous function, `ode45` can be used to generate the same result numerically as

```
» dydt=@ (t,y) 4*exp (0.8*t)-0.5*y;
» [t,y]=ode45(dydt,[0 4],2);
» y (length (t))
ans = 75.3390
```

As described in the following example, the ODE is typically stored in its own M-file when dealing with systems of equations.

Example 12. 22. Using MATLAB to Solve a System of ODEs

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 solve can be then 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. 23.2.

In addition to a time series plot, it is also instructive to generate a *phase-plane* plot-that is, a plot of the dependent variables versus each other by

```
» plot(y(:,1),y(:,2))
```

which yields Fig. 23.3.

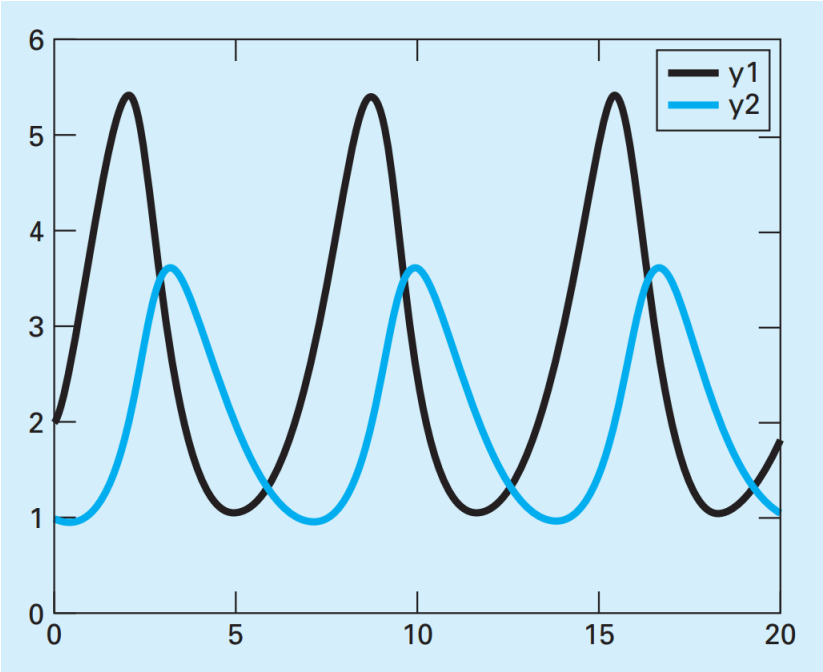


Figure 12.2: Solution of predator-prey model with MATLAB.

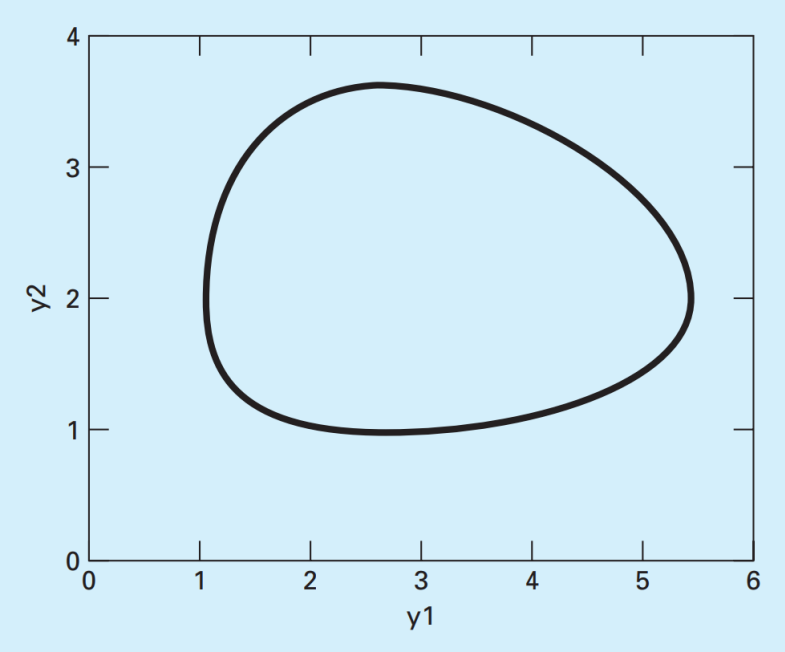


Figure 12.3: State-space plot of predator-prey model with MATLAB.

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,...)`

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 $\texttt{par}_i$ has the value $\texttt{val}_i$. 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 tspan interval. This option allows you to override this default.

Example 12.1. Using `odeset` to Control Integartions 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.

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. 23.4a, 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. 23.4*b*, the solver takes more small steps to attain the increased accuracy.

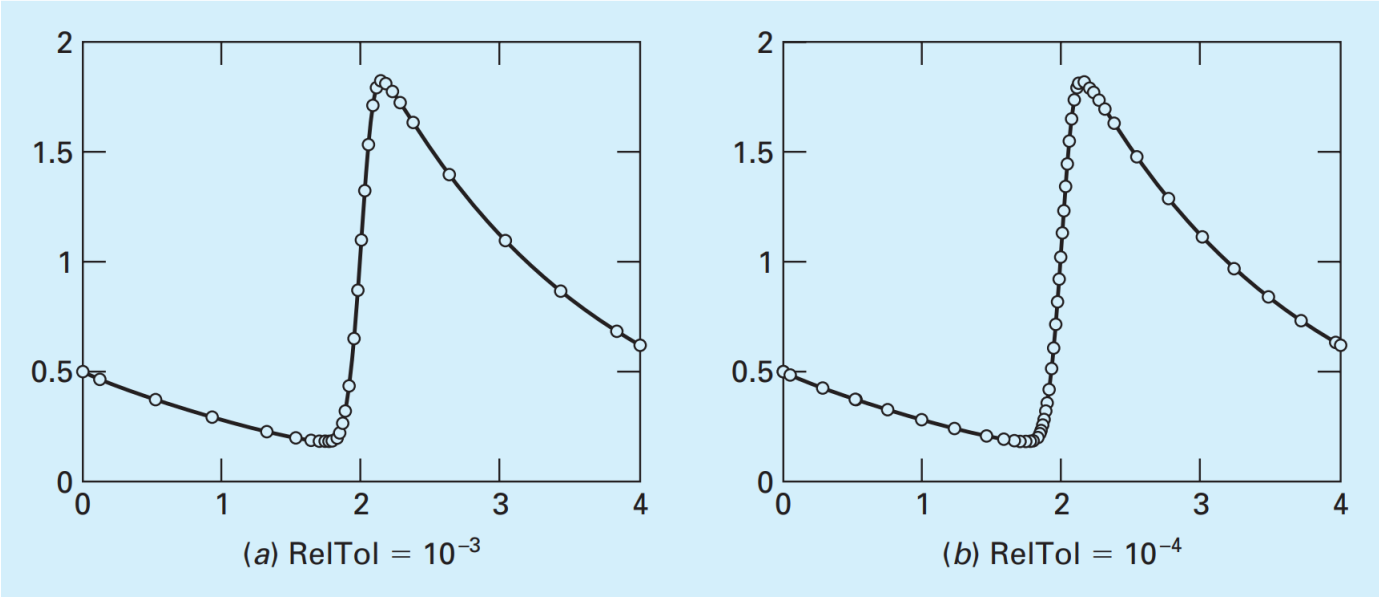


Figure 12.4: Solution of ODE with MATLAB. For (b), a smaller relative error tolerance is used and hence many more steps are taken.

■

12.1.2Events

MATLAB's ODE solvers are commonly implemented for a prespecified integration interval. That is, they are often used to obtain a solution from an initial to a final value of the dependent variable. However, there are many problems where we do not know the final time.

A nice example relates to the free-falling bungee jumper that we have been using throughout this book. Suppose that the jump master inadvertently neglects to attach the cord to the jumper. The final time for this case, which corresponds to the jumper hitting the ground, is not a given. In fact, the objective of solving the ODEs would be to determine when the jumper hit the ground.

MATLAB's `events` option provides a means to solve such problems. It works by solving differential equations until one of the dependent variables reaches zero. Of course, there may be cases where we would like to terminate the computation at a value other than zero. As described in the following paragraphs, such cases can be readily accommodated.

We will use our bungee jumper problem to illustrate the approach. The system of ODEs can be formulated as

$$\frac{dx}{dt} = v$$
$$\frac{dv}{dt} = g - \frac{c_d}{m} v|v|$$

where x = distance (m), t = time (s), v = velocity (m/s) where positive velocity is in the downward direction, g = the acceleration of gravity (= 9.81 m/s^2), c_d = a second-orderdrag coefficient (kg/m), and m = mass (kg). Note that in this formulation, distance andvelocity are both positive in the downward direction, and the ground level is defined aszero distance. For the present example, we will assume that the jumper is initially located 200 m above the ground and the initial velocity is 20 m/s in the upward direction-that is, x(0) = -200 and v(0) = -20.

The first step is to express the system ODEs as an M-File function:

```
function dydt=freeall (t, y, cd, m)
% y(1) = x and y(2) = v
grav=9.81;
dydt=[y(2);grav-cd/m*y(2)*abs(y(2))];
```

In order to implement the event, two other M-files need to be developed. These are (1) a function that defines the event, and (2) a script that generates the solution.

For our bungee jumper problem, the event function (which we have named `endevent`) can be written as

```
function [detect,stopint,direction]=endevent(t,y,varargin)
% Locate the time when height passes through zero
% and stop integration.
detect=y(1);   % Detect height = 0
stopint=1;   % Stop the integration
direction=0;   % Direction does not matter
```

This function is passed the values of the independent (`t`) and dependent variables (`y`) along with the model parameters (`varargin`). It then computes and returns three variables. The first, `detect`, specifies that MATLAB should detect the event when the dependent variable `y(1)` equals zero-that is, when the height x = 0. The second, `stopint`, is set to 1. This instructs MATLAB to stop when the event occurs. The final variable, `direction`, is set to 0 if all zeros are to be detected (this is the default), +1 if only the zeros where the event function increases are to be detected, and -1 if only the zeros where the event function decreases are to be detected. In our case, because the direction of the approach to zero is unimportant, we set `direction` to zero.[1]

Finally, a script can be developed to generate the solution:

```
opts=odeset('events',@endevent);
y0=[-200 -20];
[t,y,te,ye]=ode45(@freefall,[0 inf],y0,opts,0.25,68.1);
te,ye
plot(t,-y(:,1),'-',t,y(:,2),'-','LineWidth',2)
legend('Height (m)','Velocity (m/s)')
xlabel('time (s)');
ylabel('x (m) and v (m/s)')
```

In the first line, the `odeset` function is used to invoke the `events` option and specify that the event we are seeking is defined in the `endevent` function. Next, we set the initial conditions (`y0`) and the integration interval (`tspan`). Observe that because we do not know when the jumper will hit the ground, we set the upper limit of the integration interval to infinity. The third line then employs the `ode45` function to generate the actual solution. As in all of MATLAB's ODE solvers, the function returns the answers in the vectors `t` and `y`. In addition, when the `events` option is invoked, `ode45` can also return the time at which the event occurs (`te`), and the corresponding values of the dependent variables (`ye`). The remaining lines of the script merely display and plot the results. When the script is run, the output is displayed as

```
te =
      9.5475
ye =
      0.0000       46.2425
```

The plot is shown in Fig. 23.5. Thus, the jumper hits the ground in 9.5475 s with a velocity of 46.2454 m/s.

[1] Note that, as mentioned previously, we might want to detect a nonzero event. For example, we might want to detect when the jumper reached x = 5. To do this, we would merely set detect = y(1) - 5.

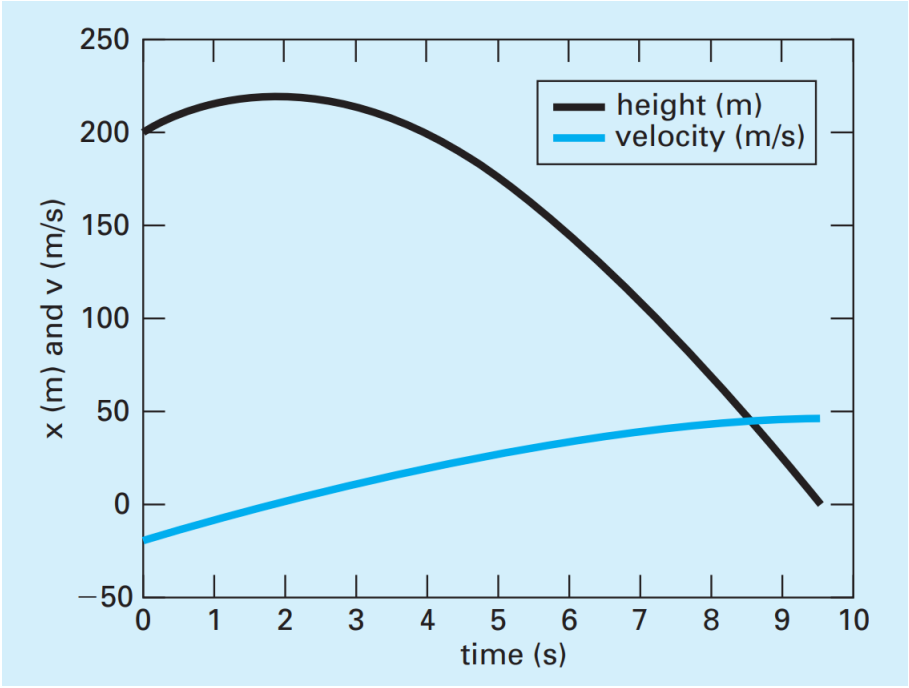


Figure 12.5: MATLAB-generated plot of the height above the ground and velocity of the free-falling bungee jumper without the cord.

12.2. 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. 23.6a). Alternative approaches, called *multistep methods* (Fig. 23.6b), are based on the insight that, once the computation has begun, valuable information from previous points is at our 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.

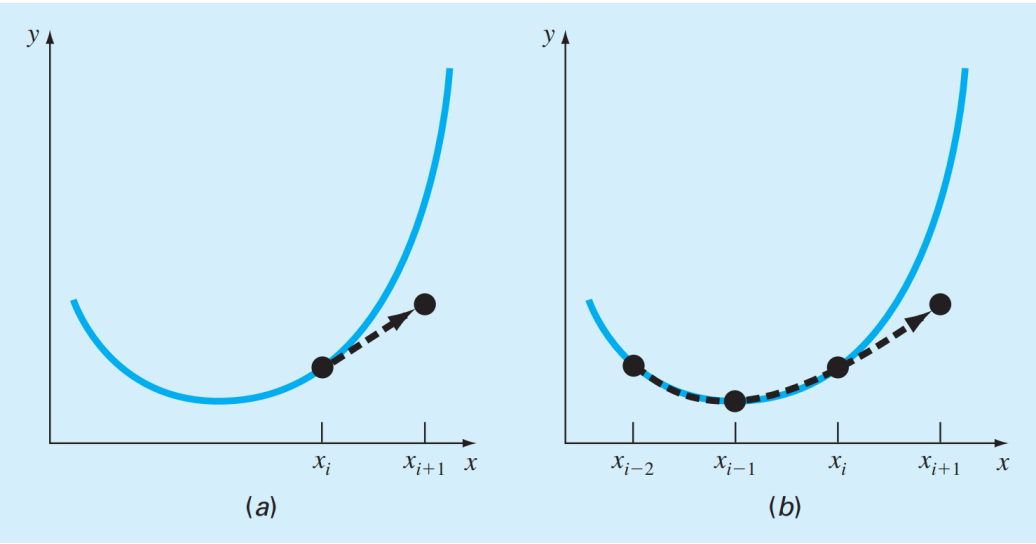


Figure 12.6: Graphical depiction of the fundamental difference between (a) one-step and (b) multistep methods for solving ODEs.

12.2.1 The Non-Self-Starting Heun Method

Recall that the Heun approach uses Euler's method as a predictor [Eq. (22.15)]:

$$y_{i+1}^0 = y_i + f(t_i, y_i)\,h \tag{23.4}$$

and the trapezoidal rule as a corrector [Eq. (22.17)]:

$$y_{i+1} = y_i + \frac{f(t_i, y_i) + f\left(t_{i+1}, y_{i+1}^0\right)}{2} h \tag{23.5}$$

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

$$y_{i+1}^0 = y_{i-1} + f(t_i, y_i)\,2h \tag{23.6}$$

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. (23.5) and (23.6) are called the *non-self-starting Heun method*. As depicted in Fig. 23.7, the derivative estimate in Eq. (23.6) 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 $O(h^3)$.

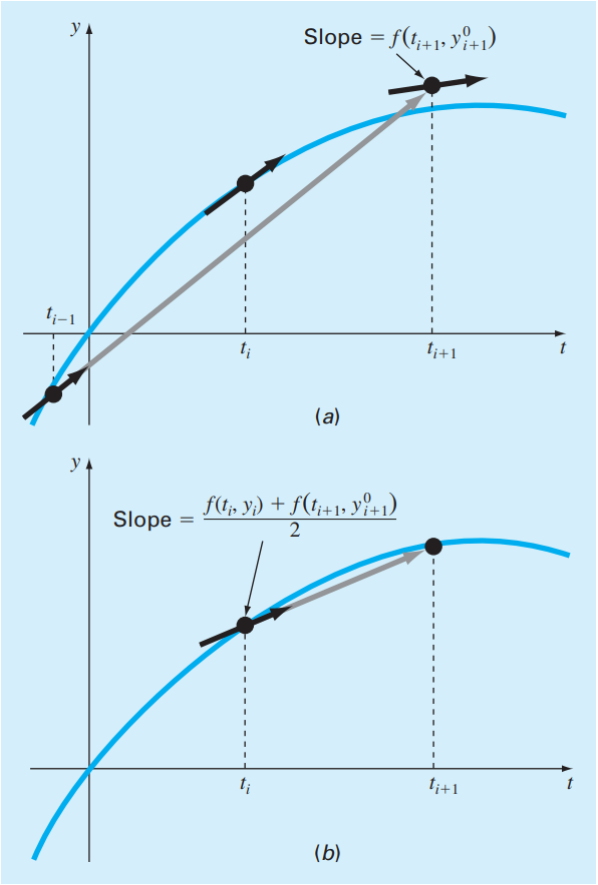


Figure 12.7: A graphical depiction of the non-self-starting Heun method. (a) The midpoint method that is used as a predictor. (b) The trapezoidal rule that is employed as a corrector.

The non-self-starting Heun method can be summarized as
Predictor (Fig. 23.7a):

$$y_{i+1}^0 = y_{i-1}^m + f(t_i, y_i^m)\,2h \tag{23.7}$$

Corrector (Fig. 23.7b):

$$y_{i+1}^j = y_i^m + \frac{f(t_i, y_i^m) + f\left(t_{i+1}, y_{i+1}^{j-1}\right)}{2} h \tag{23.8}$$

(for j = 1, 2,. . . , 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,

$$|\varepsilon_a| = \left| \frac{y_{i+1}^j - y_{i+1}^{j-1}}{y_{i+1}^j} \right| \times 100\% \tag{23.9}$$

When $|\varepsilon_a|$ is less than a prespecified error tolerance ε_s, the iterations are terminated. At this point, $j = m$. The use of Eqs. (23.7) through (23.9) to solve an ODE is demonstrated in the following example.

Example 12. 23. 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 22.2 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 22.2, 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. (23.7)] is used to extrapolate linearly from t = -1 to 1:

$$y_1^0 = -0.3929953 + \left[4e^{0.8(0)} - 0.5(2)\right] 2 = 5.607005$$

The corrector [Eq. (23.8)] 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. (23.8) 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. (23.9):

$$|\varepsilon_a| = \left| \frac{6.313749 - 6.549331}{6.313749} \right| \times 100\% = 3.7\%$$

Equation (23.8) can be applied iteratively until ε_a falls below a prespecified value of ε_s. As as the case with the Heun method (recall Example 22.2), 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

$$y_2^0 = 2 + \left[4e^{0.8(1)} - 0.5(6.36087)\right] 2 = 13.44346 \quad \varepsilon_t = 9.43\%$$

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.

12.2.2Error 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 23.1, 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 19.4)

$$E_p = \frac{1}{3} h^3 y^{(3)}(\xi_p) = \frac{1}{3} h^3 f''(\xi_p) \tag{23.10}$$

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

$$True\ value = y_{i+1}^0 + \frac{1}{3} h^3 y^{(3)}(\xi_p) \tag{23.11}$$

By recognizing that the corrector is equivalent to the trapezoidal rule, a similar estimate of the local truncation error for the corrector is (Table 19.2)

$$E_c = -\frac{1}{12} h^3 y^{(3)}(\xi_c) = -\frac{1}{12} h^3 f''(\xi_c) \tag{23.12}$$

This error estimate can be combined with the corrector result y_{i+1} to give

$$True\ value = y_{i+1}^m - \frac{1}{12}h^3 y^{(3)}(\xi_c) \tag{23.13}$$

Equation (23.11) can be subtracted from Eq. (23.13) to yield

$$0 = y_{i+1}^m - y_{i+1}^0 - \frac{5}{12}h^3 y^{(3)}(\xi) \tag{23.14}$$

where ξ is now between t_{i-1} and t_i. Now, dividing Eq. (23.14) by 5 and rearranging the result gives

$$\frac{y_{i+1}^0 - y_{i+1}^m}{5} = -\frac{1}{12}h^3 y^{(3)}(\xi) \tag{23.15}$$

Notice that the right-hand sides of Eqs. (23.12) and (23.15) 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

$$E_c = -\frac{y_{i+1}^0 - y_{i+1}^m}{5} \tag{23.16}$$

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 $\left(y_{i+1}^0\right)$ and the corrector $\left(y_{i+1}^m\right)$.

Example 12. 24. Estimate of Per-Step Truncation Error

Problem Statement. Use Eq. (23.16) to estimate the per-step truncation error of Example 23.3. 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. (23.16) 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, 2010). 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.

12.3. 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.

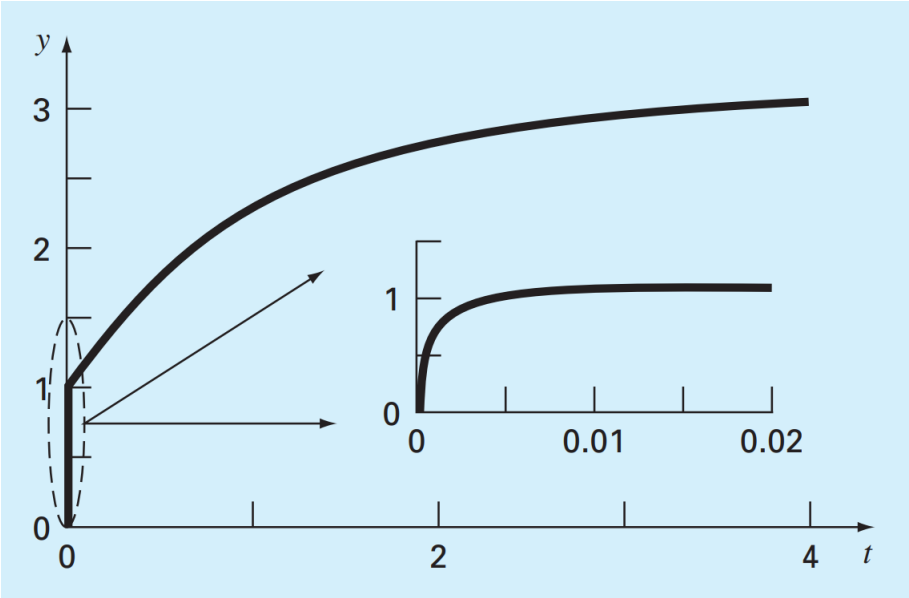


Figure 12.8: Plot of a stiff solution of a single ODE. Although the solution appears to start at 1, there is actually a fast transient from y = 0 to 1 that occurs in less than the 0.005 time unit. This transient is perceptible only when the response is viewed on the finer timescale in the inset.

Both individual and systems of ODEs can be stiff. An example of a single stiff ODE is

$$\frac{dy}{dt} = -1000y + 3000 - 2000e^{-t} \tag{23.17}$$

If $y(0) = 0$, the analytical solution can be developed as

$$y = 3 - 0.998e^{-1000t} - 2.002e^{-t} \tag{23.18}$$

As in Fig. 23.8, the solution is initially dominated by the fast exponential term $\left(e^{-1000t}\right)$. After a short period $(t < 0.005)$, this transient dies out and the solution becomes governed by the slow exponential (e^{-t}).

Insight into the step size required for stability of such a solution can be gained by examining the homogeneous part of Eq. (23.17):

$$\frac{dy}{dt} = -ay \tag{23.19}$$

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. (23.19) gives

$$y_{i+1} = y_i - ay_i h$$

or

$$y_{i+1} = y_i(1 - ah) \tag{23.20}$$

The stability of this formula clearly depends on the step size h. That is, $|1 - ah|$ must be less than 1. Thus, if $h > 2/a, |y_i| \to \infty$ as $i \to \infty$.

For the fast transient part of Eq. (23.18), 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. (23.19) yields

$$y_{i+1} = y_i - ay_{i+1}h$$

which can be solved for

$$y_{i+1} = \frac{y_i}{1+ah} \tag{23.21}$$

For this case, regardless of the size of the step, $|y_i| \to 0$ as $i \to \infty$. Hence, the approach is *called unconditionally stable*.

Example 12. 25. Explicit and Implicit Euler

Problem statement. Use both the explicit and implicit Euler methods to solve Eq. (23.17), 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

$$y_{i+1} = y_i + \left(-1000y_i + 3000 - 2000e^{-t_i}\right)h$$

The result for $h = 0.0005$ is displayed in Fig. 23.9a 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

$$y_{i+1} = y_i + \left(-1000y_{i+1} + 3000 - 2000e^{-t_{i+1}}\right)h$$

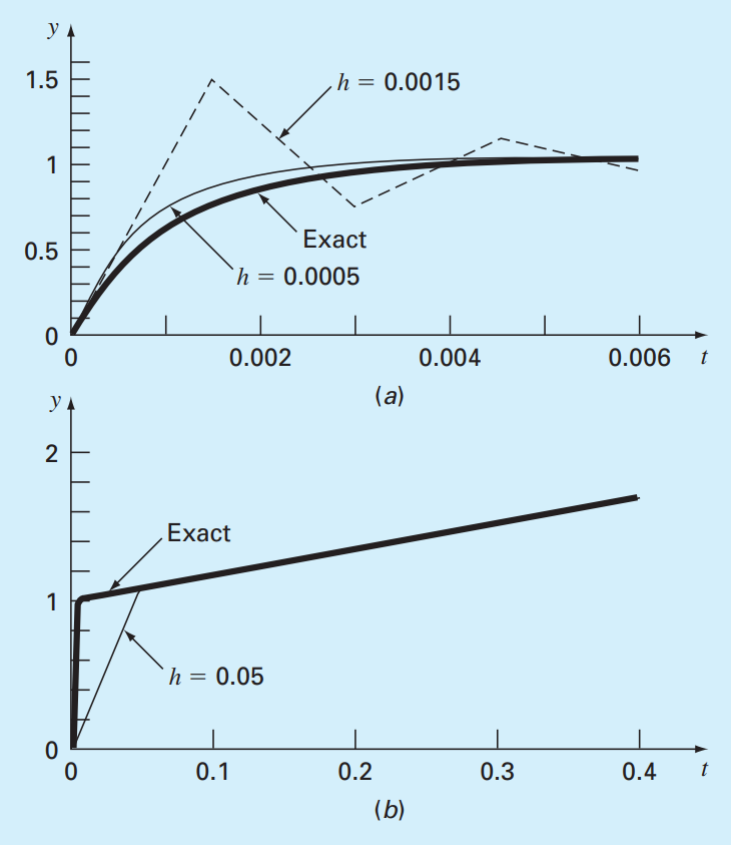


Figure 12.9: Solution of a stiff ODE with (a) the explicit and (b) implicit Euler methods.

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. 23.9*b* 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

$$\frac{dy_1}{dt} = -5y_1 + 3y_2 \tag{23.22a}$$

$$\frac{dy_2}{dt} = 100y_1 - 301y_2 \tag{23.22b}$$

For the initial conditions $y_1(0) = 52.29$ and $y_2(0) = 83.82$, the exact solution is

$$y_1 = 52.96e^{-3.9899t} - 0.67e^{-302.0101t} \tag{23.23a}$$

$$y_2 = 17.83e^{-3.9899t} + 65.99e^{-302.0101t} \tag{23.23b}$$

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

$$y_{1,i+1} = y_{1,i} + (-5y_{1,i+1} + 3y_{2,i+1})\,h \tag{23.24a}$$

$$y_{2,i+1} = y_{2,i} + (100y_{1,i+1} - 301y_{2,i+1})\,h \tag{23.24b}$$

Collecting terms gives

$$(1+5h)y_{1,i+1} - 3y_{2,i+1} = y_{1,i} \tag{23.25a}$$

$$-100y_{1,i+1} + (1+301h)y_{2,i+1} = y_{2,i} \tag{23.25b}$$

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 Sec. 12.2). Thus, although stability is gained through implicit approaches, a price is paid in the form of added solution complexity.

12.3.1MATLAB Functions for Stiff Systems

MATLAB has a number of built-in functions for solving stiff systems of ODEs. These are

ode15s. 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.

ode23s. 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`.

ode23t. 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.

ode23tb. 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 12. 26. MATLAB for Stiff ODE's
Problem statement. The van der Pol equation is a model of an electronic circuit that arose back in the days of vacuum tubes,

$$\frac{d^2y_1}{dt^2} - \mu\left(1 - y_1^2\right)\frac{dy_1}{dt} + y_1 = 0 \tag{E23.6.1}$$

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 ode 45 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, (E23.6.1) can be written as

$$\frac{dy_2}{dt} = \mu\left(1 - y_1^2\right)y_2 - y_1 = 0$$

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)];
```

Notice how the value of μ is passed as a parameter. As in Example 23.1, `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. 23.10a) 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))
```

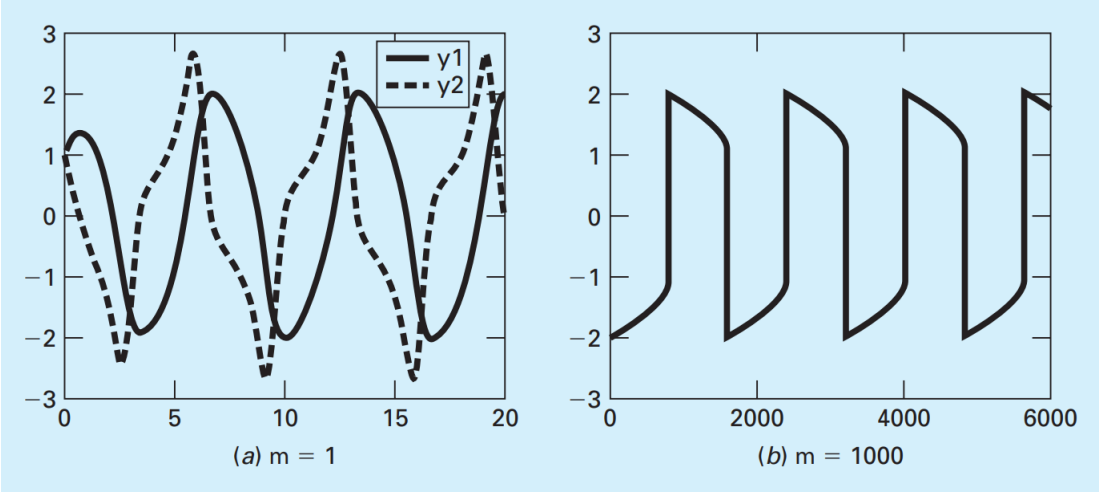


Figure 12.10: Solutions for van der Pol's equation. (a) Nonstiff form solved with ode45 and (b) stiff form solved with `ode23s`.

We have only displayed the $y1$ component because the result for $y2$ has a much larger scale. Notice how this solution (Fig. 23.10b) has much sharper edges than is the case in Fig. 23.10a. This is a visual manifestation of the"stiffnes" of the solution.

12.4.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. 22, the problem consisted of solving two coupled ODEs for vertical position and velocity. The differential equation for position isolated

$$\frac{dx}{dt} = v \tag{23.26}$$

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,

$$\frac{dv}{dt} = g - \text{sign}(v)\frac{c_d}{m}v^2 \tag{23.27a}$$

Once the cord begins to stretch, the spring and dampening forces of the cord must also be included:

$$\frac{dv}{dt} = g - \text{sign}(v)\frac{c_d}{m}v^2 - \frac{k}{m}(x-L) - \frac{\gamma}{m}v \tag{23.27b}$$

The following example shows how MATLAB can be used to solve this problem.

Example 12. 27. Bungee Jumper with cord

Problem statement. Determine the position and velocity of a bungee jumper with the following parameters: $L = 30$ m, $g = 9.81$ m/s^2, $m = 68.1$ kg, $c_d = 0.25$ kg/m, $k = 40$ N/m, and $\gamma = 8$ N $\cdot$ 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];
```

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],[],30,0.25,68.1,40,8);
» plot(t,-y(:,1),'-',t,y(:,2),':')
» legend('x (m)','v (m/s)')
```

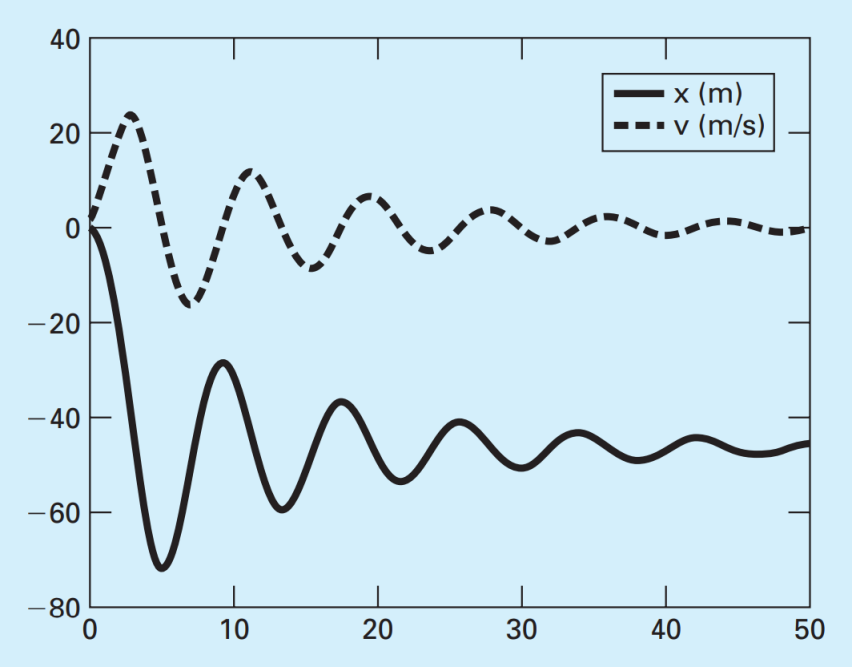


Figure 12.11: Plot of distance and velocity of a bungee jumper.

12.5. CASE STUDY: PLINY'S INTERMITTENT FOUNTAIN

Background. The Roman natural philosopher, Pliny the Elder, purportedly had an intermittent fountain in his garden. As in Fig. 23.12, water enters a cylindrical tank at a constant flow rate Q_{in} and fills until the water reaches y_{high}. At this point, water siphons out of the tank through a circular discharge pipe, producing a fountain at the pipe's exit. The fountain runs until the water level decreases to y_{low}, whereupon the siphon fills with air and the fountain stops. The cycle then repeats as the tank fills until the water reaches y_{high}, and the fountain flows again.

When the siphon is running, the outflow Q_{out} can be computed with the following formula based on Torricelli's law:

$$Q_{\text{out}} = C\sqrt{2gy}\pi r^2 \tag{23.28}$$

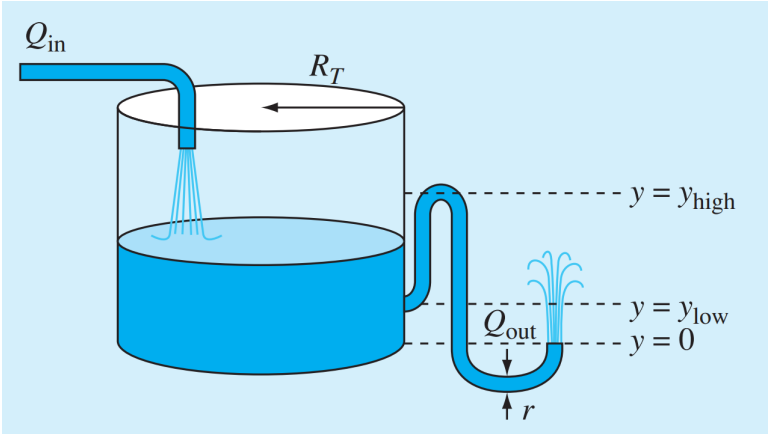


Figure 12.12: An intermittent fountain.

Neglecting the volume of water in the pipe, compute and plot the level of the water in the tank as a function of time

over 100 seconds. Assume an initial condition of an empty tank y(0) = 0, and employ the following parameters for your computation:

$$R_T = 0.05 \text{ m} \quad r = 0.007 \text{ m} \quad y_{\text{low}} = 0.025 \text{ m}$$
$$y_{\text{high}} = 0.1 \text{ m} \quad C = 0.6 \quad g = 9.81 \text{ m/s}^2$$
$$Q_{\text{in}} = 50 \times 10^{-6} \text{ m}^3/\text{s}$$

Solution. When the fountain is running, the rate of change in the tanks volume V (m^3) is determined by a simple balance of inflow minus the outflow:

$$\frac{dV}{dt} = Q_{\text{in}} - Q_{\text{out}} \tag{23.29}$$

where $V = \text{volume}\left(\text{m}^3\right)$. Because the tank is cylindrical, $V = \pi R_t^2 y$. Substituting this relationship along with Eq. (23.28) into Eq. (23.29) gives

$$\frac{dy}{dt} = \frac{Q_{\text{in}} - C\sqrt{2gy}\pi r^2}{\pi R_t^2} \tag{23.30}$$

When the fountain is not running, the second term in the numerator goes to zero. We can incorporate this mechanism in the model by introducing a new dimensionless variable siphon that equals zero when the fountain is off and equals one when it is flowing:

$$\frac{dy}{dt} = \frac{Q_{\text{in}} - \text{siphon} \times C\sqrt{2gy}\pi r^2}{\pi R_t^2} \tag{23.31}$$

In the present context, siphon can be thought of as a switch that turns the fountain off and on. Such two-state variables are called *Boolean* or *logical variables*, where zero is equivalent to false and one is equivalent to true.

Next we must relate *siphon* to the dependent variable *y*. First, *siphon* is set to zero whenever the level falls below y_{low}. Conversely, *siphon* is set to one whenever the level rises above y_{high}. The following M-file function follows this logic in computing the derivative:

```
function dy = Plinyode(t,y)
global siphon
Rt = 0.05; r = 0.007; yhi = 0.1; ylo = 0.025;
C = 0.6; g = 9.81; Qin = 0.00005;
if y(1) <= ylo
 siphon = 0;
elseif y(1) >= yhi
 siphon = 1;
end
Qout = siphon * C * sqrt(2 * g * y(1)) * pi * r ^ 2;
dy = (Qin - Qout) / (pi * Rt ^ 2);
```

Notice that because its value must be maintained between function calls, `siphon` is declared as a global variable. Although the use of global variables is not encouraged (particularly in larger programs), it is useful in the present context.

The following script employs the built-in `ode45` function to integrate `Plinyode` and generate a plot of the solution:

```
global siphon
siphon = 0;
tspan = [0 100]; y0 = 0;
[tp,yp]=ode45(@Plinyode,tspan,y0);
plot(tp,yp)
xlabel('time, (s)')
ylabel('water level in tank, (m)')
```

As shown in Fig. 23.13, the result is clearly incorrect. Except for the original filling period, the level seems to start emptying prior to reaching y_{high}. Similarly, when it is draining, the siphon shuts off well before the level drops to y_{low}.

At this point, suspecting that the problem demands more firepower than the trusty `ode45` routine, you might be tempted to use one of the other MATLAB ODE solvers such as `ode23s` or `ode23tb`. But if you did, you would discover that although these routines yield somewhat different results, they would still generate incorrect solutions.

The difficulty arises because the ODE is discontinuous at the point that the siphon switches on or off. For example, as the tank is filling, the derivative is dependent only on the constant inflow and for the present parameters has a constant value of 6.366×10^{-3} m/s. However, as soon as the level reaches y_{high}, the outflow kicks in and the derivative abruptly drops to -1.013×10^{-2} m/s. Although the adaptive step-size routines used by MATLAB work marvelously for many problems, they often get heartburn when dealing with such discontinuities. Because they infer the behavior of the solution by comparing the results of different steps, a discontinuity represents something akin to stepping into a deep pothole on a dark street.

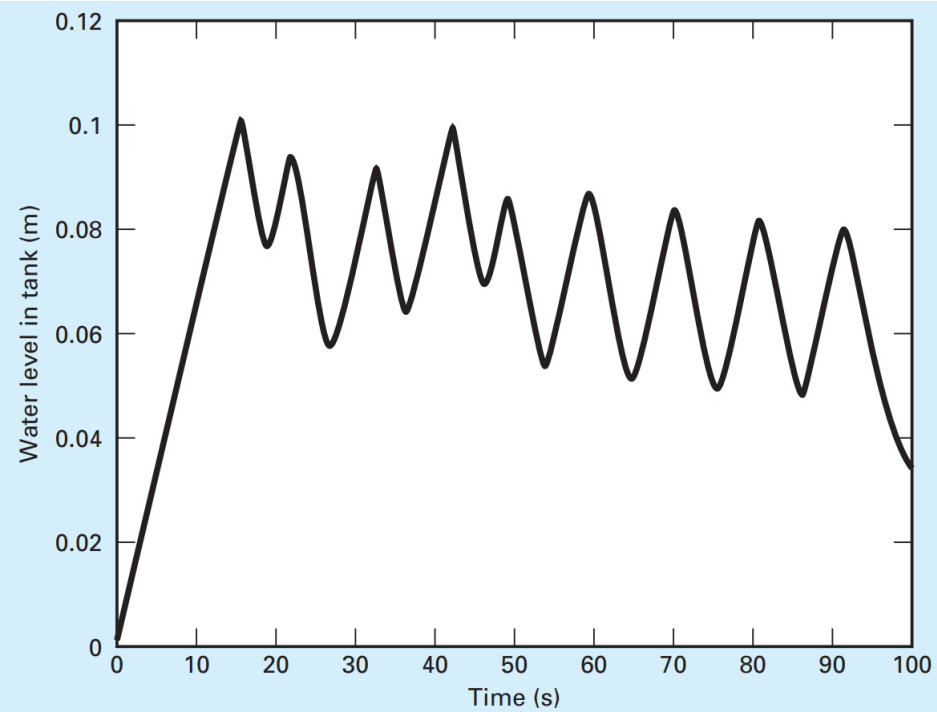


Figure 12.13: The level in Pliny's fountain versus time as simulated with `ode45`.

At this point, your first inclination might be to just give up. After all, if it's too hard for MATLAB, no reasonable person could expect you to come up with a solution. Because professional engineers and scientists rarely get away with such excuses, your only recourse is to develop a remedy based on your knowledge of numerical methods.

Because the problem results from adaptively stepping across a discontinuity, you might revert to a simpler approach and use a constant, small step size. If you think about it, that's precisely the approach you would take if you were traversing a dark, pothole-filled street. We can implement this solution strategy by merely replacing `ode45` with the constant-step `rk4sys` function from Chap. 22 (Fig. 22.8). For the script outlined above, the fourth line would be formulated as

```
[tp,yp] = rk4sys(@Plinyode,tspan,y0,0.0625);
```

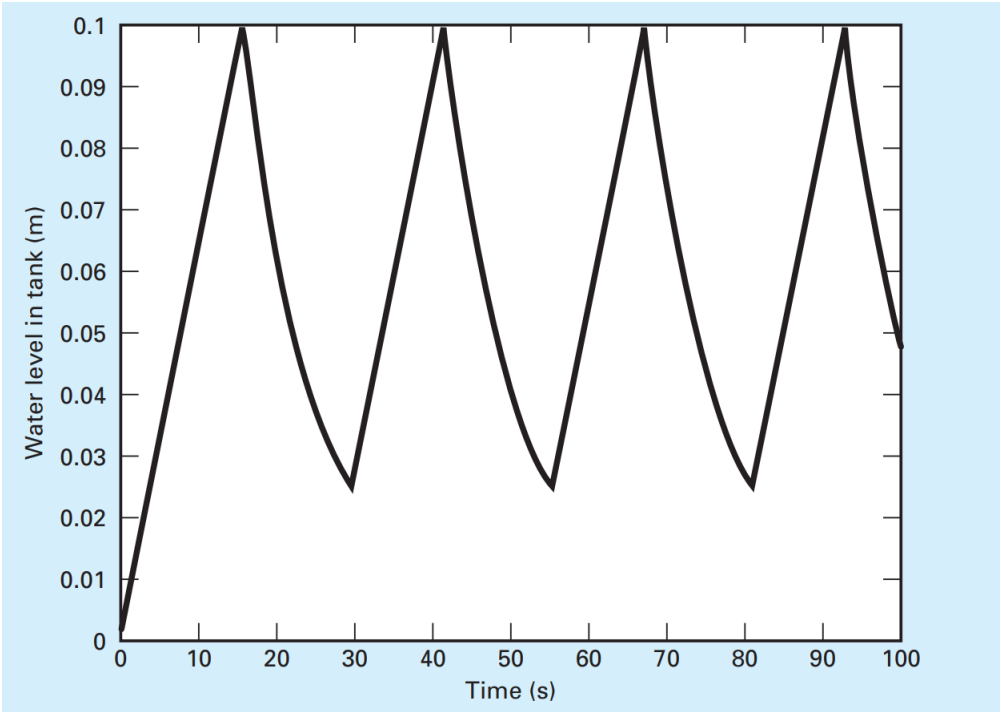


Figure 12.14: The level in Pliny's fountain versus time as simulated with a small, constant step size using the `rk4sys` function (Fig. 22.8).

As in Fig. 23.14, the solution now evolves as expected. The tank fills to y_{high} and then empties until it reaches y_{low}, when the cycle repeats.

There are a two take-home messages that can be gleaned from this case study. First, although it's human nature to think the opposite, simpler is sometimes better. After all, to paraphrase Einstein, "Everything should be as simple as possible, but no simpler." Second, you should never blindly believe every result generated by the computer. You've probably heard the old chestnut, "garbage in, garbage out" in reference to the impact of data quality on the validity of computer output. Unfortunately, some individuals think that regardless of what went in (the data) and what's going on inside (the algorithm), it's always"gospel out" Situations like the one depicted in Fig. 23.13 are particularly dangerous-that is, although the output is incorrect, it's not obviously wrong. That is, the simulation does not go unstable or yield negative levels. In fact, the solution moves up and down in the manner of an intermittent fountain, albeit incorrectly. Hopefully, this case study illustrates that even a great piece of software such as MATLAB is not foolproof. Hence, sophisticated engineers and scientists always examine numerical output with a healthy skepticism based on their considerable experience and knowledge of the problems they are solving.

12.6.PROBLEMS

23.1 Repeat the same simulations as in Section 23.5 for Pliny's fountain, but generate the solutions with `ode23`, `ode23s`, and `ode113`. Use subplot to develop a vertical three-pane plot of the time series.

23.2 The following ODEs have been proposed as a model of an epidemic:

$$\frac{dS}{dt} = -aSI$$
$$\frac{dI}{dt} = aSI - rI$$
$$\frac{dR}{dt} = rI$$

where $S =$ the susceptible individuals, $I =$ the infected, $R =$ the recovered, $a =$ the infection rate, and $r =$ the recovery rate. A city has 10,000 people, all of whom are susceptible.
(a) If a single infectious individual enters the city at $t = 0$, compute the progression of the epidemic until the number of infected individuals falls below 10 . Use the following parameters: $a = 0.002/$ (person $\cdot$ week) and $r = 0.15/\text{d}$. Develop time-series plots of all the state variables. Also generate a phase-plane plot of S versus I versus R.
(b) Suppose that after recovery, there is a loss of immunity that causes recovered individuals to become susceptible. This reinfection mechanism can be computed as ρR, where $\rho =$ the reinfection rate. Modify the model to include this mechanism and repeat the computations in (a) using $\rho = 0.03/\text{d}$.

23.3 Solve the following initial-value problem over the interval from $t = 2$ to 3 :

$$\frac{dy}{dt} = -0.5y + e^{-t}$$

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 $\varepsilon_s = 0.1\%$. Compute the percent relative errors for your results based on the exact solutions obtained analytically: $y(2.5) = 3.273888$ and $y(3.0) = 2.577988$.

23.4 Solve the following initial-value problem over the interval from $t = 0$ to 0.5 :

$$\frac{dy}{dt} = yt^2 - y$$

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: $y(0) = 1$.

23.5 Given

$$\frac{dy}{dt} = -100,000y + 99,999e^{-t}$$

(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.

23.6 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.

23.7 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.

23.8 The following nonlinear, parasitic ODE was suggested by Hornbeck (1975):

$$\frac{dy}{dt} = 5\left(y - t^2\right)$$

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.

23.9 Recall from Example 20.5 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` and **(b)** the `ode45` functions.

23.10 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/s^2. 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 ($\theta = \pi/2$). For each case, plot the linear and nonlinear simulations on the same plot.

23.11 Employ the events option described in Section 23.1.2 to determine the period of a $1-$m long, linear pendulum (see description in Prob. 23.10). Compute the period for the following initial conditions: **(a)** $\theta = \pi/8$, **(b)** $\theta = \pi/4$, and **(c)** $\theta = \pi/2$. For all three cases, set the initial angular velocity at zero. (**Hint:** A good way to compute the period is to determine how long it takes for the pendulum to reach $\theta = 0$ [i.e., the bottom of its arc]). The period is equal to four times this value.

23.12 Repeat Prob. 23.11, but for the nonlinear pendulum described in Prob. 23.10.

23.13 The following system is a classic example of stiff ODEs that can occur in the solution of chemical reaction kinetics:

$$\frac{dc_1}{dt} = -0.013c_1 - 1000c_1c_3$$
$$\frac{dc_2}{dt} = -2500c_2c_3$$
$$\frac{dc_3}{dt} = -0.013c_1 - 1000c_1c_3 - 2500c_2c_3$$

Solve these equations from $t = 0$ to 50 with initial conditions $c_1(0) = c_2(0) = 1$ and $c_3(0) = 0$. If you have access to MATLAB software, use both standard (e.g., `ode45`) and stiff (e.g., `ode23s`) functions to obtain your solutions.

23.14 The following second-order ODE is considered to be stiff:

$$\frac{d^2y}{dx^2} = -1001\frac{dy}{dx} - 1000y$$

Solve this differential equation (a) analytically and (b) numerically for x = 0 to 5. For (b) use an implicit approach with h = 0.5. Note that the initial conditions are y(0) = 1 and $y'(0) = 0$. Display both results graphically.

23.15 Consider the thin rod of length l moving in the $x-y$ plane as shown in Fig. P23. 15. The rod is fixed with a pin on one end and a mass at the other. Note that $g = 9.81$ m/s^2 and $l = 0.5$ m. This system can be solved using

$$\ddot{\theta} - \frac{g}{l}\theta = 0$$

Let $\theta(0) = 0$ and $\dot{\theta}(0) = 0.25$rad/s. Solve using any method studied in this chapter. Plot the angle versus time and the angular velocity versus time. (**Hint:** Decompose the secondorder ODE.)

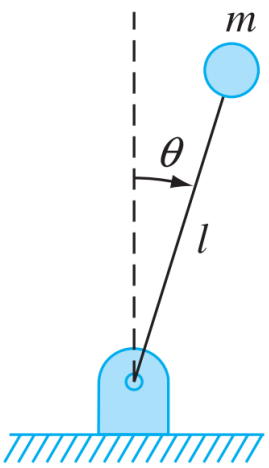


Figure 12.15:

23.16 Given the first-order ODE:

$$\frac{dx}{dt} = -700x - 1000e^{-t}$$
$$x(t=0) = 4$$

Solve this stiff differential equation using a numerical method over the time period $0 \le t \le 5$. Also solve analytically and plot the analytic and numerical solution for both the fast transient and slow transition phase of the time scale.

23.17 Solve the following differential equation from $t = 0$ to 2

$$\frac{dy}{dt} = -10y$$

with the initial condition $y(0) = 1$. Use the following techniques to obtain your solutions: **(a)** analytically, **(b)** the explicit Euler method, and **(c)** the implicit Euler method. For **(b)** and **(c)** use $h = 0.1$ and 0.2. Plot your results.

23.18 The Lotka-Volterra equations described in Section 22.6 have been refined to include additional factors that impact predator-prey dynamics. For example, over and above predation, prey population can be limited by other factors such as space. Space limitation can be incorporated into the model as a carrying capacity (recall the logistic model described in Prob. 22.5) as in

$$\frac{dx}{dt} = a\left(1 - \frac{x}{K}\right)x - bxy$$
$$\frac{dy}{dt} = -cy + dxy$$

where K = the carrying capacity. Use the same parameter values and initial conditions as in Section 22.6 to integrate these equations from $t = 0$ to 100 using `ode45`, and develop both time series and phase plane plots of the results. **(a)** Employ a very large value of $K = 10^8$ to validate that you obtain the same results as in Section 22.6. **(b)** Compare **(a)** with the more realistic carrying capacity of $K = 200$. Discuss your results.

23.19 Two masses are attached to a wall by linear springs (Fig. P23.19). Force balances based on Newton's second law can be written as

$$\frac{d^2x_1}{dt^2} = -\frac{k_1}{m_1}(x_1 - L_1) + \frac{k_2}{m_1}(x_2 - x_1 - w_1 - L_2)$$
$$\frac{d^2x_2}{dt^2} = -\frac{k_2}{m_2}(x_2 - x_1 - w_1 - L_2)$$

where $k =$ the spring constants, $m =$ mass, $L =$ the length of the unstretched spring, and $w =$ the width of the mass. Compute the positions of the masses as a function of time using the following parameter values: $k_1 = k_2 = 5, m_1 = m_2 = 2$, $w_1 = w_2 = 5$, and $L_1 = L_2 = 2$. Set the initial conditions as $x_1 = L_1$ and $x_2 = L_1 + w_1 + L_2 + 6$. Perform the simulation from $t = 0$ to 20. Construct time-series plots of both the displacements and the velocities. In addition, produce a phaseplane plot of x_1 versus x_2.

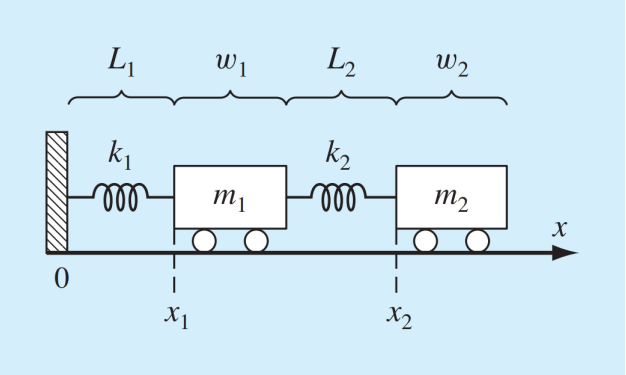


Figure 12.16:

23.20 Use `ode45` to integrate the differential equations for the system described in Prob. 23.19. Generate vertically stacked subplots of displacements (top) and velocities (bottom). Employ the `fft` function to compute the discrete Fourier transform (DFT) of the first mass's displacement. Generate and plot a power spectrum in order to identify the system's resonant frequencies.

23.21 Perform the same computations as in Prob. 23.20 but for the structure in Prob. 22.22.

23.22 Use the approach and example outlined in Section 23.1.2, but determine the time, height, and velocity when the bungee jumper is the farthest above the ground, and generate a plot of the solution.

Chapter 13

Boundary-Value Problems

CHAPTER OBJECTIVES

The primary objective of this chapter is to introduce you to solving boundary-value problems for ODEs. Specific objectives and topics covered are

- Understanding the difference between initial-value and boundary-value problems
- Knowing how to express an *n*th-order ODE as a system of n first-order ODEs.
- Knowing how to implement the shooting method for linear ODEs by using linear interpolation to generate accurate "shots."
- Understanding how derivative boundary conditions are incorporated into the shooting method.
- Knowing how to solve nonlinear ODEs with the shooting method by using root location to generate accurate "shots."
- Knowing how to implement the finite-difference method.
- Understanding how derivative boundary conditions are incorporated into the finite-difference method.
- Knowing how to solve nonlinear ODEs with the finite-difference method by using root-location methods for systems of nonlinear algebraic equations.

You've got a problem.
To this point, we have been computing the velocity of a free-falling bungee jumper by integrating a single ODE:

$$\frac{dv}{dt} = g - \frac{c_d}{m}v^2 \tag{24.1}$$

Suppose that rather than velocity, you are asked to determine the position of the jumper as a function of time. One way to do this is to recognize that velocity is the first derivative of distance:

$$\frac{dx}{dt} = v \tag{24.2}$$

Thus, by solving the system of two ODEs represented by Eqs. (24.1) and (24.2), we can simultaneously determine both the velocity and the position.

However, because we are now integrating two ODEs, we require two conditions to obtain the solution. We are already familiar with one way to do this for the case where we have values for both position and velocity at the initial time:

$$x(t = 0) = x_i$$
$$v(t = 0) = v_i$$

Given such conditions, we can easily integrate the ODEs using the numerical techniques described in Chaps. 22 and 23. This is referred to as an *initial-value problem*.

But what if we do not know values for both position and velocity at $t = 0$? Let's say that we know the initial position but rather than having the initial velocity, we want the jumper to be at a specified position at a later time. In other words:

$$x(t = 0) = x_i$$
$$x(t = t_f) = x_f$$

Because the two conditions are given at different values of the independent variable, this is called a *boundary-value problem*. Such problems require special solution techniques. Some of these are related to the methods for initial value problems that were described in the previous two chapters. However, others employ entirely different strategies to obtain solutions. This chapter is designed to introduce you to the more common of these methods.

13.1. INTRODUCITON AND BACKGROUND

13.1.1 What are Boundary-Value Problems?

An ordinary differential equation is accompanied by auxiliary conditions, which are used to evaluate the constants of integration that result during the solution of the equation. For an nth-order equation, n conditions are required. If all the conditions are specified at the same value of the independent variable, then we are dealing with an *initial-value problem* (Fig. 24.1*a*). To this point, the material in Part Six (Chaps. 22 and 23) has been devoted to this type of problem.

In contrast, there are often cases when the conditions are not known at a single point but rather are given at different values of the independent variable. Because these values are often specified at the extreme points or boundaries of a system, they are customarily referred to as *boundary-value problems* (Fig. 24.1*b*). A variety of significant engineering applications fall within this class. In this chapter, we discuss some of the basic approaches for solving such problems.

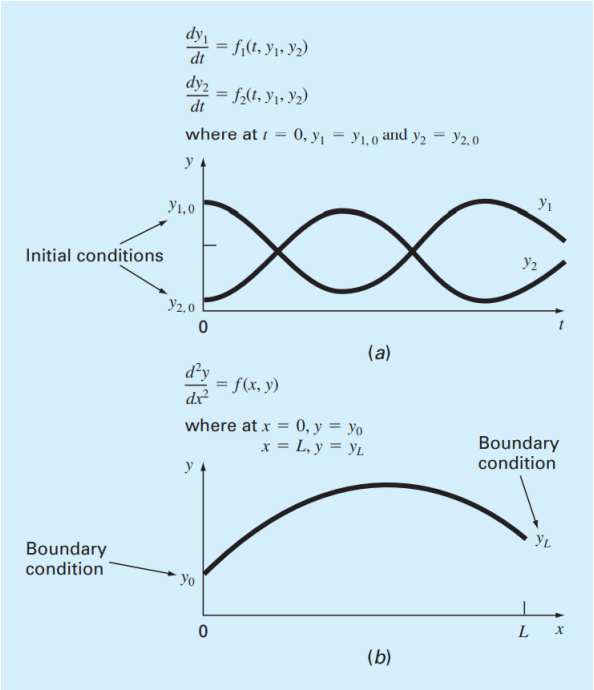


Figure 13.1: Initial-value versus boundary-value problems. (a) An initial-value problem where all the conditions are specified at the same value of the independent variable. (b) A boundary-value problem where the conditions are specified at different values of the independent variable.

13.1.2Boundary-Value Problems in Engineering and Science

At the beginning of this chapter, we showed how the determination of the position and velocity of a falling object could be formulated as a boundary-value problem. For that example, a pair of ODEs was integrated in time. Although other time-variable examples can be developed, boundary-value problems arise more naturally when integrating in space. This occurs because auxiliary conditions are often specified at different positions in space.

A case in point is the simulation of the steady-state temperature distribution for a long, thin rod positioned between two constant-temperature walls (Fig. 24.2). The rod's crosssectional dimensions are small enough so that radial temperature gradients are minimal and, consequently, temperature is a function exclusively of the axial coordinate x. Heat is transferred along the rod's longitudinal axis by conduction and between the rod and the surrounding gas by convection. For this example, radiation is assumed to be negligible. [1]

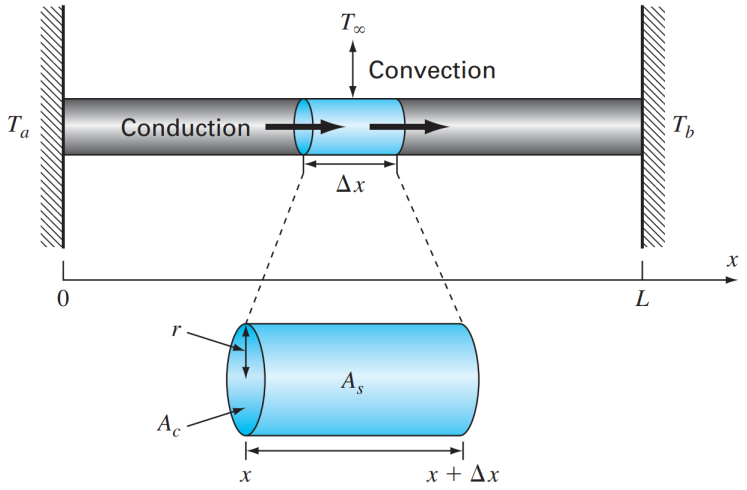


Figure 13.2: A heat balance for a differential element of a heated rod subject to conduction and convection.

[1] We incorporate radiation into this problem later in this chapter in Example 24.4.

As depicted in Fig. 24.2, a heat balance can be taken around a differential element of thickness Δx as

$$0 = q(x)A_c - q(x+\Delta x)A_c + hA_s(T_\infty - T) \tag{24.3}$$

where $q(x)$ = flux into the element due to conduction $\left[\mathrm{J}/\left(\mathrm{m}^2\cdot\mathrm{s}\right)\right]$; $q(x+\Delta x)$ = flux out of the element due to conduction $\left[\mathrm{J}/\left(\mathrm{m}^2\cdot\mathrm{s}\right)\right]$; A_c = cross-sectional area $\left[\mathrm{m}^2\right] = \pi r^2$, r = the radius [m]; h = the convection heat transfer coefficient $\left[\mathrm{J}/\left(\mathrm{m}^2\cdot\mathrm{K}\cdot\mathrm{s}\right)\right]$; A_s = the element's surface area $\left[\mathrm{m}^2\right] = 2\pi r\Delta x$; T_∞ = the temperature of the surrounding gas [K]; and T = the rod's temperature [K].

Equation (24.3) can be divided by the element's volume $\left(\pi r^2\Delta x\right)$ to yield

$$0 = \frac{q(x) - q(x+\Delta x)}{\Delta x} + \frac{2h}{r}(T_\infty - T)$$

Taking the limit $\Delta x \to 0$ gives

$$0 = -\frac{dq}{dx} + \frac{2h}{r}(T_\infty - T) \tag{24.4}$$

The flux can be related to the temperature gradient by Fourier's law:

$$q = -k\frac{dT}{dx} \tag{24.5}$$

where k = the coefficient of thermal conductivity $[\mathrm{J}/(\mathrm{s}\cdot\mathrm{m}\cdot\mathrm{K})]$. Equation (24.5) can be differentiated with respect to x, substituted into Eq. (24.4), and the result divided by k to yield,

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T) \tag{24.6}$$

where h' = a bulk heat-transfer parameter reflecting the relative impacts of convection and conduction $\left[\mathrm{m}^{-2}\right] = 2h/(rk)$.

Equation (24.6) represents a mathematical model that can be used to compute the temperature along the rod's axial dimension. Because it is a second-order ODE, two conditions are required to obtain a solution. As depicted in Fig. 24.2, a common case is where the temperatures at the ends of the rod are held at fixed values. These can be expressed mathematically as

$$T(0) = T_a$$
$$T(L) = T_b$$

The fact that they physically represent the conditions at the rod's "boundaries" is the origin of the terminology: boundary conditions.

Given these conditions, the model represented by Eq. (24.6) can be solved. Because this particular ODE is linear, an analytical solution is possible as illustrated in the following example.

Example 13. 28. Analytical Solution for a Heated Rod

Problem statement Use calculus to solve Eq. (24.6) for a $10-\mathrm{m}$ rod with $h' = 0.005m^{-2}[h = 1J/(m^2\cdot K\cdot s)]$, $r = 0.2\ m$, $k = 200J/(s\cdot m\cdot K)]$, $T_\infty = 200K$ and the boundary conditions:

$$T(0) = 300\ \mathrm{K} \quad T(10) = 400\ \mathrm{K}$$

Solution. This ODE can be solved in a number of ways. A straightforward approach is to first express the equation as

$$\frac{d^2T}{dx^2} - h'T = -h'T_\infty$$

Because this is a linear ODE with constant coefficients, the general solution can be readily obtained by setting the right-hand side to zero and assuming a solution of the form $T = e^{\lambda x}$. Substituting this solution along with its second derivative into the homogeneous form of the ODE yields

$$\lambda^2 e^{\lambda x} - h'e^{\lambda x} = 0$$

which can be solved for $\lambda = \pm\sqrt{h'}$. Thus, the general solution is

$$T = Ae^{\lambda x} + Be^{-\lambda x}$$

where A and B are constants of integration. Using the method of undetermined coefficients we can derive the particular solution $T = T_\infty$. Therefore, the total solution is

$$T = T_\infty + Ae^{\lambda x} + Be^{-\lambda x}$$

The constants can be evaluated by applying the boundary conditions

$$T_a = T_\infty + A + B$$
$$T_b = T_\infty + Ae^{\lambda L} + Be^{-\lambda L}$$

These two equations can be solved simultaneously for

$$A = \frac{(T_a - T_\infty)\, e^{-\lambda L} - (T_b - T_\infty)}{e^{-\lambda L} - e^{\lambda L}}$$
$$B = \frac{(T_b - T_\infty) - (T_a - T_\infty)\, e^{\lambda L}}{e^{-\lambda L} - e^{\lambda L}}$$

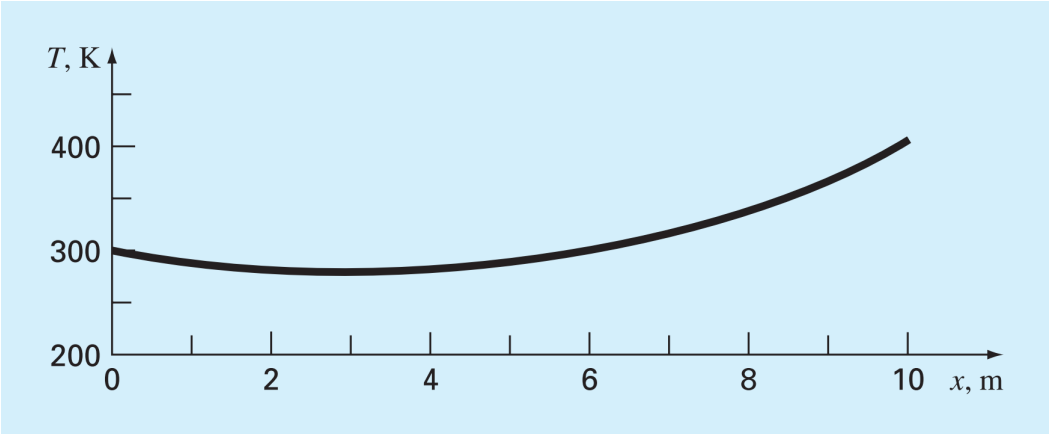


Figure 13.3: Analytical solution for the heated rod.

Substituting the parameter values from this problem gives $A = 20.4671$ and $B = 79.5329$. Therefore, the final solution is

$$T = 200 + 20.4671e^{\sqrt{0.05}x} + 79.5329e^{-\sqrt{0.05}x} \tag{24.7}$$

As can be seen in Fig. 24.3, the solution is a smooth curve connecting the two boundary temperatures. The temperature in the middle is depressed due to the convective heat loss to the cooler surrounding gas.

In the following sections, we will illustrate numerical approaches for solving the same problem we just solved analytically in Example 24.1. The exact analytical solution will be useful in assessing the accuracy of the solutions obtained with the approximate, numerical methods.

13.2. THE SHOOTING METHOD

The shooting method is based on converting the boundary-value problem into an equivalent initial-value problem. A trial-and-error approach is then implemented to develop a solution for the initial-value version that satisfies the given boundary conditions.

Although the method can be employed for higher-order and nonlinear equations, it is nicely illustrated for a second-order, linear ODE such as the heated rod described in the previous section:

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T) \tag{24.8}$$

subject to boundary conditions:

$$T(0) = T_a$$
$$T(L) = T_b$$

We convert this boundary-value problem into an initial-value problem by defining the rate of change of temperature, or *gradient*, as

$$\frac{dT}{dx} = z \tag{24.9}$$

and reexpressing Eq. (24.8) as

$$\frac{dz}{dx} = -h'(T_\infty - T) \tag{24.10}$$

Thus, we have converted the single second-order equation (Eq. 24.8) into a pair of firstorder ODEs (Eqs. 24.9 and 24.10).

If we had initial conditions for both T and z, we could solve these equations as an initialvalue problem with the methods described in Chaps. 22 and 23 . However, because we only have an initial value for one of the variables $T(0) = T_a$ we simply make a guess for the other $z(0) = z_{a1}$ and then perform the integration.

After performing the integration, we will have generated a value of T at the end of the interval, which we will call T_{b1}. Unless we are incredibly lucky, this result will differ from the desired result T_b.

Now, let's say that the value of T_{b1} is too high $(T_{b1} > T_b)$, it would make sense that a lower value of the initial slope $z(0) = z_{a2}$ might result in a better prediction. Using this new guess, we can integrate again to generate a second result at the end of the interval T_{b2}. We could then continue guessing in a trial-and-error fashion until we arrived at a guess for $z(0)$ that resulted in the correct value of $T(L) = T_b$.

At this point, the origin of the name shooting method should be pretty clear. Just as you would adjust the angle of a cannon in order to hit a target, we are adjusting the trajectory of our solution by guessing values of $z(0)$ until we hit our target $T(L) = T_b$.

Although we could certainly keep guessing, a more efficient strategy is possible for linear ODEs. In such cases, the trajectory of the perfect shot z_a is linearly related to the results of our two erroneous shots (z_{a1}, T_{b1}) and (z_{a2}, T_{b2}). Consequently, linear interpolation can be employed to arrive at the required trajectory:

$$z_a = z_{a1} + \frac{z_{a2} - z_{a1}}{T_{b2} - T_{b1}}(T_b - T_{b1}) \tag{24.11}$$

The approach can be illustrated by an example.

Example 13. 29. The Shooting Mehtod for a Linear ODE

Problem Statement. Use the shooting method to solve Eq. (24.6) for the same coditions as Example 24.1: $L = 10m$, $h' = 0.005m^{-2}$, $T_\infty = 200K$, $T(0) = 300K$, and $T(10) = 400K$.

Solution. Equation (24.6) is first expressed as a pair of first-order ODEs:

$$\frac{dT}{dx} = z$$
$$\frac{dz}{dx} = -0.05(200 - T)$$

Along with the initial value for temperature $T(0) = 300$ K, we arbitrarily guess a value of $z_{a1} = -5$ K/m for the initial value for $z(0)$. The solution is then obtained by integrating the pair of ODEs from $x = 0$ to 10. We can do this with MATLAB's `ode45` function by first setting up an M-file to hold the differential equations:

```
function dy=Ex2402(x,y)
dy=[y(2);-0.05*(200-y(1))];
```

We can then generate the solution as:

```
» [t,y]=ode45(@Ex2402,[0 10],[300,-5]);
» Tb1=y(length(y))
Tb1 =
 569.7539
```

Thus, we obtain a value at the end of the interval of $T_{b1} = 569.7539$ (Fig. 24.4a), which differs from the desired boundary condition of $T_b = 400$. Therefore, we make another guess $z_{a2} = -20$ and perform the computation again. This time, the result of $T_{b2} = 259.5131$ is obtained (Fig. 24.4b).

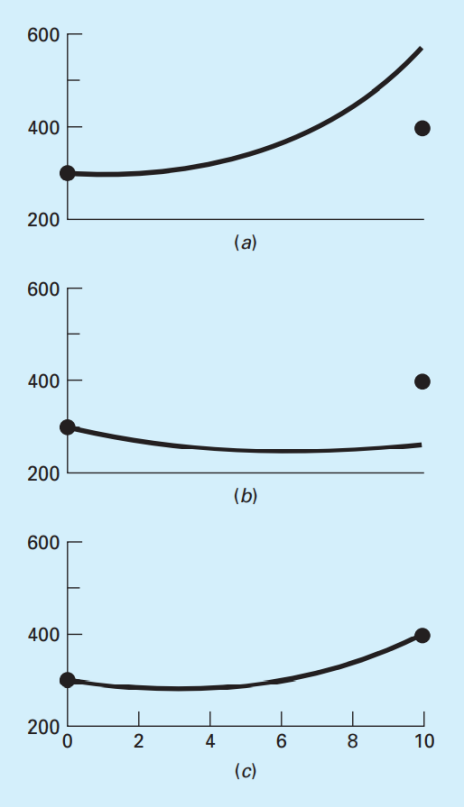


Figure 13.4: Temperature (K) versus distance (m) computed with the shooting method: (a) the first "shot," (b) the second "shot," and (c) the final exact "hit."

Now, because the original ODE is linear, we can use Eq. (24.11) to determine the correct trajectory to yield the perfect shot:

$$z_a = -5 + \frac{-20-(-5)}{259.5131-569.7539}(400-569.7539) = -13.2075$$

This value can then be used in conjunction with `ode45` to generate the correct solution, asdepicted in Fig. 24.4c.

Although it is not obvious from the graph, the analytical solution is also plotted on Fig. 24.4c. Thus, the shooting method yields a solution that is virtually indistinguishable from the exact result.

13.2.1Deriative Boundry Conditions

The fixed or Dirichlet boundary condition discussed to this point is but one of several types that are used in engineering and science. A common alternative is the case where the derivative is given. This is commonly referred to as a Neumann boundary condition. Because it is already set up to compute both the dependent variable and its derivative, incorporating derivative boundary conditions into the shooting method is relatively straightforward. Just as with the fixed-boundary condition case, we first express the second-order ODE as a pair of first-order ODEs. At this point, one of the required

initial conditions, whether the dependent variable or its derivative, will be unknown. Based on guesses for the missing initial condition, we generate solutions to compute the given end condition. As with the initial condition, this end condition can either be for the dependent variable or its derivative. For linear ODEs, interpolation can then be used to determine the value of the missing initial condition required to generate the final, perfect "shot" that hits the end condition.

Example 13. 30. The Shooting Method with Derivative Boundary Conditions

Problem statement. Use the shooting method to solve Eq. (24.6) for the rod in Example 24.1: $L = 10$ m, $h' = 0.05$ m^{-2} $\left[h = 1 \text{ J}/\left(\text{m}^2 \cdot \text{K} \cdot \text{s}\right)\right.$, $r = 0.2$ m, $k = 200 \text{ J}/\ (\text{s} \cdot \text{m} \cdot \text{K})]$, $T_\infty = 200$ K, and $T(10) = 400$ K. However, for this case, rather than having a fixed temperature of 300 K, the left end is subject to convection as in Fig. 24.5. For simplicity, we will assume that the convection heat transfer coefficient for the end area is the same as for the rod's surface.

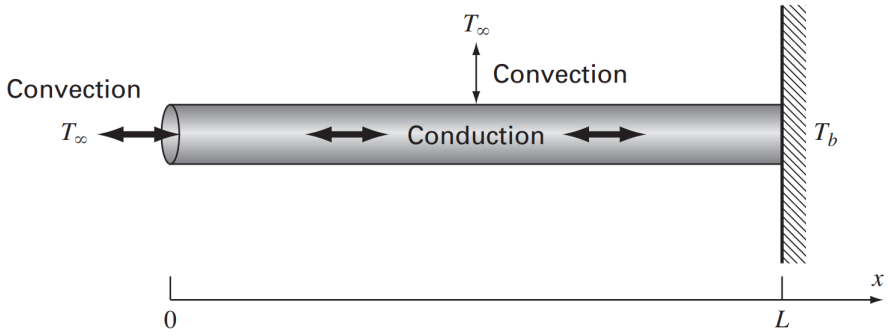


Figure 13.5: A rod with a convective boundary condition at one end and a fixed temperature at the other.

Solution. As in Example 24.2, Eq. (24.6) is first expressed as

$$\frac{dT}{dx} = z$$
$$\frac{dz}{dx} = -0.05(200 - T)$$

Although it might not be obvious, convection through the end is equivalent to specifying a gradient boundary condition. In order to see this, we must recognize that because the system is at steady state, convection must equal conduction at the rod's left boundary $(x = 0)$. Using Fourier's law (Eq. 24.5) to represent conduction, the heat balance at the end can be formulated as

$$hA_c\left(T_\infty - T(0)\right) = -kA_c\frac{dT}{dx}(0) \tag{24.12}$$

This equation can be solved for the gradient

$$\frac{dT}{dx}(0) = \frac{h}{k}\left(T(0) - T_\infty\right) \tag{24.13}$$

If we guess a value for temperature, we can see that this equation specifies the gradient.

Theshooting method is implemented by arbitrarily guessing a value for $T(0)$. If we choose a value of $T(0) = T_{a1} = 300$ K, Eq. (24.13) then yields the initial value for the gradient

$$z_{a1} = \frac{dT}{dx}(0) = \frac{1}{200}(300 - 200) = 0.5$$

The solution is obtained by integrating the pair of ODEs from $x = 0$ to 10 . We can do this with MATLAB's `ode45` function by first setting up an M-file to hold the differential equations in the same fashion as in Example 24.2. We can then generate the solution as

```
» [t,y]=ode45(@Ex2402,[0 10],[300,0.5]);
» Tb1=y(length(y))
Tb1 =
 683.5088
```

As expected, the value at the end of the interval of $T_{b1} = 683.5088K$ differs from the desired boundary condition of $T_b = 400$. Therefore, we make another guess $T_{a2} = 150K$, which corresponds to $z_{a2} = -0.25$, and perform the computation again.

```
» [t,y]=ode45(@Ex2402,[0 10],[150,-0.25]);
» Tb2=y(length(y))
Tb2 =
 -41.7544
```

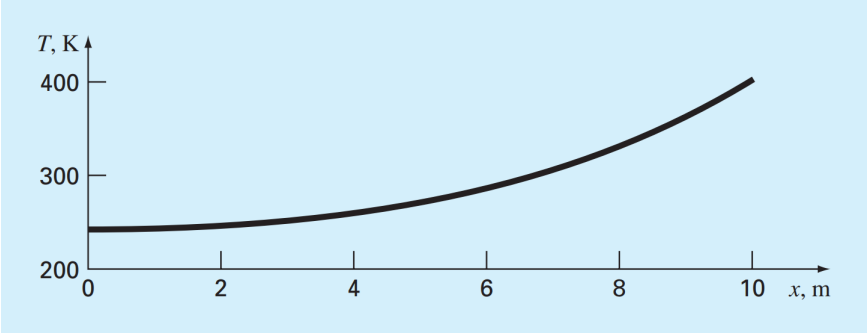


Figure 13.6: The solution of a second-order ODE with a convective boundary condition at one end and a fixed temperature at the other.

Linear interpolation can then be employed to compute the correct initial temperature:

$$T_a = 300 + \frac{150 - 300}{-41.7544 - 683.5088}(400 - 683.5088) = 241.3643 \text{ K}$$

which corresponds to a gradient of $z_a = 0.2068$. Using these initial conditions, `ode45` can be employed to generate the correct solution, as depicted in Fig. 24.6.

Note that we can verify that our boundary condition has been satisfied by substituting the initial conditions into Eq. (24.12) to give

$$1\frac{\text{J}}{\text{m}^2\ \text{K s}}\pi \times (0.2\ \text{m})^2 \times (200\ \text{K} - 241.3643\ \text{K}) = -200\frac{\text{J}}{\text{mKs}}\pi \times (0.2\ \text{m})^2 \times 0.2068\frac{\text{K}}{\text{m}}$$

which can be evaluated to yield $-5.1980\ \text{J/s} = -5.1980\ \text{J/s}$. Thus, conduction and convection are equal and transfer heat out of the left end of the rod at a rate of 5.1980 W.

13.2.2The Shooting Method for Nonlinear ODEs

For nonlinear boundary-value problems, linear interpolation or extrapolation through two solution points will not necessarily result in an accurate estimate of the required boundary condition to attain an exact solution. An alternative is to perform three applications of the shooting method and use a quadratic interpolating polynomial to estimate the proper boundary condition. However, it is unlikely that such an approach would yield the exact answer, and additional iterations would be necessary to home in on the solution.

Another approach for a nonlinear problem involves recasting it as a roots problem. Recall that the general goal of a roots problem is to find the value of x that makes the function $f(x) = 0$. Now, let us use the heated rod problem to understand how the shooting method can be recast in this form.

First, recognize that the solution of the pair of differential equations is also a "function" in the sense that we guess a condition at the left-hand end of the rod z_a, and the integration yields a prediction of the temperature at the right-hand end T_b. Thus, we can think of the integration as

$$T_b = f(z_a)$$

That is, it represents a process whereby a guess of z_a yields a prediction of T_b. Viewed in this way, we can see that what we desire is the value of z_a that yields a specific value of T_b. If, as in the example, we desire $T_b = 400$, the problem can be posed as

$$400 = f(z_a)$$

By bringing the goal of 400 over to the right-hand side of the equation, we generate a new function res (z_a) that represents the difference, or *residual*, between what we have, $f(z_a)$, and what we want, 400.

$$\text{res}(z_a) = f(z_a) - 400$$

If we drive this new function to zero, we will obtain the solution. The next example illustrates the approach.

Example 13. 31. The Shooting Method for Nonlinear ODEs

Problem statement. Although it served our purposes for illustrating the shooting method, Eq. (24.6) was not a completely realistic model for a heated rod. For one thing, such a rod would lose heat by mechanisms such as radiation that are nonlinear.

Suppose that the following nonlinear ODE is used to simulate the temperature of the heated rod:

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T) + \sigma'\left(T_\infty^4 - T^4\right)$$

where σ' = a bulk heat-transfer parameter reflecting the relative impacts of radiation and conduction $= 2.7 \times 10^{-9}$ K^{-3} m^{-2}. This equation can serve to illustrate how the shooting method is used to solve a two-point nonlinear boundary-value problem. The remaining problem conditions are as specified in Example 24.2: $L = 10$ m, $h' = 0.05$ m^{-2}, $T_\infty = 200$ K, $T(0) = 300$ K, and $T(10) = 400$ K.

Solution. Just as with the linear ODE, the nonlinear second-order equation is first expressed as two first-order ODEs:

$$\frac{dT}{dx} = z$$

$$\frac{dz}{dx} = -0.05(200 - T) - 2.7 \times 10^{-9}\left(1.6 \times 10^9 - T^4\right)$$

An M-file can be developed to compute the right-hand sides of these equations:

```
function dy=dydxn(x,y)
dy=[y(2);-0.05*(200-y(1))-2.7e-9*(1.6e9-y(1)^4)];
```

Next, we can build a function to hold the residual that we will try to drive to zero as

```
function r=res(za)
[x,y]=ode45(@dydxn,[0 10],[300 za]);
r=y(length(x),1)-400;
```

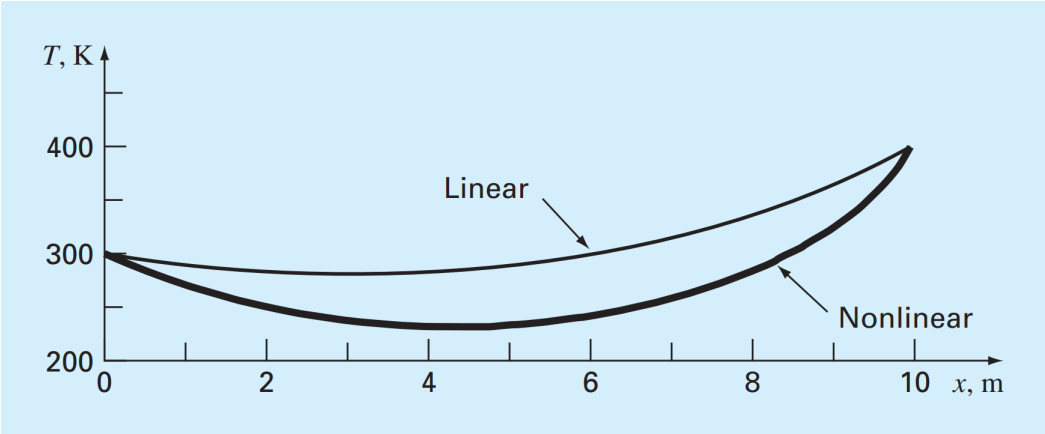


Figure 13.7: The result of using the shooting method to solve a nonlinear problem.

Notice how we use the `ode45` function to solve the two ODEs to generate the temperature at the rod;s end: `y(length(x),1)`. We can then find the root with the `fzero` function:

```
» fzero(@res,-50)

ans =
  -41.7434
```

Thus, we see that if we set the initial trajectory $z(0)$ = -41.7434, the residual function will be driven to zero and the temperature boundary condition $T(10) = 400$ at the end of the rod should be satisfied. This can be verified by generating the entire solution and plotting the temperatures versus x:

```
» [x,y]=ode45(@dydxn,[0 10],[300 fzero(@res,-50)]);
» plot(x,y(:,1))
```

The result is shown in Fig. 24.7 along with the original linear case from Example 24.2. As expected, the nonlinear case is depressed lower than the linear model due to the additional heat lost to the surrounding gas by radiation.

13.3. FINITE-DIFFERENCE METHODS

The most common alternatives to the shooting method are finite-difference approaches. In these techniques, finite differences (Chap. 21) are substituted for the derivatives in the original equation. Thus, a linear differential equation is transformed into a set of simultaneous algebraic equations that can be solved using the methods from Part Three.

We can illustrate the approach for the heated rod model (Eq. 24.6):

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T) \tag{24.14}$$

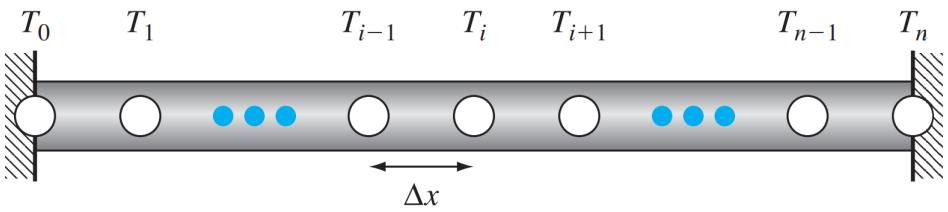


Figure 13.8: In order to implement the finite-difference approach, the heated rod is divided into a series of nodes.

The solution domain is first divided into a series of nodes (Fig. 24.8). At each node, finite-difference approximations can be written for the derivatives in the equation. For example, at node i, the second derivative can be represented by (Fig. 21.5):

$$\frac{d^2T}{dx^2} = \frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} \tag{24.15}$$

This approximation can be substituted into Eq. (24.14) to give

$$\frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} + h'(T_\infty - T_i) = 0$$

Thus, the differential equation has been converted into an algebraic equation. Collecting terms gives

$$-T_{i-1} + \left(2 + h'\Delta x^2\right) T_i - T_{i+1} = h'\Delta x^2 T_\infty \tag{24.16}$$

This equation can be written for each of the $n-1$ interior nodes of the rod. The first and last nodes T_0 and T_n, respectively, are specified by the boundary conditions. Therefore, the problem reduces to solving $n-1$ simultaneous linear algebraic equations for the $n-1$ unknowns.

Before providing an example, we should mention two nice features of Eq. (24.16). First, observe that since the nodes are numbered consecutively, and since each equation consists of a node (i) and its adjoining neighbors $(i-1$ and $i+1)$, the resulting set of linear algebraic equations will be tridiagonal. As such, they can be solved with the efficient algorithms that are available for such systems (recall Sec. 9.4).

Further, inspection of the coefficients on the left-hand side of Eq. (24.16) indicates that the system of linear equations will also be diagonally dominant. Hence, convergent solutions can also be generated with iterative techniques like the Gauss-Seidel method (Sec. 12.1).

Example 13. 32. Finite-Difference Approximation of Boundary-Value Problems

Problem statement. Use the finite-difference approach to solve the same problem as in Examples 24.1 and 24.2. Use four interior nodes with a segment length of $\Delta x = 2$ m.

Solution. Employing the parameters in Example 24.1 and $\Delta x = 2$ m, we can write Eq. (24.16) for each of the rod's interior nodes. For example, for node 1:

$$-T_0 + 2.2T_1 - T_2 = 40$$

Substituting the boundary condition $T_0 = 300$ gives

$$2.2T_1 - T_2 = 340$$

After writing Eq. (24.16) for the other interior nodes, the equations can be assembled in matrix form as

$$\begin{bmatrix} 2.2 & -1 & 0 & 0 \\ -1 & 2.2 & -1 & 0 \\ 0 & -1 & 2.2 & -1 \\ 0 & 0 & -1 & 2.2 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 340 \\ 40 \\ 40 \\ 440 \end{Bmatrix}$$

Notice that the matrix is both tridiagonal and diagonally dominant.

MATLAB can be used to generate the solution:

```
» A=[2.2 -1 0 0;
-1 2.2 -1 0;
0 -1 2.2 -1;
0 0 -1 2.2];
» b=[340 40 40 440]';
» T=A\b
T =
  283.2660
  283.1853
  299.7416
  336.2462
```

Table 24.1 provides a comparison between the analytical solution (Eq. 24.7) and the numerical solutions obtained with the shooting method (Example 24.2) and the finite-difference method (Example 24.5). Note that although there are some discrepancies, the numerical approaches agree reasonably well with the analytical solution. Further, the biggest discrepancy occurs for the finite-difference method due to the coarse node spacing we used in Example 24.5. Better agreement would occur if a finer nodal spacing had been used.

x	Analytical Solution	Shooting Method	Finite Difference
0	300	300	300
2	282.8634	282.8889	283.2660
4	282.5775	282.6158	283.1853
6	299.0843	299.1254	299.7416
8	335.7404	335.7718	336.2462
10	400	400	400

Figure 13.9: In order to implement the finite-difference approach, the heated rod is divided into a series of nodes.

13.3.1 Derivative Boundary Conditions

As mentioned in our discussion of the shooting method, the fixed or *Dirichlet boundary condition* is but one of several types that are used in engineering and science. A common alternative, called the *Neumann boundary condition*, is the case where the derivative is given.

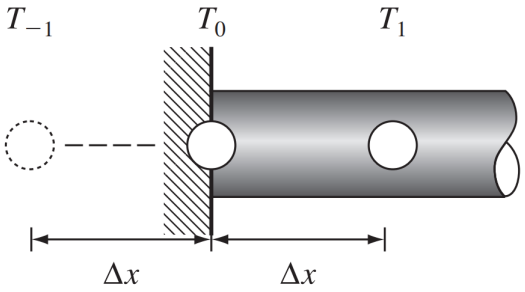


Figure 13.10: A boundary node at the left end of a heated rod. To approximate the derivative at the boundary, an imaginary node is located a distance Δx to the left of the rod's end.

We can use the heated rod introduced earlier in this chapter to demonstrate how a derivative boundary condition can be incorporated into the finite-difference approach:

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T)$$

However, in contrast to our previous discussions, we will prescribe a derivative boundary condition at one end of the rod:

$$\frac{dT}{dx}(0) = T'_a$$
$$T(L) = T_b$$

Thus, we have a derivative boundary condition at one end of the solution domain and a fixed boundary condition at the other.

Just as in the previous section, the rod is divided into a series of nodes and a finitedifference version of the differential equation (Eq. 24.16) is applied to each interior node. However, because its temperature is not specified, the node at the left end must also be included. Fig. 24.9 depicts the node (0) at the left edge of a heated plate for which the derivative boundary condition applies. Writing Eq. (24.16) for this node gives

$$-T_{-1} + \left(2 + h'\Delta x^2\right) T_0 - T_1 = h'\Delta x^2 T_\infty \tag{24.17}$$

Notice that an imaginary node (-1) lying to the left of the rod's end is required for this equation. Although this exterior point might seem to represent a difficulty, it actually serves as the vehicle for incorporating the derivative boundary condition into the problem. This is done by representing the first derivative in the x dimension at (0) by the centered difference (Eq. 4.25):

$$\frac{dT}{dx} = \frac{T_1 - T_{-1}}{2\Delta x}$$

which can be solved for

$$T_{-1} = T_1 - 2\Delta x \frac{dT}{dx}$$

Now we have a formula for T_{-1} that actually reflects the impact of the derivative. It can be substituted into Eq. (24.17) to give

$$\left(2 + h'\Delta x^2\right) T_0 - 2T_1 = h'\Delta x^2 T_\infty - 2\Delta x \frac{dT}{dx} \tag{24.18}$$

Consequently, we have incorporated the derivative into the balance.

A common example of a derivative boundary condition is the situation where the end of the rod is insulated. In this case, the derivative is set to zero. This conclusion follows directly from Fourier's law (Eq. 24.5), because insulating a boundary means that the heat flux (and consequently the gradient) must be zero. The following example illustrates how the solution is affected by such boundary conditions.

Example 13. 33. Incorporating Derivative Boundary Conditions

Problem Statement. Generate the finite-difference solution for a $10-$ m rod with $\Delta x = 2$ m, $h' = 0.05$ m^{-2}, $T_\infty = 200$ K, and the boundary conditions: $T'_a = 0$ and $T_b = 400$ K. Note that the first condition means that the slope of the solution should approach zero at the rod's left end. Aside from this case, also generate the solution for $dT/dx = -20$ at $x = 0$.

Solution. Equation (24.18) can be used to represent node 0 as

$$2.2T_0 - 2T_1 = 40$$

We can write Eq. (24.16) for the interior nodes. For example, for node 1 ,

$$-T_0 + 2.2T_1 - T_2 = 40$$

A similar approach can be used for the remaining interior nodes. The final system of equations can be assembled in matrix form as

$$\begin{bmatrix} 2.2 & -2 & & & \\ -1 & 2.2 & -1 & & \\ & -1 & 2.2 & -1 & \\ & & -1 & 2.2 & -1 \\ & & & -1 & 2.2 \end{bmatrix} \begin{Bmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 40 \\ 40 \\ 40 \\ 40 \\ 440 \end{Bmatrix}$$

These equations can be solved for

$$T_0 = 243.0278$$
$$T_1 = 247.3306$$
$$T_2 = 261.0994$$
$$T_3 = 287.0882$$
$$T_4 = 330.4946$$

As displayed in Fig. 24.10, the solution is flat at $x = 0$ due to the zero derivative condition and then curves upward to the fixed condition of $T = 400$ at $x = 10$.

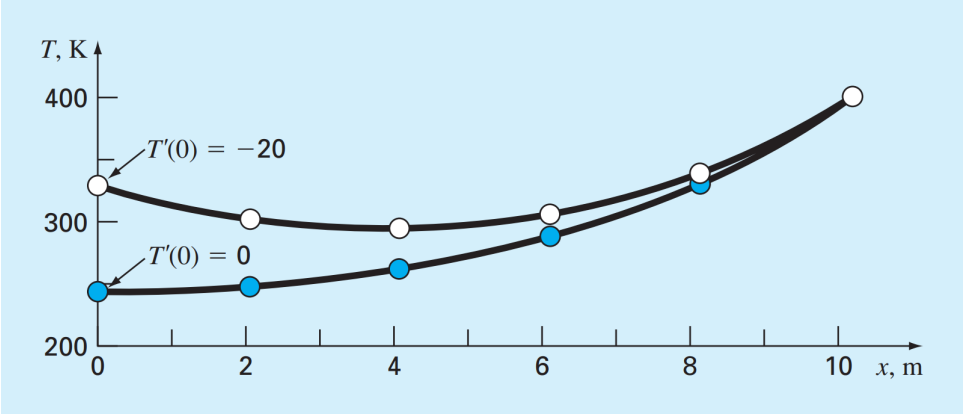


Figure 13.11: The solution of a second-order ODE with a derivative boundary condition at one end and a fixed boundary condition at the other. Two cases are shown reflecting different derivative values at x = 0.

For the case where the derivative at $x = 0$ is set to -20, the simultaneous equations are

$$\begin{bmatrix} 2.2 & -2 & & & \\ -1 & 2.2 & -1 & & \\ & -1 & 2.2 & -1 & \\ & & -1 & 2.2 & -1 \\ & & & -1 & 2.2 \end{bmatrix} \begin{Bmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 120 \\ 40 \\ 40 \\ 40 \\ 440 \end{Bmatrix}$$

which can be solved for

$$T_0 = 328.2710$$
$$T_1 = 301.0981$$
$$T_2 = 294.1448$$
$$T_3 = 306.0204$$
$$T_4 = 339.1002$$

As in Fig. 24.10, the solution at $x = 0$ now curves downward due to the negative derivative we imposed at the boundary.

13.3.2Finite-Difference Approaches for Nonlinear ODEs

For nonlinear ODEs, the substitution of finite differences yields a system of nonlinear simultaneous equations. Thus, the most general approach to solving such problems is to use root-location methods for systems of equations such as the Newton-Raphson method described in Section 12.2.2. Although this approach is certainly feasible, an adaptation of successive substitution can sometimes provide a simpler alternative.

The heated rod with convection and radiation introduced in Example 24.4 provides a nice vehicle for demonstrating this approach,

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T) + \sigma'\left(T_\infty^4 - T^4\right)$$

We can convert this differential equation into algebraic form by writing it for a node i and substituting Eq. (24.15) for the second derivative:

$$0 = \frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} + h'(T_\infty - T_i) + \sigma'\left(T_\infty^4 - T_i^4\right)$$

Collecting terms gives

$$-T_{i-1} + \left(2 + h'\Delta x^2\right)T_i - T_{i+1} = h'\Delta x^2 T_\infty + \sigma''\Delta x^2\left(T_\infty^4 - T_i^4\right)$$

Notice that although there is a nonlinear term on the right-hand side, the left-hand side is expressed in the form of a linear algebraic system that is diagonally dominant. If we assume that the unknown nonlinear term on the right is equal to its value from the previous iteration, the equation can be solved for

$$T_i = \frac{h'\Delta x^2 T_\infty + \sigma'\Delta x^2\left(T_\infty^4 - T_i^4\right) + T_{i-1} + T_{i+1}}{2 + h'\Delta x^2} \tag{24.19}$$

As in the Gauss-Seidel method, we can use Eq. (24.19) to successively calculate the temperature of each node and iterate until the process converges to an acceptable tolerance. Although this approach will not work for all cases, it converges for many ODEs derived from physically based systems. Hence, it can sometimes prove useful for solving problems routinely encountered in engineering and science.

Example 13. 34. The Finite-Difference Method for Nonlinear ODEs *Problem Statement.* Use the finite-difference approach to simulate the temperature of a heated rod subject to both convection and radiation:

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T) + \sigma'\left(T_\infty^4 - T^4\right)$$

where $\sigma' = 2.7 \times 10^{-9}$ K^{-3} m^{-2}, $L = 10$ m, $h' = 0.05$ m^{-2}, $T_\infty = 200$ K, $T(0) = 300$ K, and $T(10) = 400$ K. Use four interior nodes with a segment length of $\Delta x = 2$ m. Recall that we solved the same problem with the shooting method in Example 24.4.

Solution. Using Eq. (24.19) we can successively solve for the temperatures of the rod's interior nodes. As with the standard Gauss-Seidel technique, the initial values of the interior nodes are zero

with the boundary nodes set at the fixed conditions of $T_0 = 300$ and $T_5 = 400$. The results for the first iteration are

$$T_1 = \frac{0.05(2)^2 200 + 2.7 \times 10^{-9'}(2)^2\left(200^4 - 0^4\right) + 300 + 0}{2 + 0.05(2)^2} = 159.2432$$

$$T_2 = \frac{0.05(2)^2 200 + 2.7 \times 10^{-9'}(2)^2\left(200^4 - 0^4\right) + 159.2432 + 0}{2 + 0.05(2)^2} = 97.9674$$

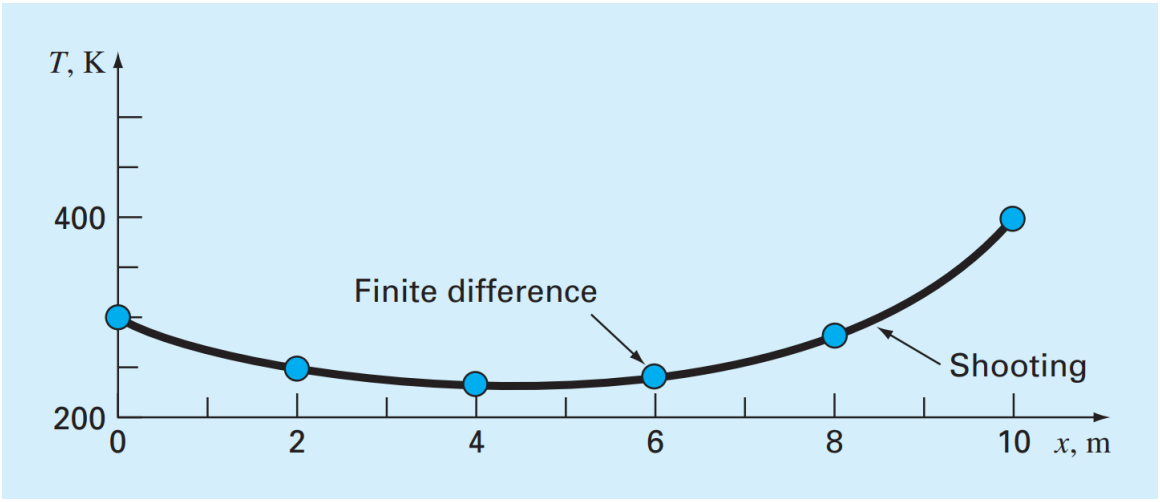


Figure 13.12: The filled circles are the result of using the finite-difference method to solve a nonlinear problem. The line generated with the shooting method in Example 24.4 is shown for comparison.

$$T_3 = \frac{0.05(2)^2 200 + 2.7 \times 10^{-9'}(2)^2\left(200^4 - 0^4\right) + 97.9674 + 0}{2 + 0.05(2)^2} = 70.4461$$

$$T_4 = \frac{0.05(2)^2 200 + 2.7 \times 10^{-9'}(2)^2\left(200^4 - 0^4\right) + 70.4461 + 400}{2 + 0.05(2)^2} = 226.8704$$

The process can be continued until we converge on the final result:

$$T_0 = 300$$
$$T_1 = 250.4827$$
$$T_2 = 236.2962$$
$$T_3 = 245.7596$$
$$T_4 = 286.4921$$
$$T_5 = 400$$

These results are displayed in Fig. 24.11 along with the result generated in Example 24.4 with the shooting method.

13.4. PROBLEMS

24.1 A steady-state heat balance for a rod can be represented as

$$\frac{d^2}{dx^2} - 0.15T = 0$$

Obtain a solution for a 10-m rod with T (0) = 240 and T(10) = 150 **(a)** analytically, **(b)** with the shooting method, and **(c)** using the finite-difference approach with $\Delta x = 1$.

24.2 Repeat Prob. 24.1 but with the right end insulated and the left end temperature fixed at 240.

24.3 Use the shooting method to solve

$$7\frac{d^2y}{dx^2} - 2\frac{dy}{dx} - y + x = 0$$

with the boundary conditions y(0) = 5 and y(20) = 8.

24.4 Solve Prob. 24.3 with the finite-difference approach using $\Delta x = 2$.

24.5 The following nonlinear differential equation was solved in Examples 24.4 and 24.7.

$$0 = \frac{d^2T}{dx^2} + h'(T_\infty - T) + \sigma'\left(T_\infty^4 - T^4\right) \quad \text{(P24.5)}$$

Such equations are sometimes linearized to obtain an approximate solution. This is done by employing a first-order Taylor series expansion to linearize the quartic term in the equation as

$$\sigma' T^4 = \sigma' \bar{T}^4 + 4\sigma' \bar{T}^3 (T - \bar{T})$$

where $\overline{T}$ is a base temperature about which the term is linearized. Substitute this relationship into Eq. (P24.5), and then solve the resulting linear equation with the finitedifference approach. Employ $\overline{T} = 300, \Delta x = 1$ m, and the parameters from Example 24.4 to obtain your solution. Plot your results along with those obtained for the nonlinear versions in Examples 24.4 and 24.7.

24.6 Develop an M-file to implement the shooting method for a linear second-order ODE. Test the program by duplicating Example 24.2.

24.7 Develop an M-file to implement the finite-difference approach for solving a linear second-order ODE with Dirichlet boundary conditions. Test it by duplicating Example 24.5.

24.8 An insulated heated rod with a uniform heat source can be modeled with the *Poisson equation:*

$$\frac{d^2T}{dx^2} = -f(x)$$

Given a heat source $f(x) = 25°\text{C/m}^2$ and the boundary conditions $T(x = 0) = 40°\text{C}$ and $T(x = 10) = 200°\text{C}$, solve for the temperature distribution with **(a)** the shooting method and **(b)** the finite-difference method ($\Delta x = 2$).

24.9 Repeat Prob. 24.8, but for the following spatially varying heat source: $f(x) = 0.12x^3 - 2.4x^2 + 12x$.

24.10 The temperature distribution in a tapered conical cooling fin (Fig. P24.10) is described by the following differential equation, which has been nondimensionalized:

$$\frac{d^2u}{dx^2} + \left(\frac{2}{x}\right)\left(\frac{du}{dx} - pu\right) = 0$$

where $u =$ temperature $(0 \le u \le 1), x =$ axial distance $(0 \le x \le 1)$, and p is a nondimensional parameter that describes the heat transfer and geometry:

$$p = \frac{hL}{k}\sqrt{1 + \frac{4}{2m^2}}$$

where $h =$ a heat transfer coefficient, $k =$ thermal conductivity, $L =$ the length or height of the cone, and $m =$ the slope of the cone wall. The equation has the boundary conditions:

$$u(x = 0) = 0 \quad u(x = 1) = 1$$

Solve this equation for the temperature distribution using finite-difference methods. Use second-order accurate finitedifference formulas for the derivatives. Write a computer program to obtain the solution and plot temperature versus axial distance for various values of $p = 10, 20, 50,$ and 100.

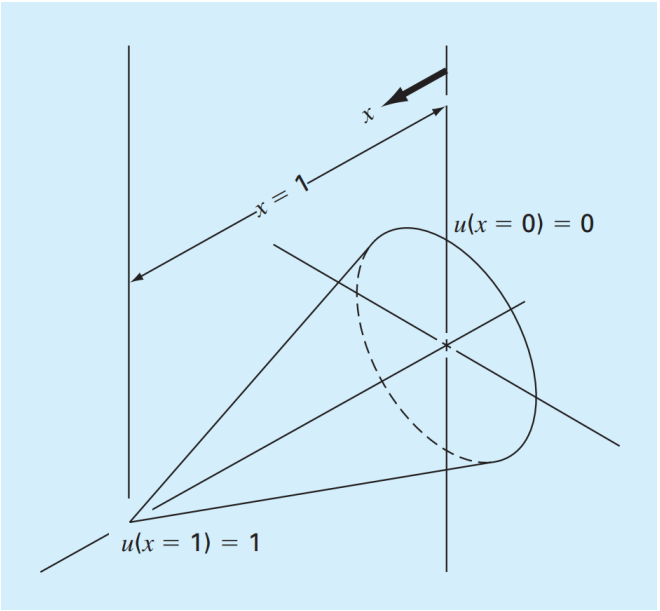


Figure 13.13

24.11 24.11 Compound A diffuses through a 4-cm-long tube and reacts as it diffuses. The equation governing diffusion with reaction is

$$D\frac{d^2A}{dx^2} - kA = 0$$

At one end of the tube $(x = 0)$, there is a large source of A that results in a fixed concentration of 0.1M. At the other end of the tube there is a material that quickly absorbs any A, making the concentration 0M. If $D = 1.5 \times 10^{-6}$ cm^2/s and $k = 5 \times 10^{-6}$ s^{-1}, what is the concentration of A as a function of distance in the tube?

24.12 The following differential equation describes the steady-state concentration of a substance that reacts with first-order kinetics in an axially dispersed plug-flow reactor (Fig. P24.12):

$$D\frac{d^2c}{dx^2} - U\frac{dc}{dx} - kc = 0$$

where D = the dispersion coefficient $\left(\text{m}^2/\text{hr}\right), c$ = concentration (mol/L), x = distance $(\text{m}), U$ = the velocity (m/hr), and k = the reaction rate (/hr). The boundary conditions can be formulated as

$$Uc_{\text{in}} = Uc(x = 0) - D\frac{dc}{dx}(x = 0)$$

$$\frac{dc}{dx}(x = L) = 0$$

where c_{in} = the concentration in the inflow $(\text{mol/L}), L$ = the length of the reactor (m). These are called Danckwerts boundary conditions.

Use the finite-difference approach to solve for concentration as a function of distance given the following parameters: $D = 5000\ \text{m}^2/\text{hr}, U = 100\ \text{m/hr}, k = 2/\text{hr}, L = 100\ \text{m}$, and $c_{\text{in}} = 100\ \text{mol/L}$. Employ centered finite-difference approximations with $\Delta x = 10$ m to obtain your solutions. Compare your numerical results with the analytical solution: $c = \frac{Uc_{\text{in}}}{(U-D\lambda_1)\lambda_2 e^{\lambda_2 L} - (U-D\lambda_2)\lambda_1 e^{\lambda_1 L}} \times \left(\lambda_2 e^{\lambda_2 L} e^{\lambda_1 x} - \lambda_1 e^{\lambda_1 L} e^{\lambda_2 x}\right)$ where

$$\frac{\lambda_1}{\lambda_2} = \frac{U}{2D}\left(1 \pm \sqrt{1 + \frac{4kD}{U^2}}\right)$$

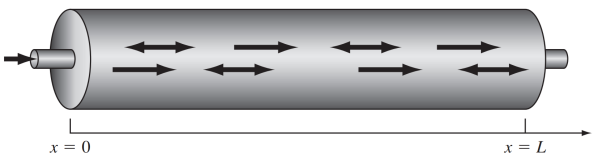


Figure 13.14

24.13 A series of first-order, liquid-phase reactions create a desirable product (B) and an undesirable byproduct (C):

$$\text{A} \xrightarrow{k_1} \text{B} \xrightarrow{k_2} \text{C}$$

If the reactions take place in an axially dispersed plug-flow reactor (Fig. P24.12), steady-state mass balances can be used to develop the following second-order ODEs:

$$D\frac{d^2c_a}{dx^2} - U\frac{dc_a}{dx} - k_1c_a = 0$$

$$D\frac{d^2c_b}{dx^2} - U\frac{dc_b}{dx} + k_1c_a - k_2c_b = 0$$

$$D\frac{d^2c_c}{dx^2} - U\frac{dc_c}{dx} + k_2c_b = 0$$

Use the finite-difference approach to solve for the concentration of each reactant as a function of distance given: $D = 0.1\ \text{m}^2/\text{min}$, $U = 1\ \text{m/min}$, $k_1 = 3/\text{min}$, $k_2 = 1/\text{min}$, $L = 0.5$ m, $c_{a,\text{in}} = 10\ \text{mol/L}$. Employ centered finite-difference approximations with $\Delta x = 0.05$ m to obtain your solutions and assume Danckwerts boundary conditions as described in Prob. 24.12. Also, compute the sum of the reactants as a function of distance. Do your results make sense?

24.14 A biofilm with a thickness $L_f(\text{ cm})$, grows on the surface of a solid (Fig. P24.14). After traversing a diffusion layer of thickness $L(\text{ cm})$, a chemical compound A diffuses into the biofilm where it is subject to an irreversible firstorder reaction that converts it to a product B.

Steady-state mass balances can be used to derive the following ordinary differential equations for compound A :

$$D\frac{d^2c_a}{dx^2} = 0 \quad 0 \le x < L$$

$$D_f\frac{d^2c_a}{dx^2} - kc_a = 0 \quad L \le x < L + L_f$$

where D = the diffusion coefficient in the diffusion layer $= 0.8\ \text{cm}^2/\text{d}, D_f$ = the diffusion coefficient in the biofilm $= 0.64\ \text{cm}^2/\text{d}$, and k = the first-order rate for the conversion of A to $B = 0.1/\text{d}$. The following boundary conditions hold: $c_a = c_{a0}$ at $x = 0$ $\frac{dc_a}{dx} = 0$ at $x = L + L_f$ where c_{a0} = the concentration of A in the bulk liquid $= 100\ \text{mol/L}$. Use the finite-difference method to compute the steady-state distribution of A from $x = 0$ to $L + L_f$, where $L = 0.008$ cm and $L_f = 0.004$ cm. Employ centered finite differences with $\Delta x = 0.001$ cm.

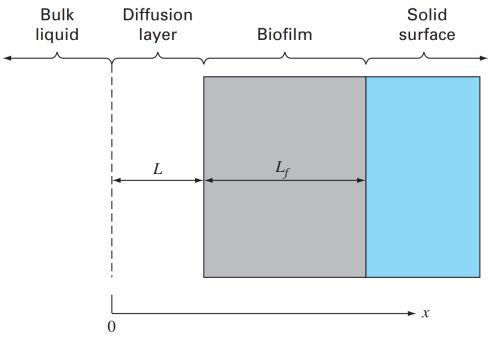


Figure 13.15: A biofilm growing on a solid surface.

24.15 A cable is hanging from two supports at A and B (Fig. P24.15). The cable is loaded with a distributed load whose magnitude varies with x as

$$w = w_o\left[1+\sin\left(\frac{\pi x}{2l_A}\right)\right]$$

where $w_o = 450$ N/m. The slope of the cable $(dy/dx) = 0$ at $x = 0$, which is the lowest point for the cable. It is also the point where the tension in the cable is a minimum of T_o. The differential equation which governs the cable is

$$\frac{d^2y}{dx^2} = \frac{w_o}{T_o}\left[1+\sin\left(\frac{\pi x}{2l_A}\right)\right]$$

Solve this equation using a numerical method and plot the shape of the cable (y versus x). For the numerical solution, the value of T_o is unknown, so the solution must use an iterative technique, similar to the shooting method, to converge on a correct value of h_A for various values of T_σ.

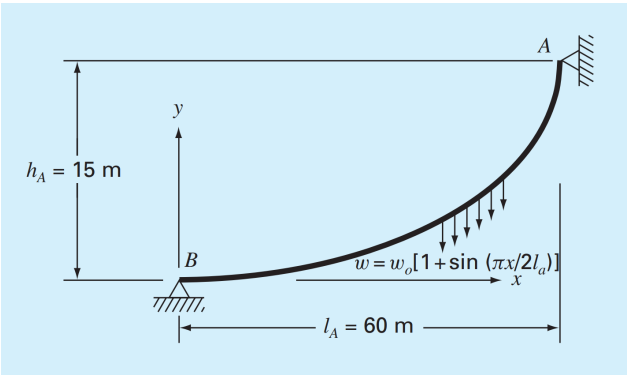


Figure 13.16

24.16 The basic differential equation of the elastic curve for a simply supported, uniformly loaded beam (Fig. P24.16) is given as

$$EI\frac{d^2y}{dx^2} = \frac{wLx}{2} - \frac{wx^2}{2}$$

where E = the modulus of elasticity, and I = the moment of inertia. The boundary conditions are $y(0) = y(L) = 0$. Solve for the deflection of the beam using (a) the finite-difference approach $(\Delta x = 0.6\text{ m})$ and (b) the shooting method. The following parameter values apply: $E = 200\text{GPa}, I = 30{,}000\text{ cm}^4, w = 15\text{kN/m}$, and $L = 3$ m. Compare your numerical results to the analytical solution:

$$y = \frac{wLx^3}{12EI} - \frac{wx^4}{24EI} - \frac{wL^3x}{24EI}$$

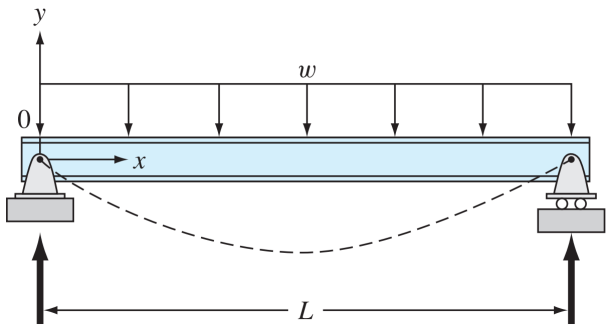


Figure 13.17

24.17 In Prob. 24.16, the basic differential equation of the elastic curve for a uniformly loaded beam was formulated as

$$EI\frac{d^2y}{dx^2} = \frac{wLx}{2} - \frac{wx^2}{2}$$

Note that the right-hand side represents the moment as a function of x. An equivalent approach can be formulated in terms of the fourth derivative of deflection as

$$EI\frac{d^4y}{dx^4} = -w$$

For this formulation, four boundary conditions are required. For the supports shown in Fig. P24.16, the conditions are that the end displacements are zero, $y(0) = y(L) = 0$, and that the end moments are zero, $y''(0) = y''(L) = 0$. Solve for the deflection of the beam using the finite-difference approach $(\Delta x = 0.6\text{ m})$. The following parameter values apply: $E = 200\text{GPa}, I = 30{,}000\text{ cm}^4, w = 15\text{kN/m}$, and $L = 3$ m. Compare your numerical results with the analytical solution given in Prob. 24.16.

24.18 Under a number of simplifying assumptions, the steady-state height of the water table in a one-dimensional, unconfined groundwater aquifer (Fig. P24.18) can be modeled with the following second-order ODE:

$$K\bar{h}\frac{d^2h}{dx^2} + N = 0$$

where x = distance (m), K = hydraulic conductivity (m/d), h = height of the water table (m), $\bar{h}$ = the average height of the water table (m), and N = infiltration rate (m/d).

Solve for the height of the water table for $x = 0$ to 1000 m where $h(0) = 10$ m and $h(1000) = 5$ m. Use the following parameters for the calculation: $K = 1$ m/d and $N = 0.0001$ m/d. Set the average height of the water table as the average of the boundary conditions. Obtain your solution with **(a)** the shooting method and **(b)** the finite-difference method ($\Delta x = 100$ m).

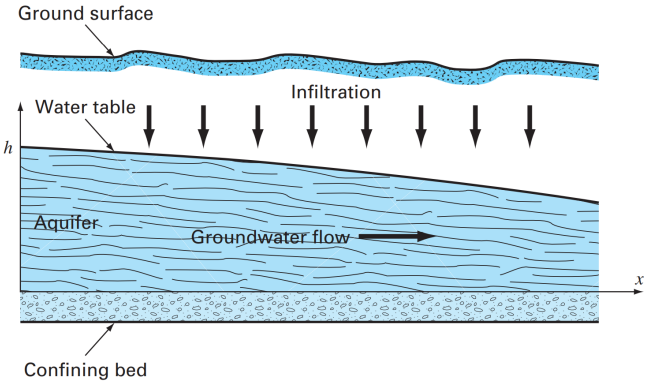


Figure 13.18: An unconfined or "phreatic" aquifer.

24.19 24.19 In Prob. 24.18, a linearized groundwater model was used to simulate the height of the water table for an unconfined aquifer. A more realistic result can be obtained by using the following nonlinear ODE:

$$\frac{d}{dx}\left(Kh\frac{dh}{dx}\right) + N = 0$$

where x = distance (m), K = hydraulic conductivity (m/d), h = height of the water table (m), and N = infiltration rate (m/d). Solve for the height of the water table for the same case as in Prob. 24.18. That is, solve from $x = 0$ to 1000 m with $h(0) = 10$ m, $h(1000) = 5$ m, $K = 1$ m/d, and $N = 0.0001$ m/d. Obtain your solution with **(a)** the shooting method and **(b)** the finite-difference method ($\Delta x = 100$ m).

24.20 Just as Fourier's law and the heat balance can be employed to characterize temperature distribution, analogous relationships are available to model field problems in other areas of engineering. For example, electrical engineers use a similar approach when modeling electrostatic fields. Under a number of simplifying assumptions, an analog of Fourier's law can be represented in one-dimensional form as

$$D = -\varepsilon\frac{dV}{dx}$$

where D is called the electric flux density vector, ε = permittivity of the material, and V = electrostatic potential. Similarly, a Poisson equation (see Prob. 24.8) for electrostatic fields can be represented in one dimension as

$$\frac{d^2V}{dx^2} = -\frac{\rho_v}{\varepsilon}$$

where ρ_v = charge density. Use the finite-difference technique with $\Delta x = 2$ to determine V for a wire where $V(0) = 1000$, $V(20) = 0$, $\varepsilon = 2$, $L = 20$, and $\rho_v = 30$.

24.21 24.21 Suppose that the position of a falling object is governed by the following differential equation:

$$\frac{d^2x}{dt^2} + \frac{c}{m}\frac{dx}{dt} - g = 0$$

where c = a first-order drag coefficient = 12.5 kg/s, m = mass = 70 kg, and g = gravitational acceleration = 9.81 m/s^2. Use the shooting method to solve this equation for the boundary conditions:

$$x(0) = 0$$
$$x(12) = 500$$

24.22 As in Fig. P24.22, an insulated metal rod has a fixed temperature (T_0) boundary condition at its left end. On it right end, it is joined to a thin-walled tube filled with water through which heat is conducted. The tube is insulated at its right end and convects heat with the surrounding fixedtemperature air (T_∞). The convective heat flux at a location x along the tube $\left(\text{W/m}^2\right)$ is represented by

$$J_{\text{conv}} = h\left(T_\infty - T_2(x)\right)$$

where h = the convection heat transfer coefficient $\left[\text{W/}\left(\text{m}^2\cdot\text{K}\right)\right]$. Employ the finite-difference method with $\Delta x = 0.1$ m to compute the temperature distribution for the case where both the rod and tube are cylindrical with the same radius r(m). Use the following parameters for your analysis: $L_{\text{rod}} = 0.6$ m, $L_{\text{tube}} = 0.8$ m, $T_0 = 400$ K, $T_\infty = 300$ K, $r = 3$ cm, $\rho_1 = 7870$ kg/m^3, $C_{p1} = 447$ J/(kg · K), $k_1 = 80.2$ W/(m · K), $\rho_2 = 1000$ kg/m^3, $C_{p2} = 4.18$ kJ/(kg · K), $k_2 = 0.615$ W/(m·K), and $h = 3000$ W/$\left(\text{m}^2\cdot\text{K}\right)$. The subscripts designate the rod (1) and the tube (2).

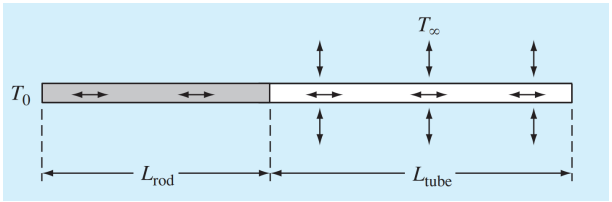


Figure 13.19

24.23 Perform the same calculation as in Prob. 24.22, butfor the case where the tube is also insulated (i.e., no convection) and the right-hand wall is held at a fixed boundary temperature of 200 K.

APPENDIX A: MATLAB BUILT-IN FUNCTIONS

APPENDIX B: MATLAB M-FILE FUNCTIONS

M-file Name	Description	Page
bisect	Root location with bisection	139
eulode	Integration of a single ordinary differential equation with Euler's method	560
f zerosimp	Brent's method for root location	167
GaussNaive	Solving linear systems with Gauss elimination without pivoting	239
GaussPivot	Solving linear systems with Gauss elimination with partial pivoting	244
GaussSeidel	Solving linear systems with the Gauss-Seidel method	289
goldmin	Minimum of one-dimensional function with golden-section search	192
incsearch	Root location with an incremental search	132
IterMeth	General algorithm for iterative calculation	94
Lagrange	Interpolation with the lagrange polynomial	419
linregr	Fitting a straight line with linear regression	350
natspline	Cubic spline with natural end conditions	453
Newtint	Interpolation with the Newton polynomial	416
newtmult	Root location for nonlinear systems of equations	297
newtraph	Root location with the Newton-Raphson method	161
quadadapt	Adaptive quadrature	512
rk4sys	Integration of system of ODEs with 4th-order RK method	576
romberg	Integration of a function with Romberg integration	503
TableLook	Table lookup with linear interpolation	434
trap	Integration of a function with the composite trapezoidal rule	474
trapuneq	Integration of unequispaced data with the trapezoidal rule	483
Tridiag	Solving tridiagonal linear systems	247

BIBLIOGRAPHY

D

E

F

G

H

O

P

U

V

W

Y

Z

BIBLIOGRAPHY

Anscombe, F. J., *"Graphs in Statistical Analysis," Am. Stat.*, 27(1):17-21, 1973.

Attaway, S., MATLAB: *A Practical Introduction to Programming and Problem Solving*, Elsevier Science, Burlington, MA, 2009.

Bogacki, P. and L. F. Shampine, "A 3(2) Pair of Runge-Kutta Formulas," *Appl. Math. Letters*, 2(1989):1-9, 1989.

Brent, R. P., *Algorithms for Minimization Without Derivatives*, Prentice Hall, Englewood Cliffs, NJ, 1973.

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*, 6th ed., McGraw-Hill, New York, 2010.

Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.*, 19:297-301, 1965.

Dekker, T. J., "Finding a Zero by Means of Successive Linear Interpolation." In B. Dejon and P. Henrici (editors), *Constructive Aspects of the Fundamental Theorem of Algebra*, Wiley-Interscience, New York, 1969, pp. 37-48.

Dormand, J. R. and P. J. Prince, "A Family of Embedded Runge-Kutta Formulae," *J. Comp. 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.

Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computation*, Prentice Hall, Englewood Cliffs, NJ, 1977.

Gabel, R. A. and R. A. Roberts, *Signals and Linear Systems*, Wiley, New York, 1987.

Gander, W. and W. Gautschi, *Adaptive Quadrature- Revisited*, BIT *Num. Math.*, 40:84-101, 2000.

Gerald, C. F. and P. O. Wheatley, *Applied Numerical Analysis*, 3d ed., Addison-Wesley, Reading, MA, 1989.

Hanselman, D. and B. Littlefield, *Mastering* MATLAB 7, Prentice Hall, Upper Saddle River, NJ, 2005.

Hayt, W. H. and J. E. Kemmerly, *Engineering Circuit Analysis*, McGraw-Hill, New York, 1986.

Heideman, M. T., D. H. Johnson, and C. S. Burrus, "Gauss and the History of the Fast Fourier Transform," *IEEE ASSP Mag.*, 1(4):14-21, 1984.

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.

Moler, C. B., *Numerical Computing with MATLAB*, SIAM, Philadelphia, 2004.

Moore, H., *MATLAB for Engineers*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2008.

Ortega, J. M., *Numerical Analysis-A Second Course*, Academic Press, New York, 1972.

Palm, W. J. III, *A Concise Introduction to MATLAB*, McGraw-Hill, New York, 2007.

Ralston, A., "Runge-Kutta Methods with Minimum Error Bounds," *Match. Comp.*, 16:431, 1962.

Ralston, A. and P. Rabinowitz, *A First Course in Numerical Analysis*, 2d ed., McGraw-Hill, New York, 1978.

Ramirez, R. W., *The FFT, Fundamentals and Concepts*, Prentice Hall, Englewood Cliffs, NJ, 1985.

Recktenwald, G., *Numerical Methods with MATLAB*, Prentice Hall, Englewood Cliffs, NJ, 2000.

Scarborough, I. 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.

Van Valkenburg, M. E., *Network Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1974.

White, F. M., *Fluid Mechanics*. McGraw-Hill, New York, 1999.